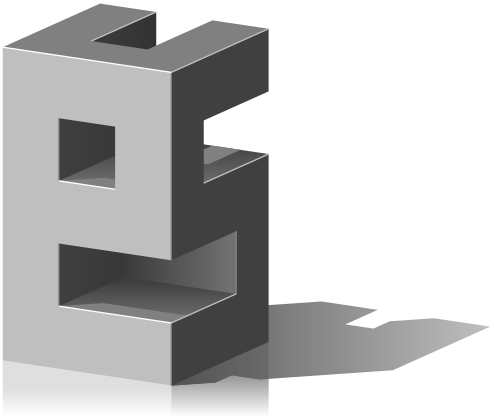
Eigen Compiler Suite

User Manual



August 10, 2026

Contents

List of Figures

List of Tables

Preface

This user manual documents all components of a free and self-hosted software development toolchain called the *Eigen Compiler Suite* and explains how to use them. It is partitioned into four parts. The first part describes the overall mission, design, and the common user interface of the Eigen Compiler Suite. The second part lists the programming languages supported by the Eigen Compiler Suite, while the third part summarizes the hardware architectures and how they are supported. The last part of this manual finally depicts some internal functionality and explains how the Eigen Compiler Suite can be extended in order to support additional programming languages, hardware architectures, and runtime environments. The addendum elaborates on the implementation of the Eigen Compiler Suite and includes its licenses.

Il semble que la perfection soit atteinte
non quand il n'y a plus rien à ajouter,
mais quand il n'y a plus rien à retrancher.

— Antoine de Saint-Exupéry

Part I

Using the Eigen Compiler Suite

1 Introduction

Eigen Compiler Suite is the name of a free software collection of development tools like compilers, assemblers, and linkers targeting different programming languages and different hardware architectures. This chapter gives a general overview over the Eigen Compiler Suite and describes its features and design in detail.

I will not follow where the path may lead,
instead I will go where there is no path
and leave a trail.

— Muriel Strode

1.1 Features

The Eigen Compiler Suite is a software development toolchain. It contains tools like compilers, pretty printers, interpreters, assemblers, and linkers targeting a variety of programming languages and hardware architectures. The Eigen Compiler Suite features pretty printers, interpreters, and compilers for the following programming languages:

- C++

  C++ is a general-purpose programming language with a bias toward systems programming. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

- FALSE

  FALSE is an esoteric stack-oriented programming language that provides lambda abstractions and is quite powerful for its size. For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6.

- Oberon

  Oberon is a general-purpose programming language that supports type extension with type-bound procedures which makes it an object-oriented language. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

Besides the support for these programming languages, the Eigen Compiler Suite also features assemblers, disassemblers, and compilers targeting the following hardware architectures:

- AMD64

  AMD64 is a 64-bit instruction set architecture developed by AMD. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9.

- ARM

  ARM is a 32-bit and 64-bit instruction set architecture developed by ARM Holdings. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10.

- AVR

  AVR is the name of an 8-bit microcontroller architecture developed by Atmel. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11.

- AVR32

  AVR32 is the name of an 32-bit microcontroller architecture developed by Atmel. For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12.

- M68000

  M68000 is a 16/32-bit instruction set architecture developed by Motorola. For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13.

- MicroBlaze

  MicroBlaze is a 32-bit instruction set architecture developed by Xilinx. For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14.

- MIPS

  MIPS32 and MIPS64 are 32-bit and 64-bit instruction set architectures developed by MIPS Technologies. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15.

- MMIX

  MMIX is a 64-bit instruction set architecture designed by Donald E. Knuth. For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16.

- OpenRISC 1000

  OpenRISC 1000 is a 32/64-bit instruction set architecture developed by OpenCores. For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17.

- PowerPC

  PowerPC is 32-bit and 64-bit instruction set architecture developed by AIM. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18.

- RISC

  RISC is a 32-bit instruction set architecture designed by Niklaus Wirth. For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19.

- WebAssembly

  WebAssembly is a 32-bit instruction set architecture designed by the World Wide Web Consortium (W3C). For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20.

- Xtensa

  Xtensa is a 32-bit instruction set architecture designed by Cadence Design Systems. For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21.

Finally, the Eigen Compiler Suite features various linkers and runtime support for the following runtime environments:

- Atari TOS

  The Eigen Compiler Suite provides runtime support and a linker for programs that can be executed under Atari TOS.

- AVR Microcontrollers

  The Eigen Compiler Suite provides runtime support for AVR microcontrollers. Additionally, it features a linker that generates Intel HEX files that can be used to program these microcontrollers.

- BIOS

  The Eigen Compiler Suite provides runtime support and a linker for bootloaders executed by the BIOS.

- Darwin

  The Eigen Compiler Suite provides runtime support and a linker for programs that can be executed under Darwin.

- DOS

  The Eigen Compiler Suite features runtime support and a linker for programs that are executed under DOS.

- EFI

  The Eigen Compiler Suite provides runtime support and a linker for 32-bit as well as 64-bit EFI applications that can be executed in an EFI boot console.

- Linux

  The Eigen Compiler Suite features runtime support and a linker for programs that can be executed under Linux-based operating systems.

- MMIX Simulator

  The Eigen Compiler Suite provides runtime support and a linker for object files that are executed within the MMIX simulator.

- OpenRISC 1000 Simulator

  The Eigen Compiler Suite provides runtime support and a linker for object files that are executed within the OpenRISC 1000 simulator.

- Raspberry Pi

  The Eigen Compiler Suite provides runtime support and a linker for bootloaders running on the Raspberry Pi 2 Model B.

- RISC Microcontrollers

  The Eigen Compiler Suite provides runtime support for RISC microcontrollers.

- WebAssembly Environments

  The Eigen Compiler Suite provides runtime support and a linker for WebAssembly modules that can be executed in corresponding web environments.

- Windows

  The Eigen Compiler Suite provides runtime support and a linker for 32-bit as well as 64-bit programs that can be executed under Windows.

The Eigen Compiler Suite features additional linkers that are not mentioned above. See Chapter 22 for a list of all linkers provided by the Eigen Compiler Suite.

1.2 Design

The Eigen Compiler Suite was designed to be a simple and minimalistic but complete and self-contained toolchain for several different programming languages and hardware architectures. While the main objective of the Eigen Compiler Suite is to be self-hosting, its design strives for the following goals:

- Usability

  All tools featured by the Eigen Compiler Suite shall be easy to use and should not require complex installations or other prerequisites. Their user-friendliness shall be enabled by a common basic user interface which does not demand any options or configurations of the user. This design enforces stability and reproducibility because the result of executing any tool depends solely on the contents of its input files.

- Reliability

  All programming tools of the Eigen Compiler Suite shall be reliable and produce correct results as well as comprehensive diagnostic messages. For this purpose, the Eigen Compiler Suite contains several test and validation suites that enable automatic regression testing of its various tools. In general, the implementation of the Eigen Compiler Suite is mainly driven by correctness, robustness, and simplicity. Therefore, optimizations and performance are explicitly secondary.

- Portability

  The Eigen Compiler Suite is completely written using standard and portable programming language features in order to guarantee the portability of its source code. All tools of the Eigen Compiler Suite are therefore compilable and executable within different runtime environments. Thus, all compilers featured by the Eigen Compiler Suite are cross compilers by design.

- Interoperability

  Tools like compilers and assemblers shall enable interoperability in-between all programming languages supported by the Eigen Compiler Suite. Usually, programs are written using a single programming language. Interoperability in this case means, that some parts of the program can also be implemented using a different programming language or with the help of an assembler. In the end, all these parts work seamlessly together and constitute the complete program.

- Textual Intermediate Representations

  The various tools of the Eigen Compiler Suite are typically used in a chain, such that the output of one tool becomes the input of the next one. However, all data that is transported this way should be represented using a human-readable and machine-independent text format. This design enables programmers and maintainers to view, modify and even manually create all kinds of intermediate data. In addition to the tools themselves, also the temporary data they generate is therefore portable across different runtime environments.

- Reuse of Generic Abstractions

  The implementation of the Eigen Compiler Suite shall provide generic abstractions that help to achieve all goals mentioned above. Additionally, the abstractions shall enable a good code reuse within the implementation itself. The most important abstractions provided by the Eigen Compiler Suite are described in Section 1.3.

- Complete and Consistent Documentation

  All features of the Eigen Compiler Suite and especially the usage of its toolchain shall be documented in detail. For all major components like implementations of a programming language or supported hardware architectures, there are consistent documentations available which describe the usage of the corresponding component and its implementation by the Eigen Compiler Suite. This user manual merges all of these documentations into a single document.

Some design guidelines like reliability and interoperability have also been incoorporated into the logo of the Eigen Compiler Suite which combines the three self-contained letters of its abbreviation into a robust three-dimensional structure as shown in Figure 1.1.

1.3 Abstractions

The Eigen Compiler Suite supports several different programming languages as well as several different target hardware architectures. One goal of the Eigen Compiler Suite is to provide compilers, assemblers, and linkers for all possible combinations of programming languages

Figure 1.1: The logo of the Eigen Compiler Suite

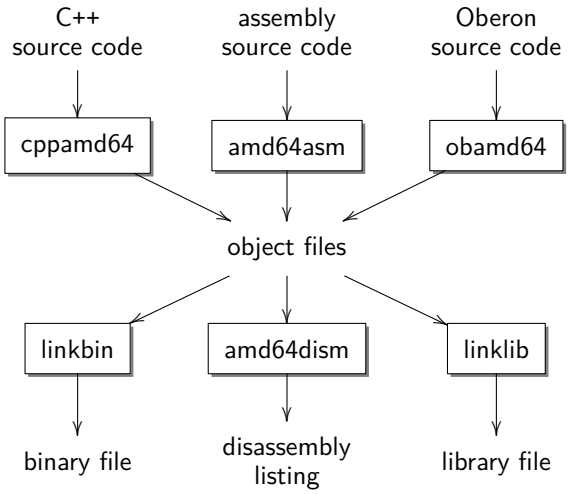


Figure 1.2: Some tools of the Eigen Compiler Suite that process object files

and target architectures. For this reason, the Eigen Compiler Suite defines the following generic abstractions:

- Object Files

  Object files are the key abstraction in order to enable interoperability between all compilers and assemblers of the Eigen Compiler Suite. All these tools generate the same kind of output called object files which can be processed by all linkers and disassemblers of the Eigen Compiler Suite. Figure 1.2 shows how object files are used in-between some compilers, assemblers, linkers, and disassemblers for the AMD64 hardware architecture. For more information about object files and their purpose, see Chapter 22.

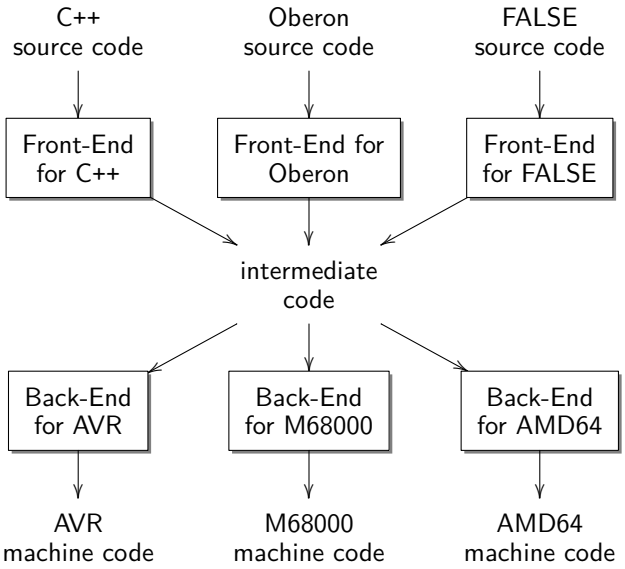


Figure 1.3: Some front-ends and back-ends provided by the Eigen Compiler Suite

- Intermediate Code

  Intermediate code is the generic abstraction in-between *front-ends* implementing programming languages and *back-ends* targeting hardware architectures. Front-ends implement the actual translation of source code into intermediate code, whereas back-ends translate intermediate code into machine code for the respective architecture. Figure 1.3 shows some front-ends and back-ends supported by the Eigen Compiler Suite. Each data flow within this diagram depicts the input and output of a single compiler for a particular programming language targeting a particular hardware architecture. As a consequence, programmers only have to provide a single front-end or back-end in order to establish all possible combinations with existing implementations of programming languages and target architectures. For more information about intermediate code and its purpose, see Chapter 23.

- Generic Assembly Language

  The generic assembly language is an abstraction for the common features of all assemblers provided by the Eigen Compiler Suite. It is capable of representing complete assembly programs including all the different instruction sets supported by the Eigen Compiler Suite. Concrete assembler tools only have to implement the translation of

architecture specific instructions into their binary representation. Everything else that is actually not dependent on the actual hardware architecture can therefore be translated separately. For more information about the generic assembly language and how to use it, see Chapter 8.

- Debugging Information

  Debugging information is a generic abstraction of symbolic metadata about a program generated by compilers. It represents all programming language constructs like functions, variables, data types, and statements compiled into an object file. Converter tools convert debugging information into a binary debugging data format which decouples compilers from the debugger of the target runtime environment. For more information about debugging information and its representation, see Chapter 24.

Concrete information about how programmers can make use of these abstractions in order to extend the Eigen Compiler Suite with additional support for programming languages and hardware architectures is given in Chapter 26.

2 User Interface

The Eigen Compiler Suite features a variety of different tools like compilers, assemblers, and linkers. This chapter describes their input and output and explains the common user interface that is shared across all of these tools.

The trivial round, the common task,
would furnish all we ought to ask.

— John Keble

2.1 Introduction

The Eigen Compiler Suite is a toolchain that consists of several different programming tools targeting a variety of programming languages, hardware architectures, and runtime environments. The following list categorizes all tools provided by the Eigen Compiler Suite and visualizes their input and output:

- Preprocessors

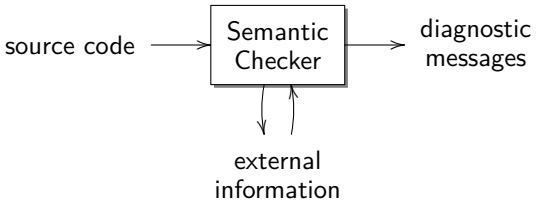
  Preprocessors are tools that perform simple lexical substitutions like macro expansions and conditional inclusion on source code written in a programming language. This very first phase of translating the source code is also part of all subsequent tools if the corresponding programming language supports preprocessing. An example of a preprocessor tool is `cppprep`.

- Pretty Printers

  Pretty printers are tools that reformat source code written in a programming language by printing it again using a consistent layout. Examples of pretty printer tools are `asmprint` and `falprint`.

- Semantic Checkers

  Semantic checkers perform syntactic and semantic checks on source code written in a programming language and print diagnostic messages. Some checkers may have to process additional semantic information stored in separate files. This phase of translating the source code is also part of all subsequent tools. Examples of semantic checker tools are `cppcheck` and `cdcheck`.

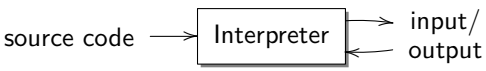


- Serializers

  Serializers are debugging tools that dump the complete internal representation of a program in a human readable format. This potentially also allows external development tools to make use of the same internal program representation. Examples of serialization tools are `obdump` and `faldump`.

- Interpreters

  Interpreters read and process source code for a programming language by emulating a runtime environment for this language and executing the program accordingly. The actual interpreted program defines the input and output the interpreter processes and produces. Examples of interpreter tools are `cpprun` and `cdrun`.

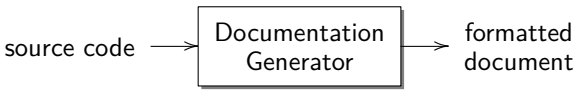


- Transpilers

  Transpilers translate source code written in one programming language into source code for another programming language. They store the resulting code in corresponding source code files. Examples of transpiler tools are `falcpp` and `obcpp`.

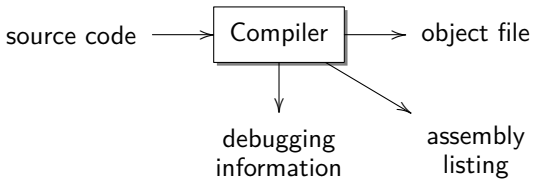
- Documentation Generators

  Documentation generators extract the structure and comments of the source code in order to generate documentations about the program. They store the resulting documentation in document files of various formats. For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25. Examples of documentation generator tools are `cpphtml` and `oblatex`.



- Compilers

  Compilers translate programs written in a programming language into machine code for some hardware processors. They store the resulting binary representation of the source code in object files. For more information about object files and their purpose, see Chapter 22. In addition, they also create debugging information and assembly code listings of the generated machine code. Examples of compiler tools are `cppamd64` and `obavr`.

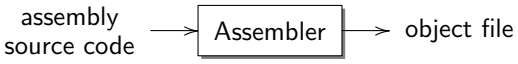


- Debugging Information Converters

  Debugging information converters process debugging information files and generate debugging object files storing a binary representation thereof in a format suitable for the debugger of a specific runtime environment. For more information about debugging information and its representation, see Chapter 24. An example of a debugging information converter tool is `dbgdwarf`.

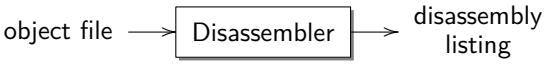
- Assemblers

  Assemblers translate assembly code for some specific hardware architecture into equivalent binary machine code. They store the resulting binary representation of the source assembly code in object files. For more information about the generic assembly language and how to use it, see Chapter 8. Examples of assembler tools are `arma32asm` and `ppcasm`.



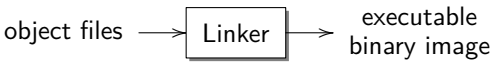
- Disassemblers

  Disassemblers translate machine code for some specific hardware architecture stored in object files by printing it using a human-readable disassembly listing. Examples of disassembler tools are `mibldism` and `arma64dism`.



- Linkers

  Linkers assemble the binary representation of a program stored in object files to generate output files that are executable on some target platforms. Examples of linker tools are `linkbin` and `linkprg`.



Although all of these tools process and produce different kinds of input and output, all of them present the same command-line interface to the user which is described in the following sections.

2.2 Command-Line Arguments

Each tool of the Eigen Compiler Suite provides a command-line user interface and accepts zero, one or more command-line arguments. If there are one or more command-line arguments, each of them is treated as the name of an input file relative to the current working directory. The notion of command-line options is not supported in order to make the contents of output files independent from the tool invocation. Some tools like preprocessors however may require additional user-defined information in which case they read documented environment variables.

Input files are plain text files containing the source code or other contents to be processed. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. If the user does not provide any command-line argument, the tool reads the source code from the standard input stream and prints its version and a short copyright notice on terminals.

If there are more than one command-line argument, each input file is processed in sequence according to the given order of the arguments. If there are errors during the processing of an input file, subsequent command-line arguments are ignored. Some tools like assemblers and disassemblers do not depend on other input except for the actual source code. In this case, the order of the command-line arguments is irrelevant and it would be equivalent to execute the tool once for every input file.

However, there are tools that depend on the actual order of the command-line arguments. They even may behave differently if executed several times with the same command-line argument. This may especially be the case for compilers which may store additional semantic information in separate files for every input file they compile. In addition, interpreters may also depend on the actual order of the input files they have processed so far.

On the other hand, it is crucial for some tools to process several input files during the same execution of the tool. Particularly linkers have to collect the information stored in several object files to generate an executable binary image. In this case, the order of the arguments is not important, since all information is collected first before the linking process is started. However, linkers generate output files named after the very last argument with the filename extension being replaced by an appropriate suffix.

2.3 Diagnostic Messages

All tools of the Eigen Compiler Suite use the same scheme for the diagnostic messages they generate. The following list describes the different types of messages used by the Eigen Compiler Suite:

- Errors

  Error messages indicate a problem within the contents of an input file that prohibits the tool from proceeding successfully. Compilers for example use error messages to output syntactic or semantic errors that have to be fixed by the programmer. In general, tools diagnose only the first few of the encountered problems but always yield a return code indicating the failure.

- Fatal Errors

  Fatal errors indicate a general failure of the tool that cannot be fixed by changing the contents of the input file. This includes for example internal program errors, failures to open files, or critical system conditions like out of memory.

- Warnings

  Warning messages indicate potential flaws within the contents of an input file. They do not cause the tool to fail but they identify issues that may lead to unexpected behavior. Therefore, warnings may be prevented by changing the contents of an input file without changing its semantics.

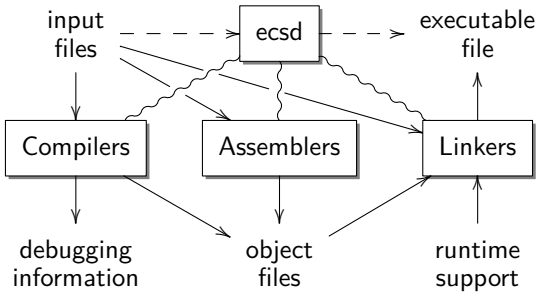
- Notes

  Notes give additional information about the result of processing an input file. They are often used for debugging purposes or to diagnose violations of coding conventions.

All diagnostic messages are written to the standard error stream. Each message consists of a short line of text and the name of the input file or the tool that caused the diagnostic message. The message also contains the position within the input file, if the problem can be located. Since input files are text files in general, the position is usually given as the number of the text line containing the problem followed by the column therein. For convenience, some tools like compilers also indicate the problematic position by reproducing the respective line of text from the input file.

2.4 The Eigen Compiler Suite Driver

The Eigen Compiler Suite provides a utility tool called `ecsd` which autonomously drives its toolchain in order to build complete executable files for a specific runtime environment. It infers the set of required tools like compilers, assemblers, and linkers from the type of its input files and automatically invokes one tool after the other with appropriate command-line arguments and environment variables.



In contrast to all other tools of the Eigen Compiler Suite, this utility tool does support command-line options which influence how tools are identified and invoked. The following list describes the form and behavior of the most important options:

`-b` *directory* | `--base` *directory* — Use Specified Base Directory

Allows overriding the base directory provided by the `ECSBASE` environment variable which is used to locate tools and the necessary runtime support. If omitted, the tool infers the base directory from its own location.

`-c` | `--compile` — Compile and Assemble Only

Compiles and assembles input files without invoking the linker at the end. The output is a set of object files rather than an executable file.

`-d` | `--disassemble` — Disassemble Object Files

Invokes the disassembler instead of the linker at the end. The output is a set of disassembly listings rather than an executable file.

`-f` *format* | `--file` *format* — Use Specified File Format

Specifies the output file format of the linker. The available file formats correspond to the suffixes of the linker tools listed in Chapter 22. If omitted, the tool uses the default file format of the target environment.

`-g` | `--generate` — Generate Debugging Information

Generates and includes debugging information during linking. The resulting executable file can be executed and tested using a symbolic debugger.

`-h` | `--help` — Print Command-Line Help

Prints information about all supported command-line arguments. This especially also includes all supported source types and available target environments.

`-i` *tool* | `--invoke` *tool* — Invoke Specified Tool Only

Processes either input files or the standard input stream using the specified tool. The output is whatever that tool generates rather than an executable file. See page 529 for an index of all available tools.

`-l` | `--library` — Create Library File

Combines input files into a single object file. The output is a library file rather than an executable file.

`-p` | `--protect` — Protect Environment Variables

Prevents the environment from being automatically modified when invoking tools that depend on environment variables.

`-r` *support* | `--runtime` *support* — Include Specified Runtime Support

Includes additional runtime support stored in the specified object or library file during linking. See page 537 for an index of all available runtime support.

`-s` *type* | `--source` *type* — Use Specified Source Type

Allows defining the source type of input files for which the corresponding tool could not be inferred from the filename extension.

`-t` *environment* | `--target` *environment* — Target Specified Environment

Specifies the target environment for cross compilations. See page 541 for an index of all available target environments. If omitted, the tool tries to target its own runtime environment.

`-v` | `--verbose` — Print Command Line

Enables verbose mode which prints the actual command line before invoking any tool. This is helpful for examining how and which tools are invoked and what runtime support is provided.

The `ecsd` driver tool is most useful when it is accessible via the `PATH` environment variable of the runtime environment and is part of an appropriate installation of the Eigen Compiler Suite. See Chapter 3 for some examples of using this utility tool in practice.

3 Getting Started

This chapter provides a basic guide for novice users of the Eigen Compiler Suite. It explains how to get started with the toolchain and shows some typical use cases. In the end, users will understand the functionality of the Eigen Compiler Suite by building executable programs.

Man fühlt den Glanz von einer neuen Seite,
auf der noch alles werden kann.

— Rainer Maria Rilke

3.1 Introduction

The Eigen Compiler Suite features a set of development tools for several programming languages targeting a variety of hardware architectures. This chapter demonstrates how to use these tools in order to build executable applications. The assumption is that the user already knows how to program in the programming languages presented in this chapter. For each programming language supported by the Eigen Compiler Suite there are the following tools for processing and translating the source code:

- Pretty printers for reformatting the source code
- Semantic checkers for analyzing the source code for semantic errors
- Interpreters for executing the source code line by line
- Compilers for translating the source code into executable machine code

All of these tools have the same functionality regardless of which programming language they actually implement. Without loss of generality, the remainder of this guide therefore focuses exemplarily on the Oberon programming language. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. Regarding the hardware architectures on the other hand, the Eigen Compiler Suite provides the following tools for each supported hardware architecture:

- Assemblers for translating assembly code into binary machine code
- Disassemblers for translating machine code into human-readable text

The Eigen Compiler Suite provides a couple of assemblers, all of which implement the same generic assembly language. For more information about the generic assembly language and how to use it, see Chapter 8. In general, users can target any hardware architecture supported by the Eigen Compiler Suite by just replacing the corresponding tools and their runtime support. For instructional purposes, the remainder of this chapter therefore assumes that the user targets the AMD64 hardware architecture. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. Finally, for building executable programs generated by compilers and assemblers the Eigen Compiler Suite provides the following tools:

- Linkers for combining binary data into an executable file
- Debugging tools for processing debugging information

The debugging information generated by compilers and linkers allows debugging problematic programs and consists of a symbolic address mapping. For more information about debugging information and its representation, see Chapter 24.

3.2 Prerequisites

In order to process and build the examples shown in this guide, the user needs access to the following tools:

- `obprint` for pretty printing Oberon modules
- `obcheck` for analyzing Oberon modules
- `obrun` for interpreting Oberon modules
- `obamd64` for compiling Oberon modules
- `amd64asm` for translating assembly code
- `amd64dism` for translating the generated machine code
- `linklib` for combining object files
- `linkbin` for linking an executable program
- `dbgdwarf` for converting debugging information
- `mapplace` and `mapsearch` for debugging programs

The examples shown in Sections 3.3 to 3.6 assume that all of these tools are executable from the current working directory. The illustrated commands to invoke these tools may therefore have to be adjusted to their actual location and may additionally require a suffix for executable files. Furthermore, all executable programs created using the compiler require runtime support which is provided by the following object and library files:

- `obamd64run.lib` required by the programming language
- `amd64run.obf` required by the hardware architecture
- `win64run.obf`, `amd64darwinrun.obf`, or `amd64linuxrun.obf` required by runtime environments like Windows, Darwin, or Linux-based operating systems

These files are assumed to be located in the current working directory as well and may therefore also have to be adjusted accordingly. They store binary data such as the machine code of precompiled runtime support for the respective hardware architecture and runtime environment. For more information about object files and their purpose, see Chapter 22.

Installations of the Eigen Compiler Suite typically feature a driver utility tool called `ecsd` which is able to automatically locate and invoke all of the tools discussed in this section. It also provides the necessary runtime support without requiring users to explicitly name any of the prerequisites listed above. See Section 3.7 for more information about how to run the examples described in Sections 3.3 to 3.6 using the Eigen Compiler Suite driver. For more information about this tool in general, see Chapter 2.

The remainder of this section gives more detailed information about the required tools. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

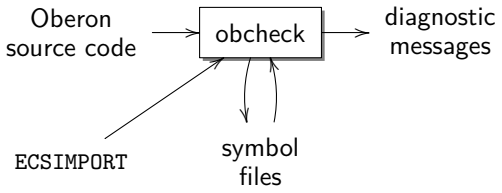
3.2.1 obprint

`obprint` is a pretty printer for the Oberon programming language. It reformats the source code of Oberon modules and writes it to the standard output stream.

3.2.2 obcheck

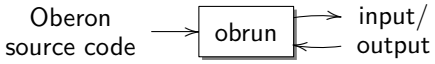
`obcheck` is a syntactic and semantic checker for the Oberon programming language. It just performs syntactic and semantic checks on Oberon modules and writes its diagnostic messages

to the standard error stream. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.



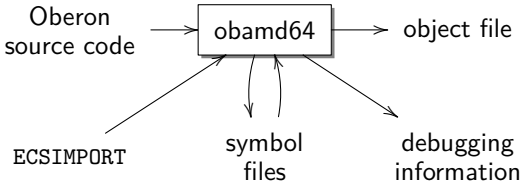
3.2.3 obrun

`obrun` is an interpreter for the Oberon programming language. It processes and executes modules written in Oberon. This tool does neither generate nor process symbol files while interpreting modules. If a module is imported by another one, its filename has to be named before the other one in the list of command-line arguments.



3.2.4 obamd64

`obamd64` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd64run` library file.



3.2.5 amd64asm

`amd64asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the AMD64 architecture.

3.2.6 amd64dism

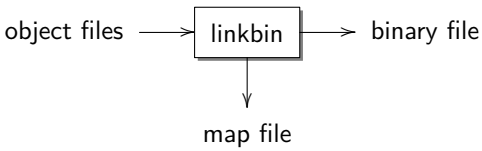
`amd64dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the AMD64 architecture.

3.2.7 linklib

`linklib` is an object file combiner. It creates a static library file by combining all object files given to it into a single one.

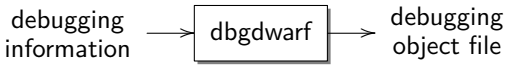
3.2.8 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



3.2.9 dbgdwarf

`dbgdwarf` is a DWARF debugging information converter tool. It converts debugging information into the DWARF debugging data format and stores it in corresponding object files [1]. The resulting debugging object files can be combined with runtime support that creates Executable and Linking Format (ELF) files [2].

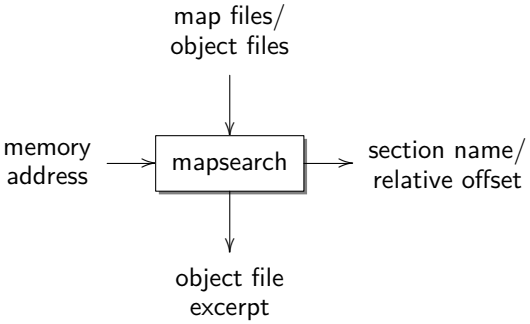


3.2.10 mapplace

`mapplace` is a map tool provided for debugging purposes. It converts map files generated by linker tools into an object file called map place result by placing all listed binary sections at their corresponding address using fixed phantom sections. Map place results can be linked together with debugging object files to generate separate symbol files for external debuggers.

3.2.11 mapsearch

`mapsearch` is map tool provided for debugging purposes. It searches map files generated by linker tools for the name of a binary section that encompasses a memory address read from the standard input stream. If additionally provided with one or more object files, it also stores an excerpt thereof in a separate object file called map search result which only contains the identified binary section for disassembling purposes.



3.3 Basic Processing

The first example is a very simple Oberon module that defines a global variable, assigns a value to it and prints that value. The corresponding source code looks as follows:

```
MODULE Simple;
VAR variable: INTEGER;
BEGIN
  variable := 10;
  TRACE (variable);
END Simple.
```

The remainder of this section assumes that these six lines of source code are stored in a plain text file called `simple.mod` in the current working directory.

3.3.1 Pretty Printing

Executing the pretty printer should yield the very same code in a slightly different layout and helps to check whether the source was syntactically recognized. The corresponding command-line call looks like the following and produces the subsequent output:

```
obprint simple.mod
MODULE Simple;

VAR
  variable: INTEGER;

BEGIN
  variable := 10;
  TRACE (variable);
END Simple.
```

3.3.2 Semantic Checking

For checking whether the source code is also semantically valid one can execute the semantic checker using the following command-line call:

```
obcheck simple.mod
```

Since the input was semantically valid, the invocation of this tool does not generate any output message and succeeds. However, if line 5 of the original source code was for example changed to `TRACE (x);` then the tool would identify a violation of the language rules:

```
obcheck simple.mod
simple.mod:5:10: error: undeclared identifier 'x'
```

If the semantic checker succeeds, it generates a plain text file called `simple.sym` in the current working directory which contains additional semantic information about the source code. In the case of Oberon, this file is a symbol file containing the interface of the sample module which is required by other modules importing it.

3.3.3 Interpreting

Valid modules can be executed by the interpreter which simulates a runtime environment for the Oberon language. The corresponding command-line call looks like the following and produces the subsequent output:

```
obrun simple.mod
simple.mod:5:10: note: 'variable' = 10
```

The interpreter does not generate any output files. It just executes the source code line by line by first assigning the value 10 to the variable and then printing its value.

3.4 Building Simple Programs

Running code using the interpreter is quite slow compared to executing the same program directly on the machine. This section shows how the same source code can be translated into an executable program.

3.4.1 Compiling

The compiler translates the source code of the sample module into machine code using the following command-line call:

```
obamd64 simple.mod
```

If its invocation is successful, the compiler generates four different plain text files for this module. The first file `simple.sym` is the same symbol file as generated by the semantic checker. The second file `simple.obf` is an object file which stores the binary encoding of the machine code translated from the source code. The third file `simple.lst` is a human-readable assembly code listing of the generated machine code. The fourth file `simple.dbg` contains additional symbolic debugging information about the module. The last two files are actually not required for building the executable program but are generated for debugging purposes. Invoking the assembler on the assembly code listing using the following command-line call yields exactly the same object file:

```
amd64asm simple.lst
```

3.4.2 Linking

Although the generated object file contains executable machine code, it is not executable by itself. The machine code has first to be combined with the required runtime support for the programming language, the hardware architecture, as well as the runtime environment. This process is called linking and can be executed by invoking the linker using the following command-line call. The actual order of the command-line arguments is not important but the linker takes the filename of the last object file as a basis for its output files:

```
linkbin obamd64run.lib amd64run.obf win64run.obf simple.obf
```

If this call succeeds, the linker generates a binary image of an executable program and stores it in a file called `simple.exe`. This file can be executed on a machine running the Windows operating system producing exactly the same output as the interpreter above. For creating a similar executable file called `simple` for the Darwin operating system, only the corresponding command-line argument has to be changed:

```
linkbin obamd64run.lib amd64run.obf amd64darwinrun.obf simple.obf
```

This executable program still needs the same runtime support for the programming language and hardware architecture. The same holds for programs that can be executed on Linux-based operating systems. Again only the corresponding command-line argument has to be modified accordingly:

```
linkbin obamd64run.lib amd64run.obf amd64linuxrun.obf simple.obf
```

In all three cases, the linker also generates a plain text file called `simple.map` summarizing the mapping of symbol names in the source code to their linked addresses. This information is provided for debugging purposes and used in Section 3.6.

3.5 Building Complex Programs

The second example makes use of the interoperability features of the Eigen Compiler Suite. It shows how code from one programming language can access code and data defined in another language and vice versa.

3.5.1 Compiling

Here is an extended version of the first example. Instead of assigning the value to the global variable directly, this version calls a procedure that overwrites the value of the variable. In the end however, it should generate the same output as the previous example:

```
MODULE Complex;
IMPORT SYSTEM;
VAR variable: INTEGER;
PROCEDURE ^ Assign ["assign"] (value: INTEGER);
BEGIN
  Assign (10);
  TRACE (variable);
END Complex.
```

The forward declaration of the procedure is marked as external which causes the compiler to refer to the procedure using the given external name instead of defining a new procedure. If the source code is stored in a plain text file called `complex.mod`, the compiler can be invoked using the following command-line call:

```
obamd64 complex.mod
```

The resulting plain text files are called `complex.sym`, `complex.obf`, `complex.lst`, and `complex.dbg`. However, linking the generated object file together with the runtime support for any runtime environment as before yields the following linker error:

```
linkbin obamd64run.lib amd64run.obf amd64linuxrun.obf complex.obf
Complex._body: error: unresolved symbol 'assign'
```

The diagnostic message states that the body of the module calls a procedure called `assign` which is defined nowhere. This behavior is intended because all external procedures need to be defined by another part of the program. The following section shows how to implement it using assembly language.

3.5.2 Assembling

The missing `assign` procedure is defined in the following assembly source code. It contains instructions which copy the procedure argument into a register and from there into the global variable:

```
.code assign
  mov  ebx, [rsp + 8]
  mov  [@Complex.variable], ebx
  ret
```

Assuming these four lines of source code are stored in a plain text file called `assign.asm`, the assembler can be invoked using the following command-line call:

```
amd64asm assign.asm
```

This generates an object file called `assign.obf` which stores the binary representation of the corresponding machine code of this procedure. Its contents can be listed by the disassembler using the following command-line call which produces the subsequent output:

```
amd64dism assign.obf
.code assign
   0  8b5c2408        mov  ebx, dword [rsp + 8]
   4  891c2500000000  mov  dword [0], ebx  ; @Complex.variable
  11  c3              ret
  12
```

The disassembler lists all three instructions which require twelve octets of code space in total. Each instruction is prefixed with its relative offset and binary encoding.

3.5.3 Combining

The following step is optional but shows how the object files generated by the compiler and the assembler can be merged together. The result is a new object file called a library file which combines the machine code of the Oberon module with the procedure defined in the assembly code. It is generated using the following command-line call:

```
linklib assign.obf complex.obf
```

This call generates a new library file called `complex.lib` which contains the contents of both object files. This file can be used to link the executable program in the next step.

3.5.4 Linking

Since the library file contains the machine code of both the module and the `assign` procedure, the linker should not fail any longer. The corresponding command-line call looks like the following:

```
linkbin obamd64run.lib amd64run.obf amd64linuxrun.obf complex.lib
```

This call generates a new binary executable file called `complex` that produces the same output as before when executed. The second generated file `complex.map` contains the mapping of symbol names in the source code to their linked addresses and includes the new `assign` procedure.

The linker could have also been called using the two object files generated by the assembler and the compiler instead of the single library file yielding exactly the same result. This combining and linking works for arbitrary object files generated by the various development tools of the Eigen Compiler Suite as long as they target the same hardware architecture.

3.6 Debugging Programs

Compilers and linkers provided by the Eigen Compiler Suite generate various files for debugging purposes. This section shows how this symbolic debugging information can be used to debug problematic programs.

3.6.1 Basic Debugging

If an erroneous program fails, the corresponding error message of the runtime environment or debugger often mentions the address of the first problematic instruction. The Eigen Compiler Suite provides a debugging tool called `mapsearch` which allows searching map files generated by linker tools for the section encompassing the reported address. This is helpful for identifying the executed code section and all other active procedures in a stack trace. The tool reads a single address from the standard input stream and prints the name of the identified section if it is listed in the specified map file. For convenience, the address can be provided by a single command-line call as follows:

```
echo 0x80482a7 | mapsearch complex.map
mapsearch: note: offset 4 in code section 'assign'
```

The address in this particular example corresponds to the instruction at offset four relative to the `assign` procedure. If additionally provided with the object file containing the identified section, the tool also generates a separate object file called `complex.msr` which can be used to disassemble only that section:

```
echo 0x80482a7 | mapsearch complex.map complex.lib
amd64dism complex.msr
```

Searching map files generated by linkers for problematic addresses is a rather limited debugging facility but always available.

3.6.2 Advanced Debugging

For runtime environments that support more sophisticated debuggers, the Eigen Compiler Suite provides tools that convert the symbolic debugging information generated by its compilers alongside an object file into a suitable format. This debugging information consists of an abstract representation of all programming language constructs compiled into the object file and can be converted as follows for example:

```
dbgdwarf complex.dbg
```

This generates a debugging object file called `complex.dbf` which can optionally be provided together with the corresponding object file when linking for Linux-based operating systems:

```
linkbin obamd64run.lib amd64run.obf amd64linuxrun.obf
        assign.obf complex.obf complex.dbf
```

The resulting binary executable file behaves as before but additionally contains debugging information that enables interactive program animation and memory inspection when loaded in an external debugger.

3.6.3 Remote Debugging

Some debuggers support loading the symbolic debugging information from separate symbol files alongside the executable file which is often required when remote debugging. The following command generates a binary image for the Windows operating system without debugging information:

```
linkbin obamd64run.lib amd64run.obf win64run.obf complex.lib
```

The map file generated by the linker can be converted into an additonal object file which contains the same mapping of symbol names:

```
mapplace complex.map
```

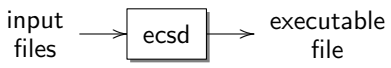
The resulting object file called `complex.mpr` can then be used to generate a symbol file suitable for the debugger of a Linux-based operating system for example:

```
linkbin amd64run.obf amd64linuxrun.obf complex.mpr complex.dbf
```

The resulting symbol file called `complex` resembles an executable file but should merely be used to load symbols in the debugger since it contains only symbolic debugging information. This conversion of symbolic debugging information works as long as the remote system and the debugger of the host system target the same hardware architecture.

3.7 Using the Eigen Compiler Suite Driver

In addition to invoking tools directly as shown above, users can also call the `ecsd` driver utility tool which is typically provided when installing the Eigen Compiler Suite. It allows compiling and linking complete executable files for a specific runtime environment in one step and automatically provides the necessary runtime support.



In contrast to all other tools of the Eigen Compiler Suite, the driver tool does support the notion of command-line options to allow users to influence how tools are identified and invoked. The following command-line call prints a list of all supported command-line arguments:

```
ecsd -h
```

The remainder of this section consistently uses a short notation for options although all of them have also a longer and more descriptive form prefixed by two dashes. For more information about the user interface of this tool, see Chapter 2.

3.7.1 Basic Processing

All of the tools mentioned in Section 3.3 can also be invoked using the Eigen Compiler Suite driver by putting `ecsd -i` in front of the corresponding command-line call. This particular option prompts the driver to process the input files given as command-line arguments using the specified tool. Provided that the corresponding tools are available, the command-line calls for pretty printing, semantic checking, and interpreting for example look like the following:

```
ecsd -i obprint simple.mod
ecsd -i obcheck simple.mod
ecsd -i obrun simple.mod
```

The actual command-line call used by the driver to invoke the respective tool can be shown by adding the `-v` flag. This enables verbose mode which is especially helpful when building programs as shown in the following section.

3.7.2 Building Programs

If the driver is called without command-line options, it builds complete executable files for a specific runtime environment as described in Section 3.4. It infers the set of required tools like compilers, assemblers, and linkers from the type of its input files and automatically invokes one tool after the other with appropriate command-line arguments. By default, it tries to target its own runtime environment:

```
ecsd simple.mod
```

In order to explicitly target a specific runtime environment like Windows, Darwin, or Linux-based operating systems for example, use the `-t` option as follows. A list of all available target environments is accessible using the `-h` flag:

```
ecsd -t win64 simple.mod
ecsd -t amd64darwin simple.mod
ecsd -t amd64linux simple.mod
```

Using the driver to build programs still generates intermediate files like object files and assembly code listings as before. Use the `-c` flag to compile and assemble input files without invoking the linker at the end:

```
ecsd -c simple.mod
```

The driver can also be called using the generated intermediate files. However, because the driver cannot infer which compiler was used to generate object files or assembly code listings in the first place, it must be explicitly told which additional runtime support to include this time. In the case of the Oberon programming language, the required runtime support is made available using the `-O` flag:


```
ecsd -O simple.obf
ecsd -O simple.lst
```

The driver is able to process several different input files at once which allows building complex programs as described in Section 3.5:

```
ecsd assign.asm complex.mod
```

Combining input files into a single object file can be achieved using the `-l` flag. As before, the resulting library file requires additional runtime support to be specified explicitly when building the program:

```
ecsd -l assign.obf complex.mod
ecsd -O complex.lib
```

In order to disassemble source code or object files instead of linking them, the driver can be invoked using the `-d` flag:

```
ecsd -d assign.asm
ecsd -d assign.obf
```

Generally, the type of an input file is inferred from its filename extension. The `-s` option allows users to explicitly specify the source type of unknown input files. A list of all supported source types is accessible using the `-h` flag.

3.7.3 Debugging Programs

The Eigen Compiler Suite driver also allows processing map files generated by linker tools. This is useful for identifying sections that contain problematic instructions in erroneous programs as described in Section 3.6. Building a complex program as described in the previous section for example generates a file called `complex.map`:

```
ecsd assign.asm complex.mod
```

This file can be passed to the driver which then invokes a debugging tool that reads an address from the standard input stream in order to search for a linked section that encompasses that address. If the address is mapped, the tool prints the name of the corresponding section and the relative offset of the address:

```
echo 0x4011fe | ecsd complex.map
mapsearch: note: offset 4 in code section 'assign'
```

If additionally provided with the object file or runtime support containing the identified section, the driver subsequently disassembles the latter when using the `-d` flag. In this particular example, the input address points to the second instruction of the `assign` procedure written in assembly code:

```
echo 0x4011fe | ecsd -d complex.map assign.obf
mapsearch: note: offset 4 in code section 'assign'
.code assign
   0  8b5c2408        mov  ebx, dword [rsp + 8]
   4  891c2500000000  mov  dword [0], ebx  ; @Complex.variable
  11  c3              ret
  12
```

For more advanced debugging using external debuggers on runtime environments that support them, the Eigen Compiler Suite driver automatically converts and includes the symbolic debugging information generated by compilers using the `-g` flag:

```
ecsd -g simple.mod
ecsd -g assign.asm complex.mod
```

This generates executable files that incorporate the address mapping of all symbols and instructions necessary for interactive program animation and memory inspection when loaded in a debugger. In order to store this symbolic debugging information in a separate symbol file alongside the executable file as is often required for remote debugging on a different system, the `-g` flag can also be used to automatically convert the information stored in the map file generated above:

```
ecsd -g complex.map complex.dbg
```

The resulting symbol file can then be loaded in the debugger when remote debugging as long as the remote system and the debugger of the host system target the same hardware architecture.

4 Tool Reference

This chapter lists all 162 tools provided by the Eigen Compiler Suite in alphabetical order. All of them share the same command-line user interface as detailed in Chapter 2. The implementation-defined behavior of programming language-specific tools like pretty printers, semantic checkers, serializers, interpreters, and compilers is described in Chapters 5 to 7. For more information about assembler and disassembler tools, see Chapter 8. The hardware architectures targeted by all of these tools are described in Chapters 9 to 21 whereas Chapter 22 details linker tools and their functionality. For more information about debugging tools and documentation generators, see Chapters 23 to 25.

This world is but canvas to our imaginations.

— Henry David Thoreau

amd16asm is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. *See Chapter 8/9/22*

amd16dism is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 16-bit operating mode defined by the AMD64 architecture. *See Chapter 8/9/22*

amd32asm is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. *See Chapter 8/9/22*

amd32dism is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 32-bit operating mode defined by the AMD64 architecture. *See Chapter 8/9/22*

amd64asm is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. *See Chapter 8/9/22*

amd64dism is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the AMD64 architecture. *See Chapter 8/9/22*

arma32asm is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing A32 instructions and stores it in corresponding object files. *See Chapter 8/10/22*

arma32dism is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing A32 instructions into assembly code and writes it to the standard output stream. *See Chapter 8/10/22*

arma64asm is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing A64 instructions and stores it in corresponding object files. *See Chapter 8/10/22*

arma64dism is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing A64 instructions into assembly code and writes it to the standard output stream. *See Chapter 8/10/22*

armt32asm is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing T32 instructions and stores it in corresponding object files. *See Chapter 8/10/22*

armt32dism is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing T32 instructions into assembly code and writes it to the standard output stream. *See Chapter 8/10/22*

asmprint is a pretty printer for generic assembly code. It reformats generic assembly code and writes it to the standard output stream. *See Chapter 8*

avrasm is an assembler for the AVR hardware architecture. It translates assembly code into machine code for AVR processors and stores it in corresponding object files. The identifiers `RXL`, `RXH`, `RYL`, `RYH`, `RZL`, and `RZH` are predefined and name the corresponding registers. The identifiers `SPL` and `SPH` are also predefined and evaluate to the address of the corresponding registers. *See Chapter 8/11/22*

avrdism is a disassembler for the AVR hardware architecture. It translates machine code from object files targeting AVR processors into assembly code and writes it to the standard output stream. *See Chapter 8/11/22*

avr32asm is an assembler for the AVR32 hardware architecture. It translates assembly code into machine code for AVR32 processors and stores it in corresponding object files. *See Chapter 8/12/22*

avr32dism is a disassembler for the AVR32 hardware architecture. It translates machine code from object files targeting AVR32 processors into assembly code and writes it to the standard output stream. *See Chapter 8/12/22*

cdamd16 is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/9/22/23/24*

cdamd32 is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an in-

ternal data structure that is usually not accessible.
See Chapter 8/9/22/23/24

cdamd64 is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.
See Chapter 8/9/22/23/24

cdarma32 is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.
See Chapter 8/10/22/23/24

cdarma64 is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.
See Chapter 8/10/22/23/24

cdarmt32 is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.
See Chapter 8/10/22/23/24

cdarmt32fpe is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.
See Chapter 8/10/22/23/24

cdavr is a compiler for intermediate code targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/11/22/23/24*

cdavrxt is a compiler for intermediate code targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.
See Chapter 8/11/22/23/24

cdavr32 is a compiler for intermediate code targeting the AVR32 hardware architecture. It generates

machine code for AVR32 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/12/22/23/24*

cdcheck is a syntactic and semantic checker for intermediate code. It just performs syntactic and semantic checks on programs written in intermediate code and writes its diagnostic messages to the standard error stream. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/23*

cdm68k is a compiler for intermediate code targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/13/22/23/24*

cdmibl is a compiler for intermediate code targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/14/22/23/24*

cdmips32 is a compiler for intermediate code targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/15/22/23/24*

cdmips64 is a compiler for intermediate code targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/15/22/23/24*

cdmmix is a compiler for intermediate code targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/16/22/23/24*

cdopt is an optimizer for intermediate code. It performs various optimizations on programs written in intermediate code and writes the result to the standard output stream. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/23*

cdor1k is a compiler for intermediate code targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an in-

ternal data structure that is usually not accessible. *See Chapter 8/17/22/23/24*

cdppc32 is a compiler for intermediate code targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/18/22/23/24*

cdppc64 is a compiler for intermediate code targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/18/22/23/24*

cdrisc is a compiler for intermediate code targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/19/22/23/24*

cdrun is an interpreter for intermediate code. It processes and executes programs written in intermediate code. The following code sections are predefined and have the usual semantics: `abort`, `_Exit`, `fflush`, `floor`, `fputc`, `free`, `getchar`, `malloc`, and `putchar`. Diagnostic messages about invalid operations include the name of the executed code section and the index of the erroneous instruction. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/23*

cdwasm is a compiler for intermediate code targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/20/22/23/24*

cdxtensa is a compiler for intermediate code targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 8/21/22/23/24*

cppamd16 is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd16__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require

additional runtime support that is stored in the `cppamd16run` library file. *See Chapter 5/8/9/22/24*

cppamd32 is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd32run` library file. *See Chapter 5/8/9/22/24*

cppamd64 is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd64run` library file. *See Chapter 5/8/9/22/24*

cpparma32 is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__arma32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparma32run` library file. *See Chapter 5/8/10/22/24*

cpparma64 is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__arma64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparma64run` library file. *See Chapter 5/8/10/22/24*

cpparmt32 is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__armt32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparmt32run` library file. *See Chapter 5/8/10/22/24*

cpparmt32fpe is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__armt32fpe__` is predefined in order to enable

programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparmt32fperun` library file. *See Chapter 5/8/10/22/24*

cppavr is a compiler for the C++ programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avr__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavrrun` library file. *See Chapter 5/8/11/22/24*

cppavrxt is a compiler for the C++ programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avrxt__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavrxtrun` library file. *See Chapter 5/8/11/22/24*

cppavr32 is a compiler for the C++ programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avr32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavr32run` library file. *See Chapter 5/8/12/22/24*

cppcheck is a syntactic and semantic checker for the C++ programming language. It just performs syntactic and semantic checks on C++ programs and writes its diagnostic messages to the standard error stream. *See Chapter 5*

cppcode is an intermediate code generator for the C++ programming language. It generates intermediate code from programs written in C++ and stores it in corresponding assembly files. The macro `__code__` is predefined in order to enable programmers to identify this tool while generating intermediate code. Programs generated with this tool require additional runtime support that is stored in the `cppcoderun` assembly file. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 5/8/23*

cppdoc is a generic documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into a generic documentation. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 5/25*

cppdump is a serializer for the C++ programming language. It dumps the complete internal representation of programs written in C++ into an XML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 5*

cpphtml is an HTML documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into an HTML document. *See Chapter 5/25*

cpplatex is a Latex documentation generator for the C++ programming language. It processes sev-

eral C++ source files and assembles all information therein into a Latex document. *See Chapter 5/25*

cppm68k is a compiler for the C++ programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__m68k__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppm68krun` library file. *See Chapter 5/8/13/22/24*

cppmibl is a compiler for the C++ programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mibl__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmiblrun` library file. *See Chapter 5/8/14/22/24*

cppmips32 is a compiler for the C++ programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mips32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmips32run` library file. *See Chapter 5/8/15/22/24*

cppmips64 is a compiler for the C++ programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mips64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmips64run` library file. *See Chapter 5/8/15/22/24*

cppmmix is a compiler for the C++ programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mmix__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmmixrun` library file. *See Chapter 5/8/16/22/24*

cppor1k is a compiler for the C++ programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__or1k__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppor1krun` library file. *See Chapter 5/8/17/22/24*

cppppc32 is a compiler for the C++ programming language targeting the PowerPC hardware architec-

ture. It generates machine code for PowerPC processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__ppc32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppppc32run` library file. *See Chapter 5/8/18/22/24*

cppppc64 is a compiler for the C++ programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__ppc64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppppc64run` library file. *See Chapter 5/8/18/22/24*

cpppprep is a preprocessor for the C++ programming language. It preprocesses source code according to the rules of C++ and writes it to the standard output stream. Only the macro names `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__` are predefined. *See Chapter 5*

cppprint is a pretty printer for the C++ programming language. It reformats the source code of C++ programs and writes it to the standard output stream. *See Chapter 5*

cpprisc is a compiler for the C++ programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__risc__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpprisrun` library file. *See Chapter 5/8/19/22/24*

cpprun is an interpreter for the C++ programming language. It processes and executes programs written in C++. The macro `__run__` is predefined in order to enable programmers to identify this tool while interpreting. *See Chapter 5*

cppwasm is a compiler for the C++ programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__wasm__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppwasmrun` library file. *See Chapter 5/8/20/22/24*

cppxtensa is a compiler for the C++ programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__xtensa__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppxtensarun` library file. *See Chapter 5/8/21/22/24*

dbgdwarf is a DWARF debugging information converter tool. It converts debugging information into the DWARF debugging data format and stores it in corresponding object files [1]. The resulting debugging object files can be combined with runtime support that creates Executable and Linking Format (ELF) files [2]. *See Chapter 22/24*

doccheck is a syntactic and semantic checker for generic documentations. It just performs syntactic and semantic checks on generic documentations and writes its diagnostic messages to the standard error stream. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 25*

dochtml is an HTML documentation generator for generic documentations. It processes several generic documentations and assembles all information therein into an HTML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 25*

doclatex is a Latex documentation generator for generic documentations. It processes several generic documentations and assembles all information therein into a Latex document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 25*

docprint is a pretty printer for generic documentations. It reformats generic documentations and writes it to the standard output stream. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 25*

falamd16 is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. *See Chapter 6/9/22*

falamd32 is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. *See Chapter 6/9/22*

falamd64 is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. *See Chapter 6/9/22*

falarma32 is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/10/22*

falarma64 is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/10/22*

falarmt32 is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/10/22*

falarmt32fpe is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM

processors with floating-point extension executing T32 instructions from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/10/22*

falavr is a compiler for the FALSE programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/11/22*

falavrxt is a compiler for the FALSE programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/11/22*

falavr32 is a compiler for the FALSE programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/12/22*

falcheck is a syntactic and semantic checker for the FALSE programming language. It just performs syntactic and semantic checks on FALSE programs and writes its diagnostic messages to the standard error stream. *See Chapter 6*

falcode is an intermediate code generator for the FALSE programming language. It generates intermediate code from programs written in FALSE and stores it in corresponding assembly files. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 6/8/23*

falcpp is a transpiler for the FALSE programming language. It translates programs written in FALSE into C++ programs and stores them in corresponding source files. *See Chapter 6/5*

faldump is a serializer for the FALSE programming language. It dumps the complete internal representation of programs written in FALSE into an XML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 6*

falm68k is a compiler for the FALSE programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/13/22*

falmibl is a compiler for the FALSE programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/14/22*

falmips32 is a compiler for the FALSE programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/15/22*

falmips64 is a compiler for the FALSE programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/15/22*

falmmix is a compiler for the FALSE programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/16/22*

falor1k is a compiler for the FALSE programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/17/22*

falppc32 is a compiler for the FALSE programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC

processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. *See Chapter 6/18/22*

falppc64 is a compiler for the FALSE programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. *See Chapter 6/18/22*

falprint is a pretty printer for the FALSE programming language. It reformats the source code of FALSE programs and writes it to the standard output stream. *See Chapter 6*

falrisc is a compiler for the FALSE programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/19/22*

falrun is an interpreter for the FALSE programming language. It processes and executes programs written in FALSE. *See Chapter 6*

falwasm is a compiler for the FALSE programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/20/22*

falxtensa is a compiler for the FALSE programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in FALSE and stores it in corresponding object files. *See Chapter 6/21/22*

linkbin is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`. *See Chapter 22*

linkhex is a linker for Intel HEX files. It links all code sections of the object files given to it into single image and stores the image in an Intel HEX file [3] that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. *See Chapter 22*

linklib is an object file combiner. It creates a static library file by combining all object files given to it into a single one. *See Chapter 22*

linkmem is a linker for plain binary files partitioned into random-access and read-only memory. It links all object files given to it into two distinct images, one for data sections and one for code and constant data sections, and stores each image in a binary file that begins with the first linked section of the corresponding type. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. *See Chapter 22*

linkprg is a linker for GEMDOS executable files. It links all object files given to it into a single image and stores the image in an Atari GEMDOS executable file [4]. It also creates a map file that lists the address relative to the text segment, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting executable file can be specified by putting it into a constant data section called `_extension`. The GEMDOS executable file format requires all patch patterns of absolute link patches to consist of four full bitmasks with descending offsets. *See Chapter 22*

m68kasm is an assembler for the M68000 hardware architecture. It translates assembly code into machine code for M68000 processors and stores it in corresponding object files. *See Chapter 8/13/22*

m68kdism is a disassembler for the M68000 hardware architecture. It translates machine code from object files targeting M68000 processors into assembly code and writes it to the standard output stream. *See Chapter 8/13/22*

mapplace is a map tool provided for debugging purposes. It converts map files generated by linker tools into an object file called map place result by placing all listed binary sections at their corresponding address using fixed phantom sections. Map place results can be linked together with debugging object files to generate separate symbol files for external debuggers. *See Chapter 22/24*

mapsearch is map tool provided for debugging purposes. It searches map files generated by linker tools for the name of a binary section that encompasses a memory address read from the standard input stream. If additionally provided with one or more object files, it also stores an excerpt thereof in a separate object file called map search result which only contains the identified binary section for disassembling purposes. *See Chapter 22/24*

miblasm is an assembler for the MicroBlaze hardware architecture. It translates assembly code into machine code for MicroBlaze processors and stores it in corresponding object files. *See Chapter 8/14/22*

mibldism is a disassembler for the MicroBlaze hardware architecture. It translates machine code from object files targeting MicroBlaze processors into assembly code and writes it to the standard output stream. *See Chapter 8/14/22*

mips32asm is an assembler for the MIPS32 hardware architecture. It translates assembly code into machine code for MIPS32 processors and stores it in corresponding object files. *See Chapter 8/15/22*

mips32dism is a disassembler for the MIPS32 hardware architecture. It translates machine code from object files targeting MIPS32 processors into assembly code and writes it to the standard output stream. *See Chapter 8/15/22*

mips64asm is an assembler for the MIPS64 hardware architecture. It translates assembly code into machine code for MIPS64 processors and stores it in corresponding object files. *See Chapter 8/15/22*

mips64dism is a disassembler for the MIPS64 hardware architecture. It translates machine code from object files targeting MIPS64 processors into assembly code and writes it to the standard output stream. *See Chapter 8/15/22*

mmixasm is an assembler for the MMIX hardware architecture. It translates assembly code into machine code for MMIX processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number. *See Chapter 8/16/22*

mmixdism is a disassembler for the MMIX hardware architecture. It translates machine code from object files targeting MMIX processors into assembly code and writes it to the standard output stream. *See Chapter 8/16/22*

obamd16 is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd16run` library file. *See Chapter 7/8/9/22/24*

obamd32 is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating

mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd32run` library file. *See Chapter 7/8/9/22/24*

obamd64 is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd64run` library file. *See Chapter 7/8/9/22/24*

obarma32 is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarma32run` library file. *See Chapter 7/8/10/22/24*

obarma64 is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarma64run` library file. *See Chapter 7/8/10/22/24*

obarmt32 is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarmt32run` library file. *See Chapter 7/8/10/22/24*

obarmt32fpe is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarmt32fperun` library file. *See Chapter 7/8/10/22/24*

obavr is a compiler for the Oberon programming language targeting the AVR hardware architecture.

It generates machine code for AVR processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavrrun` library file. *See Chapter 7/8/11/22/24*

obavrxt is a compiler for the Oberon programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavrxtrun` library file. *See Chapter 7/8/11/22/24*

obavr32 is a compiler for the Oberon programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavr32run` library file. *See Chapter 7/8/12/22/24*

obcheck is a syntactic and semantic checker for the Oberon programming language. It just performs syntactic and semantic checks on Oberon modules and writes its diagnostic messages to the standard error stream. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. *See Chapter 7*

obcode is an intermediate code generator for the Oberon programming language. It generates intermediate code from modules written in Oberon and stores it in corresponding assembly files. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this tool require additional runtime support that is stored in the `obcoderun` assembly file. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 7/8/23*

obcpp is a transpiler for the Oberon programming language. It translates programs written in Oberon into C++ programs and stores them in corresponding source and header files. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. The same interface is provided by the generated header file which can be used in other parts of the C++ program. *See Chapter 7/5*

obdoc is a generic documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into a generic documentation. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 7/25*

obdump is a serializer for the Oberon programming language. It dumps the complete internal representation of modules written in Oberon into an XML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible. *See Chapter 7*

obhtml is an HTML documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into an HTML document. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. *See Chapter 7/25*

oblatex is a Latex documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into a Latex document. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. *See Chapter 7/25*

obm68k is a compiler for the Oberon programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obm68krun` library file. *See Chapter 7/8/13/22/24*

obmibl is a compiler for the Oberon programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmiblrun` library file. *See Chapter 7/8/14/22/24*

obmips32 is a compiler for the Oberon programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmips32run` library file. *See Chapter 7/8/15/22/24*

obmips64 is a compiler for the Oberon programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmips64run` library file. *See Chapter 7/8/15/22/24*

obmmix is a compiler for the Oberon programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmmixrun` library file. *See Chapter 7/8/16/22/24*

obor1k is a compiler for the Oberon programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for

OpenRISC 1000 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obor1krun` library file. *See Chapter 7/8/17/22/24*

obppc32 is a compiler for the Oberon programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obppc32run` library file. *See Chapter 7/8/18/22/24*

obppc64 is a compiler for the Oberon programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obppc64run` library file. *See Chapter 7/8/18/22/24*

obprint is a pretty printer for the Oberon programming language. It reformats the source code of Oberon modules and writes it to the standard output stream. *See Chapter 7*

obrisc is a compiler for the Oberon programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obriscrun` library file. *See Chapter 7/8/19/22/24*

obrun is an interpreter for the Oberon programming language. It processes and executes modules written in Oberon. This tool does neither generate nor process symbol files while interpreting modules. If a module is imported by another one, its filename has to be named before the other one in the list of command-line arguments. *See Chapter 7*

obwasm is a compiler for the Oberon programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obwasmrun` library file. *See Chapter 7/8/20/22/24*

obxtensa is a compiler for the Oberon programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from modules written in Oberon and stores

it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obxtensarun` library file. *See Chapter 7/8/21/22/24*

or1kasm is an assembler for the OpenRISC 1000 hardware architecture. It translates assembly code into machine code for OpenRISC 1000 processors and stores it in corresponding object files. *See Chapter 8/17/22*

or1kdism is a disassembler for the OpenRISC 1000 hardware architecture. It translates machine code from object files targeting OpenRISC 1000 processors into assembly code and writes it to the standard output stream. *See Chapter 8/17/22*

ppc32asm is an assembler for the PowerPC hardware architecture. It translates assembly code into machine code for PowerPC processors and stores it in corresponding object files. By default, the assembler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. *See Chapter 8/18/22*

ppc32dism is a disassembler for the PowerPC hardware architecture. It translates machine code from object files targeting PowerPC processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 32-bit operating mode defined by the PowerPC architecture. *See Chapter 8/18/22*

ppc64asm is an assembler for the PowerPC hardware architecture. It translates assembly code into machine code for PowerPC processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. *See Chapter 8/18/22*

ppc64dism is a disassembler for the PowerPC hardware architecture. It translates machine code from object files targeting PowerPC processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the PowerPC architecture. *See Chapter 8/18/22*

riscasm is an assembler for the RISC hardware architecture. It translates assembly code into machine code for RISC processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number. *See Chapter 8/19/22*

riscdism is a disassembler for the RISC hardware architecture. It translates machine code from object files targeting RISC processors into assembly code and writes it to the standard output stream. *See Chapter 8/19/22*

wasmasm is an assembler for the WebAssembly architecture. It translates assembly code into machine code for WebAssembly targets and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number. *See Chapter 8/20/22*

wasmdism is a disassembler for the WebAssembly architecture. It translates machine code from object files targeting WebAssembly targets into assembly code and writes it to the standard output stream. *See Chapter 8/20/22*

xtensaasm is an assembler for the Xtensa hardware architecture. It translates assembly code into machine code for Xtensa processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number. *See Chapter 8/21/22*

xtensadism is a disassembler for the Xtensa hardware architecture. It translates machine code from object files targeting Xtensa processors into assembly code and writes it to the standard output stream. *See Chapter 8/21/22*

Table 4.1 lists all 82 compiler and assembler tools and shows their naming scheme consisting of common programming language prefixes and hardware architecture suffixes.

Hardware Architecture \ Programming Language		C++ *Chapter 5*	FALSE *Chapter 6*	Oberon *Chapter 7*	Generic Assembly *Chapter 8*
AMD64	16-bit	`cppamd16` *5.6.10*	`falamd16` *6.3.7*	`obamd16` *7.7.10*	`amd16asm` *8.6.2*
	32-bit	`cppamd32` *5.6.11*	`falamd32` *6.3.8*	`obamd32` *7.7.11*	`amd32asm` *8.6.3*
	64-bit	`cppamd64` *5.6.12*	`falamd64` *6.3.9*	`obamd64` *7.7.12*	`amd64asm` *8.6.4*
See Chapter 9					
ARM	A32	`cpparma32` *5.6.13*	`falarma32` *6.3.10*	`obarma32` *7.7.13*	`arma32asm` *8.6.5*
	A64	`cpparma64` *5.6.14*	`falarma64` *6.3.11*	`obarma64` *7.7.14*	`arma64asm` *8.6.6*
	T32	`cpparmt32` *5.6.15*	`falarmt32` *6.3.12*	`obarmt32` *7.7.15*	`armt32asm` *8.6.7*
		`cpparmt32fpe`	`falarmt32fpe`	`obarmt32fpe`	
See Chapter 10		*5.6.16*	*6.3.13*	*7.7.16*	
AVR		`cppavr` *5.6.17*	`falavr` *6.3.14*	`obavr` *7.7.17*	`avrasm` *8.6.8*
		`cppavrxt`	`falavrxt`	`obavrxt`	
See Chapter 11		*5.6.18*	*6.3.15*	*7.7.18*	
AVR32		`cppavr32`	`falavr32`	`obavr32`	`avr32asm`
See Chapter 12		*5.6.19*	*6.3.16*	*7.7.19*	*8.6.9*
M68000		`cppm68k`	`falm68k`	`obm68k`	`m68kasm`
See Chapter 13		*5.6.20*	*6.3.17*	*7.7.20*	*8.6.10*
MicroBlaze		`cppmibl`	`falmibl`	`obmibl`	`miblasm`
See Chapter 14		*5.6.21*	*6.3.18*	*7.7.21*	*8.6.11*
MIPS	32-bit	`cppmips32` *5.6.22*	`falmips32` *6.3.19*	`obmips32` *7.7.22*	`mips32asm` *8.6.12*
	64-bit	`cppmips64`	`falmips64`	`obmips64`	`mips64asm`
See Chapter 15		*5.6.23*	*6.3.20*	*7.7.23*	*8.6.13*
MMIX		`cppmmix`	`falmmix`	`obmmix`	`mmixasm`
See Chapter 16		*5.6.24*	*6.3.21*	*7.7.24*	*8.6.14*
OpenRISC 1000		`cppor1k`	`falor1k`	`obor1k`	`or1kasm`
See Chapter 17		*5.6.25*	*6.3.22*	*7.7.25*	*8.6.15*
PowerPC	32-bit	`cppppc32` *5.6.26*	`falppc32` *6.3.23*	`obppc32` *7.7.26*	`ppc32asm` *8.6.16*
	64-bit	`cppppc64`	`falppc64`	`obppc64`	`ppc64asm`
See Chapter 18		*5.6.27*	*6.3.24*	*7.7.27*	*8.6.17*
RISC		`cpprisc`	`falrisc`	`obrisc`	`riscasm`
See Chapter 19		*5.6.28*	*6.3.25*	*7.7.28*	*8.6.18*
WebAssembly		`cppwasm`	`falwasm`	`obwasm`	`wasmasm`
See Chapter 20		*5.6.29*	*6.3.26*	*7.7.29*	*8.6.19*
Xtensa		`cppxtensa`	`falxtensa`	`obxtensa`	`xtensaasm`
See Chapter 21		*5.6.30*	*6.3.27*	*7.7.30*	*8.6.20*

Table 4.1: References to all 82 compiler and assembler tools

Part II

Supported Programming Languages

5 C++

C++ is a general-purpose programming language with a bias toward systems programming. It is based on the programming language C and enhances it by supporting data abstraction, object-oriented programming, and generic programming. This chapter describes the implementation of C++ by the Eigen Compiler Suite.

Greatness lies not in being strong,
but in the using of strength.

— Henry Ward Beecher

5.1 Introduction

The Eigen Compiler Suite implements the C++ programming language according to the ISO C++ Standard ISO/IEC 14882:2023 [5]. C++ is a general-purpose programming language based on the C programming language as described in the C Standard ISO/IEC 9899:2018 [6].

The remainder of this chapter describes the implementation-defined behavior of the implementation by the Eigen Compiler Suite as required by both of these international standards.

5.2 Implementation-Defined Behavior

The C++ Standard requires all conforming implementations to include documentation describing its characteristics and behavior in areas designated as implementation-defined. This section lists all implementation-defined behavior including conditionally-supported constructs and locale-specific characteristics along with the corresponding section and paragraph numbers of the C++ Standard.

5.2.1 Terms and Definitions 3 [intro.defs]

- Diagnostic messages 3.8 [defns.diagnostic] 1

  The set of diagnostic messages consist of all output messages issued by the tools of the Eigen Compiler Suite. Diagnostic messages are either errors, warnings, or notes and are meant to be self-explanatory in the context they occur: Errors indicate a violation of a diagnosable rule or an occurrence of a unsupported construct described in the C++ Standard as conditionally-supported. Fatal errors flag internal program errors or environmental problems like failures to open source files or critical system conditions like out of memory. Warning messages identify issues that may lead to unexpected behavior, whereas notes diagnose violations of common coding conventions.

5.2.2 General Principles 4 [intro]

- Number of contiguous bits in a byte 4.4 [intro.memory] 1

  The Eigen Compiler Suite uniformly uses octets as representation for bytes.

- Interactive devices 4.6 [intro.execution] 7

  A device is considered to be interactive when it has to wait for input from the user or some other entity in order to proceed the execution of a program.

- Multiple threads of execution 4.7 [intro.multithread] 1

  A program can have more than one thread of execution in a freestanding environment.

5.2.3 Lexical Conventions 5 [lex]

- Mapping of source file characters to the basic source character set 5.2 [lex.phases] 1

  The Eigen Compiler Suite assumes source files to be written using the ASCII character set. This set allows mapping most of the characters to the basic source character set. Any other character not in the basic source character set is replaced by the universal character name that designates that character.

- Nonempty sequences of white-space characters 5.2 [lex.phases] 1

  Nonempty sequences of white-space characters are replaced by one space character.

- Conversion of source characters to execution characters 5.2 [lex.phases] 1

  All members of the source character set have a corresponding member in the execution character set.

- Source of template definitions 5.2 [lex.phases] 1

  Instantiations of templates require the source of the translation units containing their definitions.

- Appearance of special characters in header names 5.8 [lex.header] 2

  Quotation marks, backslashes, and the character sequences `/*` or `//` may appear in header names but bear no special meaning and are therefore treated like any other character.

- Character literals containing multiple characters 5.13.3 [lex.ccon] 2/6

  Ordinary and wide character literals may contain more than one character. However, all but the first character are ignored when determining the value of these literals.

- Additional escape sequences 5.13.3 [lex.ccon] 7

  The Eigen Compiler Suite does only support the standard escape sequences.

- Character values outside valid range 5.13.3 [lex.ccon] 8/9

  The value of an escape sequence or a universal character name in a character literal must be representable using the underlying type of the literal.

- String literal concatenations 5.13.5 [lex.string] 13

  Concatenations of adjacent string literals with different encoding prefixes are not supported.

5.2.4 Basic Concepts 6 [basic]

- Definition of functions or variables with origin 6.2 [basic.def.odr] 4

  A non-inline function or variable does not require a definition if it is declared using the origin attribute, see Section 5.2.7.

- Definition of `main` in freestanding environments 6.6.1 [basic.start.main] 1

  A program in a freestanding environment is required to define a `main` function.

- Linkage of `main` 6.6.1 [basic.start.main] 3

  The `main` function has external linkage.

- Use of invalid pointer values 6.7 [basic.stc] 4

  There are no restrictions for using an invalid pointer value except for passing it to a deallocation function and accessing the referenced region of storage.

- Pointer safety 6.7.4.3 [basic.stc.dynamic.safety] 4

  The Eigen Compiler Suite has relaxed pointer safety. A pointer value is valid regardless of whether it is safely derived or not.

- Extended integer types 6.9.1 [basic.fundamental] 2

  The Eigen Compiler Suite does not provide any extended integer types.

- Fundamental alignments 6.11 [basic.align] 2

  The alignment of a fundamental type depends on the size of the type as well as the execution environment. Table 5.1 lists the alignment of fundamental types for each C++ compiler and type size supported by the Eigen Compiler Suite. The sizes of fundamental types are shown in Table 5.2 and Table 5.3. The interpreter and all other tools reuse the respective fundamental alignments of their own execution environment instead.

- Extended alignments 6.11 [basic.align] 3

  The Eigen Compiler Suite supports extended alignments for variables with static storage duration. The set of valid alignments consists of all non-negative integral powers of two representable in the type `std::size_t`.

5.2.5 Standard Conversions 7 [conv]

- Ranks of extended signed integer types 7.15 [conv.rank] 1

  The Eigen Compiler Suite does not provide any extended integer types.

5.2.6 Expressions 8 [expr]

- Sizes of fundamental types 8.3.3 [expr.sizeof] 1

  Table 5.2 lists the size and value range of each fundamental type as defined by the Eigen Compiler Suite. The sizes of the types `int`, `unsigned int`, and `double` depend on the execution environment and are listed in Table 5.3 for each C++ compiler provided by the Eigen Compiler Suite. The interpreter and all other tools reuse the respective type sizes of their own execution environment instead.

- Resulting type of pointer subtractions 8.7 [expr.add] 5

  The type of the result of subtracting two pointer values is the signed counterpart of `std::size_t`.

Hardware Architecture		C++ Compiler	Type Size 1	2	4	8
AMD64	16-bit	`cppamd16`	1	2	4	4
	32-bit	`cppamd32`	1	2	4	4
	64-bit	`cppamd64`	1	2	4	8
ARM	A32	`cpparma32`	1	2	4	8
	A64	`cpparma64`	1	2	4	8
	T32	`cpparmt32`	1	2	4	8
		`cpparmt32fpe`	1	2	4	8
AVR		`cppavr`	1	1	1	1
		`cppavrxt`	1	1	1	1
AVR32		`cppavr32`	1	2	4	8
M68000		`cppm68k`	1	2	2	2
MicroBlaze		`cppmibl`	1	2	4	8
MIPS	32-bit	`cppmips32`	1	2	4	8
	64-bit	`cppmips64`	1	2	4	8
MMIX		`cppmmix`	1	2	4	8
OpenRISC 1000		`cppor1k`	1	2	4	4
PowerPC	32-bit	`cppppc32`	1	2	4	8
	64-bit	`cppppc64`	1	2	4	8
RISC		`cpprisc`	1	2	4	4
WebAssembly		`cppwasm`	1	2	4	8
Xtensa		`cpprisc`	1	2	4	4

Table 5.1: Alignments of fundamental C++ types

Category	Type	Size	Value Range
Boolean	`bool`	1	`false` or `true`
Character	`char`	1	0 to 255
	`signed char`	1	-128 to $+127$
	`unsigned char`	1	0 to 255
	`char16_t`	2	0 to 65 535
	`char32_t`	4	0 to 4 294 967 295
	`wchar_t`	4	0 to 4 294 967 295
Integer	`short int`	2	-2^{15} to $+2^{15}-1$
	`unsigned short int`	2	0 to $2^{16}-1$
	`int`	2/4	*See Table 5.3*
	`unsigned int`	2/4	*See Table 5.3*
	`long int`	4	-2^{31} to $+2^{31}-1$
	`unsigned long int`	4	0 to $2^{32}-1$
	`long long int`	8	-2^{63} to $+2^{63}-1$
	`unsigned long long int`	8	0 to $2^{64}-1$
Floating-Point	`float`	4	$\pm 3.4028234 \times 10^{38}$
	`double`	4/8	*See Table 5.3*
	`long double`	8	$\pm 1.7976931348623157 \times 10^{308}$

Table 5.2: Sizes and value ranges of fundamental C++ types

5.2.7 Declarations — 10 [dcl.dcl]

- Meaning of attribute declarations — 10 [dcl.dcl] 3

  Attribute declarations appertain to the current translation unit and may contain the standard `deprecated` attribute which applies to included headers and source files.

- The `asm` declaration — 10.4 [dcl.asm] 1

  The `asm` declaration is supported and allows writing inline assembly code using one of the various compilers for the C++ programming language. All of them pass the string literal of the `asm` declaration to the assembler used to generate the machine code. The available instruction set therefore depends on the actual compiler used to compile the source code. For more information about the generic assembly language and how to use it, see Chapter 8. The interpreter does not support executing `asm` declarations.

  The assembly code of an `asm` declaration appearing at namespace scope has unrestricted access to all features of the assembler. It must therefore first create a code or data section before any other directive or instruction can be used. At block scope, the inline assembly code is part of a code section predefined by the compiler which also provides constant

Hardware Architecture		C++ Compiler	(unsigned) int	double	pointer types std::size_t	function pointers
AMD64	16-bit	cppamd16	2	8	2	2
	32-bit	cppamd32	4	8	4	4
	64-bit	cppamd64	4	8	8	8
ARM	A32	cpparma32	4	8	4	4
	A64	cpparma64	4	8	8	8
	T32	cpparmt32	4	4	4	4
		cpparmt32fpe	4	8	4	4
AVR		cppavr	2	4	2	2
		cppavrxt	2	4	2	3
AVR32		cppavr32	4	4	4	4
M68000		cppm68k	2	4	4	4
MicroBlaze		cppmibl	4	4	4	4
MIPS	32-bit	cppmips32	4	8	4	4
	64-bit	cppmips64	4	8	8	8
MMIX		cppmmix	4	8	8	8
OpenRISC 1000		cppor1k	4	4	4	4
PowerPC	32-bit	cppppc32	4	4	4	4
	64-bit	cppppc64	4	4	4	4
RISC		cpprisc	4	4	4	4
WebAssembly		cppwasm	4	8	4	4
Xtensa		cppwasm	4	4	4	4

Table 5.3: Sizes of hardware-dependent C++ types

definitions for all labels, variables, parameters, enumerations and their enumerators that are accessible from that scope. The name of a label is predefined as the address of the corresponding labeled statement and can therefore be used as the target of branch instructions. The names of all parameters and variables with automatic storage duration evaluate to the offset of the corresponding entity relative to the frame pointer. The name of a variable with static storage duration evaluates to its address. If a variable or parameter is declared using the register attribute, its name is an alias for the corresponding processor register. The name of an enumerator or a non-volatile constant variable initialized with a constant expression of fundamental or pointer type is predefined to its value. For debugging purposes, all names predefined in inline assembly code and their actual values are accessible using the expression evaluation directive.

- Language linkages 10.5 [dcl.link] 2

  Besides the two language linkages `"C"` and `"C++"` required by the C++ Standard, the Eigen Compiler Suite also supports the `"Oberon"` language linkage. It allows accessing global procedures and variables defined in Oberon modules. In order to identify the containing module and its package, the corresponding functions and objects declared with this language linkage must be members of named namespaces. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

- Non-standard attributes 10.6.1 [dcl.attr.grammar] 6

  Besides the attributes specified in the C++ Standard, the Eigen Compiler Suite also recognizes the following non-standard attributes. All of them may only appear once in an attribute list and cannot be used as pack expansions. Except where otherwise noted, they apply only to functions and variables with static storage duration and do not accept arguments:

  – Alias attribute `ecs::alias (` *string-literal* `)`

    The alias attribute specifies that the corresponding entity may also be accessible in assembly code or other programming languages using the name given in the non-empty string literal.

  – Code attribute `ecs::code`

    The code attribute applies to `asm` declarations and causes their string literal to be interpreted as intermediate code. For more information about intermediate code and its purpose, see Chapter 23.

  – Default attribute `ecs::default`

    The default attribute specifies that linkers replace the definition of the corresponding entity if there is another entity with the same name.

– Group attribute `ecs::group` (*string-literal*)

The group attribute specifies that linkers place the corresponding entity adjacent to other entities that also belong to the group named in the non-empty string literal.

– Origin attribute `ecs::origin` (*constant-expression*)

The origin attribute specifies the desired address of the variable or function. It requires an integral constant expression as argument that is convertible to `std::size_t`. This attribute may not be combined with alignment specifiers.

– Phantom attribute `ecs::phantom`

The phantom attribute specifies that linkers omit the definition of the corresponding entity even if is used.

– Pure attribute `ecs::pure`

The pure attribute applies to non-void functions and specifies that the function does not have any side effects. A pure function can modify only local variables and call only functions without side effects.

– Register attribute `ecs::register`

The register attribute applies to up to four non-volatile parameters and variables with automatic storage duration of fundamental or pointer type and specifies that compilers store the corresponding variable or parameter in a register.

– Required attribute `ecs::required`

The required attribute specifies that linkers use the definition of the corresponding entity even if it is unreferenced.

– Shared attribute `ecs::shared`

The shared attribute specifies that linkers merge the definition of the corresponding entity with other entities that have the same definition.

Any entity declared with one of these attributes can later be redeclared without that attribute and vice versa. However, redeclarations and declarations in other translation units using different forms of the same attribute are not allowed.

- Noreturn `asm` declarations 9.12.10 [dcl.attr.noreturn] 1

  The noreturn attribute may be applied to `asm` declarations at block scope and specifies that the corresponding assembly code does not return.

5.2.8 Declarators 11 [dcl.decl]

- String value of `__func__` 11.4.1 [dcl.fct.def.general] 8

  The string resulting from the function-local predefined variable `__func__` matches the fully qualified section name of the function as described in Section 5.7.1.

5.2.9 Classes 12 [class]

- Allocation and alignment of bit-fields 12.2.4 [class.bit] 1

  Bit-fields are allocated consecutively and packed adjacent to each other, as long as they do not straddle allocation units and the alignment of their respective types does not require padding bits.

5.2.10 Preprocessing Directives 19 [cpp]

- Additional preprocessing directives 19 [cpp] 2

  The Eigen Compiler Suite does not support additional preprocessing directives.

- Interpretation of character literals 19.1 [cpp.cond] 10

  The numeric value for character literals within a `#if` or `#elif` directive is non-negative and matches the value obtained when an identical character literal occurs in an expression.

- Source file inclusion 19.2 [cpp.include] 2/3

  Source files identified between a pair of " delimiters in `#include` directives are searched relative to the directory of the current source file. Headers identified between the < and > delimiters are searched in the relative directory given by the `ECSINCLUDE` environment variable. Two or more directories can be separated by semicolons and are searched in order of occurrence.

- Combination of preprocessing tokens into one header name token 19.2 [cpp.include] 4

  A pair of " delimiters is treated as a string literal token. All preprocessing tokens between < and > delimiters are concatenated to form a single header name token. Identifiers therein are subject to macro replacement.

- Pragma directive 19.6 [cpp.pragma] 1

  The Eigen Compiler Suite recognizes the `#pragma end` pragma directive which simulates the end of a source file. The remaining part of the source file following this pragma directive is completely ignored. All other pragma directives are ignored.

- Predefined macro names 19.8 [cpp.predefined] 1

  If the date of translation is not available, the macros `__DATE__` and `__TIME__` are predefined as `"May 10 2023"` and `"00:00:00"` respectively. The macro `__STD_HOSTED__` is defined as `1` since the C++ implementation of the Eigen Compiler Suite is hosted.

- Conditionally-defined macro names 16.8 [cpp.predefined] 2

  The macros `__STDC__` and `__STDC_VERSION__` are not predefined.

- Implementation-defined macro names 16.8 [cpp.predefined] 4

  The macro `__ecs__` is predefined in order to enable programmers to detect the Eigen Compiler Suite while processing compiler-specific source code. The value of the predefined macro `__sizeof_`*type*`__` where *type* is either `double`, `float`, `int`, `long`, `long_double`, `pointer`, or `short` is the size of the corresponding type. The interpreter and all compilers predefine an additional macro name of the form `__`*target*`__` for identifying the target environment where *target* is either `amd16`, `amd32`, `amd64`, `arma32`, `arma64`, `armt32`, `armt32fpe`, `avr`, `avr32`, `code`, `m68k`, `mibl`, `mips32`, `mips64`, `mmix`, `or1k`, `ppc32`, `ppc64`, `risc`, `run`, `wasm`, or `xtensa`. Further macros are predefined for all valid identifiers separated by semicolons in the `ECSDEFINE` environment variable.

5.2.11 Library Introduction 20 [library]

- Declaration of additional functions from the standard C library 20.5.1.2 [headers] 9

  The functions described in Annex K of the C standard are not declared when including C++ headers.

- Freestanding implementations 20.5.1.3 [compliance] 2

  The Eigen Compiler Suite provides an implementation of C++ that can be used in hosted as well as freestanding environments. In both cases, the complete set of headers described by the C++ Standard is available.

- Linkage of names from the standard C library 20.5.2.3 [using.linkage] 2

  Names from the C standard library declared with external linkage have `extern "C"` linkage.

5.2.12 Language Support Library 21 [language.support]

- Definition of macro `NULL` 21.2.3 [support.types.nullptr] 2

  The macro `NULL` is defined as `nullptr`.

5.2.13 Implementation Quantities B [implimits]

According the C++ Standard, each implementation shall document the limitations of the programs they can successfully process. This section lists all known limitations of the Eigen Compiler Suite:

- Size of an object 6.9.2 [basic.compound] 2

  The size of an object is limited by the maximal value representable in the type `std::size_t`.

- Nesting levels for `#include` files 19.2 [cpp.include] 6

  The nesting level of `#include` directives has a limit of 256 in order to detect potentially infinite recursions.

- Functions registered by `atexit()` 21.5 [support.start.term] 6

  The Eigen Compiler Suite supports the registration of at most 32 functions.

- Functions registered by `at_quick_exit()` 21.5 [support.start.term] 10

  The Eigen Compiler Suite supports the registration of at most 32 functions.

- Recursive `constexpr` function invocations 8.20 [expr.const] 2

  The total number of nested calls of `constexpr` functions is limited to 512.

- Recursively nested template instantiations 17.7.1 [temp.inst] 15

  The total depth of recursive template instantiations, including substitution during template argument deduction, has a limit of 1024 in order to detect potentially infinite recursions.

- Number of placeholders 23.14.11.4 [func.bind.place] 1

  The implementation-defined number of placeholders *M* is 10.

All other quantities listed in Annex B of the C++ Standard that are not mentioned above have no intrinsic limit and are only restricted by available memory. The actual limit depends therefore only on the execution environment of the tool used to process the C++ source file.

5.3 The Standard C++ Library

The Eigen Compiler Suite provides its own implementation of the C++ standard library called the *Standard C++ Library*. Its facilities are made available by including one or more of its headers and providing the necessary runtime support as described in Section 5.5. All of these headers are governed by the ECS Runtime Support Exception which is an additional permission to the GNU General Public License that allows users of the Eigen Compiler Suite to create proprietary programs. Copies of these licenses are included in Appendices C and D on pages 495 and 509 respectively.

5.4 Documentation Generation

The Eigen Compiler Suite provides several tools that are able to extract the structure of programs written in C++ and generate documentations for them. This section describes the contents of the extracted information and explains how programmers can provide a user-defined description of it.

5.5 Runtime Support

Some language features of C++ such as exceptions require some additional runtime support. This runtime support is stored in library files which are combinations of object files. For more information about object files and their purpose, see Chapter 22. The Eigen Compiler Suite provides the required runtime support in one library file for each hardware architecture it supports. The name of the corresponding library file consists of a leading `cpp`, the name of the target hardware architecture, and a trailing `run` as in `cppamd64run`.

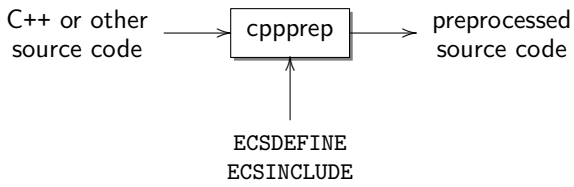
5.6 C++ Tools

The Eigen Compiler Suite provides several different tools that process source files written in C++. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

The tools process C++ translation units in several consecutive stages. In each stage, the internal representation of the translation unit is changed and transformed. Figure 5.1 shows all stages and the different representations.

5.6.1 cppprep

`cppprep` is a preprocessor for the C++ programming language. It preprocesses source code according to the rules of C++ and writes it to the standard output stream. Only the macro names `__DATE__`, `__FILE__`, `__LINE__`, and `__TIME__` are predefined.



5.6.2 cppprint

`cppprint` is a pretty printer for the C++ programming language. It reformats the source code of C++ programs and writes it to the standard output stream.

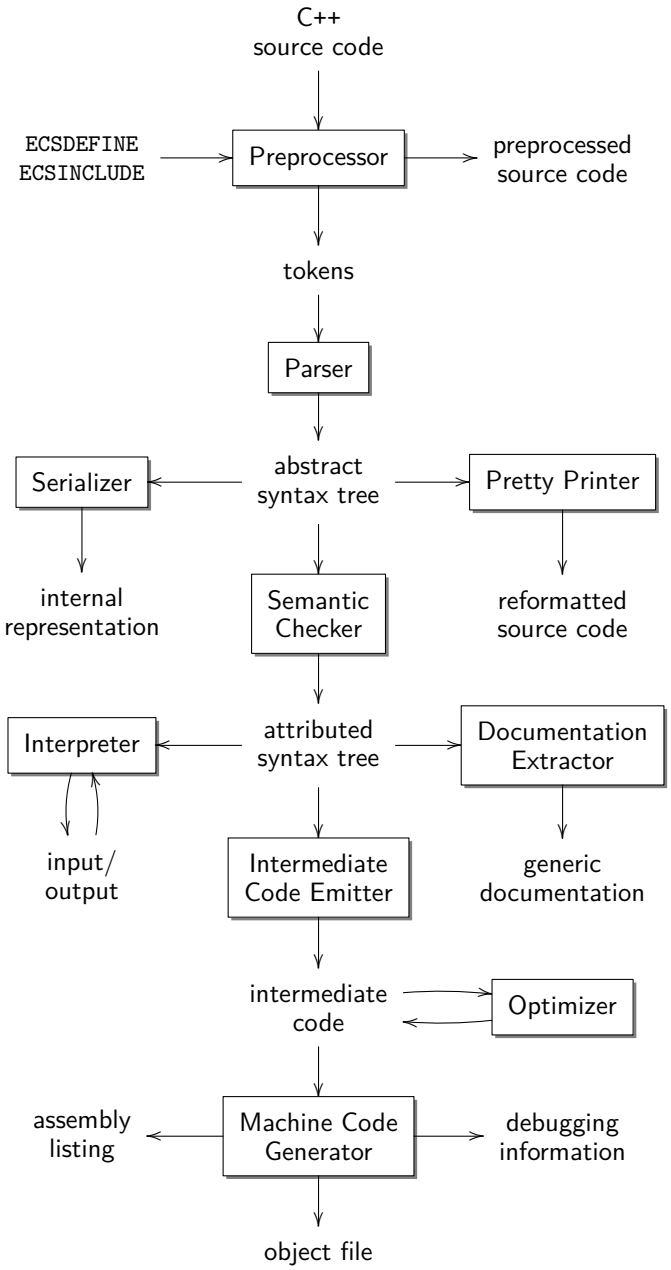
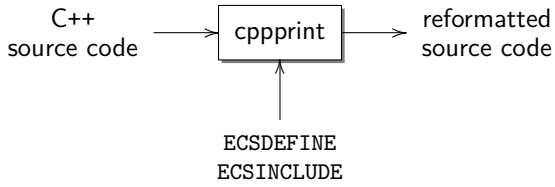
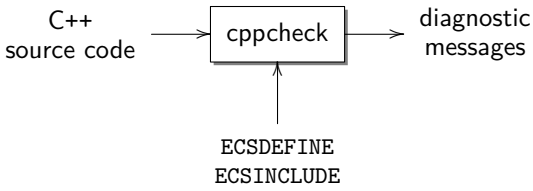


Figure 5.1: Data flow within the tools for C++



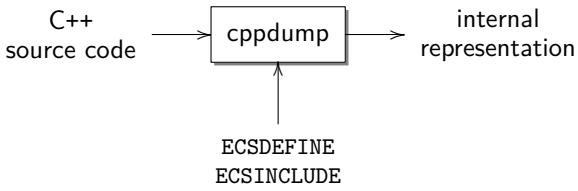
5.6.3 cppcheck

`cppcheck` is a syntactic and semantic checker for the C++ programming language. It just performs syntactic and semantic checks on C++ programs and writes its diagnostic messages to the standard error stream.



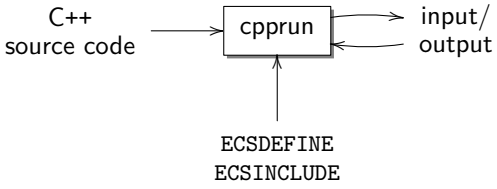
5.6.4 cppdump

`cppdump` is a serializer for the C++ programming language. It dumps the complete internal representation of programs written in C++ into an XML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



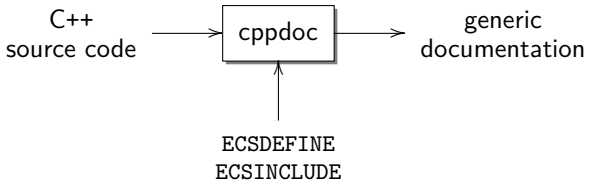
5.6.5 cpprun

`cpprun` is an interpreter for the C++ programming language. It processes and executes programs written in C++. The macro `__run__` is predefined in order to enable programmers to identify this tool while interpreting.



5.6.6 cppdoc

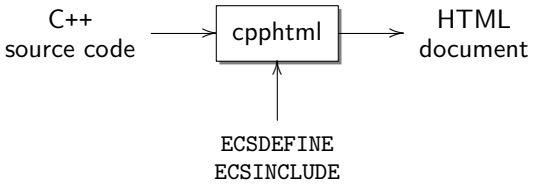
`cppdoc` is a generic documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into a generic documentation. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

5.6.7 cpphtml

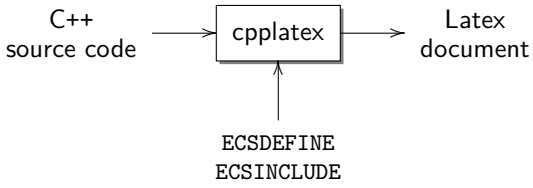
`cpphtml` is an HTML documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into an HTML document.



For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

5.6.8 cpplatex

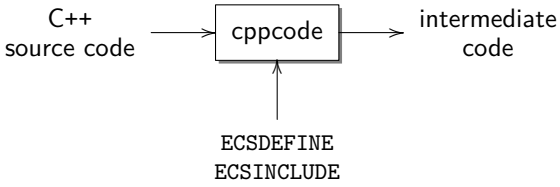
`cpplatex` is a Latex documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into a Latex document.



For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

5.6.9 cppcode

`cppcode` is an intermediate code generator for the C++ programming language. It generates intermediate code from programs written in C++ and stores it in corresponding assembly files. The macro `__code__` is predefined in order to enable programmers to identify this tool while generating intermediate code. Programs generated with this tool require additional runtime support that is stored in the `cppcoderun` assembly file. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.

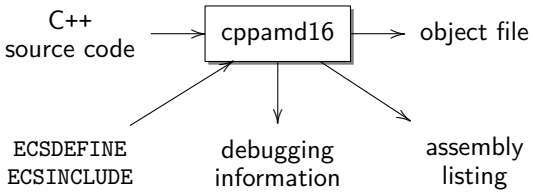


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about intermediate code and its purpose, see Chapter 23.

5.6.10 cppamd16

`cppamd16` is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated

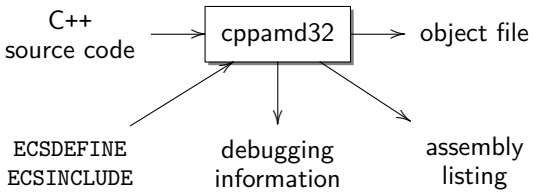
machine code. The macro `__amd16__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd16run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.11 cppamd32

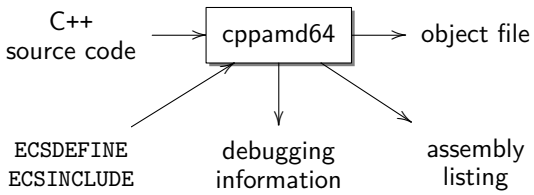
`cppamd32` is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.12 cppamd64

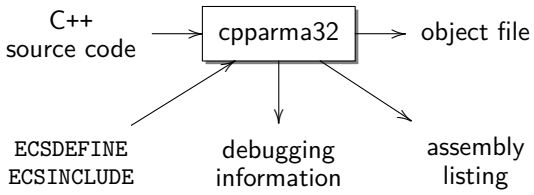
`cppamd64` is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.13 cpparma32

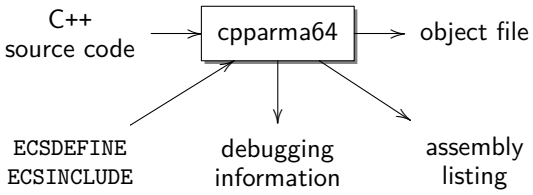
`cpparma32` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__arma32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparma32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.14 cpparma64

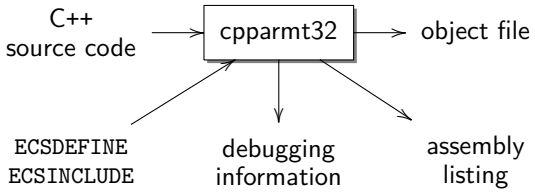
`cpparma64` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__arma64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparma64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.15 cpparmt32

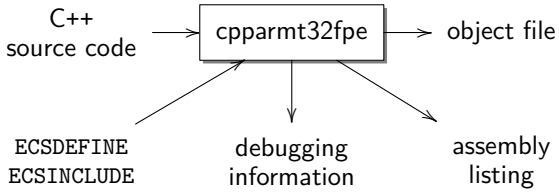
`cpparmt32` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__armt32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparmt32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.16 cpparmt32fpe

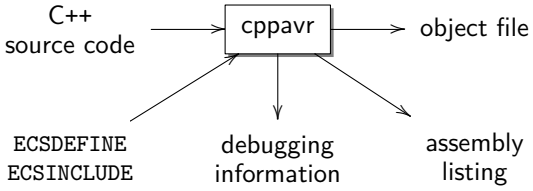
`cpparmt32fpe` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__armt32fpe__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparmt32fperun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.17 cppavr

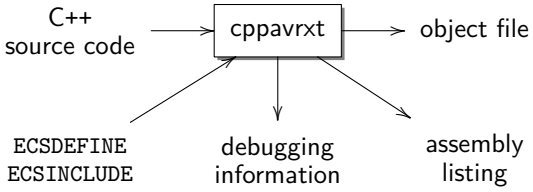
`cppavr` is a compiler for the C++ programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avr__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavrrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.18 cppavrxt

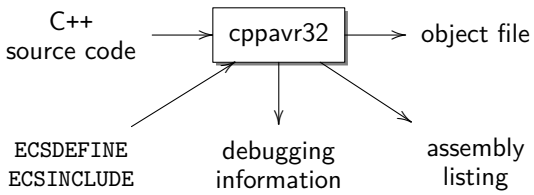
`cppavrxt` is a compiler for the C++ programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avrxt__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavrxtrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.19 cppavr32

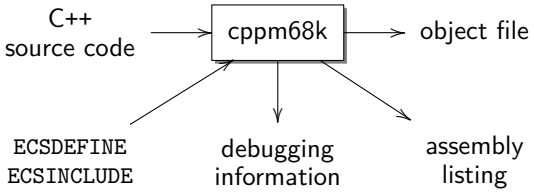
`cppavr32` is a compiler for the C++ programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avr32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavr32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.20 cppm68k

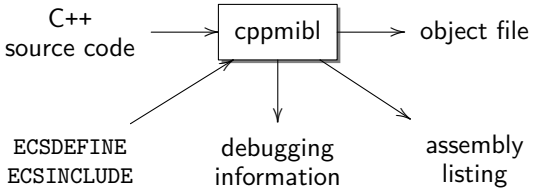
`cppm68k` is a compiler for the C++ programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__m68k__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppm68krun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.21 cppmibl

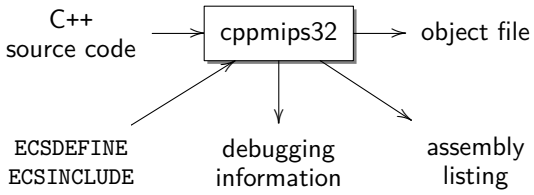
`cppmibl` is a compiler for the C++ programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mibl__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmiblrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.22 cppmips32

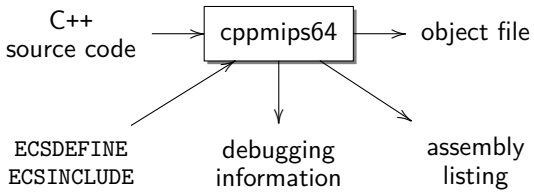
`cppmips32` is a compiler for the C++ programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mips32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmips32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.23 cppmips64

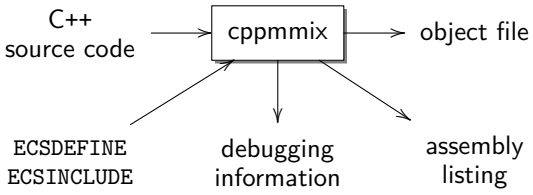
`cppmips64` is a compiler for the C++ programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mips64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmips64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.24 cppmmix

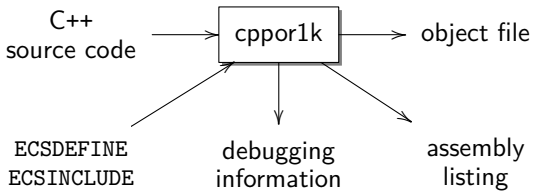
`cppmmix` is a compiler for the C++ programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mmix__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmmixrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.25 cppor1k

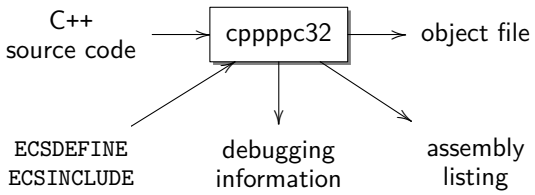
`cppor1k` is a compiler for the C++ programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__or1k__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppor1krun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.26 cppppc32

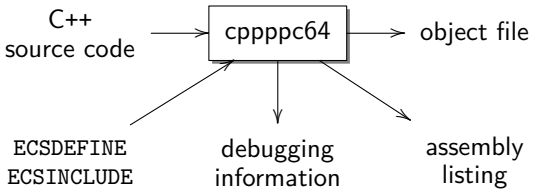
`cppppc32` is a compiler for the C++ programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__ppc32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppppc32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.27 cppppc64

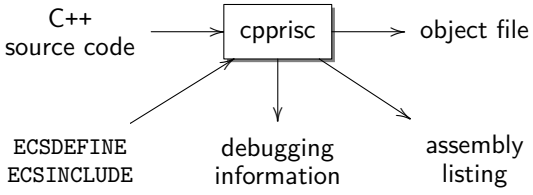
cppppc64 is a compiler for the C++ programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__ppc64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppppc64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.28 cpprisc

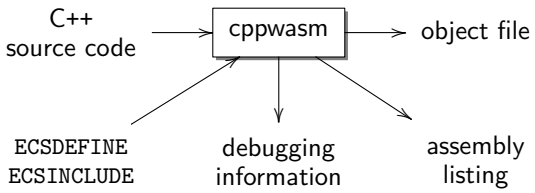
cpprisc is a compiler for the C++ programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__risc__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppriscrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.29 cppwasm

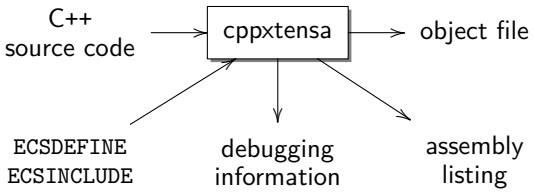
`cppwasm` is a compiler for the C++ programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__wasm__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppwasmrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.6.30 cppxtensa

`cppxtensa` is a compiler for the C++ programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__xtensa__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppxtensarun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

5.7 Interoperability

In accordance with the goal of the Eigen Compiler Suite to enable interoperability between its implemented programming languages, the compilers for C++ provide different mechanisms to exchange data with programs written with other tools of the Eigen Compiler Suite. The interoperability is enabled by a common intermediate code representation and calling convention. For more information about intermediate code and its purpose, see Chapter 23.

This section describes the naming conventions used to uniquely identify the intermediate code sections defined by the compilers for C++ as well as the ways of accessing sections defined by other compilers and assemblers.

5.7.1 Naming Conventions

The compilers for the C++ programming language emit a code section for each function definition, function bodies of lambda expressions, and non-local variable with dynamic initialization. Additional data sections are defined for non-local variables and type information required at runtime.

The name of each section equals to the name of the corresponding entity and is prefixed by the name of its containing scope in case the entity does not have C language linkage. The names of entities and their scope are delimited by scope operators such that names resemble qualified identifiers and no name mangling is necessary.

5.7.2 Accessing Sections

C++ as implemented by the Eigen Compiler Suite allows two different ways of accessing sections defined by other compilers or assemblers.

- The `asm` declaration allows writing inline assembly code which naturally enables arbitrary access to any section, see Section 5.2.7. For more information about the generic assembly language and how to use it, see Chapter 8.
- The `extern` specifier allows referring to data and code sections that are defined elsewhere. In cases where the different language linkages supported by the Eigen Compiler Suite do not automatically yield the required section name for a suitably declared function or variable, the alias attribute can be used to provide an arbitrary section name by hand, see Section 5.2.7.

5.7.3 Program Execution

The entry point of a program is represented by the `main` function. Unless provided by any other programming language, it must be defined in one of the linked C++ translation units. Any dynamic initialization of global variables with static storage duration is executed before the `main` function. The destruction of variables with static storage duration on the other hand requires an explicit or implicit call of the standard `exit` function.

6 FALSE

FALSE is a stack-oriented programming language that supports lambda abstractions and is quite powerful for its size. Although FALSE is quite cryptic by design and therefore considered esoteric, it provides language features that enable interoperability with other programming languages. This chapter describes the language and its implementation by the Eigen Compiler Suite.

What is written without effort
is in general read without pleasure.

— Samuel Johnson

6.1 Introduction

FALSE is an esoteric but still powerful programming language designed by Wouter van Oortmerssen, named after his favorite truth value [7].

Each FALSE program consists of comments enclosed in braces and expressions. Expressions consist of symbols which are mostly represented as a single character. Table 6.1 lists all of these symbols and the way the operate on the stack and the 26 variables predefined by the FALSE programming language. The notation $u, v \rightarrow s, t$ means that u and v are popped from the stack in that order and s and t are pushed onto it afterward. An ε on either side of that notation denotes that the corresponding operation does not push or pop anything respectively. The symbols listed under the category named special-purpose are actually implementation-defined and are explained in more detail in Section 6.2.

In general, there are three kinds of symbols which either push values onto the stack, change values on the stack, or just pop values from it in order to perform memory accesses or input and output. The type of the actual value that is pushed onto or popped from the stack is defined by the operation performed by the symbol. Arithmetic operations like + and − for example pop two integer values from the stack and push their result back onto the stack again. A single

Category	Symbol	Description	Stack Operation
Literals	`number`	Value of integer n	$\varepsilon \rightarrow n$
	`'character`	Value of character c	$\varepsilon \rightarrow c$
	`a...z`	Address a of variable	$\varepsilon \rightarrow a$
	`[...]`	Address f of function	$\varepsilon \rightarrow f$
Arithmetic Operations	`_`	Negation	$x \rightarrow -x$
	`+`	Add	$y,x \rightarrow x+y$
	`-`	Subtract	$y,x \rightarrow x-y$
	`*`	Multiply	$y,x \rightarrow x \times y$
	`/`	Divide	$y,x \rightarrow x \div y$
Logical Operations	`~`	Complement	$x \rightarrow \neg x$
	`&`	And	$y,x \rightarrow x \wedge y$
	`|`	Or	$y,x \rightarrow x \vee y$
Comparison Operations	`=`	Test if equal	$y,x \rightarrow x = y$
	`>`	Test if greater	$y,x \rightarrow x > y$
Stack Operations	`$`	Duplicate	$x \rightarrow x,x$
	`%`	Delete	$x \rightarrow \varepsilon$
	`\`	Swap	$x,y \rightarrow x,y$
	`@`	Rotate	$x,y,z \rightarrow y,x,z$
	`ø`	Select	$n \rightarrow n^{th}$ element
Statements	`:`	Write x to address a	$a,x \rightarrow \varepsilon$
	`;`	Read x from address a	$a \rightarrow x$
	`!`	Call function f	$f \rightarrow \varepsilon$
	`?`	Call function f if x is true	$f,x \rightarrow \varepsilon$
	`#`	Call function f while function g returns true	$f,g \rightarrow \varepsilon$
Input/ Output	`.`	Print x as integer	$x \rightarrow \varepsilon$
	`,`	Print x as character	$x \rightarrow \varepsilon$
	`^`	Read x as character	$\varepsilon \rightarrow x$
	`"..."`	Print string s	$\varepsilon \rightarrow \varepsilon$
	`ß`	Flush output stream	$\varepsilon \rightarrow \varepsilon$
Language extensions	`"..."V`	Address a of external variable s	$\varepsilon \rightarrow a$
	`"..."F`	Address a of external function s	$\varepsilon \rightarrow a$
	`"..."` `	Emit inline assembly	$\varepsilon \rightarrow \varepsilon$

Table 6.1: The symbols of the FALSE programming language

character symbol like `a` or `b` on the other hand names a variable and pushes the address of that variable onto the stack. The actual contents of the variable can be an integer or another address and is accessed using the dereference symbol `;`. The third kind of symbols are statements like assignments that perform their operation without pushing any new values onto the stack.

6.2 Implementation-Defined Behavior

Some issues in the specification of the otherwise abstract programming language are either left unspecified or depend on its original implementation. This section describes the concrete implementation specific behavior defined by the Eigen Compiler Suite:

- The actual size of the values stored on the stack and in variables is equal to the address size of the target hardware architecture.

- The logical and comparison operations generate Boolean values where false is represented using the integer zero and true is represented using the integer one. For evaluations of Boolean values the integer zero denotes false while all other values denote true.

- The behavior of programs that pop more elements from the stack than were pushed onto it beforehand is undefined. The same holds for programs that access memory using invalid addresses.

- After the execution of each program, the value on top of the stack is taken as the return code for the runtime environment. In order to allow programs that do not explicitly push a return code, each program first pushes a value indicating successful execution.

- The original specification allowed emitting inline assembly for the Motorola M68000 family of microprocessors using integer numbers in the range 0 up to 65335 followed by an apostrophe. The Eigen Compiler Suite does not provide this form of code generation because it supports more than one hardware architecture. All of them define their own instruction set encoding which affects the length of an instruction as well as endianness issues.

  Instead, the Eigen Compiler Suite allows emitting inline assembly by providing the actual inline assembly code as a string followed by an apostrophe. This is a more generic solution which does not need the programmer to apply the instruction set encoding manually. Nevertheless, representing binary code and data with plain integers is still possible using the double byte data directive. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13.

- The Eigen Compiler Suite defines two new operations that allow retrieving the address of variables and functions that are not defined by FALSE programs. This enables interoperability with other languages as described in Section 6.4.

Accessing external symbols as well as the emission of inline assembly is only available in the compilers tools provided by the Eigen Compiler Suite. The interpreter does not support these operations and issues a corresponding error message.

6.3 FALSE Tools

The Eigen Compiler Suite provides several different tools that process programs written in FALSE. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

The tools process FALSE programs in several consecutive stages. In each stage, the internal representation of the program is changed and transformed. Figure 6.1 shows all stages and the different representations.

6.3.1 falprint

`falprint` is a pretty printer for the FALSE programming language. It reformats the source code of FALSE programs and writes it to the standard output stream.

6.3.2 falcheck

`falcheck` is a syntactic and semantic checker for the FALSE programming language. It just performs syntactic and semantic checks on FALSE programs and writes its diagnostic messages to the standard error stream.

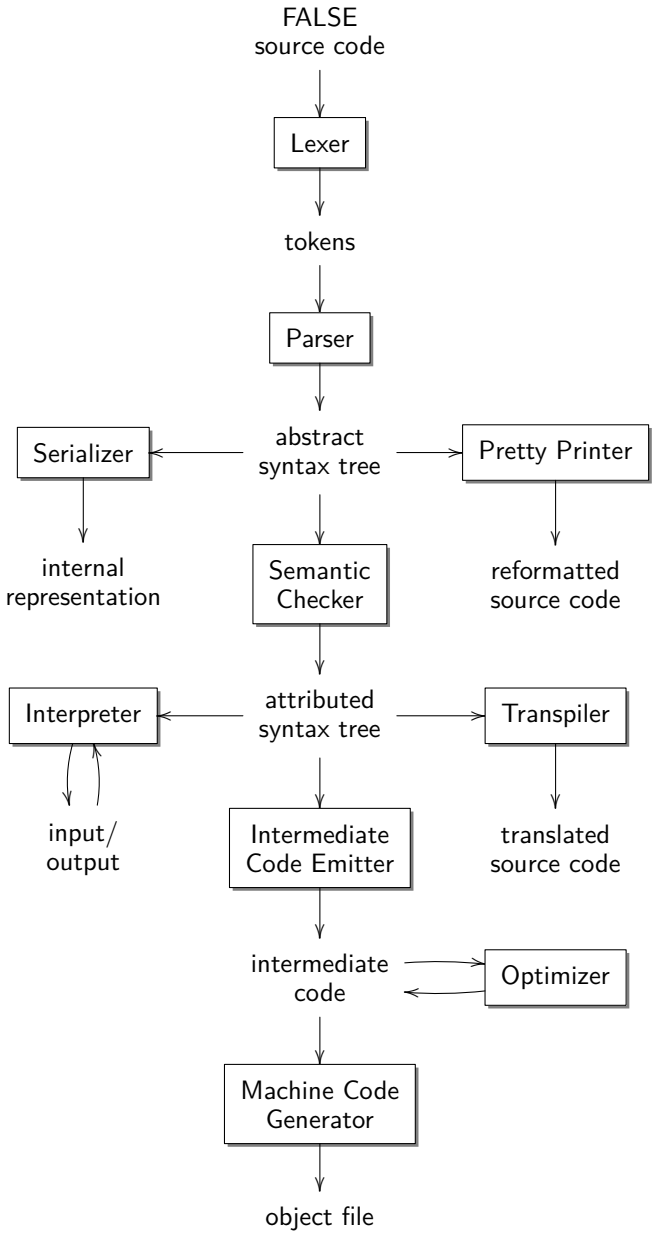
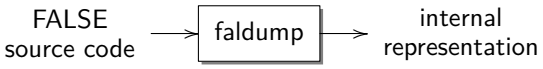


Figure 6.1: Data flow within the tools for FALSE

6.3.3 faldump

`faldump` is a serializer for the FALSE programming language. It dumps the complete internal representation of programs written in FALSE into an XML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



6.3.4 falrun

`falrun` is an interpreter for the FALSE programming language. It processes and executes programs written in FALSE.

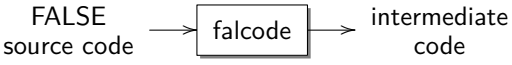
6.3.5 falcpp

`falcpp` is a transpiler for the FALSE programming language. It translates programs written in FALSE into C++ programs and stores them in corresponding source files.

For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

6.3.6 falcode

`falcode` is an intermediate code generator for the FALSE programming language. It generates intermediate code from programs written in FALSE and stores it in corresponding assembly files. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about intermediate code and its purpose, see Chapter 23.

6.3.7 falamd16

`falamd16` is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture.

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

6.3.8 falamd32

`falamd32` is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture.

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

6.3.9 falamd64

`falamd64` is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture.

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

6.3.10 falarma32

`falarma32` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

6.3.11 falarma64

`falarma64` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

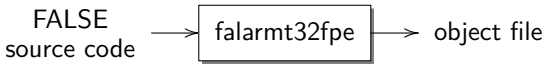
6.3.12 falarmt32

`falarmt32` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

6.3.13 falarmt32fpe

`falarmt32fpe` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in FALSE and stores it in corresponding object files.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

6.3.14 falavr

`falavr` is a compiler for the FALSE programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22.

6.3.15 falavrxt

`falavrxt` is a compiler for the FALSE programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22.

6.3.16 falavr32

`falavr32` is a compiler for the FALSE programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12. For more information about object files and their purpose, see Chapter 22.

6.3.17 falm68k

`falm68k` is a compiler for the FALSE programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13. For more information about object files and their purpose, see Chapter 22.

6.3.18 falmibl

`falmibl` is a compiler for the FALSE programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14. For more information about object files and their purpose, see Chapter 22.

6.3.19 falmips32

`falmips32` is a compiler for the FALSE programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22.

6.3.20 falmips64

`falmips64` is a compiler for the FALSE programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22.

6.3.21 falmmix

`falmmix` is a compiler for the FALSE programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16. For more information about object files and their purpose, see Chapter 22.

6.3.22 falor1k

`falor1k` is a compiler for the FALSE programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17. For more information about object files and their purpose, see Chapter 22.

6.3.23 falppc32

`falppc32` is a compiler for the FALSE programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture.

For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22.

6.3.24 falppc64

`falppc64` is a compiler for the FALSE programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture.

For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22.

6.3.25 falrisc

`falrisc` is a compiler for the FALSE programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19. For more information about object files and their purpose, see Chapter 22.

6.3.26 falwasm

`falwasm` is a compiler for the FALSE programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20. For more information about object files and their purpose, see Chapter 22.

6.3.27 falxtensa

`falxtensa` is a compiler for the FALSE programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in FALSE and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21. For more information about object files and their purpose, see Chapter 22.

6.4 Interoperability

The compilers for FALSE enable interoperability with other programming languages implemented by the Eigen Compiler Suite. The interoperability is enabled by a common intermediate code representation and calling convention. For more information about intermediate code and its purpose, see Chapter 23. The compilers define several intermediate code sections for each program and maintain the following naming convention.

The main program is defined in a code section called `main`. For each function inside the main program, the compilers define a code section called `function` followed by an integer index. The index is incremented with each function discovered lexicographically in the source code and begins with zero. In addition, each variable that is actually used is defined in a data section with the same name.

Accessing code and data sections that are defined elsewhere is enabled by two special-purpose symbols that push the corresponding address onto the stack. Since all stack operations operate on the actual call stack, it is even possible to provide arguments for external functions. However, the arguments are always passed as addresses. The return value of a function is pushed onto the stack after the call.

Functions defined by the compilers can also be called from other programs. They first pop the return address from the stack such that the top of the stack corresponds to the last argument passed by the caller. In the end, right before returning to the caller, the top of the stack is taken as return value of the function.

In addition, the compilers for the FALSE programming language also allow writing inline assembly code. The corresponding symbol is described in Section 6.2 and enables arbitrary access to any section. For more information about the generic assembly language and how to use it, see Chapter 8.

7 Oberon

Oberon is a general-purpose programming language based on Pascal and Modula-2. It supports type extension with type-bound procedures which makes it an object-oriented language. This chapter describes Oberon and its implementation by the Eigen Compiler Suite.

Die Kunst ist eine Tochter der Freiheit.

— Friedrich Schiller

7.1 Introduction

The Eigen Compiler Suite implements the Oberon programming language according to the Oberon-2 language report [8]. The most important language features are block structure, modularity, separate compilation, static typing with strong type checking, and type extension with type-bound procedures.

The Eigen Compiler Suite adds some completely backwards-compatible features for portability, genericity, and interoperability which are described in the remainder of this chapter.

7.2 Implementation-Defined Behavior

Although the semantics of the Oberon programming language is well defined, there are some issues that depend on its actual implementation and need detailed description. This section lists all implementation-defined behavior and language extensions along with the corresponding section numbers of the Oberon-2 language report where all changes to the syntax and the set of predeclared identifiers are underlined.

7.2.1 Vocabulary and Representation (3)

- Underscore characters in identifiers (3.1)

  For interoperability with other languages and interfacing with external libraries the Eigen Compiler Suite allows underscore characters in identifiers. The first character must still be a letter.

- Binary integer constants (3.2)

  If an integer constant is specified with the suffix `B`, its digits and representation are binary.

- Octal integer constants (3.2)

  If an integer constant is specified with the suffix `O`, its digits and representation are octal.

- Digit grouping (3.2)

  The integer part of a number may contain separating single quotes between its digits which are ignored when determining its value.

- Type of real numbers (3.2)

  The type of a real number is the minimal type to which the constant value belongs. The letter contained in the scale factor has no meaning with respect to the type.

- Line breaks in strings (3.4)

  The Eigen Compiler Suite allows line breaks in strings in order to support inline assembly code with multiple lines, see Section 7.2.9.

7.2.2 Declarations and Scope Rules (4)

- Nested qualified identifiers (4)

  Qualified identifiers may be nested:

  Qualident = [*Qualident* .] *ident*

- External declarations (4)

  The identifier definition of a variable or procedure declared in a module block may be followed by a constant expression enclosed in brackets which marks the declaration as *external*:

  IdentDef = *ident*[*|-] [[*ConstExpression*]]

Category	Identifier	Section
Constants	I, INF, NAN	7.2.3
Types	CARDINAL, COMPLEX, COMPLEX32, COMPLEX64, HUGECARD, HUGEINT, LENGTH, LONGCARD, LONGCOMPLEX, LONGSET, REAL32, REAL64, SET8, SET16, SET32, SET64, SHORTCARD, SHORTCOMPLEX, SHORTREAL, SIGNED8, SIGNED16, SIGNED32, SIGNED64, UNSIGNED8, UNSIGNED16, UNSIGNED32, UNSIGNED64	7.2.3
Procedures	DISPOSE, IGNORE, IM, PTR, RE, SEL, TRACE	7.2.7

Table 7.1: Additionally predeclared identifiers in Oberon

An external declaration either defines a global alias name using a string or specifies its address using an integer included by SYSTEM.ADDRESS and requires the module SYSTEM to be imported, see Section 7.2.9. Forward declarations marked as external do not have to be declared later in the text and refer to arbitrary entities with the respective name or address.

- Predeclared identifiers (4)

  The Eigen Compiler Suite adds the identifiers listed in Table 7.1 to the set of predeclared identifiers.

7.2.3 Type Declarations (6)

- Additional real values (6.1)

  Real numbers also contain the predeclared values INF and NAN denoting positive infinity and a quiet not a number.

- Additional complex values (6.1)

  The Eigen Compiler Suite supports complex numbers which belong to the numeric types and contain two real numbers representing the real and imaginary part of the complex number. Complex numbers also contain the predeclared value I denoting the imaginary unit *i*.

- Additional numeric types (6.1)

  The Eigen Compiler Suite provides additional predeclared identifiers for numeric types with a fixed bit length. They consist of four signed integer types called SIGNED8, SIGNED16, SIGNED32, and SIGNED64, four unsigned integer types called UNSIGNED8,

UNSIGNED16, UNSIGNED32, and UNSIGNED64, two real types called REAL32 and REAL64, as well as two complex types called COMPLEX32 and COMPLEX64. They form the following hierarchy where larger types include smaller ones:

```
COMPLEX32 ⊇  REAL32   ⊇ UNSIGNED64 ⊇  SIGNED64
    ∩|          ∩|          |∪           |∪
COMPLEX64 ⊇  REAL64     UNSIGNED32 ⊇  SIGNED32
                            |∪           |∪
                        UNSIGNED16 ⊇  SIGNED16
                            |∪           |∪
                         UNSIGNED8 ⊇  SIGNED8
```

The predefined set of numeric types is additionally extended by a signed integer type called HUGEINT, four unsigned integer types called SHORTCARD, CARDINAL, LONGCARD, and HUGECARD, a real type called SHORTREAL, as well as three complex types called SHORTCOMPLEX, COMPLEX, and LONGCOMPLEX. They form the following hierarchy where the types INTEGER, CARDINAL, and REAL name the default integer and real types of the execution environment:

```
SHORTCOMPLEX ⊇  SHORTREAL  ⊇  HUGECARD  ⊇  HUGEINT
     ∩|            ∩|           |∪           |∪
  COMPLEX    ⊇     REAL        LONGCARD  ⊇  LONGINT
     ∩|            ∩|           |∪           |∪
LONGCOMPLEX  ⊇  LONGREAL       CARDINAL  ⊇  INTEGER
                                  |∪           |∪
                               SHORTCARD ⊇  SHORTINT
```

Finally, the Eigen Compiler Suite also predeclares a signed integer type called LENGTH which is big enough to represent any array length of the execution environment.

- Additional set types (6.1)

The Eigen Compiler Suite provides additional predeclared identifiers for set types with a fixed bit length. They consist of four set types called SET8, SET16, SET32, and SET64. They form the following hierarchy where larger types include smaller ones:

```
SET64 ⊇ SET32 ⊇ SET16 ⊇ SET8
```

The predefined set of set types is additionally extended by three set types called SHORTSET, LONGSET, and HUGESET. They form the following hierarchy where the type SET names the default set type of the execution environment:

HUGESET ⊇ LONGSET ⊇ SET ⊇ SHORTSET

- Type values (6.1)

  The sizes and value ranges of all basic types as defined by the Eigen Compiler Suite are listed in Table 7.2. The sizes of the types CARDINAL, INTEGER, LENGTH, REAL, SET, and pointer types depend on the execution environment and are listed in Table 7.3 for each Oberon compiler provided by the Eigen Compiler Suite. The interpreter and all other tools reuse the respective type sizes of their own execution environment instead.

- Abstract and final record types (6.3)

  The RECORD keyword of a record type declaration may be followed by an export mark where "*" indicates that the record type is *abstract* and "-" indicates that it is *final*:

  RecordType = RECORD [*|-] [(*BaseType*)] *FieldList* {; *FieldList*} END

  An abstract record type must be extended before it can be assigned, allocated, or used as the type of array elements, variables, fields, value parameters, or function results. A final record type is not extensible.

- Initialization of pointers (6.4)

  All pointer variables are initialized to NIL.

- Pointer to variables (6.4)

  A pointer type containing the VAR keyword may point to any variable:

  PointerType = POINTER TO [VAR] [-] *Type*.

  A pointer to a variable can be obtained using the predeclared procedure PTR, see Section 7.2.7, and is valid as long as the variable exists. Pointers to variables are only expression compatible with other pointers to variables of extended or array compatible types where any type extends itself.

- Read-only pointers (6.4)

  Pointer type declarations may be marked as read-only:

  PointerType = POINTER TO [VAR] [-] *Type*.

  Read-only pointers are only assignment compatible with read-only pointers. The referenced variable of a read-only pointer is also read-only.

Category	Type	Alias	Size	Value Range
Boolean	BOOLEAN		1	TRUE or FALSE
Character	CHAR		1	0X to 0FFX
Signed integer	SIGNED8		1	-2^7 to $+2^7-1$
	SIGNED16	SHORTINT	2	-2^{15} to $+2^{15}-1$
		INTEGER	2/4	*See Table 7.3*
	SIGNED32	LONGINT	4	-2^{31} to $+2^{31}-1$
	SIGNED64	HUGEINT	8	-2^{63} to $+2^{63}-1$
		LENGTH	2/4/8	*See Table 7.3*
Unsigned integer	UNSIGNED8		1	0 to 2^8-1
	UNSIGNED16	SHORTCARD	2	0 to $2^{16}-1$
		CARDINAL	2/4	*See Table 7.3*
	UNSIGNED32	LONGCARD	4	0 to $2^{32}-1$
	UNSIGNED64	HUGECARD	8	0 to $2^{64}-1$
Real number	REAL32	SHORTREAL	4	$\pm 3.4028234 \times 10^{38}$
		REAL	4/8	*See Table 7.3*
	REAL64	LONGREAL	8	$\pm 1.7976931348623157 \times 10^{308}$
Complex number	COMPLEX32	SHORTCOMPLEX	8	$\pm 3.4028234 \times 10^{38} i$
		COMPLEX	8/16	*Two real numbers*
	COMPLEX64	LONGCOMPLEX	16	$\pm 1.7976931348623157 \times 10^{308} i$
Set	SET8		1	{} to {0..7}
	SET16	SHORTSET	2	{} to {0..15}
		SET	2/4	*See Table 7.3*
	SET32	LONGSET	4	{} to {0..31}
	SET64	HUGESET	8	{} to {0..63}

Table 7.2: Sizes and value ranges of basic Oberon types

Hardware Architecture		Oberon Compiler	CARDINAL, INTEGER, SET	REAL	LENGTH, POINTER	PROCEDURE
AMD64	16-bit	obamd16	2	8	2	2
	32-bit	obamd32	4	8	4	4
	64-bit	obamd64	4	8	8	8
ARM	A32	obarma32	4	8	4	4
	A64	obarma64	4	8	8	8
	T32	obarmt32	4	4	4	4
		obarmt32fpe	4	8	4	4
AVR		obavr	2	4	2	2
		obavrxt	2	4	2	3
AVR32		obavr32	4	4	4	4
M68000		obm68k	2	4	4	4
MicroBlaze		obmibl	4	4	4	4
MIPS	32-bit	obmips32	4	8	4	4
	64-bit	obmips64	4	8	8	8
MMIX		obmmix	4	8	8	8
OpenRISC 1000		obor1k	4	4	4	4
PowerPC	32-bit	obppc32	4	4	4	4
	64-bit	obppc64	4	4	4	4
RISC		obrisc	4	4	4	4
WebAssembly		obwasm	4	8	4	4
Xtensa		obxtensa	4	4	4	4

Table 7.3: Sizes of hardware-dependent Oberon types

7.2.4 Variable Declarations (7)

- Forward declarations (7)

  A forward declaration allows forward references to variables whose actual declaration appears later in the text. The data type of the forward declaration and the actual declaration must be the same.

  VariableDeclaration = [^] *IdentList* : *Type*

7.2.5 Expressions (8)

- Module designators (8.1)

  The identifier of a designator may refer to an imported module. Selectors following a module designator are treated as qualified identifiers.

- Value conversions (8.2)

  Any identifier denoting a basic type can also be used as the name of a function procedure that accepts a single value parameter of basic type. The result of calling such a procedure is the value of the parameter converted to the named type.

- Typed set constructors (8.2.3)

  A set constructor may be prefixed by an identifier which specifies the set type of the result.

  Set = [*Qualident*] { [*Element* { , *Element*}] }

  If omitted, the type of the result defaults to the minimal set type to which its constant value belongs, or SET if the result is not constant.

- Same type test (8.2.4)

  Type tests are also applicable on two types and yield whether they are the same.

7.2.6 Procedure Declarations (10)

- Abstract and final procedures (10)

  The PROCEDURE keyword of a type-bound procedure declaration may be followed by an export mark where "*" indicates that the procedure is *abstract* and "-" indicates that it is *final*. The corresponding export marks of a forward declaration and its actual declaration must be identical:

  ProcedureHeading = PROCEDURE [*|-] [*Receiver*] *IdentDef* [*FormalParameters*]

  ForwardDeclaration = PROCEDURE [*|-] ^ [*Receiver*] *IdentDef* [*FormalParameters*]

An abstract procedure has no procedure body and requires all record types to which it is bound to be abstract as well. A final procedure is not redefinable.

- Read-only parameters (10.1)

  Receivers and formal parameters may be marked as read-only just like variables and fields:

  FPSection = [VAR] *IdentList*: *Type*
  Receiver = ([VAR] *IdentDef*: *ident*)

  A type-bound procedure can be called using a read-only variable or field of record type if its receiver is read-only as well.

- Structured result types (10.1)

  The result type of a function can be structured but neither an abstract record nor an open array.

7.2.7 Predeclared Procedures (10.3)

The Eigen Compiler Suite supports all function procedures predeclared in the Oberon-2 language report and adds the following extensions:

- Arithmetic shift ASH

  The result type of the predeclared function procedure ASH is the integer type of its arguments.

- Imaginary part IM

  The predeclared function procedure IM accepts a complex value and returns its imaginary part.

- Array length LEN

  The result type of the predeclared function procedure LEN is LENGTH, see Section 7.2.3.

- Variable pointer PTR

  The predeclared function procedure PTR accepts a variable of any type and returns a pointer to it, see Section 7.2.3.

- Real part RE

  The predeclared function procedure RE accepts a complex value and returns its real part.

- Binary selection SEL

  The predeclared function procedure SEL accepts a Boolean expression and two compatible expressions of any basic, pointer or procedure type. If the condition specified by the first argument is satisfied, it evaluates the second argument and otherwise the third.

The Eigen Compiler Suite supports all proper procedures predeclared in the Oberon-2 language report and adds the following extensions:

- Assertion ASSERT

  The optional second argument of the predeclared proper procedure ASSERT may be an integer constant between 0 and 255 that represents the exit status of the program. If the first argument has a constant value, the assertion is static and terminates the compilation rather than the program execution.

- Variable deallocation DISPOSE

  The predeclared proper procedure DISPOSE accepts a pointer variable that stores a pointer to a variable previously allocated with NEW. It deallocates the referenced variable and sets the pointer variable to NIL.

- Program termination HALT

  The predeclared proper procedure HALT accepts an integer constant between 0 and 255 that represents the exit status of the program.

- Value ignoring IGNORE

  The predeclared proper procedure IGNORE accepts an expression of any type and ignores its value.

- Variable allocation NEW

  The type of the array lengths accepted by the predeclared proper procedure NEW is LENGTH. If a variable allocation fails, the corresponding pointer variable assumes the value NIL.

- Expression evaluation TRACE

  The predeclared proper procedure TRACE allows printing the value of an arbitrary expression of any basic, pointer or procedure type for debugging purposes.

7.2.8 Modules (11)

- Generic modules (11)

  Modules with a list of identifiers following their name are called *generic* and can be parameterized using the same number of constant expressions when they are imported:

Module = MODULE *ident* [(*IdentList*)] [IN *Qualident*];
{*ImportList*} *DeclarationSequence*
[BEGIN *StatementSequence*] END *ident*.
Import = [*ident* :=] *ident* [(*ExpressionList*)] [IN *Qualident*]

During an import of a generic module, each identifier is declared as an optionally exported constant or type using the constant value or type identifier of the corresponding parameter.

- Module packages (11)

  Modules can be associated with a *package* which provides a separate scope for all modules contained therein. Module packages are named using the optional IN keyword:

  Module = MODULE *ident* [(*IdentList*)] [IN *Qualident*];
  {*ImportList*} *DeclarationSequence*
  [BEGIN *StatementSequence*] END *ident*.

  The package of imported modules can be specified either in front of the import list or following their name:

  ImportList = [IN *Qualident*] IMPORT *Import* {, *Import*};
  Import = [*ident* :=] *ident* [(*ExpressionList*)] [IN *Qualident*]

- Multiple import lists (11)

  A module may contain multiple import lists:

  Module = MODULE *ident* [(*IdentList*)] [IN *Qualident*];
  {*ImportList*} *DeclarationSequence*
  [BEGIN *StatementSequence*] END *ident*.

- Text following a module (11)

  The remaining text of the source file after the concluding dot of a module is completely ignored.

- Module loading (11)

  A program can consist of an arbitrary number of modules which are loaded automatically. At least one of them however must have a module body which serves as the entry point of the program if not provided by other compilers or assemblers.

7.2.9 The Module SYSTEM (C)

The Eigen Compiler Suite omits the procedure SYSTEM.CC and adds the types SYSTEM.ADDRESS and SYSTEM.SET. These are unsigned integer and set types that have the same size as LENGTH,

see Section 7.2.3. All other function procedures predeclared in the Oberon-2 language report are supported with the following extensions:

- Memory address `SYSTEM.ADR`

  The result type of the predeclared function procedure `SYSTEM.ADR` is `SYSTEM.ADDRESS`.

- Memory bit `SYSTEM.BIT`

  The types of the arguments of the predeclared function procedure `SYSTEM.BIT` are `SYSTEM.ADDRESS` and `LENGTH` respectively.

- Type interpretation `SYSTEM.VAL`

  If the second argument of the predeclared function procedure `SYSTEM.VAL` refers to a variable, the result is that variable interpreted as if it was declared with the specified type. Otherwise, the result is the value of the second argument converted to the specified basic, pointer, or procedure type.

The Eigen Compiler Suite omits the procedures `SYSTEM.GETREG` and `SYSTEM.PUTREG`. All other proper procedures predeclared in the Oberon-2 language report are supported with the following extensions:

- Inline assembly code `SYSTEM.ASM`

  The predeclared proper procedure `SYSTEM.ASM` allows writing inline assembly code using one of the various compilers for the Oberon programming language. It accepts a string storing the actual assembly code and passes it to the assembler used to generate the machine code. The available instruction set therefore depends on the actual compiler used to compile the module. For more information about the generic assembly language and how to use it, see Chapter 8.

  The names and values of all accessible constants with boolean, character, integer, or set type are predefined in the assembly code. Within a procedure, the names of its local variables and parameters are also predefined. The values of these definitions correspond to the offset of the variable or parameter relative to the frame pointer. For debugging purposes, all names predefined in inline assembly code and their actual values are accessible using the expression evaluation directive.

- Inline intermediate code `SYSTEM.CODE`

  The predeclared proper procedure `SYSTEM.CODE` allows writing inline intermediate code using one of the various compilers for the Oberon programming language. It accepts a string storing the actual intermediate code and predefines the same names as inline assembly code. For more information about intermediate code and its purpose, see Chapter 23.

- Memory deallocation SYSTEM.DISPOSE

  The predeclared proper procedure SYSTEM.DISPOSE accepts any pointer variable that stores a pointer to memory previously allocated with SYSTEM.NEW. It deallocates the memory and sets the pointer variable to NIL.

- Memory read SYSTEM.GET

  The type of first argument of the predeclared proper procedure SYSTEM.GET is SYSTEM.ADDRESS.

- Memory copy SYSTEM.MOVE

  The type of the first two arguments of the predeclared function procedure SYSTEM.MOVE is SYSTEM.ADDRESS whereas the type of the third argument is LENGTH.

- Memory allocation SYSTEM.NEW

  The type of the second argument of the predeclared proper procedure SYSTEM.NEW is LENGTH.

- Memory write SYSTEM.PUT

  The type of the first argument of the predeclared proper procedure SYSTEM.PUT is SYSTEM.ADDRESS.

7.2.10 The Oberon Environment (D)

- Commands (D1)

  The Oberon environment provided by the Eigen Compiler Suite does not support a shell that allows activating commands. It does however make the command-line arguments passed to the program itself available, see Section 7.3.1.

- Dynamic loading of modules (D2)

  The Eigen Compiler Suite does not support dynamic loading of modules.

- Garbage collection (D3)

  The Eigen Compiler Suite does not support garbage collection but provides a predeclared proper procedure called DISPOSE which allows deallocating variables manually, see Section 7.2.7.

- Browser (D4)

  The Oberon environment does not provide a separate browser tool for extracting the human-readable interface of a compiled module since it is already available in its symbol file, see Section 7.7. Instead, the Eigen Compiler Suite provides tools that can

Trap	Meaning	Context	
0	Division by zero (unused)	Arithmetic operators	(8.2)
1	Unsatisfied type test	With statements	(9.11)
2	Unmatched case label	Case statements	(9.5)
3	Missing return (unused)	Function procedures	(9.10)
4	Integer overflow (unused)	Arithmetic operators	(8.2)
5	Implicit type guard violation	Record assignments	(9.1)
6	Explicit type guard violation	Type guards	(8.1)
7	Index out of range	Array designators	(8.1)
8	Failed assertion	Assert procedures	(10.3)
9	Invalid array dimension	Array allocations	(6.4)

Table 7.4: Oberon trap numbers

generate cross-referenced documents from the interfaces of modules and their user-defined annotations, see Section 7.5.

- Run Time Data Structures (D5)

  All quantities like the number of modules, imports, declarations, type extension levels, fields, parameters, statements and expressions, or the length of identifiers, strings, and arrays have no intrinsic limit and are only restricted by available memory. The actual limit therefore depends only on the execution environment of the tool used to process the Oberon module.

- Trap Numbers

  Program execution is aborted when assertions like type guards are not satisfied. The runtime environment may indicate the reason for program terminations using trap numbers whose meaning is listed in Table 7.4. Some trap numbers are unused in which case the corresponding errors are not checked at runtime.

7.3 The Oberon Library

The Eigen Compiler Suite provides a collection of modules that comprise the *Oberon Library*. This library provides some useful utilities for programmers and is available by importing one or more of the modules listed in Table 7.5 from package `OBL`. All of these modules are governed by the ECS Runtime Support Exception which is an additional permission to the GNU General Public License that allows users of the Eigen Compiler Suite to create proprietary programs. Copies of these licenses are included in Appendices C and D on pages 495 and 509 respectively.

Category	Module	Section	Module	Section
Basic Types	`Booleans`	7.3.4	`Characters`	7.3.5
Containers	`DynamicArrays`	7.3.8	`HashMaps`	7.3.15
	`Iterators`	7.3.18	`Lists`	7.3.19
	`Trees`	7.3.27		
Coroutines	`Coroutines`	7.3.6	`Functions`	7.3.12
	`Generators`	7.3.13		
Input/ Output	`Devices`	7.3.7	`Files`	7.3.11
	`In`	7.3.16	`IOStreams`	7.3.17
	`Out`	7.3.21	`Streams`	7.3.25
Numerics	`Math`	7.3.20	`Random`	7.3.23
Utilities	`Arguments`	7.3.1	`Arrays`	7.3.2
	`BasicTypes`	7.3.3	`Environment`	7.3.9
	`Exceptions`	7.3.10	`Hashes`	7.3.14
	`Pairs`	7.3.22	`Sets`	7.3.24
	`Strings`	7.3.26		

Table 7.5: Modules of the Oberon Library

The remainder of this section lists all modules provided by the Oberon Library in alphabetical order and describes their exported interface.

7.3.1 Arguments Module

The module `Arguments` provides access to the command-line arguments passed to the program from the environment in which the program is run.

```
MODULE Arguments IN OBL;
```

Types (1)	`Arguments.Argument`
Variables (2)	`Arguments.count`
	`Arguments.values`

Arguments.Argument Type Alias Represents a null-terminated command-line argument.

```
TYPE Argument* = POINTER TO - ARRAY (* LENGTH *) OF CHAR;
```

Arguments.count Variable Provides the number of command-line arguments passed to the program.

```
VAR count-: INTEGER;
```

Arguments.values Variable Stores a pointer to an array of command-line arguments passed to the program. The number of valid array elements is stored in `count`. If that number is nonzero the first argument is either the name used to invoke the program or an empty string.

```
VAR values-: POINTER TO - ARRAY (* LENGTH *) OF Argument;
```

7.3.2 Arrays Module

The generic module `Arrays` provides an iterator type for iterating over arrays of type `Element`.

```
MODULE Arrays(Element*) IN OBL;
IMPORT Iterators(Element) IN OBL;
```

Parameters (1)	`Arrays.Element`
Procedures (5)	`Arrays.Fill`
	`Arrays.GetIterator`
	`Arrays.IsSorted`
	`Arrays.Reverse`
	`Arrays.Sort`
Types (1)	`Arrays.Iterator`

Arrays.Element Parameter The type of an element contained in an array.

```
Element* = (* GENERIC *);
```

Arrays.Fill Procedure Fills the array in the range [`begin`, `end`) with the given value.

```
PROCEDURE Fill* (
    VAR array: ARRAY OF Element;
    begin: LENGTH;
    end: LENGTH;
    value-: Element
);
```

Arrays.GetIterator Procedure Returns an iterator for the given open array.

```
PROCEDURE GetIterator* (
    array-: ARRAY OF Element;
    VAR iterator: Iterator
);
```

Arrays.IsSorted Procedure Returns whether the array is sorted in the range [`begin`, `end`) with the given element ordering.

```
PROCEDURE IsSorted* (
    array-: ARRAY OF Element;
    begin: LENGTH;
    end: LENGTH;
    compare: PROCEDURE (left-: Element; right-: Element): BOOLEAN
): BOOLEAN;
```

Arrays.Iterator Record Type Represents an iteration over an array.

```
TYPE Iterator* = RECORD- (Iterators(Element).Iterator) END;
```

Base Types (1)	`Iterators(Element).Iterator`
Procedures (1)	`Iterator.GetNext`

Arrays.Iterator.GetNext Procedure Returns the next element in the array iteration if available as indicated by the boolean result.

```
PROCEDURE- (VAR iterator: Iterator) GetNext* (
    VAR element: Element
): BOOLEAN;
```

Arrays.Reverse Procedure Reverses the order of the elements of the array in the range [`begin`, `end`) using the given element swap operation.

```
PROCEDURE Reverse* (
    VAR array: ARRAY OF Element;
    begin: LENGTH;
    end: LENGTH;
    swap: PROCEDURE (VAR left: Element; VAR right: Element)
);
```

Arrays.Sort Procedure Sorts the array in the range [`begin`, `end`) with the given element ordering and swap operation.

```
PROCEDURE Sort* (
    VAR array: ARRAY OF Element;
    begin: LENGTH;
    end: LENGTH;
    compare: PROCEDURE (left-: Element; right-: Element): BOOLEAN;
    swap: PROCEDURE (VAR left: Element; VAR right: Element)
);
```

7.3.3 BasicTypes Module

The generic module `BasicTypes` provides utility operations for basic types.

```
MODULE BasicTypes(Value*) IN OBL;
```

Constants (8)	`BasicTypes.IsBoolean`
	`BasicTypes.IsCharacter`
	`BasicTypes.IsInteger`
	`BasicTypes.IsNumeric`
	`BasicTypes.IsReal`
	`BasicTypes.IsSet`
	`BasicTypes.IsSignedInteger`
	`BasicTypes.IsUnsignedInteger`
Parameters (1)	`BasicTypes.Value`
Procedures (10)	`BasicTypes.Hash`
	`BasicTypes.IsEqual`
	`BasicTypes.IsGreaterOrEqual`
	`BasicTypes.IsGreaterThan`
	`BasicTypes.IsLessOrEqual`
	`BasicTypes.IsLessThan`
	`BasicTypes.IsNotEqual`
	`BasicTypes.Maximum`
	`BasicTypes.Minimum`
	`BasicTypes.Swap`

BasicTypes.Hash Procedure Returns a hash value for the specified value.

```
PROCEDURE Hash* (
    value-: Value
): LENGTH;
```

BasicTypes.IsBoolean Constant Indicates whether the basic type is a boolean type.

```
CONST IsBoolean* = (* BOOLEAN *);
```

BasicTypes.IsCharacter Constant Indicates whether the basic type is a character type.

```
CONST IsCharacter* = (* BOOLEAN *);
```

BasicTypes.IsEqual Procedure Returns whether `x` is equal to `y}`.

```
PROCEDURE IsEqual* (
    x-: Value;
```

```
        y-: Value
    ): BOOLEAN;
```

BasicTypes.IsGreaterOrEqual Procedure Returns whether x is greater than or equal to y}.

```
    PROCEDURE IsGreaterOrEqual* (
        x-: Value;
        y-: Value
    ): BOOLEAN;
```

BasicTypes.IsGreaterThan Procedure Returns whether x is greater than y}.

```
    PROCEDURE IsGreaterThan* (
        x-: Value;
        y-: Value
    ): BOOLEAN;
```

BasicTypes.IsInteger Constant Indicates whether the basic type is an integer type.

```
    CONST IsInteger* = (* BOOLEAN *);
```

BasicTypes.IsLessOrEqual Procedure Returns whether x is less than or equal to y}.

```
    PROCEDURE IsLessOrEqual* (
        x-: Value;
        y-: Value
    ): BOOLEAN;
```

BasicTypes.IsLessThan Procedure Returns whether x is less than y}.

```
    PROCEDURE IsLessThan* (
        x-: Value;
        y-: Value
    ): BOOLEAN;
```

BasicTypes.IsNotEqual Procedure Returns whether x is not equal to y}.

```
    PROCEDURE IsNotEqual* (
        x-: Value;
        y-: Value
    ): BOOLEAN;
```

BasicTypes.IsNumeric Constant Indicates whether the basic type is a numeric type.

```
    CONST IsNumeric* = (* BOOLEAN *);
```

BasicTypes.IsReal Constant Indicates whether the basic type is a real type.

```
CONST IsReal* = (* BOOLEAN *);
```

BasicTypes.IsSet Constant Indicates whether the basic type is a set type.

```
CONST IsSet* = (* BOOLEAN *);
```

BasicTypes.IsSignedInteger Constant Indicates whether the basic type is a signed integer type.

```
CONST IsSignedInteger* = (* BOOLEAN *);
```

BasicTypes.IsUnsignedInteger Constant Indicates whether the basic type is an unsigned integer type.

```
CONST IsUnsignedInteger* = (* BOOLEAN *);
```

BasicTypes.Maximum Procedure Returns the greater of the two specified values.

```
PROCEDURE Maximum* (
    x-: Value;
    y-: Value
): Value;
```

BasicTypes.Minimum Procedure Returns the smaller of the two specified values.

```
PROCEDURE Minimum* (
    x-: Value;
    y-: Value
): Value;
```

BasicTypes.Swap Procedure Swaps the values of two variables.

```
PROCEDURE Swap* (
    VAR first: Value;
    VAR second: Value
);
```

BasicTypes.Value Parameter The underlying basic type for all operations.

```
Value* = (* GENERIC *);
```

7.3.4 Booleans Module

The module `Booleans` provides utility operations for boolean values.

```
MODULE Booleans IN OBL;
```

Procedures (4)	`Booleans.Hash`
	`Booleans.IsEqual`
	`Booleans.IsNotEqual`
	`Booleans.Swap`
Types (1)	`Booleans.Value`

Booleans.Hash Procedure Returns a hash value for the specified value.

```
PROCEDURE Hash* (
    value-: BOOLEAN
): LENGTH;
```

Booleans.IsEqual Procedure Returns whether `x` is equal to `y`.

```
PROCEDURE IsEqual* (
    x-: BOOLEAN;
    y-: BOOLEAN
): BOOLEAN;
```

Booleans.IsNotEqual Procedure Returns whether `x` is not equal to `y`.

```
PROCEDURE IsNotEqual* (
    x-: BOOLEAN;
    y-: BOOLEAN
): BOOLEAN;
```

Booleans.Swap Procedure Swaps the values of two variables.

```
PROCEDURE Swap* (
    VAR first: BOOLEAN;
    VAR second: BOOLEAN
);
```

Booleans.Value Type Alias The underlying basic type for all operations.

```
TYPE Value* = BOOLEAN;
```

7.3.5 Characters Module

The generic module `Characters` provides utility operations for character values.

```
MODULE Characters(Value*) IN OBL;
```

Parameters (1)	`Characters.Value`
Procedures (8)	`Characters.Hash`

```
Characters.IsEqual
Characters.IsGreaterOrEqual
Characters.IsGreaterThan
Characters.IsLessOrEqual
Characters.IsLessThan
Characters.IsNotEqual
Characters.Swap
```

Characters.Hash Procedure Returns a hash value for the specified value.

```
PROCEDURE Hash* (
    value-: Value
): LENGTH;
```

Characters.IsEqual Procedure Returns whether x is equal to y.

```
PROCEDURE IsEqual* (
    x-: Value;
    y-: Value
): BOOLEAN;
```

Characters.IsGreaterOrEqual Procedure Returns whether x is greater than or equal to y.

```
PROCEDURE IsGreaterOrEqual* (
    x-: Value;
    y-: Value
): BOOLEAN;
```

Characters.IsGreaterThan Procedure Returns whether x is greater than y.

```
PROCEDURE IsGreaterThan* (
    x-: Value;
    y-: Value
): BOOLEAN;
```

Characters.IsLessOrEqual Procedure Returns whether x is less than or equal to y.

```
PROCEDURE IsLessOrEqual* (
    x-: Value;
    y-: Value
): BOOLEAN;
```

Characters.IsLessThan Procedure Returns whether x is less than y.

```
PROCEDURE IsLessThan* (
    x-: Value;
```

```
    y-: Value
): BOOLEAN;
```

Characters.IsNotEqual Procedure Returns whether x is not equal to y.

```
PROCEDURE IsNotEqual* (
    x-: Value;
    y-: Value
): BOOLEAN;
```

Characters.Swap Procedure Swaps the values of two variables.

```
PROCEDURE Swap* (
    VAR first: Value;
    VAR second: Value
);
```

Characters.Value Parameter The underlying character type for all operations.

```
Value* = (* GENERIC *);
```

7.3.6 Coroutines Module

The module `Coroutines` provides non-preemptive coroutines that share a common address space. Coroutines can explicitly transfer control to other coroutines which are then resumed from their last transfer of control.

```
MODULE Coroutines IN OBL;
```

Types (1) `Coroutines.Coroutine`

Coroutines.Coroutine Record Type Represents a non-preemptive coroutine.

```
TYPE Coroutine* = RECORD* END;
```

Procedures (2) `Coroutine.Call`
`Coroutine.Transfer`

Coroutines.Coroutine.Call Procedure Starts the operation of the coroutine. This abstract procedure is called by the first transfer to the coroutine.

```
PROCEDURE* (VAR coroutine: Coroutine) Call*;
```

Coroutines.Coroutine.Transfer Procedure Transfers control from the currently executing coroutine to `target`. This procedure may raise an exception of type `AllocationFailure`.

```
PROCEDURE- (VAR coroutine: Coroutine) Transfer* (
    VAR target: Coroutine
);
```

7.3.7 Devices Module

The module `Devices` provides a generic abstraction for devices that can read and write arbitrary data.

```
MODULE Devices IN OBL;
```

Types (1) `Devices.Device`

Devices.Device Record Type `TYPE Device* = RECORD* END;`

Procedures (2) `Device.Read`
`Device.Write`

Devices.Device.Read Procedure Reads arbitrary data from the device and returns wether the operation was successful.

```
PROCEDURE* (VAR device: Device) Read* (
    VAR data: ARRAY OF SYSTEM.BYTE
): BOOLEAN;
```

Devices.Device.Write Procedure Writes arbitrary data to the device and returns wether the operation was successful.

```
PROCEDURE* (VAR device: Device) Write* (
    VAR data-: ARRAY OF SYSTEM.BYTE
): BOOLEAN;
```

7.3.8 DynamicArrays Module

The generic module `DynamicArrays` provides an iterator type for iterating over dynamic arrays of type `Element`.

```
MODULE DynamicArrays(Element*) IN OBL;
IMPORT Iterators(Element) IN OBL;
```

Parameters (1) `DynamicArrays.Element`
Procedures (1) `DynamicArrays.Swap`
Types (2) `DynamicArrays.Array`
`DynamicArrays.Iterator`

DynamicArrays.Array Record Type Represents a dynamic array container with a dynamic size.

```
TYPE Array* = RECORD- END;
```

Procedures (12)
- Array.Add
- Array.Clear
- Array.Fill
- Array.GetElement
- Array.GetIterator
- Array.GetSize
- Array.IsEmpty
- Array.IsSorted
- Array.Reserve
- Array.Reverse
- Array.SetElement
- Array.Sort

DynamicArrays.Array.Add Procedure Appends the specified element to the end of the array. This procedure may raise an exception of type `AllocationFailure`.

```
PROCEDURE- (VAR array: Array) Add* (
    element-: Element
);
```

DynamicArrays.Array.Clear Procedure Removes all elements from the array. This procedure must be called when an array variable goes out of scope in order to deallocate its elements.

```
PROCEDURE- (VAR array: Array) Clear*;
```

DynamicArrays.Array.Fill Procedure Fills the array in the range [`begin`, `end`) with the given value.

```
PROCEDURE- (VAR array: Array) Fill* (
    begin: LENGTH;
    end: LENGTH;
    value-: Element
);
```

DynamicArrays.Array.GetElement Procedure Returns the element at the specified position in the array.

```
PROCEDURE- (VAR array-: Array) GetElement* (
    index: LENGTH
): Element;
```

DynamicArrays.Array.GetIterator Procedure Returns an iterator for the array.

```
PROCEDURE- (VAR array-: Array) GetIterator* (
    VAR iterator: Iterator
);
```

DynamicArrays.Array.GetSize Procedure Returns the size of the array.

```
PROCEDURE- (VAR array-: Array) GetSize* (): LENGTH;
```

DynamicArrays.Array.IsEmpty Procedure Returns whether the array is empty.

```
PROCEDURE- (VAR array-: Array) IsEmpty* (): BOOLEAN;
```

DynamicArrays.Array.IsSorted Procedure Returns whether the array is sorted in the range [begin, end) with the given element ordering.

```
PROCEDURE- (VAR array: Array) IsSorted* (
    begin: LENGTH;
    end: LENGTH;
    compare: PROCEDURE (left-: Element; right-: Element): BOOLEAN
): BOOLEAN;
```

DynamicArrays.Array.Reserve Procedure Reserves space for size array elements. This procedure may raise an exception of type AllocationFailure.

```
PROCEDURE- (VAR array: Array) Reserve* (
    size: LENGTH
);
```

DynamicArrays.Array.Reverse Procedure Reverses the order of the elements of the array in the range [begin, end) using the given element swap operation.

```
PROCEDURE- (VAR array: Array) Reverse* (
    begin: LENGTH;
    end: LENGTH;
    swap: PROCEDURE (VAR left: Element; VAR right: Element)
);
```

DynamicArrays.Array.SetElement Procedure Replaces the element at the specified position in the array.

```
PROCEDURE- (VAR array: Array) SetElement* (
    index: LENGTH;
    element-: Element
);
```

DynamicArrays.Array.Sort Procedure Sorts the array in the range [`begin`, `end`) with the given element ordering and swap operation.

```
PROCEDURE- (VAR array: Array) Sort* (
    begin: LENGTH;
    end: LENGTH;
    compare: PROCEDURE (left-: Element; right-: Element): BOOLEAN;
    swap: PROCEDURE (VAR left: Element; VAR right: Element)
);
```

DynamicArrays.Element Parameter The type of an element contained in an array.

```
Element* = (* GENERIC *);
```

DynamicArrays.Iterator Record Type Represents an iteration over a dynamic array.

```
TYPE Iterator* = RECORD- (Iterators(Element).Iterator) END;
```

Base Types (1)	`Iterators(Element).Iterator`
Procedures (1)	`Iterator.GetNext`

DynamicArrays.Iterator.GetNext Procedure Returns the next element in the array iteration if available as indicated by the boolean result.

```
PROCEDURE- (VAR iterator: Iterator) GetNext* (
    VAR element: Element
): BOOLEAN;
```

DynamicArrays.Swap Procedure Exchanges the contents of an array with that of another one.

```
PROCEDURE Swap* (
    VAR first: Array;
    VAR second: Array
);
```

7.3.9 Environment Module

The module `Environment` provides access to the environment in which the program is run.

```
MODULE Environment IN OBL;
```

Procedures (2)	`Environment.Call`
	`Environment.Get`
Types (1)	`Environment.Variable`

Variables (1) `Environment.name`

Environment.Call Procedure Calls the command processor of the host environment.

```
PROCEDURE Call* (
    command-: ARRAY OF CHAR
): INTEGER;
```

Environment.Get Procedure Returns the value of the environment variable with the specified name.

```
PROCEDURE Get* (
    name-: ARRAY OF CHAR
): Variable;
```

Environment.Variable Type Alias Represents a null-terminated environment variable.

```
TYPE Variable* = POINTER TO - ARRAY (* LENGTH *) OF CHAR;
```

Environment.name Variable The name of the environment.

```
VAR name-: ARRAY (* LENGTH *) OF CHAR;
```

7.3.10 Exceptions Module

The module `Exceptions` provides a generic exception handling facility. Exceptions are extensions of `Exception` and should only be used as the type of local variables.

```
MODULE Exceptions IN OBL;
```

Types (2) `Exceptions.AllocationFailure`
`Exceptions.Exception`

Exceptions are handled by calling `Try` on a local variable in the guard of an if statement. This procedure initially returns `TRUE` such that the guarded statement sequence of the if statement is executed. If this sequence directly or indirectly calls `Raise` on an exception of the same type or an extension thereof, the procedure returns again with the value `FALSE` such that the execution resumes in the `ELSE` part of the if statement. Otherwise the exception stays registered for exception handling until `End` is called, typically at the end of the guarded statement sequence.

Exceptions.AllocationFailure Record Type Represents a memory allocation failure exception.

```
TYPE AllocationFailure* = RECORD- (Exception) END;
```

Base Types (1)	`Exceptions.Exception`
From `Exception` (3)	`Exception.End`
	`Exception.Raise`
	`Exception.Try`

Exceptions.Exception Record Type Represents a generic exception that can be raised and handled.

```
TYPE Exception* = RECORD* END;
```

Procedures (3)	`Exception.End`
	`Exception.Raise`
	`Exception.Try`

Exceptions.Exception.End Procedure Unregisters `exception` from exception handling. Exceptions should be unregistered before leaving the guarded statement sequence of the if statement that registers and handles them.

```
PROCEDURE- (VAR exception: Exception) End*;
```

Exceptions.Exception.Raise Procedure Resumes control at the most current registration of an exception of the same or a base type and copies the value of `exception` to the corresponding variable.

```
PROCEDURE- (VAR exception-: Exception) Raise*;
```

Exceptions.Exception.Try Procedure Registers `exception` for exception handling and returns `TRUE`. Raising an exception of the same or an extended type transfers control to the most current call of this procedure and returns `FALSE`. Exceptions should be registered in the guard of an if statement and handled in its `ELSE` part.

```
PROCEDURE- (VAR exception: Exception) Try* (): BOOLEAN;
```

7.3.11 Files Module

The module `Files` provides a generic file operations.

```
MODULE Files IN OBL;
IMPORT Devices IN OBL;
```

Types (1)	`Files.File`

Files.File Record Type Represents a file.

```
TYPE File* = RECORD- (Devices.Device) END;
```

Base Types (1)	`Devices.Device`
Procedures (4)	`File.Close`
	`File.Open`
	`File.Read`
	`File.Write`

Files.File.Close Procedure Closes the file and returns wether the operation was successful.

```
PROCEDURE- (VAR file: File) Close* (): BOOLEAN;
```

Files.File.Open Procedure Opens a file using the specified mode and returns wether the operation was successful.

```
PROCEDURE- (VAR file: File) Open* (
    filename-: ARRAY OF CHAR;
    write: BOOLEAN
): BOOLEAN;
```

Files.File.Read Procedure Reads arbitrary data from the file and returns wether the operation was successful.

```
PROCEDURE- (VAR file: File) Read* (
    VAR data: ARRAY OF SYSTEM.BYTE
): BOOLEAN;
```

Files.File.Write Procedure Writes arbitrary data to the file and returns wether the operation was successful.

```
PROCEDURE- (VAR file: File) Write* (
    VAR data-: ARRAY OF SYSTEM.BYTE
): BOOLEAN;
```

7.3.12 Functions Module

The generic module `Functions` provides generators that yield a value of type `Result`. Functions are extensions of `Function` that represent their parameters and results as fields.

```
MODULE Functions(Result*) IN OBL;

IMPORT Coroutines IN OBL;
IMPORT Generators IN OBL;
IMPORT Iterators(Result) IN OBL;
```

Parameters (1)	`Functions.Result`
Types (5)	`Functions.Filter`
	`Functions.Function`
	`Functions.Iterate`
	`Functions.Iterator`
	`Functions.Take`

Functions.Filter Record Type Represents a function that filters all results from another function for which the predicate is satisfied.

```
TYPE Filter* = RECORD- (Function)
    function*: POINTER TO VAR Function;
    predicate*: PROCEDURE (value-: Result): BOOLEAN;
END;
```

Base Types (3)	`Coroutines.Coroutine`
	`Generators.Generator`
	`Functions.Function`
Fields (2)	`Filter.function`
	`Filter.predicate`
From `Coroutines.Coroutine` (1)	`Coroutine.Transfer`
From `Function` (2)	`Function.Return`
	`Function.result`
From `Generators.Generator` (2)	`Generator.Await`
	`Generator.Yield`
Procedures (1)	`Filter.Call`

Functions.Filter.Call Procedure Filters all results from another function for which the predicate is satisfied.

```
PROCEDURE- (VAR filter: Filter) Call*;
```

Functions.Filter.function Field The function to filter results from.

```
function*: POINTER TO VAR Function;
```

Functions.Filter.predicate Field The predicate used for filter results.

```
predicate*: PROCEDURE (value-: Result): BOOLEAN;
```

Functions.Function Record Type Represents a generator that may return a result multiple times to its caller.

```
TYPE Function* = RECORD* (Generators.Generator)
    result-: POINTER TO VAR - Result;
END;
```

Base Types (2)	`Coroutines.Coroutine`
	`Generators.Generator`
Fields (1)	`Function.result`
From `Coroutines.Coroutine` (2)	`Coroutine.Call`
	`Coroutine.Transfer`
From `Generators.Generator` (2)	`Generator.Await`
	`Generator.Yield`
Procedures (1)	`Function.Return`

Functions.Function.Return Procedure Returns `result` to the caller of the function.

```
PROCEDURE- (VAR function: Function) Return* (
    result-: Result
);
```

Functions.Function.result Field The result of a function call.

```
result-: POINTER TO VAR - Result;
```

Functions.Iterate Record Type Represents a function that iterates over an iterator.

```
TYPE Iterate* = RECORD- (Function)
    iterator*: POINTER TO VAR Iterators(Result).Iterator;
END;
```

Base Types (3)	`Coroutines.Coroutine`
	`Generators.Generator`
	`Functions.Function`
Fields (1)	`Iterate.iterator`
From `Coroutines.Coroutine` (1)	`Coroutine.Transfer`
From `Function` (2)	`Function.Return`
	`Function.result`
From `Generators.Generator` (2)	`Generator.Await`
	`Generator.Yield`
Procedures (1)	`Iterate.Call`

Functions.Iterate.Call Procedure Iterates over the specified iterator.

```
PROCEDURE- (VAR iterate: Iterate) Call*;
```

Functions.Iterate.iterator Field The iterator to iterate over.

```
iterator*: POINTER TO VAR Iterators(Result).Iterator;
```

Functions.Iterator Record Type Represents an iterator over the results of calling a function.

```
TYPE Iterator* = RECORD- (Iterators(Result).Iterator)
    function*: POINTER TO VAR Function;
END;
```

Base Types (1)	`Iterators(Result).Iterator`
Fields (1)	`Iterator.function`
Procedures (1)	`Iterator.GetNext`

Functions.Iterator.GetNext Procedure Returns the next element in the function iteration if available as indicated by the boolean result.

```
PROCEDURE- (VAR iterator: Iterator) GetNext* (
    VAR element: Result
): BOOLEAN;
```

Functions.Iterator.function Field The function to call.

```
function*: POINTER TO VAR Function;
```

Functions.Result Parameter The type of the result returned by functions.

```
Result* = (* GENERIC *);
```

Functions.Take Record Type Represents a function that takes a number of results from another function.

```
TYPE Take* = RECORD- (Function)
    function*: POINTER TO VAR Function;
    count*: LENGTH;
END;
```

Base Types (3)	`Coroutines.Coroutine`
	`Generators.Generator`
	`Functions.Function`
Fields (2)	`Take.count`
	`Take.function`
From `Coroutines.Coroutine` (1)	`Coroutine.Transfer`
From `Function` (2)	`Function.Return`

	Function.result
From Generators.Generator (2)	Generator.Await
	Generator.Yield
Procedures (1)	Take.Call

Functions.Take.Call Procedure Takes the specified number of results from another function.

```
PROCEDURE- (VAR take: Take) Call*;
```

Functions.Take.count Field The number of results.

```
count*: LENGTH;
```

Functions.Take.function Field The function to take results from.

```
function*: POINTER TO VAR Function;
```

7.3.13 Generators Module

The module Generators provides restricted coroutines that transfer control back to their callers multiple times.

```
MODULE Generators IN OBL;
IMPORT Coroutines IN OBL;
```


Types (1) Generators.Generator

Generators.Generator Record Type Represents a coroutine that may return multiple times to its caller.

```
TYPE Generator* = RECORD* (Coroutines.Coroutine) END;
```

Base Types (1)	Coroutines.Coroutine
From Coroutines.Coroutine (2)	Coroutine.Call
	Coroutine.Transfer
Procedures (2)	Generator.Await
	Generator.Yield

Generators.Generator.Await Procedure Waits from a call to the generator to return. The result indicates whether the generator yielded and thus will return again.

```
PROCEDURE- (VAR generator: Generator) Await* (): BOOLEAN;
```

Generators.Generator.Yield Procedure Returns to the caller of the generator.

```
PROCEDURE- (VAR generator: Generator) Yield*;
```

7.3.14 Hashes Module

The module Hashes provides utility procedures to calculate and combine hashes of arbitrary values.

```
MODULE Hashes IN OBL;
```

Procedures (2)	Hashes.Combine
	Hashes.Hash

Hashes.Combine Procedure Combines two hashes.

```
PROCEDURE Combine* (
    VAR seed: LENGTH;
    hash: LENGTH
);
```

Hashes.Hash Procedure Calculates the hash of value.

```
PROCEDURE Hash* (
    VAR value-: ARRAY OF SYSTEM.BYTE
): LENGTH;
```

7.3.15 HashMaps Module

The generic module HashMaps provides a hash table that maps key of type Key to values of type Value.

```
MODULE HashMaps(Key*, Value*, Hash*) IN OBL;
IMPORT Iterators(Pairs(Key,Value).Pair) IN OBL;
IMPORT Lists(Pairs(Key,Value).Pair) IN OBL;
IMPORT Pairs(Key,Value) IN OBL;
```

Parameters (3)	HashMaps.Hash
	HashMaps.Key
	HashMaps.Value
Procedures (1)	HashMaps.Swap
Types (3)	HashMaps.Element
	HashMaps.Iterator
	HashMaps.Map

HashMaps.Element Type Alias Represents an element of a hash map.

```
TYPE Element* = Pairs(Key,Value).Pair;
```

HashMaps.Hash Parameter The hash function for keys in a hash map.

```
Hash* = (* GENERIC *);
```

HashMaps.Iterator Record Type Represents an iteration over a list.

```
TYPE Iterator* = RECORD- (Iterators(Pairs(Key,Value).Pair).Iterator)
END;
```

Base Types (1) `Iterators(Pairs(Key,Value).Pair).Iterator`
Procedures (1) `Iterator.GetNext`

HashMaps.Iterator.GetNext Procedure Returns the next element in the hash map iteration if available as indicated by the boolean result.

```
PROCEDURE- (VAR iterator: Iterator) GetNext* (
    VAR element: Element
): BOOLEAN;
```

HashMaps.Key Parameter The type of a key associated with an element in a hash map.

```
Key* = (* GENERIC *);
```

HashMaps.Map Record Type A hash map as to be created using `New` and disposed using `Dispose`.

```
TYPE Map* = RECORD- END;
```

Procedures (11) `Map.Add`
`Map.Clear`
`Map.Contains`
`Map.Dispose`
`Map.Find`
`Map.GetCapacity`
`Map.GetIterator`
`Map.GetLoadFactor`
`Map.GetSize`
`Map.Lookup`
`Map.New`

HashMaps.Map.Add Procedure Adds an element with the specified key and value to the hash map. This procedure may raise an exception of type `AllocationFailure`.

```
PROCEDURE- (VAR map: Map) Add* (
    key-: Key;
```

```
        value-: Value
    );
```

HashMaps.Map.Clear Procedure Removes all elements from the hash map.

```
PROCEDURE- (VAR map: Map) Clear*;
```

HashMaps.Map.Contains Procedure Returns whether the hash map contains an element with the specified key.

```
PROCEDURE- (VAR map-: Map) Contains* (
    key-: Key
): BOOLEAN;
```

HashMaps.Map.Dispose Procedure Disposes a previously created hash map. This procedure must be called when a hash map variable goes out of scope in order to deallocate its hash table.

```
PROCEDURE- (VAR map: Map) Dispose*;
```

HashMaps.Map.Find Procedure Returns an iterator for all elements with the specified key.

```
PROCEDURE- (VAR map-: Map) Find* (
    key-: Key;
    VAR iterator: Lists(Pairs(Key,Value).Pair).Iterator
);
```

HashMaps.Map.GetCapacity Procedure Returns the capacity of the hash map.

```
PROCEDURE- (VAR map-: Map) GetCapacity* (): LENGTH;
```

HashMaps.Map.GetIterator Procedure Returns an iterator for the hash map.

```
PROCEDURE- (VAR map-: Map) GetIterator* (
    VAR iterator: Iterator
);
```

HashMaps.Map.GetLoadFactor Procedure Returns the load factor of the hash map.

```
PROCEDURE- (VAR map-: Map) GetLoadFactor* (): REAL;
```

HashMaps.Map.GetSize Procedure Returns the number of elements in the hash map.

```
PROCEDURE- (VAR map-: Map) GetSize* (): LENGTH;
```

HashMaps.Map.Lookup Procedure Returns the value of the element with the specified key if available.

```
PROCEDURE- (VAR map-: Map) Lookup* (
    key-: Key;
    VAR value: Value
): BOOLEAN;
```

HashMaps.Map.New Procedure Creates a new hash map with the specified capacity. This procedure may raise an exception of type `AllocationFailure`.

```
PROCEDURE- (VAR map: Map) New* (
    capacity: LENGTH
);
```

HashMaps.Swap Procedure Exchanges the contents of a hash map with that of another one.

```
PROCEDURE Swap* (
    VAR first: Map;
    VAR second: Map
);
```

HashMaps.Value Parameter The type of a value associated with an element in a hash map.

```
Value* = (* GENERIC *);
```

7.3.16 In Module

The module `In` provides utility procedures to read values of basic type from the standard input stream.

```
MODULE In IN OBL;
```

Procedures (8)	`In.Char`
	`In.Ignore`
	`In.Integer`
	`In.Peek`
	`In.SkipLn`
	`In.SkipSpace`
	`In.SkipWhiteSpace`
	`In.String`
Variables (1)	`In.Done`

In.Char Procedure Reads a character value from the standard input stream.

```
PROCEDURE Char* (
    VAR char: CHAR
);
```

In.Done Variable A Boolean value indicating whether the last input operation was successful.

```
VAR Done-: BOOLEAN;
```

In.Ignore Procedure Reads a character from the standard input stream without returning its value.

```
PROCEDURE Ignore*;
```

In.Integer Procedure Reads an integer value from the standard input stream.

```
PROCEDURE Integer* (
    VAR value: INTEGER
);
```

In.Peek Procedure Returns the next available character without removing it from the standard input stream. If there are no more characters available the result value is `0X`.

```
PROCEDURE Peek* (): CHAR;
```

In.SkipLn Procedure Skips the contents of the current line until the next new-line character.

```
PROCEDURE SkipLn*;
```

In.SkipSpace Procedure Skips all space characters. This does not include new-line characters.

```
PROCEDURE SkipSpace*;
```

In.SkipWhiteSpace Procedure Skips all white-space characters. This includes new-line characters.

```
PROCEDURE SkipWhiteSpace*;
```

In.String Procedure Reads a string value from the standard input stream. The number of read characters equals the length of the string but does not exceed the array length.

```
PROCEDURE String* (
    VAR string: ARRAY OF CHAR
);
```

7.3.17 IOStreams Module

The module `IOStreams` provides a standard I/O streams.

```
MODULE IOStreams IN OBL;
IMPORT Devices IN OBL;
```

Variables (3)	IOStreams.stderr
	IOStreams.stdin
	IOStreams.stdout

IOStreams.stderr Variable The standard error stream.

```
VAR stderr*: Devices.Device;
```

IOStreams.stdin Variable The standard input stream.

```
VAR stdin*: Devices.Device;
```

IOStreams.stdout Variable The standard output stream.

```
VAR stdout*: Devices.Device;
```

7.3.18 Iterators Module

The generic module `Iterators` provides an abstract type for iterating over containers that store elements of type `Element`.

```
MODULE Iterators(Element*) IN OBL;
```


Parameters (1)	Iterators.Element
Types (1)	Iterators.Iterator

Iterators.Element Parameter The type of an element returned by an iterator.

```
Element* = (* GENERIC *);
```

Iterators.Iterator Record Type Represents an iteration over a container.

```
TYPE Iterator* = RECORD* END;
```

Procedures (1)	Iterator.GetNext

Iterators.Iterator.GetNext Procedure Returns the next element in the iteration if available as indicated by the boolean result.

```
PROCEDURE* (VAR iterator: Iterator) GetNext* (
    VAR element: Element
): BOOLEAN;
```

7.3.19 Lists Module

The generic module `Lists` provides an iterator type for iterating over lists of type `Element`.

```
MODULE Lists(Element*) IN OBL;
IMPORT Iterators(Element) IN OBL;
```

Parameters (1)	`Lists.Element`
Procedures (1)	`Lists.Swap`
Types (2)	`Lists.Iterator`
	`Lists.List`

Lists.Element Parameter The type of an element contained in a list.

```
Element* = (* GENERIC *);
```

Lists.Iterator Record Type Represents an iteration over a list.

```
TYPE Iterator* = RECORD- (Iterators(Element).Iterator) END;
```

Base Types (1)	`Iterators(Element).Iterator`
Procedures (1)	`Iterator.GetNext`

Lists.Iterator.GetNext Procedure Returns the next element in the list iteration if available as indicated by the boolean result.

```
PROCEDURE- (VAR iterator: Iterator) GetNext* (
    VAR element: Element
): BOOLEAN;
```

Lists.List Record Type Represents a list container.

```
TYPE List* = RECORD- END;
```

Procedures (8)	`List.Add`
	`List.Clear`
	`List.GetElement`
	`List.GetIterator`
	`List.GetSize`
	`List.IsEmpty`
	`List.Reverse`
	`List.SetElement`

Lists.List.Add Procedure Appends the specified element to the end of the list. This procedure may raise an exception of type `AllocationFailure`.

```
PROCEDURE- (VAR list: List) Add* (
    element-: Element
);
```

Lists.List.Clear Procedure Removes all elements from the list. This procedure must be called when a list variable goes out of scope in order to deallocate its elements.

```
PROCEDURE- (VAR list: List) Clear*;
```

Lists.List.GetElement Procedure Returns the element at the specified position in the list.

```
PROCEDURE- (VAR list-: List) GetElement* (
    index: LENGTH
): Element;
```

Lists.List.GetIterator Procedure Returns an iterator for the list.

```
PROCEDURE- (VAR list-: List) GetIterator* (
    VAR iterator: Iterator
);
```

Lists.List.GetSize Procedure Returns the size of the list.

```
PROCEDURE- (VAR list-: List) GetSize* (): LENGTH;
```

Lists.List.IsEmpty Procedure Returns whether the list is empty.

```
PROCEDURE- (VAR list-: List) IsEmpty* (): BOOLEAN;
```

Lists.List.Reverse Procedure Reverses the order of the elements of the list.

```
PROCEDURE- (VAR list: List) Reverse*;
```

Lists.List.SetElement Procedure Replaces the element at the specified position in the list.

```
PROCEDURE- (VAR list: List) SetElement* (
    index: LENGTH;
    element-: Element
);
```

Lists.Swap Procedure Exchanges the contents of a list with that of another one.

```
PROCEDURE Swap* (
    VAR first: List;
    VAR second: List
);
```

7.3.20 Math Module

The generic module `Math` provides mathematical functions on values of type `Real`.

```
MODULE Math(Real*) IN OBL;
```

Constants (3)	`Math.E`
	`Math.Pi`
	`Math.Tau`
Parameters (1)	`Math.Real`
Procedures (16)	`Math.Abs`
	`Math.CopySign`
	`Math.Cos`
	`Math.Fraction`
	`Math.Integer`
	`Math.IsFinite`
	`Math.IsInf`
	`Math.IsNan`
	`Math.IsNegative`
	`Math.IsNormal`
	`Math.Ma`
	`Math.Max`
	`Math.Min`
	`Math.Sin`
	`Math.Sqrt`
	`Math.Tan`

Math.Abs Procedure Computes the absolute value of `x`.

```
PROCEDURE Abs* (
    x: Real
): Real;
```

Math.CopySign Procedure Composes a value with the magnitude of `m` and the sign of `s`.

```
PROCEDURE CopySign* (
    m: Real;
    s: Real
): Real;
```

Math.Cos Procedure Computes the cosine of `x`.

```
PROCEDURE Cos* (
    x: Real
): Real;
```

Math.E Constant The base of natural logarithms.

```
CONST E* = (* GENERIC *);
```

Math.Fraction Procedure Returns the fractional part of x.

```
PROCEDURE Fraction* (
    x: Real
): Real;
```

Math.Integer Procedure Returns the integral part of x.

```
PROCEDURE Integer* (
    x: Real
): Real;
```

Math.IsFinite Procedure Determines whether x has a finite value.

```
PROCEDURE IsFinite* (
    x: Real
): BOOLEAN;
```


Math.IsInf Procedure Determines whether x is an infinity.

```
PROCEDURE IsInf* (
    x: Real
): BOOLEAN;
```

Math.IsNan Procedure Determines whether x is not a number.

```
PROCEDURE IsNan* (
    x: Real
): BOOLEAN;
```

Math.IsNegative Procedure Determines whether x has a negative value.

```
PROCEDURE IsNegative* (
    x: Real
): BOOLEAN;
```

Math.IsNormal Procedure Determines whether x has a normal value.

```
PROCEDURE IsNormal* (
    x: Real
): BOOLEAN;
```

Math.Ma Procedure Computes the fused multiplication and accumulation of x, y, and z.

```
PROCEDURE Ma* (
    x: Real;
    y: Real;
    z: Real
): Real;
```

Math.Max Procedure Computes the maximum of x and y.

```
PROCEDURE Max* (
    x: Real;
    y: Real
): Real;
```

Math.Min Procedure Computes the minimum of x and y.

```
PROCEDURE Min* (
    x: Real;
    y: Real
): Real;
```

Math.Pi Constant The ratio of the circumference of a circle to its diameter.

```
CONST Pi* = (* GENERIC *);
```

Math.Real Parameter The real type used for mathematical functions.

```
Real* = (* GENERIC *);
```

Math.Sin Procedure Computes the sine of x.

```
PROCEDURE Sin* (
    x: Real
): Real;
```

Math.Sqrt Procedure Computes the square root of x.

```
PROCEDURE Sqrt* (
    x: Real
): Real;
```

Math.Tan Procedure Computes the tangent of x.

```
PROCEDURE Tan* (
    x: Real
): Real;
```

Math.Tau Constant The ratio of the circumference of a circle to its radius.

```
CONST Tau* = (* GENERIC *);
```

7.3.21 Out Module

The module `Out` provides utility procedures to print values of basic type to the standard output stream.

```
MODULE Out IN OBL;
```

Procedures (14)	`Out.Address`
	`Out.Boolean`
	`Out.Byte`
	`Out.Char`
	`Out.Complex`
	`Out.Integer`
	`Out.Ln`
	`Out.Pointer`
	`Out.Procedure`
	`Out.Real`
	`Out.Set`
	`Out.Signed`
	`Out.String`
	`Out.Unsigned`
Variables (1)	`Out.Done`

Out.Address Procedure Prints a hexademical address value to the standard output stream.

```
PROCEDURE Address* (
    value: SYSTEM.ADDRESS
);
```

Out.Boolean Procedure Prints a Boolean value to the standard output stream.

```
PROCEDURE Boolean* (
    value: BOOLEAN
);
```

Out.Byte Procedure Prints a hexademical byte value to the standard output stream.

```
PROCEDURE Byte* (
    value: SYSTEM.BYTE
);
```

Out.Char Procedure Prints a character value to the standard output stream.

```
PROCEDURE Char* (
    value: CHAR
);
```

Out.Complex Procedure Prints a complex value to the standard output stream.

```
PROCEDURE Complex* (
    value: LONGCOMPLEX;
    precision: INTEGER
);
```

Out.Done Variable A Boolean value indicating whether the last output operation was successful.

```
VAR Done-: BOOLEAN;
```

Out.Integer Procedure Prints a decimal integer value to the standard output stream.

```
PROCEDURE Integer* (
    value: HUGEINT
);
```

Out.Ln Procedure Prints a new-line character to the standard output stream.

```
PROCEDURE Ln*;
```

Out.Pointer Procedure Prints a pointer value to the standard output stream.

```
PROCEDURE Pointer* (
    value: SYSTEM.PTR
);
```

Out.Procedure Procedure Prints a procedure value to the standard output stream.

```
PROCEDURE Procedure* (
    value: PROCEDURE
);
```

Out.Real Procedure Prints a real value to the standard output stream.

```
PROCEDURE Real* (
    value: LONGREAL;
    precision: INTEGER
);
```

Out.Set Procedure Prints a set value to the standard output stream.

```
PROCEDURE Set* (
    value: SET64
);
```

Out.Signed Procedure Prints a signed integer value to the standard output stream.

```
PROCEDURE Signed* (
    value: SIGNED64
);
```

Out.String Procedure Prints a string value to the standard output stream.

```
PROCEDURE String* (
    value-: ARRAY OF CHAR
);
```

Out.Unsigned Procedure Prints an unsigned integer value to the standard output stream.

```
PROCEDURE Unsigned* (
    value: UNSIGNED64
);
```

7.3.22 Pairs Module

The generic module `pairs` provides a way of storing a pair of values with different type.

```
MODULE Pairs(First*, Second*) IN OBL;
```

Parameters (2)	`Pairs.First`
	`Pairs.Second`
Procedures (1)	`Pairs.Make`
Types (1)	`Pairs.Pair`

Pairs.First Parameter The type of the first value in a pair.

```
First* = (* GENERIC *);
```

Pairs.Make Procedure Creates a pair with the given values.

```
PROCEDURE Make* (
    first-: First;
    second-: Second
): Pair;
```

Pairs.Pair Record Type Represents a pair of values.

```
TYPE Pair* = RECORD-
    first*: First;
    second*: Second;
END;
```

Fields (2) `Pair.first`
`Pair.second`

Pairs.Pair.first Field `first*: First;`

Pairs.Pair.second Field `second*: Second;`

Pairs.Second Parameter The type of the second value in a pair.

```
Second* = (* GENERIC *);
```

7.3.23 Random Module

The module `Random` provides several functions for generating uniform pseudo-random numbers. Each function requires a *seed* which defines the next number in the predefined sequence of the underlying random number generator.

```
MODULE Random IN OBL;
```

Procedures (5) `Random.Create`
`Random.Dice`
`Random.Initialize`
`Random.Integer`
`Random.Uniform`
Types (1) `Random.Seed`

Random.Create Procedure Creates an arbitrary random value.

```
PROCEDURE Create* (
    VAR seed: Seed;
    VAR value: ARRAY OF SYSTEM.BYTE
);
```

Random.Dice Procedure Returns the value of a roll of a uniform dice with the specified number of sides.

```
PROCEDURE Dice* (
    VAR seed: Seed;
    sides: INTEGER
): INTEGER;
```

Random.Initialize Procedure Initializes the seed of the random number generator with the specified value.

```
PROCEDURE Initialize* (
    VAR seed: Seed;
    value: INTEGER
);
```

Random.Integer Procedure Returns a uniform random integer number between zero and `MAX(INTEGER)`.

```
PROCEDURE Integer* (
    VAR seed: Seed
): INTEGER;
```

Random.Seed Record Type Represents the current seed for the next number generated by the random number generator.

```
TYPE Seed* = RECORD- END;
```

Random.Uniform Procedure Returns a uniformly distributed random real number between zero and one.

```
PROCEDURE Uniform* (
    VAR seed: Seed
): REAL;
```


7.3.24 Sets Module

The generic module `Sets` provides set types of arbitrary sizes.

```
MODULE Sets(Size*) IN OBL;
```

Constants (2)	`Sets.Maximum`
	`Sets.Minimum`
Parameters (1)	`Sets.Size`
Procedures (1)	`Sets.Swap`
Types (1)	`Sets.Set`

Sets.Maximum Constant The maximal element of a set.

```
CONST Maximum* = (* GENERIC *);
```

Sets.Minimum Constant The minimal element of a set.

```
CONST Minimum* = (* GENERIC *);
```

Sets.Set Record Type Set of integers between `Minimum` and `Maximum`.

```
TYPE Set* = RECORD- END;
```

Procedures (13)	`Set.Add`
	`Set.Clear`
	`Set.Complement`
	`Set.Count`
	`Set.Differ`
	`Set.Equals`
	`Set.Exclude`
	`Set.Include`
	`Set.Intersect`
	`Set.IsEmpty`
	`Set.Set`
	`Set.Subtract`
	`Set.Test`

Sets.Set.Add Procedure Computes the union of the set with another set.

```
PROCEDURE- (VAR set: Set) Add* (
    other-: Set
);
```

Sets.Set.Clear Procedure Removes all integers from the set.

```
PROCEDURE- (VAR set: Set) Clear*;
```

Sets.Set.Complement Procedure Computes the complement of the set.

```
PROCEDURE- (VAR set: Set) Complement*;
```

Sets.Set.Count Procedure Counts all integers in the set.

```
PROCEDURE- (VAR set-: Set) Count* (): LENGTH;
```

Sets.Set.Differ Procedure Computes the symmetric difference of the set with another set.

```
PROCEDURE- (VAR set: Set) Differ* (
    other-: Set
);
```

Sets.Set.Equals Procedure Returns whether the set equals another set.

```
PROCEDURE- (VAR set-: Set) Equals* (
    other-: Set
): BOOLEAN;
```

Sets.Set.Exclude Procedure Excludes an integer from the set.

```
PROCEDURE- (VAR set: Set) Exclude* (
    element: LENGTH
);
```

Sets.Set.Include Procedure Includes an integer in the set.

```
PROCEDURE- (VAR set: Set) Include* (
    element: LENGTH
);
```

Sets.Set.Intersect Procedure Computes the intersection of the set with another set.

```
PROCEDURE- (VAR set: Set) Intersect* (
    other-: Set
);
```

Sets.Set.IsEmpty Procedure Returns whether the set is empty.

```
PROCEDURE- (VAR set-: Set) IsEmpty* (): BOOLEAN;
```

Sets.Set.Set Procedure Adds all integers to the set.

```
PROCEDURE- (VAR set: Set) Set*;
```


Sets.Set.Subtract Procedure Computes the difference of the set with another set.

```
PROCEDURE- (VAR set: Set) Subtract* (
    other-: Set
);
```

Sets.Set.Test Procedure Returns whether an integer is element of the set.

```
PROCEDURE- (VAR set-: Set) Test* (
    element: LENGTH
): BOOLEAN;
```

Sets.Size Parameter The maximal number of different integers in a set.

```
Size* = (* GENERIC *);
```

Sets.Swap Procedure Exchanges the contents of a set with that of another one.

```
PROCEDURE Swap* (
    VAR first: Set;
    VAR second: Set
);
```

7.3.25 Streams Module

The module `Streams` provides abstract stream types for reading and writing characters of type `Character`.

```
MODULE Streams(Character*) IN OBL;
```

Parameters (1)	`Streams.Character`
Types (2)	`Streams.Reader`
	`Streams.Writer`

Streams.Character Parameter The character type used for input and output.

```
Character* = (* GENERIC *);
```

Streams.Reader Record Type Represents an input stream reader.

```
TYPE Reader* = RECORD* END;
```

Procedures (1)	`Reader.Read`

Streams.Reader.Read Procedure Reads a value of type `Character` from the stream and returns whether that operation succeeded.

```
PROCEDURE* (VAR reader: Reader) Read* (
    VAR value: Character
): BOOLEAN;
```

Streams.Writer Record Type Represents an output stream writer.

```
TYPE Writer* = RECORD* END;
```

Procedures (5)	`Writer.Boolean`
	`Writer.Char`
	`Writer.Ln`
	`Writer.String`
	`Writer.Write`

Streams.Writer.Boolean Procedure Writes a Boolean value to the stream.

```
PROCEDURE- (VAR writer: Writer) Boolean* (
    value: BOOLEAN
);
```

Streams.Writer.Char Procedure Writes a character value to the stream.

```
PROCEDURE- (VAR writer: Writer) Char* (
    value: CHAR
);
```

Streams.Writer.Ln Procedure Writes a new-line character to the stream.

```
PROCEDURE- (VAR writer: Writer) Ln*;
```

Streams.Writer.String Procedure Writes a string value to the stream.

```
PROCEDURE- (VAR writer: Writer) String* (
    value-: ARRAY OF CHAR
);
```

Streams.Writer.Write Procedure Writes a value of type `Character` to the stream.

```
PROCEDURE* (VAR writer: Writer) Write* (
    value: Character
);
```

7.3.26 Strings Module

The generic module `Strings` provides utility procedures for strings that are represented by arrays of type `Char`. All of the following procedures operate on the length of a string which is equal to the index of the first null character in the character array.

```
MODULE Strings(Char*) IN OBL;
```

Parameters (1)	`Strings.Char`
Procedures (7)	`Strings.Compare`
	`Strings.Concatenate`
	`Strings.Copy`
	`Strings.FindCharacter`
	`Strings.GetLength`
	`Strings.IsEmpty`
	`Strings.Truncate`

Strings.Char Parameter The element type of a character array.

```
Char* = (* GENERIC *);
```

Strings.Compare Procedure This procedure compares two strings lexicographically. It returns `-1` if the first string is lexicographically less than the second one. It returns `+1` if

the first string is lexicographically greater than the second one. If they compare equal, this procedure returns zero.

```
PROCEDURE Compare* (
    string1-: ARRAY OF Char;
    string2-: ARRAY OF Char
): INTEGER;
```

Strings.Concatenate Procedure Appends a string to another one. The length of the destination array may not be less than the length of the string it contains plus the length of the appendix.

```
PROCEDURE Concatenate* (
    VAR string: ARRAY OF Char;
    appendix-: ARRAY OF Char
);
```

Strings.Copy Procedure Copies some characters of a string into another one starting at a given position in the source string. The length of the destination array may not be less than the given length.

```
PROCEDURE Copy* (
    source-: ARRAY OF Char;
    VAR dest: ARRAY OF Char;
    pos: LENGTH;
    length: LENGTH
);
```

Strings.FindCharacter Procedure Returns `TRUE` if it finds a certain character in a string beginning at a given position.

```
PROCEDURE FindCharacter* (
    value: Char;
    string-: ARRAY OF Char;
    VAR pos: LENGTH
): BOOLEAN;
```

Strings.GetLength Procedure Returns the length of the string. The length of a string equals to the index of the first null character in the character array.

```
PROCEDURE GetLength* (
    string-: ARRAY OF Char
): LENGTH;
```

Strings.IsEmpty Procedure Returns `TRUE` if the length of the string is zero.

```
PROCEDURE IsEmpty* (
    string-: ARRAY OF Char
): BOOLEAN;
```

Strings.Truncate Procedure Truncates the contents of a string by shortening its length.

```
PROCEDURE Truncate* (
    VAR string: ARRAY OF Char;
    length: LENGTH
);
```

7.3.27 Trees Module

The generic module `Trees` provides an iterator type for iterating over binary trees of type `Element`.

```
MODULE Trees(Element*) IN OBL;
IMPORT Iterators(Element) IN OBL;
```

Parameters (1)	`Trees.Element`
Procedures (1)	`Trees.Swap`
Types (3)	`Trees.Iterator`
	`Trees.Node`
	`Trees.Tree`

Trees.Element Parameter The type of an element contained in a binary tree.

```
Element* = (* GENERIC *);
```

Trees.Iterator Record Type Represents an iteration over a binary tree.

```
TYPE Iterator* = RECORD- (Iterators(Element).Iterator) END;
```

Base Types (1)	`Iterators(Element).Iterator`
Procedures (1)	`Iterator.GetNext`

Trees.Iterator.GetNext Procedure Returns the next element in the binary tree iteration if available as indicated by the boolean result.

```
PROCEDURE- (VAR iterator: Iterator) GetNext* (
    VAR element: Element
): BOOLEAN;
```

Trees.Node Type Alias Represents a tree node.

```
TYPE Node* = POINTER TO RECORD- END;
```

Trees.Swap Procedure Exchanges the contents of a binary tree with that of another one.

```
PROCEDURE Swap* (
    VAR first: Tree;
    VAR second: Tree
);
```

Trees.Tree Record Type Represents a binary tree container.

```
TYPE Tree* = RECORD- END;
```

Procedures (7)	`Tree.Clear`
	`Tree.Find`
	`Tree.GetIterator`
	`Tree.GetSize`
	`Tree.Insert`
	`Tree.IsEmpty`
	`Tree.Remove`

Trees.Tree.Clear Procedure Removes all elements from the tree. This procedure must be called when a tree variable goes out of scope in order to deallocate its nodes.

```
PROCEDURE- (VAR tree: Tree) Clear*;
```

Trees.Tree.Find Procedure Returns the tree node storing the specified element.

```
PROCEDURE- (VAR tree-: Tree) Find* (
    element-: Element
): Node;
```

Trees.Tree.GetIterator Procedure Returns an iterator for the binary tree.

```
PROCEDURE- (VAR tree-: Tree) GetIterator* (
    VAR iterator: Iterator
);
```

Trees.Tree.GetSize Procedure Returns the size of the binary tree.

```
PROCEDURE- (VAR tree-: Tree) GetSize* (): LENGTH;
```

Trees.Tree.Insert Procedure Inserts the specified element to the binary tree. This procedure may raise an exception of type `AllocationFailure`.

```
PROCEDURE- (VAR tree: Tree) Insert* (
    element-: Element
);
```

Category	Module	Section
Basic Modules	`In`	7.4.2
	`Math`	7.4.3
	`MathL`	7.4.5
	`Out`	7.4.7
	`Strings`	7.4.8
	`XYplane`	7.4.9
Additional Modules	`Coroutines`	7.4.1
	`MathC`	7.4.4
	`MathLC`	7.4.6

Table 7.58: Modules of the Oakwood Library

Trees.Tree.IsEmpty Procedure Returns whether the tree is empty.

```
PROCEDURE- (VAR tree-: Tree) IsEmpty* (): BOOLEAN;
```

Trees.Tree.Remove Procedure Removes the specified node from the binary tree.

```
PROCEDURE- (VAR tree: Tree) Remove* (
    node: Node
);
```


7.4 The Oakwood Library

For compatibility with other implementations, the Eigen Compiler Suite provides additional library modules recommended by the Oakwood guidelines for Oberon-2 compiler developers [9]. This library is called the *Oakwood Library* and is available by importing one or more of the modules listed in Table 7.58. All of these modules are governed by the ECS Runtime Support Exception which is an additional permission to the GNU General Public License that allows users of the Eigen Compiler Suite to create proprietary programs. Copies of these licenses are included in Appendices C and D on pages 495 and 509 respectively. The remainder of this section lists all supported modules of the Oakwood Library in alphabetical order and describes their exported interface.

7.4.1 Coroutines Module

Module Coroutines provides non-preemptive threads each with its own stack but otherwise sharing a common address space. Coroutines can explicitly transfer control to other coroutines which are then resumed from the point where they did their last transfer of control.

```
MODULE Coroutines;
```

Procedures (2)	`Coroutines.Init`
	`Coroutines.Transfer`
Types (2)	`Coroutines.Body`
	`Coroutines.Coroutine`

Coroutines.Body Procedure Type Represents the body of a coroutine.

```
TYPE Body* = PROCEDURE;
```

Coroutines.Coroutine Record Type Represents a coroutine.

```
TYPE Coroutine* = RECORD END;
```

Coroutines.Init Procedure Creates and initialises a new coroutine with a specified stack size and body provided as a procedure. An initialised coroutine can be started by a `Transfer` to it. In this case its execution will start at the first instruction of its body which must never return.

```
PROCEDURE Init* (
    body: Body;
    stackSize: LONGINT;
    VAR cor: Coroutine
);
```

Coroutines.Transfer Procedure Transfers control from the currently executing coroutine to another one. The state of the currently executing coroutine is saved. When control is transferred back later, the coroutine will be restarted in the saved state.

```
PROCEDURE Transfer* (
    VAR from: Coroutine;
    VAR to: Coroutine
);
```

7.4.2 In Module

The module `In` provides a set of basic routines for formatted input of characters, character sequences, numbers, and names. All operations except `Char` skip leading blanks, tabs or end-of-line characters.

```
MODULE In;
```

Procedures (8)	`In.Char`

	In.Int
	In.LongInt
	In.LongReal
	In.Name
	In.Open
	In.Real
	In.String
Variables (1)	In.Done

In.Char Procedure Returns the character at the current position.

```
PROCEDURE Char* (
    VAR ch: CHAR
);
```

In.Done Variable Indicates the success of the last input operation.

```
VAR Done-: BOOLEAN;
```

In.Int Procedure Returns the integer constant at the current position.

```
PROCEDURE Int* (
    VAR i: INTEGER
);
```


In.LongInt Procedure Returns the long integer constant at the current position.

```
PROCEDURE LongInt* (
    VAR i: LONGINT
);
```

In.LongReal Procedure Returns the long real constant at the current position.

```
PROCEDURE LongReal* (
    VAR x: LONGREAL
);
```

In.Name Procedure Returns the name at the current position.

```
PROCEDURE Name* (
    VAR name: ARRAY OF CHAR
);
```

In.Open Procedure Sets the current position to the beginning of the input stream.

```
PROCEDURE Open*;
```

In.Real Procedure Returns the real constant at the current position.

```
PROCEDURE Real* (
    VAR x: REAL
);
```

In.String Procedure Returns the string at the current position.

```
PROCEDURE String* (
    VAR str: ARRAY OF CHAR
);
```

7.4.3 Math Module

The module `Math` provides a basic set of general purpose functions using `REAL` arithmetic.

```
MODULE Math;
```

Constants (2)	`Math.e`
	`Math.pi`
Procedures (1)	`Math.sqrt`

Math.e Constant The base of natural logarithms.

```
CONST e* = (* REAL *);
```

Math.pi Constant The ratio of the circumference of a circle to its diameter.

```
CONST pi* = (* REAL *);
```

Math.sqrt Procedure Returns the square root of x, where x must be positive.

```
PROCEDURE sqrt* (
    x: REAL
): REAL;
```

7.4.4 MathC Module

The module `MathC` provides a basic set of general purpose functions using `COMPLEX` arithmetic.

```
MODULE MathC;
```

Procedures (1)	`MathC.abs`

MathC.abs Procedure Returns the absolute value or magnitude of x.

```
PROCEDURE abs* (
    x: COMPLEX
): REAL;
```

7.4.5 MathL Module

The module MathL provides a basic set of general purpose functions using LONGREAL arithmetic.

```
MODULE MathL;
```

Constants (2)	MathL.e
	MathL.pi
Procedures (1)	MathL.sqrt

MathL.e Constant The base of natural logarithms.

```
CONST e* = (* LONGREAL *);
```

MathL.pi Constant The ratio of the circumference of a circle to its diameter.

```
CONST pi* = (* LONGREAL *);
```

MathL.sqrt Procedure Returns the square root of x, where x must be positive.

```
PROCEDURE sqrt* (
    x: LONGREAL
): LONGREAL;
```


7.4.6 MathLC Module

The module MathLC provides a basic set of general purpose functions using LONGCOMPLEX arithmetic.

```
MODULE MathLC;
```

Procedures (1)	MathLC.abs

MathLC.abs Procedure Returns the absolute value or magnitude of x.

```
PROCEDURE abs* (
    x: LONGCOMPLEX
): LONGREAL;
```

7.4.7 Out Module

The module Out provides a set of basic routines for formatted output of characters, numbers, and strings.

```
MODULE Out;
```

Procedures (7)	`Out.Char`
	`Out.Int`
	`Out.Ln`
	`Out.LongReal`
	`Out.Open`
	`Out.Real`
	`Out.String`

Out.Char Procedure Writes the character to the end of the output stream.

```
PROCEDURE Char* (
    ch: CHAR
);
```

Out.Int Procedure Writes the integer number to the end of the output stream padded with the specified number of spaces at the left end.

```
PROCEDURE Int* (
    i: LONGINT;
    n: LONGINT
);
```

Out.Ln Procedure Writes an end-of-line character to the end of the output stream.

```
PROCEDURE Ln*;
```

Out.LongReal Procedure Writes the long real number to the end of the output stream using an exponential form padded with the specified number of spaces at the left end.

```
PROCEDURE LongReal* (
    x: LONGREAL;
    n: INTEGER
);
```

Out.Open Procedure Initializes the output stream.

```
PROCEDURE Open*;
```

Out.Real Procedure Writes the real number to the end of the output stream using an exponential form padded with the specified number of spaces at the left end.

```
PROCEDURE Real* (
    x: REAL;
    n: INTEGER
);
```

Out.String Procedure Writes the null-terminated character sequence to the end of the output stream.

```
PROCEDURE String* (
    str: ARRAY OF CHAR
);
```

7.4.8 Strings Module

Module `Strings` provides a set of operations on string constants and character arrays, both of which contain the character `0X` as a terminator.

```
MODULE Strings;
```

Procedures (2)	`Strings.Cap`
	`Strings.Length`

Strings.Cap Procedure Replaces each lower case letter within *s* by its upper case equivalent.

```
PROCEDURE Cap* (
    VAR s: ARRAY OF CHAR
);
```

Strings.Length Procedure Returns the number of characters in *s* up to and excluding the first 0X.

```
PROCEDURE Length* (
    s: ARRAY OF CHAR
): INTEGER;
```

7.4.9 XYplane Module

Module `XYplane` provides basic facilities for graphics programming. It provides a Cartesian plane of pixels that can be drawn and erased. The plane is mapped to some location on the screen.

```
MODULE XYplane;
```

Constants (2)	`XYplane.draw`
	`XYplane.erase`
Procedures (6)	`XYplane.Clear`
	`XYplane.Close`
	`XYplane.Dot`
	`XYplane.IsDot`

	`XYplane.Key`
	`XYplane.Open`
Variables (4)	`XYplane.H`
	`XYplane.W`
	`XYplane.X`
	`XYplane.Y`

XYplane.Clear Procedure Erases all pixels in the drawing plane.

```
PROCEDURE Clear*;
```

XYplane.Close Procedure Terminates the drawing plane.

```
PROCEDURE Close*;
```

XYplane.Dot Procedure Draws or erases the pixel at the coordinates (x, y) relative to the lower left corner of the plane. *mode* must be either `draw` or `erase`.

```
PROCEDURE Dot* (
    x: INTEGER;
    y: INTEGER;
    mode: INTEGER
);
```

XYplane.H Variable Height of the screen.

```
VAR H-: INTEGER;
```

XYplane.IsDot Procedure Returns whether the pixel at the coordinates (x, y) relative to the lower left corner of the screen is drawn.

```
PROCEDURE IsDot* (
    x: INTEGER;
    y: INTEGER
): BOOLEAN;
```

XYplane.Key Procedure Reads the keyboard. If a key was pressed prior to invocation, its character value is returned, otherwise the result is `0X`.

```
PROCEDURE Key* (): CHAR;
```

XYplane.Open Procedure Initializes the drawing plane.

```
PROCEDURE Open*;
```

XYplane.W Variable Width of the screen.

```
VAR W-: INTEGER;
```

XYplane.X Variable Left border of the screen.

```
VAR X-: INTEGER;
```

XYplane.Y Variable Lower border of the screen.

```
VAR Y-: INTEGER;
```

XYplane.draw Constant Draw pixel mode.

```
CONST draw* = (* SIGNED8 *);
```

XYplane.erase Constant Erase pixel mode.

```
CONST erase* = (* SIGNED8 *);
```

7.5 Documentation Generation

The Eigen Compiler Suite provides several tools that are able to extract the structure of modules written in Oberon and generate documentations for them. This section describes the contents of the extracted information and explains how programmers can provide a user-defined description of it. An example of a completely annotated module is given in Figure 7.1 at the end of this section.

7.5.1 Annotations

Programmers can annotate their program using a special notation for comments. Comments beginning with two asterisks are still ignored during parsing but recognized and merged into a single *annotation* for the immediately following syntax element. Annotations can be formatted using a lightweight markup language which is an extension of the Creole markup language used for formatting wikis [10]. Table 25.1 on page 456 categorizes all elements of this markup language and shows how they are formatted. For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

In addition to annotations for syntax elements like modules and procedures, the very last annotation after the end of a module is also recognized. The documentation for a module begins with the user-defined content consisting of paragraphs and articles defined in this annotation. Afterward there is an article for the module and each of its exported declarations. Since all these articles are predefined, the article markup has a different meaning therein and allows tagging an element of the article with further information. These so-called *tags* begin with one at sign for general-purpose tags and with two at signs for descriptions of nested syntax elements.

The following sections define how these articles look like and what tags are available therein. The tag `@remarks` is always available and allows giving more detailed information at the end of an article.

7.5.2 Modules

A module can be annotated in front of the MODULE keyword. The generated article is labeled with the name and optional package of the module and contains its description as well as a summary of all exported declarations. The summary consists of a table that groups these declarations by topic which can be defined using the @topic tag. It defaults to the type of the declaration if no tag is given. If the module is generic, the article additionally provides a fully qualified label for each exported parameter and a nested tag for its description:

```
(** description for a module *)
(** @@Value its first parameter *)
(** @remarks further information about the module *)
MODULE Module (Value*);
END Module.
(** the documentation starts with this description *)
(** @ and may contain user-defined articles *)
(** as well as links to the [[Module]] *)
```

7.5.3 Constant Declarations

A constant can be annotated in front of its identifier or the preceding CONST keyword. The generated article is labeled with the fully qualified name of the constant and contains its description as well as its syntax definition:

```
(** description of a constant *)
(** @topic grouping of the constant *)
(** @remarks further information about the constant *)
CONST Pi* = 3.1415;
```

7.5.4 Type Declarations

A type can be annotated in front of its identifier or the preceding TYPE keyword. The generated article is labeled with the fully qualified name of the type and contains its description as well as its syntax definition. For record type declarations it additionally contains a summary of the record interface similar to the summary of the module. Fields can be annotated like variables:

```
(** description of a record *)
TYPE Record* = RECORD
  (** description of the first field *)
  field-: Number;
END;
```

7.5.5 Variable Declarations

A variable can be annotated in front of its identifier or the preceding VAR keyword. The generated article is labeled with the fully qualified name of the variable and contains its description as well as its syntax definition:

```
(** description of a variable *)
VAR features*: SET;
```

7.5.6 Procedure Declarations

A procedure can be annotated in front of the PROCEDURE keyword. The generated article is labeled with the fully qualified name of the procedure and contains its description as well as its syntax definition. It additionally provides a fully qualified label and a nested tag for the description of each parameter and the optional receiver. Function procedures have an additional tag called @result that allows describing the result value:

```
(** description for a procedure *)
(** @@number its first parameter *)
(** @result and its result value *)
PROCEDURE Increment* (VAR number: Number): Number;
BEGIN INC (number, Value); RETURN number;
END Increment;
```


7.6 Runtime Support

Some language features of Oberon such as predeclared procedures require some additional runtime support. This runtime support is stored in library files which are combinations of object files. For more information about object files and their purpose, see Chapter 22. The Eigen Compiler Suite provides the required runtime support in one library file for each hardware architecture it supports. The name of the corresponding library file consists of a leading ob, the name of the target hardware architecture, and a trailing run as in obarma64run.

The procedures of the module Math described in Section 7.3.20 are wrappers for the corresponding functions of the Standard C++ Library. Programs using these procedures therefore require additional runtime support for C++. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

```
(** description for a module *)
(** @@Value its first parameter *)
(** @remarks further information about the module *)
MODULE Module (Value*);

(** description of a constant *)
(** @topic grouping of the constant *)
(** @remarks further information about the constant *)
CONST Pi* = 3.1415;

TYPE
  (** description of a type declaration *)
  Number* = INTEGER;

  (** description of a record *)
  Record* = RECORD
    (** description of the first field *)
    field-: Number;
  END;

(** description of a variable *)
VAR features*: SET;

(** description for a procedure *)
(** @@number its first parameter *)
(** @result and its result value *)
PROCEDURE Increment* (VAR number: Number): Number;
BEGIN INC (number, Value); RETURN number;
END Increment;

END Module.
(** the documentation starts with this description *)
(** @ and may contain user-defined articles *)
(** as well as links to the [[Module]] *)
```

Figure 7.1: Example of a completely annotated Oberon module

7.7 Oberon Tools

The Eigen Compiler Suite provides several different tools that process modules written in Oberon. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools. For basic examples of using some of these tools in practice, see Chapter 3.

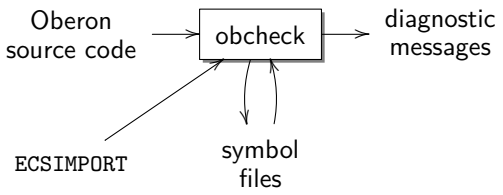
The tools process Oberon modules in several consecutive stages. In each stage, the internal representation of the module is changed and transformed. Figure 7.2 shows all stages and the different representations. Additionally, each tool except for the pretty printer and the interpreter generates a so-called *symbol file* for each source file being processed. Symbol files contain information about the interface of a module and are needed whenever a module attempts to import another one. When importing a module, its symbol file is first searched in the current working directory and then in the relative directory given by the `ECSIMPORT` environment variable.

7.7.1 obprint

`obprint` is a pretty printer for the Oberon programming language. It reformats the source code of Oberon modules and writes it to the standard output stream.

7.7.2 obcheck

`obcheck` is a syntactic and semantic checker for the Oberon programming language. It just performs syntactic and semantic checks on Oberon modules and writes its diagnostic messages to the standard error stream. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.



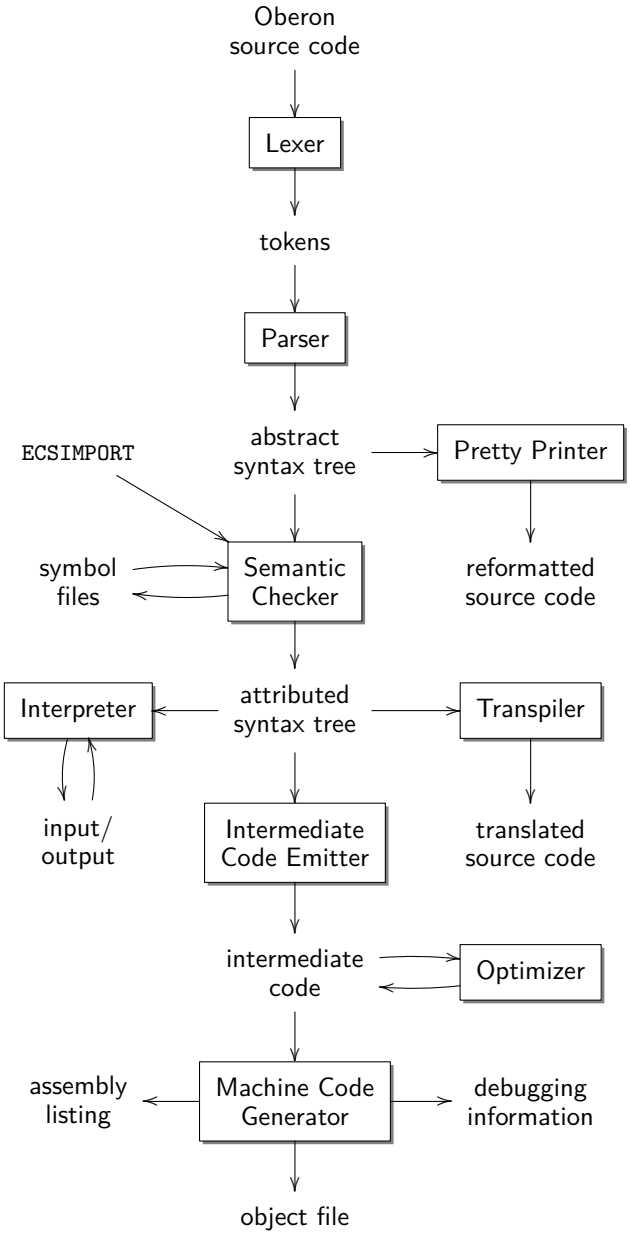
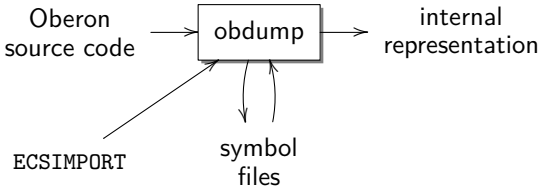


Figure 7.2: Data flow within the tools for Oberon

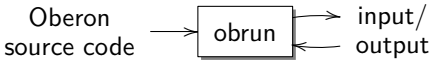
7.7.3 obdump

obdump is a serializer for the Oberon programming language. It dumps the complete internal representation of modules written in Oberon into an XML document. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



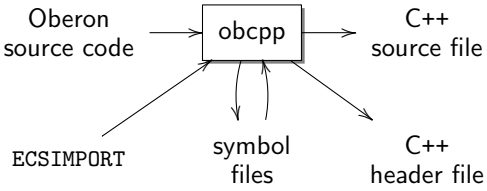
7.7.4 obrun

obrun is an interpreter for the Oberon programming language. It processes and executes modules written in Oberon. This tool does neither generate nor process symbol files while interpreting modules. If a module is imported by another one, its filename has to be named before the other one in the list of command-line arguments.



7.7.5 obcpp

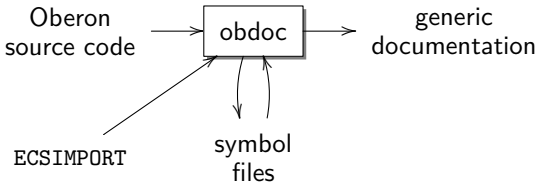
obcpp is a transpiler for the Oberon programming language. It translates programs written in Oberon into C++ programs and stores them in corresponding source and header files. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. The same interface is provided by the generated header file which can be used in other parts of the C++ program.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

7.7.6 obdoc

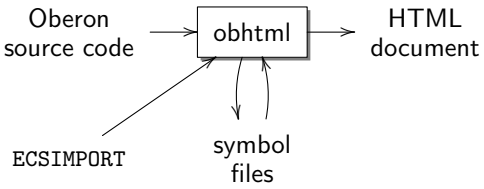
obdoc is a generic documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into a generic documentation. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

7.7.7 obhtml

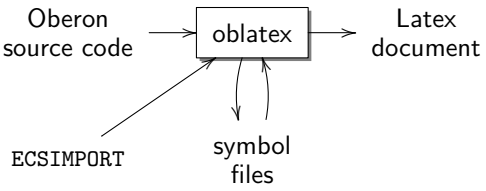
obhtml is an HTML documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into an HTML document. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.



For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

7.7.8 oblatex

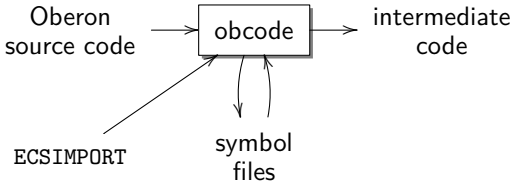
oblatex is a Latex documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into a Latex document. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.



For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

7.7.9 obcode

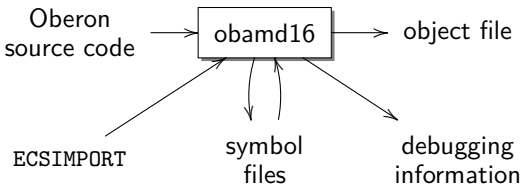
`obcode` is an intermediate code generator for the Oberon programming language. It generates intermediate code from modules written in Oberon and stores it in corresponding assembly files. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this tool require additional runtime support that is stored in the `obcoderun` assembly file. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about intermediate code and its purpose, see Chapter 23.

7.7.10 obamd16

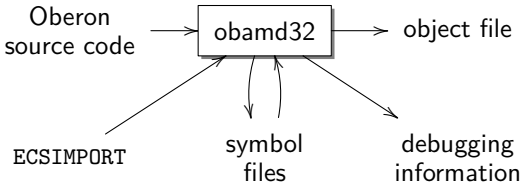
`obamd16` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd16run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.11 obamd32

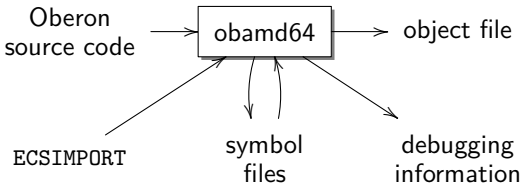
`obamd32` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.12 obamd64

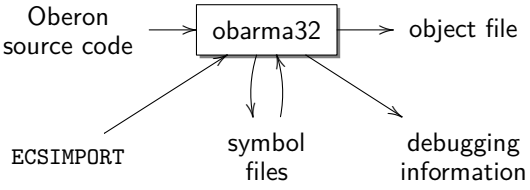
`obamd64` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.13 obarma32

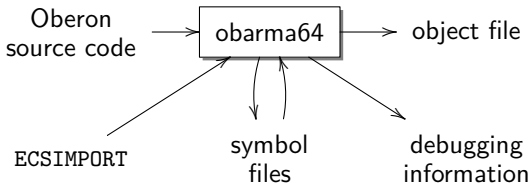
`obarma32` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarma32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.14 obarma64

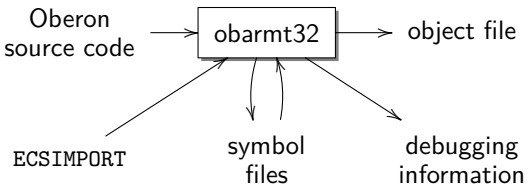
`obarma64` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarma64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.15 obarmt32

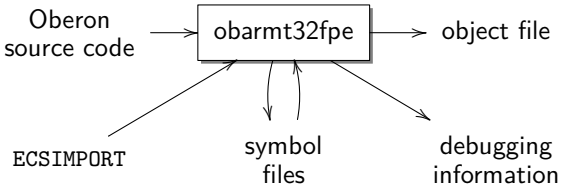
`obarmt32` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarmt32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.16 obarmt32fpe

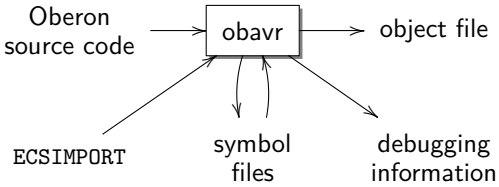
`obarmt32fpe` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarmt32fperun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.17 obavr

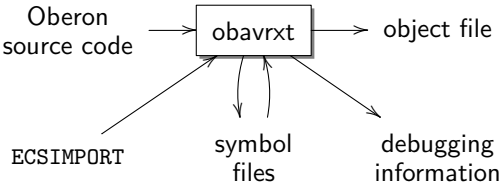
`obavr` is a compiler for the Oberon programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavrrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.18 obavrxt

`obavrxt` is a compiler for the Oberon programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavrxtrun` library file.

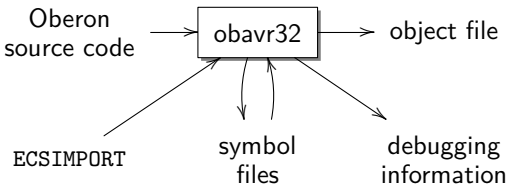


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR hardware

architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.19 obavr32

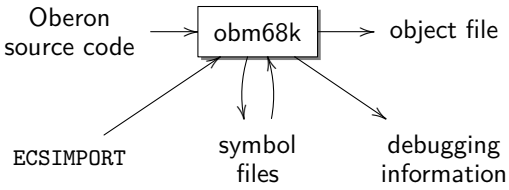
`obavr32` is a compiler for the Oberon programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavr32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.20 obm68k

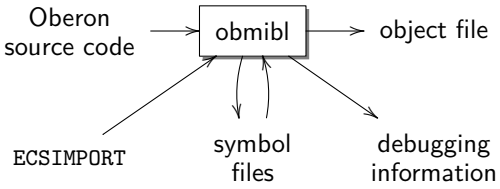
`obm68k` is a compiler for the Oberon programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obm68krun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.21 obmibl

`obmibl` is a compiler for the Oberon programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmiblrun` library file.

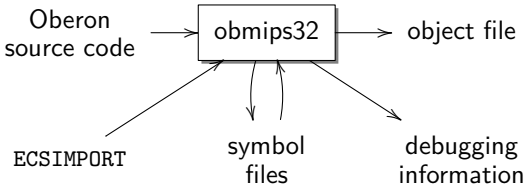


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.22 obmips32

`obmips32` is a compiler for the Oberon programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from modules written in Oberon

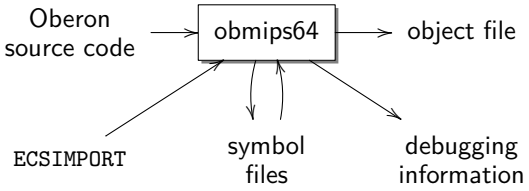
and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmips32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.23 obmips64

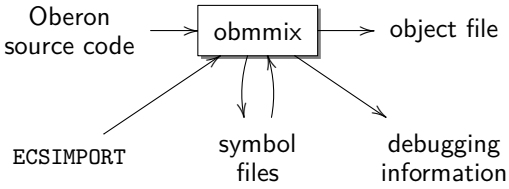
`obmips64` is a compiler for the Oberon programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmips64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.24 obmmix

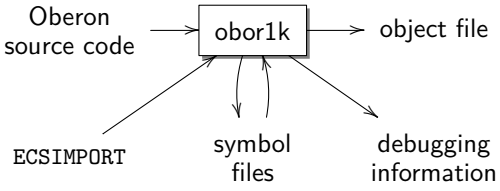
`obmmix` is a compiler for the Oberon programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmmixrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.25 obor1k

`obor1k` is a compiler for the Oberon programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obor1krun` library file.

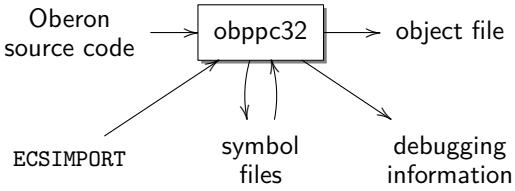


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the OpenRISC 1000

hardware architecture, see Chapter 17. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.26 obppc32

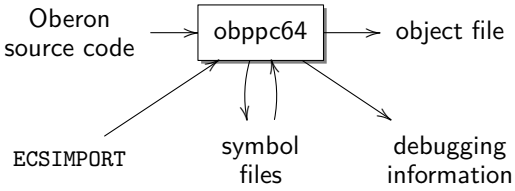
`obppc32` is a compiler for the Oberon programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obppc32run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.27 obppc64

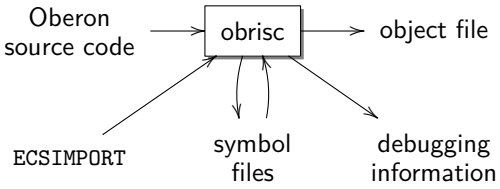
`obppc64` is a compiler for the Oberon programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obppc64run` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.28 obrisc

`obrisc` is a compiler for the Oberon programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obriscrun` library file.

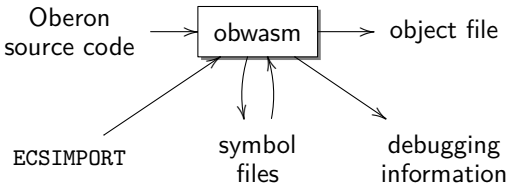


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.29 obwasm

`obwasm` is a compiler for the Oberon programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from modules written in

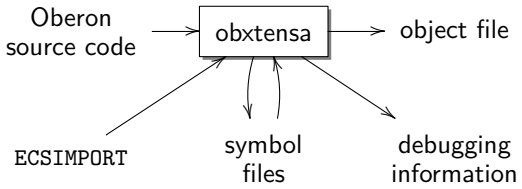
Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obwasmrun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.7.30 obxtensa

`obxtensa` is a compiler for the Oberon programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obxtensarun` library file.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

7.8 Interoperability

In accordance with the goal of the Eigen Compiler Suite to enable interoperability between its implemented programming languages, the compilers for Oberon provide different mechanisms to exchange data with programs written with other tools of the Eigen Compiler Suite. The interoperability is enabled by a common intermediate code representation and calling convention. For more information about intermediate code and its purpose, see Chapter 23.

This section describes the naming conventions used to uniquely identify the intermediate code sections defined by the compilers for Oberon as well as the ways of accessing sections defined by other compilers and assemblers.

7.8.1 Naming Conventions

The compilers for the Oberon programming language define a code section for each body of a module and each procedure defined therein. Additional data sections are defined for global variables as well as record descriptors which contain the type information of records needed at runtime. The name of each section is the identifier of the actual declaration prefixed by the name of its containing scope and an attached period such that names resemble qualified identifiers. The name of module packages are followed by colons rather than periods.

7.8.2 Accessing Sections

Oberon as implemented by the Eigen Compiler Suite allows two different ways of accessing sections defined by other compilers or assemblers.

- The predeclared procedure `SYSTEM.ASM` allows writing inline assembly code which naturally enables arbitrary access to any section, see Section 7.2.9. For more information about the generic assembly language and how to use it, see Chapter 8.
- Forward declarations marked as external allow referring to data and code sections that are defined elsewhere, see Section 7.2.2.

8 Generic Assembly Language

The Eigen Compiler Suite provides assemblers for several different hardware architectures. All of these tools are based on a generic assembly language that supports all of their common features. This chapter describes the generic assembly language and its implementation by the Eigen Compiler Suite.

Die Wahrheit und Einfachheit der Natur
sind immer die letzten Grundlagen
einer bedeutenden Kunst.

— Paul Ernst

8.1 Introduction

The most obvious task of an assembler is to translate the textual representation of some processor instructions into their binary form. The target hardware architecture of the assembler specifies the instruction set and the actual encoding of its instructions. Therefore, an assembler implementing one specific instruction set is in general not able to translate instructions of another instruction set.

There is some functionality however that all assemblers do have in common. This includes for example the support for constant definitions, symbolic names for the target of branch instructions, or sections that group instructions together. The Eigen Compiler Suite defines a generic assembly language that supports all of these common features. The generic assembly language is therefore an abstraction for all concrete assembly languages and instruction sets supported by the assemblers of the Eigen Compiler Suite. These assemblers only have to implement the translation of those parts of a program that are actually dependent on the hardware architecture.

8.2 Representation

The Eigen Compiler Suite provides several different assemblers. They target different hardware architectures and therefore implement the encoding of different instruction sets. However, each assembler additionally implements the generic assembly language which is able to represent all these instruction sets in an abstract way. This section describes how assembly programs are represented by the generic assembly language.

8.2.1 Programs

An assembly program expressed in generic assembly language consists of an arbitrary number of interconnected units called *sections*. Sections are used as containers for data and machine code. Each section contains *instructions* and *directives* that textually describe the actual binary contents of the section.

An instruction is a single operation of a processor implementing the target hardware architecture. The architecture defines the available set of instructions and their textual and binary representations. An assembler for that architecture uses this encoding to generate the actual machine code for each instruction in the source code. Directives on the other hand do usually not generate any machine code but rather influence the actual encoding and state of the assembler.

8.2.2 Section Types

Sections represent contiguous memory regions containing either data or executable machine code. The *type* of a section describes the intended purpose of this memory region:

- Code Sections

  Code sections contain the binary representation of machine code for a specific hardware architecture. *Standard code sections* are typically used to represent functions which are usually called by machine code contained in other code sections. *Initializing code sections* on the other hand represent machine code that has to be executed automatically at the begin of a program. *Data initializing code sections* are executed prior to other initializing code sections. Their purpose is to initialize data sections with the data needed by initializing code sections.

- Data Sections

  Data sections are not executed but contain the binary representation of the global data of a program. *Standard data sections* usually contain the modifiable global data of a program like global variables. Often times, standard data sections only reserve space for data because it cannot be statically initialized and has to be generated at runtime. *Constant data sections* on the other hand represent global data that is not variable and

remains constant throughout the execution of a program. Constant data sections are typically used to represent character strings.

- Metadata Sections

  Metadata sections are not part of the actual program but contain metadata about it. *Heading metadata sections* are placed in front of all other sections while *trailing metadata sections* are placed behind all other sections. This allows mimicking the layout of some specific binary file format when linking object files into a single binary file.

The type of a section influences its actual placement in memory. Constant data sections for example may be placed in a read-only memory area. Initializing code sections are placed in front of all other code sections in order to guarantee their automatic execution. In general, the distinction between code and data sections enables supporting separate memories as required by Harvard architectures.

8.2.3 Section Options

Each section has a symbolic name that represents its actual start address in memory. In addition, optional aliases allow referring to specific offsets within a section by name. The *options* of a section describe what happens if two or more sections can be referred to by the same name or are never used. Sections can have the following freely combinable options:

- Default Sections

  Sections marked as *default* are replaced if there is another section with the same name. Default sections are typically used for code sections providing replaceable functionality.

- Phantom Sections

  Sections marked as *phantom* get omitted even when they are used.

- Required Sections

  Sections marked as *required* are always used while all other sections are only used if referenced by other used sections.

- Shared Sections

  Sections marked as *shared* are merged together if they contain exactly the same binary data. Shared sections are typically used for constant data sections representing data that does not need to be duplicated in memory.

Assembly programs that begin with instructions rather than section creation directives implicitly define a standard code section called `main`. These sections represent entry points of programs and are therefore always implicitly required. They are executed automatically after initializing code sections and have to eventually return the control of execution to the host environment.

8.2.4 Syntax

Each line of code of a program written in generic assembly language contains some combination of the following three fields:

$$
\begin{aligned}
\textit{Line} &= \textit{Label}_{opt}\ \textit{Statement}_{opt}\ \textit{Comment}_{opt} \\
\textit{Label} &= \textit{Identifier}\texttt{:} \\
\textit{Statement} &= \textit{Instruction} \mid \textit{Directive} \\
\textit{Instruction} &= \textit{Mnemonic}\ \textit{Expressions}_{opt} \\
\textit{Directive} &= \texttt{.}\textit{Mnemonic}\ \textit{Expressions}_{opt} \mid \texttt{\#}\textit{Mnemonic}\ \textit{Expressions}_{opt} \\
\textit{Mnemonic} &= \textit{Identifier}
\end{aligned}
$$

Each instruction and directives defining constants or binary data may be associated with a *label*. Labels are unique identifiers that are followed by a colon. A label can be used to refer to the numerical offset of the labeled instruction within the current section using a unique symbolic name. It can also be used to refer to the expression of a *constant definition* within the current section.

The mnemonic is a symbolic name that identifies the actual instruction or directive. Mnemonics of directives are prefixed with a dot or a number sign. Most instructions and directives take one or more operand specifiers that are separated by commas. Operands are arbitrary expressions that specify numeric values, registers, labels, or memory addresses. Section 8.3 describes the capabilities and representation of expressions. Sections 8.4 and 8.5 give more information about the valid operands of instructions and directives.

Finally, the generic assembly language supports comments that are ignored during the translation and may therefore contain a human-readable description of the source code. Comments are introduced with a semicolon, may contain any text, and extend to the end of the line.

8.2.5 Translation

The generic syntax defined by the generic assembly language allows supporting any textual representation. Assemblers implementing the generic assembly language translate the source code in two different stages:

1. The source code is decomposed into an abstract syntax tree representing each line of the source code. If a line contains a directive, it is executed according to Section 8.5 and the next line of code is translated. If a line contains an instruction, its operands are resolved and the whole instruction is rewritten using these results. Resolving in this context means evaluating subexpressions or mapping labels to offsets. If the operand or its subexpressions contain identifiers or operations that cannot be evaluated, they are rewritten as they are. This most often includes instruction mnemonics or the names of registers only the concrete assembler in use knows about.

2. The rewritten instruction is passed to the concrete assembler in use. This assembler in turn assembles one instruction at a time and returns its equivalent binary encoding.

Decomposing the translation into two consecutive stages offers some advantages. First, all of the common tasks of the assemblers can be performed in the first stage. Features supported by this stage are automatically available in any assembler by design. This includes managing sections, writing the resulting object file, or supporting forward referencing labels using a prior assembly pass that only records instruction offsets. Second, the actual translation of the rewritten instruction in the second stage is much easier because all symbolic names and expressions are already resolved beforehand. This enables a loose coupling between the generic and the concrete part of an assembler. These two parts only interface by exchanging a simple textual representation of an instruction by its binary equivalent as shown in Figure 8.1.

8.3 Expressions

The generic assembly language is able to handle arbitrarily complex expressions that can be used as operands for instructions and directives. Sections 8.4 and 8.5 give more information about the valid operands of instructions and directives. This section describes how expressions are evaluated according to the generic assembly language. They are textually represented according to the following syntax:

Expressions = *Expression* | *Expressions* , *Expression*
Expression = *Character* | *String* | *Number* | *Identifier* |
Address | *Unary-Operation* | *Binary-Operation*

Oftentimes, composite expressions like unary and binary operations as well as some instructions and directives need numeric operands. In these cases, expressions are evaluated yielding a constant number as result. However, there are expressions that cannot be evaluated and do therefore not yield a result. Such expressions are literally treated as text during the translation as described in Section 8.2.5.

8.3.1 Characters

The generic assembly language treats any non-empty sequence of up to eight characters enclosed in single quotes as character literals. A character literal can be evaluated and yields an integer number that corresponds to the character sequence as represented in memory of the target hardware architecture. Special characters can be defined using the escape sequences as shown in Table 8.1.

Sequence	Description
`\0`	null character
`\a`	alert
`\b`	backspace
`\t`	horizontal tab
`\n`	new-line
`\v`	vertical tab
`\f`	form feed
`\r`	carriage return
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\?`	question mark
`\x`*n*	literal character with ordinal value expressed by hexadecimal number *n*

Table 8.1: Escape sequences supported by the generic assembly language

8.3.2 Strings

Assemblers treat any character sequences enclosed in double quotes as string literals. Strings cannot be evaluated, but assemblers are able to generate the binary data representation of each character in a string literal. Special characters can be defined using the escape sequences as shown in Table 8.1. The number of octets representing a single character depends on the actual directive used to generate the binary data.

8.3.3 Numbers

The generic assembly language supports numeric literals for the evaluation of expressions. It allows specifying integer and real numbers using different number bases according to the following syntax:

Number = *Integer* | *Real*
Integer = *Digits* | *Binary* | *Octal* | *Decimal* | *Hexadecimal*
Binary = `0b` *Digits* | *Digits* `b`
Octal = `0o` *Digits* | *Digits* `o` | `0` *Digits*
Decimal = `0d` *Digits* | *Digits* `d` | *Integer*
Hexadecimal = `0h` *Digits* | *Digits* `h` | `0x` *Digits*
Real = *Digits* `.` *Digits* Exponent_{opt} | `inf` | `nan`

Exponent = e $Sign_{opt}$ *Digits*
Sign = + | −

The digit sequence of a number may only contain digits that are valid for its number base. It may also contain separating single quotes in its integral part which are ignored when determining its value.

8.3.4 Identifiers

An identifier is any sequence of characters like letters and digits that does not begin with a digit. It either refers to a label, a section name, or to a predefined identifier of the actual instruction set. This includes mnemonics and register names defined by the instruction set for example.

If the identifier names a label, its value is the offset of the label within the binary representation of the section. Usually, this offset is expressed in octets. If an instruction expects other units or a relative operand however, the offset is automatically modified accordingly. If the identifier names a constant definition, its value corresponds to the expression of that definition. The mnemonic of some directives can also be used in expressions and yield the value of their current setting, see Section 8.3.10. In all other cases, identifiers cannot be evaluated and are simply treated as text.

8.3.5 Addresses

Addresses are identifiers that are prefixed with an at sign and allow referencing code or data sections by name. They do not yield a value, but they act as a placeholder for the future memory address of the referenced section. Addresses are usually absolute and expressed in octets, but if there are instructions that expect operands with other units or relative references, they are automatically modified accordingly. All expressions involving an address may additionally be incremented or decremented by a signed displacement and shifted to the right by a scale.

An address identifier may be enclosed in double quotes and contain any character sequence including escape sequences like strings where an empty address identifier always yields address zero. If it contains question marks, the actual name of the referenced section begins behind the last question mark. If none of the sections named in front of a question mark are actually used, the name of the section instead begins behind an optional colon following the last question mark.

8.3.6 Arithmetic Operations

The generic assembly language supports the following arithmetic operations. The precedence of each operation is shown in Table 8.2. All unary operations have right-to-left associativity. All binary operations have left-to-right associativity:

Precedence	Operator	Operation
1	`( )`	Grouping or memory access
	`[ ]`	Memory access
	`{ }`	Register list
2	`+`	Identity
	`-`	Negation
	`~`	Bitwise NOT
	`!`	Logical NOT
3	`*`	Multiplication
	`/`	Division
	`%`	Modulo
4	`+`	Addition
	`-`	Subtraction
5	`<<`	Bitwise left shift
	`>>`	Bitwise right shift
6	`<`	Less-than comparison
	`<=`	Less-than-or-equal-to comparison
	`>`	Greater-than comparison
	`>=`	Greater-than-or-equal-to comparison
7	`==`	Equal comparison
	`!=`	Unequal comparison
8	`===`	Identical comparison
	`!==`	Unidentical comparison
9	`&`	Bitwise AND
10	`^`	Bitwise exclusive OR
11	`|`	Bitwise inclusive OR
12	`&&`	Logical AND
13	`||`	Logical OR

Table 8.2: Operator precedence defined by the generic assembly language

- Identity *+ Expression*

  The result of the identity operation is the value of its operand.

- Negation *− Expression*

  The negation operation negates the value of its operand.

- Addition *Expression + Expression*

  The addition operation computes the sum of the values of its operands.

- Subtraction *Expression − Expression*

  The subtraction operation computes the difference of the values of its operands.

- Multiplication *Expression ∗ Expression*

  The multiplication operation computes the product of the values of its operands.

- Division *Expression / Expression*

  The division operation computes the integer quotient of the division of the value of its first operand by the value of the second value.

- Modulo *Expression % Expression*

  The modulo operation computes the integer remainder of the division of the value of its first operand by the value of the second value.

8.3.7 Bitwise Operations

The generic assembly language supports the following bitwise operations. The precedence of each operation is shown in Table 8.2:

- Bitwise NOT *~ Expression*

  The bitwise NOT operation computes the one's complement of the value of its operand.

- Bitwise left shift *Expression << Expression*

  The bitwise left shift operation computes the value of its first operand shifted left by the value of its second operand.

- Bitwise right shift *Expression >> Expression*

  The bitwise right shift operation computes the value of its first operand shifted right by the value of its second operand.

- Bitwise AND *Expression & Expression*

  The bitwise AND operation yields the bitwise AND function of the values of its operands.

- Bitwise inclusive OR *Expression* | *Expression*

  The bitwise inclusive OR operation yields the bitwise inclusive OR function of the values of its operands.

- Bitwise exclusive OR *Expression* ^ *Expression*

  The bitwise exclusive OR operation yields the bitwise exclusive OR function of the values of its operands.

8.3.8 Logical Operations

The generic assembly language supports the following logical operations. The precedence of each operation is shown in Table 8.2. During all these logical operations, an evaluated value of zero denotes the Boolean value false. All other values represent the Boolean value true. Logical operations yield either the value one or the value zero representing the Boolean values true and false respectively:

- Logical NOT ! *Expression*

  The logical NOT operation yields the complement of the Boolean value of its operand.

- Logical AND *Expression* && *Expression*

  The logical AND operation performs a logical conjunction of the Boolean values of its operands. The second operand is not evaluated if the first operand evaluates to false.

- Logical OR *Expression* || *Expression*

  The logical OR operation performs a logical disjunction of the Boolean values of its operands. The second operand is not evaluated if the first operand evaluates to true.

8.3.9 Comparison Operations

The generic assembly language supports the following comparison operations. The precedence of each operation is shown in Table 8.2. Comparison operations yield either the value one or the value zero representing the Boolean values true and false respectively:

- Equal comparison *Expression* == *Expression*

  The equal comparison operation returns a Boolean value indicating whether the values of both of its operands are equal.

- Unequal comparison *Expression* != *Expression*

  The unequal comparison operation returns a Boolean value indicating whether the values of both of its operands are unequal.

- Less-than comparison *Expression* < *Expression*

  The less-than comparison operation returns a Boolean value indicating whether the value of its first operand is less than the value of its second operand.

- Less-than-or-equal comparison *Expression* <= *Expression*

  The less-than-or-equal comparison operation returns a Boolean value indicating whether the value of its first operand is less than or equal to the value of its second operand.

- Greater-than comparison *Expression* > *Expression*

  The greater-than comparison operation returns a Boolean value indicating whether the value of its first operand is greater than the value of its second operand.

- Greater-than-or-equal comparison *Expression* >= *Expression*

  The greater-than-or-equal comparison operation returns a Boolean value indicating whether the value of its first operand is greater than or equal to the value of its second operand.

- Identical comparison *Expression* === *Expression*

  The identical comparison operation returns a Boolean value indicating whether the lexical token sequences of both of its operands are identical.

- Unidentical comparison *Expression* !== *Expression*

  The unidentical comparison operation returns a Boolean value indicating whether the lexical token sequences of both of its operands are unidentical.

8.3.10 Special-Purpose Operations

In addition to all arithmetic, bitwise, logical, and comparison operations, the generic assembly language supports the following special-purpose operations:

- Bit Mode `.bitmode`

  The bit mode operation returns the current bit mode that is used to translate instructions into machine code. This operation returns the default bit mode of the target hardware architecture if available and left unchanged by the bit mode directive, see Section 8.5.6.

- Little-Endian Mode `.little`

  The little-endian mode operation returns a Boolean value indicating whether generated binary data is currently ordered least significant octet first. This operation returns the default endianness of the target hardware architecture if left unchanged by endian mode directives, see Sections 8.5.5 and 8.5.28.

- Big-Endian Mode `.big`

  The big-endian mode operation returns a Boolean value indicating whether generated binary data is currently ordered most significant octet first. This operation returns the default endianness of the target hardware architecture if left unchanged by endian mode directives, see Sections 8.5.5 and 8.5.28.

- Section Alignment `.alignment`

  The section alignment operation returns the alignment of the current aligned section as specified by the section alignment directive, see Section 8.5.3.

- Section Origin `.origin`

  The section origin operation returns the origin of the current fixed section as specified by the section origin directive, see Section 8.5.30.

- Default Section `.default`

  The default section operation returns a Boolean value indicating whether the current section is a default section as marked by the default section directive, see Section 8.5.12.

- Phantom Section `.phantom`

  The phantom section operation returns a Boolean value indicating whether the current section is a phantom section as marked by the phantom section directive, see Section 8.5.32.

- Required Section `.required`

  The required section operation returns a Boolean value indicating whether the current section is a required section as marked by the required section directive, see Section 8.5.36.

- Shared Section `.shared`

  The shared section operation returns a Boolean value indicating whether the current section is a shared section as marked by the shared section directive, see Section 8.5.38.

The following special-purpose operations have a functional syntax accepting a single argument and modify the result of referencing labels or sections by name:

- Relative Offset `offset (` *Expression* `)`

  The relative offset operation returns the offset of the named label or section relative to the current position expressed in octets.

- Section Size `size (` *Expression* `)`

  The section size operation returns the binary size of the named section expressed in octets.

- Section Extent `extent (` *Expression* `)`

  The section extent operation returns the address of the named section plus its binary size.

- Member Position `position (` *Expression* `)`

  The member position operation returns the position of the named section relative to the beginning of its group expressed in octets. See Section 8.5.22 for more information about the section group directive.

- Member Index `index (` *Expression* `)`

  The member index operation returns the index of the named section within the sequence of sections of its group. See Section 8.5.22 for more information about the section group directive.

- Member Count `count (` *Expression* `)`

  The member count operation returns the total number of sections contained in the named group. See Section 8.5.22 for more information about the section group directive.

8.4 Instructions

The actually available set of instructions and its textual representation are defined by the target hardware architecture. The generic assembly language therefore does not know anything about the syntax or semantics of the concrete instruction set in use. For a complete description of the concrete instruction set and its usage, users have to refer to the documentation of the corresponding target hardware architecture. The remainder of this section lists all instruction sets supported by Eigen Compiler Suite.

8.4.1 AMD64 Instruction Set

The Eigen Compiler Suite supports the AMD64 instruction set as listed below and uses the same assembly syntax as predefined by AMD [11, 12, 13]. The only exception are instructions for far procedure calls and far jumps which take the selector and offset of the immediate far pointer as two separate operands. The actual set of supported instructions depends on the operating mode that is used. If there are two or more possible encodings of an instruction, the shortest one is used by default. In order to specify the encoding of an instruction explicitly, the size of its operands can be overridden by prefixing them with one of the following identifiers: `byte` for 8-bit, `word` for 16-bit, `dword` for 32-bit, `qword` for 64-bit, `oword` for 128-bit, `yword` for 256-bit, and `zword` for 512-bit. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9.

aaa aad aam aas adc adcx add addpd addps addsd addss addsubpd addsubps adox aesdec aesdeclast aesenc aesenclast aesimc aeskeygenassist and andn andnpd andnps andpd andps arpl bextr blcfill blci blcic blcmsk blcs blendpd blendps blendvpd blendvps blsfill blsi blsic blsmsk blsr bound bsf bsr bswap bt btc btr bts bzhi call callfar cbw cdq cdqe clac clc cld clflush clflushopt clgi cli clrssbsy clts clwb clzero cmc cmova cmovae cmovb cmovbe cmovc cmove cmovg cmovge cmovl cmovle cmovna cmovnae cmovnb cmovnbe cmovnc cmovne cmovng cmovnge cmovnl cmovnle cmovno cmovnp cmovns cmovnz cmovo cmovp cmovpe cmovpo cmovs cmovz cmp cmpeqpd cmpeqps cmpeqsd cmpeqss cmplepd cmpleps cmplesd cmpless cmpltpd cmpltps cmpltsd cmpltss cmpneqpd cmpneqps cmpneqsd cmpneqss cmpnlepd cmpnleps cmpnlesd cmpnless cmpnltpd cmpnltps cmpnltsd cmpnltss cmpordpd cmpordps cmpordsd cmpordss cmppd cmpps cmpsb cmpsd cmpsq cmpss cmpsw cmpunordpd cmpunordps cmpunordsd cmpunordss cmpxchg cmpxchg16b cmpxchg8b comisd comiss cpuid cqo crc32 cvtdq2pd cvtdq2ps cvtpd2dq cvtpd2pi cvtpd2ps cvtpi2pd cvtpi2ps cvtps2dq cvtps2pd cvtps2pi cvtsd2si cvtsd2ss cvtsi2sd cvtsi2ss cvtss2sd cvtss2si cvttpd2dq cvttpd2pi cvttps2dq cvttps2pi cvttsd2si cvttss2si cwd cwde daa das dec div divpd divps divsd divss dppd dpps emms enter extractps extrq f2xm1 fabs fadd faddp fbld fbstp fchs fcmovb fcmovbe fcmove fcmovnb fcmovnbe fcmovne fcmovnu fcmovu fcom fcomi fcomip fcomp fcompp fcos fdecstp fdiv fdivp fdivr fdivrp femms ffree fiadd ficom ficomp fidiv fidivr fild fimul fincstp fist fistp fisttp fisub fisubr fld fld1 fldcw fldenv fldl2e fldl2t fldlg2 fldln2 fldpi fldz fmul fmulp fnclex fninit fnop fnsave fnstcw fnstenv fnstsw fpatan fprem fprem1 fptan frndint frstor fscale fsin fsincos fsqrt fst fstp fsub fsubp fsubr fsubrp ftst fucom fucomi fucomip fucomp fucompp fwait fxam fxch fxrstor fxsave fxtract fyl2x fyl2xp1 haddpd haddps hlt hsubpd hsubps idiv imul in inc incssp insb insd insertps insertq insw int int3 into invd invlpg invlpga invlpgb invpcid iret iretd iretq ja jae jb jbe jc jcxz je jecxz jg jge jl jle jmp jmpfar jna jnae jnb jnbe jnc jne jng jnge jnl jnle jno jnp jns jnz jo jp jpe jpo jrcxz js jz lahf lar lddqu ldmxcsr lds lea leave les lfence lfs lgdt lgs lidt lldt llwpcb lmsw lodsb lodsd lodsq lodsw loop loope loopne loopnz loopz lsl lss ltr lwpins lwpval lzcnt maskmovdqu maskmovq maxpd maxps maxsd maxss mcommit mfence minpd minps minsd minss monitor monitorx mov movapd movaps movbe movd movddup movdir64b movdiri movdq2q movdqa movdqu movhlps movhpd movhps movlhps movlpd movlps movmskpd movmskps movntdq movntdqa movnti movntpd movntps movntq movntsd movntss movq movq2dq movsb movsd movshdup movsldup movsq movss movsw movsx movsxd movupd movups movzx mpsadbw mul mulpd mulps mulsd mulss mulx mwait mwaitx neg nop not or orpd orps out outsb outsd outsw pabsb pabsd pabsw packssdw packsswb packusdw packuswb paddb paddd paddq paddsb paddsw paddusb paddusw paddw palignr pand pandn pause pavgb pavgusb pavgw pblendvb pblendw pclmulqdq pcmpeqb pcmpeqd pcmpeqq pcmpeqw pcmpestri pcmpestrm pcmpgtb pcmpgtd pcmpgtq pcmpgtw pcmpistri pcmpistrm pdep pext pextrb pextrd pextrq pextrw pf2id pf2iw pfacc pfadd pfcmpeq pfcmpge pfcmpgt pfmax pfmin pfmul pfnacc pfpnacc pfrcp pfrcpit1 pfrcpit2 pfrsqit1 pfrsqrt pfsub pfsubr phaddd phaddsw phaddw phminposuw phsubd phsubsw phsubw pi2fd pi2fw pinsrb pinsrd pinsrq pinsrw pmaddubsw pmaddwd pmaxsb pmaxsd pmaxsw pmaxub pmaxud pmaxuw pminsb pminsd pminsw pminub pminud pminuw pmovmskb pmovsxbd pmovsxbq pmovsxbw pmovsxdq pmovsxwd pmovsxwq pmovzxbd pmovzxbq pmovzxbw pmovzxdq pmovzxwd pmovzxwq pmuldq pmulhrsw pmulhrw pmulhuw pmulhw pmulld pmullw pmuludq pop popa popad popcnt popf popfd popfq por prefetch prefetchnta prefetcht0 prefetcht1 prefetcht2 prefetchw psadbw

pshufb pshufd pshufhw pshuflw pshufw psignb psignd psignw pslld pslldq psllq psllw psmash psrad psraw psrld psrldq psrlq psrlw psubb psubd psubq psubsb psubsw psubusb psubusw psubw pswapd ptest punpckhbw punpckhdq punpckhqdq punpckhwd punpcklbw punpckldq punpcklqdq punpcklwd push pusha pushad pushf pushfd pushfq pvalidate pxor rcl rcpps rcpss rcr rdfsbase rdgsbase rdmsr rdpid rdpkru rdpmc rdpru rdrand rdseed rdsspd rdsspq rdtsc rdtscp ret retf rmpadjust rmpquery rmpread rmpupdate rol ror rorx roundpd roundps roundsd roundss rsm rsqrtps rsqrtss rstorssp sahf sal sar sarx saveprevssp sbb scasb scasd scasq scasw seta setae setb setbe setc sete setg setge setl setle setna setnae setnb setnbe setnc setne setng setnge setnl setnle setno setnp setns setnz seto setp setpe setpo sets setssbsy setz sfence sgdt sha1msg1 sha1msg2 sha1nexte sha1rnds4 sha256msg1 sha256msg2 sha256rnds2 shl shld shlx shr shrd shrx shufpd shufps sidt skinit sldt slwpcb smsw sqrtpd sqrtps sqrtsd sqrtss stac stc std stgi sti stmxcsr stosb stosd stosq stosw str sub subpd subps subsd subss swapgs syscall sysenter sysexit sysret t1mskc test tlbsync tzcnt tzmsk ucomisd ucomiss ud0 ud1 ud2 unpckhpd unpckhps unpcklpd unpcklps vaddpd vaddps vaddsd vaddss vaddsubpd vaddsubps vaesdec vaesdeclast vaesenc vaesenclast vaesimc vaeskeygenassist vandnpd vandnps vandpd vandps vblendpd vblendps vblendvpd vblendvps vbroadcastf128 vbroadcasti128 vbroadcastsd vbroadcastss vcmpeqpd vcmpeqps vcmpeqsd vcmpeqss vcmplepd vcmpleps vcmplesd vcmpless vcmpltpd vcmpltps vcmpltsd vcmpltss vcmpneqpd vcmpneqps vcmpneqsd vcmpneqss vcmpnlepd vcmpnleps vcmpnlesd vcmpnless vcmpnltpd vcmpnltps vcmpnltsd vcmpnltss vcmpordpd vcmpordps vcmpordsd vcmpordss vcmppd vcmpps vcmpsd vcmpss vcmpunordpd vcmpunordps vcmpunordsd vcmpunordss vcomisd vcomiss vcvtdq2pd vcvtdq2ps vcvtpd2dq vcvtpd2ps vcvtph2ps vcvtps2dq vcvtps2pd vcvtps2ph vcvtsd2si vcvtsd2ss vcvtsi2sd vcvtsi2ss vcvtss2sd vcvtss2si vcvttpd2dq vcvttps2dq vcvttsd2si vcvttss2si vdivpd vdivps vdivsd vdivss vdppd vdpps verr verw vextractf128 vextracti128 vextractps vfmadd132pd vfmadd132ps vfmadd132sd vfmadd132ss vfmadd213pd vfmadd213ps vfmadd213sd vfmadd213ss vfmadd231pd vfmadd231ps vfmadd231sd vfmadd231ss vfmaddpd vfmaddps vfmaddsd vfmaddss vfmaddsub132pd vfmaddsub132ps vfmaddsub213pd vfmaddsub213ps vfmaddsub231pd vfmaddsub231ps vfmaddsubpd vfmaddsubps vfmsub132pd vfmsub132ps vfmsub132sd vfmsub132ss vfmsub213pd vfmsub213ps vfmsub213sd vfmsub213ss vfmsub231pd vfmsub231ps vfmsub231sd vfmsub231ss vfmsubadd132pd vfmsubadd132ps vfmsubadd213pd vfmsubadd213ps vfmsubadd231pd vfmsubadd231ps vfmsubaddpd vfmsubaddps vfmsubpd vfmsubps vfmsubsd vfmsubss vfnmadd132pd vfnmadd132ps vfnmadd132sd vfnmadd132ss vfnmadd213pd vfnmadd213ps vfnmadd213sd vfnmadd213ss vfnmadd231pd vfnmadd231ps vfnmadd231sd vfnmadd231ss vfnmaddpd vfnmaddps vfnmaddsd vfnmaddss vfnmsub132pd vfnmsub132ps vfnmsub132sd vfnmsub132ss vfnmsub213pd vfnmsub213ps vfnmsub213sd vfnmsub213ss vfnmsub231pd vfnmsub231ps vfnmsub231sd vfnmsub231ss vfnmsubpd vfnmsubps vfnmsubsd vfnmsubss vfrczpd vfrczps vfrczsd vfrczss vhaddpd vhaddps vhsubpd vhsubps vinsertf128 vinserti128 vinsertps vlddqu vldmxcsr vmaskmovdqu vmaskmovpd vmaskmovps vmaxpd vmaxps vmaxsd vmaxss vmgexit vminpd vminps vminsd vminss vmload vmmcall vmovapd vmovaps vmovd vmovddup vmovdqa vmovdqu vmovhlps vmovhpd vmovhps vmovlhps vmovlpd vmovlps vmovmskpd vmovmskps vmovntdq vmovntdqa vmovntpd vmovntps vmovq vmovsd vmovshdup vmovsldup vmovss vmovupd vmovups vmpsadbw vmrun vmsave vmulpd vmulps vmulsd vmulss vorpd vorps vpabsb vpabsd vpabsw vpackssdw vpacksswb vpackusdw vpackuswb vpaddb vpaddd vpaddq vpaddsb vpaddsw vpaddusb vpaddusw vpaddw vpalignr vpand vpandn vpavgb vpavgw vpblendd

```
vpblendvb vpblendw vpbroadcastb vpbroadcastd vpbroadcastq vpbroadcastw
vpclmulqdq vpcmov vpcmpeqb vpcmpeqd vpcmpeqq vpcmpeqw vpcmpestri vpcmpestrm
vpcmpgtb vpcmpgtd vpcmpgtq vpcmpgtw vpcmpistri vpcmpistrm vpcomb vpcomd
vpcomeqb vpcomeqd vpcomeqq vpcomequb vpcomequd vpcomequq vpcomequw vpcomeqw
vpcomfalseb vpcomfalsed vpcomfalseq vpcomfalseub vpcomfalseud vpcomfalseuq
vpcomfalseuw vpcomfalsew vpcomgeb vpcomged vpcomgeq vpcomgeub vpcomgeud
vpcomgeuq vpcomgeuw vpcomgew vpcomgtb vpcomgtd vpcomgtq vpcomgtub vpcomgtud
vpcomgtuq vpcomgtuw vpcomgtw vpcomleb vpcomled vpcomleq vpcomleub vpcomleud
vpcomleuq vpcomleuw vpcomlew vpcomltb vpcomltd vpcomltq vpcomltub vpcomltud
vpcomltuq vpcomltuw vpcomltw vpcomneqb vpcomneqd vpcomneqq vpcomnequb
vpcomnequd vpcomnequq vpcomnequw vpcomneqw vpcomq vpcomtrueb vpcomtrued
vpcomtrueq vpcomtrueub vpcomtrueud vpcomtrueuq vpcomtrueuw vpcomtruew vpcomub
vpcomud vpcomuq vpcomuw vpcomw vperm2f128 vperm2i128 vpermd vpermpd vpermps
vpermq vpextrb vpextrd vpextrq vpextrw vphaddbd vphaddbq vphaddbw vphaddd
vphadddq vphaddsw vphaddubd vphaddubq vphaddubw vphaddudq vphadduwd vphadduwq
vphaddw vphaddwd vphaddwq vphminposuw vphsubbw vphsubd vphsubdq vphsubsw
vphsubw vphsubwd vpinsrb vpinsrd vpinsrq vpinsrw vpmacsdd vpmacsdqh vpmacsdql
vpmacssdd vpmacssdqh vpmacssdql vpmacsswd vpmacssww vpmacswd vpmacsww
vpmadcsswd vpmadcswd vpmaddubsw vpmaddwd vpmaskmovd vpmaskmovq vpmaxsb
vpmaxsd vpmaxsw vpmaxub vpmaxud vpmaxuw vpminsb vpminsd vpminsw vpminub
vpminud vpminuw vpmovmskb vpmovsxbd vpmovsxbq vpmovsxbw vpmovsxdq vpmovsxwd
vpmovsxwq vpmovzxbd vpmovzxbq vpmovzxbw vpmovzxdq vpmovzxwd vpmovzxwq vpmuldq
vpmulhrsw vpmulhuw vpmulhw vpmulld vpmullw vpmuludq vpor vpperm vprotb
vprotd vprotq vprotw vpsadbw vpshab vpshad vpshaq vpshaw vpshlb vpshld
vpshlq vpshlw vpshufb vpshufd vpshufhw vpshuflw vpsignb vpsignd vpsignw
vpslld vpslldq vpsllq vpsllvd vpsllvq vpsllw vpsrad vpsravd vpsraw vpsrld
vpsrldq vpsrlq vpsrlvd vpsrlvq vpsrlw vpsubb vpsubd vpsubq vpsubsb vpsubsw
vpsubusb vpsubusw vpsubw vptest vpunpckhbw vpunpckhdq vpunpckhqdq vpunpckhwd
vpunpcklbw vpunpckldq vpunpcklqdq vpunpcklwd vpxor vrcpps vrcpss vroundpd
vroundps vroundsd vroundss vrsqrtps vrsqrtss vshufpd vshufps vsqrtpd vsqrtps
vsqrtsd vsqrtss vstmxcsr vsubpd vsubps vsubsd vsubss vtestpd vtestps vucomisd
vucomiss vunpckhpd vunpckhps vunpcklpd vunpcklps vxorpd vxorps vzeroall
vzeroupper wbinvd wbnoinvd wrfsbase wrgsbase wrmsr wrpkru wrssd wrssq wrussd
wrussq xadd xchg xgetbv xlatb xor xorpd xorps xrstor xrstors xsave xsavec
xsaveopt xsaves xsetbv
```

8.4.2 ARM Instruction Sets

The Eigen Compiler Suite supports the ARM A32 instruction set as listed below and uses the same assembly syntax as predefined by the ARM architecture [14]. The only exception are immediate values which are not prefixed by a number sign. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10.

```
adc adcs add adds and ands b bic bics bkpt bl blx bx cdp cdp2 clz cmn cmp
cpy dbg dmb dsb eor eors fabsd fabss faddd fadds fcmpd fcmped fcmpes fcmpezd
fcmpezs fcmps fcmpzd fcmpzs fcpyd fcpys fcvtds fcvtsd fdivd fdivs fldd fldmdbd
fldmdbs fldmiad fldmias flds fmacd fmacs fmdhr fmdlr fmdrr fmovd fmovs fmrdh
fmrdl fmrrd fmrrs fmrs fmscd fmscs fmsr fmsrr fmstat fmuld fmuls fnegd fnegs
fnmacd fnmacs fnmscd fnmscs fnmuld fnmuls fsitod fsitos fsqrtd fsqrts fstd
```

```
fstmdbd fstmdbs fstmiad fstmias fsts fsubd fsubs ftosid ftosis ftosizd ftosizs
ftouid ftouis ftouizd ftouizs fuitod fuitos hlt isb ldc ldcl ldc2 ldc2l ldmda
ldmdb ldmia ldmib ldr ldrb ldrbt ldrex ldrexb ldrexd ldrexh ldrh ldrsb ldrsh
ldrt mcr mcr2 mcrr mcrr2 mla mlas mov movs mrc mrc2 mrrc mrrc2 mrs msr mul
muls mvn mvns nop orr orrs pld qadd qadd16 qadd8 qaddsubx qdadd qdsub qsub
qsub16 qsub8 qsubaddx rev rev16 revsh rfeda rfedb rfeia rfeib rsb rsbs rsc
rscs sadd16 sadd8 saddsubx sbc sbcs sel setendbe setendle sev shadd16 shadd8
shaddsubx shsub16 shsub8 shsubaddx smlabb smlabt smlad smladx smlal smlals
smlalbb smlalbt smlald smlaldx smlaltb smlaltt smlatb smlatt smlawb smlawt
smlsd smlsdx smlsld smlsldx smmla smmlar smmls smmlsr smmul smmulr smuad
smuadx smulbb smulbt smull smulls smultb smultt smulwb smulwt smusd smusdx
srsda srsdb srsia srsib ssub16 ssub8 ssubaddx stc stcl stc2 stc2l stmda stmdb
stmia stmib str strb strbt strex strexb strexd strexh strh strt sub subs swi
swp swpb sxtab sxtab16 sxtah sxtb sxtb16 sxth teq tst uadd16 uadd8 uaddsubx
udf uhadd16 uhadd8 uhaddsubx uhsub16 uhsub8 uhsubaddx umaal umlal umlals umull
umulls uqadd16 uqadd8 uqaddsubx uqsub16 uqsub8 uqsubaddx usad8 usada8 usub16
usub8 usubaddx uxtab uxtab16 uxtah uxtb uxtb16 uxth wfe wfi yield
```

The Eigen Compiler Suite supports the ARM A64 instruction set as listed below and uses the same assembly syntax as predefined by the ARM architecture [14]. The only exception are immediate values which are not prefixed by a number sign.

```
abs adc adcs add addhn addhn2 addp adds addv adr adrp aesd aese aesimc aesmc
and ands asr asrv at autda autdb autdza autdzb autia autia1716 autiasp autiaz
autib autib1716 autibsp autibz autiza autizb b b.al b.cc b.cs b.eq b.ge b.gt
b.hi b.hs b.le b.lo b.ls b.lt b.mi b.ne b.nv b.pl b.vc b.vs bcax bfc bfi bfm
bfxil bic bics bif bit bl blr blraa blraaz blrab blrabz br braa braaz brab
brabz brk bsl cas casa casab casah casal casalb casalh casb cash casl caslb
caslh casp caspa caspal caspl cbnz cbz ccmn ccmp cinc cinv clrex cls clz cmeq
cmge cmgt cmhi cmhs cmle cmlt cmn cmp cmtst cneg cnt crc32b crc32cb crc32ch
crc32cw crc32cx crc32h crc32w crc32x csel cset csetm csinc csinv csneg dc
dcps1 dcps2 dcps3 dmb drps dsb dup eon eor eor3 eret eretaa eretab esb ext
extr fabd fabs facge facgt fadd faddp fcadd fccmp fccmpe fcmeq fcmge fcmgt
fcmla fcmle fcmlt fcmp fcmpe fcsel fcvt fcvtas fcvtau fcvtl fcvtl2 fcvtms
fcvtmu fcvtn fcvtn2 fcvtns fcvtnu fcvtps fcvtpu fcvtxn fcvtxn2 fcvtzs fcvtzu
fdiv fjcvtzs fmadd fmax fmaxnm fmaxnmp fmaxnmv fmaxp fmaxv fmin fminnm fminnmp
fminnmv fminp fminv fmla fmlal fmlal2 fmls fmlsl fmlsl2 fmov fmsub fmul fmulx
fneg fnmadd fnmsub fnmul frecpe frecps frecpx frinta frinti frintm frintn
frintp frintx frintz frsqrte frsqrts fsqrt fsub hint hlt hvc ic ins isb ld1
ld1r ld2 ld2r ld3 ld3r ld4 ld4r ldadd ldadda ldaddab ldaddah ldaddal ldaddalb
ldaddalh ldaddb ldaddh ldaddl ldaddlb ldaddlh ldapr ldaprb ldaprh ldar ldarb
ldarh ldaxp ldaxr ldaxrb ldaxrh ldclr ldclra ldclrab ldclrah ldclral ldclralb
ldclralh ldclrb ldclrh ldclrl ldclrlb ldclrlh ldeor ldeora ldeorab ldeorah
ldeoral ldeoralb ldeoralh ldeorb ldeorh ldeorl ldeorlb ldeorlh ldlar ldlarb
ldlarh ldnp ldp ldpsw ldr ldraa ldrab ldrb ldrh ldrsb ldrsh ldrsw ldset ldseta
ldsetab ldsetah ldsetal ldsetalb ldsetalh ldsetb ldseth ldsetl ldsetlb ldsetlh
ldsmax ldsmaxa ldsmaxab ldsmaxah ldsmaxal ldsmaxalb ldsmaxalh ldsmaxb ldsmaxh
ldsmaxl ldsmaxlb ldsmaxlh ldsmin ldsmina ldsminab ldsminah ldsminal ldsminalb
ldsminalh ldsminb ldsminh ldsminl ldsminlb ldsminlh ldtr ldtrb ldtrh ldtrsb
ldtrsh ldtrsw ldumax ldumaxa ldumaxab ldumaxah ldumaxal ldumaxalb ldumaxalh
```

```
ldumaxb ldumaxh ldumaxl ldumaxlb ldumaxlh ldumin ldumina lduminab lduminah
lduminal lduminalb lduminalh lduminb lduminh lduminl lduminlb lduminlh ldur
ldurb ldurh ldursb ldursh ldursw ldxp ldxr ldxrb ldxrh lsl lslv lsr lsrv madd
mla mls mneg mov movk movn movz mrs msr msub mul mvn neg negs ngc ngcs nop
not orn orr pacda pacdb pacdza pacdzb pacga pacia pacia1716 paciasp paciaz
pacib pacib1716 pacibsp pacibz paciza pacizb pmul pmull pmull2 prfm prfum
psb raddhn raddhn2 rax1 rbit ret retaa retab rev rev16 rev32 rev64 ror rorv
rshrn rshrn2 rsubhn rsubhn2 saba sabal sabal2 sabd sabdl sabdl2 sadalp saddl
saddl2 saddlp saddlv saddw saddw2 sbc sbcs sbfiz sbfm sbfx scvtf sdiv sdot
sev sevl sha1c sha1h sha1m sha1p sha1su0 sha1su1 sha256h sha256h2 sha256su0
sha256su1 sha512h sha512h2 sha512su0 sha512su1 shadd shl shll shll2 shrn
shrn2 shsub sli sm3partw1 sm3partw2 sm3ss1 sm3tt1a sm3tt1b sm3tt2a sm3tt2b
sm4e sm4ekey smaddl smax smaxp smaxv smc smin sminp sminv smlal smlal2 smlsl
smlsl2 smnegl smov smsubl smulh smull smull2 sqabs sqadd sqdmlal sqdmlal2
sqdmlsl sqdmlsl2 sqdmulh sqdmull sqdmull2 sqneg sqrdmlah sqrdmlsh sqrdmulh
sqrshl sqrshrn sqrshrn2 sqrshrun sqrshrun2 sqshl sqshlu sqshrn sqshrn2 sqshrun
sqshrun2 sqsub sqxtn sqxtn2 sqxtun sqxtun2 srhadd sri srshl srshr srsra sshl
sshll sshll2 sshr ssra ssubl ssubl2 ssubw ssubw2 st1 st2 st3 st4 stadd staddb
staddh staddl staddlb staddlh stclr stclrb stclrh stclrl stclrlb stclrlh
steor steorb steorh steorl steorlb steorlh stllr stllrb stllrh stlr stlrb
stlrh stlxp stlxr stlxrb stlxrh stnp stp str strb strh stset stsetb stseth
stsetl stsetlb stsetlh stsmax stsmaxb stsmaxh stsmaxl stsmaxlb stsmaxlh stsmin
stsminb stsminh stsminl stsminlb stsminlh sttr sttrb sttrh stumax stumaxb
stumaxh stumaxl stumaxlb stumaxlh stumin stuminb stuminh stuminl stuminlb
stuminlh stur sturb sturh stxp stxr stxrb stxrh sub subhn subhn2 subs suqadd
svc swp swpa swpab swpah swpal swpalb swpalh swpb swph swpl swplb swplh sxtb
sxth sxtl sxtl2 sxtw sys sysl tbl tbnz tbx tbz tlbi trn1 trn2 tst uaba uabal
uabal2 uabd uabdl uabdl2 uadalp uaddl uaddl2 uaddlp uaddlv uaddw uaddw2 ubfiz
ubfm ubfx ucvtf udf udiv udot uhadd uhsub umaddl umax umaxp umaxv umin uminp
uminv umlal umlal2 umlsl umlsl2 umnegl umov umsubl umulh umull umull2 uqadd
uqrshl uqrshrn uqrshrn2 uqshl uqshrn uqshrn2 uqsub uqxtn uqxtn2 urecpe urhadd
urshl urshr ursqrte ursra ushl ushll ushll2 ushr usqadd usra usubl usubl2
usubw usubw2 uxtb uxth uxtl uxtl2 uzp1 uzp2 wfe wfi xar xpacd xpaci xpaclri
xtn xtn2 yield zip1 zip2
```

The Eigen Compiler Suite supports the ARM T32 instruction set as listed below and uses the same assembly syntax as predefined by the ARM architecture [14]. The only exception are immediate values which are not prefixed by a number sign.

```
adc adcs add adds addw adr aesd aese aesimc aesmc and ands asr asrs b bfc
bfi bic bics bkpt bl blx bx bxj cbnz cbz clrex clz cmn cmp cps cpsid cpsie
crc32b crc32cb crc32ch crc32cw crc32h crc32w dbg dcps1 dcps2 dcps3 dmb dsb
eor eors eret esb fldmdbx fldmiax fstmdbx fstmiax hlt hvc isb it lda ldab
ldaex ldaexb ldaexd ldaexh ldah ldc ldm ldmda ldmdb ldmea ldmed ldmfa ldmfd
ldmia ldmib ldr ldrb ldrbt ldrd ldrex ldrexb ldrexd ldrexh ldrh ldrht ldrsb
ldrsbt ldrsh ldrsht ldrt lsl lsls lsr lsrs mcr mcrr mla mlas mls mov movs
movt movw mrc mrrc mrs msr mul muls mvn mvns nop orn orns orr orrs pkhbt pkhtb
pld pldw pli pop push qadd qadd16 qadd8 qasx qdadd qdsub qsax qsub qsub16
qsub8 rbit rev rev16 revsh rfe rfeda rfedb rfeia rfeib ror rors rrx rrxs rsb
rsbs rsc rscs sadd16 sadd8 sasx sbc sbcs sbfx sdiv sel setend setpan sev sevl
```

```
sha1c sha1h sha1m sha1p sha1su0 sha1su1 sha256h sha256h2 sha256su0 sha256su1
shadd16 shadd8 shasx shsax shsub16 shsub8 smc smlabb smlabt smlad smladx smlal
smlalbb smlalbt smlald smlaldx smlals smlaltb smlaltt smlatb smlatt smlawb
smlawt smlsd smlsdx smlsld smlsldx smmla smmlar smmls smmlsr smmul smmulr
smuad smuadx smulbb smulbt smull smulls smultb smultt smulwb smulwt smusd
smusdx srs srsda srsdb srsia srsib ssat ssat16 ssax ssub16 ssub8 stc stl stlb
stlex stlexb stlexd stlexh stlh stm stmda stmdb stmea stmed stmfa stmfd stmia
stmib str strb strbt strd strex strexb strexd strexh strh strht strt sub subs
subw svc sxtab sxtab16 sxtah sxtb sxtb16 sxth tbb tbh teq tst uadd16 uadd8
uasx ubfx udf udiv uhadd16 uhadd8 uhasx uhsax uhsub16 uhsub8 umaal umlal
umlals umull umulls uqadd16 uqadd8 uqasx uqsax uqsub16 uqsub8 usad8 usada8
usat usat16 usax usub16 usub8 uxtab uxtab16 uxtah uxtb uxtb16 uxth vaba vabal
vabd vabdl vabs vacge vacgt vacle vaclt vadd vaddhn vaddl vaddw vand vbic
vbif vbit vbsl vcadd vceq vcge vcgt vcle vcls vclt vclz vcmla vcmp vcmpe vcnt
vcvt vcvta vcvtb vcvtm vcvtn vcvtp vcvtr vcvtt vdiv vdup veor vext vfma vfmal
vfms vfmsl vfnma vfnms vhadd vhsub vins vjcvt vld1 vld2 vld3 vld4 vldm vldmdb
vldmia vldr vmax vmaxnm vmin vminnm vmla vmlal vmls vmlsl vmov vmovl vmovn
vmovx vmrs vmsr vmul vmull vmvn vneg vnmla vnmls vnmul vorn vorr vpadal vpadd
vpaddl vpmax vpmin vpop vpush vqabs vqadd vqdmlal vqdmlsl vqdmulh vqdmull
vqmovn vqmovun vqneg vqrdmlah vqrdmlsh vqrdmulh vqrshl vqrshrn vqrshrun vqshl
vqshlu vqshrn vqshrun vqsub vraddhn vrecpe vrecps vrev16 vrev32 vrev64 vrhadd
vrinta vrintm vrintn vrintp vrintr vrintx vrintz vrshl vrshr vrshrn vrsqrte
vrsqrts vrsra vrsubhn vsdot vseleq vselge vselgt vselvs vshl vshll vshr vshrn
vsli vsqrt vsra vsri vst1 vst2 vst3 vst4 vstm vstmdb vstmia vstr vsub vsubhn
vsubl vsubw vswp vtbl vtbx vtrn vtst vudot vuzp vzip wfe wfi yield
```

8.4.3 AVR Instruction Set

The Eigen Compiler Suite supports the AVR instruction set as listed below and uses the same assembly syntax as predefined by Atmel [15]. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11.


```
adc add adiw and andi asr bclr bld brbc brbs brcc brcs break breq brge brhc
brhs brid brie brlo brlt brmi brne brpl brsh brtc brts brvc brvs bset bst call
cbi cbr clc clh cli cln clr cls clt clv clz com cp cpc cpi cpse dec des eicall
eijmp elpm eor fmul fmuls fmulsu icall ijmp in inc jmp lac las lat ld ldd ldi
lds lpm lsl lsr mov movw mul muls mulsu neg nop or ori out pop push rcall ret
reti rjmp rol ror sbc sbci sbi sbic sbis sbiw sbr sbrc sbrs sec seh sei sen
ser ses set sev sez sleep spm st std sts sub subi swap tst wdr xch
```

8.4.4 AVR32 Instruction Set

The Eigen Compiler Suite supports the AVR32 instruction set as listed below and uses the same assembly syntax as predefined by Atmel [16]. Instructions that operate on register lists are not supported. For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12.

```
abs acall acr adc add addabs addal addcc addcs addeq addge addgt addhh.w
addhi addhs addle addlo addls addlt addmi addne addpl addqs addvc addvs
and andal andcc andcs andeq andge andgt andh andhi andhs andl andle andlo
andls andlt andmi andn andne andpl andqs andvc andvs asr bfexts bfextu
bfins bld bral brcc brcs breakpoint breq brev brge brgt brhi brhs brle
brlo brls brlt brmi brne brpl brqs brvc brvs bst cache casts.b casts.h
castu.b castu.h cbr clz com cop cp.b cp.h cp.w cpc csrf csrfcz divs divu
eor eoral eorcc eorcs eoreq eorge eorgt eorh eorhi eorhs eorl eorle eorlo
eorls eorlt eormi eorne eorpl eorqs eorvc eorvs frs icall incjosp ld.d
ld.sb ld.sbal ld.sbcc ld.sbcs ld.sbeq ld.sbge ld.sbgt ld.sbhi ld.sbhs ld.sble
ld.sblo ld.sbls ld.sblt ld.sbmi ld.sbne ld.sbpl ld.sbqs ld.sbvc ld.sbvs ld.sh
ld.shal ld.shcc ld.shcs ld.sheq ld.shge ld.shgt ld.shhi ld.shhs ld.shle
ld.shlo ld.shls ld.shlt ld.shmi ld.shne ld.shpl ld.shqs ld.shvc ld.shvs
ld.ub ld.ubal ld.ubcc ld.ubcs ld.ubeq ld.ubge ld.ubgt ld.ubhi ld.ubhs ld.uble
ld.ublo ld.ubls ld.ublt ld.ubmi ld.ubne ld.ubpl ld.ubqs ld.ubvc ld.ubvs
ld.uh ld.uhal ld.uhcc ld.uhcs ld.uheq ld.uhge ld.uhgt ld.uhhi ld.uhhs ld.uhle
ld.uhlo ld.uhls ld.uhlt ld.uhmi ld.uhne ld.uhpl ld.uhqs ld.uhvc ld.uhvs ld.w
ld.wal ld.wcc ld.wcs ld.weq ld.wge ld.wgt ld.whi ld.whs ld.wle ld.wlo ld.wls
ld.wlt ld.wmi ld.wne ld.wpl ld.wqs ld.wvc ld.wvs ldc.d ldc.w ldc0.d ldc0.w
lddpc lddsp ldins.b ldins.h ldswp.sh ldswp.uh ldswp.w lsl lsr mac machh.d
machh.w macs.d macsathh.w macu.d macwh.d max mcall memc mems memt mfdr mfsr
min mov moval movcc movcs moveq movge movgt movh movhi movhs movl movle movlo
movls movlt movmi movne movpl movqs movvc movvs mtdr mtsr mul mulhh.w mulnhh.w
mulnwh.d muls.d mulsathh.h mulsathh.w mulsatrndhh.h mulsatrndhh.w mulsatwh.w
mulu.d mulwh.d musfr mustr mvcr.d mvcr.w mvrc.d mvrc.w neg nop or oral orcc
orcs oreq orge orgt orh orhi orhs orl orle orlo orls orlt ormi orne orpl
orqs orvc orvs pabs.sb pabs.sh packsh.sb packsh.ub packw.sh padd.b padd.h
paddh.sh paddh.ub padds.sb padds.sh padds.ub padds.uh paddsub.h paddsubh.sh
paddsubs.sh paddsubs.uh paddx.h paddxh.sh paddxs.sh paddxs.uh pasr.b pasr.h
pavg.sh pavg.ub plsl.b plsl.h plsr.b plsr.h pmax.sh pmax.ub pmin.sh pmin.ub
popjc pref psad psub.b psub.h psubadd.h psubaddh.sh psubadds.sh psubadds.uh
psubh.sh psubh.ub psubs.sb psubs.sh psubs.ub psubs.uh psubx.h psubxh.sh
psubxs.sh psubxs.uh punpcksb.h punpckub.h pushjc rcall retal retcc retcs
retd rete reteq retge retgt rethi reths retj retle retlo retls retlt retmi
retne retpl retqs rets retss retvc retvs rjmp rol ror rsub rsubal rsubcc
rsubcs rsubeq rsubge rsubgt rsubhi rsubhs rsuble rsublo rsubls rsublt rsubmi
rsubne rsubpl rsubqs rsubvc rsubvs satadd.h satadd.w satrnds satrndu sats
satsub.h satsub.w satu sbc sbr scall scr sleep sral srcc srcs sreq srge srgt
srhi srhs srle srlo srls srlt srmi srne srpl srqs srvc srvs sscall ssrf st.b
st.bal st.bcc st.bcs st.beq st.bge st.bgt st.bhi st.bhs st.ble st.blo st.bls
st.blt st.bmi st.bne st.bpl st.bqs st.bvc st.bvs st.d st.h st.hal st.hcc
st.hcs st.heq st.hge st.hgt st.hhi st.hhs st.hle st.hlo st.hls st.hlt st.hmi
st.hne st.hpl st.hqs st.hvc st.hvs st.w st.wal st.wcc st.wcs st.weq st.wge
st.wgt st.whi st.whs st.wle st.wlo st.wls st.wlt st.wmi st.wne st.wpl st.wqs
st.wvc st.wvs stc.d stc.w stc0.d stc0.w stcond stdsp sthh.w stswp.h stswp.w
sub subal subcc subcs subeq subfal subfcc subfcs subfeq subfge subfgt subfhi
subfhs subfle subflo subfls subflt subfmi subfne subfpl subfqs subfvc subfvs
subge subgt subhh.w subhi subhs suble sublo subls sublt submi subne subpl
subqs subvc subvs swap.b swap.bh swap.h sync tlbr tlbs tlbw tnbz tst xchg
```

8.4.5 M68000 Instruction Set

The Eigen Compiler Suite supports the M68000 instruction set as listed below and uses the same assembly syntax as predefined by Motorola [17]. The only exception are immediate values which are not prefixed by a number sign. For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13.

```
abcd add adda addi addq addx and andi asl asr bcc bchg bclr bcs beq bge bgt
bhi bhs ble blo bls blt bmi bne bpl bra bset bsr btst bvc bvs chk clr cmp cmpa
cmpi cmpm dbcc dbcs dbeq dbge dbgt dbhi dbhs dble dblo dbls dblt dbmi dbne
dbpl dbvc dbvs divs divu eor eori exg ext illegal jmp jsr lea link lsl lsr
move movea movep moveq muls mulu nbcd neg negx nop not or ori pea reset rol
ror roxl roxr rte rtr rts sbcd scc scs seq sf sge sgt shi shs sle slo sls slt
smi sne spl st stop sub suba subi subq subx svc svs swap tas trap trapv tst
unlk
```

8.4.6 MicroBlaze Instruction Set

The Eigen Compiler Suite supports the MicroBlaze instruction set as listed below and uses the same assembly syntax as predefined by Xilinx [18]. For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14.

```
add addc addi addic addik addikc addk addkc aget agetd and andi andn andni
aput aputd beq beqd beqi beqid bge bged bgei bgeid bgt bgtd bgti bgtid ble
bled blei bleid blt bltd blti bltid bne bned bnei bneid br bra brad brai braid
brald bralid brd bri brid brk brki brld brlid bsll bslli bsra bsrai bsrl bsrli
caget cagetd caput caputd cget cgetd clz cmp cmpu cput cputd eaget eagetd
ecaget ecagetd ecget ecgetd eget egetd fadd fcmp.eq fcmp.ge fcmp.gt fcmp.le
fcmp.lt fcmp.ne fcmp.un fdiv fint flt fmul frsub fsqrt get getd idiv idivu
imm lbu lbui lbur lhu lhui lhur lw lwi lwr lwx mbar mfs msrclr msrset mts
mul mulh mulhsu mulhu muli naget nagetd naput naputd ncaget ncagetd ncaput
ncaputd ncget ncgetd ncput ncputd neaget neagetd necaget necagetd necget
necgetd neget negetd nget ngetd nop nput nputd or ori pcmpbf pcmpeq pcmpne
put putd rsub rsubc rsubi rsubic rsubik rsubikc rsubk rsubkc rtbd rted rtid
rtsd sb sbi sbr sext16 sext8 sh shi shr sra src srl sw swapb swaph swi swr
swx taget tagetd taput taputd tcaget tcagetd tcaput tcaputd tcget tcgetd
tcput tcputd teaget teagetd tecaget tecagetd tecget tecgetd teget tegetd tget
tgetd tnaget tnagetd tnaput tnaputd tncaget tncagetd tncaput tncaputd tncget
tncgetd tncput tncputd tneaget tneagetd tnecaget tnecagetd tnecget tnecgetd
tneget tnegetd tnget tngetd tnput tnputd tput tputd wdc wdc.clear wdc.flush
wic xor xori
```

8.4.7 MIPS Instruction Set

The Eigen Compiler Suite supports the MIPS32 and MIPS64 instruction sets as listed below and uses the same assembly syntax as predefined by MIPS Technologies [19, 20]. For more

information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15.

```
abs.d abs.ps abs.s add add.d add.ps add.s addi addiu addu alnv.ps and andi b
bal bc1f bc1fl bc1t bc1tl bc2f bc2fl bc2t bc2tl beq beql bgez bgezal bgezall
bgezl bgtz bgtzl blez blezl bltz bltzal bltzall bltzl bne bnel break c.eq.d
c.eq.ps c.eq.s c.f.d c.f.ps c.f.s c.ole.d c.ole.ps c.ole.s c.olt.d c.olt.ps
c.olt.s c.ueq.d c.ueq.ps c.ueq.s c.ule.d c.ule.ps c.ule.s c.ult.d c.ult.ps
c.ult.s c.un.d c.un.ps c.un.s cache ceil.l.d ceil.l.s ceil.w.d ceil.w.s cfc1
cfc2 clo clz ctc1 ctc2 cvt.d.l cvt.d.s cvt.d.w cvt.l.d cvt.l.s cvt.ps.s
cvt.s.d cvt.s.l cvt.s.pl cvt.s.pu cvt.s.w cvt.w.d cvt.w.s dadd daddi daddiu
daddu dclo dclz ddiv ddivu deret dext dextm dextu di dins dinsm dinsu div
div.d div.s divu dmfc0 dmfc1 dmfc2 dmtc0 dmtc1 dmtc2 dmult dmultu drotr drotr32
drotrv dsbh dshd dsll dsll32 dsllv dsra dsra32 dsrav dsrl dsrl32 dsrlv dsub
dsubu ehb ei eret ext floor.l.d floor.l.s floor.w.d floor.w.s ins j jal jalr
jalr.hb jalx jr jr.hb lb lbu ld ldc1 ldc2 ldl ldr ldxc1 lh lhu li ll lld lui
luxc1 lw lwc1 lwc2 lwl lwr lwu lwxc1 madd madd.d madd.ps madd.s maddu mfc0
mfc1 mfc2 mfhc1 mfhc2 mfhi mflo mov.d mov.ps mov.s movf movf.d movf.ps movf.s
movn movn.d movn.ps movn.s movt movt.d movt.ps movt.s movz movz.d movz.ps
movz.s msub msub.d msub.ps msub.s msubu mtc0 mtc1 mtc2 mthc1 mthc2 mthi mtlo
mul mul.d mul.ps mul.s mult multu neg.d neg.ps neg.s nmadd.d nmadd.ps nmadd.s
nmsub.d nmsub.ps nmsub.s nop nor or ori pause pll.ps plu.ps pref prefx pul.ps
puu.ps rdhdwr rdpgpr recip.d recip.s rotr rotrv round.l.d round.l.s round.w.d
round.w.s rsqrt.d rsqrt.s sb sc scd sd sdbbp sdc1 sdc2 sdl sdr sdxc1 seb seh
sh sll sllv slt slti sltiu sltu sqrt.d sqrt.s sra srav srl srlv ssnop sub
sub.d sub.ps sub.s subu suxc1 sw swc1 swc2 swl swr swxc1 sync syscall teq
teqi tge tgei tgeiu tgeu tlbp tlbr tlbwi tlbwr tlt tlti tltiu tltu tne tnei
trunc.l.d trunc.l.s trunc.w.d trunc.w.s wait wrpgpr wsbh xor xori
```

8.4.8 MMIX Instruction Set

The Eigen Compiler Suite supports the MMIX instruction set as listed below and uses the same assembly syntax as predefined by Donald E. Knuth [21]. The only exception are immediate values which are not prefixed by a number sign. For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16.

```
16addu 16addui 2addu 2addui 4addu 4addui 8addu 8addui add addi addu addui and
andi andn andnh andni andnl andnmh andnml bdif bdifi bev bevb bn bnb bnn bnnb
bnp bnpb bnz bnzb bod bodb bp bpb bz bzb cmp cmpi cmpu cmpui csev csevi csn
csni csnn csnni csnp csnpi csnz csnzi csod csodi csp cspi cswap cswapi csz
cszi div divi divu divui fadd fcmp fcmpe fdiv feql feqle fint fix fixu flot
floti flotu flotui fmul frem fsqrt fsub fun fune get geta getab go goi inch
incl incmh incml jmp jmpb ldb ldbi ldbu ldbui ldht ldhti ldo ldoi ldou ldoui
ldsf ldsfi ldt ldti ldtu ldtui ldunc ldunci ldvts ldvtsi ldw ldwi ldwu ldwui
mor mori mul muli mulu mului mux muxi mxor mxori nand nandi neg negi negu
negui nor nori nxor nxori odif odifi or orh ori orl ormh orml orn orni pbev
pbevb pbn pbnb pbnn pbnnb pbnp pbnpb pbnz pbnzb pbod pbodb pbp pbpb pbz pbzb
pop prego pregoi preld preldi prest presti pushgo pushgoi pushj pushjb put
puti resume sadd saddi save seth setl setmh setml sflot sfloti sflotu sflotui
```

```
sl sli slu slui sr sri sru srui stb stbi stbu stbui stco stcoi stht sthti
sto stoi stou stoui stsf stsfi stt stti sttu sttui stunc stunci stw stwi stwu
stwui sub subi subu subui swym sync syncd syncdi syncid syncidi tdif tdifi
trap trip unsave wdif wdifi xor xori zsev zsevi zsn zsni zsnn zsnni zsnp zsnpi
zsnz zsnzi zsod zsodi zsp zspi zsz zszi
```

8.4.9 OpenRISC 1000 Instruction Set

The Eigen Compiler Suite supports the OpenRISC 1000 instruction set as listed below and uses the same assembly syntax as predefined by OpenCores [22]. For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17.

```
l.add l.addc l.addi l.addic l.and l.andi l.bf l.bnf l.cmov l.csync l.div
l.divu l.extbs l.extbz l.exths l.exthz l.extws l.extwz l.ff1 l.fl1 l.j
l.jal l.jalr l.jr l.lbs l.lbz l.ld l.lhs l.lhz l.lwa l.lws l.lwz l.mac
l.maci l.macrc l.macu l.mfspr l.movhi l.msb l.msbu l.msync l.mtspr l.mul
l.muld l.muldu l.muli l.mulu l.nop l.or l.ori l.psync l.rfe l.ror l.rori
l.sb l.sd l.sfeq l.sfeqi l.sfges l.sfgesi l.sfgeu l.sfgeui l.sfgts l.sfgtsi
l.sfgtu l.sfgtui l.sfles l.sflesi l.sfleu l.sfleui l.sflts l.sfltsi l.sfltu
l.sfltui l.sfne l.sfnei l.sh l.sll l.slli l.sra l.srai l.srl l.srli l.sub
l.sw l.swa l.sys l.trap l.xor l.xori lf.add.d lf.add.s lf.div.d lf.div.s
lf.ftoi.d lf.ftoi.s lf.itof.d lf.itof.s lf.madd.d lf.madd.s lf.mul.d lf.mul.s
lf.rem.d lf.rem.s lf.sfeq.d lf.sfeq.s lf.sfge.d lf.sfge.s lf.sfgt.d lf.sfgt.s
lf.sfle.d lf.sfle.s lf.sflt.d lf.sflt.s lf.sfne.d lf.sfne.s lf.sub.d lf.sub.s
lv.add.b lv.add.h lv.adds.b lv.adds.h lv.addu.b lv.addu.h lv.addus.b
lv.addus.h lv.all_eq.b lv.all_eq.h lv.all_ge.b lv.all_ge.h lv.all_gt.b
lv.all_gt.h lv.all_le.b lv.all_le.h lv.all_lt.b lv.all_lt.h lv.all_ne.b
lv.all_ne.h lv.and lv.any_eq.b lv.any_eq.h lv.any_ge.b lv.any_ge.h
lv.any_gt.b lv.any_gt.h lv.any_le.b lv.any_le.h lv.any_lt.b lv.any_lt.h
lv.any_ne.b lv.any_ne.h lv.avg.b lv.avg.h lv.cmp_eq.b lv.cmp_eq.h lv.cmp_ge.b
lv.cmp_ge.h lv.cmp_gt.b lv.cmp_gt.h lv.cmp_le.b lv.cmp_le.h lv.cmp_lt.b
lv.cmp_lt.h lv.cmp_ne.b lv.cmp_ne.h lv.madds.h lv.max.b lv.max.h lv.merge.b
lv.merge.h lv.min.b lv.min.h lv.msubs.h lv.muls.h lv.nand lv.nor lv.or
lv.pack.b lv.pack.h lv.packs.b lv.packs.h lv.packus.b lv.packus.h lv.perm.n
lv.rl.b lv.rl.h lv.sll lv.sll.b lv.sll.h lv.sra.b lv.sra.h lv.srl lv.srl.b
lv.srl.h lv.sub.b lv.sub.h lv.subs.b lv.subs.h lv.subu.b lv.subu.h lv.subus.b
lv.subus.h lv.unpack.b lv.unpack.h lv.xor
```

8.4.10 PowerPC Instruction Set

The Eigen Compiler Suite supports the PowerPC instruction set as listed below and uses the same assembly syntax as predefined by IBM [23]. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18.

```
add add. addc addc. addco addco. adde adde. addeo addeo. addi addic addic.
addis addme addme. addmeo addmeo. addo addo. addze addze. addzeo addzeo. and
and. andc andc. andi. andis. b ba bc bca bcctr bcctrl bcl bcla bclr bclrl beq
beqa beql beqla bge bgea bgel bgela bgt bgta bgtl bgtla bl bla ble blea blel
```

```
blela blt blta bltl bltla bne bnea bnel bnela bng bnga bngl bngla bnl bnla
bnll bnlla bns bnsa bnsl bnsla bnu bnua bnul bnula bso bsoa bsol bsola bun
buna bunl bunla cmp cmpd cmpdi cmpi cmpl cmpld cmpldi cmpli cmplw cmplwi cmpw
cmpwi cntlzd cntlzd. cntlzw cntlzw. crand crandc creqv crnand crnor cror crorc
crxor dcbf dcbi dcbst dcbt dcbtst dcbz divd divd. divdo divdo. divdu divdu.
divduo divduo. divw divw. divwo divwo. divwu divwu. divwuo divwuo. eciwx ecowx
eieio eqv eqv. extsb extsb. extsh extsh. extsw extsw. fabs fabs. fadd fadd.
fadds fadds. fcfid fcfid. fcmpo fcmpu fctid fctid. fctidz fctidz. fctiw fctiw.
fctiwz fctiwz. fdiv fdiv. fdivs fdivs. fmadd fmadd. fmadds fmadds. fmr fmr.
fmsub fmsub. fmsubs fmsubs. fmul fmul. fmuls fmuls. fnabs fnabs. fneg fneg.
fnmadd fnmadd. fnmadds fnmadds. fnmsub fnmsub. fnmsubs fnmsubs. fres fres.
frsp frsp. frsqrte frsqrte. fsel fsel. fsqrt fsqrt. fsqrts fsqrts. fsub fsub.
fsubs fsubs. icbi isync la lbz lbzu lbzux lbzx ld ldarx ldu ldux ldx lfd
lfdu lfdux lfdx lfs lfsu lfsux lfsx lha lhau lhaux lhax lhbrx lhz lhzu lhzux
lhzx li lis lmw lswi lswx lwa lwarx lwaux lwax lwbrx lwz lwzu lwzux lwzx
mcrf mcrfs mcrxr mfcr mffs mffs. mfmsr mfocrf mfspr mfsr mfsrin mftb mr mr.
mtcrf mtfsb0 mtfsb0. mtfsb1 mtfsb1. mtfsf mtfsf. mtfsfi mtfsfi. mtmsr mtmsrd
mtocrf mtspr mtsr mtsrd mtsrdin mtsrin mulhd mulhd. mulhdu mulhdu. mulhw mulhw.
mulhwu mulhwu. mulld mulld. mulldo mulldo. mulli mullw mullw. mullwo mullwo.
nand nand. neg neg. nego nego. nop nor nor. not not. or or. orc orc. ori
oris rfi rfid rldcl rldcl. rldcr rldcr. rldic rldic. rldicl rldicl. rldicr
rldicr. rldimi rldimi. rlwimi rlwimi. rlwinm rlwinm. rlwnm rlwnm. sc slbia
slbie slbmfee slbmfev slbmte sld sld. slw slw. srad srad. sradi sradi. sraw
sraw. srawi srawi. srd srd. srw srw. stb stbu stbux stbx std stdcx. stdu stdux
stdx stfd stfdu stfdux stfdx stfiwx stfs stfsu stfsux stfsx sth sthbrx sthu
sthux sthx stmw stswi stswx stw stwbrx stwcx. stwu stwux stwx sub sub. subc
subc. subco subco. subf subf. subfc subfc. subfco subfco. subfe subfe. subfeo
subfeo. subfic subfme subfme. subfmeo subfmeo. subfo subfo. subfze subfze.
subfzeo subfzeo. subi subic subic. subis subo subo. sync td tdi tlbia tlbie
tlbiel tlbsync tw twi xor xor. xori xoris
```


8.4.11 RISC Instruction Set

The Eigen Compiler Suite supports the RISC instruction set as listed below and uses the same assembly syntax as predefined by Niklaus Wirth [24]. For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19.

```
adc add and ann asr b bcc bcs beq bge bgt bhi bl blcc blcs ble bleq blge blgt
blhi blle blls bllt blmi blne blpl bls blt blvc blvs bmi bne bpl bvc bvs cli
div fad fdv fml fsb ior ld ldb lsl mlu mov mul mvf mvh mvs nop ror rti sbc st
stb sti sub xor
```

8.4.12 WebAssembly Instruction Set

The Eigen Compiler Suite supports the WebAssembly instruction set as listed below and uses the same assembly syntax as predefined by the World Wide Web Consortium (W3C) [25]. The only exception are the addition of the pseudo instructions `i32`, `label`, `lane`, `s32`, `u32`, and

valtype. They encode an immediate value of the respective type with a fixed size and can be used as a replacement for operands of instructions that require immediate values of that type. For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20.

```
block br br_if br_table call call_indirect data.drop drop elem.drop else
end f32.abs f32.add f32.ceil f32.const f32.convert_i32_s f32.convert_-
i32_u f32.convert_i64_s f32.convert_i64_u f32.copysign f32.demote_f64
f32.div f32.eq f32.floor f32.ge f32.gt f32.le f32.load f32.lt f32.max
f32.min f32.mul f32.ne f32.nearest f32.neg f32.reinterpret_i32 f32.sqrt
f32.store f32.sub f32.trunc f32x4.abs f32x4.add f32x4.ceil f32x4.convert_-
i32x4_s f32x4.convert_i32x4_u f32x4.demote_f64x2_zero f32x4.div f32x4.eq
f32x4.extract_lane f32x4.floor f32x4.ge f32x4.gt f32x4.le f32x4.lt f32x4.max
f32x4.min f32x4.mul f32x4.ne f32x4.nearest f32x4.neg f32x4.pmax f32x4.pmin
f32x4.replace_lane f32x4.splat f32x4.sqrt f32x4.sub f32x4.trunc f64.abs
f64.add f64.ceil f64.const f64.convert_i32_s f64.convert_i32_u f64.convert_-
i64_s f64.convert_i64_u f64.copysign f64.div f64.eq f64.floor f64.ge f64.gt
f64.le f64.load f64.lt f64.max f64.min f64.mul f64.ne f64.nearest f64.neg
f64.promote_f32 f64.reinterpret_i64 f64.sqrt f64.store f64.sub f64.trunc
f64x2.abs f64x2.add f64x2.ceil f64x2.convert_low_i32x4_s f64x2.convert_-
low_i32x4_u f64x2.div f64x2.eq f64x2.extract_lane f64x2.floor f64x2.ge
f64x2.gt f64x2.le f64x2.lt f64x2.max f64x2.min f64x2.mul f64x2.ne
f64x2.nearest f64x2.neg f64x2.pmax f64x2.pmin f64x2.promote_low_f32x4
f64x2.replace_lane f64x2.splat f64x2.sqrt f64x2.sub f64x2.trunc global.get
global.set i16x8.abs i16x8.add i16x8.add_sat_s i16x8.add_sat_u i16x8.all_-
true i16x8.avgr_u i16x8.bitmask i16x8.eq i16x8.extadd_pairwise_i8x16_s
i16x8.extadd_pairwise_i8x16_u i16x8.extend_high_i8x16_s i16x8.extend_high_-
i8x16_u i16x8.extend_low_i8x16_s i16x8.extend_low_i8x16_u i16x8.extmul_high_-
i8x16_s i16x8.extmul_high_i8x16_u i16x8.extmul_low_i8x16_s i16x8.extmul_-
low_i8x16_u i16x8.extract_lane_s i16x8.extract_lane_u i16x8.ge_s i16x8.ge_u
i16x8.gt_s i16x8.gt_u i16x8.le_s i16x8.le_u i16x8.lt_s i16x8.lt_u i16x8.max_-
s i16x8.max_u i16x8.min_s i16x8.min_u i16x8.mul i16x8.narrow_i32x4_s
i16x8.narrow_i32x4_u i16x8.ne i16x8.neg i16x8.q15mulr_sat_s i16x8.replace_-
lane i16x8.shl i16x8.shr_s i16x8.shr_u i16x8.splat i16x8.sub i16x8.sub_-
sat_s i16x8.sub_sat_u i32 i32.add i32.and i32.clz i32.const i32.ctz
i32.div_s i32.div_u i32.eq i32.eqz i32.extend16_s i32.extend8_s i32.ge_-
s i32.ge_u i32.gt_s i32.gt_u i32.le_s i32.le_u i32.load i32.load16_s
i32.load16_u i32.load8_s i32.load8_u i32.lt_s i32.lt_u i32.mul i32.ne i32.or
i32.popcnt i32.reinterpret_f32 i32.rem_s i32.rem_u i32.rotl i32.rotr i32.shl
i32.shr_s i32.shr_u i32.store i32.store16 i32.store8 i32.sub i32.trunc_-
f32_s i32.trunc_f32_u i32.trunc_f64_s i32.trunc_f64_u i32.trunc_sat_f32_-
s i32.trunc_sat_f32_u i32.trunc_sat_f64_s i32.trunc_sat_f64_u i32.wrap_-
i64 i32.xor i32x4.abs i32x4.add i32x4.all_true i32x4.bitmask i32x4.dot_-
i16x8_s i32x4.eq i32x4.extadd_pairwise_i16x8_s i32x4.extadd_pairwise_i16x8_u
i32x4.extend_high_i16x8_s i32x4.extend_high_i16x8_u i32x4.extend_low_i16x8_-
s i32x4.extend_low_i16x8_u i32x4.extmul_high_i16x8_s i32x4.extmul_high_-
i16x8_u i32x4.extmul_low_i16x8_s i32x4.extmul_low_i16x8_u i32x4.extract_lane
i32x4.ge_s i32x4.ge_u i32x4.gt_s i32x4.gt_u i32x4.le_s i32x4.le_u i32x4.lt_s
i32x4.lt_u i32x4.max_s i32x4.max_u i32x4.min_s i32x4.min_u i32x4.mul i32x4.ne
i32x4.neg i32x4.replace_lane i32x4.shl i32x4.shr_s i32x4.shr_u i32x4.splat
```

```
i32x4.sub i32x4.trunc_sat_f32x4_s i32x4.trunc_sat_f32x4_u i32x4.trunc_sat_-
f64x2_s_zero i32x4.trunc_sat_f64x2_u_zero i64.add i64.and i64.clz i64.const
i64.ctz i64.div_s i64.div_u i64.eq i64.eqz i64.extend16_s i64.extend32_s
i64.extend8_s i64.extend_i32_s i64.extend_i32_u i64.ge_s i64.ge_u i64.gt_s
i64.gt_u i64.le_s i64.le_u i64.load i64.load16_s i64.load16_u i64.load32_-
s i64.load32_u i64.load8_s i64.load8_u i64.lt_s i64.lt_u i64.mul i64.ne
i64.or i64.popcnt i64.reinterpret_f64 i64.rem_s i64.rem_u i64.rotl i64.rotr
i64.shl i64.shr_s i64.shr_u i64.store i64.store16 i64.store32 i64.store8
i64.sub i64.trunc_f32_s i64.trunc_f32_u i64.trunc_f64_s i64.trunc_f64_u
i64.trunc_sat_f32_s i64.trunc_sat_f32_u i64.trunc_sat_f64_s i64.trunc_sat_-
f64_u i64.xor i64x2.abs i64x2.add i64x2.all_true i64x2.bitmask i64x2.eq
i64x2.extend_high_i32x4_s i64x2.extend_high_i32x4_u i64x2.extend_low_i32x4_-
s i64x2.extend_low_i32x4_u i64x2.extmul_high_i32x4_s i64x2.extmul_high_-
i32x4_u i64x2.extmul_low_i32x4_s i64x2.extmul_low_i32x4_u i64x2.extract_-
lane i64x2.ge_s i64x2.gt_s i64x2.le_s i64x2.lt_s i64x2.mul i64x2.ne
i64x2.neg i64x2.replace_lane i64x2.shl i64x2.shr_s i64x2.shr_u i64x2.splat
i64x2.sub i8x16.abs i8x16.add i8x16.add_sat_s i8x16.add_sat_u i8x16.all_true
i8x16.avgr_u i8x16.bitmask i8x16.eq i8x16.extract_lane_s i8x16.extract_lane_-
u i8x16.ge_s i8x16.ge_u i8x16.gt_s i8x16.gt_u i8x16.le_s i8x16.le_u i8x16.lt_-
s i8x16.lt_u i8x16.max_s i8x16.max_u i8x16.min_s i8x16.min_u i8x16.narrow_-
i16x8_s i8x16.narrow_i16x8_u i8x16.ne i8x16.neg i8x16.popcnt i8x16.replace_-
lane i8x16.shl i8x16.shr_s i8x16.shr_u i8x16.shuffle i8x16.splat i8x16.sub
i8x16.sub_sat_s i8x16.sub_sat_u i8x16.swizzle if label lane local.get
local.set local.tee loop memory.copy memory.fill memory.grow memory.init
memory.size nop ref.func ref.is_null ref.null return s32 select table.copy
table.fill table.get table.grow table.init table.set table.size u32
unreachable v128.and v128.andnot v128.any_true v128.bitselect v128.const
v128.load v128.load16_lane v128.load16_splat v128.load16x4_s v128.load16x4_-
u v128.load32_lane v128.load32_splat v128.load32_zero v128.load32x2_-
s v128.load32x2_u v128.load64_lane v128.load64_splat v128.load64_zero
v128.load8_lane v128.load8_splat v128.load8x8_s v128.load8x8_u v128.not
v128.or v128.store v128.store16_lane v128.store32_lane v128.store64_lane
v128.store8_lane v128.xor valtype vec
```


8.4.13 Xtensa Instruction Set

The Eigen Compiler Suite supports the Xtensa instruction set as listed below and uses the same assembly syntax as predefined by Cadence Design Systems [26]. For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21.

```
abs abs.d abs.s add add.d add.n add.s addexp.d addexp.s addexpm.d addexpm.s
addi addi.n addmi addx2 addx4 addx8 all4 all8 and andb andbc any4 any8 ball
bany bbc bbci bbci.l bbs bbsi bbsi.l beq beqi beqz beqz.n bf bge bgei bgeu
bgeui bgez blt blti bltu bltui bltz bnall bne bnei bnez bnez.n bnone break
break.n bt call0 call12 call4 call8 callx0 callx12 callx4 callx8 ceil.d ceil.s
clamps clrex const.d const.s const16 cvtd.s cvts.d dci dcwb dcwbi depbits
dhi dhi.b dhu dhwb dhwb.b dhwbi dhwbi.b dii diu div0.d div0.s divn.d divn.s
diwb diwbi diwbui.p dpfl dpfm.b dpfm.bf dpfr dpfr.b dpfr.bf dpfro dpfw dpfw.b
dpfw.bf dpfwo dsync entry esync excw extui extw float.d float.s floor.d
```

```
floor.s fsync getex idtlb ihi ihu iii iitlb iiu ill ill.n ipf ipfl isync j
j.l jx l16si l16ui l32ai l32e l32ex l32i l32i.n l32r l8ui ldct ldcw lddec
lddr32.p ldi ldinc ldip ldx ldxp loop loopgtz loopnez lsi lsip lsiu lsx lsxp
lsxu madd.d madd.s maddn.d maddn.s max maxu memw min minu mkdadj.d mkdadj.s
mksadj.d mksadj.s mov mov.d mov.n mov.s moveqz moveqz.d moveqz.s movf movf.d
movf.s movgez movgez.d movgez.s movi movi.n movltz movltz.d movltz.s movnez
movnez.d movnez.s movsp movt movt.d movt.s msub.d msub.s mul.aa.hh mul.aa.hl
mul.aa.lh mul.aa.ll mul.ad.hh mul.ad.hl mul.ad.lh mul.ad.ll mul.d mul.da.hh
mul.da.hl mul.da.lh mul.da.ll mul.dd.hh mul.dd.hl mul.dd.lh mul.dd.ll
mul.s mul16s mul16u mula.aa.hh mula.aa.hl mula.aa.lh mula.aa.ll mula.ad.hh
mula.ad.hl mula.ad.lh mula.ad.ll mula.da.hh mula.da.hh.lddec mula.da.hh.ldinc
mula.da.hl mula.da.hl.lddec mula.da.hl.ldinc mula.da.lh mula.da.lh.lddec
mula.da.lh.ldinc mula.da.ll mula.da.ll.lddec mula.da.ll.ldinc
mula.dd.hh mula.dd.hh.lddec mula.dd.hh.ldinc mula.dd.hl mula.dd.hl.lddec
mula.dd.hl.ldinc mula.dd.lh mula.dd.lh.lddec mula.dd.lh.ldinc mula.dd.ll
mula.dd.ll.lddec mula.dd.ll.ldinc mull muls.aa.hh muls.aa.hl muls.aa.lh
muls.aa.ll muls.ad.hh muls.ad.hl muls.ad.lh muls.ad.ll muls.da.hh muls.da.hl
muls.da.lh muls.da.ll muls.dd.hh muls.dd.hl muls.dd.lh muls.dd.ll mulsh
muluh neg neg.d neg.s nexp01.d nexp01.s nop nop.n nsa nsau oeq.d oeq.s ole.d
ole.s olt.d olt.s or orb orbc pdtlb pitlb pptlb quos quou rdtlb0 rdtlb1
recip0.d recip0.s rems remu rer ret ret.n retw retw.n rfdd rfde rfdo rfe rfi
rfme rfr rfrd rfue rfwo rfwu ritlb0 ritlb1 rotw round.d round.s rptlb0 rptlb1
rsil rsqrt0.d rsqrt0.s rsr rsync rur s16i s32c1i s32e s32ex s32i s32i.n s32nb
s32ri s8i salt saltu sddr32.p sdi sdip sdx sdxp sext sict sicw simcall sll
slli sqrt0.d sqrt0.s sra srai src srl srli ssa8b ssa8l ssai ssi ssip ssiu ssl
ssr ssx ssxp ssxu sub sub.d sub.s subx2 subx4 subx8 syscall trunc.d trunc.s
ueq.d ueq.s ufloat.d ufloat.s ule.d ule.s ult.d ult.s umul.aa.hh umul.aa.hl
umul.aa.lh umul.aa.ll un.d un.s utrunc.d utrunc.s waiti wdtlb wer wfr wfrd
witlb wptlb wsr wur xor xorb xsr
```

8.4.14 Intermediate Code Instruction Set

Intermediate code, as exposed by several tools of the Eigen Compiler Suite for debugging purposes, can also be represented using the generic assembly language. The corresponding instruction set is listed below. For more information about intermediate code and its purpose, see Chapter 23.

```
add alias and array asm br break breq brge brlt brne call conv copy def div
enter enum field fill fix func jump leave loc lsh mod mov mul neg nop not or
pop ptr push rec ref req res ret rsh sub sym trap type unfix value void xor
```

8.5 Directives

The generic assembly language defines a common set of directives for all concrete assemblers. In contrast to instructions, directives do not depend on the instruction set defined by the target hardware architecture. Tables 8.3 and 8.4 summarize all of these directives and the category they belong to.

Category	Mnemonic	See Section
Section Creation	`.code`	8.5.8
	`.initcode`	8.5.25
	`.initdata`	8.5.26
	`.data`	8.5.10
	`.const`	8.5.9
	`.header`	8.5.23
	`.trailer`	8.5.41
Section Options	`.default`	8.5.12
	`.phantom`	8.5.32
	`.required`	8.5.36
	`.shared`	8.5.38
Section Placement	`.alignment`	8.5.3
	`.origin`	8.5.30
	`.group`	8.5.22
Data Definition	`.byte`	8.5.7
	`.dbyte`	8.5.11
	`.tbyte`	8.5.39
	`.qbyte`	8.5.33
	`.obyte`	8.5.29
	`.embed`	8.5.16
Memory Layout	`.pad`	8.5.31
	`.align`	8.5.2
	`.reserve`	8.5.37
Mode Selection	`.big`	8.5.5
	`.little`	8.5.28
	`.bitmode`	8.5.6
Special Purpose	`.alias`	8.5.1
	`.assert`	8.5.4
	`.equals`	8.5.21
	`.require`	8.5.35
	`.trace`	8.5.40

Table 8.3: Directives of the generic assembly language

Category	Mnemonic	See Section
Code Control	`#end`	8.5.17
	`#line`	8.5.27
Conditional Inclusion	`#if`	8.5.24
	`#elif`	8.5.14
	`#else`	8.5.15
	`#endif`	8.5.19
Code Repetition	`#repeat`	8.5.34
	`#endrep`	8.5.20
Macro Definition	`#define`	8.5.13
	`#enddef`	8.5.18
	`#undef`	8.5.42

Table 8.4: Preprocessing directives of the generic assembly language

Directives usually do not generate any machine code, they rather modify the generation of binary code and data itself. They also allow describing characteristics of the current section or creating new sections. All settings and modifications performed by directives apply only to the current section and are reset afterward. Preprocessing directives on the other hand allow controlling the handling of the source code itself like repeating or conditionally including some portions of code.

The Eigen Compiler Suite features compilers for programming languages that allow users to write so-called *inline assembly code* within standard program code. Behind the scenes, compilers usually generate a code section for each functional unit of the implemented programming language. Inline assembly code allows users to describe a part of this code section using the generic assembly language. Since the placement and options of the section are still managed by the compiler however, directives for modifying these properties or creating new sections are not available in inline assembly code.

8.5.1 `.alias` *Identifier* — Alias Name

The alias name directive assigns an additional name for the data or code at the current position in the section. This data or code can be referred to by the name of the current section plus its position or simply by the new alias name. Alias names may not be duplicated and must differ from the name of the section. This directive may not be labeled.

8.5.2 `.align` *Expression* — Data Alignment

The data alignment directive allows padding binary data space in order to accommodate alignment constraints. It advances the current position in the section to the next multiple of the alignment specified by the operand. It is expressed in octets and has to be a positive power of two. This directive may not be labeled.

8.5.3 `.alignment` *Expression* — Section Alignment

The section alignment directive specifies the alignment of the section. The alignment defines the actual address of an aligned section in multiples of its alignment and may only be specified once and only if the section has no origin, see Section 8.5.30. It is expressed in multiples of octets and has to be a positive power of two. This directive is not available in inline assembly code and may not be labeled.

8.5.4 `.assert` *Expression* — Static Assertion

The static assertion directive allows asserting conditions during the translation of assembly code. It tests the Boolean value of its operand and aborts the translation if the condition is not satisfied. This directive may not be labeled.

8.5.5 `.big` — Big-Endian Mode

The big-endian mode directive changes the ordering of generated binary data to most significant octet first. Some hardware architectures allow changing the endianness of machine code, in which case this ordering also applies to the binary encoding of instructions. This directive may not be labeled and must restore the original endianness mode if it was modified in inline assembly code.

8.5.6 `.bitmode` *Expression* — Bit Mode

The bit mode directive allows changing the bit mode that is used to translate the instructions of the current section into machine code. The set of valid bit modes depends on the hardware architecture in use, where some architectures do not define bit modes at all. Users have to refer to the documentation of the respective target hardware architecture. This directive may not be labeled and must restore the original bit mode if it was modified in inline assembly code.

8.5.7 `.byte` *Expressions* — Byte Data

The byte data directive allows generating the binary representation of the value of one or more operands. Each value is represented using a single octet. If an operand for this directive is a string, each character of the string is evaluated individually.

8.5.8 `.code` *Identifier* — Standard Code Section

The standard code section directive creates a new standard code section with the specified name. See Section 8.2.2 for information about standard code sections. All labels as well as instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.9 `.const` *Identifier* — Constant Data Section

The constant data section directive creates a new constant data section with the specified name. See Section 8.2.2 for information about constant data sections. All labels as well as instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.10 `.data` *Identifier* — Standard Data Section

The standard data section directive creates a new standard data section with the specified name. See Section 8.2.2 for information about standard data sections. All labels as well as instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.11 `.dbyte` *Expressions* — Double Byte Data

The double byte data directive allows generating the binary representation of the value of one or more operands. Each value is represented using two octets, where the current endianness mode defines the ordering of the most and least significant one. If an operand for this directive is a string, each character of the string is evaluated individually.

8.5.12 `.default` — Default Section

The default section directive marks the current section as a default section. See Section 8.2.3 for information about default sections. This directive may only be specified once, is not available in inline assembly code and may not be labeled.

8.5.13 `#define` *Identifier* — Begin Macro Definition

The begin macro definition preprocessing directive allows defining pseudo instructions. Macros have a unique name and contain any sequence of assembly code until the next end macro definition directive. As long as a macro is defined, its name can be used as the mnemonic of a new pseudo instruction that takes up to ten operands separated by commas. This instruction is then replaced by the code sequence encompassed by the corresponding macro in a process called macro expansion. While expanding, number sign characters followed by a single decimal digit within any character or string literal, identifier, address, or label are replaced by the operand identified by that zero-based index. Two consecutive sign characters are replaced by the decimal line number of the macro expansion and allow defining unique labels. This directive may not be labeled and requires a subsequent end macro definition directive.

8.5.14 `#elif` *Expression* — Conditional Else-If

The conditional else-if preprocessing directive allows conditional inclusions of instructions and directives during the translation of assembly code. It tests the Boolean value of its operand and skips subsequent code until the next conditional else or end-if directive if the condition is not satisfied or any preceding conditional if or else-if directives did not skip code. This directive may not be labeled and requires a preceding conditional if directive and a subsequent conditional end-if directive.

8.5.15 `#else` — Conditional Else

The conditional else preprocessing directive allows excluding instructions and directives during the translation of assembly code. It skips subsequent code until the next conditional end-if directive if any preceding conditional if or else-if directives did not skip code. This directive may not be labeled and requires a preceding conditional if or else-if directive and a subsequent conditional end-if directive.

8.5.16 `.embed` *String* — Embed Binary File

The embed binary file directive allows copying the contents of a binary file named by the string operand into the current section.

8.5.17 `#end` — End Code

The end code preprocessing directive stops the translation of the assembly code. The remaining part of the source code is completely ignored. This directive may not be labeled.

8.5.18 `#enddef` End Macro Definition

The end macro definition preprocessing directive allows specifying the end of the assembly code sequence encompassed by the current macro definition. This directive may not be labeled and requires a preceding begin macro definition directive.

8.5.19 `#endif` Conditional End-If

The conditional end-if preprocessing directive allows ending the conditional inclusion of instructions and directives during the translation of assembly code. This directive may not be labeled and requires a preceding conditional if, else-if, or else directive.

8.5.20 `#endrep` End Repetition

The end repetition preprocessing directive allows specifying the end of the assembly code sequence encompassed by the current code repetition. This directive may not be labeled and requires a preceding begin repetition directive.

8.5.21 `.equals` *Expression* Constant Definition

The constant definition directive allows defining symbolic constants within the current section. Constant definitions require a label which can be used to refer to the corresponding constant expression by name. Constant definitions with the same name are allowed if the lexical token sequences of their operands are also identical.

8.5.22 `.group` *Identifier* Section Group

The section group directive allows grouping several sections with the same type and alignment by aligning them adjacent to each other. Consecutive section group directives define nested groups in order of occurrence, but they may not be duplicated and must differ from the name of the section. A section group is treated as the name of a virtual section that contains all sections and nested groups in that group. The groups themselves are placed consecutively in lexicographic order followed by all remaining sections of the same type. This directive may not be labeled.

8.5.23 `.header` *Identifier* Heading Metadata Section

The heading metadata section directive creates a new heading metadata section with the specified name. See Section 8.2.2 for information about heading metadata sections. All labels as well as instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.24 `#if` *Expression* — Conditional If

The conditional if preprocessing directive allows conditional inclusions of instructions and directives during the translation of assembly code. It tests the Boolean value of its operand and skips subsequent code until the next conditional else-if, else, or end-if directive if the condition is not satisfied. This directive may not be labeled and requires a subsequent conditional end-if directive.

8.5.25 `.initcode` *Identifier* — Initializing Code Section

The initializing code section directive creates a new initializing code section with the specified name. See Section 8.2.2 for information about initializing code sections. All labels as well as instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.26 `.initdata` *Identifier* — Data Initializing Code Section

The data initializing code section directive creates a new data initializing code section with the specified name. See Section 8.2.2 for information about data initializing code sections. All labels as well as instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.27 `#line` *Integer* $String_{opt}$ — Line Control

The line control preprocessing directive allow overwriting the filename of the source code and the position therein used in diagnostic messages. The second operand is optional and specifies the filename of the source code. The remaining part of the source code is treated as if it would start in that file at the line specified by the first operand.

8.5.28 `.little` — Little-Endian Mode

The little-endian mode directive changes the ordering of generated binary data to least significant octet first. Some hardware architectures allow changing the endianness of machine code, in which case this ordering also applies to the binary encoding of instructions. This directive may not be labeled and must restore the original endianness mode if it was modified in inline assembly code.

8.5.29 `.obyte` *Expressions* — Octuple Byte Data

The octuple byte data directive allows generating the binary representation of the value of one or more operands. Each value is represented using eight octets, where the current endianness

mode defines the ordering of the most and least significant one. If an operand for this directive is a string, each character of the string is evaluated individually, whereas real numbers are encoded using the IEEE double-precision format [27].

8.5.30 `.origin` *Expression* — Section Origin

The section origin directive specifies the origin of the section. The origin defines the actual address of a fixed section and may only be specified once and only if the section has no alignment, see Section 8.5.3. It is expressed in multiples of octets and may not be less than zero. This directive is not available in inline assembly code and may not be labeled.

8.5.31 `.pad` *Expression* — Data Padding

The data padding directive allows padding binary data space in order to accommodate layout constraints. It advances the current position in the section to the octet specified by the operand. This directive may not be labeled.

8.5.32 `.phantom` — Phantom Section

The phantom section directive marks the current section as a phantom section. See Section 8.2.3 for information about phantom sections. This directive may only be specified once, is not available in inline assembly code and may not be labeled.

8.5.33 `.qbyte` *Expressions* — Quadruple Byte Data

The quadruple byte data directive allows generating the binary representation of the value of one or more operands. Each value is represented using four octets, where the current endianness mode defines the ordering of the most and least significant one. If an operand for this directive is a string, each character of the string is evaluated individually, whereas real numbers are encoded using the IEEE single-precision format [27].

8.5.34 `#repeat` *Integer* — Begin Repetition

The begin repetition preprocessing directive allows translating any sequence of assembly code several times. Its operand specifies the number of repetitions of the assembly code encompassed by this directive and the subsequent end repetition directive. While repeating, two consecutive number sign characters within any character or string literal, identifier, address, or label are replaced by the decimal integer number of complete repetitions done so far. This directive may not be labeled and requires a subsequent end repetition directive.

8.5.35 `.require` *Identifier* — Name Requirement

The name requirement directive allows specifying additional dependencies on data or code sections that are otherwise not explicitly referenced in the current section. This directive may not be labeled.

8.5.36 `.required` — Required Section

The required section directive marks the current section as a required section. See Section 8.2.3 for information about required sections. This directive may only be specified once, is not available in inline assembly code and may not be labeled.

8.5.37 `.reserve` *Expression* — Data Reservation

The data reservation directive allows reserving binary data without having to initialize it. The size of the reserved data space is expressed in multiples of octets and may not be less than zero.

8.5.38 `.shared` — Shared Section

The shared section directive marks the current section as a shared section. See Section 8.2.3 for information about shared sections. This directive may only be specified once, is not available in inline assembly code and may not be labeled.

8.5.39 `.tbyte` *Expressions* — Triple Byte Data

The triple byte data directive allows generating the binary representation of the value of one or more operands. Each value is represented using three octets, where the current endianness mode defines the ordering of the most and least significant one. If an operand for this directive is a string, each character of the string is evaluated individually.

8.5.40 `.trace` $Expressions_{opt}$ — Expression Evaluation

The expression evaluation directive allows rewriting and evaluating arbitrary expressions during the translation of assembly code for debugging purposes. If no operands are provided, all currently defined labels and constant definitions are evaluated instead. This directive may not be labeled.

8.5.41 `.trailer` *Identifier* — Trailing Metadata Section

The trailing metadata section directive creates a new trailing metadata section with the specified name. See Section 8.2.2 for information about trailing metadata sections. All labels as well as

instructions and directives following this directive belong to the new section. This directive is not available in inline assembly code and may not be labeled.

8.5.42 `#undef` *Identifier* — Remove Macro Definition

The remove macro definition preprocessing directive causes the specified identifier no longer to be defined as a macro and thus allows redefining macros with the same name. This directive may not be labeled.

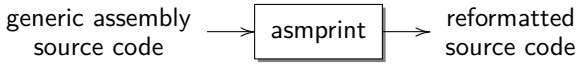
8.6 Assembler Tools

The Eigen Compiler Suite provides assemblers for several different hardware architectures. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools. For basic examples of using some of these tools in practice, see Chapter 3.

The assemblers process assembly source code in several consecutive stages. In each stage, the internal representation of the module is changed and transformed. Figure 8.1 shows all stages and the different representations.

8.6.1 asmprint

`asmprint` is a pretty printer for generic assembly code. It reformats generic assembly code and writes it to the standard output stream.



8.6.2 amd16asm

`amd16asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 16-bit operating mode defined by the AMD64 architecture.

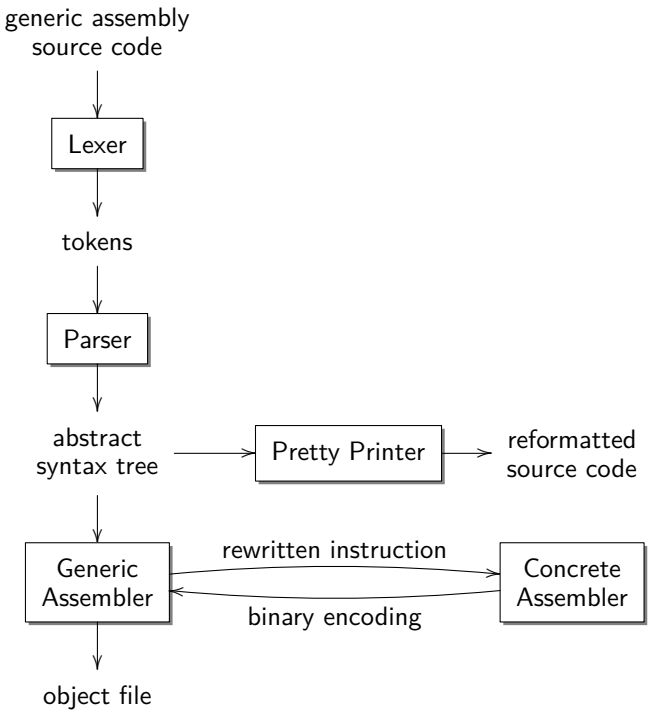


Figure 8.1: Data flow within assembler tools

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

8.6.3 amd32asm

`amd32asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 32-bit operating mode defined by the AMD64 architecture.

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

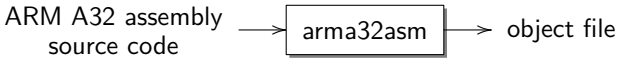
8.6.4 amd64asm

`amd64asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the AMD64 architecture.

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

8.6.5 arma32asm

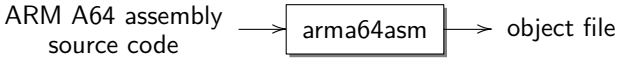
`arma32asm` is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing A32 instructions and stores it in corresponding object files.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

8.6.6 arma64asm

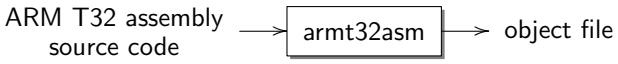
`arma64asm` is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing A64 instructions and stores it in corresponding object files.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

8.6.7 armt32asm

`armt32asm` is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing T32 instructions and stores it in corresponding object files.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

8.6.8 avrasm

`avrasm` is an assembler for the AVR hardware architecture. It translates assembly code into machine code for AVR processors and stores it in corresponding object files. The identifiers `RXL`, `RXH`, `RYL`, `RYH`, `RZL`, and `RZH` are predefined and name the corresponding registers. The identifiers `SPL` and `SPH` are also predefined and evaluate to the address of the corresponding registers.

For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22.

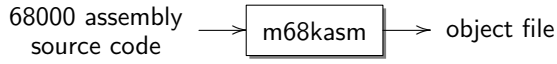
8.6.9 avr32asm

`avr32asm` is an assembler for the AVR32 hardware architecture. It translates assembly code into machine code for AVR32 processors and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12. For more information about object files and their purpose, see Chapter 22.

8.6.10 m68kasm

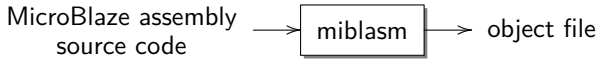
`m68kasm` is an assembler for the M68000 hardware architecture. It translates assembly code into machine code for M68000 processors and stores it in corresponding object files.



For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13. For more information about object files and their purpose, see Chapter 22.

8.6.11 miblasm

`miblasm` is an assembler for the MicroBlaze hardware architecture. It translates assembly code into machine code for MicroBlaze processors and stores it in corresponding object files.



For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14. For more information about object files and their purpose, see Chapter 22.

8.6.12 mips32asm

`mips32asm` is an assembler for the MIPS32 hardware architecture. It translates assembly code into machine code for MIPS32 processors and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22.

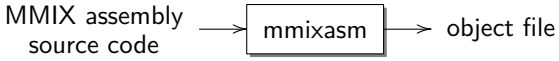
8.6.13 mips64asm

`mips64asm` is an assembler for the MIPS64 hardware architecture. It translates assembly code into machine code for MIPS64 processors and stores it in corresponding object files.

For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22.

8.6.14 mmixasm

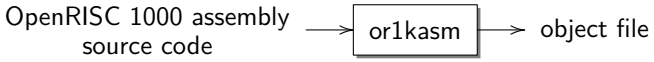
`mmixasm` is an assembler for the MMIX hardware architecture. It translates assembly code into machine code for MMIX processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.



For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16. For more information about object files and their purpose, see Chapter 22.

8.6.15 or1kasm

`or1kasm` is an assembler for the OpenRISC 1000 hardware architecture. It translates assembly code into machine code for OpenRISC 1000 processors and stores it in corresponding object files.

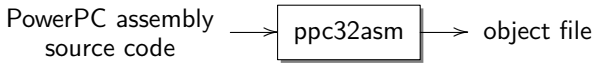


For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17. For more information about object files and their purpose, see Chapter 22.

8.6.16 ppc32asm

`ppc32asm` is an assembler for the PowerPC hardware architecture. It translates assembly code into machine code for PowerPC processors and stores it in corresponding object files. By

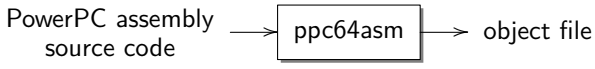
default, the assembler generates machine code for the 32-bit operating mode defined by the PowerPC architecture.



For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22.

8.6.17 ppc64asm

`ppc64asm` is an assembler for the PowerPC hardware architecture. It translates assembly code into machine code for PowerPC processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the PowerPC architecture.



For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22.

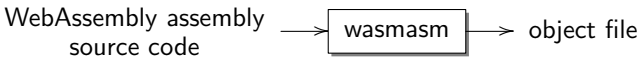
8.6.18 riscasm

`riscasm` is an assembler for the RISC hardware architecture. It translates assembly code into machine code for RISC processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.

For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19. For more information about object files and their purpose, see Chapter 22.

8.6.19 wasmasm

`wasmasm` is an assembler for the WebAssembly architecture. It translates assembly code into machine code for WebAssembly targets and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.



For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20. For more information about object files and their purpose, see Chapter 22.

8.6.20 xtensaasm

`xtensaasm` is an assembler for the Xtensa hardware architecture. It translates assembly code into machine code for Xtensa processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.

For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21. For more information about object files and their purpose, see Chapter 22.

8.7 Disassembler Tools

Each assembler provided by the Eigen Compiler Suite is accompanied by a corresponding disassembler tool which reverses its operation. Disassemblers open object files and print a human-readable disassembly listing of the machine code contained therein as shown in Figure 8.2. They accept command-line arguments which are taken as names of object files. If no arguments are provided, object files are read from the standard input stream. See Chapter 2 for more information about the common user interface of all of these tools. For basic examples of using some of these tools in practice, see Chapter 3.

8.7.1 amd16dism

`amd16dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard

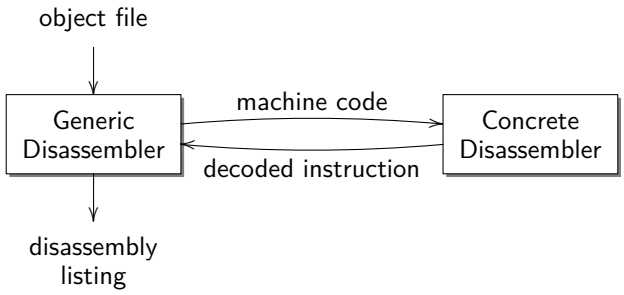


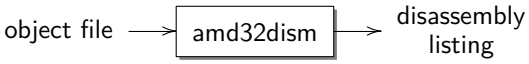
Figure 8.2: Data flow within disassembler tools

output stream. It assumes that the machine code was generated for the 16-bit operating mode defined by the AMD64 architecture.

For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

8.7.2 amd32dism

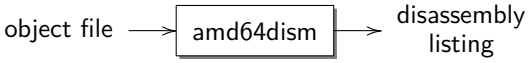
`amd32dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 32-bit operating mode defined by the AMD64 architecture.



For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

8.7.3 amd64dism

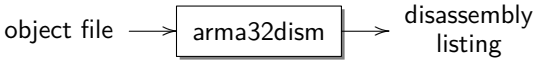
`amd64dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the AMD64 architecture.



For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22.

8.7.4 arma32dism

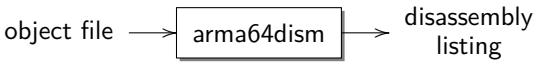
`arma32dism` is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing A32 instructions into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

8.7.5 arma64dism

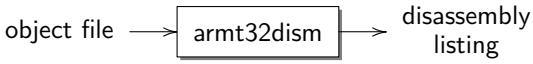
`arma64dism` is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing A64 instructions into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

8.7.6 armt32dism

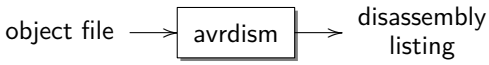
`armt32dism` is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing T32 instructions into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22.

8.7.7 avrdism

`avrdism` is a disassembler for the AVR hardware architecture. It translates machine code from object files targeting AVR processors into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22.

8.7.8 avr32dism

`avr32dism` is a disassembler for the AVR32 hardware architecture. It translates machine code from object files targeting AVR32 processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12. For more information about object files and their purpose, see Chapter 22.

8.7.9 m68kdism

`m68kdism` is a disassembler for the M68000 hardware architecture. It translates machine code from object files targeting M68000 processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13. For more information about object files and their purpose, see Chapter 22.

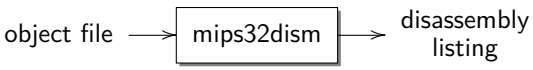
8.7.10 mibldism

`mibldism` is a disassembler for the MicroBlaze hardware architecture. It translates machine code from object files targeting MicroBlaze processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14. For more information about object files and their purpose, see Chapter 22.

8.7.11 mips32dism

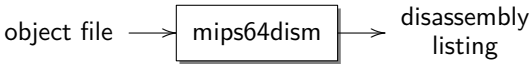
`mips32dism` is a disassembler for the MIPS32 hardware architecture. It translates machine code from object files targeting MIPS32 processors into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22.

8.7.12 mips64dism

`mips64dism` is a disassembler for the MIPS64 hardware architecture. It translates machine code from object files targeting MIPS64 processors into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22.

8.7.13 mmixdism

`mmixdism` is a disassembler for the MMIX hardware architecture. It translates machine code from object files targeting MMIX processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16. For more information about object files and their purpose, see Chapter 22.

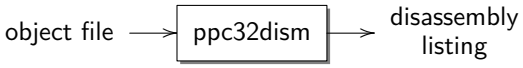
8.7.14 or1kdism

`or1kdism` is a disassembler for the OpenRISC 1000 hardware architecture. It translates machine code from object files targeting OpenRISC 1000 processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17. For more information about object files and their purpose, see Chapter 22.

8.7.15 ppc32dism

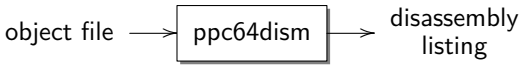
`ppc32dism` is a disassembler for the PowerPC hardware architecture. It translates machine code from object files targeting PowerPC processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 32-bit operating mode defined by the PowerPC architecture.



For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22.

8.7.16 ppc64dism

`ppc64dism` is a disassembler for the PowerPC hardware architecture. It translates machine code from object files targeting PowerPC processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the PowerPC architecture.



For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22.

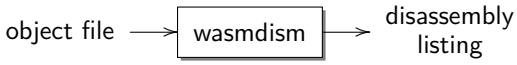
8.7.17 riscdism

`riscdism` is a disassembler for the RISC hardware architecture. It translates machine code from object files targeting RISC processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19. For more information about object files and their purpose, see Chapter 22.

8.7.18 wasmdism

`wasmdism` is a disassembler for the WebAssembly architecture. It translates machine code from object files targeting WebAssembly targets into assembly code and writes it to the standard output stream.



For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20. For more information about object files and their purpose, see Chapter 22.

8.7.19 xtensadism

`xtensadism` is a disassembler for the Xtensa hardware architecture. It translates machine code from object files targeting Xtensa processors into assembly code and writes it to the standard output stream.

For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21. For more information about object files and their purpose, see Chapter 22.

Part III

Supported Hardware Architectures

9 AMD64

This chapter describes how the Eigen Compiler Suite supports the AMD64 hardware architecture. This includes information about the assemblers, disassemblers, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

9.1 Introduction

The Eigen Compiler Suite features various compilers, assemblers, and disassemblers that target the AMD64 hardware architecture by AMD. Figure 9.1 shows the data flow in-between these tools.

All compilers targeting the AMD64 architecture translate their programs using an intermediate code representation. The AMD64 generator is able to translate the intermediate code representation of a program into machine code for AMD64 processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assemblers yielding exactly the same object file. The disassemblers are able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

9.2 Instruction Set

Tools targeting the AMD64 architecture support the instruction set listed in Table 9.1 and use the same assembly syntax as predefined by AMD [11, 12, 13]. The only exception are instructions for far procedure calls and far jumps which take the selector and offset of the immediate far pointer as two separate operands. The tools support all different modes of operation defined by the AMD64 architecture as described in Section 9.3. The actual set of supported instructions depends on the operating mode that is used. If there are two or more possible encodings of an instruction, the shortest one is used by default. In order to specify the encoding of an instruction explicitly, the size of its operands can be overridden by prefixing them with one of the following identifiers: `byte` for 8-bit, `word` for 16-bit, `dword` for 32-bit, `qword` for 64-bit, `oword` for 128-bit, `yword` for 256-bit, and `zword` for 512-bit. For more information about the generic assembly language and how to use it, see Chapter 8.

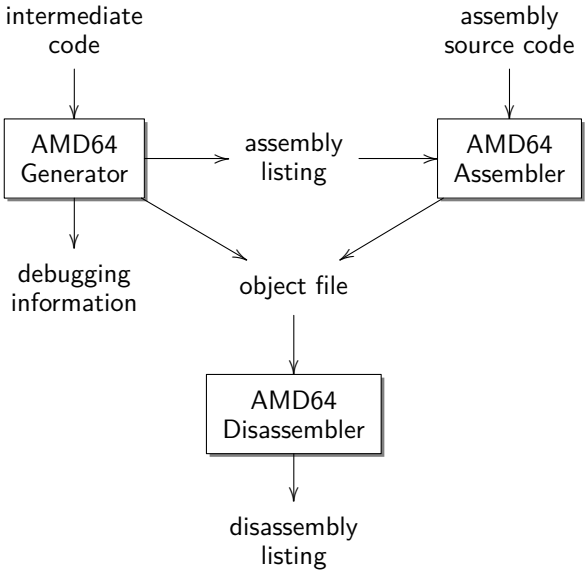


Figure 9.1: Data flow within the tools targeting the AMD64 architecture

Table 9.1: Supported AMD64 instruction set (1285 instructions)

aaa	aeskeygenassist	blsi	clc	cmovl
aad	and	blsic	cld	cmovle
aam	andn	blsmsk	clflush	cmovna
aas	andnpd	blsr	clflushopt	cmovnae
adc	andnps	bound	clgi	cmovnb
adcx	andpd	bsf	cli	cmovnbe
add	andps	bsr	clrssbsy	cmovnc
addpd	arpl	bswap	clts	cmovne
addps	bextr	bt	clwb	cmovng
addsd	blcfill	btc	clzero	cmovnge
addss	blci	btr	cmc	cmovnl
addsubpd	blcic	bts	cmova	cmovnle
addsubps	blcmsk	bzhi	cmovae	cmovno
adox	blcs	call	cmovb	cmovnp
aesdec	blendpd	callfar	cmovbe	cmovns
aesdeclast	blendps	cbw	cmovc	cmovnz
aesenc	blendvpd	cdq	cmove	cmovo
aesenclast	blendvps	cdqe	cmovg	cmovp
aesimc	blsfill	clac	cmovge	cmovpe

```
cmovpo
cmovs
cmovz
cmp
cmpeqpd
cmpeqps
cmpeqsd
cmpeqss
cmplepd
cmpleps
cmplesd
cmpless
cmpltpd
cmpltps
cmpltsd
cmpltss
cmpneqpd
cmpneqps
cmpneqsd
cmpneqss
cmpnlepd
cmpnleps
cmpnlesd
cmpnless
cmpnltpd
cmpnltps
cmpnltsd
cmpnltss
cmpordpd
cmpordps
cmpordsd
cmpordss
cmppd
cmpps
cmpsb
cmpsd
cmpsq
cmpss
cmpsw
cmpunordpd
cmpunordps
cmpunordsd
cmpunordss
cmpxchg
cmpxchg16b
cmpxchg8b
comisd
comiss
cpuid
cqo
crc32
cvtdq2pd
cvtdq2ps
cvtpd2dq
cvtpd2pi
cvtpd2ps
cvtpi2pd
cvtpi2ps
cvtps2dq
cvtps2pd
cvtps2pi
cvtsd2si
cvtsd2ss
cvtsi2sd
cvtsi2ss
cvtss2sd
cvtss2si
cvttpd2dq
cvttpd2pi
cvttps2dq
cvttps2pi
cvttsd2si
cvttss2si
cwd
cwde
daa
das
dec
div
divpd
divps
divsd
divss
dppd
dpps
emms
enter
extractps
extrq
f2xm1
fabs
fadd
faddp
fbld
fbstp
fchs
fcmovb
fcmovbe
fcmove
fcmovnb
fcmovnbe
fcmovne
fcmovnu
fcmovu
fcom
fcomi
fcomip
fcomp
fcompp
fcos
fdecstp
fdiv
fdivp
fdivr
fdivrp
femms
ffree
fiadd
ficom
ficomp
fidiv
fidivr
fild
fimul
fincstp
fist
fistp
fisttp
fisub
fisubr
fld
fld1
fldcw
fldenv
fldl2e
fldl2t
fldlg2
fldln2
fldpi
fldz
fmul
fmulp
fnclex
fninit
fnop
fnsave
fnstcw
fnstenv
fnstsw
fpatan
fprem
fprem1
fptan
frndint
frstor
fscale
fsin
fsincos
fsqrt
fst
fstp
fsub
fsubp
fsubr
fsubrp
ftst
fucom
fucomi
fucomip
fucomp
fucompp
fwait
fxam
fxch
fxrstor
fxsave
fxtract
fyl2x
fyl2xp1
haddpd
haddps
hlt
hsubpd
hsubps
idiv
imul
in
inc
incssp
insb
insd
insertps
insertq
insw
int
int3
into
invd
invlpg
invlpga
invlpgb
invpcid
iret
iretd
iretq
ja
jae
jb
jbe
jc
jcxz
je
jecxz
jg
jge
jl
jle
jmp
jmpfar
jna
jnae
jnb
jnbe
jnc
jne
jng
jnge
jnl
jnle
jno
jnp
jns
jnz
jo
jp
jpe
jpo
jrcxz
js
jz
lahf
lar
lddqu
ldmxcsr
lds
lea
leave
les
lfence
lfs
```

```
lgdt
lgs
lidt
lldt
llwpcb
lmsw
lodsb
lodsd
lodsq
lodsw
loop
loope
loopne
loopnz
loopz
lsl
lss
ltr
lwpins
lwpval
lzcnt
maskmovdqu
maskmovq
maxpd
maxps
maxsd
maxss
mcommit
mfence
minpd
minps
minsd
minss
monitor
monitorx
mov
movapd
movaps
movbe
movd
movddup
movdir64b
movdiri
movdq2q
movdqa
movdqu
movhlps
movhpd
movhps
movlhps
movlpd
movlps
movmskpd
movmskps
movntdq
movntdqa
movnti
movntpd
movntps
movntq
movntsd
movntss
movq
movq2dq
movsb
movsd
movshdup
movsldup
movsq
movss
movsw
movsx
movsxd
movupd
movups
movzx
mpsadbw
mul
mulpd
mulps
mulsd
mulss
mulx
mwait
mwaitx
neg
nop
not
or
orpd
orps
out
outsb
outsd
outsw
pabsb
pabsd
pabsw
packssdw
packsswb
packusdw
packuswb
paddb
paddd
paddq
paddsb
paddsw
paddusb
paddusw
paddw
palignr
pand
pandn
pause
pavgb
pavgusb
pavgw
pblendvb
pblendw
pclmulqdq
pcmpeqb
pcmpeqd
pcmpeqq
pcmpeqw
pcmpestri
pcmpestrm
pcmpgtb
pcmpgtd
pcmpgtq
pcmpgtw
pcmpistri
pcmpistrm
pdep
pext
pextrb
pextrd
pextrq
pextrw
pf2id
pf2iw
pfacc
pfadd
pfcmpeq
pfcmpge
pfcmpgt
pfmax
pfmin
pfmul
pfnacc
pfpnacc
pfrcp
pfrcpit1
pfrcpit2
pfrsqit1
pfrsqrt
pfsub
pfsubr
phaddd
phaddsw
phaddw
phminposuw
phsubd
phsubsw
phsubw
pi2fd
pi2fw
pinsrb
pinsrd
pinsrq
pinsrw
pmaddubsw
pmaddwd
pmaxsb
pmaxsd
pmaxsw
pmaxub
pmaxud
pmaxuw
pminsb
pminsd
pminsw
pminub
pminud
pminuw
pmovmskb
pmovsxbd
pmovsxbq
pmovsxbw
pmovsxdq
pmovsxwd
pmovsxwq
pmovzxbd
pmovzxbq
pmovzxbw
pmovzxdq
pmovzxwd
pmovzxwq
pmuldq
pmulhrsw
pmulhrw
pmulhuw
pmulhw
pmulld
pmullw
pmuludq
pop
popa
popad
popcnt
popf
popfd
popfq
por
prefetch
prefetchnta
prefetcht0
prefetcht1
prefetcht2
prefetchw
psadbw
pshufb
pshufd
pshufhw
pshuflw
pshufw
psignb
psignd
psignw
pslld
pslldq
psllq
psllw
psmash
psrad
psraw
psrld
psrldq
psrlq
psrlw
psubb
psubd
psubq
psubsb
psubsw
psubusb
psubusw
psubw
pswapd
ptest
punpckhbw
```

```
punpckhdq
punpckhqdq
punpckhwd
punpcklbw
punpckldq
punpcklqdq
punpcklwd
push
pusha
pushad
pushf
pushfd
pushfq
pvalidate
pxor
rcl
rcpps
rcpss
rcr
rdfsbase
rdgsbase
rdmsr
rdpid
rdpkru
rdpmc
rdpru
rdrand
rdseed
rdsspd
rdsspq
rdtsc
rdtscp
ret
retf
rmpadjust
rmpquery
rmpread
rmpupdate
rol
ror
rorx
roundpd
roundps
roundsd
roundss
rsm
rsqrtps
rsqrtss
rstorssp
sahf
sal
sar
sarx
saveprevssp
sbb
scasb
scasd
scasq
scasw
seta
setae
setb
setbe
setc
sete
setg
setge
setl
setle
setna
setnae
setnb
setnbe
setnc
setne
setng
setnge
setnl
setnle
setno
setnp
setns
setnz
seto
setp
setpe
setpo
sets
setssbsy
setz
sfence
sgdt
sha1msg1
sha1msg2
sha1nexte
sha1rnds4
sha256msg1
sha256msg2
sha256rnds2
shl
shld
shlx
shr
shrd
shrx
shufpd
shufps
sidt
skinit
sldt
slwpcb
smsw
sqrtpd
sqrtps
sqrtsd
sqrtss
stac
stc
std
stgi
sti
stmxcsr
stosb
stosd
stosq
stosw
str
sub
subpd
subps
subsd
subss
swapgs
syscall
sysenter
sysexit
sysret
t1mskc
test
tlbsync
tzcnt
tzmsk
ucomisd
ucomiss
ud0
ud1
ud2
unpckhpd
unpckhps
unpcklpd
unpcklps
vaddpd
vaddps
vaddsd
vaddss
vaddsubpd
vaddsubps
vaesdec
vaesdeclast
vaesenc
vaesenclast
vaesimc
vaeskeygenassist
vandnpd
vandnps
vandpd
vandps
vblendpd
vblendps
vblendvpd
vblendvps
vbroadcastf128
vbroadcasti128
vbroadcastsd
vbroadcastss
vcmpeqpd
vcmpeqps
vcmpeqsd
vcmpeqss
vcmplepd
vcmpleps
vcmplesd
vcmpless
vcmpltpd
vcmpltps
vcmpltsd
vcmpltss
vcmpneqpd
vcmpneqps
vcmpneqsd
vcmpneqss
vcmpnlepd
vcmpnleps
vcmpnlesd
vcmpnless
vcmpnltpd
vcmpnltps
vcmpnltsd
vcmpnltss
vcmpordpd
vcmpordps
vcmpordsd
vcmpordss
vcmppd
vcmpps
vcmpsd
vcmpss
vcmpunordpd
vcmpunordps
vcmpunordsd
vcmpunordss
vcomisd
vcomiss
vcvtdq2pd
vcvtdq2ps
vcvtpd2dq
vcvtpd2ps
vcvtph2ps
vcvtps2dq
vcvtps2pd
vcvtps2ph
vcvtsd2si
vcvtsd2ss
vcvtsi2sd
vcvtsi2ss
vcvtss2sd
vcvtss2si
vcvttpd2dq
vcvttps2dq
vcvttsd2si
vcvttss2si
vdivpd
vdivps
vdivsd
vdivss
vdppd
vdpps
verr
verw
vextractf128
vextracti128
vextractps
vfmadd132pd
vfmadd132ps
vfmadd132sd
vfmadd132ss
vfmadd213pd
vfmadd213ps
vfmadd213sd
vfmadd213ss
```

```
vfmadd231pd
vfmadd231ps
vfmadd231sd
vfmadd231ss
vfmaddpd
vfmaddps
vfmaddsd
vfmaddss
vfmaddsub132pd
vfmaddsub132ps
vfmaddsub213pd
vfmaddsub213ps
vfmaddsub231pd
vfmaddsub231ps
vfmaddsubpd
vfmaddsubps
vfmsub132pd
vfmsub132ps
vfmsub132sd
vfmsub132ss
vfmsub213pd
vfmsub213ps
vfmsub213sd
vfmsub213ss
vfmsub231pd
vfmsub231ps
vfmsub231sd
vfmsub231ss
vfmsubadd132pd
vfmsubadd132ps
vfmsubadd213pd
vfmsubadd213ps
vfmsubadd231pd
vfmsubadd231ps
vfmsubaddpd
vfmsubaddps
vfmsubpd
vfmsubps
vfmsubsd
vfmsubss
vfnmadd132pd
vfnmadd132ps
vfnmadd132sd
vfnmadd132ss
vfnmadd213pd
vfnmadd213ps
vfnmadd213sd
vfnmadd213ss
vfnmadd231pd
vfnmadd231ps
vfnmadd231sd
vfnmadd231ss
vfnmaddpd
vfnmaddps
vfnmaddsd
vfnmaddss
vfnmsub132pd
vfnmsub132ps
vfnmsub132sd
vfnmsub132ss
vfnmsub213pd
vfnmsub213ps
vfnmsub213sd
vfnmsub213ss
vfnmsub231pd
vfnmsub231ps
vfnmsub231sd
vfnmsub231ss
vfnmsubpd
vfnmsubps
vfnmsubsd
vfnmsubss
vfrczpd
vfrczps
vfrczsd
vfrczss
vhaddpd
vhaddps
vhsubpd
vhsubps
vinsertf128
vinserti128
vinsertps
vlddqu
vldmxcsr
vmaskmovdqu
vmaskmovpd
vmaskmovps
vmaxpd
vmaxps
vmaxsd
vmaxss
vmgexit
vminpd
vminps
vminsd
vminss
vmload
vmmcall
vmovapd
vmovaps
vmovd
vmovddup
vmovdqa
vmovdqu
vmovhlps
vmovhpd
vmovhps
vmovlhps
vmovlpd
vmovlps
vmovmskpd
vmovmskps
vmovntdq
vmovntdqa
vmovntpd
vmovntps
vmovq
vmovsd
vmovshdup
vmovsldup
vmovss
vmovupd
vmovups
vmpsadbw
vmrun
vmsave
vmulpd
vmulps
vmulsd
vmulss
vorpd
vorps
vpabsb
vpabsd
vpabsw
vpackssdw
vpacksswb
vpackusdw
vpackuswb
vpaddb
vpaddd
vpaddq
vpaddsb
vpaddsw
vpaddusb
vpaddusw
vpaddw
vpalignr
vpand
vpandn
vpavgb
vpavgw
vpblendd
vpblendvb
vpblendw
vpbroadcastb
vpbroadcastd
vpbroadcastq
vpbroadcastw
vpclmulqdq
vpcmov
vpcmpeqb
vpcmpeqd
vpcmpeqq
vpcmpeqw
vpcmpestri
vpcmpestrm
vpcmpgtb
vpcmpgtd
vpcmpgtq
vpcmpgtw
vpcmpistri
vpcmpistrm
vpcomb
vpcomd
vpcomeqb
vpcomeqd
vpcomeqq
vpcomequb
vpcomequd
vpcomequq
vpcomequw
vpcomeqw
vpcomfalseb
vpcomfalsed
vpcomfalseq
vpcomfalseub
vpcomfalseud
vpcomfalseuq
vpcomfalseuw
vpcomfalsew
vpcomgeb
vpcomged
vpcomgeq
vpcomgeub
vpcomgeud
vpcomgeuq
vpcomgeuw
vpcomgew
vpcomgtb
vpcomgtd
vpcomgtq
vpcomgtub
vpcomgtud
vpcomgtuq
vpcomgtuw
vpcomgtw
vpcomleb
vpcomled
vpcomleq
vpcomleub
vpcomleud
vpcomleuq
vpcomleuw
vpcomlew
vpcomltb
vpcomltd
vpcomltq
vpcomltub
vpcomltud
vpcomltuq
vpcomltuw
vpcomltw
vpcomneqb
vpcomneqd
vpcomneqq
vpcomnequb
vpcomnequd
vpcomnequq
vpcomnequw
vpcomneqw
vpcomq
vpcomtrueb
vpcomtrued
vpcomtrueq
vpcomtrueub
vpcomtrueud
vpcomtrueuq
vpcomtrueuw
vpcomtruew
vpcomub
vpcomud
vpcomuq
vpcomuw
vpcomw
vperm2f128
vperm2i128
vpermd
vpermpd
```

```
vpermps          vpmacsswd        vpmulhw          vpsrlvd          vsubsd
vpermq           vpmacssww        vpmulld          vpsrlvq          vsubss
vpextrb          vpmacswd         vpmullw          vpsrlw           vtestpd
vpextrd          vpmacsww         vpmuludq         vpsubb           vtestps
vpextrq          vpmadcsswd       vpor             vpsubd           vucomisd
vpextrw          vpmadcswd        vpperm           vpsubq           vucomiss
vphaddbd         vpmaddubsw       vprotb           vpsubsb          vunpckhpd
vphaddbq         vpmaddwd         vprotd           vpsubsw          vunpckhps
vphaddbw         vpmaskmovd       vprotq           vpsubusb         vunpcklpd
vphaddd          vpmaskmovq       vprotw           vpsubusw         vunpcklps
vphadddq         vpmaxsb          vpsadbw          vpsubw           vxorpd
vphaddsw         vpmaxsd          vpshab           vptest           vxorps
vphaddubd        vpmaxsw          vpshad           vpunpckhbw       vzeroall
vphaddubq        vpmaxub          vpshaq           vpunpckhdq       vzeroupper
vphaddubw        vpmaxud          vpshaw           vpunpckhqdq      wbinvd
vphaddudq        vpmaxuw          vpshlb           vpunpckhwd       wbnoinvd
vphadduwd        vpminsb          vpshld           vpunpcklbw       wrfsbase
vphadduwq        vpminsd          vpshlq           vpunpckldq       wrgsbase
vphaddw          vpminsw          vpshlw           vpunpcklqdq      wrmsr
vphaddwd         vpminub          vpshufb          vpunpcklwd       wrpkru
vphaddwq         vpminud          vpshufd          vpxor            wrssd
vphminposuw      vpminuw          vpshufhw         vrcpps           wrssq
vphsubbw         vpmovmskb        vpshuflw         vrcpss           wrussd
vphsubd          vpmovsxbd        vpsignb          vroundpd         wrussq
vphsubdq         vpmovsxbq        vpsignd          vroundps         xadd
vphsubsw         vpmovsxbw        vpsignw          vroundsd         xchg
vphsubw          vpmovsxdq        vpslld           vroundss         xgetbv
vphsubwd         vpmovsxwd        vpslldq          vrsqrtps         xlatb
vpinsrb          vpmovsxwq        vpsllq           vrsqrtss         xor
vpinsrd          vpmovzxbd        vpsllvd          vshufpd          xorpd
vpinsrq          vpmovzxbq        vpsllvq          vshufps          xorps
vpinsrw          vpmovzxbw        vpsllw           vsqrtpd          xrstor
vpmacsdd         vpmovzxdq        vpsrad           vsqrtps          xrstors
vpmacsdqh        vpmovzxwd        vpsravd          vsqrtsd          xsave
vpmacsdql        vpmovzxwq        vpsraw           vsqrtss          xsavec
vpmacssdd        vpmuldq          vpsrld           vstmxcsr         xsaveopt
vpmacssdqh       vpmulhrsw        vpsrldq          vsubpd           xsaves
vpmacssdql       vpmulhuw         vpsrlq           vsubps           xsetbv
```


9.3 Operating Modes

The AMD64 architecture defines several different operating modes which imply different operand and address sizes [28]. Table 9.2 summarizes the main differences between these operating modes. All assemblers and compilers that target the AMD64 architecture support the following operating modes:

Operating Modes		Default Address Size	Default Operand Size
Long Mode	64-bit Mode	64	32
	Compatibility Mode	32	32
	(Legacy Modes)	16	16
Legacy Mode	Protected Mode	32	32
	Virtual-8086 Mode	16	16
	Real Mode	16	16

Table 9.2: AMD64 operating modes with default address and operand sizes

- Real Mode, Virtual 8086 Mode 16-bit

  In 16-bit mode, addresses and operands have a default size of 16 bits.

- Protected Mode 32-bit

  In 32-bit mode, addresses and operands have a default size of 32 bits.

- 64-Bit Mode 64-bit

  In 64-bit mode, addresses have a default size of 64 bits, whereas the size of operands usually remains 32 bits. There are more and wider registers available in this mode. Additionally, instructions implementing the 128-bit media programming model are available by design.

The actual operating mode the compilers and assemblers generate machine code for by default, is indicated by the number in the suffix of their names. The assemblers allow users to temporarily switch between the supported operating modes by passing 16, 32 or 64 as operand to the bit mode directive.

9.4 Calling Convention

The machine code generator and runtime support for the AMD64 architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

9.4.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function

arguments are pushed on the stack in reverse order and cleaned by the caller. If the AMD64 machine code generator runs out of physical registers, previously mapped registers are spilled on the stack.

9.4.2 Floating-Point Support

The AMD64 architecture natively supports floating-point operations by providing two different floating-point programming models. The first is the x87 floating-point programming model which is used as a fallback for the newer 128-bit media programming model. Compilers for the 64-bit mode use the 128-bit media programming model by default. Other tools use the standard x87 floating-point programming model.

9.4.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to AMD64 registers `al`, `ax`, `eax`, depending on the size of the actual result value. 64-bit wide results are stored in registers `eax` and `edx`, or only in register `rax` in 64-bit mode. Floating-point results are stored in registers `st0` or `xmm0` depending on the floating-point programming model.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to AMD64 registers `sp`, `esp`, or `rsp` depending on the target operating mode.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to AMD64 registers `bp`, `ebp`, or `rbp` depending on the target operating mode.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is not supported.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls. Registers holding integer and address values are mapped to the general-purpose registers. Registers holding floating-point values are either mapped to the physical registers of the 128-bit media programming model, or stored on the floating-point stack provided by the x87 floating-point programming model.

9.5 Runtime Support

The Eigen Compiler Suite provides runtime support for the AMD64 architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the AMD64 hardware architecture as well as the required combination of linker and object files.

9.5.1 General

Basic architectural runtime support is provided by the object files `amd16run`, `amd32run`, and `amd64run`. Users should always include one of these object files during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppamd16run`, `cppamd32run`, and `cppamd64run` library files respectively. Programs written in Oberon need additional runtime support stored in the `obamd16run`, `obamd32run`, and `obamd64run` library files respectively. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

9.5.2 BIOS

The Eigen Compiler Suite provides basic runtime support for a native PC environment partially established by the BIOS. The actual runtime support is written in assembly language and stored in the object files `bios16run`, `bios32run`, and `bios64run`. Users creating their own systems may use the `linkbin` linker together with one of these object files in order create an executable bootloader. Calling the `ecsd` utility tool using the `bios16`, `bios32`, or `bios64` target environments achieves the same result.

9.5.3 Darwin

Programs targeting the Darwin operating system are created using the `linkbin` linker tool. It creates either 32-bit or 64-bit Mach-O files [29] if provided with the runtime support stored in the `amd32darwinrun` and `amd64darwinrun` object files respectively. Calling the `ecsd` utility tool using the `amd32darwin` or `amd64darwin` target environments achieves the same result.

9.5.4 DOS

Programs targeting the DOS operating system are created using the `linkbin` linker tool. It creates 16-bit COM files if provided with the runtime support stored in the `dosrun` object file. Calling the `ecsd` utility tool using the `dos` target environment achieves the same result.

9.5.5 EFI

Programs targeting the Extensible Firmware Interface are created using the `linkbin` linker tool. It creates either 32-bit or 64-bit EFI images [30] if provided with the runtime support stored in the `efi32run` and `efi64run` object files respectively. Calling the `ecsd` utility tool using the `efi32` or `efi64` target environments achieves the same result. Functions of the EFI API are available using additional runtime support stored in the `efi32api` and `efi64api` object files.

9.5.6 Linux

Programs targeting Linux-based operating systems are created using the `linkbin` linker tool. It creates either 32-bit or 64-bit Executable and Linking Format (ELF) files [2] if provided with the runtime support stored in the `amd32linuxrun` and `amd64linuxrun` object files respectively. Calling the `ecsd` utility tool using the `amd32linux` or `amd64linux` target environments achieves the same result. External libraries are available using additional runtime support stored in the following object files: `amd32libdl`, `amd32libpthread`, `amd32libsdl`, `amd64-libdl`, `amd64libpthread`, and `amd64libsdl`.

9.5.7 Windows

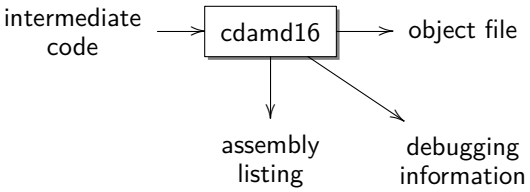
The Eigen Compiler Suite provides runtime support for 32-bit and 64-bit Windows applications. Programs targeting both flavors of this operating system are created using the `linkbin` linker tool. It creates either 32-bit or 64-bit Portable Executable (EXE) files [30] if provided with the runtime support stored in the `win32run` and `win64run` object files respectively. Calling the `ecsd` utility tool using the `win32` or `win64` target environments achieves the same result. The value of the desired image subsystem can be provided by the origin of a phantom section called `_subsystem` and defaults to the character-mode user interface. Functions of the Windows API are available using additional runtime support stored in the `win32api` and `win64api` object files. External libraries are available using additional runtime support stored in the following object files: `win32sdl` and `win64sdl`.

9.6 AMD64 Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the AMD64 hardware architecture. The number in the name of compilers, assemblers, and disassemblers specifies which operating mode defined by the AMD64 architecture is used by default, see Section 9.3. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools. For basic examples of using some of these tools in practice, see Chapter 3.

9.6.1 cdamd16

`cdamd16` is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.

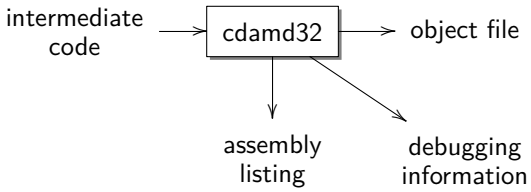


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

9.6.2 cdamd32

`cdamd32` is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is

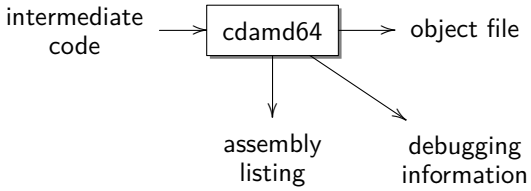
provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

9.6.3 cdamd64

`cdamd64` is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.

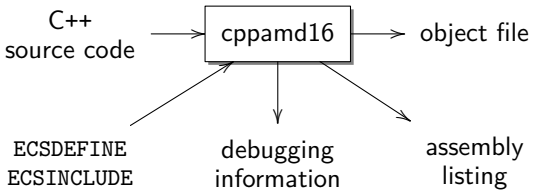


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

9.6.4 cppamd16

`cppamd16` is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++

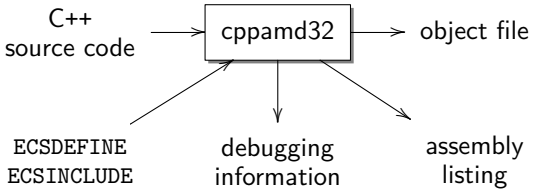
and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd16__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd16run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

9.6.5 cppamd32

`cppamd32` is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd32run` library file.

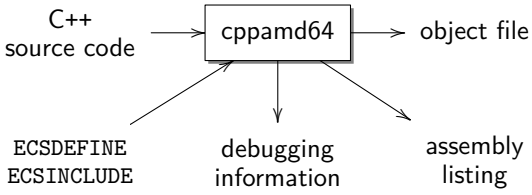


For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and

their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

9.6.6 cppamd64

`cppamd64` is a compiler for the C++ programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__amd64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppamd64run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

9.6.7 falamd16

`falamd16` is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

9.6.8 falamd32

`falamd32` is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

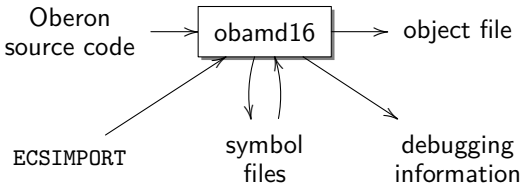
9.6.9 falamd64

`falamd64` is a compiler for the FALSE programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

9.6.10 obamd16

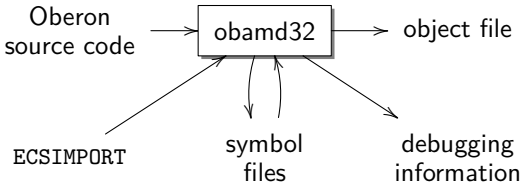
`obamd16` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd16run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

9.6.11 obamd32

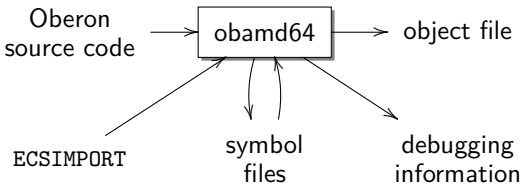
`obamd32` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd32run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

9.6.12 obamd64

`obamd64` is a compiler for the Oberon programming language targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obamd64run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

9.6.13 amd16asm

`amd16asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 16-bit operating mode defined by the AMD64 architecture.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

9.6.14 amd32asm

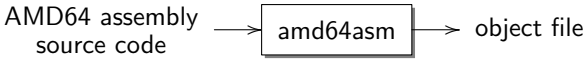
`amd32asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default,

the assembler generates machine code for the 32-bit operating mode defined by the AMD64 architecture.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

9.6.15 amd64asm

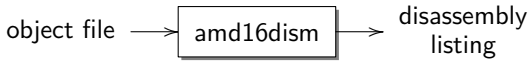
`amd64asm` is an assembler for the AMD64 hardware architecture. It translates assembly code into machine code for AMD64 processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the AMD64 architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

9.6.16 amd16dism

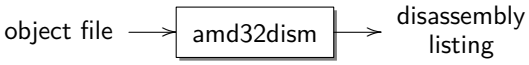
`amd16dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 16-bit operating mode defined by the AMD64 architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

9.6.17 amd32dism

`amd32dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 32-bit operating mode defined by the AMD64 architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

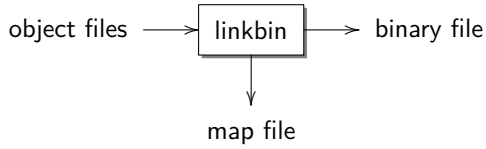
9.6.18 amd64dism

`amd64dism` is a disassembler for the AMD64 hardware architecture. It translates machine code from object files targeting AMD64 processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the AMD64 architecture.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

9.6.19 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

10 ARM

This chapter describes how the Eigen Compiler Suite supports the ARM hardware architecture. This includes information about the assemblers, disassemblers, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

10.1 Introduction

The Eigen Compiler Suite features various compilers, assemblers, and disassemblers that target the ARM hardware architecture by ARM Limited. Figure 10.1 shows the data flow in-between these tools.

All compilers targeting the ARM architecture translate their programs using an intermediate code representation. The A32, A64 and T32 generators are able to translate the intermediate code representation of a program into machine code for ARM processors executing the respective instruction set. All of them store the resulting binary code and data in so-called object files. Additionally, all generators are able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the corresponding assembler yielding exactly the same object file. The corresponding disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

10.2 Instruction Sets

Tools targeting the ARM architecture support the instruction sets listed in Tables 10.1 to 10.3 and use the same assembly syntax as predefined by the ARM architecture [14]. The only exception are immediate values which are not prefixed by a number sign. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 10.1: Supported ARM A32 instruction set (286 instructions)

adc	b	bx	cpy	fabsd
adcs	bic	cdp	dbg	fabss
add	bics	cdp2	dmb	faddd
adds	bkpt	clz	dsb	fadds
and	bl	cmn	eor	fcmpd
ands	blx	cmp	eors	fcmped

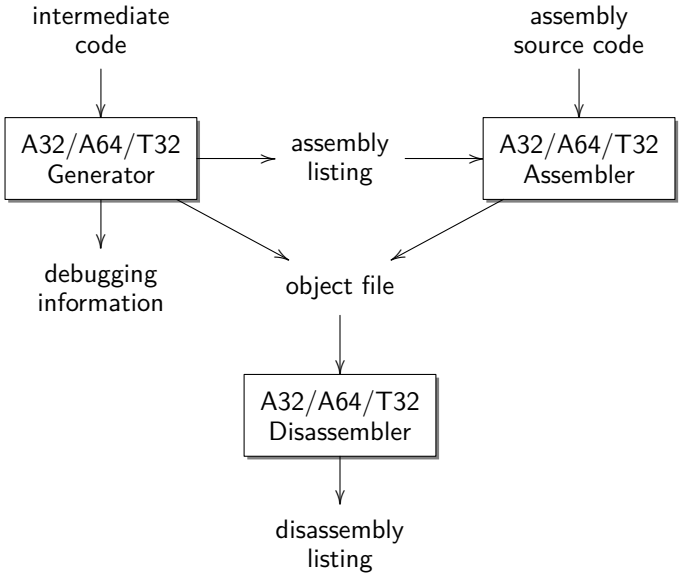


Figure 10.1: Data flow within the tools targeting the ARM architecture

fcmpes	fmdrr	fnmuls	fuitos	mcr
fcmpezd	fmovd	fsitod	hlt	mcr2
fcmpezs	fmovs	fsitos	isb	mcrr
fcmps	fmrdh	fsqrtd	ldc	mcrr2
fcmpzd	fmrdl	fsqrts	ldcl	mla
fcmpzs	fmrrd	fstd	ldc2	mlas
fcpyd	fmrrs	fstmdbd	ldc2l	mov
fcpys	fmrs	fstmdbs	ldmda	movs
fcvtds	fmscd	fstmiad	ldmdb	mrc
fcvtsd	fmscs	fstmias	ldmia	mrc2
fdivd	fmsr	fsts	ldmib	mrrc
fdivs	fmsrr	fsubd	ldr	mrrc2
fldd	fmstat	fsubs	ldrb	mrs
fldmdbd	fmuld	ftosid	ldrbt	msr
fldmdbs	fmuls	ftosis	ldrex	mul
fldmiad	fnegd	ftosizd	ldrexb	muls
fldmias	fnegs	ftosizs	ldrexd	mvn
flds	fnmacd	ftouid	ldrexh	mvns
fmacd	fnmacs	ftouis	ldrh	nop
fmacs	fnmscd	ftouizd	ldrsb	orr
fmdhr	fnmscs	ftouizs	ldrsh	orrs
fmdlr	fnmuld	fuitod	ldrt	pld

qadd	shadd16	smmul	strb	uhsubaddx
qadd16	shadd8	smmulr	strbt	umaal
qadd8	shaddsubx	smuad	strex	umlal
qaddsubx	shsub16	smuadx	strexb	umlals
qdadd	shsub8	smulbb	strexd	umull
qdsub	shsubaddx	smulbt	strexh	umulls
qsub	smlabb	smull	strh	uqadd16
qsub16	smlabt	smulls	strt	uqadd8
qsub8	smlad	smultb	sub	uqaddsubx
qsubaddx	smladx	smultt	subs	uqsub16
rev	smlal	smulwb	swi	uqsub8
rev16	smlals	smulwt	swp	uqsubaddx
revsh	smlalbb	smusd	swpb	usad8
rfeda	smlalbt	smusdx	sxtab	usada8
rfedb	smlald	srsda	sxtab16	usub16
rfeia	smlaldx	srsdb	sxtah	usub8
rfeib	smlaltb	srsia	sxtb	usubaddx
rsb	smlaltt	srsib	sxtb16	uxtab
rsbs	smlatb	ssub16	sxth	uxtab16
rsc	smlatt	ssub8	teq	uxtah
rscs	smlawb	ssubaddx	tst	uxtb
sadd16	smlawt	stc	uadd16	uxtb16
sadd8	smlsd	stcl	uadd8	uxth
saddsubx	smlsdx	stc2	uaddsubx	wfe
sbc	smlsld	stc2l	udf	wfi
sbcs	smlsldx	stmda	uhadd16	yield
sel	smmla	stmdb	uhadd8	
setendbe	smmlar	stmia	uhaddsubx	
setendle	smmls	stmib	uhsub16	
sev	smmlsr	str	uhsub8	

Table 10.2: Supported ARM A64 instruction set (741 instructions)

abs	ands	autiza	b.ne	blraa
adc	asr	autizb	b.nv	blraaz
adcs	asrv	b	b.pl	blrab
add	at	b.al	b.vc	blrabz
addhn	autda	b.cc	b.vs	br
addhn2	autdb	b.cs	bcax	braa
addp	autdza	b.eq	bfc	braaz
adds	autdzb	b.ge	bfi	brab
addv	autia	b.gt	bfm	brabz
adr	autia1716	b.hi	bfxil	brk
adrp	autiasp	b.hs	bic	bsl
aesd	autiaz	b.le	bics	cas
aese	autib	b.lo	bif	casa
aesimc	autib1716	b.ls	bit	casab
aesmc	autibsp	b.lt	bl	casah
and	autibz	b.mi	blr	casal

```
casalb
casalh
casb
cash
casl
caslb
caslh
casp
caspa
caspal
caspl
cbnz
cbz
ccmn
ccmp
cinc
cinv
clrex
cls
clz
cmeq
cmge
cmgt
cmhi
cmhs
cmle
cmlt
cmn
cmp
cmtst
cneg
cnt
crc32b
crc32cb
crc32ch
crc32cw
crc32cx
crc32h
crc32w
crc32x
csel
cset
csetm
csinc
csinv
csneg
dc
dcps1
dcps2
dcps3
dmb
drps
dsb
dup
eon
eor
eor3
eret
eretaa
eretab
esb
ext
extr
fabd
fabs
facge
facgt
fadd
faddp
fcadd
fccmp
fccmpe
fcmeq
fcmge
fcmgt
fcmla
fcmle
fcmlt
fcmp
fcmpe
fcsel
fcvt
fcvtas
fcvtau
fcvtl
fcvtl2
fcvtms
fcvtmu
fcvtn
fcvtn2
fcvtns
fcvtnu
fcvtps
fcvtpu
fcvtxn
fcvtxn2
fcvtzs
fcvtzu
fdiv
fjcvtzs
fmadd
fmax
fmaxnm
fmaxnmp
fmaxnmv
fmaxp
fmaxv
fmin
fminnm
fminnmp
fminnmv
fminp
fminv
fmla
fmlal
fmlal2
fmls
fmlsl
fmlsl2
fmov
fmsub
fmul
fmulx
fneg
fnmadd
fnmsub
fnmul
frecpe
frecps
frecpx
frinta
frinti
frintm
frintn
frintp
frintx
frintz
frsqrte
frsqrts
fsqrt
fsub
hint
hlt
hvc
ic
ins
isb
ld1
ld1r
ld2
ld2r
ld3
ld3r
ld4
ld4r
ldadd
ldadda
ldaddab
ldaddah
ldaddal
ldaddalb
ldaddalh
ldaddb
ldaddh
ldaddl
ldaddlb
ldaddlh
ldapr
ldaprb
ldaprh
ldar
ldarb
ldarh
ldaxp
ldaxr
ldaxrb
ldaxrh
ldclr
ldclra
ldclrab
ldclrah
ldclral
ldclralb
ldclralh
ldclrb
ldclrh
ldclrl
ldclrlb
ldclrlh
ldeor
ldeora
ldeorab
ldeorah
ldeoral
ldeoralb
ldeoralh
ldeorb
ldeorh
ldeorl
ldeorlb
ldeorlh
ldlar
ldlarb
ldlarh
ldnp
ldp
ldpsw
ldr
ldraa
ldrab
ldrb
ldrh
ldrsb
ldrsh
ldrsw
ldset
ldseta
ldsetab
ldsetah
ldsetal
ldsetalb
ldsetalh
ldsetb
ldseth
ldsetl
ldsetlb
ldsetlh
ldsmax
ldsmaxa
ldsmaxab
ldsmaxah
ldsmaxal
ldsmaxalb
ldsmaxalh
ldsmaxb
ldsmaxh
ldsmaxl
ldsmaxlb
ldsmaxlh
ldsmin
ldsmina
ldsminab
ldsminah
ldsminal
ldsminalb
ldsminalh
ldsminb
ldsminh
ldsminl
ldsminlb
```

```
ldsminlh
ldtr
ldtrb
ldtrh
ldtrsb
ldtrsh
ldtrsw
ldumax
ldumaxa
ldumaxab
ldumaxah
ldumaxal
ldumaxalb
ldumaxalh
ldumaxb
ldumaxh
ldumaxl
ldumaxlb
ldumaxlh
ldumin
ldumina
lduminab
lduminah
lduminal
lduminalb
lduminalh
lduminb
lduminh
lduminl
lduminlb
lduminlh
ldur
ldurb
ldurh
ldursb
ldursh
ldursw
ldxp
ldxr
ldxrb
ldxrh
lsl
lslv
lsr
lsrv
madd
mla
mls
mneg
mov
movk
movn
movz
mrs
msr
msub
mul
mvn
neg
negs
ngc
ngcs
nop
not
orn
orr
pacda
pacdb
pacdza
pacdzb
pacga
pacia
pacia1716
paciasp
paciaz
pacib
pacib1716
pacibsp
pacibz
paciza
pacizb
pmul
pmull
pmull2
prfm
prfum
psb
raddhn
raddhn2
rax1
rbit
ret
retaa
retab
rev
rev16
rev32
rev64
ror
rorv
rshrn
rshrn2
rsubhn
rsubhn2
saba
sabal
sabal2
sabd
sabdl
sabdl2
sadalp
saddl
saddl2
saddlp
saddlv
saddw
saddw2
sbc
sbcs
sbfiz
sbfm
sbfx
scvtf
sdiv
sdot
sev
sevl
sha1c
sha1h
sha1m
sha1p
sha1su0
sha1su1
sha256h
sha256h2
sha256su0
sha256su1
sha512h
sha512h2
sha512su0
sha512su1
shadd
shl
shll
shll2
shrn
shrn2
shsub
sli
sm3partw1
sm3partw2
sm3ss1
sm3tt1a
sm3tt1b
sm3tt2a
sm3tt2b
sm4e
sm4ekey
smaddl
smax
smaxp
smaxv
smc
smin
sminp
sminv
smlal
smlal2
smlsl
smlsl2
smnegl
smov
smsubl
smulh
smull
smull2
sqabs
sqadd
sqdmlal
sqdmlal2
sqdmlsl
sqdmlsl2
sqdmulh
sqdmull
sqdmull2
sqneg
sqrdmlah
sqrdmlsh
sqrdmulh
sqrshl
sqrshrn
sqrshrn2
sqrshrun
sqrshrun2
sqshl
sqshlu
sqshrn
sqshrn2
sqshrun
sqshrun2
sqsub
sqxtn
sqxtn2
sqxtun
sqxtun2
srhadd
sri
srshl
srshr
srsra
sshl
sshll
sshll2
sshr
ssra
ssubl
ssubl2
ssubw
ssubw2
st1
st2
st3
st4
stadd
staddb
staddh
staddl
staddlb
staddlh
stclr
stclrb
stclrh
stclrl
stclrlb
stclrlh
steor
steorb
steorh
steorl
steorlb
steorlh
stllr
stllrb
stllrh
stlr
stlrb
stlrh
stlxp
stlxr
stlxrb
```

stlxrh	stumin	sxtl	uhadd	ursqrte
stnp	stuminb	sxtl2	uhsub	ursra
stp	stuminh	sxtw	umaddl	ushl
str	stuminl	sys	umax	ushll
strb	stuminlb	sysl	umaxp	ushll2
strh	stuminlh	tbl	umaxv	ushr
stset	stur	tbnz	umin	usqadd
stsetb	sturb	tbx	uminp	usra
stseth	sturh	tbz	uminv	usubl
stsetl	stxp	tlbi	umlal	usubl2
stsetlb	stxr	trn1	umlal2	usubw
stsetlh	stxrb	trn2	umlsl	usubw2
stsmax	stxrh	tst	umlsl2	uxtb
stsmaxb	sub	uaba	umnegl	uxth
stsmaxh	subhn	uabal	umov	uxtl
stsmaxl	subhn2	uabal2	umsubl	uxtl2
stsmaxlb	subs	uabd	umulh	uzp1
stsmaxlh	suqadd	uabdl	umull	uzp2
stsmin	svc	uabdl2	umull2	wfe
stsminb	swp	uadalp	uqadd	wfi
stsminh	swpa	uaddl	uqrshl	xar
stsminl	swpab	uaddl2	uqrshrn	xpacd
stsminlb	swpah	uaddlp	uqrshrn2	xpaci
stsminlh	swpal	uaddlv	uqshl	xpaclri
sttr	swpalb	uaddw	uqshrn	xtn
sttrb	swpalh	uaddw2	uqshrn2	xtn2
sttrh	swpb	ubfiz	uqsub	yield
stumax	swph	ubfm	uqxtn	zip1
stumaxb	swpl	ubfx	uqxtn2	zip2
stumaxh	swplb	ucvtf	urecpe	
stumaxl	swplh	udf	urhadd	
stumaxlb	sxtb	udiv	urshl	
stumaxlh	sxth	udot	urshr	

Table 10.3: Supported ARM T32 instruction set (471 instructions)

adc	asrs	clrex	dbg	fstmiax
adcs	b	clz	dcps1	hlt
add	bfc	cmn	dcps2	hvc
adds	bfi	cmp	dcps3	isb
addw	bic	cps	dmb	it
adr	bics	cpsid	dsb	lda
aesd	bkpt	cpsie	eor	ldab
aese	bl	crc32b	eors	ldaex
aesimc	blx	crc32cb	eret	ldaexb
aesmc	bx	crc32ch	esb	ldaexd
and	bxj	crc32cw	fldmdbx	ldaexh
ands	cbnz	crc32h	fldmiax	ldah
asr	cbz	crc32w	fstmdbx	ldc

```
ldm
ldmda
ldmdb
ldmea
ldmed
ldmfa
ldmfd
ldmia
ldmib
ldr
ldrb
ldrbt
ldrd
ldrex
ldrexb
ldrexd
ldrexh
ldrh
ldrht
ldrsb
ldrsbt
ldrsh
ldrsht
ldrt
lsl
lsls
lsr
lsrs
mcr
mcrr
mla
mlas
mls
mov
movs
movt
movw
mrc
mrrc
mrs
msr
mul
muls
mvn
mvns
nop
orn
orns
orr
orrs
pkhbt
pkhtb
pld
pldw
pli
pop
push
qadd
qadd16
qadd8
qasx
qdadd
qdsub
qsax
qsub
qsub16
qsub8
rbit
rev
rev16
revsh
rfe
rfeda
rfedb
rfeia
rfeib
ror
rors
rrx
rrxs
rsb
rsbs
rsc
rscs
sadd16
sadd8
sasx
sbc
sbcs
sbfx
sdiv
sel
setend
setpan
sev
sevl
sha1c
sha1h
sha1m
sha1p
sha1su0
sha1su1
sha256h
sha256h2
sha256su0
sha256su1
shadd16
shadd8
shasx
shsax
shsub16
shsub8
smc
smlabb
smlabt
smlad
smladx
smlal
smlalbb
smlalbt
smlald
smlaldx
smlals
smlaltb
smlaltt
smlatb
smlatt
smlawb
smlawt
smlsd
smlsdx
smlsld
smlsldx
smmla
smmlar
smmls
smmlsr
smmul
smmulr
smuad
smuadx
smulbb
smulbt
smull
smulls
smultb
smultt
smulwb
smulwt
smusd
smusdx
srs
srsda
srsdb
srsia
srsib
ssat
ssat16
ssax
ssub16
ssub8
stc
stl
stlb
stlex
stlexb
stlexd
stlexh
stlh
stm
stmda
stmdb
stmea
stmed
stmfa
stmfd
stmia
stmib
str
strb
strbt
strd
strex
strexb
strexd
strexh
strh
strht
strt
sub
subs
subw
svc
sxtab
sxtab16
sxtah
sxtb
sxtb16
sxth
tbb
tbh
teq
tst
uadd16
uadd8
uasx
ubfx
udf
udiv
uhadd16
uhadd8
uhasx
uhsax
uhsub16
uhsub8
umaal
umlal
umlals
umull
umulls
uqadd16
uqadd8
uqasx
uqsax
uqsub16
uqsub8
usad8
usada8
usat
usat16
usax
usub16
usub8
uxtab
uxtab16
uxtah
uxtb
uxtb16
uxth
vaba
vabal
vabd
vabdl
vabs
vacge
vacgt
vacle
vaclt
vadd
vaddhn
```

vaddl	vfmal	vmull	vqshrn	vshrn
vaddw	vfms	vmvn	vqshrun	vsli
vand	vfmsl	vneg	vqsub	vsqrt
vbic	vfnma	vnmla	vraddhn	vsra
vbif	vfnms	vnmls	vrecpe	vsri
vbit	vhadd	vnmul	vrecps	vst1
vbsl	vhsub	vorn	vrev16	vst2
vcadd	vins	vorr	vrev32	vst3
vceq	vjcvt	vpadal	vrev64	vst4
vcge	vld1	vpadd	vrhadd	vstm
vcgt	vld2	vpaddl	vrinta	vstmdb
vcle	vld3	vpmax	vrintm	vstmia
vcls	vld4	vpmin	vrintn	vstr
vclt	vldm	vpop	vrintp	vsub
vclz	vldmdb	vpush	vrintr	vsubhn
vcmla	vldmia	vqabs	vrintx	vsubl
vcmp	vldr	vqadd	vrintz	vsubw
vcmpe	vmax	vqdmlal	vrshl	vswp
vcnt	vmaxnm	vqdmlsl	vrshr	vtbl
vcvt	vmin	vqdmulh	vrshrn	vtbx
vcvta	vminnm	vqdmull	vrsqrte	vtrn
vcvtb	vmla	vqmovn	vrsqrts	vtst
vcvtm	vmlal	vqmovun	vrsra	vudot
vcvtn	vmls	vqneg	vrsubhn	vuzp
vcvtp	vmlsl	vqrdmlah	vsdot	vzip
vcvtr	vmov	vqrdmlsh	vseleq	wfe
vcvtt	vmovl	vqrdmulh	vselge	wfi
vdiv	vmovn	vqrshl	vselgt	yield
vdup	vmovx	vqrshrn	vselvs	
veor	vmrs	vqrshrun	vshl	
vext	vmsr	vqshl	vshll	
vfma	vmul	vqshlu	vshr	

The A64 and T32 assemblers allow users to temporarily switch to the A32 instruction set by passing 32 as operand to the bit mode directive. The original instruction set can be restored using bit modes 64 and 16 respectively.

10.3 Calling Convention

The machine code generators and runtime support for the ARM architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

10.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

10.3.2 Floating-Point Support

The ARM architecture optionally supports floating-point operations. The A32 and A64 generators are able to generate native floating-point operations for processors that do support them.

10.3.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to registers `r0` and `r1` in 32-bit mode or registers `w0` or `x0` in 64-bit mode depending on the size of the actual return type. If supported natively, floating-point results are stored in registers `s0` or `d0` depending on their size.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to register `sp`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to register `r11` in 32-bit mode or `x29` in 64-bit mode.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is supported and mapped to register `lr` in 32-bit mode or register `x30` in 64-bit mode.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

10.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the ARM architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific

runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the ARM hardware architecture as well as the required combination of linker and object files.

10.4.1 General

Basic architectural runtime support is provided by the object files `arma32run`, `arma64run`, `armt32run` and `armt32fperun`. Users should always include one of these object files during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cpparma32run`, `cpparma64run`, `cpparmt32run`, and `cpparmt32fperun` library files respectively. Programs written in Oberon need additional runtime support stored in the `obarma32run`, `obarma64run`, `obarmt32run`, and `obarmt32fperun` library files respectively. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

10.4.2 Darwin

Programs targeting the Darwin operating system are created using the `linkbin` linker tool. It creates Mach-O files [29] if provided with the runtime support stored in the `arma64darwinrun` object file. Calling the `ecsd` utility tool using the `arma64darwin` target environment achieves the same result.

10.4.3 Linux

Programs targeting Linux-based operating systems are created using the `linkbin` linker tool. It creates either 32-bit or 64-bit Executable and Linking Format (ELF) files [2] if provided with the runtime support stored in the `arma32linuxrun`, `arma64linuxrun`, `armt32linuxrun`, and `armt32fpelinuxrun` object files respectively. Calling the `ecsd` utility tool using the `arma32-linux`, `arma64linux`, `armt32linux`, or `armt32fpelinux` target environments achieves the same result. External libraries are available using additional runtime support stored in the following object files: `arma32libdl`, `arma32libpthread`, `arma32libsdl`, `arma64libdl`, `arma64libpthread`, `arma64libsdl`, `armt32libdl`, `armt32libpthread`, `armt32libsdl`, `armt32fpelibdl`, `armt32fpelibpthread`, and `armt32fpelibsdl`.

10.4.4 Raspberry Pi

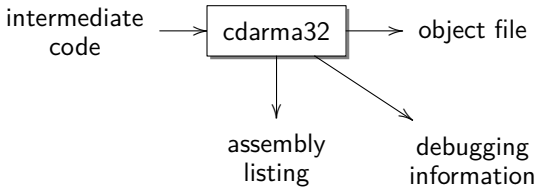
The Eigen Compiler Suite provides basic runtime support for a native Raspberry Pi 2 Model B computer environment stored in the `rpi2brun` object file. Users creating their own systems may use the `linkbin` linker together with this object file in order create an executable bootloader. Calling the `ecsd` utility tool using the `rpi2b` target environment achieves the same result. The UART communication device is configured to use a baud rate of 115200 bps with eight data bits, no parity, and one stop bit.

10.5 ARM Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the ARM hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

10.5.1 cdarma32

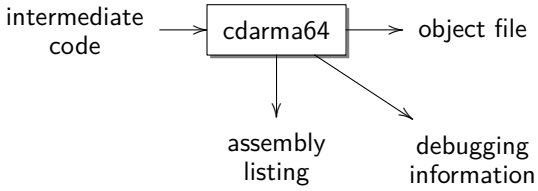
`cdarma32` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

10.5.2 cdarma64

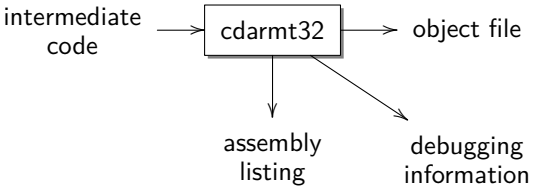
`cdarma64` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

10.5.3 cdarmt32

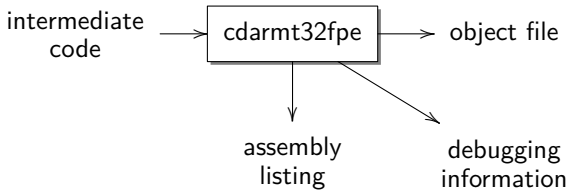
`cdarmt32` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

10.5.4 cdarmt32fpe

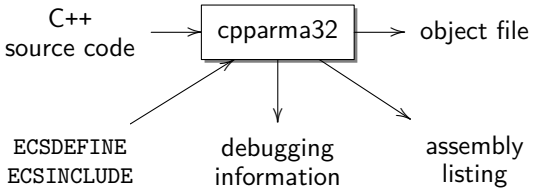
`cdarmt32fpe` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

10.5.5 cpparma32

`cpparma32` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__arma32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparma32run` library file.

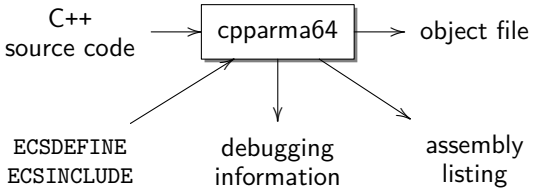


For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and

their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.6 cpparma64

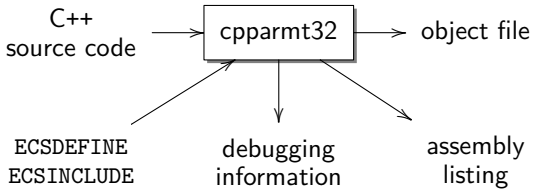
`cpparma64` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__arma64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparma64run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.7 cpparmt32

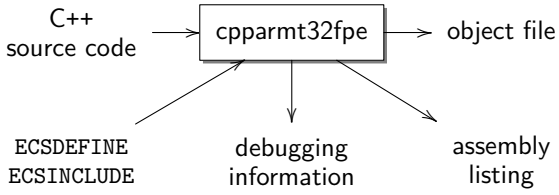
`cpparmt32` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__armt32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparmt32run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.8 cpparmt32fpe

`cpparmt32fpe` is a compiler for the C++ programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__armt32fpe__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cpparmt32fperun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.9 falarma32

`falarma32` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

10.5.10 falarma64

`falarma64` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

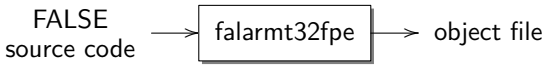
10.5.11 falarmt32

`falarmt32` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

10.5.12 falarmt32fpe

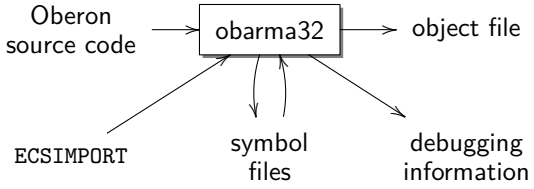
`falarmt32fpe` is a compiler for the FALSE programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in FALSE and stores it in corresponding object files.



For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

10.5.13 obarma32

`obarma32` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarma32run` library file.

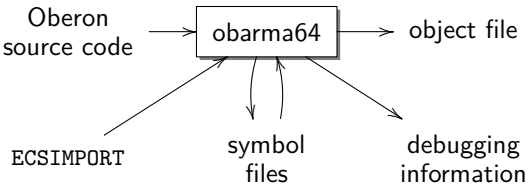


For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.14 obarma64

`obarma64` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from

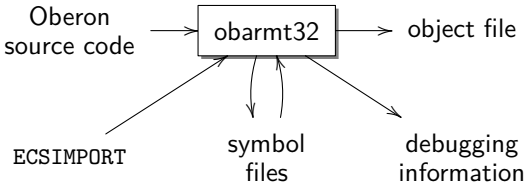
modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarma64run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.15 obarmt32

`obarmt32` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarmt32run` library file.

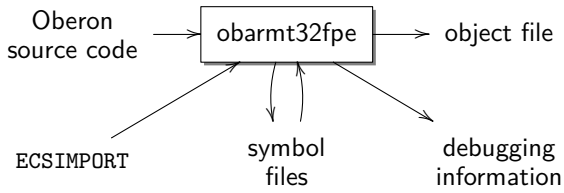


For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and

their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.16 obarmt32fpe

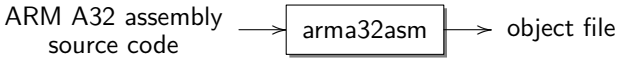
`obarmt32fpe` is a compiler for the Oberon programming language targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obarmt32fperun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

10.5.17 arma32asm

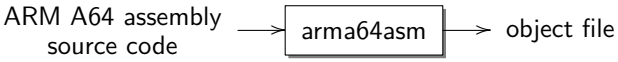
`arma32asm` is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing A32 instructions and stores it in corresponding object files.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

10.5.18 arma64asm

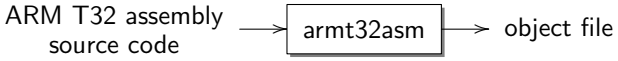
`arma64asm` is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing A64 instructions and stores it in corresponding object files.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

10.5.19 armt32asm

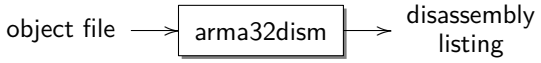
`armt32asm` is an assembler for the ARM hardware architecture. It translates assembly code into machine code for ARM processors executing T32 instructions and stores it in corresponding object files.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

10.5.20 arma32dism

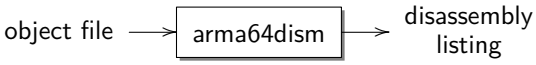
`arma32dism` is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing A32 instructions into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

10.5.21 arma64dism

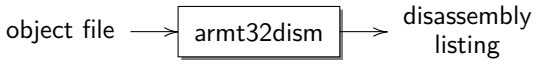
`arma64dism` is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing A64 instructions into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

10.5.22 armt32dism

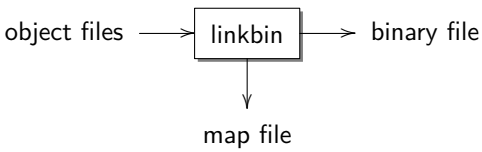
`armt32dism` is a disassembler for the ARM hardware architecture. It translates machine code from object files targeting ARM processors executing T32 instructions into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

10.5.23 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.

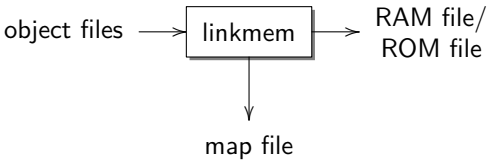


For more information about object files and their purpose, see Chapter 22.

10.5.24 linkmem

`linkmem` is a linker for plain binary files partitioned into random-access and read-only memory. It links all object files given to it into two distinct images, one for data sections and one for code and constant data sections, and stores each image in a binary file that begins with the first

linked section of the corresponding type. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order.



For more information about object files and their purpose, see Chapter 22.

11 AVR

This chapter describes how the Eigen Compiler Suite supports the AVR hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

11.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the AVR hardware architecture by Atmel. Figure 11.1 shows the data flow in-between these tools.

All compilers targeting the AVR architecture translate their programs using an intermediate code representation. The AVR generator is able to translate the intermediate code representation of a program into machine code for AVR processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

11.2 Instruction Set

Tools targeting the AVR architecture support the instruction set listed in Table 11.1 and use the same assembly syntax as predefined by Atmel [15]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 11.1: Supported AVR instruction set (119 instructions)

adc	brbc	brhs	brsh	cbi
add	brbs	brid	brtc	cbr
adiw	brcc	brie	brts	clc
and	brcs	brlo	brvc	clh
andi	break	brlt	brvs	cli
asr	breq	brmi	bset	cln
bclr	brge	brne	bst	clr
bld	brhc	brpl	call	cls

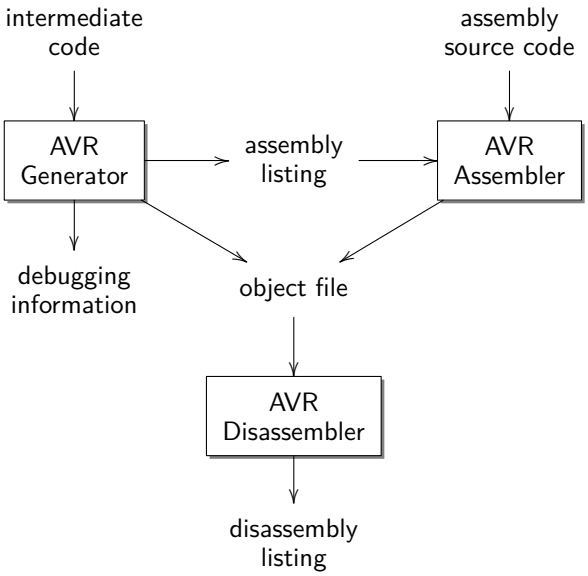


Figure 11.1: Data flow within the tools targeting the AVR architecture

clt	fmulsu	mov	rol	ses
clv	icall	movw	ror	set
clz	ijmp	mul	sbc	sev
com	in	muls	sbci	sez
cp	inc	mulsu	sbi	sleep
cpc	jmp	neg	sbic	spm
cpi	lac	nop	sbis	st
cpse	las	or	sbiw	std
dec	lat	ori	sbr	sts
des	ld	out	sbrc	sub
eicall	ldd	pop	sbrs	subi
eijmp	ldi	push	sec	swap
elpm	lds	rcall	seh	tst
eor	lpm	ret	sei	wdr
fmul	lsl	reti	sen	xch
fmuls	lsr	rjmp	ser	

11.3 Calling Convention

The machine code generator and runtime support for the AVR architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability. In

general, the order of memory accesses to values that consist of several octets is most significant byte first or big-endian respectively.

11.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller. Because the AVR instructions `push` and `pop` modify the stack register in a different order with respect to the one defined by the intermediate code, all data on the stack is accessed using a one octet displacement.

11.3.2 Floating-Point Support

The AVR architecture does not support any floating-point operations natively. The AVR runtime support has currently no software emulation for floating-point operations.

11.3.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to AVR registers `r0` up to `r7` depending on the size of the actual return type.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to AVR registers `SPL` and `SPH` respectively.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to AVR registers `r24` and `r25` respectively.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is not supported.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

11.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the AVR architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the AVR hardware architecture as well as the required combination of linker and object files.

11.4.1 General

Basic architectural runtime support is provided by the object file `avrrun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

The Eigen Compiler Suite additionally supports remote execution of programs targeting AVR processors. The corresponding runtime support is stored in the `avrremote` object file. It synchronizes the start of a program and its result with a host machine using the standard output stream.

Programs written in C++ need additional runtime support stored in the `cppavrrun` and `cppavrxtrun` library files. Programs written in Oberon need additional runtime support stored in the `obavrrun` and `obavrxtrun` library files. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

11.4.2 ATmega2560 Microcontrollers

Programs targeting the ATmega2560 microcontroller by Atmel [31] are created using the `linkhex` linker tool. It creates an Intel HEX file [3] for programming the microcontroller if provided with the runtime support stored in `axtmega2560run` object file. Calling the `ecsd` utility tool using the `axtmega2560` target environment achieves the same result. The USART communication device is configured to use a baud rate of 115200 bps with eight data bits, no parity, and one stop bit.

11.4.3 ATmega32 Microcontrollers

Programs targeting the ATmega32 microcontroller by Atmel [32] are created using the `linkhex` linker tool. It creates an Intel HEX file [3] for programming the microcontroller if provided with the runtime support stored in `atmega32run` object file. Calling the `ecsd` utility tool using the `atmega32` target environment achieves the same result. The USART communication device is configured to use a baud rate of 115200 bps with eight data bits, no parity, and one stop bit.

11.4.4 ATmega328 Microcontrollers

Programs targeting the ATmega328 microcontroller by Atmel [33] are created using the `linkhex` linker tool. It creates an Intel HEX file [3] for programming the microcontroller if provided with the runtime support stored in `atmega328run` object file. Calling the `ecsd` utility tool using the `atmega328` target environment achieves the same result. The USART communication device is configured to use a baud rate of 115200 bps with eight data bits, no parity, and one stop bit.

11.4.5 ATmega8515 Microcontrollers

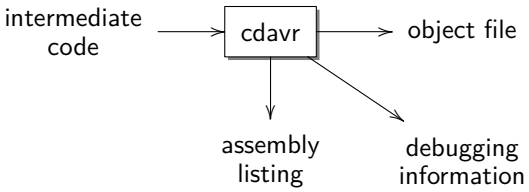
Programs targeting the ATmega8515 microcontroller by Atmel [34] are created using the `linkhex` linker tool. It creates an Intel HEX file [3] for programming the microcontroller if provided with the runtime support stored in `atmega8515run` object file. Calling the `ecsd` utility tool using the `atmega8515` target environment achieves the same result. The USART communication device is configured to use a baud rate of 115200 bps with eight data bits, no parity, and one stop bit.

11.5 AVR Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the AVR hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

11.5.1 cdavr

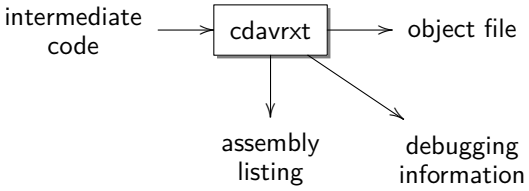
`cdavr` is a compiler for intermediate code targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

11.5.2 cdavrxt

`cdavrxt` is a compiler for intermediate code targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.

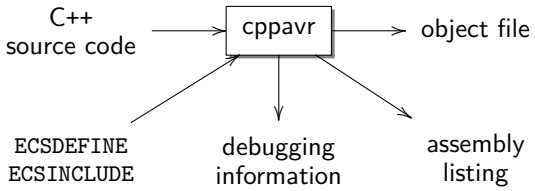


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

11.5.3 cppavr

`cppavr` is a compiler for the C++ programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro

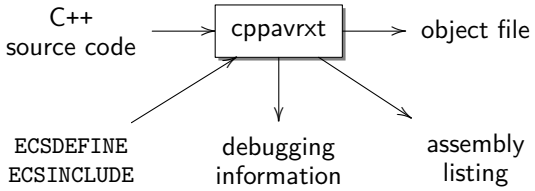
`__avr__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavrrun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

11.5.4 cppavrxt

`cppavrxt` is a compiler for the C++ programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avrxt__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavrxtrun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

11.5.5 falavr

`falavr` is a compiler for the FALSE programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

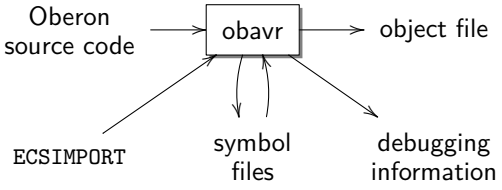
11.5.6 falavrxt

`falavrxt` is a compiler for the FALSE programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

11.5.7 obavr

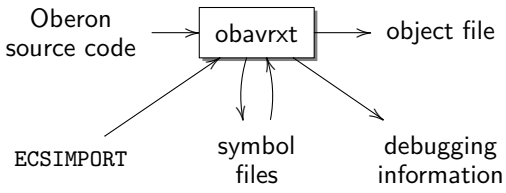
`obavr` is a compiler for the Oberon programming language targeting the AVR hardware architecture. It generates machine code for AVR processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavrrun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

11.5.8 obavrxt

`obavrxt` is a compiler for the Oberon programming language targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavrxtrun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

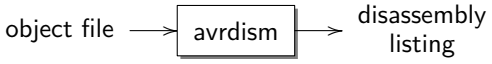
11.5.9 avrasm

`avrasm` is an assembler for the AVR hardware architecture. It translates assembly code into machine code for AVR processors and stores it in corresponding object files. The identifiers `RXL`, `RXH`, `RYL`, `RYH`, `RZL`, and `RZH` are predefined and name the corresponding registers. The identifiers `SPL` and `SPH` are also predefined and evaluate to the address of the corresponding registers.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

11.5.10 avrdism

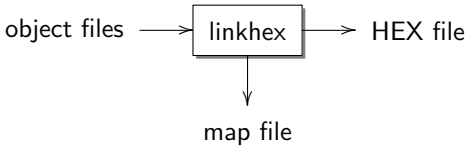
`avrdism` is a disassembler for the AVR hardware architecture. It translates machine code from object files targeting AVR processors into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

11.5.11 linkhex

`linkhex` is a linker for Intel HEX files. It links all code sections of the object files given to it into single image and stores the image in an Intel HEX file [3] that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order.



For more information about object files and their purpose, see Chapter 22.

12 AVR32

This chapter describes how the Eigen Compiler Suite supports the AVR32 hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

12.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the AVR32 hardware architecture by Atmel. Figure 12.1 shows the data flow in-between these tools.

All compilers targeting the AVR32 architecture translate their programs using an intermediate code representation. The AVR32 generator is able to translate the intermediate code representation of a program into machine code for AVR32 processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

12.2 Instruction Set

Tools targeting the AVR32 architecture support the instruction set listed in Table 12.1 and use the same assembly syntax as predefined by Atmel [16]. Instructions that operate on register lists are not supported. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 12.1: Supported AVR32 instruction set (551 instructions)

abs	addcc	addhs	addpl	andcs
acall	addcs	addle	addqs	andeq
acr	addeq	addlo	addvc	andge
adc	addge	addls	addvs	andgt
add	addgt	addlt	and	andh
addabs	addhh.w	addmi	andal	andhi
addal	addhi	addne	andcc	andhs

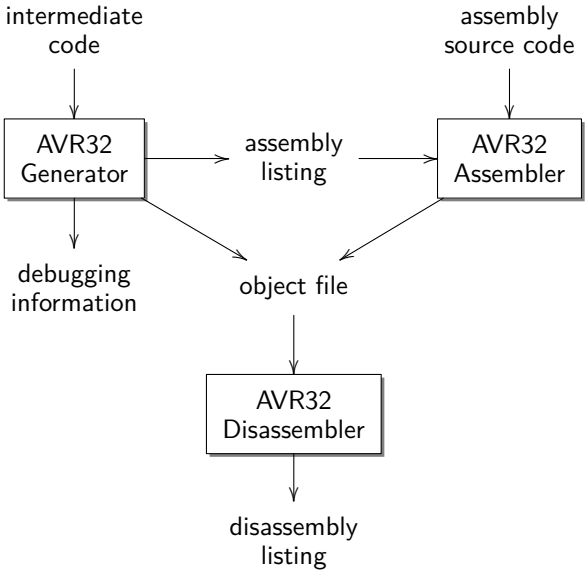


Figure 12.1: Data flow within the tools targeting the AVR32 architecture

andl	brev	clz	eorle	ld.sbhs
andle	brge	com	eorlo	ld.sble
andlo	brgt	cop	eorls	ld.sblo
andls	brhi	cp.b	eorlt	ld.sbls
andlt	brhs	cp.h	eormi	ld.sblt
andmi	brle	cp.w	eorne	ld.sbmi
andn	brlo	cpc	eorpl	ld.sbne
andne	brls	csrf	eorqs	ld.sbpl
andpl	brlt	csrfcz	eorvc	ld.sbqs
andqs	brmi	divs	eorvs	ld.sbvc
andvc	brne	divu	frs	ld.sbvs
andvs	brpl	eor	icall	ld.sh
asr	brqs	eoral	incjosp	ld.shal
bfexts	brvc	eorcc	ld.d	ld.shcc
bfextu	brvs	eorcs	ld.sb	ld.shcs
bfins	bst	eoreq	ld.sbal	ld.sheq
bld	cache	eorge	ld.sbcc	ld.shge
bral	casts.b	eorgt	ld.sbcs	ld.shgt
brcc	casts.h	eorh	ld.sbeq	ld.shhi
brcs	castu.b	eorhi	ld.sbge	ld.shhs
breakpoint	castu.h	eorhs	ld.sbgt	ld.shle
breq	cbr	eorl	ld.sbhi	ld.shlo

```
ld.shls
ld.shlt
ld.shmi
ld.shne
ld.shpl
ld.shqs
ld.shvc
ld.shvs
ld.ub
ld.ubal
ld.ubcc
ld.ubcs
ld.ubeq
ld.ubge
ld.ubgt
ld.ubhi
ld.ubhs
ld.uble
ld.ublo
ld.ubls
ld.ublt
ld.ubmi
ld.ubne
ld.ubpl
ld.ubqs
ld.ubvc
ld.ubvs
ld.uh
ld.uhal
ld.uhcc
ld.uhcs
ld.uheq
ld.uhge
ld.uhgt
ld.uhhi
ld.uhhs
ld.uhle
ld.uhlo
ld.uhls
ld.uhlt
ld.uhmi
ld.uhne
ld.uhpl
ld.uhqs
ld.uhvc
ld.uhvs
ld.w
ld.wal
ld.wcc
ld.wcs
ld.weq
ld.wge
ld.wgt
ld.whi
ld.whs
ld.wle
ld.wlo
ld.wls
ld.wlt
ld.wmi
ld.wne
ld.wpl
ld.wqs
ld.wvc
ld.wvs
ldc.d
ldc.w
ldc0.d
ldc0.w
lddpc
lddsp
ldins.b
ldins.h
ldswp.sh
ldswp.uh
ldswp.w
lsl
lsr
mac
machh.d
machh.w
macs.d
macsathh.w
macu.d
macwh.d
max
mcall
memc
mems
memt
mfdr
mfsr
min
mov
moval
movcc
movcs
moveq
movge
movgt
movh
movhi
movhs
movl
movle
movlo
movls
movlt
movmi
movne
movpl
movqs
movvc
movvs
mtdr
mtsr
mul
mulhh.w
mulnhh.w
mulnwh.d
muls.d
mulsathh.h
mulsathh.w
mulsatrndhh.h
mulsatrndhh.w
mulsatwh.w
mulu.d
mulwh.d
musfr
mustr
mvcr.d
mvcr.w
mvrc.d
mvrc.w
neg
nop
or
oral
orcc
orcs
oreq
orge
orgt
orh
orhi
orhs
orl
orle
orlo
orls
orlt
ormi
orne
orpl
orqs
orvc
orvs
pabs.sb
pabs.sh
packsh.sb
packsh.ub
packw.sh
padd.b
padd.h
paddh.sh
paddh.ub
padds.sb
padds.sh
padds.ub
padds.uh
paddsub.h
paddsubh.sh
paddsubs.sh
paddsubs.uh
paddx.h
paddxh.sh
paddxs.sh
paddxs.uh
pasr.b
pasr.h
pavg.sh
pavg.ub
plsl.b
plsl.h
plsr.b
plsr.h
pmax.sh
pmax.ub
pmin.sh
pmin.ub
popjc
pref
psad
psub.b
psub.h
psubadd.h
psubaddh.sh
psubadds.sh
psubadds.uh
psubh.sh
psubh.ub
psubs.sb
psubs.sh
psubs.ub
psubs.uh
psubx.h
psubxh.sh
psubxs.sh
psubxs.uh
punpcksb.h
punpckub.h
pushjc
rcall
retal
retcc
retcs
retd
rete
reteq
retge
retgt
rethi
reths
retj
retle
retlo
retls
retlt
retmi
retne
retpl
retqs
rets
retss
retvc
retvs
rjmp
rol
ror
rsub
rsubal
rsubcc
rsubcs
rsubeq
rsubge
rsubgt
rsubhi
rsubhs
rsuble
rsublo
```

```
rsubls          srlt            st.hcs          st.wqs          subfqs
rsublt          srmi            st.heq          st.wvc          subfvc
rsubmi          srne            st.hge          st.wvs          subfvs
rsubne          srpl            st.hgt          stc.d           subge
rsubpl          srqs            st.hhi          stc.w           subgt
rsubqs          srvc            st.hhs          stc0.d          subhh.w
rsubvc          srvs            st.hle          stc0.w          subhi
rsubvs          sscall          st.hlo          stcond          subhs
satadd.h        ssrf            st.hls          stdsp           suble
satadd.w        st.b            st.hlt          sthh.w          sublo
satrnds         st.bal          st.hmi          stswp.h         subls
satrndu         st.bcc          st.hne          stswp.w         sublt
sats            st.bcs          st.hpl          sub             submi
satsub.h        st.beq          st.hqs          subal           subne
satsub.w        st.bge          st.hvc          subcc           subpl
satu            st.bgt          st.hvs          subcs           subqs
sbc             st.bhi          st.w            subeq           subvc
sbr             st.bhs          st.wal          subfal          subvs
scall           st.ble          st.wcc          subfcc          swap.b
scr             st.blo          st.wcs          subfcs          swap.bh
sleep           st.bls          st.weq          subfeq          swap.h
sral            st.blt          st.wge          subfge          sync
srcc            st.bmi          st.wgt          subfgt          tlbr
srcs            st.bne          st.whi          subfhi          tlbs
sreq            st.bpl          st.whs          subfhs          tlbw
srge            st.bqs          st.wle          subfle          tnbz
srgt            st.bvc          st.wlo          subflo          tst
srhi            st.bvs          st.wls          subfls          xchg
srhs            st.d            st.wlt          subflt
srle            st.h            st.wmi          subfmi
srlo            st.hal          st.wne          subfne
srls            st.hcc          st.wpl          subfpl
```

12.3 Calling Convention

The machine code generator and runtime support for the AVR32 architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

12.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

12.3.2 Floating-Point Support

There is currently no support for floating-point operations.

12.3.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to AVR32 registers `r0` and `r1` depending on the size of the actual return type.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to the AVR32 register `sp`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to the AVR32 register `r12`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is supported and mapped to the AVR32 register `lr`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

12.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the AVR32 architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the AVR32 hardware architecture as well as the required combination of linker and object files.

12.4.1 General

Basic architectural runtime support is provided by the object file `avr32run`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppavr32run` library file. Programs written in Oberon need additional runtime support stored in the `obavr32run` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

12.4.2 AT32UC3A3 Microcontrollers

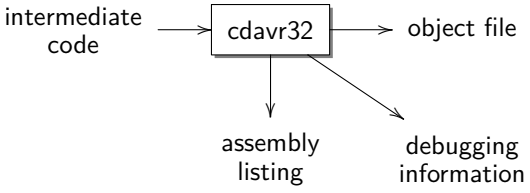
Programs targeting the AT32UC3A3 microcontroller by Atmel [35] are created using the `linkhex` linker tool. It creates an Intel HEX file [3] for programming the microcontroller if provided with the runtime support stored in `at32uc3a3run` object file. Calling the `ecsd` utility tool using the `at32uc3a3` target environment achieves the same result.

12.5 AVR32 Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the AVR32 hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

12.5.1 cdavr32

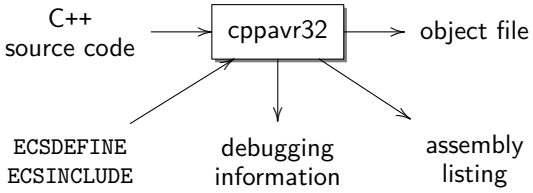
`cdavr32` is a compiler for intermediate code targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

12.5.2 cppavr32

`cppavr32` is a compiler for the C++ programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__avr32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppavr32run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

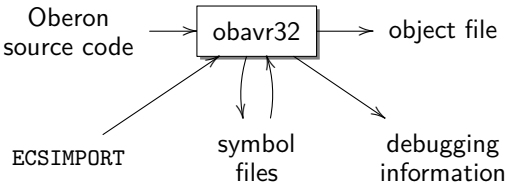
12.5.3 falavr32

`falavr32` is a compiler for the FALSE programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

12.5.4 obavr32

`obavr32` is a compiler for the Oberon programming language targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obavr32run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

12.5.5 avr32asm

`avr32asm` is an assembler for the AVR32 hardware architecture. It translates assembly code into machine code for AVR32 processors and stores it in corresponding object files.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

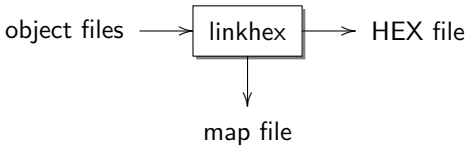
12.5.6 avr32dism

`avr32dism` is a disassembler for the AVR32 hardware architecture. It translates machine code from object files targeting AVR32 processors into assembly code and writes it to the standard output stream.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

12.5.7 linkhex

`linkhex` is a linker for Intel HEX files. It links all code sections of the object files given to it into single image and stores the image in an Intel HEX file [3] that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order.



For more information about object files and their purpose, see Chapter 22.

13 M68000

This chapter describes how the Eigen Compiler Suite supports the M68000 hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

13.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the M68000 hardware architecture by Motorola. Figure 13.1 shows the data flow in-between these tools.

All compilers targeting the M68000 architecture translate their programs using an intermediate code representation. The M68000 generator is able to translate the intermediate code representation of a program into machine code for M68000 processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

13.2 Instruction Set

Tools targeting the M68000 architecture support the instruction set listed in Table 13.1 and use the same assembly syntax as predefined by Motorola [17]. The only exception are immediate values which are not prefixed by a number sign. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 13.1: Supported M68000 instruction set (120 instructions)

abcd	andi	beq	bls	bsr
add	asl	bge	blt	btst
adda	asr	bgt	bmi	bvc
addi	bcc	bhi	bne	bvs
addq	bchg	bhs	bpl	chk
addx	bclr	ble	bra	clr
and	bcs	blo	bset	cmp

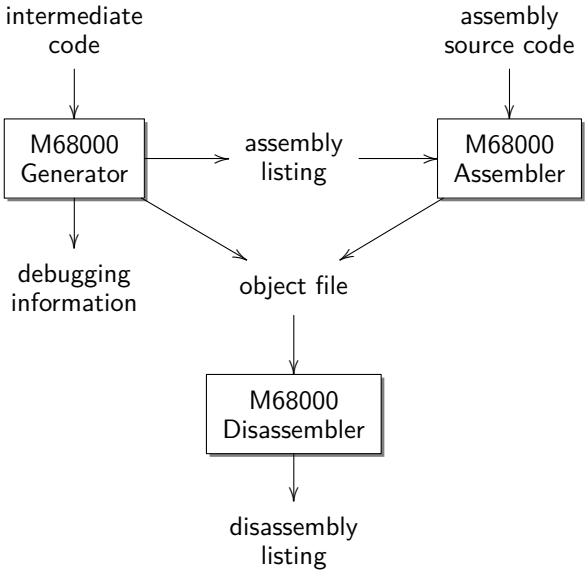


Figure 13.1: Data flow within the tools targeting the M68000 architecture

`cmpa`	`dbvc`	`movep`	`rte`	`sne`
`cmpi`	`dbvs`	`moveq`	`rtr`	`spl`
`cmpm`	`divs`	`muls`	`rts`	`st`
`dbcc`	`divu`	`mulu`	`sbcd`	`stop`
`dbcs`	`eor`	`nbcd`	`scc`	`sub`
`dbeq`	`eori`	`neg`	`scs`	`suba`
`dbge`	`exg`	`negx`	`seq`	`subi`
`dbgt`	`ext`	`nop`	`sf`	`subq`
`dbhi`	`illegal`	`not`	`sge`	`subx`
`dbhs`	`jmp`	`or`	`sgt`	`svc`
`dble`	`jsr`	`ori`	`shi`	`svs`
`dblo`	`lea`	`pea`	`shs`	`swap`
`dbls`	`link`	`reset`	`sle`	`tas`
`dblt`	`lsl`	`rol`	`slo`	`trap`
`dbmi`	`lsr`	`ror`	`sls`	`trapv`
`dbne`	`move`	`roxl`	`slt`	`tst`
`dbpl`	`movea`	`roxr`	`smi`	`unlk`

13.3 Calling Convention

The machine code generator and runtime support for the M68000 architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

13.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

13.3.2 Floating-Point Support

There is currently no support for floating-point operations.

13.3.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to M68000 registers `a0` or `d0` and `d1` depending on the actual return type.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to the M68000 register `a7` (`sp`).

- Frame Pointer Register `$fp`

  The intermediate code stack pointer register `$fp` is mapped to the M68000 register `a6`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

13.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the M68000 architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the M68000 hardware architecture as well as the required combination of linker and object files.

13.4.1 General

Basic architectural runtime support is provided by the object file `m68krun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppm68krun` library file. Programs written in Oberon need additional runtime support stored in the `obm68krun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

13.4.2 Atari TOS

Programs targeting the Atari TOS operating system are created using the `linkprg` linker tool. It creates GEMDOS executable files [4] if provided with the runtime support stored in the `tosrun` object file. Calling the `ecsd` utility tool using the `tos` target environment achieves the same result. GEMDOS, BIOS, and XBIOS functions are available using additional runtime support stored in the `tosapi` object file.

13.4.3 Linux

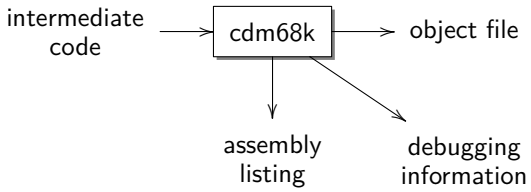
Programs targeting Linux-based operating systems are created using the `linkbin` linker tool. It creates Executable and Linking Format (ELF) files [2] if provided with the runtime support stored in the `m68klinuxrun` object file. Calling the `ecsd` utility tool using the `m68klinux` target environment achieves the same result. External libraries are available using additional runtime support stored in the following object files: `m68klibdl`, `m68klibpthread`, and `m68klibsdl`.

13.5 M68000 Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the M68000 hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

13.5.1 cdm68k

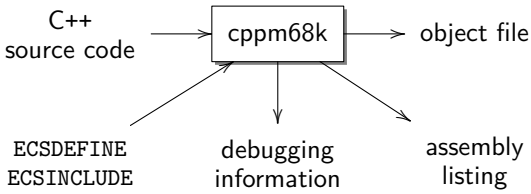
`cdm68k` is a compiler for intermediate code targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

13.5.2 cppm68k

`cppm68k` is a compiler for the C++ programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__m68k__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppm68krun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and

their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

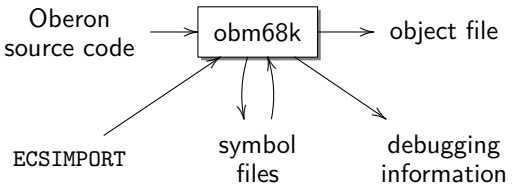
13.5.3 falm68k

`falm68k` is a compiler for the FALSE programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

13.5.4 obm68k

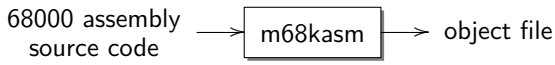
`obm68k` is a compiler for the Oberon programming language targeting the M68000 hardware architecture. It generates machine code for M68000 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obm68krun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

13.5.5 m68kasm

`m68kasm` is an assembler for the M68000 hardware architecture. It translates assembly code into machine code for M68000 processors and stores it in corresponding object files.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

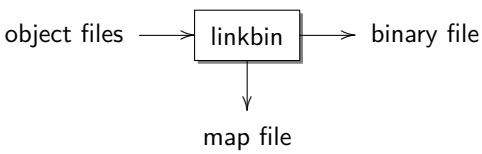
13.5.6 m68kdism

`m68kdism` is a disassembler for the M68000 hardware architecture. It translates machine code from object files targeting M68000 processors into assembly code and writes it to the standard output stream.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

13.5.7 linkbin

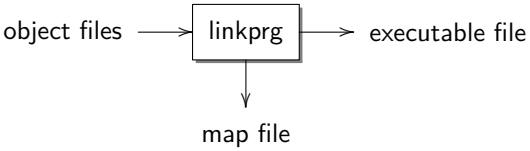
`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

13.5.8 linkprg

`linkprg` is a linker for GEMDOS executable files. It links all object files given to it into a single image and stores the image in an Atari GEMDOS executable file [4]. It also creates a map file that lists the address relative to the text segment, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting executable file can be specified by putting it into a constant data section called `_extension`. The GEMDOS executable file format requires all patch patterns of absolute link patches to consist of four full bitmasks with descending offsets.



For more information about object files and their purpose, see Chapter 22.

14 MicroBlaze

This chapter describes how the Eigen Compiler Suite supports the MicroBlaze hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

14.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the MicroBlaze hardware architecture by Xilinx. Figure 14.1 shows the data flow in-between these tools.

All compilers targeting the MicroBlaze architecture translate their programs using an intermediate code representation. The MicroBlaze generator is able to translate the intermediate code representation of a program into machine code for MicroBlaze processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

14.2 Instruction Set

Tools targeting the MicroBlaze architecture support the instruction set listed in Table 14.1 and use the same assembly syntax as predefined by Xilinx [18]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 14.1: Supported MicroBlaze instruction set (233 instructions)

`add`	`addkc`	`aput`	`bged`	`ble`
`addc`	`aget`	`aputd`	`bgei`	`bled`
`addi`	`agetd`	`beq`	`bgeid`	`blei`
`addic`	`and`	`beqd`	`bgt`	`bleid`
`addik`	`andi`	`beqi`	`bgtd`	`blt`
`addikc`	`andn`	`beqid`	`bgti`	`bltd`
`addk`	`andni`	`bge`	`bgtid`	`blti`

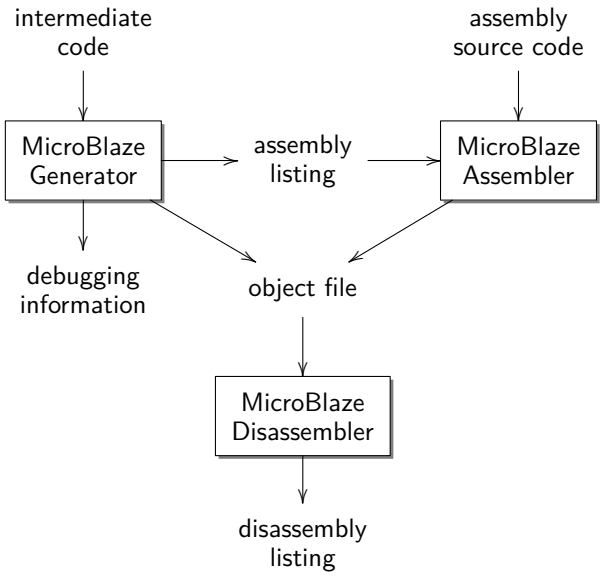


Figure 14.1: Data flow within the tools targeting the MicroBlaze architecture

bltid	bsrai	fadd	lhu	ncagetd
bne	bsrl	fcmp.eq	lhui	ncaput
bned	bsrli	fcmp.ge	lhur	ncaputd
bnei	caget	fcmp.gt	lw	ncget
bneid	cagetd	fcmp.le	lwi	ncgetd
br	caput	fcmp.lt	lwr	ncput
bra	caputd	fcmp.ne	lwx	ncputd
brad	cget	fcmp.un	mbar	neaget
brai	cgetd	fdiv	mfs	neagetd
braid	clz	fint	msrclr	necaget
brald	cmp	flt	msrset	necagetd
bralid	cmpu	fmul	mts	necget
brd	cput	frsub	mul	necgetd
bri	cputd	fsqrt	mulh	neget
brid	eaget	get	mulhsu	negetd
brk	eagetd	getd	mulhu	nget
brki	ecaget	idiv	muli	ngetd
brld	ecagetd	idivu	naget	nop
brlid	ecget	imm	nagetd	nput
bsll	ecgetd	lbu	naput	nputd
bslli	eget	lbui	naputd	or
bsra	egetd	lbur	ncaget	ori

pcmpbf	sbi	taput	tget	tnecget
pcmpeq	sbr	taputd	tgetd	tnecgetd
pcmpne	sext16	tcaget	tnaget	tneget
put	sext8	tcagetd	tnagetd	tnegetd
putd	sh	tcaput	tnaput	tnget
rsub	shi	tcaputd	tnaputd	tngetd
rsubc	shr	tcget	tncaget	tnput
rsubi	sra	tcgetd	tncagetd	tnputd
rsubic	src	tcput	tncaput	tput
rsubik	srl	tcputd	tncaputd	tputd
rsubikc	sw	teaget	tncget	wdc
rsubk	swapb	teagetd	tncgetd	wdc.clear
rsubkc	swaph	tecaget	tncput	wdc.flush
rtbd	swi	tecagetd	tncputd	wic
rted	swr	tecget	tneaget	xor
rtid	swx	tecgetd	tneagetd	xori
rtsd	taget	teget	tnecaget	
sb	tagetd	tegetd	tnecagetd	

14.3 Calling Convention

The machine code generator and runtime support for the MicroBlaze architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

14.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

14.3.2 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to the MicroBlaze registers `r4` and `r5` depending on the size of the actual return type.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to the MicroBlaze register `r1`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to the MicroBlaze register `r2`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is supported and mapped to the MicroBlaze register `r15`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

14.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the MicroBlaze architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the MicroBlaze hardware architecture as well as the required combination of linker and object files.

14.4.1 General

Basic architectural runtime support is provided by the object file `miblrun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

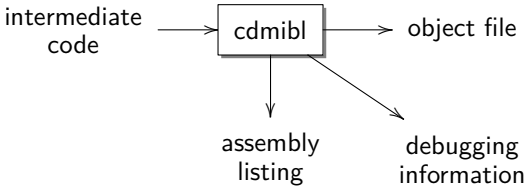
Programs written in C++ need additional runtime support stored in the `cppmiblrun` library file. Programs written in Oberon need additional runtime support stored in the `obmiblrun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

14.5 MicroBlaze Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the MicroBlaze hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

14.5.1 cdmibl

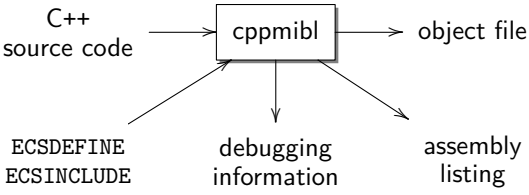
`cdmibl` is a compiler for intermediate code targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

14.5.2 cppmibl

`cppmibl` is a compiler for the C++ programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mibl__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmiblrun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and

their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

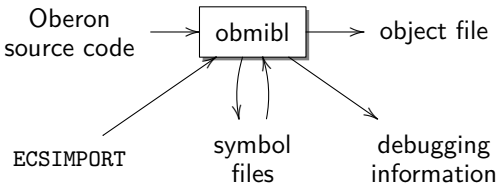
14.5.3 falmibl

`falmibl` is a compiler for the FALSE programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

14.5.4 obmibl

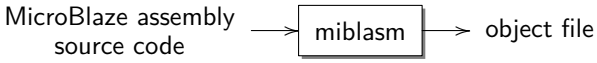
`obmibl` is a compiler for the Oberon programming language targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmiblrun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

14.5.5 miblasm

`miblasm` is an assembler for the MicroBlaze hardware architecture. It translates assembly code into machine code for MicroBlaze processors and stores it in corresponding object files.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

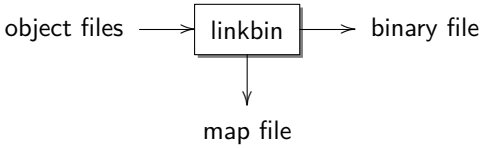
14.5.6 mibldism

`mibldism` is a disassembler for the MicroBlaze hardware architecture. It translates machine code from object files targeting MicroBlaze processors into assembly code and writes it to the standard output stream.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

14.5.7 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

15 MIPS

This chapter describes how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures. This includes information about the assemblers, disassemblers, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

15.1 Introduction

The Eigen Compiler Suite features various compilers, assemblers, and disassemblers that target the MIPS32 and MIPS64 hardware architectures by MIPS Limited. Figure 15.1 shows the data flow in-between these tools.

All compilers targeting the MIPS32 and MIPS64 architectures translate their programs using an intermediate code representation. The MIPS generator is able to translate the intermediate code representation of a program into machine code for MIPS32 and MIPS64 processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assemblers yielding exactly the same object file. The disassemblers are able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

15.2 Instruction Set

Tools targeting the MIPS32 and MIPS64 architectures support the instruction set listed in Table 15.1 and use the same assembly syntax as predefined by MIPS [19, 20]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 15.1: Supported MIPS32 and MIPS64 instruction set (316 instructions)

`abs.d`	`addi`	`bal`	`bc2t`	`bgezl`
`abs.ps`	`addiu`	`bc1f`	`bc2tl`	`bgtz`
`abs.s`	`addu`	`bc1fl`	`beq`	`bgtzl`
`add`	`alnv.ps`	`bc1t`	`beql`	`blez`
`add.d`	`and`	`bc1tl`	`bgez`	`blezl`
`add.ps`	`andi`	`bc2f`	`bgezal`	`bltz`
`add.s`	`b`	`bc2fl`	`bgezall`	`bltzal`

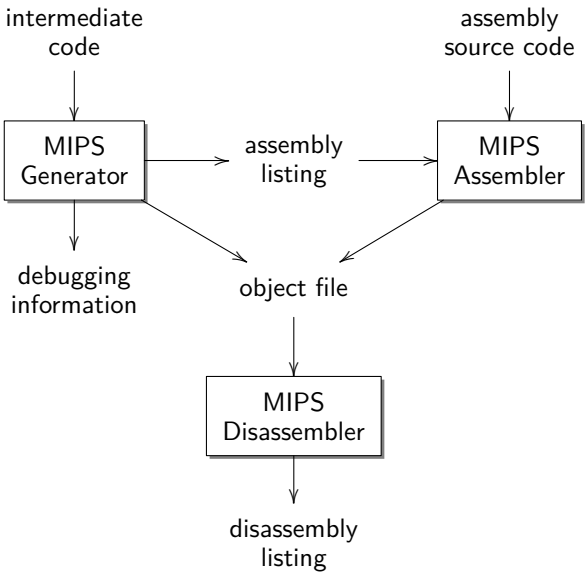


Figure 15.1: Data flow within the tools targeting the MIPS32 and MIPS64 architectures

bltzall	c.ule.s	cvt.l.s	dins	dsllv
bltzl	c.ult.d	cvt.ps.s	dinsm	dsra
bne	c.ult.ps	cvt.s.d	dinsu	dsra32
bnel	c.ult.s	cvt.s.l	div	dsrav
break	c.un.d	cvt.s.pl	div.d	dsrl
c.eq.d	c.un.ps	cvt.s.pu	div.s	dsrl32
c.eq.ps	c.un.s	cvt.s.w	divu	dsrlv
c.eq.s	cache	cvt.w.d	dmfc0	dsub
c.f.d	ceil.l.d	cvt.w.s	dmfc1	dsubu
c.f.ps	ceil.l.s	dadd	dmfc2	ehb
c.f.s	ceil.w.d	daddi	dmtc0	ei
c.ole.d	ceil.w.s	daddiu	dmtc1	eret
c.ole.ps	cfc1	daddu	dmtc2	ext
c.ole.s	cfc2	dclo	dmult	floor.l.d
c.olt.d	clo	dclz	dmultu	floor.l.s
c.olt.ps	clz	ddiv	drotr	floor.w.d
c.olt.s	ctc1	ddivu	drotr32	floor.w.s
c.ueq.d	ctc2	deret	drotrv	ins
c.ueq.ps	cvt.d.l	dext	dsbh	j
c.ueq.s	cvt.d.s	dextm	dshd	jal
c.ule.d	cvt.d.w	dextu	dsll	jalr
c.ule.ps	cvt.l.d	di	dsll32	jalr.hb

jalx	mfhi	mul.ps	rsqrt.s	swc2
jr	mflo	mul.s	sb	swl
jr.hb	mov.d	mult	sc	swr
lb	mov.ps	multu	scd	swxc1
lbu	mov.s	neg.d	sd	sync
ld	movf	neg.ps	sdbbp	syscall
ldc1	movf.d	neg.s	sdc1	teq
ldc2	movf.ps	nmadd.d	sdc2	teqi
ldl	movf.s	nmadd.ps	sdl	tge
ldr	movn	nmadd.s	sdr	tgei
ldxc1	movn.d	nmsub.d	sdxc1	tgeiu
lh	movn.ps	nmsub.ps	seb	tgeu
lhu	movn.s	nmsub.s	seh	tlbp
li	movt	nop	sh	tlbr
ll	movt.d	nor	sll	tlbwi
lld	movt.ps	or	sllv	tlbwr
lui	movt.s	ori	slt	tlt
luxc1	movz	pause	slti	tlti
lw	movz.d	pll.ps	sltiu	tltiu
lwc1	movz.ps	plu.ps	sltu	tltu
lwc2	movz.s	pref	sqrt.d	tne
lwl	msub	prefx	sqrt.s	tnei
lwr	msub.d	pul.ps	sra	trunc.l.d
lwu	msub.ps	puu.ps	srav	trunc.l.s
lwxc1	msub.s	rdhdwr	srl	trunc.w.d
madd	msubu	rdpgpr	srlv	trunc.w.s
madd.d	mtc0	recip.d	ssnop	wait
madd.ps	mtc1	recip.s	sub	wrpgpr
madd.s	mtc2	rotr	sub.d	wsbh
maddu	mthc1	rotrv	sub.ps	xor
mfc0	mthc2	round.l.d	sub.s	xori
mfc1	mthi	round.l.s	subu	
mfc2	mtlo	round.w.d	suxc1	
mfhc1	mul	round.w.s	sw	
mfhc2	mul.d	rsqrt.d	swc1	

The actual architecture the compilers and assemblers generate machine code for by default, is indicated by the number in the suffix of their names. The assemblers allow users to temporarily switch between the supported architectures by passing 32 or 64 as operand to the bit mode directive. The default ordering of the binary encoding of instructions is least significant octet first but can be changed using the big-endian mode directive.

15.3 Calling Convention

The machine code generator and runtime support for the MIPS32 and MIPS64 architectures as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

15.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

15.3.2 Floating-Point Support

The MIPS32 and MIPS64 architectures optionally support floating-point operations. The MIPS generator is able to generate native floating-point operations for processors that do support them.

15.3.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to MIPS registers `r2` and `r3` depending on the size of the actual result value. If supported natively, floating-point results are stored in the register `f0`.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to the MIPS register `r29`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to the MIPS register `r30`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is supported and mapped to the MIPS register `r31`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls. Registers holding integer and address values are mapped to the general-purpose registers, while registers holding floating-point values are mapped to the floating-point registers.

15.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the MIPS architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific

runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the MIPS hardware architecture as well as the required combination of linker and object files.

15.4.1 General

Basic architectural runtime support is provided by the object files `mips32run` and `mips64run`. Users should always include one of these object files during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppmips32run` and `cppmips64run` library files respectively. Programs written in Oberon need additional runtime support stored in the `obmips32run` and `obmips64run` library files respectively. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

15.4.2 Linux

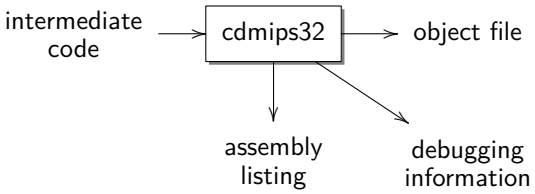
Programs targeting Linux-based operating systems are created using the `linkbin` linker tool. It creates either 32-bit or 64-bit Executable and Linking Format (ELF) files [2] if provided with the runtime support stored in the `mips32linuxrun` and `mips64linuxrun` object files respectively. Calling the `ecsd` utility tool using the `mips32linux` or `mips64linux` target environments achieves the same result. External libraries are available using additional runtime support stored in the following object files: `mips32libdl`, `mips32libpthread`, `mips32-libsdl`, `mips64libdl`, `mips64libpthread`, and `mips64libsdl`.

15.5 MIPS Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the MIPS32 and MIPS64 hardware architectures. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

15.5.1 cdmips32

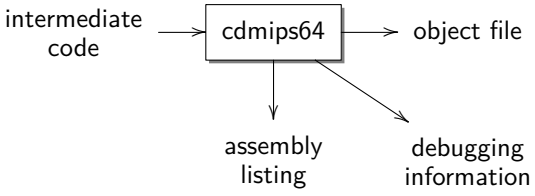
`cdmips32` is a compiler for intermediate code targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

15.5.2 cdmips64

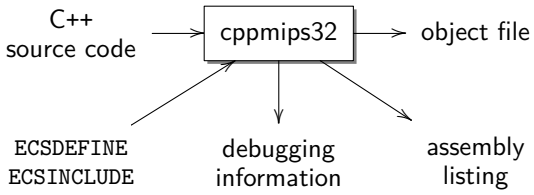
`cdmips64` is a compiler for intermediate code targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

15.5.3 cppmips32

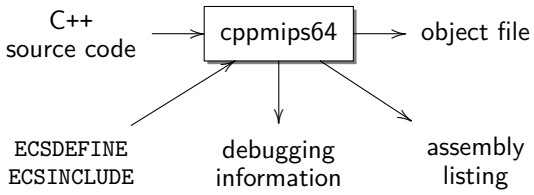
`cppmips32` is a compiler for the C++ programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mips32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmips32run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

15.5.4 cppmips64

`cppmips64` is a compiler for the C++ programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mips64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmips64run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

15.5.5 falmips32

`falmips32` is a compiler for the FALSE programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

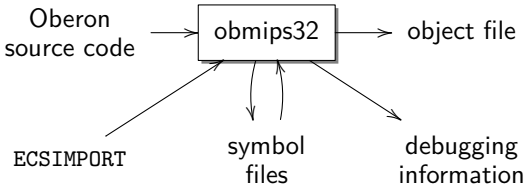
15.5.6 falmips64

`falmips64` is a compiler for the FALSE programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

15.5.7 obmips32

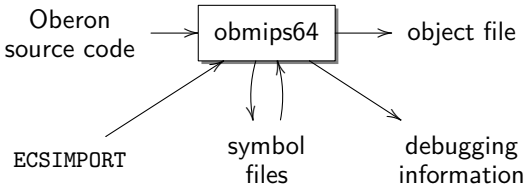
`obmips32` is a compiler for the Oberon programming language targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmips32run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

15.5.8 obmips64

`obmips64` is a compiler for the Oberon programming language targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmips64run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

15.5.9 mips32asm

`mips32asm` is an assembler for the MIPS32 hardware architecture. It translates assembly code into machine code for MIPS32 processors and stores it in corresponding object files.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

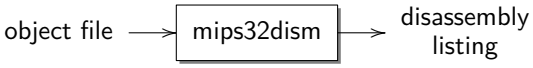
15.5.10 mips64asm

`mips64asm` is an assembler for the MIPS64 hardware architecture. It translates assembly code into machine code for MIPS64 processors and stores it in corresponding object files.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

15.5.11 mips32dism

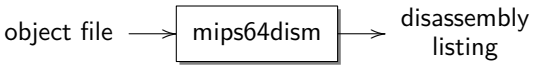
`mips32dism` is a disassembler for the MIPS32 hardware architecture. It translates machine code from object files targeting MIPS32 processors into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

15.5.12 mips64dism

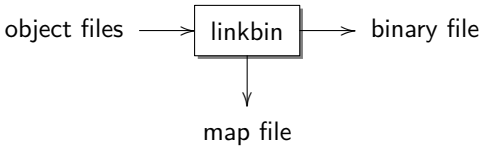
`mips64dism` is a disassembler for the MIPS64 hardware architecture. It translates machine code from object files targeting MIPS64 processors into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

15.5.13 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

16 MMIX

This chapter describes how the Eigen Compiler Suite supports the MMIX hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

16.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the MMIX hardware architecture. Figure 16.1 shows the data flow in-between these tools.

All compilers targeting the MMIX architecture translate their programs using an intermediate code representation. The MMIX generator is able to translate the intermediate code representation of a program into machine code for MMIX processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

16.2 Instruction Set

Tools targeting the MMIX architecture support the instruction set listed in Table 16.1 and use the same assembly syntax as predefined by Donald E. Knuth [21]. The only exception are immediate values which are not prefixed by a number sign. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 16.1: Supported MMIX instruction set (256 instructions)

16addu	8addui	andn	bdifi	bnp
16addui	add	andnh	bev	bnpb
2addu	addi	andni	bevb	bnz
2addui	addu	andnl	bn	bnzb
4addu	addui	andnmh	bnb	bod
4addui	and	andnml	bnn	bodb
8addu	andi	bdif	bnnb	bp

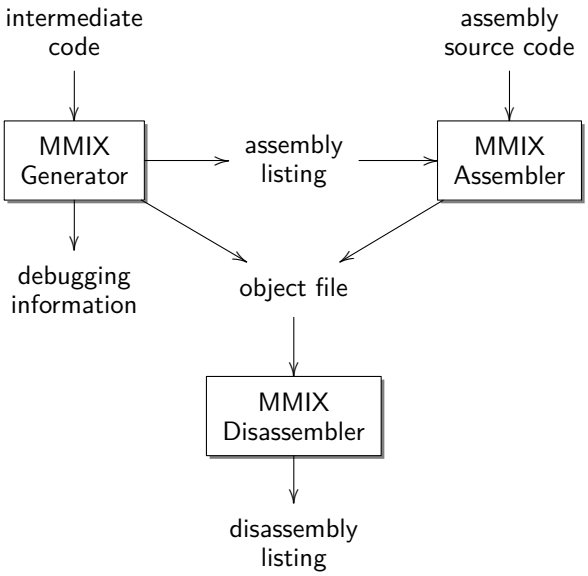


Figure 16.1: Data flow within the tools targeting the MMIX architecture

bpb	cswapi	fsqrt	ldoi	mului
bz	csz	fsub	ldou	mux
bzb	cszi	fun	ldoui	muxi
cmp	div	fune	ldsf	mxor
cmpi	divi	get	ldsfi	mxori
cmpu	divu	geta	ldt	nand
cmpui	divui	getab	ldti	nandi
csev	fadd	go	ldtu	neg
csevi	fcmp	goi	ldtui	negi
csn	fcmpe	inch	ldunc	negu
csni	fdiv	incl	ldunci	negui
csnn	feql	incmh	ldvts	nor
csnni	feqle	incml	ldvtsi	nori
csnp	fint	jmp	ldw	nxor
csnpi	fix	jmpb	ldwi	nxori
csnz	fixu	ldb	ldwu	odif
csnzi	flot	ldbi	ldwui	odifi
csod	floti	ldbu	mor	or
csodi	flotu	ldbui	mori	orh
csp	flotui	ldht	mul	ori
cspi	fmul	ldhti	muli	orl
cswap	frem	ldo	mulu	ormh

```
orml        preldi      slu         sttui       wdifi
orn         prest       slui        stunc       xor
orni        presti      sr          stunci      xori
pbev        pushgo      sri         stw         zsev
pbevb       pushgoi     sru         stwi        zsevi
pbn         pushj       srui        stwu        zsn
pbnb        pushjb      stb         stwui       zsni
pbnn        put         stbi        sub         zsnn
pbnnb       puti        stbu        subi        zsnni
pbnp        resume      stbui       subu        zsnp
pbnpb       sadd        stco        subui       zsnpi
pbnz        saddi       stcoi       swym        zsnz
pbnzb       save        stht        sync        zsnzi
pbod        seth        sthti       syncd       zsod
pbodb       setl        sto         syncdi      zsodi
pbp         setmh       stoi        syncid      zsp
pbpb        setml       stou        syncidi     zspi
pbz         sflot       stoui       tdif        zsz
pbzb        sfloti      stsf        tdifi       zszi
pop         sflotu      stsfi       trap
prego       sflotui     stt         trip
pregoi      sl          stti        unsave
preld       sli         sttu        wdif
```


16.3 Calling Convention

The machine code generator and runtime support for the MMIX architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

16.3.1 Stack Operations

Arguments for functions as well as the return address are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

16.3.2 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to MMIX register `$0`.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to MMIX register `$1`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to MMIX register `$2`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is not supported.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

16.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the MMIX architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the MMIX hardware architecture as well as the required combination of linker and object files.

16.4.1 General

Basic architectural runtime support is provided by the object file `mmixrun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppmmixrun` library file. Programs written in Oberon need additional runtime support stored in the `obmmixrun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

16.4.2 MMIX Simulator

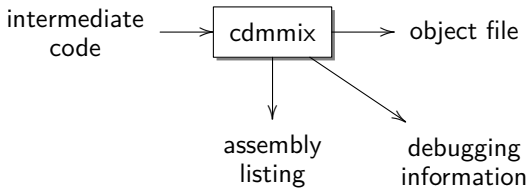
Programs targeting the MMIX simulator are created using the `linkbin` linker tool. It creates a MMIX object file [21] if provided with the runtime support stored in the `mmixsimrun` object file. Calling the `ecsd` utility tool using the `mmixsim` target environment achieves the same result.

16.5 MMIX Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the MMIX hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

16.5.1 cdmmix

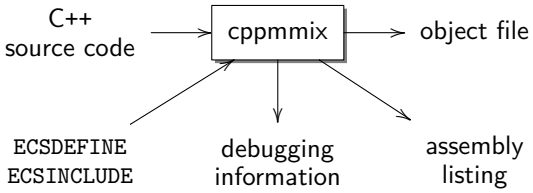
`cdmmix` is a compiler for intermediate code targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

16.5.2 cppmmix

`cppmmix` is a compiler for the C++ programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__mmix__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppmmixrun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

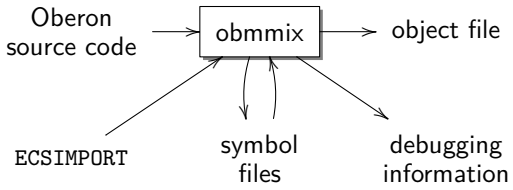
16.5.3 falmmix

`falmmix` is a compiler for the FALSE programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

16.5.4 obmmix

`obmmix` is a compiler for the Oberon programming language targeting the MMIX hardware architecture. It generates machine code for MMIX processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obmmixrun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

16.5.5 mmixasm

`mmixasm` is an assembler for the MMIX hardware architecture. It translates assembly code into machine code for MMIX processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

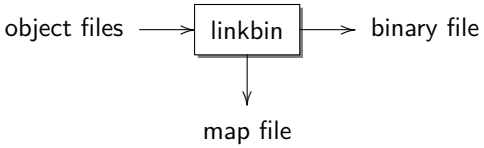
16.5.6 mmixdism

`mmixdism` is a disassembler for the MMIX hardware architecture. It translates machine code from object files targeting MMIX processors into assembly code and writes it to the standard output stream.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

16.5.7 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

17 OpenRISC 1000

This chapter describes how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

17.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the OpenRISC 1000 hardware architecture by OpenCores. Figure 17.1 shows the data flow in-between these tools.

All compilers targeting the OpenRISC 1000 architecture translate their programs using an intermediate code representation. The OpenRISC 1000 generator is able to translate the intermediate code representation of a program into machine code for OpenRISC 1000 processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

17.2 Instruction Set

Tools targeting the OpenRISC 1000 architecture support the instruction set listed in Table 17.1 and use the same assembly syntax as predefined by OpenCores [22]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 17.1: Supported OpenRISC 1000 instruction set (205 instructions)

l.add	l.bnf	l.exths	l.jal	l.lhz
l.addc	l.cmov	l.exthz	l.jalr	l.lwa
l.addi	l.csync	l.extws	l.jr	l.lws
l.addic	l.div	l.extwz	l.lbs	l.lwz
l.and	l.divu	l.ff1	l.lbz	l.mac
l.andi	l.extbs	l.fl1	l.ld	l.maci
l.bf	l.extbz	l.j	l.lhs	l.macrc

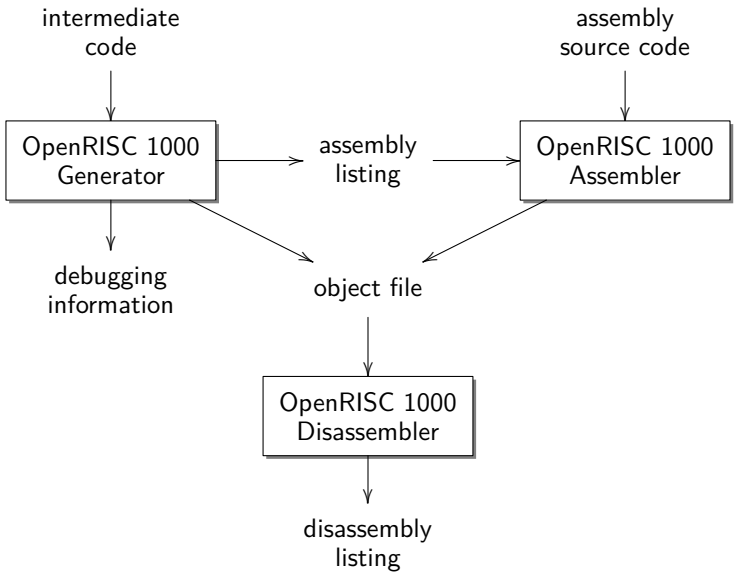


Figure 17.1: Data flow within the tools targeting the OpenRISC 1000 architecture

```
l.macu          l.sfeqi         l.sra           lf.mul.s        lv.addu.h
l.mfspr         l.sfges         l.srai          lf.rem.d        lv.addus.b
l.movhi         l.sfgesi        l.srl           lf.rem.s        lv.addus.h
l.msb           l.sfgeu         l.srli          lf.sfeq.d       lv.all_eq.b
l.msbu          l.sfgeui        l.sub           lf.sfeq.s       lv.all_eq.h
l.msync         l.sfgts         l.sw            lf.sfge.d       lv.all_ge.b
l.mtspr         l.sfgtsi        l.swa           lf.sfge.s       lv.all_ge.h
l.mul           l.sfgtu         l.sys           lf.sfgt.d       lv.all_gt.b
l.muld          l.sfgtui        l.trap          lf.sfgt.s       lv.all_gt.h
l.muldu         l.sfles         l.xor           lf.sfle.d       lv.all_le.b
l.muli          l.sflesi        l.xori          lf.sfle.s       lv.all_le.h
l.mulu          l.sfleu         lf.add.d        lf.sflt.d       lv.all_lt.b
l.nop           l.sfleui        lf.add.s        lf.sflt.s       lv.all_lt.h
l.or            l.sflts         lf.div.d        lf.sfne.d       lv.all_ne.b
l.ori           l.sfltsi        lf.div.s        lf.sfne.s       lv.all_ne.h
l.psync         l.sfltu         lf.ftoi.d       lf.sub.d        lv.and
l.rfe           l.sfltui        lf.ftoi.s       lf.sub.s        lv.any_eq.b
l.ror           l.sfne          lf.itof.d       lv.add.b        lv.any_eq.h
l.rori          l.sfnei         lf.itof.s       lv.add.h        lv.any_ge.b
l.sb            l.sh            lf.madd.d       lv.adds.b       lv.any_ge.h
l.sd            l.sll           lf.madd.s       lv.adds.h       lv.any_gt.b
l.sfeq          l.slli          lf.mul.d        lv.addu.b       lv.any_gt.h
```

lv.any_le.b	lv.cmp_gt.b	lv.merge.h	lv.packus.b	lv.srl.h
lv.any_le.h	lv.cmp_gt.h	lv.min.b	lv.packus.h	lv.sub.b
lv.any_lt.b	lv.cmp_le.b	lv.min.h	lv.perm.n	lv.sub.h
lv.any_lt.h	lv.cmp_le.h	lv.msubs.h	lv.rl.b	lv.subs.b
lv.any_ne.b	lv.cmp_lt.b	lv.muls.h	lv.rl.h	lv.subs.h
lv.any_ne.h	lv.cmp_lt.h	lv.nand	lv.sll	lv.subu.b
lv.avg.b	lv.cmp_ne.b	lv.nor	lv.sll.b	lv.subu.h
lv.avg.h	lv.cmp_ne.h	lv.or	lv.sll.h	lv.subus.b
lv.cmp_eq.b	lv.madds.h	lv.pack.b	lv.sra.b	lv.subus.h
lv.cmp_eq.h	lv.max.b	lv.pack.h	lv.sra.h	lv.unpack.b
lv.cmp_ge.b	lv.max.h	lv.packs.b	lv.srl	lv.unpack.h
lv.cmp_ge.h	lv.merge.b	lv.packs.h	lv.srl.b	lv.xor

17.3 Calling Convention

The machine code generator and runtime support for the OpenRISC 1000 architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

17.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

17.3.2 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to the OpenRISC 1000 registers `r11` and `r12` depending on the size of the actual return type.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to the OpenRISC 1000 register `r1`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to the OpenRISC 1000 register `r2`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is supported and mapped to the OpenRISC 1000 register `r9`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

17.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the OpenRISC 1000 architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the OpenRISC 1000 hardware architecture as well as the required combination of linker and object files.

17.4.1 General

Basic architectural runtime support is provided by the object file `or1krun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppor1krun` library file. Programs written in Oberon need additional runtime support stored in the `obor1krun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

17.4.2 Linux

Programs targeting Linux-based operating systems are created using the `linkbin` linker tool. It creates Executable and Linking Format (ELF) files [2] if provided with the runtime support stored in the `or1klinuxrun` object file. Calling the `ecsd` utility tool using the `or1klinux` target environment achieves the same result. External libraries are available using additional runtime support stored in the following object files: `or1klibdl`, `or1klibpthread`, and `or1klibsdl`.

17.4.3 OpenRISC 1000 Simulator

Programs targeting the OpenRISC 1000 simulator are created using the `linkbin` linker tool. It creates a Common Object File Format (COFF) file [36] if provided with the runtime support

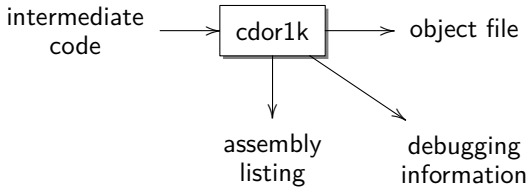
stored in the `or1ksimrun` object file. Calling the `ecsd` utility tool using the `or1ksim` target environment achieves the same result.

17.5 OpenRISC 1000 Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the OpenRISC 1000 hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

17.5.1 cdor1k

`cdor1k` is a compiler for intermediate code targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.

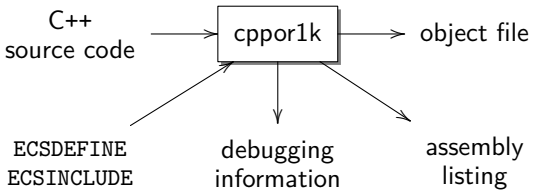


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

17.5.2 cppor1k

`cppor1k` is a compiler for the C++ programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the

generated machine code. The macro `__or1k__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppor1krun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

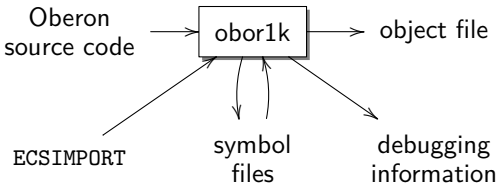
17.5.3 falor1k

`falor1k` is a compiler for the FALSE programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

17.5.4 obor1k

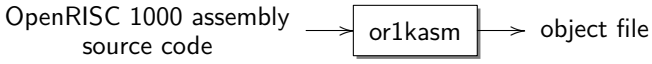
`obor1k` is a compiler for the Oberon programming language targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obor1krun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

17.5.5 or1kasm

`or1kasm` is an assembler for the OpenRISC 1000 hardware architecture. It translates assembly code into machine code for OpenRISC 1000 processors and stores it in corresponding object files.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

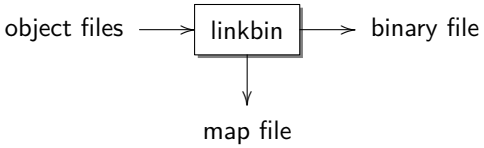
17.5.6 or1kdism

`or1kdism` is a disassembler for the OpenRISC 1000 hardware architecture. It translates machine code from object files targeting OpenRISC 1000 processors into assembly code and writes it to the standard output stream.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

17.5.7 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

18 PowerPC

This chapter describes how the Eigen Compiler Suite supports the PowerPC hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

18.1 Introduction

The Eigen Compiler Suite features various compilers, assemblers, and disassemblers that target the PowerPC hardware architecture by AIM. Figure 18.1 shows the data flow in-between these tools.

All compilers targeting the PowerPC architecture translate their programs using an intermediate code representation. The PowerPC generator is able to translate the intermediate code representation of a program into machine code for PowerPC processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assemblers yielding exactly the same object file. The disassemblers are able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

18.2 Instruction Set

Tools targeting the PowerPC architecture support the instruction set listed in Table 18.1 and use the same assembly syntax as predefined by IBM [23]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 18.1: Supported PowerPC instruction set (439 instructions)

add	addeo	addmeo	and	bc
add.	addeo.	addmeo.	and.	bca
addc	addi	addo	andc	bcctr
addc.	addic	addo.	andc.	bcctrl
addco	addic.	addze	andi.	bcl
addco.	addis	addze.	andis.	bcla
adde	addme	addzeo	b	bclr
adde.	addme.	addzeo.	ba	bclrl

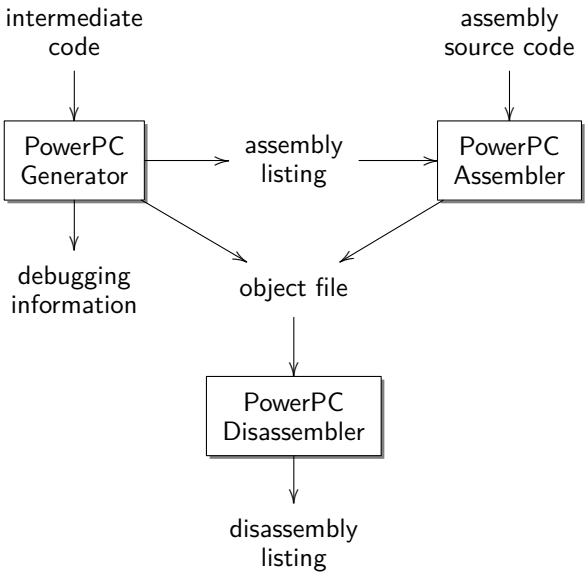


Figure 18.1: Data flow within the tools targeting the PowerPC architecture

```
beq     bne     bsol    crand   divw
beqa    bnea    bsola   crandc  divw.
beql    bnel    bun     creqv   divwo
beqla   bnela   buna    crnand  divwo.
bge     bng     bunl    crnor   divwu
bgea    bnga    bunla   cror    divwu.
bgel    bngl    cmp     crorc   divwuo
bgela   bngla   cmpd    crxor   divwuo.
bgt     bnl     cmpdi   dcbf    eciwx
bgta    bnla    cmpi    dcbi    ecowx
bgtl    bnll    cmpl    dcbst   eieio
bgtla   bnlla   cmpld   dcbt    eqv
bl      bns     cmpldi  dcbtst  eqv.
bla     bnsa    cmpli   dcbz    extsb
ble     bnsl    cmplw   divd    extsb.
blea    bnsla   cmplwi  divd.   extsh
blel    bnu     cmpw    divdo   extsh.
blela   bnua    cmpwi   divdo.  extsw
blt     bnul    cntlzd  divdu   extsw.
blta    bnula   cntlzd. divdu.  fabs
bltl    bso     cntlzw  divduo  fabs.
bltla   bsoa    cntlzw. divduo. fadd
```

```
fadd.       frsqrte.    lwz         mullwo      srad.
fadds       fsel        lwzu        mullwo.     sradi
fadds.      fsel.       lwzux       nand        sradi.
fcfid       fsqrt       lwzx        nand.       sraw
fcfid.      fsqrt.      mcrf        neg         sraw.
fcmpo       fsqrts      mcrfs       neg.        srawi
fcmpu       fsqrts.     mcrxr       nego        srawi.
fctid       fsub        mfcr        nego.       srd
fctid.      fsub.       mffs        nop         srd.
fctidz      fsubs       mffs.       nor         srw
fctidz.     fsubs.      mfmsr       nor.        srw.
fctiw       icbi        mfocrf      not         stb
fctiw.      isync       mfspr       not.        stbu
fctiwz      la          mfsr        or          stbux
fctiwz.     lbz         mfsrin      or.         stbx
fdiv        lbzu        mftb        orc         std
fdiv.       lbzux       mr          orc.        stdcx.
fdivs       lbzx        mr.         ori         stdu
fdivs.      ld          mtcrf       oris        stdux
fmadd       ldarx       mtfsb0      rfi         stdx
fmadd.      ldu         mtfsb0.     rfid        stfd
fmadds      ldux        mtfsb1      rldcl       stfdu
fmadds.     ldx         mtfsb1.     rldcl.      stfdux
fmr         lfd         mtfsf       rldcr       stfdx
fmr.        lfdu        mtfsf.      rldcr.      stfiwx
fmsub       lfdux       mtfsfi      rldic       stfs
fmsub.      lfdx        mtfsfi.     rldic.      stfsu
fmsubs      lfs         mtmsr       rldicl      stfsux
fmsubs.     lfsu        mtmsrd      rldicl.     stfsx
fmul        lfsux       mtocrf      rldicr      sth
fmul.       lfsx        mtspr       rldicr.     sthbrx
fmuls       lha         mtsr        rldimi      sthu
fmuls.      lhau        mtsrd       rldimi.     sthux
fnabs       lhaux       mtsrdin     rlwimi      sthx
fnabs.      lhax        mtsrin      rlwimi.     stmw
fneg        lhbrx       mulhd       rlwinm      stswi
fneg.       lhz         mulhd.      rlwinm.     stswx
fnmadd      lhzu        mulhdu      rlwnm       stw
fnmadd.     lhzux       mulhdu.     rlwnm.      stwbrx
fnmadds     lhzx        mulhw       sc          stwcx.
fnmadds.    li          mulhw.      slbia       stwu
fnmsub      lis         mulhwu      slbie       stwux
fnmsub.     lmw         mulhwu.     slbmfee     stwx
fnmsubs     lswi        mulld       slbmfev     sub
fnmsubs.    lswx        mulld.      slbmte      sub.
fres        lwa         mulldo      sld         subc
fres.       lwarx       mulldo.     sld.        subc.
frsp        lwaux       mulli       slw         subco
frsp.       lwax        mullw       slw.        subco.
frsqrte     lwbrx       mullw.      srad        subf
```

subf.	subfeo.	subfze	subo	tlbsync
subfc	subfic	subfze.	subo.	tw
subfc.	subfme	subfzeo	sync	twi
subfco	subfme.	subfzeo.	td	xor
subfco.	subfmeo	subi	tdi	xor.
subfe	subfmeo.	subic	tlbia	xori
subfe.	subfo	subic.	tlbie	xoris
subfeo	subfo.	subis	tlbiel	

The actual operating mode the compilers and assemblers generate machine code for by default, is indicated by the number in the suffix of their names. The assemblers allow users to temporarily switch between the supported operating modes by passing 32 or 64 as operand to the bit mode directive.

18.3 Calling Convention

The machine code generator and runtime support for the PowerPC architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

18.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

18.3.2 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to PowerPC registers `r2` and `r3` depending on the size of the actual result value. Floating-point results are stored in the register `f0`.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to the PowerPC register `r29`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to the PowerPC register `r30`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is not supported.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls. Registers holding integer and address values are mapped to the general-purpose registers, while registers holding floating-point values are mapped to the floating-point registers.

18.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the PowerPC architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the PowerPC hardware architecture as well as the required combination of linker and object files.

18.4.1 General

Basic architectural runtime support is provided by the object files `ppc32run` and `ppc64run`. Users should always include one of these object files during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

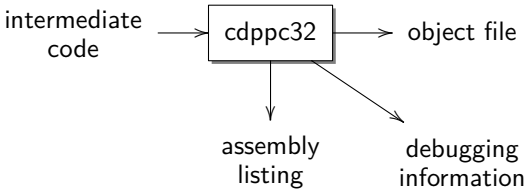
Programs written in C++ need additional runtime support stored in the `cppppc32run` and `cppppc64run` library files respectively. Programs written in Oberon need additional runtime support stored in the `obppc32run` and `obppc64run` library files respectively. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

18.5 PowerPC Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the PowerPC hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

18.5.1 cdppc32

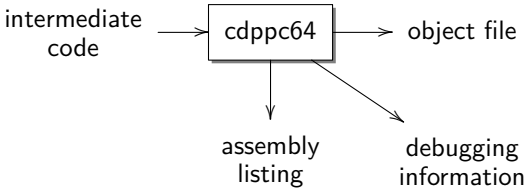
cdppc32 is a compiler for intermediate code targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

18.5.2 cdppc64

cdppc64 is a compiler for intermediate code targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.

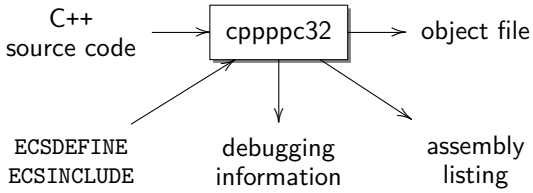


For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information

about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

18.5.3 cppppc32

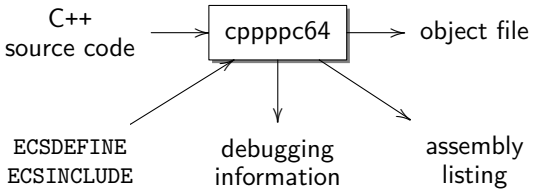
`cppppc32` is a compiler for the C++ programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__ppc32__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppppc32run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

18.5.4 cppppc64

`cppppc64` is a compiler for the C++ programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in C++ and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__ppc64__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppppc64run` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

18.5.5 falppc32

`falppc32` is a compiler for the FALSE programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

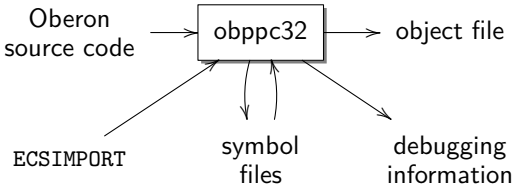
18.5.6 falppc64

`falppc64` is a compiler for the FALSE programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in FALSE and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

18.5.7 obppc32

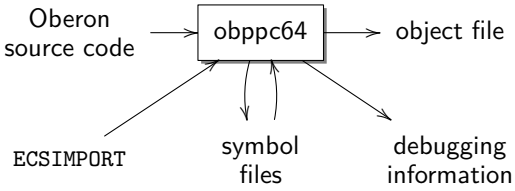
`obppc32` is a compiler for the Oberon programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obppc32run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

18.5.8 obppc64

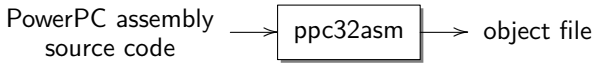
`obppc64` is a compiler for the Oberon programming language targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from modules written in Oberon and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obppc64run` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

18.5.9 ppc32asm

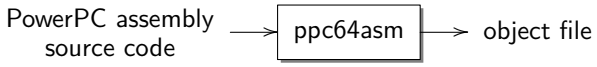
`ppc32asm` is an assembler for the PowerPC hardware architecture. It translates assembly code into machine code for PowerPC processors and stores it in corresponding object files. By default, the assembler generates machine code for the 32-bit operating mode defined by the PowerPC architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

18.5.10 ppc64asm

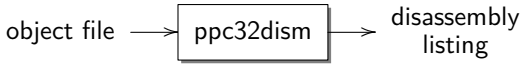
`ppc64asm` is an assembler for the PowerPC hardware architecture. It translates assembly code into machine code for PowerPC processors and stores it in corresponding object files. By default, the assembler generates machine code for the 64-bit operating mode defined by the PowerPC architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

18.5.11 ppc32dism

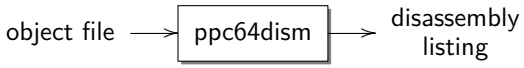
`ppc32dism` is a disassembler for the PowerPC hardware architecture. It translates machine code from object files targeting PowerPC processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 32-bit operating mode defined by the PowerPC architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

18.5.12 ppc64dism

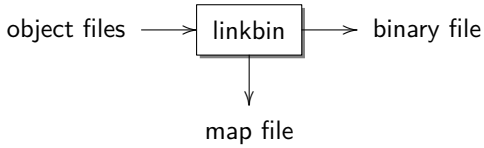
`ppc64dism` is a disassembler for the PowerPC hardware architecture. It translates machine code from object files targeting PowerPC processors into assembly code and writes it to the standard output stream. It assumes that the machine code was generated for the 64-bit operating mode defined by the PowerPC architecture.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

18.5.13 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

19 RISC

This chapter describes how the Eigen Compiler Suite supports the RISC hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

19.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the RISC hardware architecture. Figure 19.1 shows the data flow in-between these tools.

All compilers targeting the RISC architecture translate their programs using an intermediate code representation. The RISC generator is able to translate the intermediate code representation of a program into machine code for RISC processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

19.2 Instruction Set

Tools targeting the RISC architecture support the instruction set listed in Table 19.1 and use the same assembly syntax as predefined by Niklaus Wirth [24]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 19.1: Supported RISC instruction set (60 instructions)

adc	bge	blgt	blt	div
add	bgt	blhi	blvc	fad
and	bhi	blle	blvs	fdv
ann	bl	blls	bmi	fml
asr	blcc	bllt	bne	fsb
b	blcs	blmi	bpl	ior
bcc	ble	blne	bvc	ld
bcs	bleq	blpl	bvs	ldb
beq	blge	bls	cli	lsl

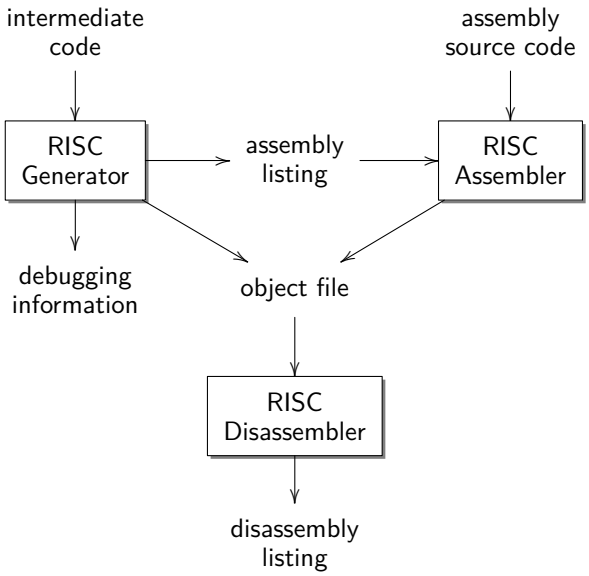


Figure 19.1: Data flow within the tools targeting the RISC architecture

`mlu`	`mvf`	`nop`	`sbc`	`sti`
`mov`	`mvh`	`ror`	`st`	`sub`
`mul`	`mvs`	`rti`	`stb`	`xor`

19.3 Calling Convention

The machine code generator and runtime support for the RISC architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

19.3.1 Stack Operations

Arguments for functions as well as the return address are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

19.3.2 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to RISC registers `r0` and `r1` depending on the size of the actual return type.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to RISC register `r14`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to RISC register `r13`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is mapped to RISC register `r15`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

19.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the RISC architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the RISC hardware architecture as well as the required combination of linker and object files.

19.4.1 General

Basic architectural runtime support is provided by the object file `riscrun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppriscrun` library file. Programs written in Oberon need additional runtime support stored in the `obriscrun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

19.4.2 RISC Microcontrollers

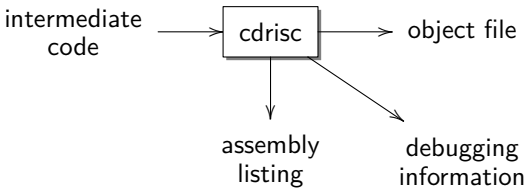
Programs targeting RISC microcontrollers are created using the `linkbin` linker tool. It creates a bootloader that can be stored in RISC disk images at block `0x80004` if provided with the runtime support stored in the `riscdiskrun` object file. Calling the `ecsd` utility tool using the `riscdisk` target environment achieves the same result.

19.5 RISC Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the RISC hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

19.5.1 cdrisc

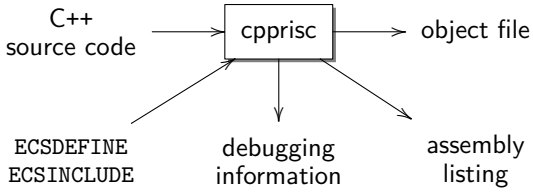
`cdrisc` is a compiler for intermediate code targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

19.5.2 cpprisc

`cpprisc` is a compiler for the C++ programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__risc__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppriscrun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

19.5.3 falrisc

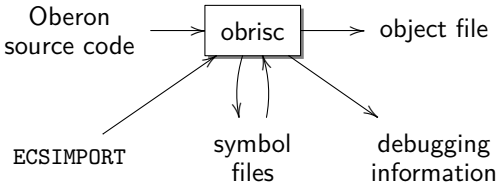
`falrisc` is a compiler for the FALSE programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

19.5.4 obrisc

`obrisc` is a compiler for the Oberon programming language targeting the RISC hardware architecture. It generates machine code for RISC processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging

information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obriscrun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

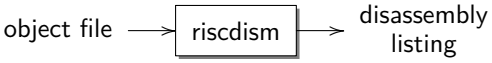
19.5.5 riscasm

`riscasm` is an assembler for the RISC hardware architecture. It translates assembly code into machine code for RISC processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.

For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

19.5.6 riscdism

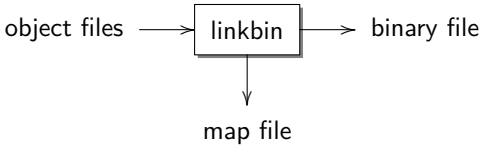
`riscdism` is a disassembler for the RISC hardware architecture. It translates machine code from object files targeting RISC processors into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

19.5.7 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

20 WebAssembly

This chapter describes how the Eigen Compiler Suite supports the WebAssembly architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

20.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the WebAssembly architecture. Figure 20.1 shows the data flow in-between these tools.

All compilers targeting the WebAssembly architecture translate their programs using an intermediate code representation. The WebAssembly generator is able to translate the intermediate code representation of a program into machine code for WebAssembly targets. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

20.2 Instruction Set

Tools targeting the WebAssembly architecture support the instruction set listed in Table 20.1 and use the same assembly syntax as predefined by the World Wide Web Consortium (W3C) [25]. The only exception are the addition of the pseudo instructions `i32`, `label`, `lane`, `s32`, `u32`, and `valtype`. They encode an immediate value of the respective type with a fixed size and can be used as a replacement for operands of instructions that require immediate values of that type. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 20.1: Supported WebAssembly instruction set (443 instructions)

`block`	`data.drop`
`br`	`drop`
`br_if`	`elem.drop`
`br_table`	`else`
`call`	`end`
`call_indirect`	`f32.abs`

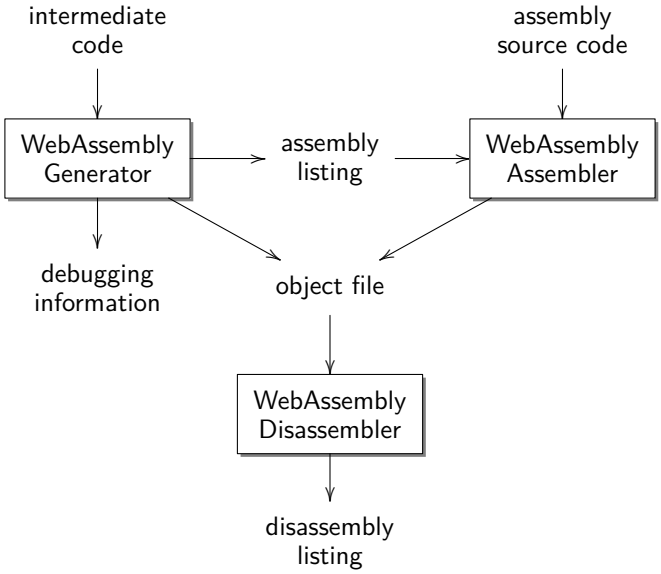


Figure 20.1: Data flow within the tools targeting the WebAssembly architecture

```
f32.add
f32.ceil
f32.const
f32.convert_i32_s
f32.convert_i32_u
f32.convert_i64_s
f32.convert_i64_u
f32.copysign
f32.demote_f64
f32.div
f32.eq
f32.floor
f32.ge
f32.gt
f32.le
f32.load
f32.lt
f32.max
f32.min
f32.mul
f32.ne
f32.nearest
f32.neg
f32.reinterpret_i32
f32.sqrt
f32.store
f32.sub
f32.trunc
f32x4.abs
f32x4.add
f32x4.ceil
f32x4.convert_i32x4_s
f32x4.convert_i32x4_u
f32x4.demote_f64x2_zero
f32x4.div
f32x4.eq
f32x4.extract_lane
f32x4.floor
f32x4.ge
f32x4.gt
f32x4.le
f32x4.lt
f32x4.max
f32x4.min
```

```
f32x4.mul
f32x4.ne
f32x4.nearest
f32x4.neg
f32x4.pmax
f32x4.pmin
f32x4.replace_lane
f32x4.splat
f32x4.sqrt
f32x4.sub
f32x4.trunc
f64.abs
f64.add
f64.ceil
f64.const
f64.convert_i32_s
f64.convert_i32_u
f64.convert_i64_s
f64.convert_i64_u
f64.copysign
f64.div
f64.eq
f64.floor
f64.ge
f64.gt
f64.le
f64.load
f64.lt
f64.max
f64.min
f64.mul
f64.ne
f64.nearest
f64.neg
f64.promote_f32
f64.reinterpret_i64
f64.sqrt
f64.store
f64.sub
f64.trunc
f64x2.abs
f64x2.add
f64x2.ceil
f64x2.convert_low_i32x4_s
f64x2.convert_low_i32x4_u
f64x2.div
f64x2.eq
f64x2.extract_lane
f64x2.floor
f64x2.ge
f64x2.gt
f64x2.le
f64x2.lt
f64x2.max
f64x2.min
f64x2.mul
f64x2.ne
f64x2.nearest
f64x2.neg
f64x2.pmax
f64x2.pmin
f64x2.promote_low_f32x4
f64x2.replace_lane
f64x2.splat
f64x2.sqrt
f64x2.sub
f64x2.trunc
global.get
global.set
i16x8.abs
i16x8.add
i16x8.add_sat_s
i16x8.add_sat_u
i16x8.all_true
i16x8.avgr_u
i16x8.bitmask
i16x8.eq
i16x8.extadd_pairwise_i8x16_s
i16x8.extadd_pairwise_i8x16_u
i16x8.extend_high_i8x16_s
i16x8.extend_high_i8x16_u
i16x8.extend_low_i8x16_s
i16x8.extend_low_i8x16_u
i16x8.extmul_high_i8x16_s
i16x8.extmul_high_i8x16_u
i16x8.extmul_low_i8x16_s
i16x8.extmul_low_i8x16_u
i16x8.extract_lane_s
i16x8.extract_lane_u
i16x8.ge_s
i16x8.ge_u
i16x8.gt_s
i16x8.gt_u
i16x8.le_s
i16x8.le_u
i16x8.lt_s
i16x8.lt_u
i16x8.max_s
i16x8.max_u
i16x8.min_s
```

```
i16x8.min_u
i16x8.mul
i16x8.narrow_i32x4_s
i16x8.narrow_i32x4_u
i16x8.ne
i16x8.neg
i16x8.q15mulr_sat_s
i16x8.replace_lane
i16x8.shl
i16x8.shr_s
i16x8.shr_u
i16x8.splat
i16x8.sub
i16x8.sub_sat_s
i16x8.sub_sat_u
i32
i32.add
i32.and
i32.clz
i32.const
i32.ctz
i32.div_s
i32.div_u
i32.eq
i32.eqz
i32.extend16_s
i32.extend8_s
i32.ge_s
i32.ge_u
i32.gt_s
i32.gt_u
i32.le_s
i32.le_u
i32.load
i32.load16_s
i32.load16_u
i32.load8_s
i32.load8_u
i32.lt_s
i32.lt_u
i32.mul
i32.ne
i32.or
i32.popcnt
i32.reinterpret_f32
i32.rem_s
i32.rem_u
i32.rotl
i32.rotr
i32.shl
i32.shr_s
i32.shr_u
i32.store
i32.store16
i32.store8
i32.sub
i32.trunc_f32_s
i32.trunc_f32_u
i32.trunc_f64_s
i32.trunc_f64_u
i32.trunc_sat_f32_s
i32.trunc_sat_f32_u
i32.trunc_sat_f64_s
i32.trunc_sat_f64_u
i32.wrap_i64
i32.xor
i32x4.abs
i32x4.add
i32x4.all_true
i32x4.bitmask
i32x4.dot_i16x8_s
i32x4.eq
i32x4.extadd_pairwise_i16x8_s
i32x4.extadd_pairwise_i16x8_u
i32x4.extend_high_i16x8_s
i32x4.extend_high_i16x8_u
i32x4.extend_low_i16x8_s
i32x4.extend_low_i16x8_u
i32x4.extmul_high_i16x8_s
i32x4.extmul_high_i16x8_u
i32x4.extmul_low_i16x8_s
i32x4.extmul_low_i16x8_u
i32x4.extract_lane
i32x4.ge_s
i32x4.ge_u
i32x4.gt_s
i32x4.gt_u
i32x4.le_s
i32x4.le_u
i32x4.lt_s
i32x4.lt_u
i32x4.max_s
i32x4.max_u
i32x4.min_s
i32x4.min_u
i32x4.mul
i32x4.ne
i32x4.neg
i32x4.replace_lane
i32x4.shl
```

```
i32x4.shr_s
i32x4.shr_u
i32x4.splat
i32x4.sub
i32x4.trunc_sat_f32x4_s
i32x4.trunc_sat_f32x4_u
i32x4.trunc_sat_f64x2_s_zero
i32x4.trunc_sat_f64x2_u_zero
i64.add
i64.and
i64.clz
i64.const
i64.ctz
i64.div_s
i64.div_u
i64.eq
i64.eqz
i64.extend16_s
i64.extend32_s
i64.extend8_s
i64.extend_i32_s
i64.extend_i32_u
i64.ge_s
i64.ge_u
i64.gt_s
i64.gt_u
i64.le_s
i64.le_u
i64.load
i64.load16_s
i64.load16_u
i64.load32_s
i64.load32_u
i64.load8_s
i64.load8_u
i64.lt_s
i64.lt_u
i64.mul
i64.ne
i64.or
i64.popcnt
i64.reinterpret_f64
i64.rem_s
i64.rem_u
i64.rotl
i64.rotr
i64.shl
i64.shr_s
i64.shr_u
i64.store
i64.store16
i64.store32
i64.store8
i64.sub
i64.trunc_f32_s
i64.trunc_f32_u
i64.trunc_f64_s
i64.trunc_f64_u
i64.trunc_sat_f32_s
i64.trunc_sat_f32_u
i64.trunc_sat_f64_s
i64.trunc_sat_f64_u
i64.xor
i64x2.abs
i64x2.add
i64x2.all_true
i64x2.bitmask
i64x2.eq
i64x2.extend_high_i32x4_s
i64x2.extend_high_i32x4_u
i64x2.extend_low_i32x4_s
i64x2.extend_low_i32x4_u
i64x2.extmul_high_i32x4_s
i64x2.extmul_high_i32x4_u
i64x2.extmul_low_i32x4_s
i64x2.extmul_low_i32x4_u
i64x2.extract_lane
i64x2.ge_s
i64x2.gt_s
i64x2.le_s
i64x2.lt_s
i64x2.mul
i64x2.ne
i64x2.neg
i64x2.replace_lane
i64x2.shl
i64x2.shr_s
i64x2.shr_u
i64x2.splat
i64x2.sub
i8x16.abs
i8x16.add
i8x16.add_sat_s
i8x16.add_sat_u
i8x16.all_true
i8x16.avgr_u
i8x16.bitmask
i8x16.eq
i8x16.extract_lane_s
i8x16.extract_lane_u
```

```
i8x16.ge_s
i8x16.ge_u
i8x16.gt_s
i8x16.gt_u
i8x16.le_s
i8x16.le_u
i8x16.lt_s
i8x16.lt_u
i8x16.max_s
i8x16.max_u
i8x16.min_s
i8x16.min_u
i8x16.narrow_i16x8_s
i8x16.narrow_i16x8_u
i8x16.ne
i8x16.neg
i8x16.popcnt
i8x16.replace_lane
i8x16.shl
i8x16.shr_s
i8x16.shr_u
i8x16.shuffle
i8x16.splat
i8x16.sub
i8x16.sub_sat_s
i8x16.sub_sat_u
i8x16.swizzle
if
label
lane
local.get
local.set
local.tee
loop
memory.copy
memory.fill
memory.grow
memory.init
memory.size
nop
ref.func
ref.is_null
ref.null
return
s32
select
table.copy
table.fill
table.get
table.grow
table.init
table.set
table.size
u32
unreachable
v128.and
v128.andnot
v128.any_true
v128.bitselect
v128.const
v128.load
v128.load16_lane
v128.load16_splat
v128.load16x4_s
v128.load16x4_u
v128.load32_lane
v128.load32_splat
v128.load32_zero
v128.load32x2_s
v128.load32x2_u
v128.load64_lane
v128.load64_splat
v128.load64_zero
v128.load8_lane
v128.load8_splat
v128.load8x8_s
v128.load8x8_u
v128.not
v128.or
v128.store
v128.store16_lane
v128.store32_lane
v128.store64_lane
v128.store8_lane
v128.xor
valtype
vec
```

20.3 Calling Convention

The machine code generator and runtime support for the WebAssembly architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

20.3.1 Stack Operations

Arguments for functions are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

20.3.2 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to mutable WebAssembly globals in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to WebAssembly globals called `_$res_i32`, `_$res_i64`, `_$res_f32`, and `_$res_f64` with the respective types.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to a WebAssembly global called `_$sp` with type `i32`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to a WebAssembly global called `_$fp` with type `i32`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` as well as return addresses are not supported.

All other intermediate code registers are mapped as needed to WebAssembly locals. Their contents and mapping are therefore not considered volatile across function calls.

20.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the WebAssembly architecture and runtime environments based on this architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime

environments based on the WebAssembly architecture as well as the required combination of linker and object files.

20.4.1 General

Basic architectural runtime support is provided by the object file `wasmrun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same architecture.

Programs written in C++ need additional runtime support stored in the `cppwasmrun` library file. Programs written in Oberon need additional runtime support stored in the `obwasmrun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

20.4.2 WebAssembly Environments

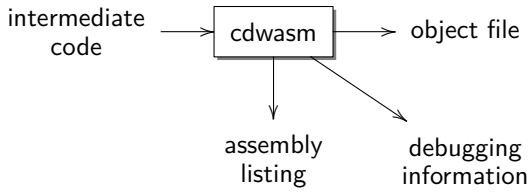
Programs targeting web environments are created using the `linkbin` linker tool. It creates a WebAssembly module [25] if provided with the runtime support stored in the `webassemblyrun` object file. Calling the `ecsd` utility tool using the `webassembly` target environment achieves the same result.

20.5 WebAssembly Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the WebAssembly architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

20.5.1 cdwasm

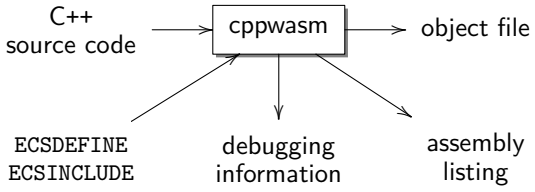
`cdwasm` is a compiler for intermediate code targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

20.5.2 cppwasm

`cppwasm` is a compiler for the C++ programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__wasm__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppwasmrun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

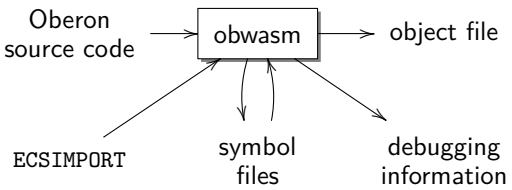
20.5.3 falwasm

`falwasm` is a compiler for the FALSE programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

20.5.4 obwasm

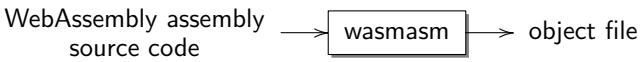
`obwasm` is a compiler for the Oberon programming language targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obwasmrun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

20.5.5 wasmasm

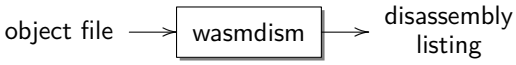
`wasmasm` is an assembler for the WebAssembly architecture. It translates assembly code into machine code for WebAssembly targets and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

20.5.6 wasmdism

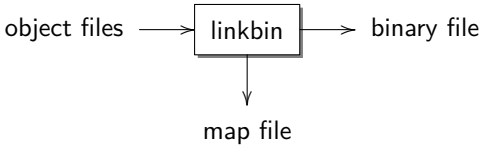
`wasmdism` is a disassembler for the WebAssembly architecture. It translates machine code from object files targeting WebAssembly targets into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

20.5.7 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

21 Xtensa

This chapter describes how the Eigen Compiler Suite supports the Xtensa hardware architecture. This includes information about the assembler, disassembler, and the various compilers featured by the Eigen Compiler Suite as well as the interoperability between these tools.

21.1 Introduction

The Eigen Compiler Suite features various compilers, an assembler, and a disassembler that target the Xtensa hardware architecture by Cadence Design Systems. Figure 21.1 shows the data flow in-between these tools.

All compilers targeting the Xtensa architecture translate their programs using an intermediate code representation. The Xtensa generator is able to translate the intermediate code representation of a program into machine code for Xtensa processors. It stores the resulting binary code and data in so-called object files. Additionally, the generator is able to create an assembly code listing of the machine code for debugging purposes. This assembly code listing can also be processed by the assembler yielding exactly the same object file. The disassembler is able to open object files and print a human-readable disassembly listing of their contents. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23.

21.2 Instruction Set

Tools targeting the Xtensa architecture support the instruction set listed in Table 21.1 and use the same assembly syntax as predefined by Cadence Design Systems [26]. For more information about the generic assembly language and how to use it, see Chapter 8.

Table 21.1: Supported Xtensa instruction set (403 instructions)

abs	addexp.s	addx8	ball
abs.d	addexpm.d	all4	bany
abs.s	addexpm.s	all8	bbc
add	addi	and	bbci
add.d	addi.n	andb	bbci.l
add.n	addmi	andbc	bbs
add.s	addx2	any4	bbsi
addexp.d	addx4	any8	bbsi.l

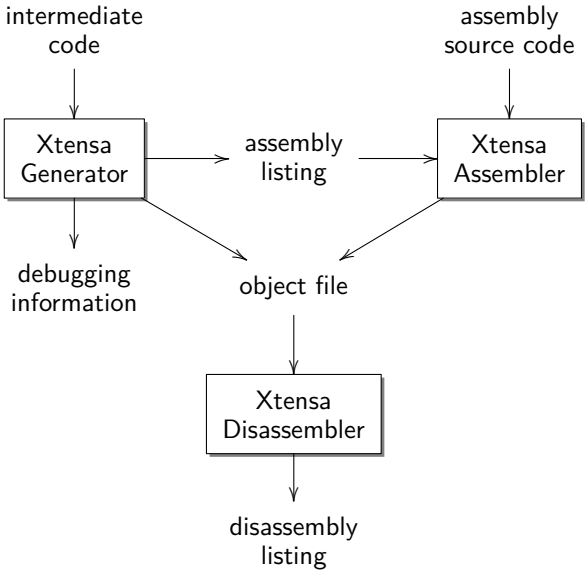


Figure 21.1: Data flow within the tools targeting the Xtensa architecture

```
beq             break.n         depbits         dpfr.bf
beqi            bt              dhi             dpfro
beqz            call0           dhi.b           dpfw
beqz.n          call12          dhu             dpfw.b
bf              call4           dhwb            dpfw.bf
bge             call8           dhwb.b          dpfwo
bgei            callx0          dhwbi           dsync
bgeu            callx12         dhwbi.b         entry
bgeui           callx4          dii             esync
bgez            callx8          diu             excw
blt             ceil.d          div0.d          extui
blti            ceil.s          div0.s          extw
bltu            clamps          divn.d          float.d
bltui           clrex           divn.s          float.s
bltz            const.d         diwb            floor.d
bnall           const.s         diwbi           floor.s
bne             const16         diwbui.p        fsync
bnei            cvtd.s          dpfl            getex
bnez            cvts.d          dpfm.b          idtlb
bnez.n          dci             dpfm.bf         ihi
bnone           dcwb            dpfr            ihu
break           dcwbi           dpfr.b          iii
```

```
iitlb
iiu
ill
ill.n
ipf
ipfl
isync
j
j.l
jx
l16si
l16ui
l32ai
l32e
l32ex
l32i
l32i.n
l32r
l8ui
ldct
ldcw
lddec
lddr32.p
ldi
ldinc
ldip
ldx
ldxp
loop
loopgtz
loopnez
lsi
lsip
lsiu
lsx
lsxp
lsxu
madd.d
madd.s
maddn.d
maddn.s
max
maxu
memw
min
minu
mkdadj.d
mkdadj.s
mksadj.d
mksadj.s
mov
mov.d
mov.n
mov.s
moveqz
moveqz.d
moveqz.s
movf
movf.d
movf.s
movgez
movgez.d
movgez.s
movi
movi.n
movltz
movltz.d
movltz.s
movnez
movnez.d
movnez.s
movsp
movt
movt.d
movt.s
msub.d
msub.s
mul.aa.hh
mul.aa.hl
mul.aa.lh
mul.aa.ll
mul.ad.hh
mul.ad.hl
mul.ad.lh
mul.ad.ll
mul.d
mul.da.hh
mul.da.hl
mul.da.lh
mul.da.ll
mul.dd.hh
mul.dd.hl
mul.dd.lh
mul.dd.ll
mul.s
mul16s
mul16u
mula.aa.hh
mula.aa.hl
mula.aa.lh
mula.aa.ll
mula.ad.hh
mula.ad.hl
mula.ad.lh
mula.ad.ll
mula.da.hh
mula.da.hh.lddec
mula.da.hh.ldinc
mula.da.hl
mula.da.hl.lddec
mula.da.hl.ldinc
mula.da.lh
mula.da.lh.lddec
mula.da.lh.ldinc
mula.da.ll
mula.da.ll.lddec
mula.da.ll.ldinc
mula.dd.hh
mula.dd.hh.lddec
mula.dd.hh.ldinc
mula.dd.hl
mula.dd.hl.lddec
mula.dd.hl.ldinc
mula.dd.lh
mula.dd.lh.lddec
mula.dd.lh.ldinc
mula.dd.ll
mula.dd.ll.lddec
mula.dd.ll.ldinc
mull
muls.aa.hh
muls.aa.hl
muls.aa.lh
muls.aa.ll
muls.ad.hh
muls.ad.hl
muls.ad.lh
muls.ad.ll
muls.da.hh
muls.da.hl
muls.da.lh
muls.da.ll
muls.dd.hh
muls.dd.hl
muls.dd.lh
muls.dd.ll
mulsh
muluh
neg
neg.d
neg.s
nexp01.d
nexp01.s
nop
nop.n
nsa
nsau
oeq.d
oeq.s
ole.d
ole.s
olt.d
olt.s
or
orb
orbc
pdtlb
pitlb
pptlb
quos
quou
rdtlb0
rdtlb1
recip0.d
recip0.s
rems
remu
rer
ret
ret.n
retw
retw.n
rfdd
rfde
rfdo
rfe
rfi
rfme
rfr
rfrd
rfue
rfwo
rfwu
ritlb0
ritlb1
rotw
round.d
round.s
rptlb0
rptlb1
```

```
rsil            sdxp            ssr             umul.aa.hh
rsqrt0.d        sext            ssx             umul.aa.hl
rsqrt0.s        sict            ssxp            umul.aa.lh
rsr             sicw            ssxu            umul.aa.ll
rsync           simcall         sub             un.d
rur             sll             sub.d           un.s
s16i            slli            sub.s           utrunc.d
s32c1i          sqrt0.d         subx2           utrunc.s
s32e            sqrt0.s         subx4           waiti
s32ex           sra             subx8           wdtlb
s32i            srai            syscall         wer
s32i.n          src             trunc.d         wfr
s32nb           srl             trunc.s         wfrd
s32ri           srli            ueq.d           witlb
s8i             ssa8b           ueq.s           wptlb
salt            ssa8l           ufloat.d        wsr
saltu           ssai            ufloat.s        wur
sddr32.p        ssi             ule.d           xor
sdi             ssip            ule.s           xorb
sdip            ssiu            ult.d           xsr
sdx             ssl             ult.s
```

21.3 Calling Convention

The machine code generator and runtime support for the Xtensa architecture as provided by the Eigen Compiler Suite use the following calling convention in order to enable interoperability.

21.3.1 Stack Operations

Arguments for functions as well as the return address are in general passed using the stack according to the intermediate code specification. See Chapter 23 for more information about the role of the stack. Function arguments are pushed on the stack in reverse order and cleaned by the caller.

21.3.2 Floating-Point Support

The Xtensa architecture optionally supports floating-point operations. The generator is able to generate native floating-point operations for processors that do support them.

21.3.3 Register Mapping

The special-purpose registers defined by the intermediate code representation are mapped to their corresponding physical registers in the following way:

- Result Register `$res`

  The intermediate code result register `$res` is mapped to Xtensa registers `a2`. 64-bit wide results are stored in registers `a2` and `a3`. If supported natively, floating-point results are stored in register `f0`.

- Stack Pointer Register `$sp`

  The intermediate code stack pointer register `$sp` is mapped to Xtensa register `a1`.

- Frame Pointer Register `$fp`

  The intermediate code frame pointer register `$fp` is mapped to Xtensa register `a15`.

- Link Register `$lnk`

  The intermediate code link register `$lnk` is supported and mapped to Xtensa register `a0`.

All other intermediate code registers are mapped as needed to the remaining physical registers. Their contents and mapping are therefore considered volatile across function calls.

21.4 Runtime Support

The Eigen Compiler Suite provides runtime support for the Xtensa architecture and runtime environments based on this hardware architecture in object files. Users targeting a specific runtime environment have to use an appropriate linker together with these object files in order create an executable program. This section gives information about all supported runtime environments based on the Xtensa hardware architecture as well as the required combination of linker and object files.

21.4.1 General

Basic architectural runtime support is provided by the object file `xtensarun`. Users should always include this object file during linking regardless of the actual target runtime environment. All other object files given to the linker should target the same hardware architecture.

Programs written in C++ need additional runtime support stored in the `cppxtensarun` library file. Programs written in Oberon need additional runtime support stored in the `ob-xtensarun` library file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

21.4.2 Linux

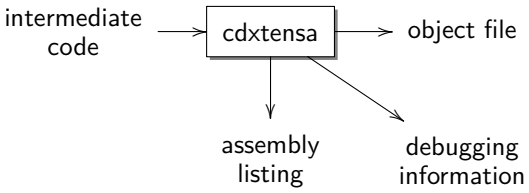
Programs targeting Linux-based operating systems are created using the `linkbin` linker tool. It creates Executable and Linking Format (ELF) files [2] if provided with the runtime support stored in the `xtensalinuxrun` object file. Calling the `ecsd` utility tool using the `xtensalinux` target environment achieves the same result. External libraries are available using additional runtime support stored in the following object files: `xtensalibdl`, `xtensalibpthread`, and `xtensalibsdl`.

21.5 Xtensa Tools

The Eigen Compiler Suite provides the following tools that are able to process object files targeting the Xtensa hardware architecture. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

21.5.1 cdxtensa

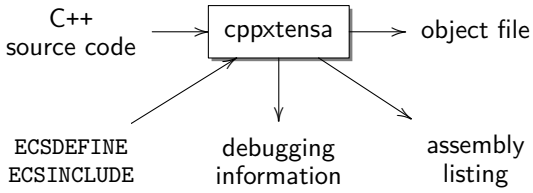
`cdxtensa` is a compiler for intermediate code targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code. This tool is provided for debugging purposes. It allows exposing and modifying an internal data structure that is usually not accessible.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about intermediate code and its purpose, see Chapter 23. For more information about debugging information and its representation, see Chapter 24.

21.5.2 cppxtensa

`cppxtensa` is a compiler for the C++ programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in C++ and stores it in corresponding object files. For debugging purposes, it also creates debugging information files as well as assembly files containing listings of the generated machine code. The macro `__xtensa__` is predefined in order to enable programmers to identify this tool and its target architecture while compiling. Programs generated with this compiler require additional runtime support that is stored in the `cppxtensarun` library file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

21.5.3 falxtensa

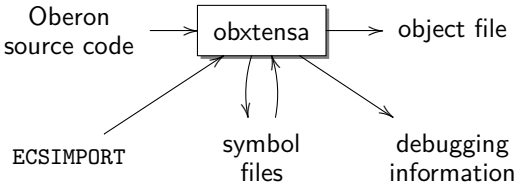
`falxtensa` is a compiler for the FALSE programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in FALSE and stores it in corresponding object files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about object files and their purpose, see Chapter 22.

21.5.4 obxtensa

`obxtensa` is a compiler for the Oberon programming language targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from modules written in Oberon and stores it in corresponding object files. For debugging purposes, it also creates debugging

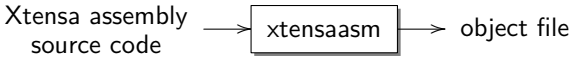
information files as well as assembly files containing listings of the generated machine code. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this compiler require additional runtime support that is stored in the `obxtensarun` library file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

21.5.5 xtensaasm

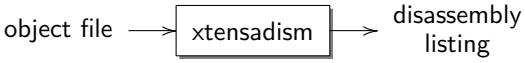
`xtensaasm` is an assembler for the Xtensa hardware architecture. It translates assembly code into machine code for Xtensa processors and stores it in corresponding object files. The names of all special registers are predefined and evaluate to the corresponding number.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

21.5.6 xtensadism

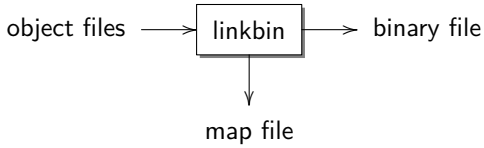
`xtensadism` is a disassembler for the Xtensa hardware architecture. It translates machine code from object files targeting Xtensa processors into assembly code and writes it to the standard output stream.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about object files and their purpose, see Chapter 22.

21.5.7 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



For more information about object files and their purpose, see Chapter 22.

Part IV

Eigen Compiler Suite Internals

22 Object Files

This chapter describes the purpose and the open format of object files which are used by the Eigen Compiler Suite to represent generic binary code and data. It also depicts the functionality and interface of the corresponding linker tools provided by the Eigen Compiler Suite.

The finest eloquence is that which gets things done.

— David Lloyd George

22.1 Introduction

Object files are containers for all binary information that is required to build executable programs. This includes information about so-called *sections* that describe the contents and location of the code and the global data of a program. Additionally, object files contain information about how these sections are interconnected.

The Eigen Compiler Suite features several assemblers for different hardware architectures as well as compilers for different programming languages. Although all of these tools process different types of input files, they all generate the same type of output file, namely one object file per translation unit. Other tools like disassemblers and linkers on the other hand are able to continue processing the generated object files. Linkers for example allocate space for all sections and establish the necessary interconnections in order to create an executable binary program. Which tools actually generated the object files in the first place is irrelevant during this linking process. Object files therefore clearly separate the source code from its binary representation by forming a generic abstraction of code and data as shown in Figure 22.1.

Combining object files to build programs offers several advantages. First, it enables *interoperability* between different compilers and assemblers which allows writing code and data in one programming language and accessing it from another. Furthermore, using object files as a binary intermediate representation of a program effectively decouples the compilation from the linking stage and allows compilers and assemblers to become completely independent from the runtime environments targeted by linkers and their actual executable file formats. Additionally, the use of object files enables *separate compilation* which allows saving build time by compiling only those parts of a program that have actually changed instead of all of them. This also allows *statically linking* against precompiled object and library files which typically provide the necessary runtime support for programming languages, the hardware architecture and runtime environment, as well as additional external libraries.

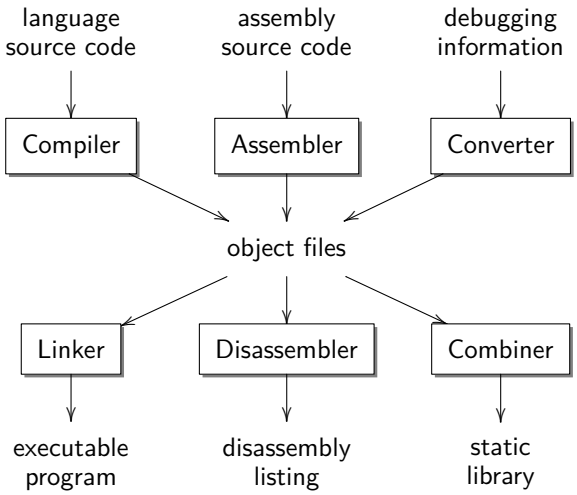


Figure 22.1: Object files as a generic abstraction of code and data in-between the various kinds of tools of the Eigen Compiler Suite

22.2 Object File Structure

Object files consist of a list of sections called *binary sections*. Each binary section describes a contiguous sequence of relocatable binary data like machine code and contains information about how the section interconnects with other binary sections. Sections have a unique name and carry additional information about their type and memory placement. This section describes all of this information and how it is used during linking.

22.2.1 Section Types

Each binary section is characterized by its *section type*. Section types distinguish between code and data as required by Harvard architectures and specify how the corresponding memory shall be placed by the linker. The Eigen Compiler Suite defines the following section types:

- Standard Code Sections `code`

  Standard code sections are typically used to model functions which are usually called by other code sections. Standard code sections have no special requirements for their placement in memory other than an optional alignment.

- Initializing Code Sections `initcode`

  Initializing code sections are placed by linkers in front of standard code sections. This guarantees that initializing code sections are executed automatically before any other code section. Unlike standard code sections, initializing code sections do not mimic functions as they are executed one after the other rather than being called.

- Data Initializing Code Sections `initdata`

  This type of section is the same as initializing code sections except that they are executed at the very beginning of the program. They are placed by linkers in front of other initializing code sections. They therefore allow global data to be initialized upon the execution of other initializing code sections. Data initializing code sections are typically used to initialize global data in environments that do not allow automatic memory initializations of data sections. This explicitly includes writing data to constant data sections.

- Standard Data Sections `data`

  Standard data sections provide the space and contents of global data. This data is usually modified by machine code during the execution of a program. Data sections may contain predefined data that is initialized automatically by the linker. If there are environments that do not allow global data to be initialized this way, data initializing code sections have to be used instead.

- Constant Data Sections `const`

  Constant data sections are data sections that are not supposed to change their contents during the execution of the program. Linkers may therefore place them together with all code sections in a special read-only memory if available.

- Metadata Sections `header` | `trailer`

  Metadata sections are data sections that contain metadata about a program. Linkers place heading metadata sections at the beginning and trailing metadata sections at the end of the resulting binary file. This allows mimicking the layout of some specific binary file format while linking several object files into a single binary file.

22.2.2 Section Options

The Eigen Compiler Suite defines four freely combinable *section options*. They describe how the linker shall treat sections that have the same name or are never used:

- Default Sections `default`

  By default, the linker diagnoses an error whenever it encounters two sections that have the same name. The default section option specifies that a section is only used as long

as there is no other section with the same name. Whenever the linker encounters two sections that have the same name and one of them is marked as default, it replaces the default one.

- Phantom Sections `phantom`

  Unused sections are usually not part of the binary program. The phantom section option causes the linker to omit even used sections.

- Required Sections `required`

  Sections are normally only used when referenced by other used sections. The required section option causes even unreferenced sections to be used. Standard code sections called `main` represent the *entry point* of programs and are always implicitly required. They are placed between initializing code sections and all other standard code sections.

- Shared Sections `shared`

  The shared section option specifies that sections with the same name may be merged together if they are also equal otherwise. Binary sections are equal if not only their name is the same but also all other information they carry including the binary data. Shared sections are typically used for constant data that may be defined in several object files but do not actually need to be duplicated in memory.

In addition to its name, a section may also have one or more so-called *alias names* which allow referring to individual parts of its binary data by name. The semantics of the section options described above also applies to all of their alias names.

22.2.3 Section Placement

After identifying all used sections, replacing default sections, and merging shared sections, linkers assign a unique memory address to every non-empty section in a process called *section placement*. References of a section to other sections are later resolved by their name and yield the assigned address of the named section. Referencing sections by name has the advantage that a programmer does not need to know or specify anything about the actual address of a section. Sometimes however, the target runtime environment or hardware architecture predetermine where a section has to be placed by the linker. The Eigen Compiler Suite therefore allows programmers to influence the section placement in the following ways:

- Aligned Sections `aligned` *Alignment*

  Aligned sections have an arbitrary address that is a multiple of a positive power of two called their *alignment*. Usually, the underlying hardware imposes requirements on the alignment of data and code when they are referenced or called. The Eigen Compiler Suite aligns the address of aligned sections according to their alignment expressed in multiples of octets.

- Fixed Sections `fixed` *Origin*

  Fixed sections have a predefined address called their *origin*. Before linkers layout any other section in memory, they place fixed sections at their origin expressed in octets. All others sections are then placed according to the semantics of their section type and alignment.

- Grouped Sections

  Sections that have the same type and alignment may additionally be assigned to a so-called *group*. This allows grouping otherwise unrelated sections together by forcing linkers to place them adjacent to each other. Each group has a unique name identifying a virtual section that encompasses the resulting section sequence. Groups may be nested and are placed consecutively in lexicographic order followed by all ungrouped sections of the same type.

With the exception of sections without binary data which always precede all other sections of the same type and group, sections are finally placed in order of occurrence. This allows initializing code sections to depend on the results of already executed code within the same object file. The overall placement of sections therefore depends first on their origin, then type, entry point, grouping, emptiness, and index. Figure 22.2 exemplifies this sequence by illustrating a typical section placement in memory and binary files.

22.2.4 Section References

Code in code sections usually needs to call functions defined in other code sections or to refer to data stored in data sections. Data sections on the other hand are sometimes used to store the address of other data and code sections. In order to establish these interconnections, the Eigen Compiler Suite defines *section links*. Section links allow a code or data section to refer to another section by name. Besides placing sections in memory, the main task of the linker is to resolve these references.

Each binary section maintains a list of links where each link refers to a different section. In addition to the name of the referenced section, each link contains a list of so-called *link patches*. A link patch defines where in the data of the binary section and how exactly linkers have to write the actual address of the referenced section. This information includes the offset within the data of the section the address has to be patched as well as a displacement that is added to the address beforehand. Additionally, the patch specifies whether the address is absolute or has to be relative to the actual memory address of the patch. The address can also be scaled according to potential alignment constraints. The *patch pattern* finally specifies how the resulting address overwrites individual bits of the target memory in order to comply with predefined space limitations, instruction encodings, and endianness constraints.

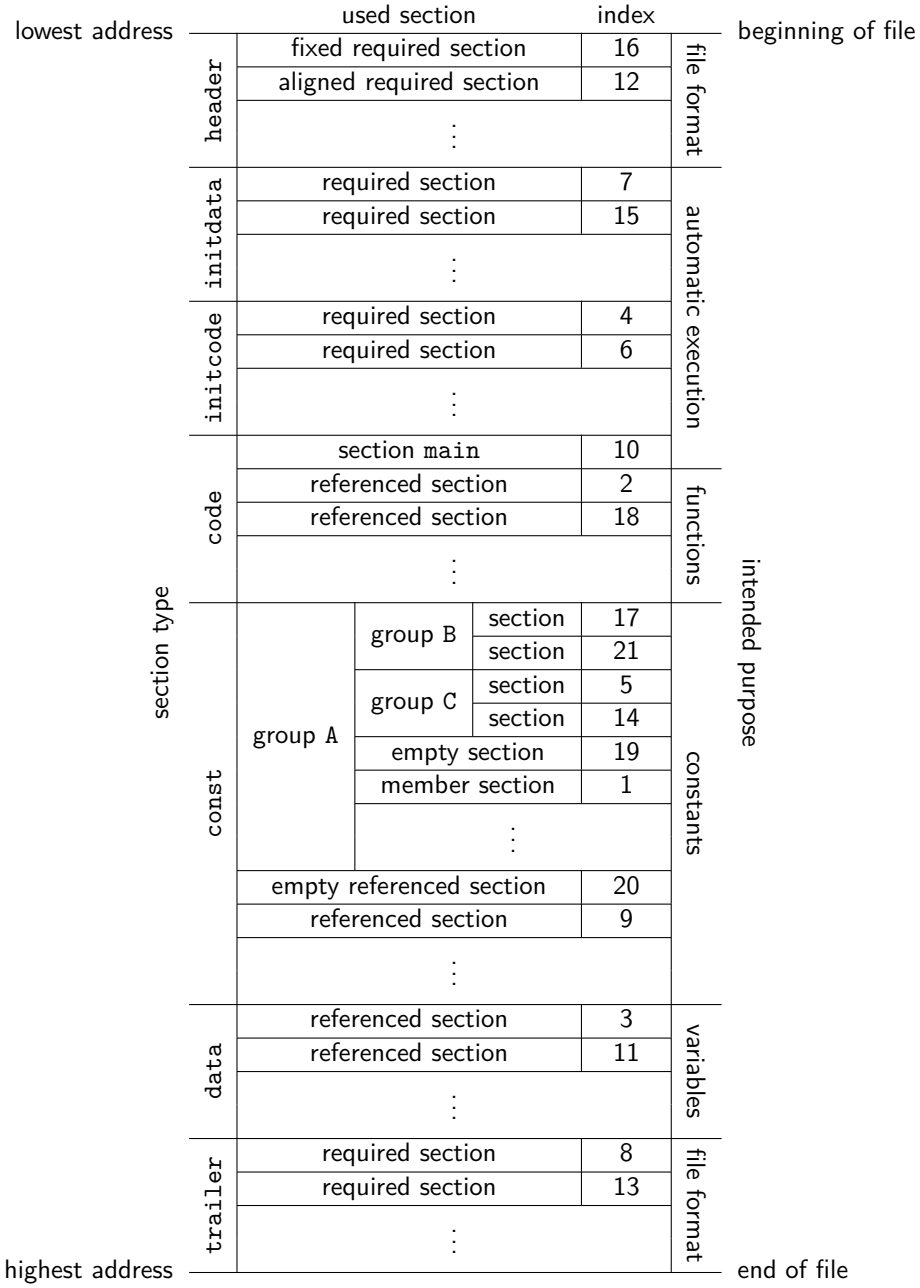


Figure 22.2: Typical section placement in memory and binary files

22.3 Object File Format

Object files are stored as plain text files according to the complete syntax specification given in Figure 22.3. They consist of the textual representation of an arbitrary number of binary sections according to the following syntax:

Object-File = *Sections*$_{opt}$

22.3.1 Binary Sections

Binary sections are represented in the object file as text according to the following syntax:

Sections = *Section* | *Sections Section*
Section = *Type Size Options*$_{opt}$ *Name Aliases*$_{opt}$
Group$_{opt}$ *Placement Segments*$_{opt}$ *Links*$_{opt}$
Type = `code` | `initcode` | `initdata` |
`data` | `const` | `header` | `trailer`
Size = decimal-integer
Name = double-quoted-string
Placement = `aligned` *Alignment* | `fixed` *Origin*
Alignment = decimal-integer
Origin = decimal-integer

The type of the binary section corresponds to the identifiers mentioned in Section 22.2.1. The section size specifies the total number of octets occupied by the binary data of the section. The alignment and origin of an aligned or fixed section influence its placement as described in Section 22.2.3.

22.3.2 Section Options

The options of a binary section are represented in the object file as text according to the following syntax:

Options = *Option* | *Options Option*
Option = `default` | `phantom` | `required` | `shared`

Section options may occur once in any order and correspond to the identifiers mentioned in Section 22.2.2.

22.3.3 Alias Names

The alias names of a binary section are represented in the object file as text according to the following syntax:

Object-File	=	*Sections*$_{opt}$
Sections	=	*Section* | *Sections Section*
Section	=	*Type Size Options*$_{opt}$ *Name Aliases*$_{opt}$ *Group*$_{opt}$ *Placement Segments*$_{opt}$ *Links*$_{opt}$
Type	=	`code` | `initcode` | `initdata` | `data` | `const` | `header` | `trailer`
Size	=	decimal-integer
Options	=	*Option* | *Options Option*
Option	=	`default` | `phantom` | `required` | `shared`
Name	=	double-quoted-string
Aliases	=	*Alias* | *Aliases Alias*
Alias	=	*Offset Name*
Offset	=	decimal-integer
Group	=	*Name Group*$_{opt}$
Placement	=	`aligned` *Alignment* | `fixed` *Origin*
Alignment	=	decimal-integer
Origin	=	decimal-integer
Segments	=	*Segment* | *Segments Segment*
Segment	=	*Offset Octets*
Octets	=	*Octet* | *Octets*␣*Octet*
Octet	=	*High-Quartet*␣*Low-Quartet*
High-Quartet	=	hexadecimal-digit
Low-Quartet	=	hexadecimal-digit
Links	=	*Link* | *Links Link*
Link	=	*Name Patches*$_{opt}$
Patches	=	*Patch* | *Patches Patch*
Patch	=	*Offset Mode Displacement Scale Pattern*
Mode	=	`abs` | `rel` | `siz` | `ext` | `pos` | `idx` | `cnt`
Displacement	=	signed-decimal-integer
Scale	=	decimal-integer
Pattern	=	*Masks* | *Flag*␣*Size*
Masks	=	*Mask* | *Masks*␣*Mask*
Mask	=	*Offset*␣*Bitmask*
Bitmask	=	*Octet*
Flag	=	`+` | `-`

Figure 22.3: Syntax of the object file format

Aliases = *Alias* | *Aliases Alias*
Alias = *Offset Name*
Offset = decimal-integer
Name = double-quoted-string

Alias names may not be duplicated and should differ from the name of the binary section. The offset of an alias allows referring to a specific octet within the data of the binary section using a different name. All names are represented using double-quoted strings and may contain standard escape sequences.

22.3.4 Section Groups

The group of a binary section is represented in the object file as text according to the following syntax:

Group = *Name* $Group_{opt}$
Name = double-quoted-string

Section groups define a potentially nested membership of the binary section and influence its placement as described in Section 22.2.3.

22.3.5 Binary Data

The data of a binary section is partitioned into segments which are represented in the object file as text according to the following syntax. There may not be any white-space character in-between the hexadecimal digits of octets:

Segments = *Segment* | *Segments Segment*
Segment = *Offset Octets*
Offset = decimal-integer
Octets = *Octet* | *Octets*␣*Octet*
Octet = *High-Quartet*␣*Low-Quartet*
High-Quartet = hexadecimal-digit
Low-Quartet = hexadecimal-digit

Each segment contains one or more octets that represent the binary data of the section starting at the corresponding offset. This offset plus the number of octets may not exceed the overall section size. Overlapping segments define the binary data of a section in order of occurrence. Binary data not covered by any segment is initialized to zero.

22.3.6 Section Links

The links of a binary section are represented in the object file as text according to the following syntax:

Links	=	*Link* | *Links Link*
Link	=	*Name Patches*$_{opt}$
Name	=	double-quoted-string

The name of a link identifies the section that has to be referenced whereas an empty name always yields address zero. If it contains question marks, the actual name of the referenced section begins behind the last question mark. If none of the sections named in front of a question mark are actually used, the name of the section instead begins behind an optional colon following the last question mark.

22.3.7 Link Patches

The patches of a section link are represented in the object file as text according to the following syntax:

Patches	=	*Patch* | *Patches Patch*
Patch	=	*Offset Mode Displacement Scale Pattern*
Offset	=	decimal-integer
Mode	=	`abs` | `rel` | `siz` | `ext` | `pos` | `idx` | `cnt`
Displacement	=	signed-decimal-integer
Scale	=	decimal-integer

The offset specifies the position of the octet within the current binary section where the referenced section has to be patched by linkers. The remainder of the patch specifies how the referenced section shall actually be evaluated. The patch mode `abs` specifies that the referenced section evaluates to its absolute address expressed in octets. The patch mode `rel` specifies that the absolute address of the referenced section shall be decremented by the target address of the patch in order to yield a relative address. If the patch mode `siz` is used, the referenced section evaluates to its binary size expressed in octets. The patch mode `ext` yields the absolute address of the referenced section plus its binary size. Using the patch mode `pos`, the referenced section evaluates to its position relative to the beginning of its associated group expressed in octets. The patch mode `idx` specifies that the referenced section evaluates to its index in the direct member sequence of its group. The patch mode `cnt` can be used to reference groups and evaluates to the number of direct group members. The evaluated value is then incremented or decremented by the optionally signed displacement and shifted to the right by the specified scale. Overlapping link patches are applied in order of occurrence.

22.3.8 Patch Patterns

The patterns of a link patch are represented in the object file as text according to the following syntax. There may not be any white-space character in-between the elements of a pattern:

Pattern	=	*Masks* | *Flag*␣*Size*
Masks	=	*Mask* | *Masks*␣*Mask*
Mask	=	*Offset*␣*Bitmask*
Offset	=	decimal-integer
Bitmask	=	*Octet*
Octet	=	*High-Quartet*␣*Low-Quartet*
High-Quartet	=	hexadecimal-digit
Low-Quartet	=	hexadecimal-digit
Flag	=	+ | -
Size	=	decimal-integer

A pattern consists of a list of one to eight masks which define how the evaluated address has to be written to binary data. Each mask contains an offset and the corresponding bitmask. The single-digit offset specifies the relative displacement of the mask with respect to the enclosing link patch which may not exceed the overall section size. The bitmask is given as the value of a single octet in hexadecimal form. It specifies the number of least significant bits consecutively taken from the address, as well as the mask which is used to write that part of its value to binary data. The pattern `5ff403` for example consists of two masks. The first mask `5ff` tells the linker to write the value of the first eight bits of the address to all bits of the octet at the target address displaced by five. The second mask `403` causes the linker to write the value of the ninth and tenth bit of the address to the lowest two bits of the octet at the target address displaced by four. Patterns which consist exclusively of full bitmasks with consecutive offsets can also be represented by a flag indicating ascending or descending offsets followed by the number of masks. The pattern `0ff1ff2ff` for example can thus be abbreviated by `+3`.

22.4 Linker Tools

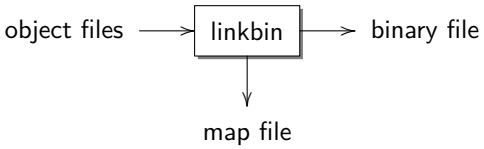
Linkers process object files that were previously generated by the various compilers and assemblers of the Eigen Compiler Suite. The task of a linker is to merge all object files given to it into an executable program and to store the resulting binary image using a specific file format. All of these linker tools accept command-line arguments which are taken as names of the actual object files. If no arguments are provided, object files are read from the standard input stream. Linkers create output files in the current working directory by reusing the name of the last object file whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools. For basic examples of using some of these tools in practice, see Chapter 3.

22.4.1 linklib

`linklib` is an object file combiner. It creates a static library file by combining all object files given to it into a single one.

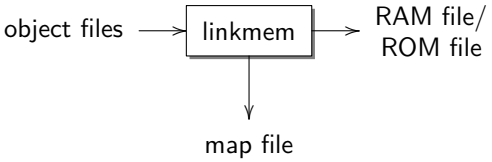
22.4.2 linkbin

`linkbin` is a linker for plain binary files. It links all object files given to it into a single image and stores it in an executable binary file that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting binary file can be specified by putting it into a constant data section called `_extension`.



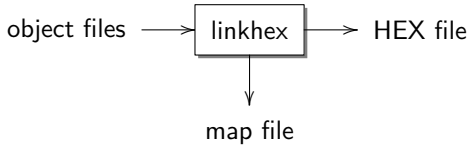
22.4.3 linkmem

`linkmem` is a linker for plain binary files partitioned into random-access and read-only memory. It links all object files given to it into two distinct images, one for data sections and one for code and constant data sections, and stores each image in a binary file that begins with the first linked section of the corresponding type. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order.



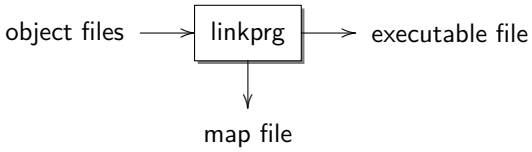
22.4.4 linkhex

`linkhex` is a linker for Intel HEX files. It links all code sections of the object files given to it into single image and stores the image in an Intel HEX file [3] that begins with the first linked section. It also creates a map file that lists the address, type, name, size, and grouping of all used sections in placement order.



22.4.5 linkprg

`linkprg` is a linker for GEMDOS executable files. It links all object files given to it into a single image and stores the image in an Atari GEMDOS executable file [4]. It also creates a map file that lists the address relative to the text segment, type, name, size, and grouping of all used sections in placement order. The filename extension of the resulting executable file can be specified by putting it into a constant data section called `_extension`. The GEMDOS executable file format requires all patch patterns of absolute link patches to consist of four full bitmasks with descending offsets.



23 Intermediate Code

This chapter describes the intermediate code representation used by the Eigen Compiler Suite as the common interface between the various programming languages and hardware architectures it supports.

Unsere Eigenschaften müssen wir kultivieren,
nicht unsere Eigenheiten.

— Johann Wolfgang von Goethe

23.1 Introduction

Rather than compiling source code directly into machine code, the Eigen Compiler Suite makes use of a so-called *intermediate code representation*. For each programming language supported by the Eigen Compiler Suite, there is a so-called *front-end* which translates source code into intermediate code. This abstract representation of the original program is then passed to a so-called *back-end* which in turn transforms it into machine code and debugging information. Figure 23.1 visualizes the role and data flow of the intermediate code representation in-between front-ends and back-ends.

The advantage of this design is a clear separation between the various front-ends for the programming languages and the back-ends for the hardware architectures. Since the intermediate code representation is the only interface between these parts of a compiler, they are completely independent from each other and may therefore be freely combined. Implementing a new programming language for all supported hardware architectures and vice versa can thus be achieved by just adding one other front-end or back-end to the Eigen Compiler Suite. Furthermore, examining the output of front-ends and providing intermediate code as input to back-ends allows their respective implementations to be validated in isolation. Using an interpreter for intermediate code that emulates its underlying abstract machine, front-end implementations can additionally be tested by executing programs directly without having to target and rely on actual hardware architectures or runtime environments.

The following sections describe the semantics of the intermediate code representation alongside the syntax of its textual representation using the generic assembly language. For more information about the generic assembly language and how to use it, see Chapter 8. For more information about debugging information and its representation, see Chapter 24.

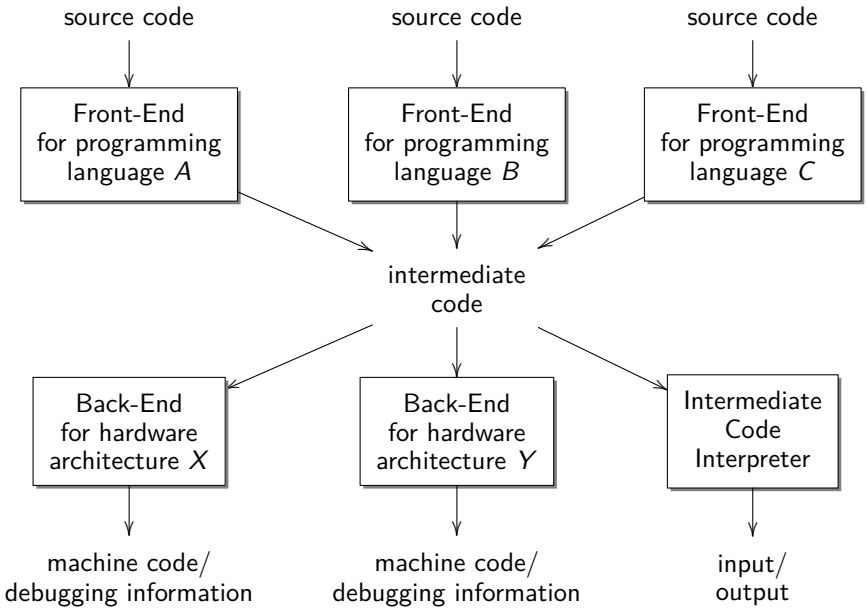


Figure 23.1: Intermediate code representation of programs in-between front-ends and back-ends

23.2 Programming Model

The underlying programming model of the intermediate code representation is based on the execution of instructions grouped into *sections*. Each section has a unique identifier that allows it to be referenced by instructions from other sections. Additionally, each section has a specific *type*, that allows distinct classes of instructions to be used. The following sections describe these section types and the overall execution environment defined by the programming model.

23.2.1 Code Sections

A code section stores the actual instructions that have to be executed at runtime. Each functional unit of a programming language is typically mapped to a single code section. Instructions in code sections can reference other sections by either calling them or accessing the data stored in them. The following kinds of code sections define when and how code is executed:

- Standard Code Sections `.code` *Identifier*

  Standard code sections model standard functions and are typically called by instructions of other code sections. Functions often create stack frames and have to return the control to their caller at the end of their execution.

- Initializing Code Sections `.initcode` *Identifier*

  Initializing code sections are executed automatically at the start of the program and allow its state to be initialized. Since these code sections are just executed in sequence as opposed to being explicitly called, there is no need to return from initializing code sections because there is no caller.

- Data Initializing Code Sections `.initdata` *Identifier*

  Data initializing code sections are executed automatically at the very beginning of a program and allow initializing data sections that are required in all other code sections. This does explicitly include constant data sections because there are architectures for which all data sections have to be initialized by code sections at runtime.

- Assembly Code Sections `.assembly`

  Assembly code sections are nameless pseudo sections for representing arbitrary assembly code that defines its own code and data sections. The actual assembly code consists of one or more inline assembly instructions as described in Section 23.4.5. Each instruction invokes the underlying assembler once and has unrestricted access to all of its features.

23.2.2 Data Sections

A data section stores instructions that describe the contents and the layout of some memory region. This memory can be accessed by the instructions of a code section. There are two kinds of data sections which specify the access rights to the contents of the memory region:

- Standard Data Sections `.data` *Identifier*

  Standard data sections model standard memory regions with read and write access. They are typically used to represent global variables.

- Constant Data Sections `.const` *Identifier*

  Constant data sections model memory regions with read-only access. Since these data sections are not supposed to change their contents, they are typically used to represent constants and strings.

23.2.3 Type Sections

A type section stores instructions that describe the type system of the programming language for debugging purposes. The following kind of type section declares the layout and representation of user-defined and predefined types:

- Standard Type Sections `.type` *Identifier*

  Standard type sections contain a single type declaration and define a new name for that type. They are typically used to represent type definitions of the programming language.

Since type sections contain only metadata about the types used by a program, they have no machine code representation. They do however contribute to the debugging information generated by the back-end, see Section 23.3.5.

23.2.4 Order of Execution

Code sections are executed automatically by the runtime system according to their type as described above. The concrete order of execution is as follows:

1. All data initializing code sections are executed in order of occurrence. They may only depend on the results of already executed data initializing code sections.
2. All remaining initializing code sections are executed in order of occurrence. They may only depend on the results of already executed initializing code sections.
3. A standard code section called `main` is executed. This section represents the actual entry point of a program and is therefore always implicitly required. It may depend on the results of both previous execution steps but has to eventually return the control of execution to the host environment.

All remaining code sections are called depending on the actual code executed in these three execution steps. They may only depend on results that are also guaranteed for the code sections they were called from.

23.2.5 Stack Operations

The programming model of the intermediate code defines local memory for every thread of execution called the *stack* as exemplified in Figure 23.2. The stack is represented by two special registers called the *stack pointer* and the *frame pointer*, see Section 23.3.2. Although they allow direct access to the stack memory, there are some intermediate code instructions that modify the contents of these registers as well as the stack memory itself. These instructions are `push`, `pop`, `call`, and `ret`. See the instruction reference in Section 23.4 for detailed information about the operation of these instructions.

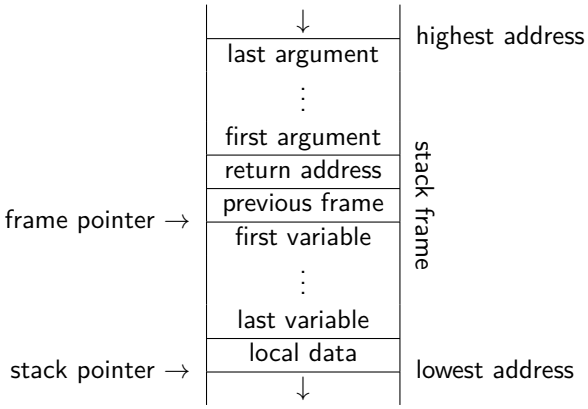


Figure 23.2: Typical intermediate code stack layout

The stack pointer always points to the topmost element on the stack. Operations like pushing and popping data to or from the stack access the topmost element and implicitly modify the stack pointer accordingly. The stack memory grows from top to bottom such that the topmost element on the stack has the lowermost address in memory. By pushing data on the stack, the stack pointer is therefore first decremented and afterward used to store the data at the decremented memory address. The actual size of this decrement depends on the target hardware architecture. There are architectures that require the stack pointer address to conform to some alignment constraints. The underlying machine code generator of a back-end provides platform-specific information that includes alignment requirements like this.

The stack memory is generally used to store arbitrary local data or to pass arguments to another code section involved in a function call. The virtual general-purpose registers of the intermediate code are in general mapped arbitrarily to the physical registers of the underlying hardware architecture. For this reason, they cannot be used in general to pass arguments to other code sections. Likewise, their contents may be invalidated in-between function calls. Arguments and values of registers that are still in use afterward the function call, have therefore to be stored on the stack.

Functions may maintain their own local memory by creating so-called *stack frames*. Stack frames allow reserving a certain amount of memory on the stack that is especially useful for local variables or recursive functions. A stack frame is represented by the frame pointer and created by the instruction `enter`, see Section 23.4.17. By creating a new stack frame, the value of the frame pointer is saved on the stack and afterward set to the current value of the stack pointer. The stack pointer is then decremented by the size of the stack frame. Since the stack pointer may be modified again afterward, it is best to access the local data using the frame pointer. A stack frame is deleted using the instruction `leave`, see Section 23.4.24. It just restores the previous values of the frame and stack pointer.

23.3 Intermediate Code Representation

The intermediate representation of all code and global data is based on instructions within code or data sections. Instructions themselves have operands of different models which may be constants, registers, or memory operands. This section describes how these components of the intermediate code are represented within the Eigen Compiler Suite.

23.3.1 Types

The intermediate code representation defines a set of several basic *types*. These types specify the way how data that is stored in memory or registers shall be interpreted by an instruction. They consist of one of the following *type models* and an additional *type size* expressed in octets:

- Signed Integers `s1` | `s2` | `s4` | `s8`

  The type model for signed integers allows accessing data of signed integral data type. There are four different type sizes available: 1, 2, 4, and 8. For the type size n all valid integers lie within the range -2^{8n-1} to $+2^{8n-1}-1$. The result of arithmetic operations causing integer overflows is undefined.

- Unsigned Integers `u1` | `u2` | `u4` | `u8`

  The type model for unsigned integers allows accessing data of unsigned integral data type. There are four different type sizes available: 1, 2, 4, and 8. For the type size n all valid integers lie within the range 0 to $2^{8n}-1$. Arithmetic operations on unsigned integers obey the laws of arithmetic modulo 2^{8n} and do therefore not overflow.

- Floating-Point Numbers `f4` | `f8`

  The type model for floating-point numbers allows accessing data stored according to formats defined in the IEEE standard for floating-point arithmetic [27]. There are two different type sizes available: 4 and 8. Floating-point numbers of size four are stored using the single-precision format, and floating-point numbers of size eight are stored using the double-precision format.

- Data Pointers `ptr`

  The type model for data pointers allows accessing data that represents the address of a datum within a data section. There is only one type size available and it is predefined by the target platform.

- Function Pointers `fun`

  The type model for function pointers allows accessing data that represents the address of an instruction within a code section. There is only one type size available and it is predefined by the target platform.

For some hardware architectures, the representation of data with the same value but different types may also be equal physically. But in order to ensure a consistent virtual representation, all data shall be read using the same type it was written beforehand. If the representation of the data type has to be explicitly changed, the Eigen Compiler Suite provides a special conversion instruction. This operation shall also be used even if the types in question differ only with respect to their size.

23.3.2 Registers

The abstract intermediate code representation defines the following set of virtual general-purpose and special-purpose registers:

- Eight General-Purpose Registers `$0` | `$1` | ... | `$7`

  These general-purpose registers can be freely used to store and restore intermediate results evaluated in expressions. Their actual values depend on the type used to access the registers and are invalidated across function calls. Before calling a function, it is therefore important to save the value of a register onto the stack in order to safely restore it afterward.

- The Result Register `$res`

  The result register can be used to store and restore the result of a function call. Its value depends on the type used to access the register and is not invalidated when returning from a function call.

- The Stack Pointer `$sp`

  The stack pointer stores an address that points to the current top of the stack. Several instructions implicitly change the stack pointer by pushing or popping values to or from the stack, see Section 23.2.5. The address stored in the stack pointer has to be aligned according to the stack alignment of the target platform.

- The Frame Pointer `$fp`

  The frame pointer stores an address that points to the beginning of the current stack frame. This register can therefore be used to access the local variables of a function or the arguments passed to it.

- The Link Register `$lnk`

  The link register stores the address of the instruction following a `call` instruction, see Section 23.4.12. This allows function calls on architectures that support this register to bypass the stack when storing the return address.

23.3.3 Operands

The intermediate code representation defines several different instructions that can take up to three operands. Each operand is an instance of one of the following *operand models*:

- Size *Number*

  The size operand model is used to express a non-negative integer number. It generally denotes a number of octets but can also refer to some other quantity.

- Offset *±Number*

  The offset operand model represents relative offsets with respect to the instruction directly following the current one. The sequential program flow of a program is altered by branching to the specified instruction within the current code section. Offsets are only used as first operand of an instruction.

- String *String*

  The string operand model stores character sequences of arbitrary length. It is used to carry metadata from the front-end to the back-end. Strings can contain standard escape sequences to carry special characters.

- Type *Type*

  The type operand model declares a single basic type as defined in Section 23.3.1.

- Immediate Value *Type Number* | *Type ±Number*

  The immediate value operand model is used to store a constant numeric value. The range and format of the valid values is predefined by the given type.

- Register *Type Register* | *Type Register ± Number*

  The register operand model can be used to access the values of one of the registers defined in Section 23.3.2. The given type specifies the representation of the value of the register and has to be of type model data pointer for the special-purpose pointer registers. If a register is accessed using type model data pointer and is not used as destination operand of an instruction, its evaluated address can be incremented or decremented by an optional constant displacement.

- Address *Type* @*Identifier* | *Type* @*Identifier ± Number*
  | *Type* @*Identifier + Register* | *Type* @*Identifier + Register ± Number*

  The address operand model represents the address of a code or data section named by the identifier. The identifier may be enclosed in double quotes and contain any character sequence including escape sequences like strings where an empty identifier always yields address zero. If the identifier contains question marks, the actual name of the referenced

section begins behind the last question mark. If none of the sections named in front of a question mark are actually used, the name of the section instead begins behind an optional colon following the last question mark.

The type of an address has to be a function pointer if it names a code section. Otherwise its type has to be a data pointer and the resulting address can be incremented by an optional constant displacement. In code sections, this address can additionally be incremented by the value of a register that stores a data pointer.

- Memory *Type* `[` *Number* `]`
  | *Type* `[` *Register* `]` | *Type* `[` *Register* ± *Number* `]`
  | *Type* `[` `@`*Identifier* `]` | *Type* `[` `@`*Identifier* ± *Number* `]`
  | *Type* `[` `@`*Identifier* + *Register* `]` | *Type* `[` `@`*Identifier* + *Register* ± *Number* `]`

  The memory operand model allows accessing the data value stored at a specified address using the representation of the given type. The actual address can either be an immediate address, a register that stores a data pointer, the name of a data section, or any combination thereof. The evaluation of the address of a section is the same as for sections named in the address operand model.

23.3.4 Instructions

Instructions are composed of one of the mnemonics listed in Table 23.1 followed by up to three operands. If the instruction generates a result, it is generally stored in the first register or memory operand. Except where otherwise noted, the types of all typed operands have to match. Operands and their different models are described in Section 23.3.3.

Table 23.1: Intermediate code instruction set (50 instructions)

`add`	`brne`	`fix`	`nop`	`ret`
`alias`	`call`	`func`	`not`	`rsh`
`and`	`conv`	`jump`	`or`	`sub`
`array`	`copy`	`leave`	`pop`	`sym`
`asm`	`def`	`loc`	`ptr`	`trap`
`br`	`div`	`lsh`	`push`	`type`
`break`	`enter`	`mod`	`rec`	`unfix`
`breq`	`enum`	`mov`	`ref`	`value`
`brge`	`field`	`mul`	`req`	`void`
`brlt`	`fill`	`neg`	`res`	`xor`

The operation of these instructions as well as all valid combinations of their operands are specified in the instruction reference in Section 23.4. The instructions themselves are grouped together in code, data, or type sections as described in Section 23.2. The actual section type defines which instruction subset is available for use. Table 23.2 categorizes all instructions pairwise according to their operation and lists the section types they are available in. The

emphasized instruction categories denote operations that depend on the target architecture and therefore must be implemented by every machine code generator.

Instructions and sections are textually represented using the generic assembly language. Tools like `cppcode` or `obcode`, see Section 23.7, translate programs into intermediate code and generate an assembly code listing thereof. Other tools like `cdcheck` or `cdrun` are able to read and process assembly files written using intermediate code. These tools predefine some constant definitions whose actual values depend on the target hardware architecture. The definition `int` evaluates to the name of its default signed integer type, while `flt` evaluates to the name of its default floating-point type. Concatenating the types `int`, `flt`, `ptr`, and `fun` as well as the return address `ret` and the link register `lnk` with an underscore followed by `size`, `align`, and `arg` yields definitions evaluating to the size, alignment, and stack argument size of the respective entity. The definition `stack_disp` evaluates to the architecture-dependent stack pointer displacement that has to be added to every memory access on the stack. The filename of the source code and current position therein are available using the definitions `file` and `line`. For more information about the generic assembly language and how to use it, see Chapter 8. Since intermediate code is an abstract representation however, all assembly language features which define, align, or otherwise refer to binary data are not available.

23.3.5 Debugging Information

Several intermediate code instructions contain metadata about the program for debugging purposes. They do not generate any machine code but contribute to the debugging information generated by the back-end. This debugging information consists of the following metadata for debuggers capable of program animation and memory inspection:

- Breakpoints

  Breakpoints map machine code addresses to their corresponding source code locations and vice versa. They enable user requests to suspend the program execution at predefined statement units of the programming language.

- Symbol Declarations

  Symbol declarations map memory addresses to their corresponding symbolic names and vice versa. They are typically used to represent storage objects like variables and parameters of the programming language.

- Type Declarations

  Type declarations describe the layout and data representation of memory regions occupied by storage objects. They are used to represent data types of the programming language and its user-defined data structures for memory inspection.

Instruction Category	Section Types	Mnemonic	See Section	Mnemonic	See Section
Memory Layout	all	`alias`	23.4.2	`req`	23.4.39
	data	`def`	23.4.15	`res`	23.4.40
Data Management	code	`mov`	23.4.28	`conv`	23.4.13
	code	`copy`	23.4.14	`fill`	23.4.20
Arithmetic	code	`add`	23.4.1	`sub`	23.4.43
	code	`mul`	23.4.29	`div`	23.4.16
	code	`mod`	23.4.27	`neg`	23.4.30
Bit Manipulation	code	`and`	23.4.3	`or`	23.4.33
	code	`xor`	23.4.50	`not`	23.4.32
	code	`lsh`	23.4.26	`rsh`	23.4.42
Function Call	code	`push`	23.4.36	`pop`	23.4.34
	code	`call`	23.4.12	`ret`	23.4.41
	code	`enter`	23.4.17	`leave`	23.4.24
Branching	code	`br`	23.4.6	`jump`	23.4.23
	code	`breq`	23.4.8	`brne`	23.4.11
	code	`brlt`	23.4.10	`brge`	23.4.9
Special Purpose	code	`nop`	23.4.31	`asm`	23.4.5
	code	`fix`	23.4.21	`unfix`	23.4.47
Debugging	all	`loc`	23.4.25		
	code	`break`	23.4.7	`trap`	23.4.45
Symbol Declaration	code	`sym`	23.4.44		
	all	`field`	23.4.19	`value`	23.4.48
Type Declaration	all	`void`	23.4.49	`type`	23.4.46
	all	`array`	23.4.4	`rec`	23.4.37
	all	`ptr`	23.4.35	`ref`	23.4.38
	all	`func`	23.4.22	`enum`	23.4.18

Table 23.2: Intermediate code instruction categories

Operand Class	Operand Models
StrTyp	*String* or *Type*
ImmAdr	*Immediate* or *Address*
RegMem	*Register* or *Memory*
ImmRegMem	*Immediate*, *Register*, or *Memory*
ImmRegAdrMem	*Immediate*, *Register*, *Address*, or *Memory*
Pointer	*ImmRegAdrMem* of type data pointer
Function	*ImmRegAdrMem* of type function pointer

Table 23.3: Intermediate code operand classes

Type declarations are represented by an arbitrary set of consecutive type declaration instructions. This allows describing complex compound types involving pointers and arrays in order of occurrence. All type declarations in code sections or type sections refer to the type of the preceding symbol, field, or enumerator declaration. Otherwise they declare the type of the result returned by the code section, or the type of the data or type section itself. For more information about debugging information and its representation, see Chapter 24.

23.4 Instruction Set Reference

This section describes the operation of all instructions available in intermediate code. Every instruction has one mnemonic and up to three operands. The valid combinations of mnemonics and operand models is given in the syntax definition for every instruction. The syntax definition uses either an operand model as described in Section 23.3.3 or an operand class as shown in Table 23.3. An operand class consists of several different operand models. The model of the actual operand has to be one of the models named in the operand class.

23.4.1 `add` *RegMem*, *ImmRegAdrMem*, *ImmRegAdrMem* — Addition

Adds the value of the third operand to the value of the second operand and stores the result in the first register or memory operand. This operation is not available for function pointers.

23.4.2 `alias` *String* — Alias Name Definition

Defines an alias name for the code or data following the current instruction. The address of the alias name equals to the name of the section plus the corresponding displacement. The operand specifies the name of the alias which may not be duplicated and should differ from the name of the section.

23.4.3 `and` *RegMem*, *ImmRegMem*, *ImmRegMem* — Logical AND

Performs a logical AND operation on the bits of the values of the second and third operand and stores the result in the first register or memory operand. This operation is not available for function pointers and floating-point numbers.

23.4.4 `array` *Size*, *Size* — Array Type Declaration

Declares an array type which represents a collection of consecutive storage objects with the same element type that can be accessed by indexing. The first operand specifies the index of the first element of the array and the second operand the number of elements contained therein. This compound type declaration requires a subsequent declaration for its element type.

23.4.5 `asm` *String*, *Size*, *String* — Inline Assembly

Passes inline assembly code to the underlying assembler of the back-end. The `asm` instruction itself is then replaced by the generated machine code. The first operand names the source file and the second operand a line within that file. This information is needed for diagnostic messages of the assembler. The third operand contains the actual assembly code in one string. For more information about the generic assembly language and how to use it, see Chapter 8.

23.4.6 `br` *Offset* — Unconditional Branch

Continues execution at the instruction specified by the relative instruction offset. Indirect branches or branches into other code sections can be performed using the `jump` instruction, see Section 23.4.23.

23.4.7 `break` — Breakpoint

Identifies the beginning of a suspendable statement unit of the programming language. Breakpoints require subsequent source code locations, see Section 23.4.25.

23.4.8 `breq` *Offset*, *ImmRegAdrMem*, *ImmRegAdrMem* — Branch if Equal

Continues execution at the instruction specified by the relative instruction offset if the values of the second and third operand match.

23.4.9 `brge` *Offset*, *ImmRegAdrMem*, *ImmRegAdrMem* — Branch if Greater Than or Equal

Continues execution at the instruction specified by the relative instruction offset if the value of the second operand is greater than or equal to the value of the third operand.

23.4.10 `brlt` *Offset*, *ImmRegAdrMem*, *ImmRegAdrMem* — Branch if Less Than

Continues execution at the instruction specified by the relative instruction offset if the value of the second operand is less than the value of the third operand.

23.4.11 `brne` *Offset*, *ImmRegAdrMem*, *ImmRegAdrMem* — Branch if Not Equal

Continues execution at the instruction specified by the relative instruction offset if the values of the second and third operand do not match.

23.4.12 `call` *Function*, *Size* — Function Call

Continues execution in the code section specified by the function pointer of the first operand and stores the address of the immediately following instruction in order to resume its execution after returning from the function. This address is either stored in the link register or pushed onto the stack if the target architecture does not support the link register. The second operand specifies the amount of octets to be additionally popped from the stack upon returning from a function that needed arguments. This number has to be aligned according to the stack alignment of the target platform. For more information about stack operations, see Section 23.2.5.

23.4.13 `conv` *RegMem*, *ImmRegAdrMem* — Datum Conversion

Converts the value of the second operand into the type specified by the first register or memory operand and stores the result there. Converting two non-floating-point values yields the least integer congruent to the source operand modulo 2^{8n} where n is the size of the destination type. Converting a floating-point value to an integer value truncates and discards the fractional part. This operation is not available for conversions between pointers and floating-point numbers.

23.4.14 `copy` *Pointer*, *Pointer*, *ImmRegMem* — Data Copy

Copies data from the memory address named in the second operand to the memory address named in the first operand. The third operand specifies the amount of octets to be copied.

23.4.15 `def` *ImmAdr* — Datum Definition

Enlarges the current data section by a single datum and initializes the resulting memory with the binary representation of the specified value. The datum has to be aligned according to the data alignment of the target platform. If the operand is an address, it cannot be incremented by a register, since registers are only available in code sections.

23.4.16 `div` *RegMem*, *ImmRegMem*, *ImmRegMem* — Division

Divides the value of the second operand by the value of the third operand and stores the result in the first register or memory operand. For signed and unsigned integers, the fractional part of the quotient is discarded. This operation is undefined if the third operand evaluates to zero, and is not available for pointers.

23.4.17 `enter` *Size* — Stack Frame Creation

Pushes the value of the frame pointer onto the stack and replaces it with the stack pointer. The stack pointer is then decremented by the amount of octets specified by the operand. This number has to be aligned according to the stack alignment of the target platform. For more information about stack frames, see Section 23.2.5.

23.4.18 `enum` *Offset* — Enumeration Type Declaration

Declares an enumeration type that stores a single value from a predefined set of named constants called enumerators. The operand defines the extent of the declaration in terms of the instruction range given by the relative instruction offset. All enumerators declared within this extent belong to the most current enumeration declaration, see Section 23.4.48. This compound type declaration requires a subsequent declaration for its underlying type.

23.4.19 `field` *String*, *Size*, *Immediate* — Field Declaration

Declares a field which associates the symbolic name of a storage object with its offset in the most current record declaration, see Section 23.4.37. The first operand specifies the name of the field where an empty name conventionally denotes inheritance rather than composition. The second operand defines the offset of the field while the third stores the occupied bits in case it is a bit field. Field declarations require subsequent source code locations and type declarations.

23.4.20 `fill` *Pointer*, *Pointer*, *ImmRegAdrMem* — Data Initialization

Initializes a memory region starting from the address named in first operand with the value of the third operand. The second operand specifies the number of data values to be initialized. The type of the third operand does not have to be a pointer.

23.4.21 `fix` *Register* — Fix Register Mapping

Fixes the current mapping of an unfixed general-purpose register given by the operand to its corresponding physical register. This mapping stays fixed until the next occurrence of a corresponding `unfix` instruction in the instruction sequence, see Section 23.4.47. A fixed

mapping is required whenever an instruction sequence needs to refer to the same physical register before and after overwriting the value of a general-purpose register.

23.4.22 `func` *Offset* — Function Type Declaration

Declares a function type which represents the type of a functional unit of the programming language. The operand defines the extent of the declaration in terms of the instruction range given by the relative instruction offset. All freestanding types declared within this extent belong to the most current function type declaration and correspond to its parameters in order of occurrence. This compound type declaration requires a subsequent type declaration for the type of the returned result.

23.4.23 `jump` *Function* — Indirect Branch

Continues execution in the code section specified by the function pointer of the operand. This instruction is useful for branching into other code sections or indirect branches like returning from a function using the link register. Direct branches within the same code section can be performed using the `br` instruction, see Section 23.4.6.

23.4.24 `leave` — Stack Frame Deletion

Sets the stack pointer to the value of the frame pointer and pops the previous value of the frame pointer from the stack. For more information about stack frames, see Section 23.2.5.

23.4.25 `loc` *String*, *Size*, *Size* — Source Code Location

Maps the current section or a preceding breakpoint, symbol declaration, or type declaration to its source code location. The first operand names the source file and the second and third operands a line and column within that file.

23.4.26 `lsh` *RegMem*, *ImmRegMem*, *ImmRegMem* — Left Shift

Performs a left shift on the bits of the value of the second operand by the amount specified by the third operand and stores the result in the first register or memory operand. This operation is undefined if the third operand is negative or greater than or equal to its size in bits, and is not available for pointers and floating-point numbers.

23.4.27 `mod` *RegMem*, *ImmRegMem*, *ImmRegMem* — Modulo

Divides the value of the second operand by the value of the third operand and stores the remainder in the first register or memory operand. For signed integers, the remainder has the

same sign as the dividend. This operation is undefined if the third operand evaluates to zero, and is not available for pointers and floating-point numbers.

23.4.28 `mov` *RegMem*, *ImmRegAdrMem* — Datum Copy

Copies the value of the second operand to the first register or memory operand.

23.4.29 `mul` *RegMem*, *ImmRegMem*, *ImmRegMem* — Multiplication

Multiplies the value of the second operand by the value of the third operand and stores the result in the first register or memory operand. This operation is not available for function pointers.

23.4.30 `neg` *RegMem*, *ImmRegMem* — Negation

Negates the value of the second operand and stores the result in the first register or memory operand. This operation is not available for pointers.

23.4.31 `nop` — No Operation

Performs no operation. This instruction allows an optimizer to elide unnecessary instructions as it is completely ignored by machine code generators.

23.4.32 `not` *RegMem*, *ImmRegMem* — Logical NOT

Performs a one's complement negation on the bits of the value of the second operand and stores the result in the first register or memory operand. This operation is not available for pointers and floating-point numbers.

23.4.33 `or` *RegMem*, *ImmRegMem*, *ImmRegMem* — Logical OR

Performs a logical OR operation on the bits of the values of the second and third operand and stores the result in the first register or memory operand. This operation is not available for pointers and floating-point numbers.

23.4.34 `pop` *RegMem* — Pop from Stack

Stores the value pointed to by the stack pointer register into the first register or memory operand and increments the stack pointer register by the aligned size of that value. For more information about stack operations, see Section 23.2.5.

23.4.35 `ptr` Pointer Type Declaration

Declares a pointer type which may represent the address of a storage object that can be dereferenced. This compound type declaration requires a subsequent declaration for the type of the referenced storage object.

23.4.36 `push` *ImmRegAdrMem* Push onto Stack

Decrements the stack pointer register by the aligned size of the value of the operand and stores that value to the memory pointed to by the stack pointer. For more information about stack operations, see Section 23.2.5.

23.4.37 `rec` *Offset*, *Size* Record Type Declaration

Declares a record type which represents a compound data structure with an arbitrary number of elements called fields. The first operand defines the extent of the declaration in terms of the instruction range given by the relative instruction offset. All fields declared within this extent belong to the most current record declaration, see Section 23.4.19. The second operand defines the overall size of the data structure.

23.4.38 `ref` Reference Type Declaration

Declares a reference type which represents the address of a storage object that can be dereferenced. This compound type declaration requires a subsequent declaration for the type of the referenced storage object.

23.4.39 `req` *String* Name Requirement

Adds a dependency on a data or code section that is not explicitly referenced by any other instruction of the current section. The operand specifies the name of the required section.

23.4.40 `res` *Size* Space Reservation

Enlarges the current data section by the specified amount without initializing the resulting memory.

23.4.41 `ret` Return from Function

Pops the function pointer from the stack and resumes the execution of the calling code section. For more information about stack operations, see Section 23.2.5.

23.4.42 `rsh` *RegMem*, *ImmRegMem*, *ImmRegMem* — Right Shift

Performs a right shift on the bits of the value of the second operand by the amount specified by the third operand and stores the result in the first register or memory operand. The shift operation is arithmetic for signed integers and logical otherwise. This operation is undefined if the third operand is negative or greater than or equal to its size in bits, and is not available for pointers and floating-point numbers.

23.4.43 `sub` *RegMem*, *ImmRegAdrMem*, *ImmRegAdrMem* — Subtraction

Subtracts the value of the third operand from the value of the second operand and stores the result in the first register or memory operand. This operation is not available for function pointers.

23.4.44 `sym` *Offset*, *String*, *ImmRegMem* — Symbol Declaration

Declares a symbol which associates the symbolic name of a storage object with its value or memory address. The first operand defines the lifetime of the symbol in terms of the instruction range given by the relative instruction offset. The second operand specifies the name of the symbol where an empty name conventionally denotes the result of the code section rather than a local variable or parameter. The model of the third operand describes the kind of the symbol:

- An immediate value operand denotes a constant with the value of the operand.
- A register operand declares an alias name for the currently mapped physical register of the operand and automatically fixes its mapping during the lifetime of the symbol.
- A memory operand denotes a storage object at the address of the operand.

In all three cases the symbol declaration is also available as an additional constant definition in the assembly code of all `asm` instructions occurring within the lifetime of the symbol. Symbol declarations require subsequent source code locations and type declarations.

23.4.45 `trap` *Size* — Abnormal Program Termination

Terminates the program abnormally due to an exceptional condition like an unsatisfied runtime check. The operand provides implementation-defined information for a debugger in cases where its value can be encoded in the generated machine code.

23.4.46 `type` *StrTyp* — Basic Type Declaration

Declares a basic type which corresponds to the type of the named type section or the operand itself.

23.4.47 `unfix` *Register* — Unfix Register Mapping

Unfixes the current mapping of a general-purpose register which was previously fixed by a corresponding `fix` instruction, see Section 23.4.21.

23.4.48 `value` *String*, *Imm* — Enumerator Declaration

Declares an enumerator with a symbolic name and a constant value for the most current enumeration declaration, see Section 23.4.18. The first operand specifies the name of the enumerator and the second operand its value. Enumerator declarations require subsequent source code locations.

23.4.49 `void` — Void Type Declaration

Declares a void type which represents an unspecified, ambiguous, or nonexistent type of the programming language.

23.4.50 `xor` *RegMem*, *ImmRegMem*, *ImmRegMem* — Logical Exclusive OR

Performs a logical exclusive OR operation on the bits of the values of the second and third operand and stores the result in the first register or memory operand. This operation is not available for pointers and floating-point numbers.

23.5 Code Optimizations

The Eigen Compiler Suite features an optimizer that is able to improve programs represented using intermediate code. This section describes all improvements and modifications on programs performed by the optimizer. Since the intermediate code representation of a program is the only interface between the various front-ends and back-ends, all compilers of the Eigen Compiler Suite benefit from these optimizations by design. Figure 23.3 shows the typical data flow within an optimizing compiler.

Memory operands followed by an exclamation point denote strict data accesses which are not subject to code optimizations. This ensures accesses to memory that is considered volatile because it is shared with peripheral devices, signal handlers, or other threads of execution. In general however, memory accesses are neither atomic nor ordered and should therefore never be used as a synchronization mechanism.

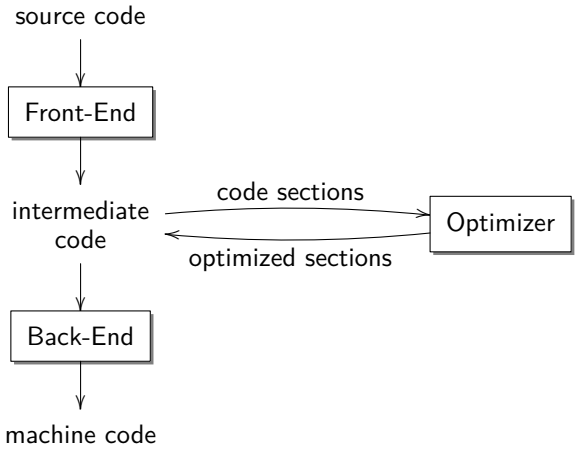


Figure 23.3: Data flow within an optimizing compiler

23.6 Runtime Support

The Eigen Compiler Suite provides an interpreter called `cdrun` which executes programs written in intermediate code. Basic runtime support for the emulated runtime environment is provided by the assembly file `coderun`. Programs written in C++ need additional runtime support stored in the `cppcoderun` assembly file. Programs written in Oberon need additional runtime support stored in the `obcoderun` assembly file. For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

23.7 Intermediate Code Tools

The Eigen Compiler Suite provides several different tools that generate or process files written in intermediate code. While real compilers use intermediate code just as an internal and therefore inaccessible representation of the code to be compiled, these tools allow exposing and modifying the actual intermediate code for debugging purposes. For this reason, all optimizations mentioned in Section 23.5 are disabled by default. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed

whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

23.7.1 cdcheck

`cdcheck` is a syntactic and semantic checker for intermediate code. It just performs syntactic and semantic checks on programs written in intermediate code and writes its diagnostic messages to the standard error stream.

For more information about the generic assembly language and how to use it, see Chapter 8.

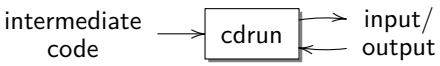
23.7.2 cdopt

`cdopt` is an optimizer for intermediate code. It performs various optimizations on programs written in intermediate code and writes the result to the standard output stream.

For more information about the generic assembly language and how to use it, see Chapter 8.

23.7.3 cdrun

`cdrun` is an interpreter for intermediate code. It processes and executes programs written in intermediate code. The following code sections are predefined and have the usual semantics: `abort`, `_Exit`, `fflush`, `floor`, `fputc`, `free`, `getchar`, `malloc`, and `putchar`. Diagnostic messages about invalid operations include the name of the executed code section and the index of the erroneous instruction.

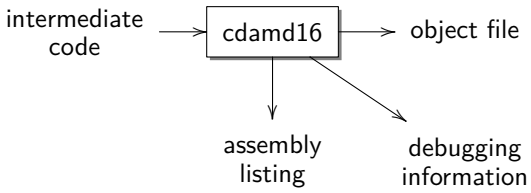


For more information about the generic assembly language and how to use it, see Chapter 8.

23.7.4 cdamd16

`cdamd16` is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code

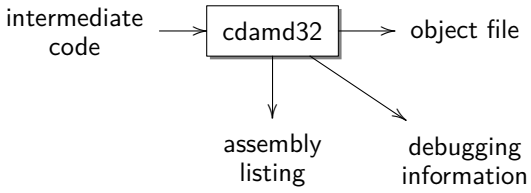
and stores it in corresponding object files. The compiler generates machine code for the 16-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.5 cdamd32

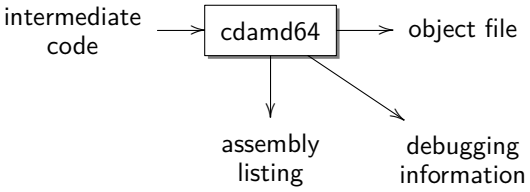
`cdamd32` is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.6 cdamd64

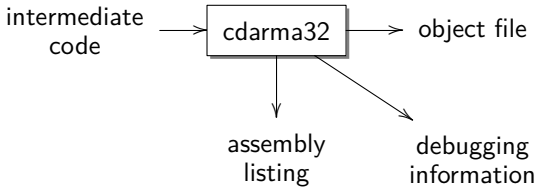
`cdamd64` is a compiler for intermediate code targeting the AMD64 hardware architecture. It generates machine code for AMD64 processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the AMD64 architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AMD64 hardware architecture, see Chapter 9. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.7 cdarma32

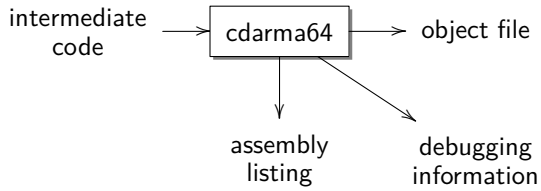
`cdarma32` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors executing A32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.8 cdarma64

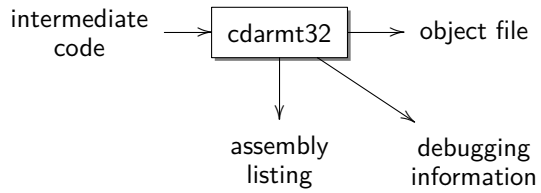
`cdarma64` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors executing A64 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.9 cdarmt32

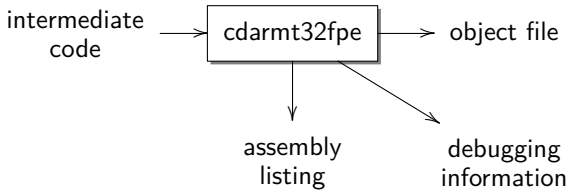
`cdarmt32` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors without floating-point extension executing T32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.10 cdarmt32fpe

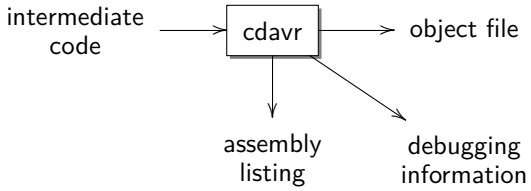
`cdarmt32fpe` is a compiler for intermediate code targeting the ARM hardware architecture. It generates machine code for ARM processors with floating-point extension executing T32 instructions from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the ARM hardware architecture, see Chapter 10. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.11 cdavr

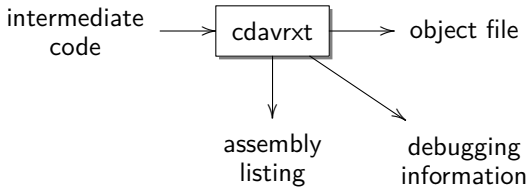
`cdavr` is a compiler for intermediate code targeting the AVR hardware architecture. It generates machine code for AVR processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.12 cdavrxt

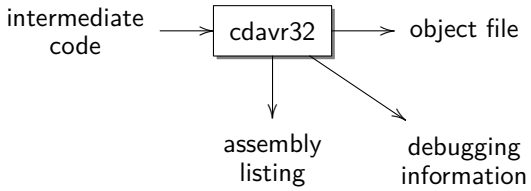
`cdavrxt` is a compiler for intermediate code targeting the AVR hardware architecture. It generates machine code for AVR processors with extended core from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR hardware architecture, see Chapter 11. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.13 cdavr32

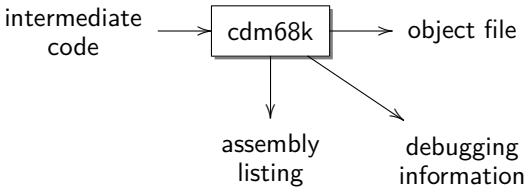
`cdavr32` is a compiler for intermediate code targeting the AVR32 hardware architecture. It generates machine code for AVR32 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the AVR32 hardware architecture, see Chapter 12. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.14 cdm68k

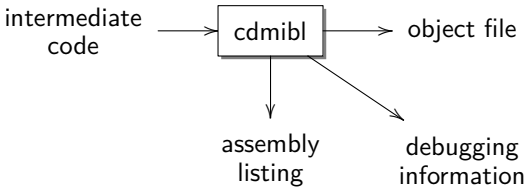
`cdm68k` is a compiler for intermediate code targeting the M68000 hardware architecture. It generates machine code for M68000 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the M68000 hardware architecture, see Chapter 13. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.15 cdmibl

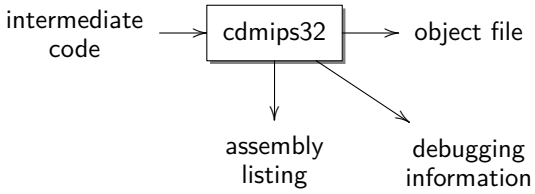
`cdmibl` is a compiler for intermediate code targeting the MicroBlaze hardware architecture. It generates machine code for MicroBlaze processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MicroBlaze hardware architecture, see Chapter 14. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.16 cdmips32

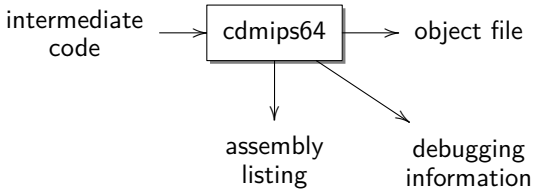
`cdmips32` is a compiler for intermediate code targeting the MIPS32 hardware architecture. It generates machine code for MIPS32 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.17 cdmips64

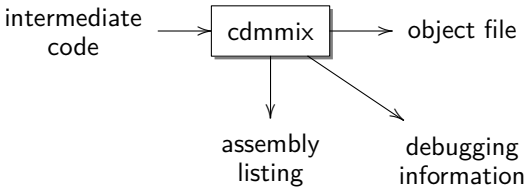
`cdmips64` is a compiler for intermediate code targeting the MIPS64 hardware architecture. It generates machine code for MIPS64 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MIPS32 and MIPS64 hardware architectures, see Chapter 15. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.18 cdmmix

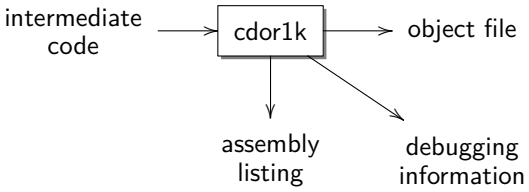
`cdmmix` is a compiler for intermediate code targeting the MMIX hardware architecture. It generates machine code for MMIX processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the MMIX hardware architecture, see Chapter 16. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.19 cdor1k

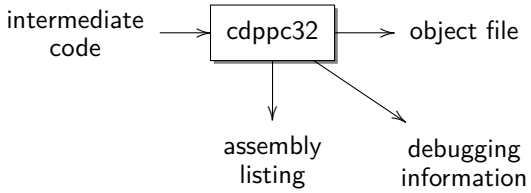
`cdor1k` is a compiler for intermediate code targeting the OpenRISC 1000 hardware architecture. It generates machine code for OpenRISC 1000 processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the OpenRISC 1000 hardware architecture, see Chapter 17. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.20 cdppc32

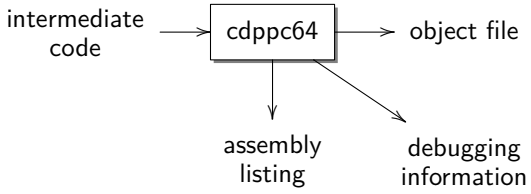
`cdppc32` is a compiler for intermediate code targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 32-bit operating mode defined by the PowerPC architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.21 cdppc64

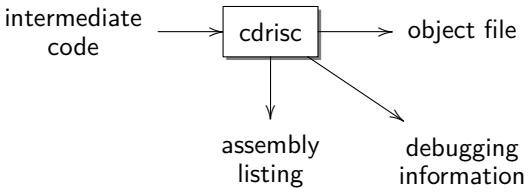
`cdppc64` is a compiler for intermediate code targeting the PowerPC hardware architecture. It generates machine code for PowerPC processors from programs written in intermediate code and stores it in corresponding object files. The compiler generates machine code for the 64-bit operating mode defined by the PowerPC architecture. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the PowerPC hardware architecture, see Chapter 18. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.22 cdrisc

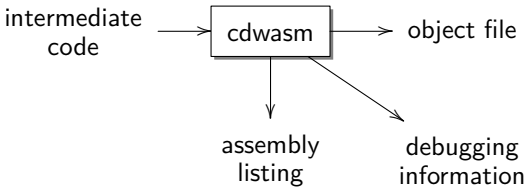
`cdrisc` is a compiler for intermediate code targeting the RISC hardware architecture. It generates machine code for RISC processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the RISC hardware architecture, see Chapter 19. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.23 cdwasm

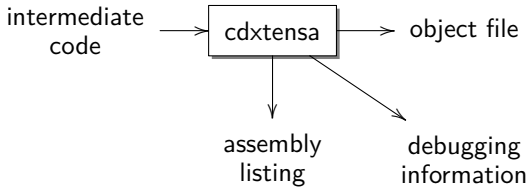
`cdwasm` is a compiler for intermediate code targeting the WebAssembly architecture. It generates machine code for WebAssembly targets from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the WebAssembly architecture, see Chapter 20. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.24 cdxtensa

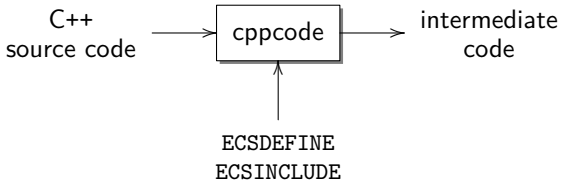
`cdxtensa` is a compiler for intermediate code targeting the Xtensa hardware architecture. It generates machine code for Xtensa processors from programs written in intermediate code and stores it in corresponding object files. It also creates debugging information files as well as assembly files containing listings of the generated machine code.



For more information about the generic assembly language and how to use it, see Chapter 8. For more information about how the Eigen Compiler Suite supports the Xtensa hardware architecture, see Chapter 21. For more information about object files and their purpose, see Chapter 22. For more information about debugging information and its representation, see Chapter 24.

23.7.25 cppcode

`cppcode` is an intermediate code generator for the C++ programming language. It generates intermediate code from programs written in C++ and stores it in corresponding assembly files. The macro `__code__` is predefined in order to enable programmers to identify this tool while generating intermediate code. Programs generated with this tool require additional runtime support that is stored in the `cppcoderun` assembly file.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5. For more information about the generic assembly language and how to use it, see Chapter 8.

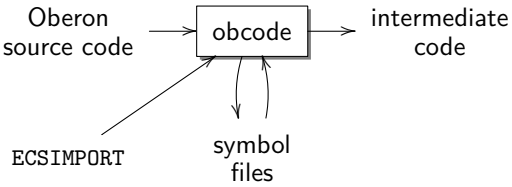
23.7.26 falcode

`falcode` is an intermediate code generator for the FALSE programming language. It generates intermediate code from programs written in FALSE and stores it in corresponding assembly files.

For more information about the FALSE programming language and its implementation by the Eigen Compiler Suite, see Chapter 6. For more information about the generic assembly language and how to use it, see Chapter 8.

23.7.27 obcode

`obcode` is an intermediate code generator for the Oberon programming language. It generates intermediate code from modules written in Oberon and stores it in corresponding assembly files. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module. Programs generated with this tool require additional runtime support that is stored in the `obcoderun` assembly file.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7. For more information about the generic assembly language and how to use it, see Chapter 8.

24 Debugging Information

This chapter describes the debugging information generated by the compilers of the Eigen Compiler Suite and its open file format. It additionally describes the functionality and interface of related tools provided by the Eigen Compiler Suite for debugging purposes.

Every failure is a step to success.

— William Whewell

24.1 Introduction

Although the Eigen Compiler Suite does not provide its own debugger tool, its compiler tools do collect and store *debugging information* for external debuggers. The debugging information generated alongside an object file consists of an abstract representation of all programming language constructs like functions, variables, data types, and statements compiled into the object file. It is designed to enable debuggers to support *program animation* and *memory inspection*. For this purpose, the Eigen Compiler Suite provides converter tools which generate a binary representation of the debugging information and store it in specific debugging data formats using additional object files as shown in Figure 24.1. The resulting debugging object files can later be optionally linked together with the original object file. For more information about object files and their purpose, see Chapter 22.

Converting debugging information into separate object files effectively decouples a compiler from the debugger of the target runtime environment and its required debugging data format. It also ensures that compilers always generate the same output regardless of whether the resulting binary executable is subject to debugging or not. The following sections describe the semantics of the debugging information representation alongside the syntax of its textual file format.

24.2 Debugging Information Structure

The debugging information generated by the various compiler tools of the Eigen Compiler Suite always consists of a short description of their target hardware architecture. It also contains a list of *information entries* which are generic representations of programming language constructs like functions, variables, or data types.

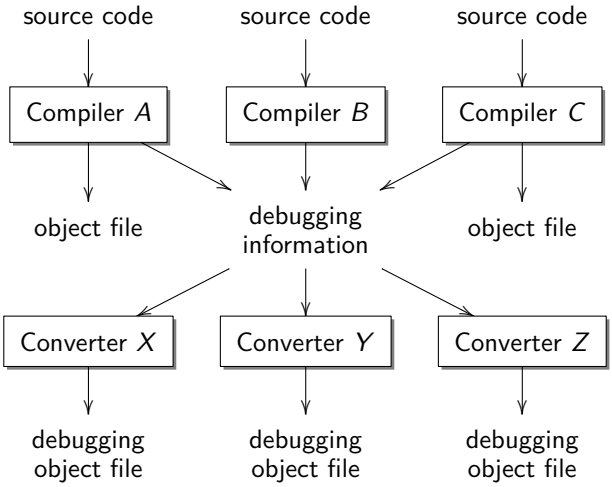


Figure 24.1: Debugging information representation of programs in-between compilers and converters

24.2.1 Information Entries

Each information entry has a unique name and a *source code location* referring to its textual declaration in the source code. The Eigen Compiler Suite supports the following kinds of information entries:

- Code Entry `code`

  A code entry represents a functional unit of the programming language like a function or a procedure and corresponds to the code section in the object file with the same name. It contains a *type declaration* which describes the memory layout and data representation of the returned result. It additionally holds a list of *symbol declarations* which describe the storage objects declared by this information entry. It also contains a list of *breakpoints* which associate source code locations of statements with the corresponding machine code offsets in the code section.

- Data Entry `data`

  A data entry represents a data unit of the programming language like a global variable and corresponds to the data section in the object file with the same name. It contains a type declaration for the storage object represented by the data section.

- Type Entry `type`

  A type entry represents a named type definition of the programming language. It contains a type declaration and may omit the source code location for predefined types.

24.2.2 Symbol Declarations

A symbol declaration refers to a single storage object managed by a code section like a local variable or parameter. It has a unique name and a source code location referring to its corresponding declaration in the source code. It also contains a type declaration for the data type of its storage object and a description of its *lifetime* in terms of a range of offsets in the code section wherein the symbol is considered alive. The Eigen Compiler Suite supports the following kinds of symbol declarations:

- Constant Declaration

  A constant declaration refers to a constant or storage object that is not supposed to change its value. Since a constant declaration must not necessarily refer to an actually existing storage object, it is represented using its constant value.

- Register Declaration

  A register declaration refers to a storage object which is stored in a register. The register of the target hardware architecture is represented using its name.

- Variable Declaration

  A variable declaration refers to a storage object that has an actual memory address. The address refers either to a data section or to a displaced register that typically names the frame pointer.

24.2.3 Type Declarations

A type declaration describes the layout and data representation of memory regions occupied by storage objects. The Eigen Compiler Suite supports the following kinds of type declarations:

- Void Type `void`

  A void type represents an unspecified, ambiguous, or nonexistent type of the programming language.

- Type Name *Name*

  A type name refers to the corresponding type entry, see Section 24.2.1.

- Signed Integer Type `signed`

  A signed integer type represents a storage object with a given size that stores a single signed integer value using two's complement as signed magnitude representation.

- Unsigned Integer Type `unsigned`

  An unsigned integer type represents a storage object with a given size that stores a single unsigned integer value.

- Floating-point Number Type `float`

  A floating-point number type represents a storage object with a given size that stores a single floating-point number value according to formats defined in the IEEE standard for floating-point arithmetic [27].

- Enumeration Type `enumeration`

  An enumeration type represents a storage object that stores a single value from a predefined set of named constants called *enumerators*. It contains the type of the underlying type and declarations for each enumerator.

- Array Type `array`

  An array type represents a collection of consecutive storage objects with the same type called its elements. It contains the index of its first element, the number of elements in case the array has a static size, and a declaration for the element type.

- Record Type `record`

  A record type describes a user-defined data structure that contains an arbitrary number of storage objects called fields. It contains an overall size of the data structure and declarations for each field.

- Pointer Type `pointer`

  A pointer type represents a dereferencable storage object that stores a potentially invalid address of another storage object. It contains the type of the referenced storage object.

- Reference Type `reference`

  A reference type represents a dereferencable storage object that stores a valid address of another storage object. It contains the type of the referenced storage object.

- Function Type `function`

  A function type represents the type of a functional unit of the programming language. It contains type declarations for the returned result and each function parameter.

24.3 Debugging Information Format

The debugging information generated by compiler tools is stored in plain text files according to the complete syntax specification given in Figure 24.2. A debugging information file consists of a description of its target as well as an arbitrary number of sources and information entries according to the following syntax:

Information = *Target Sources* $Entries_{opt}$
Sources = *Source* | *Sources Source*
Source = double-quoted-string

The sequence of sources lists all filenames used during compilation and is referenced by index in source code locations.

24.3.1 Target Description

The description of the target architecture is represented in the debugging information file as text according to the following syntax:

Target = *Name Endianness Pointer*
Name = double-quoted-string
Endianness = `little` | `big`
Pointer = *Size*
Size = decimal-integer

The description specifies the unique name of the target architecture as well as its endianness and pointer size expressed in octets.

24.3.2 Information Entries

Information entries are represented in the debugging information file as text according to the following syntax:

Entries = *Entry* | *Entries Entry*
Entry = `code` *Name Location Type Size* $Symbols_{opt}$ $Breakpoints_{opt}$ |
`data` *Name Location Type Size* |
`type` *Name* $Location_{opt}$ *Type*
Name = double-quoted-string
Size = decimal-integer

The valid identifiers for the kind of the information entry correspond to the information entries described in Section 24.2.1. The size of an entry is expressed in octets.

Information = *Target Sources* $Entries_{opt}$
Target = *Name Endianness Pointer*
Name = double-quoted-string
Endianness = `little` | `big`
Pointer = *Size*
Size = decimal-integer
Sources = *Source* | *Sources Source*
Source = double-quoted-string
Entries = *Entry* | *Entries Entry*
Entry = `code` *Name Location Type Size* $Symbols_{opt}$ $Breakpoints_{opt}$ |
`data` *Name Location Type Size* |
`type` *Name* $Location_{opt}$ *Type*
Location = *Index Line Column*
Index = decimal-integer
Line = decimal-integer
Column = decimal-integer
Type = `void` | *Name* | `signed` *Size* | `unsigned` *Size* | `float` *Size* |
`array` *Index Size Type* | `record` *Size* $Fields_{opt}$ |
`pointer` *Type* | `reference` *Type* |
`function` *Size Type* $Parameters_{opt}$ |
`enumeration` *Type Enumerators*
Fields = *Field* | *Fields Field*
Field = *Name* $Location_{opt}$ *Type Offset Bitmask*
Offset = decimal-integer
Bitmask = decimal-integer
Parameters = *Parameter* | *Parameters Parameter*
Parameter = *Type*
Enumerators = *Enumerator* | *Enumerators Enumerator*
Enumerator = *Name* $Location_{opt}$ *Value*
Value = `signed` signed-decimal-integer |
`unsigned` decimal-integer |
`float` decimal-floating-point
Symbols = *Symbol* | *Symbols Symbol*
Symbol = *Name Location Kind Type Lifetime*
Kind = *Constant* | *Register* | *Variable*
Constant = *Value*
Register = *Name*
Variable = *Displacement Register*
Displacement = signed-decimal-integer
Lifetime = *Begin End*
Begin = *Offset*
End = *Offset*
Breakpoints = *Breakpoint* | *Breakpoints Breakpoint*
Breakpoint = *Offset Location*

Figure 24.2: Syntax of the debugging information file format

24.3.3 Source Code Locations

Source code locations associated with information entries, symbol declarations, field declarations, enumerator declarations, and breakpoints are represented in the debugging information file as text according to the following syntax:

Location = *Index Line Column*
Index = decimal-integer
Line = decimal-integer
Column = decimal-integer

The zero-based index refers to one of the sources listed at the beginning of the debugging information in order of occurrence. Line and column numbering starts with one where all white-space characters are counted as single characters.

24.3.4 Symbol Declarations

Symbol declarations are represented in the debugging information file as text according to the following syntax:

Symbols = *Symbol* | *Symbols Symbol*
Symbol = *Name Location Kind Type Lifetime*
Name = double-quoted-string
Kind = *Constant* | *Register* | *Variable*
Constant = *Value*
Value = `signed` signed-decimal-integer |
`unsigned` decimal-integer |
`float` decimal-floating-point
Register = *Name*
Variable = *Displacement Register*
Displacement = signed-decimal-integer
Lifetime = *Begin End*
Begin = *Offset*
End = *Offset*
Offset = decimal-integer

The valid kinds of symbols correspond to the symbol declarations described in Section 24.2.2. The offsets of symbol lifetimes refer to machine code instructions relative to the beginning of the code section corresponding to the enclosing code entry and are expressed in octets. A symbol declaration with an empty name conventionally denotes the result of the code section rather than a local variable or parameter. A variable with a displacement of zero names a data section rather than a register whereas a positive displacement denotes a parameter.

24.3.5 Type Declarations

Type declarations are represented in the debugging information file as text according to the following syntax:

Type = `void` | *Name* | `signed` *Size* | `unsigned` *Size* | `float` *Size* |
`array` *Index Size Type* | `record` *Size* $Fields_{opt}$ |
`pointer` *Type* | `reference` *Type* |
`function` *Size Type* $Parameters_{opt}$ |
`enumeration` *Type Enumerators*

Name = double-quoted-string

Size = decimal-integer

Index = decimal-integer

Parameters = *Parameter* | *Parameters Parameter*

Parameter = *Type*

The valid kinds of types correspond to the type declarations described in Section 24.2.3. The size of a type is expressed in octets except for array types where it denotes the number of array elements, and for function types where it denotes the number of parameters.

24.3.6 Field Declarations

Field declarations are represented in the debugging information file as text according to the following syntax:

Fields = *Field* | *Fields Field*

Field = *Name* $Location_{opt}$ *Type Offset Bitmask*

Name = double-quoted-string

Offset = decimal-integer

Bitmask = decimal-integer

The offset specifies the address of the storage object relative to the beginning of the enclosing data structure and is expressed in octets. A non-zero bitmask declares a bit field that partially occupies the storage object using the consecutive sequence of masked bits given by its binary value. A field declaration with an empty name conventionally denotes inheritance rather than composition.

24.3.7 Enumerator Declarations

Enumerator declarations are represented in the debugging information file as text according to the following syntax:

Enumerators = *Enumerator* | *Enumerators Enumerator*
Enumerator = *Name* $Location_{opt}$ *Value*
Name = double-quoted-string
Value = `signed` signed-decimal-integer |
`unsigned` decimal-integer |
`float` decimal-floating-point

The constant value of an enumerator belongs to the underlying type of the corresponding enumeration.

24.3.8 Breakpoints

Breakpoints are represented in the debugging information file as text according to the following syntax:

Breakpoints = *Breakpoint* | *Breakpoints Breakpoint*
Breakpoint = *Offset Location*
Offset = decimal-integer

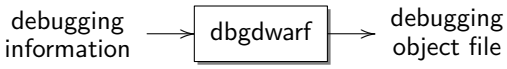
The offset refers to a machine code instruction relative to the beginning of the code section corresponding to the enclosing code entry and is expressed in octets.

24.4 Debugging Tools

Debugging tools process debugging information and other files that were previously generated by the various compiler and linker tools of the Eigen Compiler Suite for debugging purposes. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

24.4.1 dbgdwarf

`dbgdwarf` is a DWARF debugging information converter tool. It converts debugging information into the DWARF debugging data format and stores it in corresponding object files [1]. The resulting debugging object files can be combined with runtime support that creates Executable and Linking Format (ELF) files [2].



For more information about object files and their purpose, see Chapter 22.

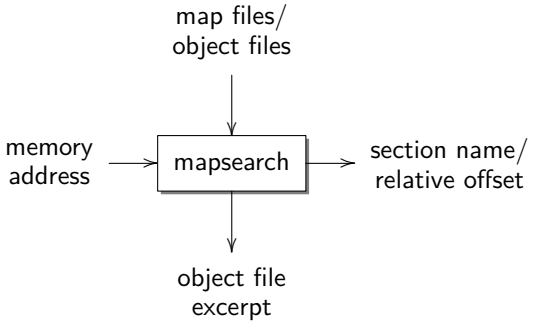
24.4.2 mapplace

`mapplace` is a map tool provided for debugging purposes. It converts map files generated by linker tools into an object file called map place result by placing all listed binary sections at their corresponding address using fixed phantom sections. Map place results can be linked together with debugging object files to generate separate symbol files for external debuggers.

For more information about object files and their purpose, see Chapter 22.

24.4.3 mapsearch

`mapsearch` is map tool provided for debugging purposes. It searches map files generated by linker tools for the name of a binary section that encompasses a memory address read from the standard input stream. If additionally provided with one or more object files, it also stores an excerpt thereof in a separate object file called map search result which only contains the identified binary section for disassembling purposes.



For more information about object files and their purpose, see Chapter 22.

25 Generic Documentation

This chapter describes the generic documentation facility used by the Eigen Compiler Suite in order to automatically generate consistent documentations from source code written in different programming languages.

Learn as much by writing as by reading.

— John Dalberg-Acton

25.1 Introduction

Almost all programming languages allow programmers to annotate their code with additional notes and comments. Oftentimes these annotations are part of the interface exposed to users of libraries or applications. The Eigen Compiler Suite provides a facility for extracting this information in order to automatically generate detailed and appealing documentations about this interface. This framework is called *generic documentation generation*.

The core of this framework consists of a generic representation of the program interface as well as the source code annotations provided by the programmer. The Eigen Compiler Suite provides so-called *documentation extractors* for some of the supported programming languages. They extract the structure and comments of source code and create a consistent documentation of it using the generic documentation representation. So called *documentation formatters* on the other hand are able to transform the generic representation of this documentation into documents of different formats. Figure 25.1 visualizes the role and data flow of the generic documentation representation in-between extractors and formatters.

In addition to the automatically extracted information, the framework is also able to format the annotations given by the programmer. For this purpose, the framework defines a lightweight markup language which is an extension of the Creole markup language used for formatting wikis [10]. Table 25.1 categorizes all elements of this markup language and shows how they are formatted.

25.2 Generic Documentation Representation

Generic documentations are represented using a hierarchical structure of *documents*, *articles*, *paragraphs*, and *texts*. Figure 25.2 visualizes this hierarchy.

Category	Markup	Formatting
Emphasis	`//italics//`	*italics*
	`**bold**`	**bold**

Lists	`# ordered list`
	`# second item`
	`## sub item`
		1 ordered list
		2 second item
		1 sub item

	`* unordered list`
	`* second item`
	`** sub item`
		• unordered list
		• second item
		• sub item

Links	`link to [[label]]`	link to label
	`<<label>> definition`	definition
	`[[url|link name]]`	link name
Headings	`= large heading`	large heading
	`== medium heading`	medium heading
	`=== small heading`	small heading

Line breaks	`no line break`
	`for paragraphs`
	
	`use empty line`
		no line break for paragraphs
		
		use empty line

	`forced\\line break`	forced line break
Lines	`--`	

Tables	`|=a|=table|=header`
	`|a|table|row`
	`|b|table|row`
		a table header
		a table row
		b table row

Code	`{{{`
	`unformatted`
	`code`
	`}}}`
		`unformatted`
		`code`

Articles	`@ new article`	1. new article
	`@ second article`	2. second article
	`@@ sub article`	2.1. sub article

Table 25.1: Elements of the generic documentation markup language

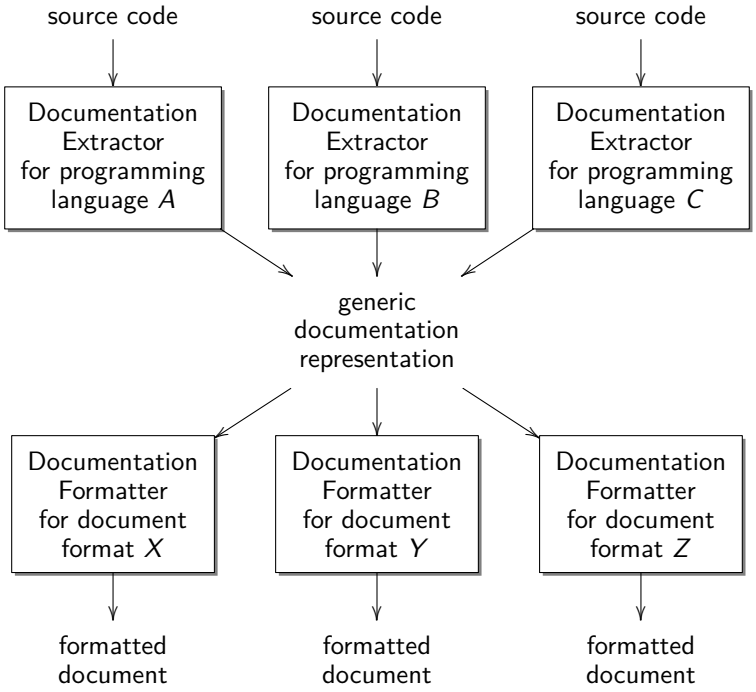


Figure 25.1: Extracting and formatting generic documentation

The remainder of this section describes the syntax of the markup language that is used to represent generic documentations. The complete syntax specification is given in Figure 25.3.

25.2.1 Documents

Each document consists of a general description of its contents and a list of articles. Documents are stored in plain text files according to the following syntax:

$$
\begin{aligned}
\mathit{Documentation} &= \mathit{Documents}_{opt} \\
\mathit{Documents} &= \mathit{Document} \mid \mathit{Documents}\ \mathit{Document} \\
\mathit{Document} &= \mathit{Description}\ \mathit{Articles}_{opt} \\
\mathit{Description} &= \mathit{Paragraphs}_{opt}
\end{aligned}
$$

If the documentation consists of several documents, the paragraphs of their descriptions are merged and placed at the beginning of the generated documentation.

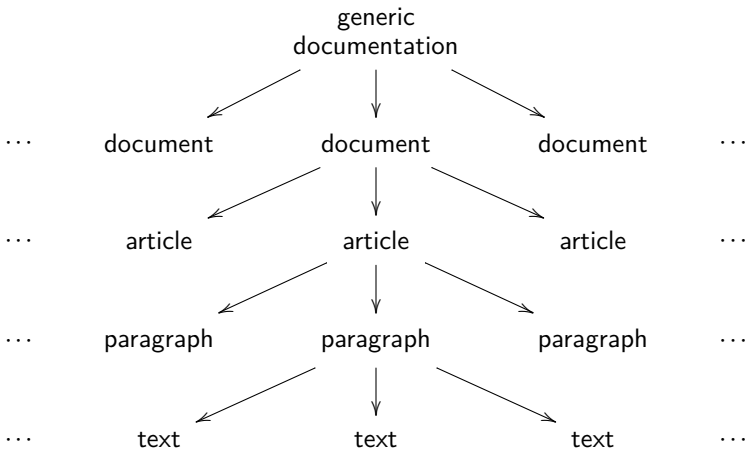


Figure 25.2: Structure of generic documentations

25.2.2 Articles

Articles allow giving more detailed information and correspond to structured sections in traditional documents. They are represented according to the following syntax:

$$
\begin{array}{lll}
\textit{Articles} & = & \textit{Article} \mid \textit{Articles Article} \\
\textit{Article} & = & \textit{Article-Marker Title Paragraphs}_{opt} \\
\textit{Article-Marker} & = & \texttt{@} \mid \texttt{@@} \mid \texttt{@@@} \mid \texttt{@@@@} \mid \ldots \\
\textit{Title} & = & \textit{Text}
\end{array}
$$

The title of an article corresponds to the caption of a section within a traditional document. In generated documentations, they are additionally listed in the table of contents. The number of at signs in the marker of an article specifies the corresponding nesting level of the article. The difference of the nesting levels of two consecutive articles within the same document may not be higher than one.

25.2.3 Paragraphs

Paragraphs are text blocks that have a specific type which specifies how the text is formatted. They are represented according to the following syntax:

$$
\textit{Paragraph} = \textit{Text-Block} \mid \textit{Heading} \mid \textit{List-Item} \mid \textit{Code-Block} \mid \textit{Table} \mid \textit{Line}
$$

The type of a paragraph affects the formatting of its contents and may be one of the following:

Documentation = *Documents*$_{opt}$
Documents = *Document* | *Documents Document*
Document = *Description Articles*$_{opt}$
Description = *Paragraphs*$_{opt}$
Articles = *Article* | *Articles Article*
Article = *Article-Marker Title Paragraphs*$_{opt}$
Article-Marker = `@` | `@@` | `@@@` | `@@@@` | ...
Title = *Text*
Paragraph = *Text-Block* | *Heading* | *List-Item* | *Code-Block* | *Table* | *Line*
Text-Block = *Text*
Heading = *Heading-Marker Text*
Heading-Marker = `=` | `==` | `===`
List-Item = *Item-Marker Text*
Item-Marker = *Number-Marker* | *Bullet-Marker*
Number-Marker = `#` | `##` | `###`
Bullet-Marker = `*` | `**` | `***`
Code-Block = `{{{` any text `}}}`
Table = *Rows*
Rows = *Row* | *Rows Row*
Row = *Cells*
Cells = *Cell* | *Cells Cell*
Cell = *Cell-Marker Text*
Cell-Marker = `|` | `|=`
Line = `----`
Text = *Text-Elements*$_{opt}$
Text-Elements = *Text-Element* | *Text-Elements Text-Element*
Text-Element = *Default* | *Italic* | *Bold* | *Link* | *URL* | *Label* | *Code* | *Line-Break*
Default = word
Italic = `//` *Text* `//`
Bold = `**` *Text* `**`
Link = `[[` *Target* `]]` | `[[` *Target* `|` *Text* `]]`
Target = word
URL = `[[` url `]]` | `[[` url `|` *Text* `]]`
Label = `<<` *Target* `>>`
Code = `{{{` *Text* `}}}`
Line-Break = `\\`

Figure 25.3: Syntax of the generic documentation markup language

- Text Block

  Plain text blocks are represented according to the following syntax:

  Text-Block = *Text*

- Heading

  Headings allow authors of an article to logically structure its contents. They are represented according to the following syntax:

  Heading = *Heading-Marker Text*
  Heading-Marker = `=` | `==` | `===`

  The number of equal signs in the marker of a heading specifies the corresponding nesting level of the heading. The difference of the nesting levels of two consecutive headings within the same article may not be higher than one.

- List Item

  List items may be ordered or unordered and are represented according to the following syntax:

  List-Item = *Item-Marker Text*
  Item-Marker = *Number-Marker* | *Bullet-Marker*
  Number-Marker = `#` | `##` | `###`
  Bullet-Marker = `*` | `**` | `***`

  The number of symbols in the marker of a list item specifies the corresponding nesting level of the item within a list. The difference of the nesting levels of two consecutive items within the same list may not be higher than one.

- Code Blocks

  Code text blocks allow marking passages of source code within the documentation. They are represented according to the following syntax:

  Code-Block = `{{{` any text `}}}`

  The code in-between the two markers is not subject to any further processing and is always formatted as is.

- Table

  Tables consist of cell rows and are represented according to the following syntax:

  Table = *Rows*
  Rows = *Row* | *Rows Row*
  Row = *Cells*

Cells = *Cell* | *Cells Cell*
Cell = *Cell-Marker Text*
Cell-Marker = `|` | `|=`

An equal sign immediately following a pipe symbol in the marker of a cell turns the cell into a header.

- Line

  Lines are formatted as horizontal rules and are represented according to the following syntax:

  Line = `----`

Paragraphs and articles may be separated by any number of new-line characters. A single new-line character between two text lines of the same paragraph is considered as a white-space character.

25.2.4 Texts

The contents of most paragraphs as well as the title of an article consist of text. A text is a list of so-called *text elements* which specify the formatting of the text. Text elements may also contain text themselves according to the following syntax:

Text = $\textit{Text-Elements}_{opt}$
Text-Elements = *Text-Element* | *Text-Elements Text-Element*
Text-Element = *Default* | *Italic* | *Bold* | *Link* | *URL* | *Label* | *Code* | *Line-Break*

The formatting style of a text element may be one of the following:

- Default

  The default style does not change the current formatting. Default text elements are just strings according to the following syntax:

  Default = word

- Italic

  Italic text passages are represented according to the following syntax:

  Italic = `//` *Text* `//`

- Bold

  Bold text passages are represented according to the following syntax:

  Bold = `**` *Text* `**`

- Link

  Internal links point to a label defined elsewhere in the generic documentation and are represented according to the following syntax:

  Link = `[[` *Target* `]]` | `[[` *Target* `|` *Text* `]]`
  Target = word

  The formatted text of an internal link equals its target if the alternative text is not given. A link is only valid, if the target names a label that was actually defined in one of the documents of a generic documentation.

- URL

  External links are URLs that point to external resources. They are represented according to the following syntax:

  URL = `[[` url `]]` | `[[` url `|` *Text* `]]`

  The formatted text of an external link equals its target if the alternative text is not given. Currently, only the URL schemes "http:" and "https:" are recognized. Freestanding URLs are automatically turned into external links.

- Label

  Labels name targets for internal links and are represented according to the following syntax:

  Label = << *Target* >>
  Target = word

- Code

  Code text passages are represented according to the following syntax:

  Code = `{{{` *Text* `}}}`

- Line Breaks

  Line breaks request the beginning of a new line. They are represented according to the following syntax:

  Line-Break = \\

25.3 Documentation Formatters

This section lists all document formats supported by the Eigen Compiler Suite and describes how the corresponding documentation formatters process generic documentations.

25.3.1 HTML Documentations

The HTML documentation formatter merges several generic documentations into a single HTML document. This file consists of a merged description of all documentations and all of their articles. In addition, it inserts a linked table of contents using the titles and hierarchical structure of all articles. All articles are formatted the same way and begin with a horizontal rule. At the end of each article, a link to the top of the HTML page is inserted.

25.3.2 Latex Documentations

The Latex documentation formatter merges several generic documentations into a single Latex document. It uses the article document style and uses all flavors of the section and paragraph commands to map the structure of all articles. Headings are formatted using the starred versions of these commands. After the merged description of all generic documentations, the formatter issues a table of contents command. Internal and external links are formatted using the `hyperref` package.

The generated Latex code does not use built-in commands directly but defines and calls wrappers for most of them instead. This allows including the file in larger documents which may provide user-defined versions of these commands beforehand.

25.4 Generic Documentation Tools

The Eigen Compiler Suite provides several different tools that generate or process generic documentations. While real documentation generators use generic documentations just as an internal and therefore inaccessible representation, these tools allow inspecting and modifying the actual documentation that is processed for debugging purposes. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

25.4.1 docprint

`docprint` is a pretty printer for generic documentations. It reformats generic documentations and writes it to the standard output stream.

25.4.2 doccheck

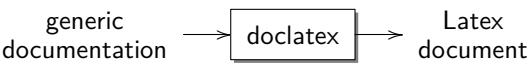
`doccheck` is a syntactic and semantic checker for generic documentations. It just performs syntactic and semantic checks on generic documentations and writes its diagnostic messages to the standard error stream.

25.4.3 dochtml

`dochtml` is an HTML documentation generator for generic documentations. It processes several generic documentations and assembles all information therein into an HTML document.

25.4.4 doclatex

`doclatex` is a Latex documentation generator for generic documentations. It processes several generic documentations and assembles all information therein into a Latex document.



25.5 Documentation Generation Tools

The Eigen Compiler Suite provides several different documentation generators that process commented source code files written in miscellaneous programming languages in order to automatically extract documentations like user guides or reference manuals out of them. Figure 25.4 shows the typical data flow within these documentation generators. For concrete information about how these tools extract the structure of the source code, users have to refer to the documentation of the corresponding programming language. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

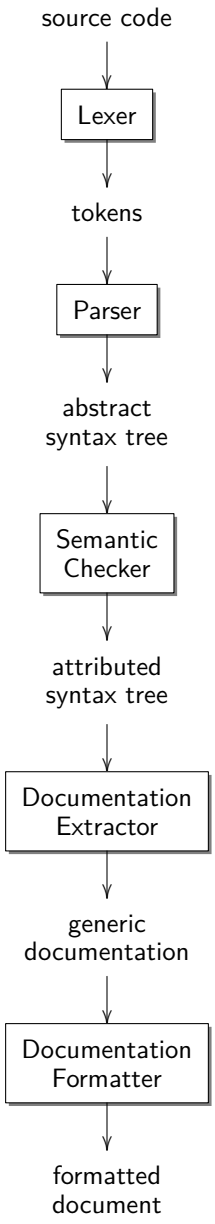
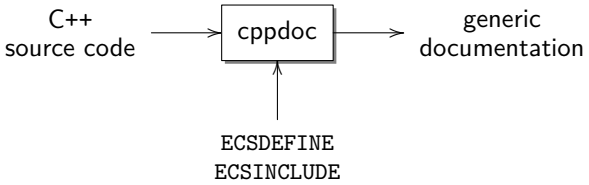


Figure 25.4: Data flow within a typical documentation generator

25.5.1 cppdoc

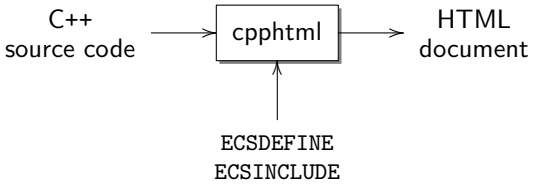
`cppdoc` is a generic documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into a generic documentation.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

25.5.2 cpphtml

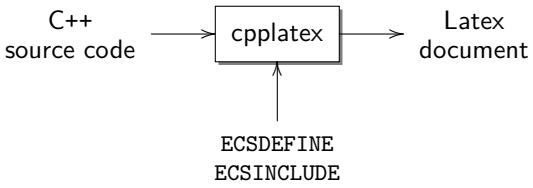
`cpphtml` is an HTML documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into an HTML document.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

25.5.3 cpplatex

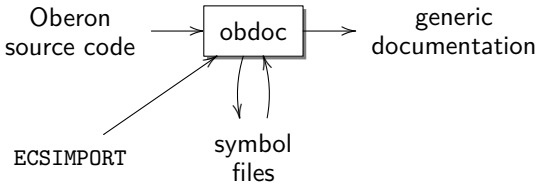
`cpplatex` is a Latex documentation generator for the C++ programming language. It processes several C++ source files and assembles all information therein into a Latex document.



For more information about the C++ programming language and its implementation by the Eigen Compiler Suite, see Chapter 5.

25.5.4 obdoc

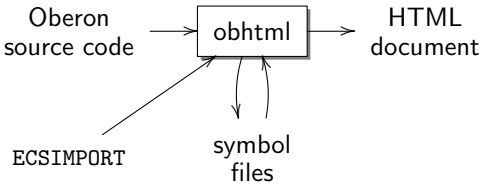
`obdoc` is a generic documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into a generic documentation. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

25.5.5 obhtml

`obhtml` is an HTML documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into an HTML document. In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.

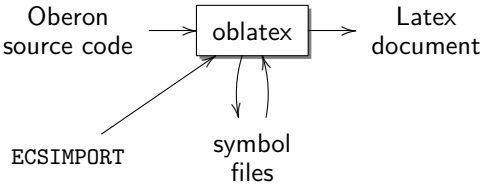


For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

25.5.6 oblatex

`oblatex` is a Latex documentation generator for the Oberon programming language. It processes several Oberon modules and assembles all information therein into a Latex document.

In addition, it stores the interface of each module in a symbol file which is required when other modules import the module.



For more information about the Oberon programming language and its implementation by the Eigen Compiler Suite, see Chapter 7.

26 Extensions

The Eigen Compiler Suite features a variety of tools like compilers, assemblers, and linkers. All of these tools implement different programming languages, target different hardware architectures, or support different runtime environments respectively. This chapter describes the abstractions and utilities provided by the Eigen Compiler Suite that facilitate the development of additional tools and runtime environments.

Nought may endure but mutability.

— Percy Bysshe Shelley

26.1 Introduction

The Eigen Compiler Suite is a complete tool chain targeting a variety of programming languages, hardware architectures, and runtime environments. Internally, it features several abstractions that are helpful for programmers implementing these tools. Figure 26.1 shows all abstractions and visualizes possible combinations of the different tools.

The goal of the abstractions is to enable and simplify all possible combinations of the different aspects of the tool chain. This is achieved by reducing the required programming interfaces to a minimum. As a result, adding support for another programming language, hardware architecture, or runtime environment is also simplified. The following sections describe the steps that are required to extend the Eigen Compiler Suite into this direction.

26.2 Diagnostics

Compilers and assemblers as provided by the Eigen Compiler Suite translate source code written by programmers. Therefore, source code may contain syntax or semantic errors which have to be diagnosed by these tools. The Eigen Compiler Suite provides a generic diagnostics facility for programmers that enables a consistent reporting of errors, warnings, and additional information. Besides the actual text, diagnostic messages also include the name of the diagnosed source code and the position therein. Unless otherwise specified, the source code position is based on text lines.

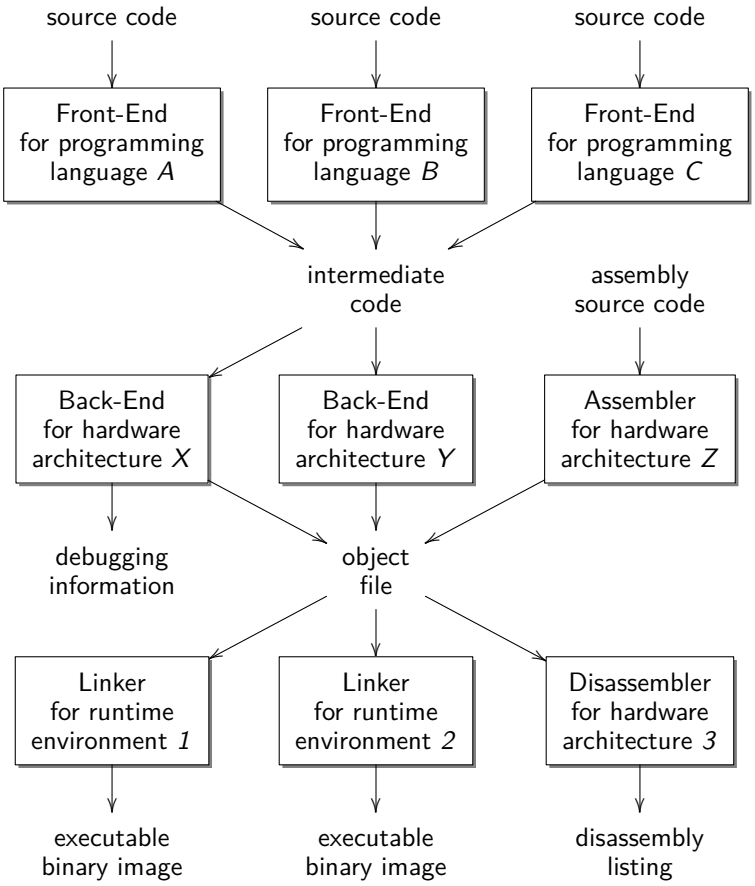


Figure 26.1: The main abstractions of the Eigen Compiler Suite

26.3 Application Drivers

The Eigen Compiler Suite provides a generic driver facility for programmers that allows all applications written with it to share the same user interface. This framework covers the overall error handling as well as the processing scheme for all kinds of input given to the applications. All tools accept command-line arguments which are taken as names of plain text files containing the source code. If no arguments are provided, the standard input stream is used instead. Output files are generated in the current working directory and have the same name as the input file being processed whereas the filename extension gets replaced by an appropriate suffix. See Chapter 2 for more information about the common user interface of all of these tools.

26.4 Programming Languages

This section describes the tools and representations typically provided by the Eigen Compiler Suite for a specific programming language and how they are implemented. Figure 26.2 shows the tools and representations of typical implementations of programming languages.

The source code of a program written in a programming language is most often represented in an abstract syntax tree. This tree is created by a *parser* that recognizes the syntax of the programming language. Parsers typically use a so-called *lexer* that is able to tokenize the source code and extract symbols like keywords and operators. The abstract syntax tree is the base for all tools described in the following sections. Therefore, all of these tools contain a parser and a lexer in order to create the syntax tree. Although these tools use the information represented in the syntax tree for different purposes, they all traverse it in one or several consecutive stages.

26.4.1 Pretty Printers

Pretty printers just traverse the complete abstract syntax tree by reconverting its nodes into tokens again. These tokens are then printed using a consistent and well-arranged layout. Pretty printers usually do not alter the abstract syntax tree and are often able to reformat source code that contains semantic errors.

26.4.2 Semantic Checkers

Semantic checkers traverse the abstract syntax tree and attribute its nodes with semantic information. They are able to diagnose violations of the semantic rules of the programming language. For that purpose, semantic checkers sometimes use additional semantic information that is stored separately. The functionality provided by semantic checkers is in many cases reused by the remaining tools of this section. Standalone semantic checkers are useful for automated testing and verification of the implementation of a programming language.

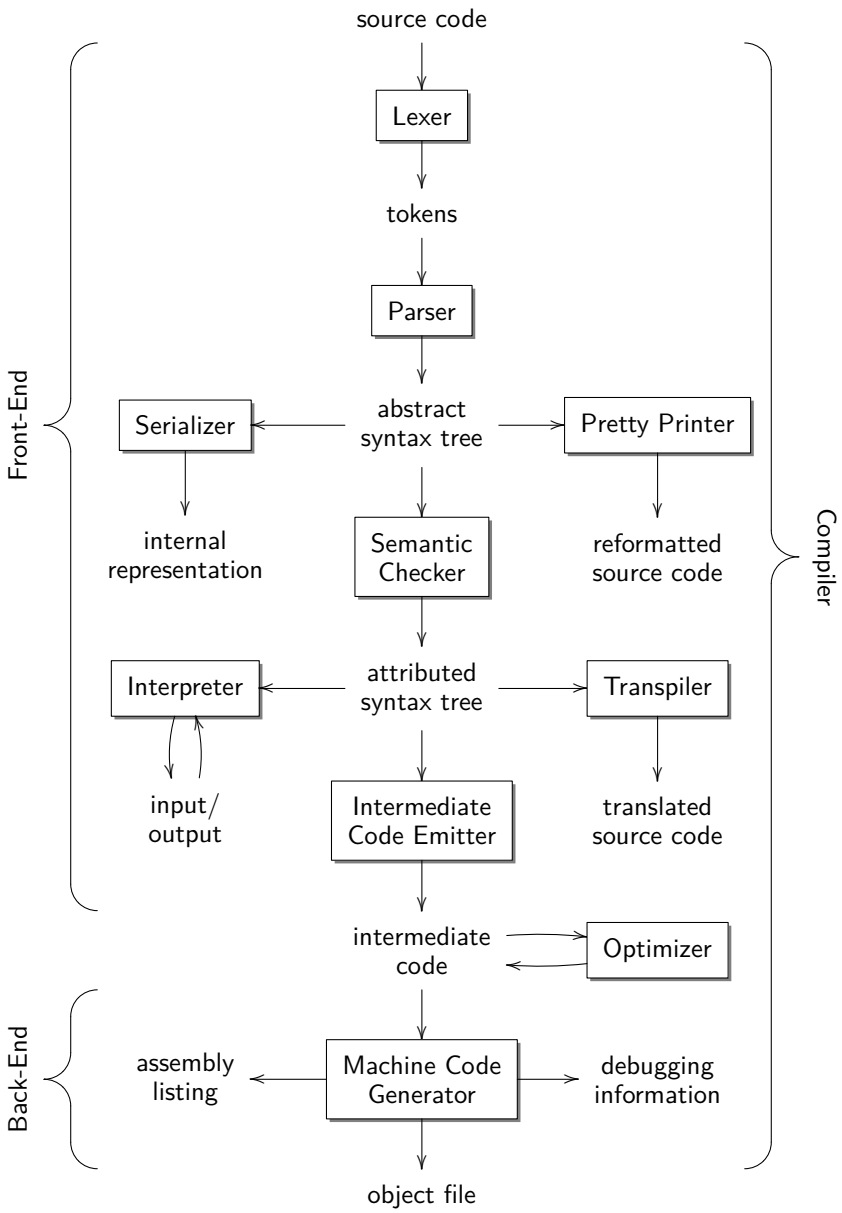


Figure 26.2: Data flow within a typical implementation of a programming language

26.4.3 Serializers

Serializers dump all information stored in the internal representation of a program in a human readable format for debugging purposes. The Eigen Compiler Suite provides a generic serialization facility for representing attributed syntax trees as XML documents. This serialization format has the advantage of being standardized and easy to parse for other development tools potentially making use of the same internal program representation.

26.4.4 Interpreters

Interpreters are able to execute the program given in the syntax tree and to emulate a complete runtime environment for it. They do not translate the syntax tree into another form, but traverse its nodes by simulating their runtime behavior as defined by the programming language. Interpreters are useful for automated testing and verification of the implementation of a programming language or executing scripts inside a program. For this particular case the Eigen Compiler Suite provides a generic framework that allows embedded interpreters to interface with their environment.

26.4.5 Transpilers

Transpilers translate programs written in one programming language into programs written in other programming languages. They therefore behave like pretty printers, except that the target programming language differs from the source programming language. Additionally, transpilers usually need the semantic information provided by semantic checkers in order to translate the source code correctly.

26.4.6 Documentation Generators

Documentation generators extract the structure of the source code and combine it with annotations provided by the programmer into generic documentations. This generic representation of the extracted information is used afterward to generate documents of different formats. For more information about generic documentations and their generation by the Eigen Compiler Suite, see Chapter 25.

26.4.7 Intermediate Code Emitters

The Eigen Compiler Suite defines an intermediate code representation that is able to represent arbitrary programs using instructions for an abstract machine. This intermediate representation can be translated into actual machine code by machine code generators, see Section 26.5.3 An intermediate code emitter traverses syntax trees and translates theirs nodes into intermediate code sections and instructions that correspond to the runtime behavior of the original programs.

The Eigen Compiler Suite provides a generic intermediate code emitter that is a base for all concrete emitters and simplifies the generation of intermediate code. For more information about intermediate code and its purpose, see Chapter 23.

26.4.8 Front-Ends

A front-end combines all of the previous stages required to transform the original source code into an intermediate code representation thereof. This most often includes the parser, the semantic checker, and the intermediate code emitter. Since this intermediate representation can be translated later into machine code for a variety of hardware architectures, programmers only have to provide a single front-end instead of one for each possible combination thereof. In combination with the intermediate code interpreter, they are useful for automated testing and verification of the generated intermediate code.

26.4.9 Compilers

Compilers translate source code written in a specific programming language into machine code targeting a specific hardware architecture. They therefore just combine the output of one specific front-end with the input for a specific back-end, see Sections 26.4.8 and 26.5.4. In the end, compilers write the resulting machine code into object files. For more information about object files and their purpose, see Chapter 22. If programmers add support for an additional programming language by implementing a front-end accordingly, or if they support a new hardware architecture by implementing an additional back-end, compilers for all new combinations just get available without further ado.

26.5 Hardware Architectures

This section describes the tools and components typically provided by the Eigen Compiler Suite for a specific hardware architecture and how they are implemented. The support for a hardware architecture is generally based on a powerful representation of its processor instructions as shown in Figure 26.3. This abstraction must be able to represent any valid combination of mnemonic and operands of the instruction set of the target hardware architecture. In order to be most useful, this abstract representation must be able to be instantiated in the following three ways:

- Translation from Text

  A concrete instruction and its operands have to be recognized by their textual representation. Usually, the documentation of the target hardware architecture defines how its instructions are textually represented. This functionality is needed by assemblers, see Section 26.5.1.

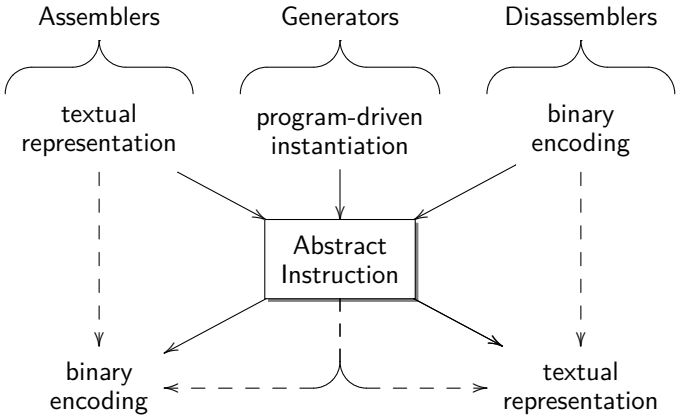


Figure 26.3: Abstract representation of instructions

- Instantiation by Code

  A concrete instruction shall be instantiatable by providing its mnemonic and an abstract representation of its operands. The operands themselves are composed of immediate values, registers, or special-purpose data addressing. This functionality is needed for the translation from text as well as by machine code generators, see Section 26.5.3.

- Decoding from Machine Code

  A representation of a concrete instruction with its operands must also be creatable by decoding one or more binary octets. The actual encoding and decoding of instructions is defined by the instruction set of the target hardware architecture. This functionality is needed by disassemblers, see Section 26.5.2.

Additionally, instances of the abstract instruction representation must also be able to emit themselves in the following two ways:

- Encoding into Machine Code

  Any representation of a valid instruction shall be encoded into one or more binary octets. This functionality is needed by assemblers and machine code generators, see Sections 26.5.1 and 26.5.3.

- Translation into Text

  Any representation of a valid instruction and its operands shall be translated into its textual representation. This functionality is needed by disassemblers and machine code generators, see Sections 26.5.2 and 26.5.3.

Usually, all combinations of instruction mnemonics and operand types that are valid according to the instruction set of the target hardware architecture are stored in a table. This table often also contains information about how a concrete instruction with its mnemonic and operands is represented using machine code. This information is needed for encoding and decoding instructions as discussed above.

26.5.1 Assemblers

All assemblers featured by the Eigen Compiler Suite implement the generic assembly language. It provides a generic abstraction for any textual representation of processor instructions. It also supports common functionality like directives or the evaluation of arithmetic, bitwise, and logical operations. For more information about the generic assembly language and how to use it, see Chapter 8. The Eigen Compiler Suite provides a generic assembler that is a base for all concrete assemblers and implements the generic assembly language. It performs tasks like managing sections, evaluating expressions, and writing the resulting object files. Concrete assemblers only have to apply the translation of a simple textual representation of a single instruction into its machine code encoding.

26.5.2 Disassemblers

The Eigen Compiler Suite provides a generic disassembler that is a base for all concrete disassemblers. It is able to process the sections stored in an object file and to print their textual representation. Data sections are represented using the generic data directives supported by the generic assembly language, while the machine code stored in code sections are passed to the concrete disassemblers. They only have to provide the encoding of some binary octets representing a single instruction and to print its textual representation.

26.5.3 Machine Code Generators

Machine code generators translate intermediate code into equivalent machine code for the target hardware architecture. The Eigen Compiler Suite provides a generic machine code generator that is a base for all concrete generators. It is able to manage intermediate code sections and process the instructions that have the same binary encoding across all concrete generators. This includes all instructions contained in data sections or instructions that are not natively supported by the concrete machine code generator. The generators translate one intermediate section at a time and create a corresponding object file section and a debugging information entry. Additionally, generators are able to create an assembly code listing of the generated machine code for verification and debugging purposes. Since the resulting assembly code listing is by design based on the generic assembly language, it can also be processed by the corresponding assembler yielding an exact copy of the original machine code.

26.5.4 Back-Ends

Back-ends combine the assembler and the machine code generator for a specific hardware architecture and provide an abstract description of some of the architectural characteristics that are necessary for the generation of intermediate code. This includes platform-specific information about address and default integer sizes as well as data alignment constraints for example. The abstract description is designed to be passed to front-ends for programming languages that need this information, see Section 26.4.8.

26.6 Runtime Environments

The Eigen Compiler Suite supports several different runtime environments. A runtime environment is characterized by the actual hardware architecture it is running on, as well as the underlying operating system.

26.6.1 Linkers

In order to run programs created by compilers and assemblers of the Eigen Compiler Suite on a specific operating system or machine, there must be a linker that generates the output files that are executable on the target platform. The Eigen Compiler Suite provides a generic linker that is able to map all sections of several object files into one or two binary arrangements and do the necessary linking therein. Two binary arrangements are needed, when data and code have to be arranged in separate address spaces. A concrete linker has just to write the binary data into a file using the target file format. Oftentimes however, binary file formats can also be represented using only the features already provided by the generic assembly language. In these cases, the executable output files can also be created using the plain binary file linker provided by the Eigen Compiler Suite and no specific linker is necessary.

26.6.2 Runtime Support

Runtime support is needed for the initialization at the beginning of the program execution as well as the implementation of some standard functions that interface the environment. This support depends on the actual hardware architecture, the file format of the executable file, as well as the underlying operating system. Additional runtime support is needed for intermediate code instructions that are not natively supported by the machine code generator for a specific hardware architecture. This support depends only on the actual hardware architecture and is the same across different operating systems. Some programming languages may also require runtime support for their language features. The runtime support for a language is bundled into so-called *library files*. A library file is a combination of object files targeting a single hardware architecture.

26.7 Development Tools

The source code of the Eigen Compiler Suite is accompanied by a makefile and some utility tools for developers. These facilities are not required to build, execute, test, or modify the Eigen Compiler Suite but are intended to simplify these activities. The remainder of this section describes all utilities provided by the Eigen Compiler Suite and their individual command-line interfaces. The capabilities of the makefile on the other hand are described in a readme file also supplied with the source code.

26.7.1 depwalk

`depwalk` is a utility tool for dependency walking. It accepts a list of C++ source files or Oberon modules which are scanned for included header files or imported modules respectively. The resulting dependencies of each file are written to the standard output stream and get processed in the same way. The output is designed to replace the corresponding section of the makefile in order to consistently maintain all dependencies whenever the source code of the Eigen Compiler Suite is modified accordingly.

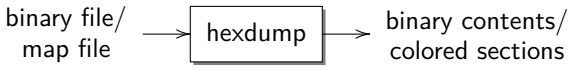
26.7.2 ecsd

`ecsd` is a driver utility tool which conveniently invokes the tools of the Eigen Compiler Suite with appropriate command-line arguments and environment variables. Its functionality and command-line interface are described in Chapter 2.

For developers of the Eigen Compiler Suite it additionally provides the `-m` command-line flag which automatically builds all dependencies of a tool before invoking it. A special target environment called `code` allows exposing and executing the intermediate code representation of a program passed between front-ends and back-ends. For the purpose of creating and debugging runtime environments on the other hand, the driver allows targeting freestanding environments for all supported hardware architectures using the `-t` option. The available names correspond to the common architecture prefixes and suffixes of the 82 compiler and assembler tools listed in Table 4.1 on page 56.

26.7.3 hexdump

`hexdump` is a utility tool for viewing binary files. It accepts the name of a binary file and writes its contents in hexadecimal form to the standard output stream. If additionally provided with a corresponding map file as generated by linkers, it lists all sections contained therein and colors their contents.

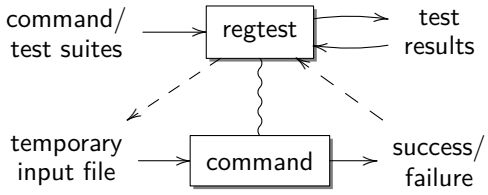


26.7.4 linecheck

`linecheck` is a utility tool for checking text lines. It accepts a list of plain text files and checks their contents for consistent use of white space. The complete source code of the Eigen Compiler Suite is stored in plain text files and validated using this tool.

26.7.5 regtest

`regtest` is a utility tool for regression testing. It accepts a quoted command line and a list of plain text files called test suites which contain an arbitrary number of tests. It executes the command once for each test and summarizes differing results between consecutive runs which are stored in a specified result file for regression testing. Each test consists of a short description followed by some contrived input that gets stored in a specified temporary file and is intended to cause the command execution to either succeed or fail.



Each test is described by a single line of text beginning with the identifier `positive` or `negative` indicating the expected result of the command execution, followed by a colon and a unique name. The actual input of a test consists of all indented text lines following its description ignoring the first horizontal tab character of each line. Lines beginning with a number sign character denote comments for annotating and structuring test suites and are completely ignored.

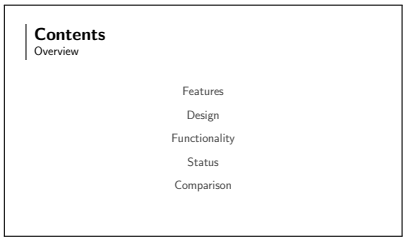
Addendum

A | Presentation Material

This chapter is a compilation of presentation material that can be used for presenting the design, functionality, and various implementation details of the Eigen Compiler Suite. Each presentation consists of a set of slides and a detailed explanation of their contents.

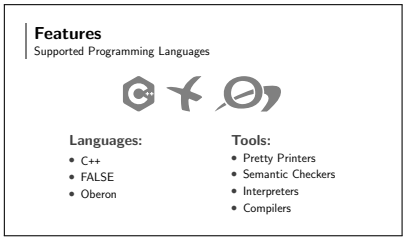
A.1 Overview

Eigen Compiler Suite is the name of a free software collection of development tools. This presentation gives a general overview over the Eigen Compiler Suite by describing its features, design, functionality and status, and comparing it to related free software.



A.1.1 Features

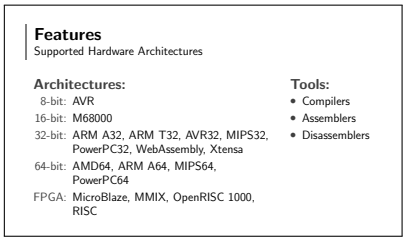
The Eigen Compiler Suite is a toolchain for developing software that contains various tools like compilers, interpreters, assemblers, and linkers targeting a variety of programming languages and hardware architectures. It implements each of its supported programming languages by providing tools like pretty printers, semantic checkers, interpreters, and frontends for compilers. Depending on the programming language, other tools like preprocessors, transpilers, and documentation generators are provided as well. The Eigen Compiler Suite additionally defines the generic assembly language that provides a common programming framework for all supported assemblers.

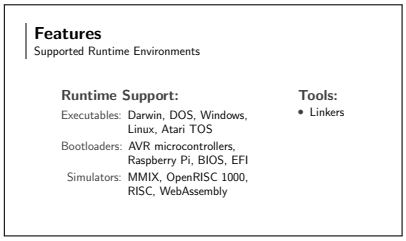


The Eigen Compiler Suite supports various hardware architectures by providing tools like assemblers, disassemblers, and machine code generators for compilers. Some of these architectures like MIPS, AMD64, and PowerPC define different operating modes or bit sizes in which case the Eigen Compiler Suite pro-

vides a set of tools for each variant. Some of the supported architectures like RISC, MMIX, MicroBlaze, and OpenRISC 1000 may only be available on simulators or FPGA implementations.

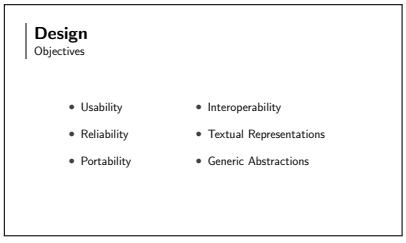
Concerning its supported runtime environments, the Eigen Compiler Suite provides several different linker tools that generate files like executables, bootloaders, or disk images for simulators.





A.1.2 Design

The Eigen Compiler Suite was primarily designed to be a simple but complete and self-contained toolchain. In particular, all tools featured by the Eigen Compiler Suite share the same easy-to-use command-line user interface and do not require complex installations or other prerequisites. Another objective is to produce correct results as well as comprehensive diagnostic messages. Optimizations and performance are explicitly secondary, as the implementation of the Eigen Compiler Suite is mainly driven by correctness, reliability, and simplicity. The Eigen Compiler Suite is completely written using standard and portable programming language features in order to guarantee the portability of its source code. Therefore, all compilers featured by the Eigen Compiler Suite are cross compilers by design. Furthermore, tools like compilers and assemblers shall enable interoperability in-between all programming languages supported by the Eigen Compiler Suite. Since all intermediate data is represented using a human-readable and machine-independent text format, programmers and maintainers are able to view, modify and even manually create all kinds of intermediate data. Finally, the implementation of the Eigen Compiler Suite shall provide generic abstractions in order to reuse code wherever possible while implementing these objectives.



In order to ensure that all programming tools of the Eigen Compiler Suite are reliable and produce correct results, several test and validation suites are provided

which enable automatic regression testing. Test Suites for programming languages are separated into compilation and execution tests in order to check for code that should or should not compile, as well as complete and valid programs that either generate a runtime error or run successfully to completion. Regarding the generic assembly language, there exists a test suite for each supported hardware architecture that features at least one compilation test for each element of the corresponding instruction set.

Reliability
32 Regression Test Suites

Subject	Tests
C++	3 400
FALSE	70
Oberon	2 400
Assembly Code	17 200
Intermediate Code	4 100
Linker	1 200

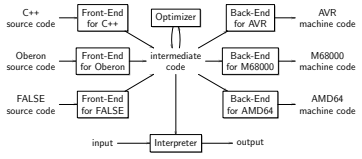
Each test suite tests one particular tool and is represented using a single text file that contains an arbitrary number of tests. Each test is identified using a unique description for regression testing and is followed by the corresponding input for the tool under test. A positive test requires the tool to succeed when given the corresponding input while a negative test expects a failure. This textual representation is very compact and has proven itself to be very versatile because it allows testing any command-line tool.

Reliability
Sample Test File

```
# Oberon execution tests

positive:  empty module

           MODULE Test;
           END Test.

negative:  simple halt

           MODULE Test;
           BEGIN HALT (123);
           END Test.
```

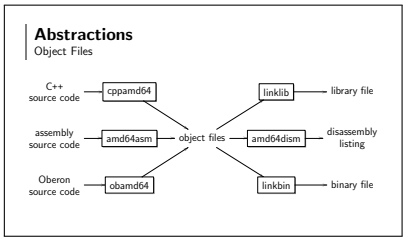
One goal of the Eigen Compiler Suite is to provide compilers, assemblers, and linkers for all possible combinations of programming languages and target architectures. For this reason, the Eigen Compiler Suite uses an intermediate code representation that acts as a generic abstraction in-between *front-ends* implementing programming languages and *back-ends* targeting hardware architectures. A single data flow of the shown diagram depicts a single compiler for a particular programming language targeting a particular hardware architecture. For more information about intermediate code and its purpose, see Chapter 23.

Abstractions
Intermediate Code



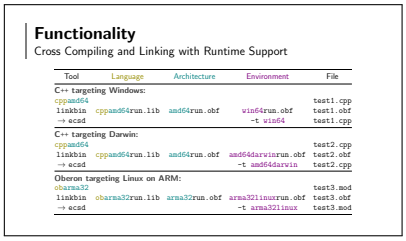
Object files on the other hand are the key abstraction that enables interoperability between all compilers and assemblers of the Eigen Compiler Suite. All these tools generate the same kind of output called object files which can be processed by all linkers and disassemblers of the Eigen Compiler Suite. In addition, the use of object files enables separate compilation which allows compiling only those parts of a program that

have actually changed. For more information about object files and their purpose, see Chapter 22.



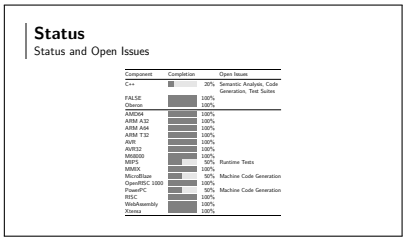
A.1.3 Functionality

Each tool of the Eigen Compiler Suite provides the same command-line user interface which treats each command-line argument as the name of a textual input file. Command-line options are not supported in order to make the contents of output files independent from the actual tool invocation. For compiling programs for example, the compiler as to be invoked using the corresponding name of the source code file. This results in a new object file for the provided source code which can be used to link the executable program. Depending on the used compiler, the hardware architecture it targets, as well as the desired runtime environment, the corresponding runtime support files have to be linked as well. The same result can also be achieved using a utility tool called `ecsd` which automatically infers the required tools from the source file and the specified target environment and conveniently invokes them with the necessary runtime support.



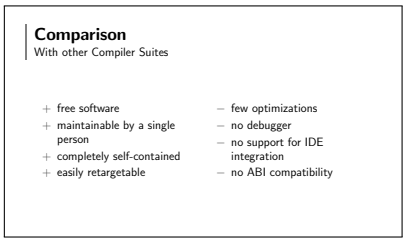
A.1.4 Status

Except for C++ all programming language implementations provided by the Eigen Compiler Suite are completed. The C++ implementation features a preprocessor, parser, and pretty printer but currently lacks proper support for the semantic checker, interpreter, documentation generator, and intermediate code emitter. Once C++ is supported sufficiently, its test suite intended to cover all of the ISO C++ Standard ISO/IEC 14882:2023 [5] can be completed as well. Some architectures like MIPS or MicroBlaze have not yet been tested on simulators or actual hardware.



A.1.5 Comparison

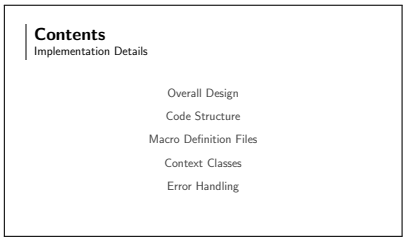
In contrast to comparable proprietary compilers, the Eigen Compiler Suite is free software which allows it to be studied, modified, and used for any purpose. In comparison to other free software compiler suites like GCC or Clang however, the code base of the Eigen Compiler Suite is much smaller and allows it to be maintained by a single person. In addition, the Eigen Compiler Suite is completely self-contained and requires no other software in order to build programs. Furthermore, all of its tools are completely portable and therefore executable in many runtime environments. In order to target different systems, it therefore often suffices to exchange the runtime support for the corresponding runtime environment.



On the other hand however, the Eigen Compiler Suite is not highly optimized and does not provide a debugger. Since it defines its own object file format, the produced object files and binaries are not ABI compatible with existing formats.

A.2 Implementation Details

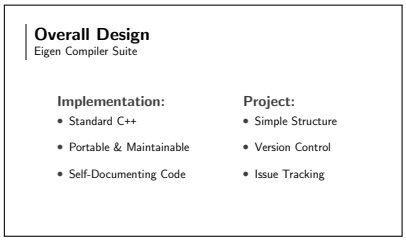
The Eigen Compiler Suite is a completely free and self-contained toolchain written in C++. This presentation gives some background information about its implementation and is mainly intended for developers and maintainers of the Eigen Compiler Suite.



The Eigen Compiler Suite provides a makefile which allows building all of its tools in environments like Windows, Darwin, and Linux-based operating systems. Its source code however is does not depend on any particular operating system, underlying hardware architecture, or any tools other than a conforming C++ compiler.

A.2.1 Overall Design

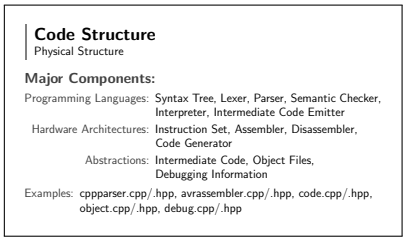
The tools of the Eigen Compiler Suite are completely written in standard C++. The only issue where its implementation currently depends on the execution environment is the handling of directory path delimiting characters. Apart from that, the source code of the Eigen Compiler Suite is completely portable and designed to be maintainable and self-documenting. By convention, it uniformly uses a consistent naming convention and self-explanatory function predicates which purposely render a lot of comments unnecessary.



The project itself has a simple structure with meaningful filenames that directly follows its logical layout. From the beginning, all of the files contained therein have been plain text files and under revision control. Quality assurance is achieved using an issue tracking system and build automation utilities.

A.2.2 Code Structure

The implementation of each major component of the Eigen Compiler Suite is physically stored in one source and one header file. Components of a typical programming language implementation are its syntax representation, the lexer, the parser, the semantic checker, and the intermediate code emitter. The implementation of a hardware architecture on the other hand is composed of a representation of its instruction set, an assembler, a disassembler, and a machine code generator. Other examples of major abstractions are the representations of intermediate code, object files, and debugging information.

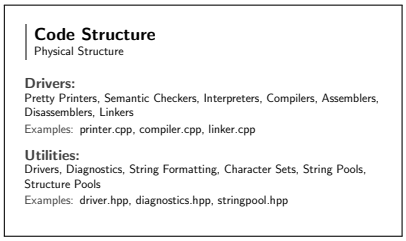


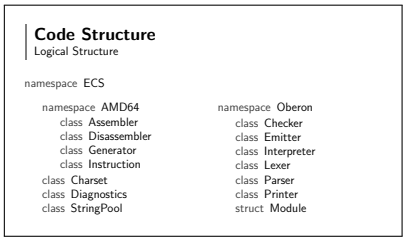
Each tool of the Eigen Compiler Suite combines one or more of these components into an executable file. For each tool there is a corresponding source file that contains the `main` function and is called a driver. Examples of drivers are pretty printers, semantic checkers, interpreters, and compilers for various

programming languages. Furthermore there are one or more assemblers and disassemblers for each hardware architecture, as well as linkers and debugging tools.

Finally there are common utilities that are used throughout the whole implementation of the Eigen Compiler Suite. These utilities are provided by header files and provide frameworks for writing drivers, or generic abstractions like character sets and diagnostic interfaces.

Logically, each programming language, hardware architecture, and major abstraction provides its components as classes in a distinct namespace. These namespaces and all common utilities reside in a global namespace called `ECS`. This allows any component of the Eigen Compiler Suite to be used as a library in other projects without provoking name clashes. Drivers are examples of how that library interface is used to combine various components of the Eigen Compiler Suite into executable files.

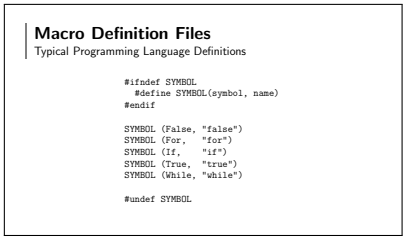




A.2.3 Macro Definition Files

In addition to source and header files, the implementations of almost all major components make use of so-called *macro definition files*. These are essentially header files that contain only sequential invocations of preprocessor macros typically known as X macros. This allows including the same macro definition file in various places using different macro definitions. The idea is to repeat the sequence of macro invocations in various different contexts while maintaining the actual macro arguments in a single place.

A typical use case of macro definition files is the definition of enumerations in header files where the macro is defined to provide the name of an enumerator. The same macro definition file is then typically also included in the corresponding source file where the macro may for example be defined to provide the textual representation of each enumerator in an array of constant strings. This is useful for maintaining a sorted



list of the keywords of a programming language for example. Since its lexer can compare scanned identifiers with elements of the constant string array, modifying the macro definition file is usually sufficient to add or remove keywords.

Other components that profit from the same technique are instruction set implementations which can represent complete instruction set tables in a single macro definition file. Using an appropriate macro definition, the corresponding set of instruction mnemonics can even be included in the documentation of the Eigen Compiler Suite. Chapter 8 for example uses this technique to list all instruction sets supported by the Eigen Compiler Suite.

Macro Definition Files
Typical Usage

```
enum Symbol {
  #define SYMBOL(symbol, name) symbol,
  #include "language.def"
};

const char*const symbols[] {
  #define SYMBOL(symbol, name) name,
  #include "language.def"
};
```

A.2.4 Context Classes

Each major component described above is represented using a distinct class that provides an interface for processing or transforming some kind of intermediate representation. A parser for example is a class that provides a function which takes a character stream as input and transforms it into an abstract syntax tree. The implementation of that interface however is typically not provided by the component class itself but by a private and nested class. These classes are conventionally called context classes and have several advantages over more traditional designs which implement the interface directly in the component class.

Context Classes
Light-Weight Interface in Header File

```
namespace ECS {
  namespace Oberon {
    class Parser;
    struct Module;
  }
}

class ECS::Oberon::Parser {
  class Context;
public:
  void Parse (std::istream&, Module&);
};
```

Since the implementation of an interface is just forwarded to the context class, the latter can be defined and implemented exclusively in the source file. All of the processing state that is necessary during an invocation of the interface can therefore be stored by the temporary instance of the context class rather than the component. As a consequence, the only information that remains necessary to represent directly in the component class is some optional constant configuration. This renders the actual definition of the component

class in the header file light-weight, stable, and by design thread-safe.

Context Classes
Forwarded Implementation in Source File

```
class ECS::Oberon::Parser::Context {
  ...
public:
  Context (std::istream&);
  Parse (Module&);
};

void ECS::Oberon::Parser::Parse
    (std::istream& is, Module& module) {
  Context context {is}.Parse (module);
}
```

A.2.5 Error Handling

The Eigen Compiler Suite provides a generic interface for reporting different kinds of diagnostic messages like errors, warnings, and notes. For drivers, it provides an implementation thereof which prints consistent diagnostic messages to standard output streams. In contrast to warnings and notes however which have merely informational purposes, errors additionally indicate failures to proceed. After emitting an arbitrarily detailed diagnostic message, an error is therefore also reported by throwing an exception. This technique conveniently allows programmers to prematurely abort any processing that results in an error without having to provide any information about the error in the exception itself. The resulting code can therefore treat error conditions like assertions and does not actually need to handle errors after they have been reported. As a consequence, most components of the Eigen Compiler Suite typically abort after the first encounter of an error and do not recover without further precaution. Usually however, subsequent errors are aftereffects anyway whereas independent exceptions can easily be batched together if necessary.

Error Handling
Reporting Diagnostic Messages

Type	Indication	Resolution
Errors Fatal Errors	Failure	Abort
Warnings Notes	Information	Proceed

1. Report message
2. Abort with exception

\+ Assert semantics
− No recovery

B Questions and Answers

This chapter answers frequently asked questions about the Eigen Compiler Suite and how to use its tools to accomplish common tasks.

B.1 General Questions

This section assembles questions frequently asked about the Eigen Compiler Suite in general.

What is the Eigen Compiler Suite? Why does it exist?

The Eigen Compiler Suite is a completely self-contained collection of software development tools. It exists to be recognized and adopted as a free development toolchain which is hopefully as useful and easy to use as its source code is intended to be approachable and comprehensible for developers and students wanting to learn, maintain, and customize a complete toolchain.

What does the name Eigen Compiler Suite mean?

The Eigen Compiler Suite provides all essential freedoms guaranteed by free software and is maintainable by a single person. The term eigen in its name thus means that the Eigen Compiler Suite is literally your own, regardless of whether you are a user or a developer.

Does the Eigen Compiler Suite support Unicode source files?

The Eigen Compiler Suite currently only supports the ASCII character encoding. It will however eventually support Unicode source files encoded in UTF-8.

B.2 How-Tos

This section describes how to accomplish commonly requested tasks using the tools of the Eigen Compiler Suite and therefore assumes some basic knowledge of its features and functionality.

How to disassemble plain binary files?

Since disassemblers can only process object files which are usually generated by other tools of the Eigen Compiler Suite, a plain binary file must first be converted into an object file. The simplest way to achieve this conversion is to assemble an assembly source file consisting of a single line of code which just copies the contents of the binary file using the embed binary file directive.

C | GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions

be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your

unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work

in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the

covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail. If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do,

use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

D | ECS Runtime Support Exception

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This ECS Runtime Support Exception (“Exception”) is an additional permission under section 7 of the GNU General Public License, version 3 (“GPLv3”). It applies to a given file (the “Runtime Support”) that bears a notice placed by the copyright holder of the file stating that the file is governed by GPLv3 along with this Exception.

When you use the ECS to compile a program, it may combine portions of certain ECS header and runtime support files with the compiled program. The purpose of this Exception is to allow compilation of non-GPL (including proprietary) programs to use, in this way, the header and runtime support files covered by this Exception.

0. Definitions

A file is an “Independent Module” if it either requires the Runtime Support for execution after a Compilation Process, or makes use of an interface provided by the Runtime Support, but is not otherwise based on the Runtime Support.

“ECS” means a version of the Eigen Compiler Suite, with or without modifications, governed by version 3 (or a specified later version) of the GNU General Public License (GPL) with the option of using any subsequent versions published by the FSF.

“GPL-compatible Software” is software whose conditions of propagation, modification and use would permit combination with the ECS in accord with the license of the ECS.

“Target Code” refers to output from any compiler for a real or virtual target processor architecture, in executable form or suitable for input to an assembler, loader, linker and/or execution phase. Notwithstanding that, Target Code does not include data in any format that is used as a compiler intermediate representation, or used for producing a compiler intermediate representation.

The “Compilation Process” transforms code entirely represented in non-intermediate languages designed for human-written code, and/or in Java Virtual Machine byte code, into Target Code. Thus, for example, use of source code generators and preprocessors need not be considered part of the Compilation Process, since the Compilation Process can be understood as starting with the output of the generators or preprocessors.

A Compilation Process is “Eligible” if it is done using the ECS, alone or with other GPL-compatible software, or if it is done without using any work based on the ECS. For example, using non-GPL-compatible Software to optimize any ECS intermediate representations would not qualify as an Eligible Compilation Process.

1. Grant of Additional Permission

You have permission to propagate a work of Target Code formed by combining the Runtime Support with Independent Modules, even if such propagation would otherwise violate the terms of GPLv3, provided that all Target Code was generated by Eligible Compilation Processes. You may then convey such a combination under terms of your choice, consistent with the licensing of the Independent Modules.

2. No Weakening of ECS Copyleft

The availability of this Exception does not imply any general presumption that third-party software is unaffected by the copyleft requirements of the license of the ECS.

E | GNU Free Documentation License

Version 1.3, 3 November 2008

0. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only

one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation of Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. Relicensing

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Addendum:
How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[1] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format*, 2010. Version 4.

[2] Tool Interface Standard (TIS). *Executable and Linking Format (ELF) Specification*, May 1995. Version 1.2.

[3] Intel Coorperation. *Hexadecimal Object File Format Specification*, January 1988. Revision A.

[4] Atari Coorperation. *Atari GEMDOS Reference Manual*, April 1986.

[5] ISO. *ISO/IEC 14882:2023: Information technology — Programming languages — C++.* International Organization for Standardization, December 2023.

[6] ISO. *ISO/IEC 9899:2018: Programming languages — C.* International Organization for Standardization, June 2018.

[7] Wouter van Oortmerssen. *The FALSE Programming Language*, 1993. Version 1.3.

[8] Hanspeter Mössenböck and Niklaus Wirth. *The Programming Language Oberon-2*. Institut für Computersysteme, ETH Zürich, July 1996.

[9] Brian Kirk et al. *The Oakwood Guidelines for Oberon-2 Compiler Developers*, October 1995.

[10] Christoph Sauer, Chuck Smith, and Tomas Benz. WikiCreole: A Common Wiki Markup. In *Proceedings of the 2007 International Symposium on Wikis*, pages 131–142. ACM Press, 2007.

[11] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, July 2025. Revision 3.37.

[12] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions*, November 2021. Revision 3.25.

[13] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions*, November 2021. Revision 3.16.

[14] ARM Limited. *ARM Architecture Reference Manual*, ARMv8 edition, December 2017. Issue C.a.

[15] Atmel Coorperation. *8-bit AVR Instruction Set*, November 2005. Revision E.

[16] Atmel Coorperation. *AVR32 Architecture Manual*, April 2011. Revision D.

[17] Motorola Inc. *M68000 Family Programmer's Reference Manual*, 1992. Revision 1.

[18] Xilinx Inc. *MicroBlaze Processor Reference Guide*, April 2012. Revision 14.1.

[19] MIPS Technologies. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS64 Architecture*, March 2010. Revision 3.00.

[20] MIPS Technologies. *MIPS Architecture For Programmers Volume II-A: The MIPS64 Instruction Set*, March 2010. Revision 3.00.

[21] Donald E. Knuth. *MMIXware: A RISC Computer for the Third Millennium*. Springer, 1999.

[22] OpenCores. *OpenRISC 1000 Architecture Manual*, August 2017. Revision 1.

[23] International Business Machines Corporation. *PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors*, March 2005. Version 2.3.

[24] Niklaus Wirth. *The RISC Architecture*, February 2012. Revision 9.8.2018.

[25] WebAssembly Community Group. *WebAssembly Specification*, February 2024. Release 2.0.

[26] Cadence Design Systems. *Xtensa Instruction Set Architecture (ISA) Summary*, April 2022. Product Release RI-2021.8.

[27] IEEE. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, 1985.

[28] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, October 2020. Revision 3.23.

[29] Apple Inc. *OS X ABI Mach-O File Format Reference*, April 2009.

[30] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*, September 2010. Revision 8.2.

[31] Atmel Coorperation. *8-bit AVR ATmega2560 Microcontroller*, February 2014. Revision 2549Q.

[32] Atmel Coorperation. *8-bit AVR ATmega32 Microcontroller*, November 2004. Revision 2503G.

[33] Atmel Coorperation. *8-bit AVR ATmega328 Microcontroller*, August 2010. Revision 8271C.

[34] Atmel Coorperation. *8-bit AVR ATmega8515 Microcontroller*, October 2006. Revision 2512J.

[35] Atmel Coorperation. *32-bit AVR AT32UC3A3 Microcontroller*, October 2012. Revision H.

[36] Texas Instruments Inc. *Common Object File Format*, April 2009.

Index

Index of Tools

Index of Library Names

Index of Runtime Support

Index of Target Environments