CODASIP STUDIO TECHNICAL REFERENCE MANUAL

Version: 6.10.0

Release Date: November 14, 2017

TABLE OF CONTENTS

1 PREFACE

1.1 About

This document contains reference material for the tools and Intellectual Property blocks associated with Codasip Studio.

Application-specific instruction set processors (ASIPs) are key building blocks in the Systems on a Chip (SoCs) that power today's electronic systems. ASIPs deliver greater computational efficiency than general purpose processors and more flexibility than fixed-function RTL designs. They are the ideal technology to consider for performance and power sensitive design elements in the next-generation of SoCs.

Codasip Studio provides an integrated development environment that helps designers to rapidly create optimized ASIPs. It includes pre-designed ASIP blocks that designers can either exploit directly or modify.

Very significant time savings are achieved through the automation of tasks that would otherwise be done manually, including the fully automatic generation of the programming and simulation toolchain, the generation of synthesizable RTL and the automatic generation of Functional Verification Testbenches (Codasip Studio supports UVM-based Functional Verification).

1.1.1 Intended Audience

This document is intended for the use of engineers working with Codasip Studio at a quite advanced level. Beginners to the product should first refer to the Codasip tutorial documents, which include the *Codasip Studio Quick Start Tutorial*.

1.1.2 Release Information

This is the first release of this document in the present form. Much of the material herein is derived from the former Codasip Framework Manual, which is now obsolete.

1.1.3 Product Revisions

Codasip Studio 6.10.0

1.1.4 Typographical Conventions

Table 1: Typographical conventions

Convention	Usage	Example
Capitalised	Standardized terms, defined earlier in the text or in the Glossary	Window, Project
Important	Important text	*Do not forget to ...*
Document ref	Reference to other Codasip and non-Codasip documents	Please refer to the *CodAL Language Reference Manual*.
`Code, filenames etc.`	Code, code values, Unix file names, prompts, etc.	File name `ca_utils.hcodal`
<*abstract name*>	Field for substitution with user data.	<*project*>/`model`
`keyword`	Inline references to CodAL keywords (lower case).	**`element`**, **`event`**
IDE_word	Inline references to keywords of the IDE (usually starts in upper case)	The **Project Explorer** window
Option→Suboption	Command path, typically starting from the main toolbar.	**File → New → CodAL Project**
`Example`	Examples - typically snippets of code.	`register bit[DATA_W] test;`
`Syntax explained`	Explanation of syntax	`StartSection: "start" "{"`

1.2 References

1.2.1 Other Codasip Documents

The following documents may be helpful to anyone using Codasip Studio :

- *Codasip Studio Quick Start Tutorial*
- Other Codasip tutorials
- *CodAL Language Reference Manual*
- *Codasip Studio User Guide*

Here is a complete list of the documentation for Codasip Studio:

Guides:

Document	Description
Codasip Studio Installation Guide	How to install the Codasip Studio software package.
Codasip Studio User Guide	Detailed guidance on the use of Codasip Studio and the tools that it contains.

Reference Manuals:

Document	Description
CodAL Language Reference Manual	A complete presentation of the CodAL language and how to use it for writing ASIP models.
Codasip Studio Technical Reference Manual	Reference information on Codasip Studio and the tools that it contains.
Codasip Program Description Model Language Manual	A complete presentation of the PDML language and how to use it for writing constraints for random applications generator.
Codasip Studio Message Reference Manual	A list of Codasip errors, warnings and notes that user can encounter during his work with Codasip Studio with descriptions, explanations, and possible solutions.

Tutorials:

Document	Description
Codasip Studio Quick Start Tutorial	A step-by-step introduction to the essentials of Codasip Studio.
Codasip Instruction Accurate Model Tutorial	A step-by-step introduction to writing Instruction Accurate ASIP models in CodAL.
Codasip Compiler Generation Tutorial	A step-by-step introduction to generating a C/C++ compiler from an Instruction Accurate ASIP model written in CodAL.
Codasip Cycle Accurate Model Tutorial	A step-by-step introduction to writing a Cycle Accurate CodAL model.
Codasip Interrupts and Peripherals Tutorial	A step-by-step introduction to adding external devices to an ASIP CodAL model.
Codasip JTAG Extension Tutorial	A step-by-step introduction to Codasip's JTAG extension.
Codasip SIMD Extension Tutorial	Tutorial showing the implementation of SIMD extensions in the Codasip uRISC

Codasip Custom Components Verification Tutorial	Tutorial desccribing proccess of adding manually modified UVM test-bench for a component into the ASIP or the top-level UVM test-bench.
Codasip uRISC VLIW Extension Tutorial	Tutorial showing modifications to Codasip uRISC to create a simple VLIW architecture.

1.2.2 Other References

There are no other references.

1.3 Feedback

1.3.1 Feedback on Codasip Products

If you have any comments or suggestions about Codasip products, please contact your supplier or send an email to support@codasip.com. Give:

- The product name
- The product revision or version
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

1.3.2 Feedback on this Document

If you have comments on this document, please send an email to feedback@codasip.com. Give:

- The document title and format (pdf, web page, etc)
- The chapter number, page numbers and version to which your comments apply
- A concise explanation of your comments.

Codasip also welcomes general suggestions for additions and improvements.

2 CODASIP STUDIO OVERVIEW

This chapter is organized as follows:

2.1 Organization of Codasip Studio

Codasip Studio is composed of the User Interface Layer (i.e. Codasip Studio), Codasip Commandline, and the tools, such as a linker, and generators of tools, such as a generator of C/C++ compiler.

Codasip Commandline is responsible for a correct execution of the generators and tools, checking their return codes and transferring outputs to the correct output directories.

For comfortable and intuitive ASIP design the Eclipse- based Codasip Studio is available that uses Codasip Commandline as a backend.

Advanced users can use the Codasip Commandline for scripting, automatic testing and other advanced work.

The Codasip Commandline accepts commands from the user directly or via Codasip Studio and processes them. It contains:

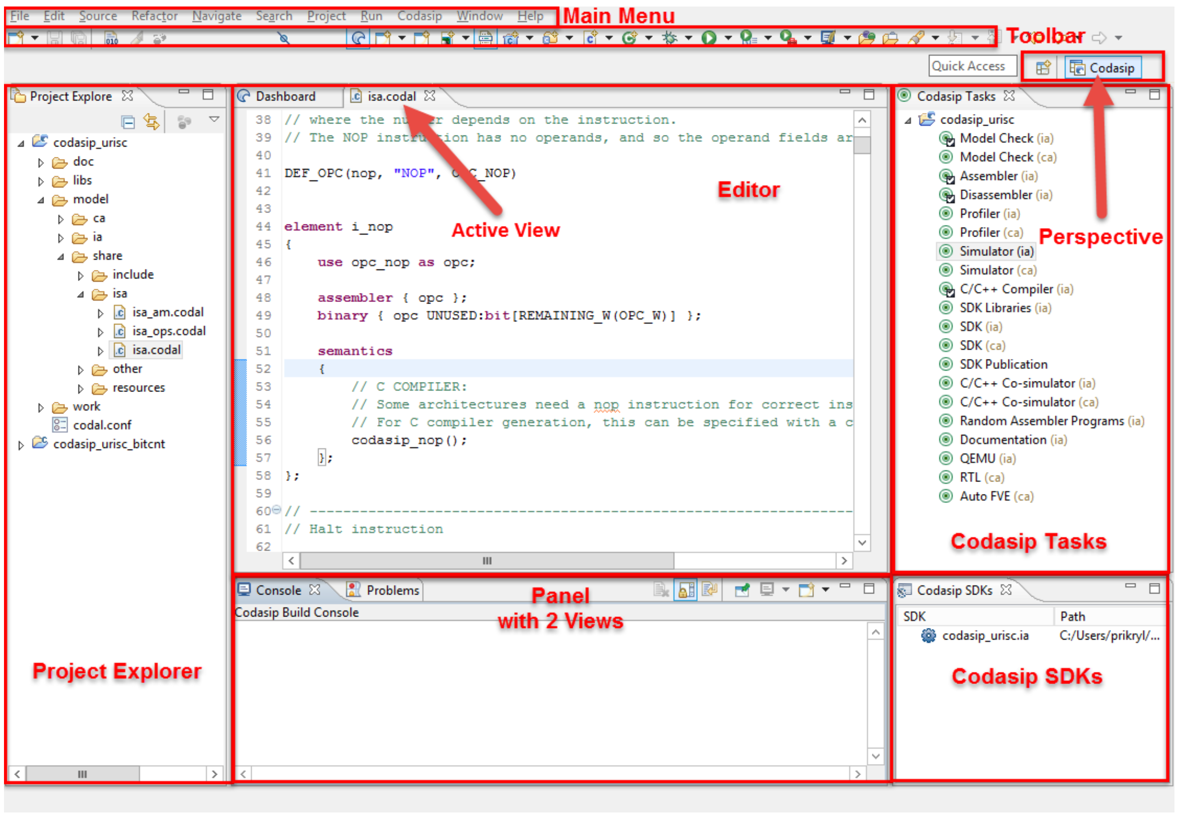
- The Software Development Kit (SDK) generators, for producing the assembler, compiler, simulator, etc needed for the ASIP under development
- The resulting SDK
- Important APIs: for the simulation of external models and for co-simulation with, for example, System Verilog simulators
- A Functional Verification environment generator

- A High Level Synthesis tool, for the production of synthesisable RTL from the ASIP model
- etc.

All of the functions mentioned above are driven from within Codasip Studio. In addition to this, an ASIP's SDK can be exported from Codasip Studio and used independently.

2.2 User Interface

2.2.1 Codasip Studio



```
// where the number depends on the instruction.
// The NOP instruction has no operands, and so the operand fields ar

DEF_OPC(nop, "NOP", OPC_NOP)


element i_nop
{
    use opc_nop as opc;

    assembler { opc };
    binary { opc UNUSED:bit[REMAINING_W(OPC_W)] };

    semantics
    {
        // C COMPILER:
        // Some architectures need a nop instruction for correct ins
        // For C compiler generation, this can be specified with a c
        codasip_nop();
    };
};

// ----------------------------------------------------------------------
// Halt instruction

```

Figure 1: Codasip Studio showing the Codasip Perspective

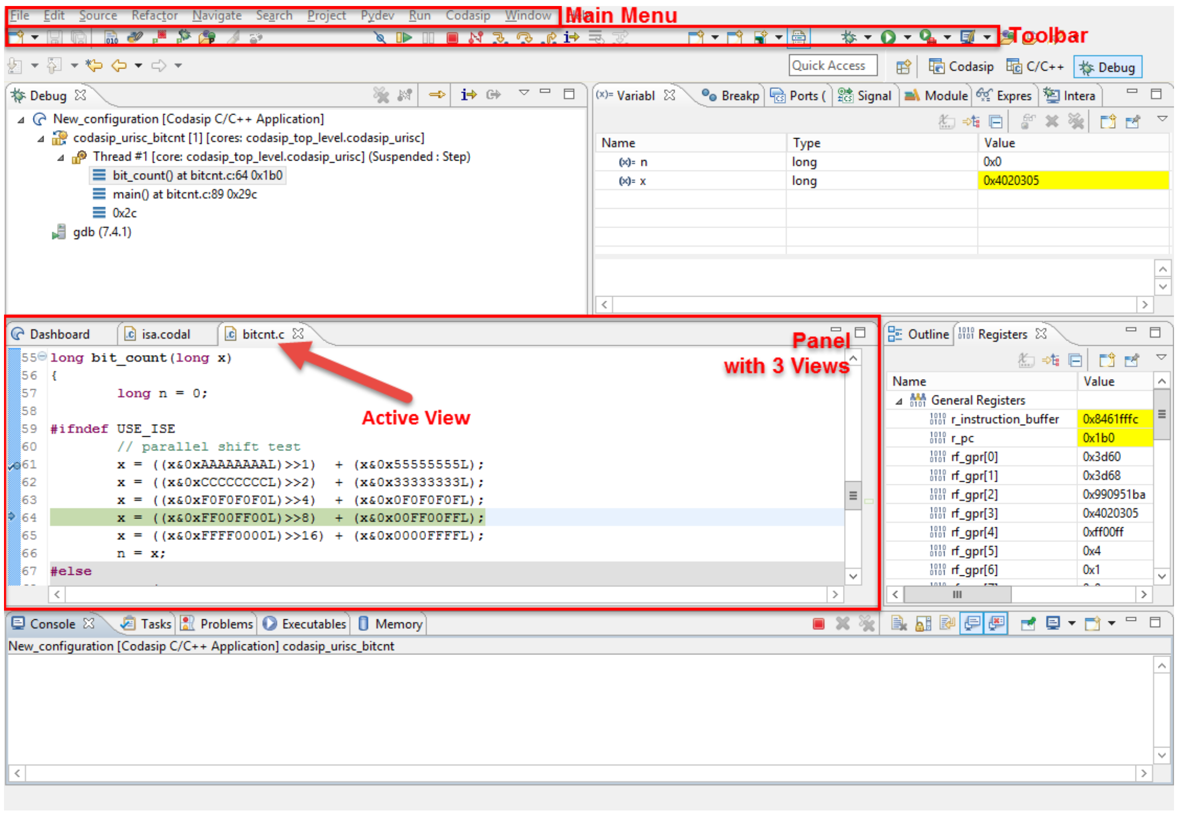
The figure above shows the main window of the IDE. Working from top to bottom, you first see the window's label.

Below the label, the **Main Menu** is located.

Below the **Main Menu**, there is the Toolbar (which can be toggled on and off from within the **Window** item of the **Main Menu**).

The remaining part of the window contains **Panels**, and each **Panel** contains one or more **Views**. A different set of Views are available for each **Perspective**, and they can be found and activated using **Window** → **Show View**.

The active **Perspective** can be changed using **Window** → **Perspective** → **Open Perspective** → **Other...**. The figure below is an example of the IDE with the **Debug Perspective** activated.



```
New_configuration [Codasip C/C++ Application]
  codasip_urisc_bitcnt [1] [cores: codasip_top_level.codasip_urisc]
    Thread #1 [core: codasip_top_level.codasip_urisc] (Suspended : Step)
      bit_count() at bitcnt.c:64 0x1b0
      main() at bitcnt.c:89 0x29c
      0x2c
  gdb (7.4.1)
```

Name	Type	Value
n	long	0x0
x	long	0x4020305

```
long bit_count(long x)
{
        long n = 0;

#ifndef USE_ISE
        // parallel shift test
        x = ((x&0xAAAAAAAAL)>>1)  + (x&0x55555555L);
        x = ((x&0xCCCCCCCCL)>>2)  + (x&0x33333333L);
        x = ((x&0xF0F0F0F0L)>>4)  + (x&0x0F0F0F0FL);
        x = ((x&0xFF00FF00L)>>8)  + (x&0x00FF00FFL);
        x = ((x&0xFFFF0000L)>>16) + (x&0x0000FFFFL);
        n = x;
#else
```

Name	Value
General Registers	
r_instruction_buffer	0x8461fffc
r_pc	0x1b0
rf_gpr[0]	0x3d60
rf_gpr[1]	0x3d68
rf_gpr[2]	0x990951ba
rf_gpr[3]	0x4020305
rf_gpr[4]	0xff00ff
rf_gpr[5]	0x4
rf_gpr[6]	0x1

```
New_configuration [Codasip C/C++ Application] codasip_urisc_bitcnt
```

Figure 2: IDE showing the Debug Perspective

The use model of the IDE is explained further in the *Codasip Studio User Guide* and Tutorials. This manual will enumerate and describe the features of the main views in the IDE.

For more information on the IDE, go to section "Codasip Studio Integrated Development Environment" on page 17. Refer also to the *Codasip Studio User Guide*.

2.2.2 Codasip Commandline

The Codasip Commandline tool is an essential part of the Codasip Studio. It is a Python interpreter (see http://docs.python.org) with built-in Python `codasip` module, which contains Codasip build system and all Codasip Studio generators and tools.

For more information on the Codasip Commandline, go to section "Command-line" on page 33.

2.2.3 Build System

The Codasip Studio projects are built by Codasip build system based on Python DoIt module (http://pydoit.org/). The build system allows users to modify, add, and automate build tasks, such as verification and non-regression testing. The same build system is also used by Codasip Studio.

The build system automatically loads project, its configuration and all referenced projects. It also manages dependencies between tasks, so only necessary tasks are executed. This greatly improves and speeds up usage of Codasip Studio tools.

For more information on the Codasip build system, go to section "Build System" on page 71.

2.2.4 File Organization

A software project that creates not only programs to run but also the processor(s) to run them on and the tools to assemble and compile them with must manage many files and file types. To facilitate this task, Codasip has defined a methodology and a file organization associated with it.

The Codasip build system then uses this methodology to detect all input files and automatically creates tasks dependencies, so no additional configuration is needed and only necessary tasks are executed.

For more information on the file organization methodology, go to "File Organization" on page 119.

2.3 Software Development Kits

Following tools are part of a generated SDK.

2.3.1 Assembler

The Assembler is a tool that translates human-readable assembly code into a binary object file. Object files are afterward linked by the Codasip Linker to processor-readable binary code.

Basically, the Codasip Assembler is able to parse two languages at once. The first being the instruction set language described by the CodAL model (instruction set syntax). The second being the language that is used to specify assembler directives and symbols.

The generated Assembler supports the compilation of assembly programs in the form commonly used by the GNU assembler. The Codasip C/C++ Compiler, as well as the GCC and LLVM compilers, produces the outputs in this format. Thus, the Codasip Assembler is able to assemble their outputs as well.

The language used to specify directives can be changed in order to be compatible with the company's standard or to handle some of the compiler's output. Changing the directives and symbols format is relatively straightforward.

For more information on the Assembler, go to section "Assembler, Disassembler and Linker" on page 129.

2.3.2 Disassembler

The Disassembler is a tool that reads an executable file (created as output of an assembler or a linker) and transforms it into an original assembly code. The Codasip Disassembler was designed in such way that its output is valid input to the Codasip Assembler and can be thus assembled again.

The generated Disassembler automatically treats object file data sections as data and generates constant data definitions to the output assembly code. As for code sections, it is slightly more complex. The Disassembler tries to recognize instructions, if it succeeds, it generates a textual instruction format. Otherwise it generates constant data definition and retries if the following data can be recognized as an instruction.

For more information on the Disassembler, go to section"Assembler, Disassembler and Linker" on page 129.

2.3.3 Linker

The Linker is a tool that links together multiple object files and resolves address relocations that were unknown during the assembly time. The Codasip Linker is independent of the target architecture, so it is not necessary to generate it.

For more information on the Linker, go to section"Assembler, Disassembler and Linker" on page 129.

2.3.4 C/C++ Compiler

For ASIP programming, Codasip provides a retargetable C/C++ compiler. The compiler is based on a popular and widely used open-source LLVM Framework that is strongly supported by the Apple, Google, Intel and other major companies. LLVM is a highly optimizing production-quality compiler and the most important extensions provided by Codasip are profile-based optimizations and support for VLIW architectures. Based on the execution profile, larger atomic code blocks (superblocks) are created and this allows much better scheduling of instructions both for single-issue and mainly VLIW architectures. Many features for VLIW support are provided, which allows obtaining high instruction-level parallelism (ILP) and higher performance with lower clock frequency.

Another unique feature of the Codasip C/C++ compiler is the ability to set various constraints, e.g. the compiler can handle most instruction hazards automatically and your processor core can be much smaller in area and have lower power consumption.

The C/C++ compiler comes with a set of tests from the gcc-torture testsuite. The standard C library (Newlib) and the compiler runtime library for software-implemented operations are provided. The retargetable C/C++ compiler translates the application from the C/C++ language into the GNU assembler format. The assembly program is then translated by the assembler tool into the object file in the ELF format. From several object files one executable (.xexe) file is created by the GNU based linker.

For more information on compiler generation, go to section "Compiler Generator" on page 182. You may also refer to the "Compiler Generator Guide" chapter of the *Codasip Studio User Guide*.

2.3.5 Simulators

Simulators generated in Codasip Studio enable designers to debug and test the embedded system hardware and software. All simulator types are generated from CodAL projects.

Main features:

- Multiprocessor simulation
- Integrated debugger with GDB interface
- Integrated profiling data generation
- Cooperation with Codasip Studio and Codasip Codespace (debugging, profiling data view)

- Plugin system for non-CodAL components
- Logging and loading of resource value changes
- Basic Python scripting

Codasip Studio supports generation of three simulator types:

- interpreting instruction-accurate simulator
- interpreting cycle-accurate simulator

The first type is the interpreting instruction-accurate simulator, a.k.a. instruction set simulator. The concept of this simulator is based on a constant fetching, decoding and execution of instructions from the memory. It is created from the instruction-accurate description.

The second type is interpreting cycle-accurate simulator. The simulation concept is the same as in the first type. The simulator is generated from the cycle-accurate description.

Previous types of simulators can be also exported as dynamic libraries with a wrapper. The wrapper can be generated with SystemC, pure C/C++, or DPI interface, so the simulator can be plugged into the SystemC simulation platform, RTL simulators, etc.

In all cases, the simulators are not event driven. It means, that the behavior within ASIP or design level is simulated sequentially. The simulation order follow a simple rule. The simulation starts from the top design level. **`extern`**s or other components placed in this level are simulated in an alphabetical order or, if a simulator settings is present on the level, they are simulated in user defined order.

Let's assume a top design level with two **`extern`**s, one for ASIP, called *01_master_cpu*, and the second for another design level, called *02_helper*. Let's assume that this level contains only one **`extern`**, called *slave_cpu*. The final order of the simulation is *01_master_cpu*, *02_helper*, and finally *slave_cpu*.

Simulator may be created even for a CodAL Project without an **`extern`** for a memory. In this case, the tools automatically instantiate memories. A memory is instantiated for each address space. Interfaces that are listed within the address space definition are connected to the memory. If an interface is not used in any address space, then it is not connected at all.

2.3.6 Debugger

Codasip Studio simulators can be generated with the debugger support. This allows execution of the simulator in a debug mode and debugging of running applications or directly CodAL source code. The Codasip Debugger is a part of the simulator executable.

The debugger is the most important tool for software developers. It enables debugging at multiple levels:

- Source level - one step corresponds to execution of a single line of source code. Codasip Debugger supports stepping in C code, Assembly (direct compilation from assembly code) or CodAL code.
- Instruction level - one step corresponds to execution of a single instruction (IA simulator) or single clock cycle (CA simulator).
- CodAL level - one step corresponds to execution of a single CodAL statement.

Codasip Debugger provides following features:

- Controlling the execution - suspend, step, or continue the simulated application execution.
- Call-stack - shows activated subroutines in the application that have resulted into current state of the execution.
- Breakpoints - suspend the execution at a given place in code. Each one can be ignored for given number of hits, temporarily disabled, or conditioned by an expression (e.g. `$r10==12`).
- Watchpoints - suspend the execution when a read or write operation occurs for a specific memory resource and/or a specific address. Each one can be ignored, temporarily disabled, or conditioned same as breakpoints.
- Resources - read/write value from/to resource. When the resource profiler is enabled, profiling data can also be retrieved.
- Expressions - evaluates expressions in C subset in current state of execution.

Codasip Debugger supports GDB compatible interface (GDB/MI and GDB/CLI) that allows software developers to use standard integrated development platforms for software development and debugging (currently, Eclipse CDT, Codasip Codespace and Codasip Studio are supported).

For correct mapping of instructions to source, C/C++ code DWARF debugging information format is used. This information is used by debugger for correct stepping, setting breakpoints on given source file and line, showing call-stack, etc. When the C/C++ compiler has used any optimizations (e.g., to better utilize the ASIP's pipeline), generated instructions are commonly shuffled to increase performance. Therefore debugger won't follow execution path of the source code but will follow the optimized code. Some variables might be even rearranged or optimized out of existence.

The debugger can also be used to debug on instruction level in assembly code. For such debugging, assembly source file or simulator built-in disassembler is used.

2.3.7 Profiler

Functional simulation is often not sufficient for creation of optimal design. More detailed information about the simulation process is usually required. For this purpose, the Codasip Profiler can be used. It tracks and logs all the important information during the simulation (e.g. how many clock cycles a particular function takes or usage of instructions during simulation). The results can be viewed in Codasip Studio/Codasip Codespace in the form of lists of instructions or functions with additional information or graphs.

Profiling information can be used by the designer to further optimize the target ASIP model (e.g. remove unused instructions, optimize those most used) or even running program (e.g. the algorithm can be rewritten or functionality from the software can be moved to the hardware or vice versa).

Two levels of profiling are supported:

1. Low level - profiling of resources read and write access count. For memories read/write access is stored for every word. No report is provided, it can be seen during a debugging session.
2. High level - the profiler gives a broad range of information, such as usage and coverage of instructions used during program execution, pipeline statistics, source code coverage and many more.

Profiling reduces performance of the simulation because of additional information that needs to be stored. Higher level of profiling has higher impact on the performance of the simulation.

There are two steps during the High level profiling.

1. Collection of information during simulation
2. Analysis of the collected information.

By default, simulator does not have a profiling support (because of performance reasons), so a user should enable it in a project configuration(**Simulator** > **Profiling** >

Enable profiling support). Moreover, simulator with the profiling result can track a different set of features.

High level profiling can affect the simulator performance considerably. Therefore, several features can be turned off in the project configuration(**Simulator** > **Profiling** > **High Profiler Level Options**), to increase the performance (CodAL Coverage, Decoders Coverage and PPA).

Profiler, which later on analyzes the output of the simulator, produces several types of outputs (e.g. HTML, text, ...). It has many options that control which information is analyzed and printed.

2.4 Simulation of External Models (Plugins)

A component is required when an `extern` construct is specified in a CodAL model.

The component is always associated with its type and so it can exist in multiple instances. If the component is using shared resources, it is necessary to ensure mutually exclusive access to these resources. There's no guarantee that components are running in the same thread.

The main interface has to be written in C++ language. When there is a requirement to write the component in another language or to use a specialized library, then the C++ interface becomes just a wrapper providing a bridge between the simulator and the component implementation in question.

For more information on plug-ins, go to section "Codasip Components" on page 291.

Plugins frequently communicate with other plugins and with ASIPs by using the Codasip Local Bus. This is described in "Codasip Local Bus" on page 304.

2.5 Co-Simulation

One of the usual designer's needs is to create an ASIP simulator, optionally with debugging features, in the form of a component for the SoC simulation model. Such components can be modeled using the standardized IEEE SystemC simulation platform or other simulation platforms, such as Questa® Advanced Simulator. Hence, the designer can optionally select the SystemC option in the Codasip Studio and a SystemC component is generated. Then it can be easily integrated into the SoC simulation model. Note that components with DPI and C/C++ interfaces can be generated as well.

Each simulation platform needs a compatible plugin, which must have a proper interface. This interface is stored in a header file, which is delivered together with the simulation platform. Paths to these files must be provided to Codasip Studio.

For more information on co-simulation, go to section "Co-simulation" on page 320.

2.6 Functional Verification

The goal of verification in Codasip Studio is to check that a design of an ASIP is functionally correct with respect to the high-level specification. Verification is divided into three stages:

1. In the first stage, functional consistency of IA CodAL model and CA CodAL model of an ASIP can be checked in the tool called Consistency Checker. Consistency checker evaluates programs in the IA simulator generated from the IA CodAL model and in the CA simulator generated from the CA CodAL model. It monitors all data approaches during the processing of instructions and compares whether data are handled the same way in both models.
2. Failing or inconsistent programs can be debugged in more detail in the debug perspective of the IA or the CA simulator. A huge advantage is that the debug perspective can now be started directly from the Consistency Checker tool.
3. After all programs passed the consistency checking, it is recommended to generate the synthesizable RTL representation of the ASIP processor and to use UVM-based verification to check its correctness. The reason for this ordering of verification actions is that UVM-based verification has much larger time overhead, it requires a third-party RTL simulator (for example QuestaSim) and is focused strictly on RTL debugging, not debugging of the model - this can be done much faster directly in Codasip Studio.

For more information on verification, please refer to the "Verification Guide" chapter of the *Codasip Studio User Guide*.

2.7 High Level Synthesis

When an architecture design is stable enough, a synthesizable RTL ASIP representation can be generated. It is possible to generate test benchmarks for the architecture, to generate asserts into the RTL description, or optionally, to generate support for a JTAG/Nexus debugging interface. Note that the generated synthesizable RTL is well proven by the 3rd party ASIC and

FPGA synthesizers.

Additionally, equivalence between the simulator and the hardware representation has to be ensured. Bear in mind that the behavior of this simulator should be the same as the behavior of the real hardware. In the terms of the Codasip Studio, this equivalence is guaranteed by the fact that all principles and algorithms are based on formal models and are well proven. Moreover, the simulator and hardware generators use the same algorithms for generation.

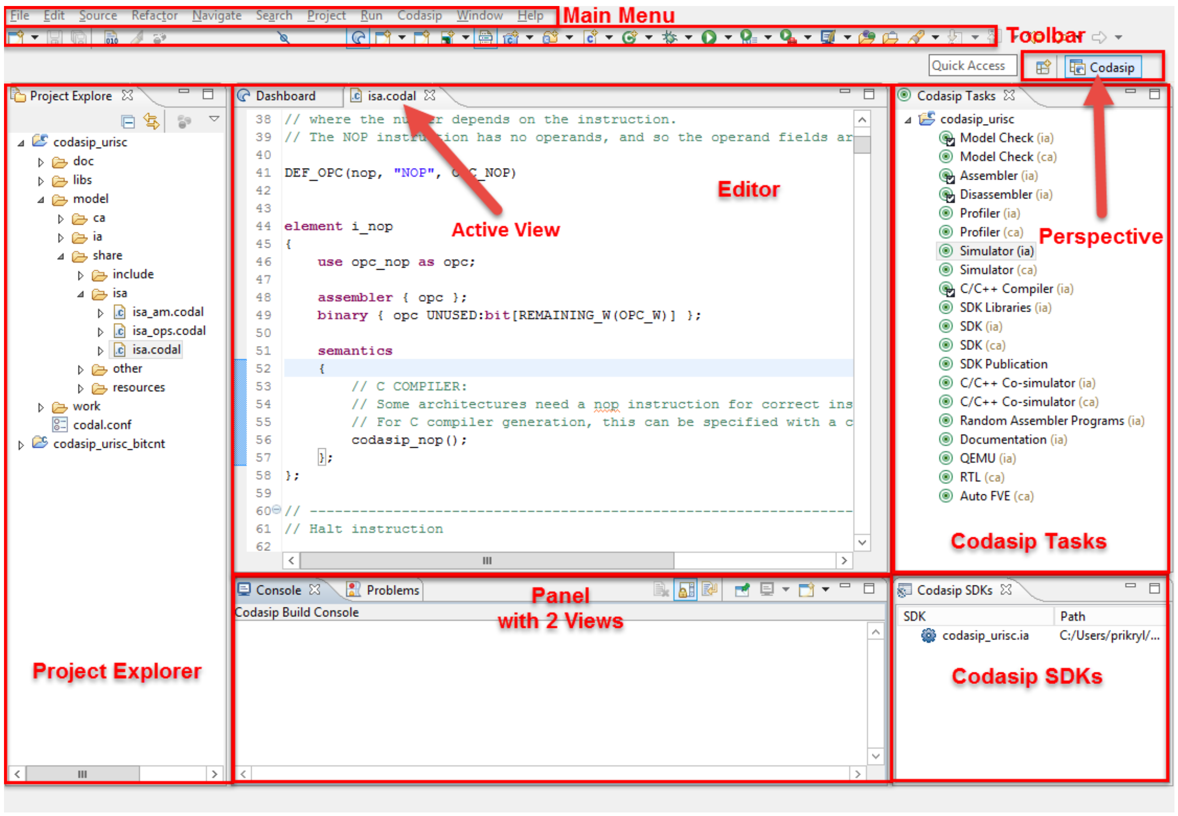
For more information on High Level Synthesis, go to section "High Level Synthesis" on page 326.

2.8 Exported Software Development Kits

The Codasip Software Development Kit (SDK) is a collection of programming tools based on the clang compiler driver that can be used for the development of applications. The main goal of the clang compiler driver is gcc command line compatibility, so the majority of existing projects based on the GNU makefile system can be compiled with minimal or no effort.

For more information on Exported SDKs, go to section "Codasip SDK Reference" on page 378.

3 CODASIP STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT



```
// where the number depends on the instruction.
// The NOP instruction has no operands, and so the operand fields ar

DEF_OPC(nop, "NOP", OPC_NOP)


element i_nop
{
    use opc_nop as opc;

    assembler { opc };
    binary { opc UNUSED:bit[REMAINING_W(OPC_W)] };

    semantics
    {
        // C COMPILER:
        // Some architectures need a nop instruction for correct ins
        // For C compiler generation, this can be specified with a c
        codasip_nop();
    };
};

// -----------------------------------------------------------------
// Halt instruction
```

Figure 3: Codasip Studio showing the Codasip Perspective

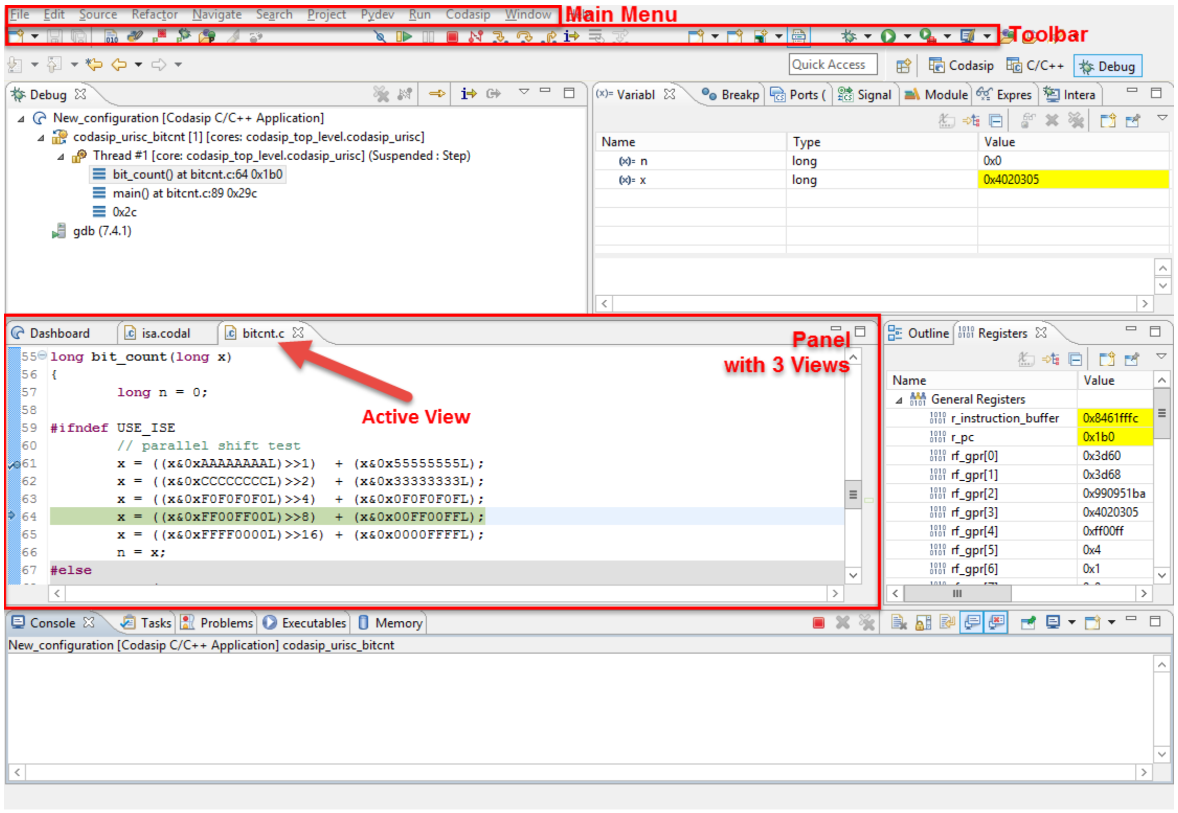
The figure above shows the main window of the IDE. Working from top to bottom, you first see the window's label.

Below the label, the **Main Menu** is located.

Below the **Main Menu**, there is the Toolbar (which can be toggled on and off from within the **Window** item of the **Main Menu**).

The remaining part of the window contains **Panels**, and each **Panel** contains one or more **Views**. A different set of Views are available for each **Perspective**, and they can be found and activated using **Window** → **Show View**.

The active **Perspective** can be changed using **Window** → **Perspective** → **Open Perspective** → **Other...**. The figure below is an example of the IDE with the **Debug Perspective** activated.


File Edit Source Refactor Navigate Search Project Pydev Run Codasip Window
Main Menu
Toolbar
Quick Access
Codasip C/C++ Debug
Debug
Variabl Breakp Ports (Signal Module Expres Intera
Dashboard isa.codal bitcnt.c
Panel with 3 Views
Active View
Outline Registers
Console Tasks Problems Executables Memory

New_configuration [Codasip C/C++ Application]
codasip_urisc_bitcnt [1] [cores: codasip_top_level.codasip_urisc]
Thread #1 [core: codasip_top_level.codasip_urisc] (Suspended : Step)
bit_count() at bitcnt.c:64 0x1b0
main() at bitcnt.c:89 0x29c
0x2c
gdb (7.4.1)

Name	Type	Value
n	long	0x0
x	long	0x4020305

```
long bit_count(long x)
{
        long n = 0;

#ifndef USE_ISE
        // parallel shift test
        x = ((x&0xAAAAAAAAL)>>1)  + (x&0x55555555L);
        x = ((x&0xCCCCCCCCL)>>2)  + (x&0x33333333L);
        x = ((x&0xF0F0F0F0L)>>4)  + (x&0x0F0F0F0FL);
        x = ((x&0xFF00FF00L)>>8)  + (x&0x00FF00FFL);
        x = ((x&0xFFFF0000L)>>16) + (x&0x0000FFFFL);
        n = x;
#else
```

Name	Value
General Registers	
r_instruction_buffer	0x8461fffc
r_pc	0x1b0
rf_gpr[0]	0x3d60
rf_gpr[1]	0x3d68
rf_gpr[2]	0x990951ba
rf_gpr[3]	0x4020305
rf_gpr[4]	0xff00ff
rf_gpr[5]	0x4
rf_gpr[6]	0x1

New_configuration [Codasip C/C++ Application] codasip_urisc_bitcnt

Figure 4: IDE showing the Debug Perspective

The use model of the IDE is explained further in the *Codasip Studio User Guide* and Tutorials. This manual will enumerate and describe the features of the main views in the IDE.

This chapter is organized as follows:

3.1 Views Summary

The Views described in this section can be accessed in multiple ways from the IDE but, if in doubt, a sure way to find them is with **Window** -> **Show View**-> **Other...**. Examples of selected Views are given in the next section.

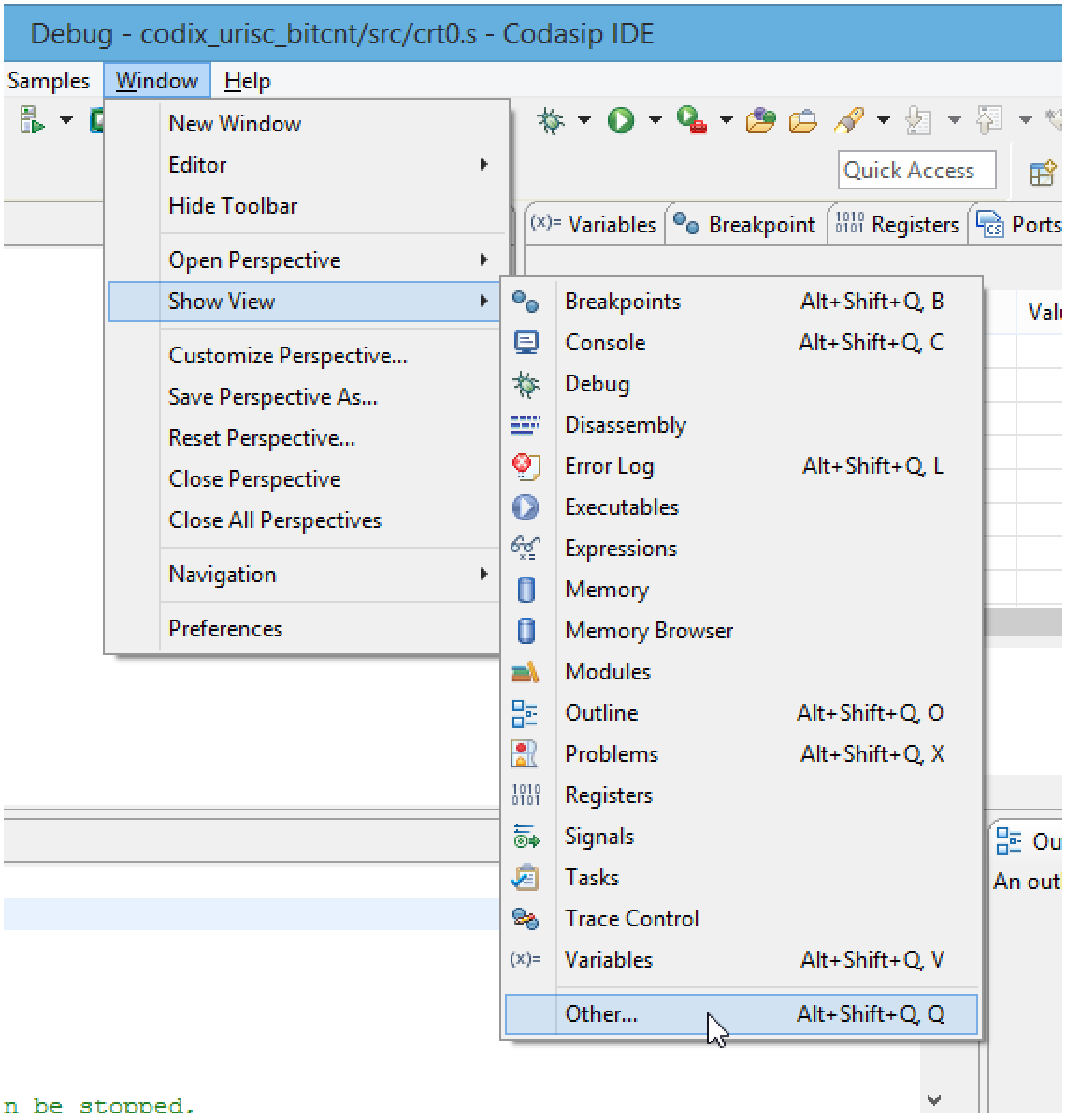


Figure 5 : Others menu item for accessing all available Views

As well the Views, Codasip-specific Dialogs, Wizards and Preferences panels feature in the IDE. These are described in a later section also.

The Codasip Studio is based on the Eclipse environment and the Views that make it up are in three Categories:

1. Codasip-specific views
2. Eclipse Views that have been enhanced or modified by Codasip
3. Pure Eclipse Views

Knowing the Category of a particular View can help both in understanding its behaviour and in finding information about it. The Category is therefore indicated in the third column of the following table, which lists all the Views available in the IDE:

IDE Views

Topic	View name	Cat.	Comment
General			Most views in this topic are in Category 3, and more information on them can therefore be found at help.eclipse.org. Exceptions are noted below.
	Bookmarks	3	
	Classic Search		
	Console	2	There are different types of Console window, and several may be active at the same time. You can switch between them using the "Display Selected Console" icon in the Console toolbar.
	Error Log	3	
	Internal Web Browser		
	Markers		
	Navigator		
	Outline		
	Palette		
	Problems		
	Progress		
	Project Explorer	2	An example is given in the following section.
	Properties	3	
	Search		
	Tasks		
	Templates		

Topic	View name	Cat.	Comment
C/C++		3	See help.eclipse.org
	C/C+ Index		
	C/C++ Projects		
	Call Hierarchy		
	Include Browser		
	Problem Details		
	Type Hierarchy		
Codasip			
	Codasip SDKs		This window is used to see, register and manage SDKs.
	Codasip Tasks		Use this window to build the models and tools associated with ASIP and Component projects. See the Codasip Instruction Accurate Model Tutorial and other tutorials for examples.
	IA/CA Consistency Checker		This window displays results of the Consistency Checker
Codasip Debug			
	Ports (Codasip)	1	If your design contains Ports (for example, if there Components connected to an ASIP in a CodAL project, then there are likely to be Ports in the interconnections) then the activity on these Ports may be seen in this View.
	Signals (Codasip)	1	Ay cycle-accurate model contains Signals, and they may be seen in this View.
Codasip Profiling			
	Profiling Sessions	1	
	Profiling Result Web View	2	Showing profiling results in internal web browser.
CVS		3	See help.eclipse.org
	CVS Editors		
	CVS Repositories		
Debug			

Topic	View name	Cat.	Comment
	Breakpoints		
	Debug		
	Disassembly		
	Executables		
	Expressions		
	Memory	2	Codasip Studio allows the inspection of multiple memories - see the example in the Debug Views section below.
	Modules		
	OS Resources		
	Registers	2	See the example in the Debug Views section below.
	Signals	3	Not to be confused with the Signals View in the Codasip Debug topic
	Trace Control		
	Variables		
Dynamic Langs.		3	See help.eclipse.org
	Call Hierarchy		
	Interactive Console		
	Script Debug Log		
	Script Explorer		
	Script Unit Test		
	Type Hierarchy		
Git		3	See wiki.eclipse.org/EGit/User_Guide for all views in this topic.
	Git Interactive Rebase		
	Git Reflog		
	Git Repositories		
	Git Staging		
	Git Tree Compare		
Help			
	Cheat Sheets		

Topic	View name	Cat.	Comment
	Help		
Make		3	See https://eclipse.org/cdt/
	Make Target		
Mylyn		3	See help.eclipse.org
	Task List		
	Task Repositories		
	Team Repositories		
PyDev		3	See http://www.pydev.org/manual.html for all Views in this topic.
	Caught Exceptions		
	Code Coverage		
	Hierarchical View		
	Index View		
	Profile		
	PyDev Package Explorer		
	PyUnit		
	Referrers Views		
SVEditor			
	Design Hierarchy		
	Diagram		
	Hierarchy		
	Objects		
TCL		3	See help.eclipse.org
	TCL Doc.		
	TCL Functions		
	TCL Members		
	TCL Namespaces		
	TCL Namespaces/Classes		
	TCL Packages		
	TCL Projects		

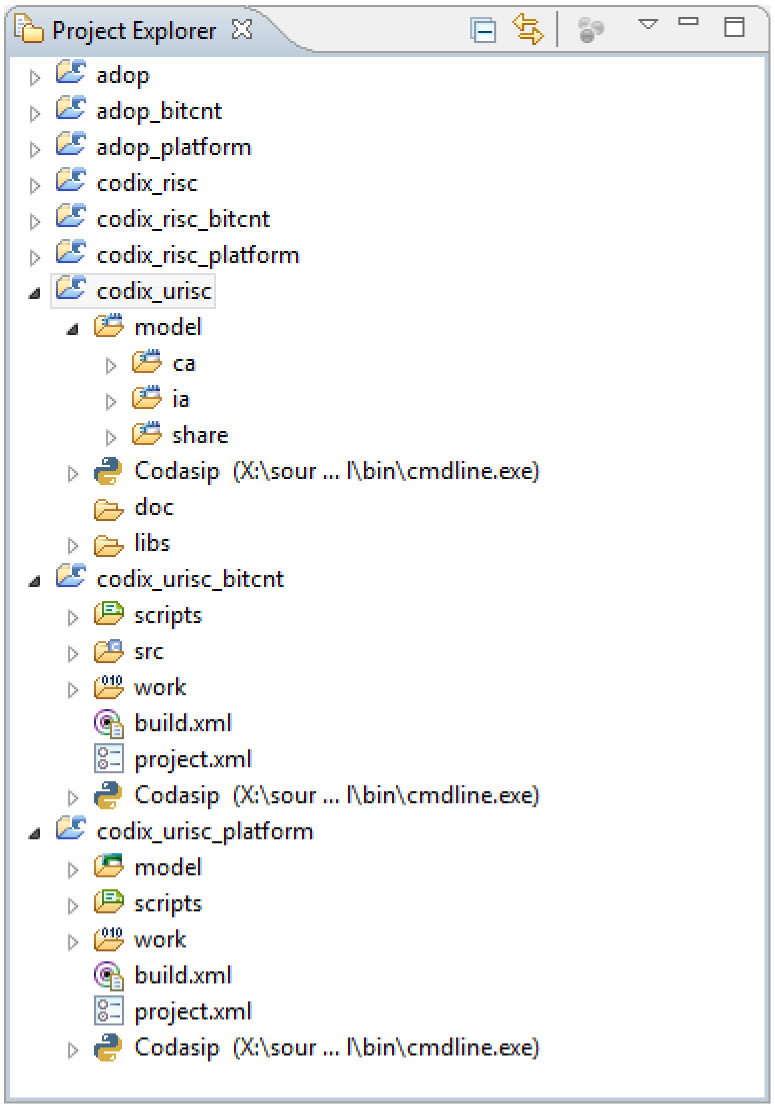
Topic	View name	Cat.	Comment
Team		3	See http://wiki.eclipse.org/EGit/User_Guide
	History		
	Synchronize		
Verilog/VHDL Editor			
	Hierarchy		
XML			
	Content Model		
	Documentation		

3.2 View Examples

This section illustrates some of the Views cited in the previous ones. Views not shown here can be seen using the command **Window** -> **Show View**-> **Other...** in the Codasip Studio.

3.2.1 Project Explorer

The **Project Explorer** View is used to display projects and their resources. Codasip adds specific icons for the folders and files recognized by Codasip Studio. Note that the rendered structure might not reflect the real filesystem structure if there's a more convenient way to display the resources.

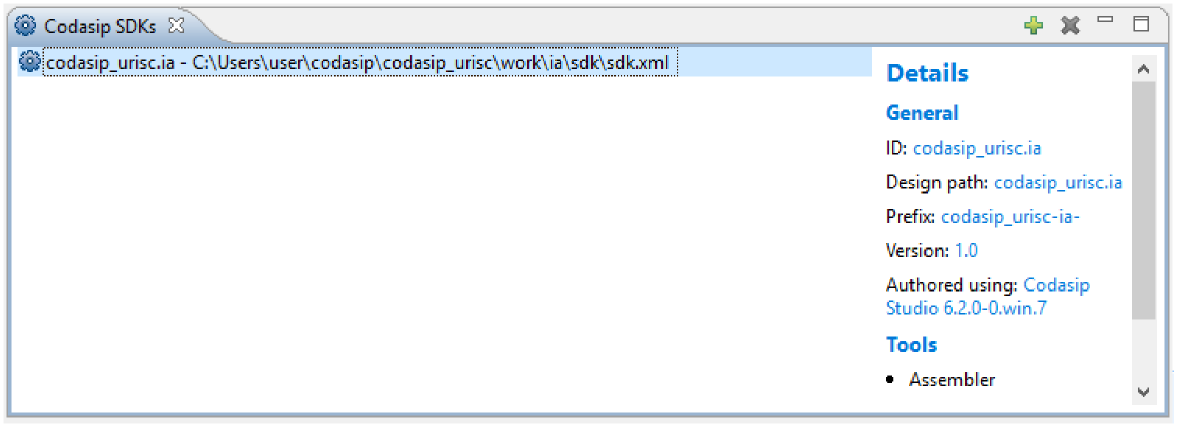


```
▷ adop
▷ adop_bitcnt
▷ adop_platform
▷ codix_risc
▷ codix_risc_bitcnt
▷ codix_risc_platform
◢ codix_urisc
    ◢ model
        ▷ ca
        ▷ ia
        ▷ share
    ▷ Codasip  (X:\sour ... l\bin\cmdline.exe)
      doc
    ▷ libs
◢ codix_urisc_bitcnt
    ▷ scripts
    ▷ src
    ▷ work
      build.xml
      project.xml
    ▷ Codasip  (X:\sour ... l\bin\cmdline.exe)
◢ codix_urisc_platform
    ▷ model
    ▷ scripts
    ▷ work
      build.xml
      project.xml
    ▷ Codasip  (X:\sour ... l\bin\cmdline.exe)
```

Figure 6 : Example Project Explorer View

3.2.2 Codasip SDKs

This View allows the user to manage the SDKs, which can then be used in various development environments, including Eclipse CDT (Eclipse support for C/C++ projects, also part of Codasip IDE). After the SDK is generated from the **Codasip Tasks** Viewin Codasip Studio, it is automaticly registered. Clicking on the green "plus" button allows you to register a new Codasip SDK, that have been generated previously outside the current workspace.



codasip_urisc.ia - C:\Users\user\codasip\codasip_urisc\work\ia\sdk\sdk.xml

Details

General

ID: codasip_urisc.ia

Design path: codasip_urisc.ia

Prefix: codasip_urisc-ia-

Version: 1.0

Authored using: Codasip Studio 6.2.0-0.win.7

Tools

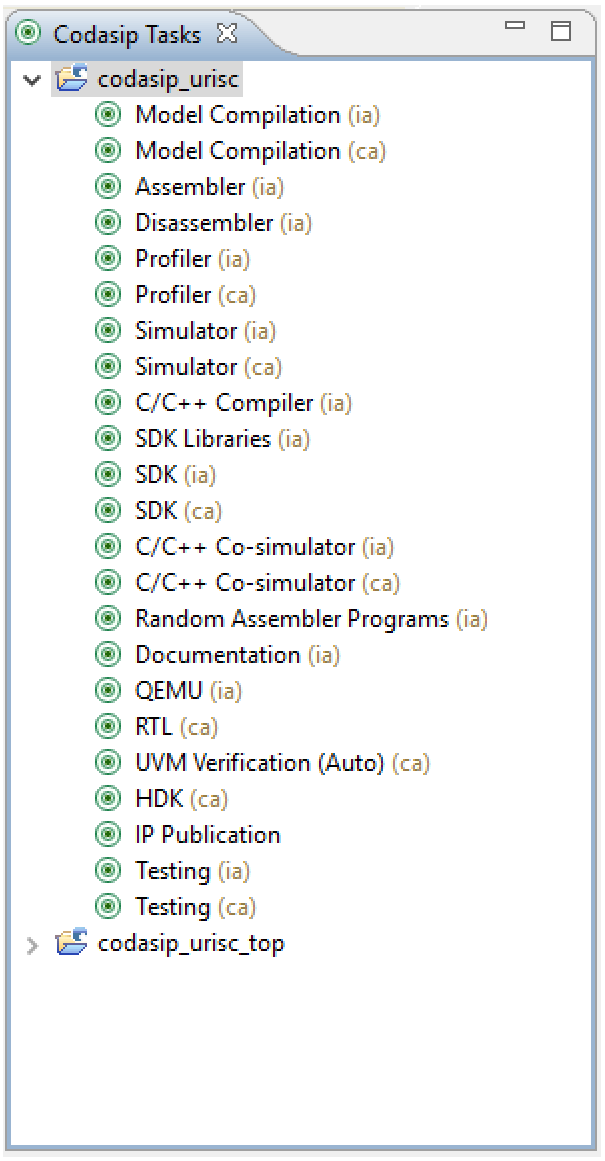
- Assembler

Figure 7 : A Codasip SDKs View showing one SDK

3.2.3 Codasip Tasks View

This View gives a list of build tasks - clicking on a given task will cause it to be built, together with all the tasks that it depends on.

The list of available targets is customizable through the tasks.py configuration file in CodAL projects. This functionality is tightly related to the Python scripting support – Python scripts are the behavioral part of the tasks.


Codasip Tasks

- codasip_urisc
  - Model Compilation (ia)
  - Model Compilation (ca)
  - Assembler (ia)
  - Disassembler (ia)
  - Profiler (ia)
  - Profiler (ca)
  - Simulator (ia)
  - Simulator (ca)
  - C/C++ Compiler (ia)
  - SDK Libraries (ia)
  - SDK (ia)
  - SDK (ca)
  - C/C++ Co-simulator (ia)
  - C/C++ Co-simulator (ca)
  - Random Assembler Programs (ia)
  - Documentation (ia)
  - QEMU (ia)
  - RTL (ca)
  - UVM Verification (Auto) (ca)
  - HDK (ca)
  - IP Publication
  - Testing (ia)
  - Testing (ca)
- codasip_urisc_top

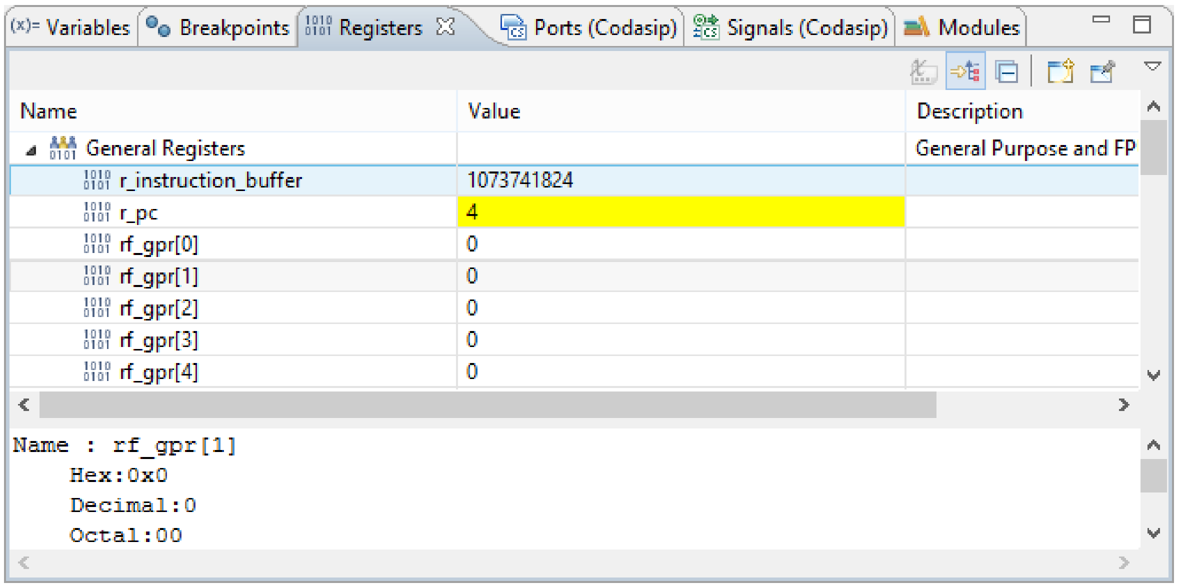
Figure 8 : Example Codasip Tasks View

3.2.4 IA/CA Consistency Checker View

This View displays results of IA/CA Consistency Checking, a verification tool builtin into Codasip Studio. The View is described in greater detail in the *Codasip Studio User Guide*, chapter "Verification Guide".

3.2.5 Debug Views

Several of the standard Eclipse Debug Views have been modified for Codasip Studio. The Registers View is one example:



Name	Value	Description
General Registers		General Purpose and FP
r_instruction_buffer	1073741824	
r_pc	4	
rf_gpr[0]	0	
rf_gpr[1]	0	
rf_gpr[2]	0	
rf_gpr[3]	0	
rf_gpr[4]	0	

```
Name : rf_gpr[1]
    Hex:0x0
    Decimal:0
    Octal:00
```

Figure 9 : Example Registers View

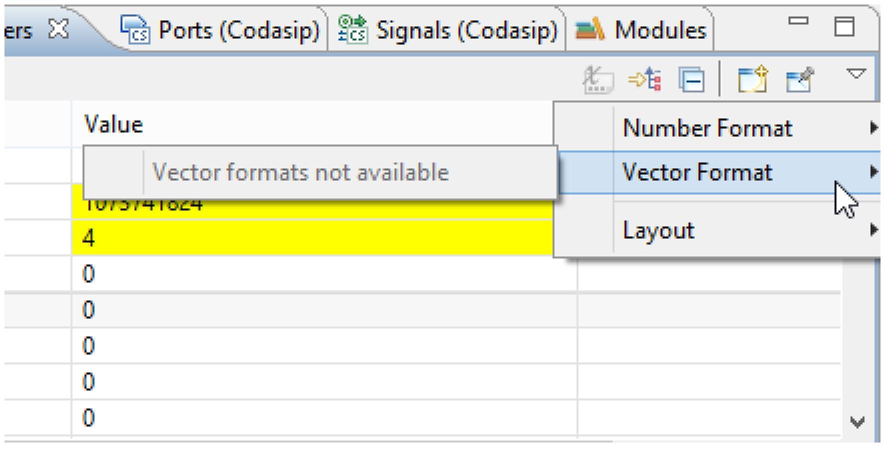
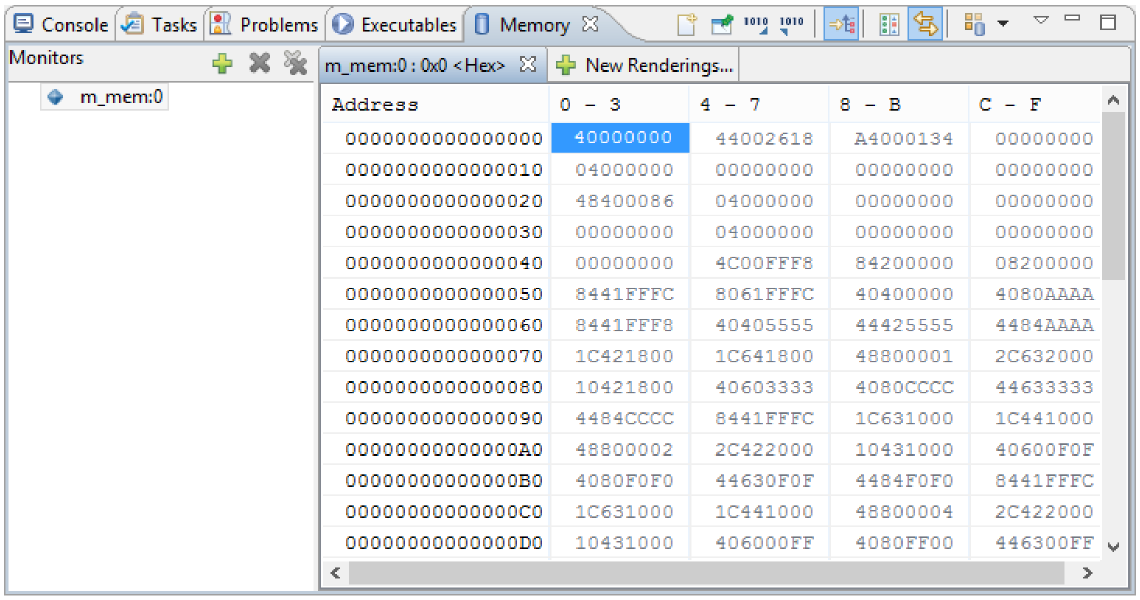


Figure 10 : Formatting control in the Registers View

The Memory View is also customised for Codasip Studio, allowing for multiple memories in a design:



```
Address            0 - 3     4 - 7     8 - B     C - F
0000000000000000   40000000  44002618  A4000134  00000000
0000000000000010   04000000  00000000  00000000  00000000
0000000000000020   48400086  04000000  00000000  00000000
0000000000000030   00000000  04000000  00000000  00000000
0000000000000040   00000000  4C00FFF8  84200000  08200000
0000000000000050   8441FFFC  8061FFFC  40400000  4080AAAA
0000000000000060   8441FFF8  40405555  44425555  4484AAAA
0000000000000070   1C421800  1C641800  48800001  2C632000
0000000000000080   10421800  40603333  4080CCCC  44633333
0000000000000090   4484CCCC  8441FFFC  1C631000  1C441000
00000000000000A0   48800002  2C422000  10431000  40600F0F
00000000000000B0   4080F0F0  44630F0F  4484F0F0  8441FFFC
00000000000000C0   1C631000  1C441000  48800004  2C422000
00000000000000D0   10431000  406000FF  4080FF00  446300FF
```

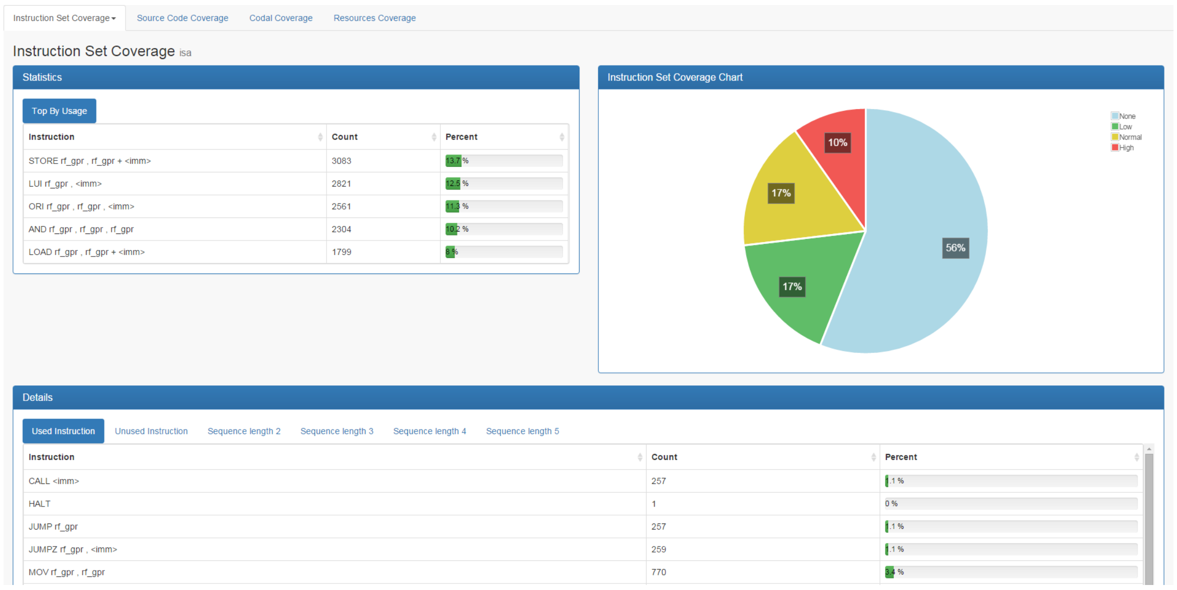
Figure 11 : Memory View

3.2.6 Profiling Result Web View

This View is in the Codasip Profiling category and it can be opened for each profiling session by double click on selected session in Profiling sessions view. Standard Eclipse internal web browser is embedded into this view for showing HTML content.

For each ASIP is possible to show information about:

- **Instruction Set Coverage** - information about each decoder. Instruction set coverage pie chart, Top instruction by usage, Used and unused instruction, Instruction sequences by length.
- **Source Code Coverage** - Flat model for functions, Call graph, Source code coverage pie chart, Address spaces with hits for reads and writes.
- **Codal Coverage** - Codal coverage by location, Codal coverage pie chart.
- **Resources Coverage** - Basic resources with read and write count, Caches with hits and miss count, Pipelines.



Instruction	Count	Percent
STORE rf_gpr , rf_gpr + <imm>	3083	13.7 %
LUI rf_gpr , <imm>	2821	12.5 %
ORI rf_gpr , rf_gpr , <imm>	2561	11.3 %
AND rf_gpr , rf_gpr , rf_gpr	2304	10.2 %
LOAD rf_gpr , rf_gpr + <imm>	1799	8 %

Instruction	Count	Percent
CALL <imm>	257	1.1 %
HALT	1	0 %
JUMP rf_gpr	257	1.1 %
JUMPZ rf_gpr , <imm>	259	1.1 %
MOV rf_gpr , rf_gpr	770	3.4 %

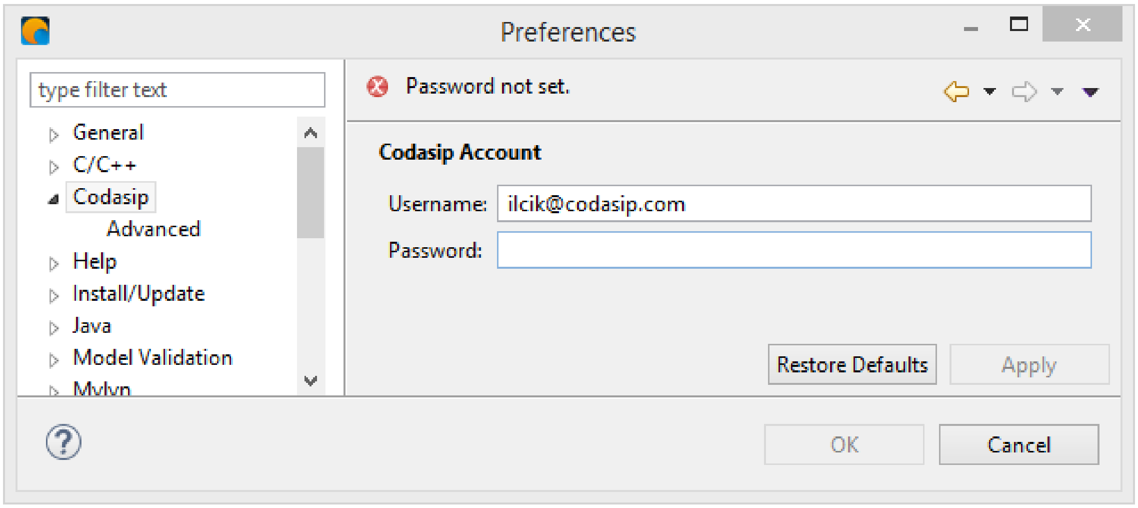
Figure 12 : Profiling result in HTML format

3.3 Codasip-specific Dialogs, Wizards and Preferences

Of the many Codasip-specific dialogs, wizards and preferences, the following deserve special mention.

3.3.1 Preferences

The organisation of Preferences reflects the organisation of Views and the associated dialogs can be accessed via **Window** → **Preferences**:



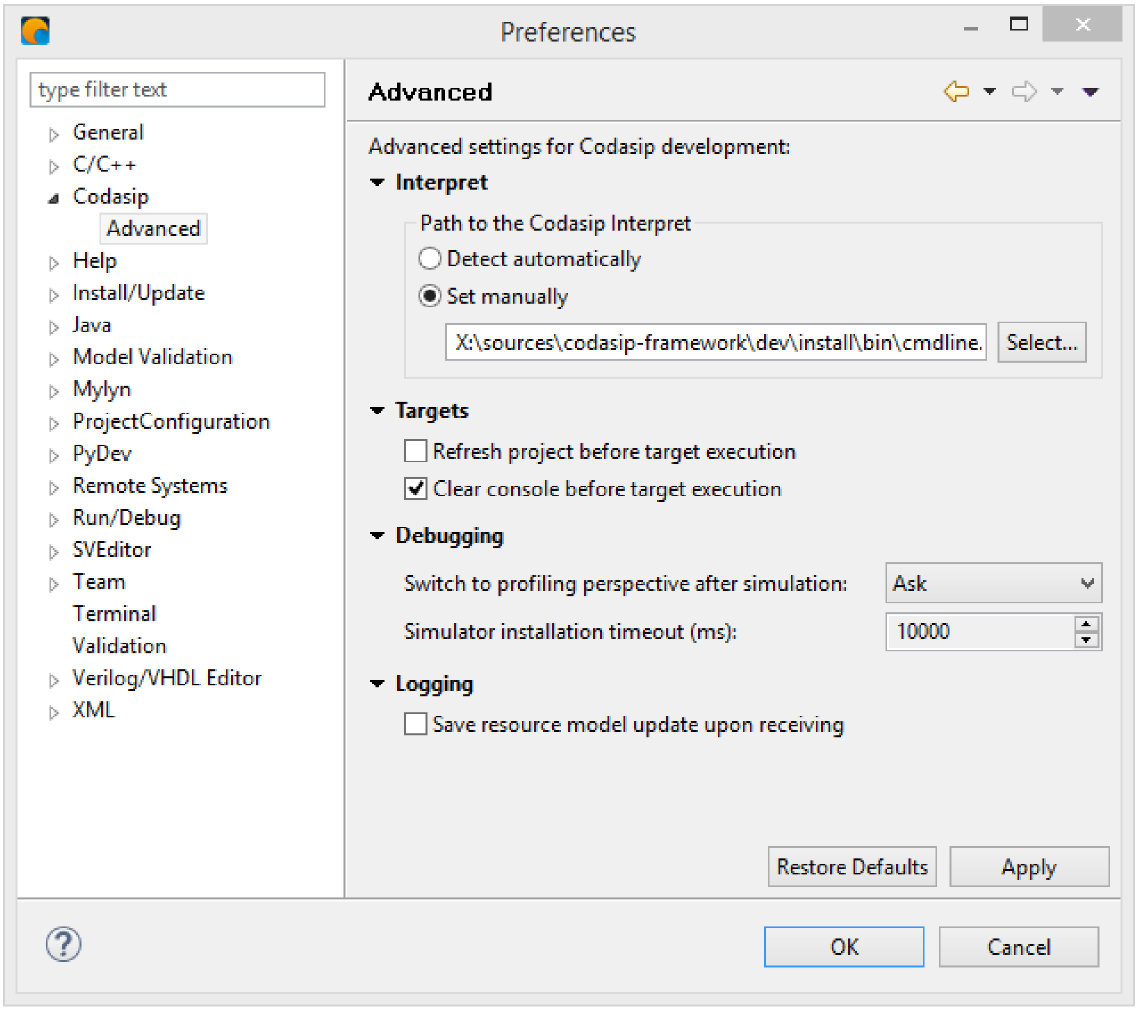
Password not set.

Codasip Account

Username: ilcik@codasip.com

Password:

Figure 13 : Accessing the Codasip Preferences



Advanced

Advanced settings for Codasip development:

Interpret

Path to the Codasip Interpret

Detect automatically

● Set manually

X:\sources\codasip-framework\dev\install\bin\cmdline.

Select...

Targets

☐ Refresh project before target execution

☑ Clear console before target execution

Debugging

Switch to profiling perspective after simulation: Ask

Simulator installation timeout (ms): 10000

Logging

☐ Save resource model update upon receiving

Figure 14 : Codasip Advanced Preferences

4 COMMAND-LINE

The Codasip Commandline tool is an essential part of the Codasip Studio. It is a Python interpreter (see http://docs.python.org) with built-in Python `codasip` module, which contains Codasip build system and all Codasip Studio generators and tools.

This chapter is organized as follows:

4.1 Documentation Conventions

The table below defines the syntax conventions used in Codasip Commandline commands.

`-l`	Letters starting with one dash indicate letters you enter literally. These letters represent short option names.
`--literal`	Words starting with two dashes indicate keywords you enter literally. These keywords represent long option names.
`|`	Vertical bars (OR-bars) separate possible choices for a single argument.
`<arguments>`	Words inside of angle brackets indicate arguments for which you must substitute a name or a value. When used with OR-bars, they enclose a list of choices from which you can choose one.
`[]`	Brackets indicate non-required options. More options inside of brackets indicate state option. All options inside of brackets are binded to the first one. `[-s <state> ... [-b <binded1>] [-b <binded2>]]`
`{}`	Braces indicate that a choice is required from the list of options separated by OR-bars. Choose one from the list. `{--option1|--option2|--option3}`
`...`	Three dots indicate that you can repeat the previous option.

4.2 Python commands reference

4.2.1 build_libraries

Copyright (C) 2017 Codasip Ltd. Build C/C++ libraries.

4.2.1.1 Usage

```
 --design-path <design-path> [--defines <string>] [--configuration-name <string>] [--include <file>] ... [--source-directory <folder>] [--library <lib>] [--prefix <prefix>] [-q] [-V] ... [--] [--version] [-h]
```

4.2.1.2 Descriptions

--design-path <design-path>
(required) Design path identifying single ASIP project instance.

--defines <string>
CMake defines for Newlib library.

--configuration-name <string>
Configuration name for Newlib library.

--include <file> (accepted multiple times)
Process file as input before processing the regular input file.

--source-directory <folder>
Source directory of library in model.

--library <lib>
Library to build.

--prefix <prefix>
Prefix for binaries.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.2 compile

Copyright (C) 2017 Codasip Ltd. Compile input CodAL source files into output XML model.

4.2.2.1 Usage

```
 {--ia|--ca} -o <output> -m <model> -p <project> <input> ...
[--Wnodiscarded] [--Wdiscarded] [--Wnoall-unused] [--Wall-
unused] [--Wnounused] [--Wunused] [--Wnobit-width] [--Wbit-
width] [--Wnoassignment] [--Wassignment] [--Wnosign] [--Wsign]
[--Wnoalias] [--Walias] [--Wnostd] [--Wstd] [--Werror] [--
Wall] [--Wnone] [--watermark] [--dump-codal-macros] [--
analyse] [-D <define>] ... [--include <file>] ... [-I <dir>]
... [--temp-path <temp-path>] [--preprocessor <command>] [-e
<extern>] ... [--design-path <design-path>] [-q] [-V] ... [--]
[--version] [-h]
```

4.2.2.2 Descriptions

Mutually exclusive:

--ia
(OR required) Enable IA mode of compilation (default).

--ca
(OR required) Enable CA mode of compilation.

-o <output>, --output <output>
(required) Path to output XML model.

-m <model>, --model <model>
(required) Name of a model.

-p <project>, --project <project>
(required) Name of a project.

<input> (accepted multiple times)
(required) Input source .codal files.

--Wnodiscarded
Disable discarded unused elements warnings

--Wdiscarded
Enable discarded unused elements warnings

--Wnoall-unused
Disable all unused ids warnings (includes checks with less importance)

--Wall-unused
Enable all unused ids warnings (includes checks with less importance)

--Wnounused
Disable unused ids warnings

--Wunused
Enable unused ids warnings

--Wnobit-width
Disable bit width warnings

--Wbit-width
Enable bit width warnings

--Wnoassignment
Disable assignment warnings

--Wassignment
Enable assignment warnings

--Wnosign
Disable sign warnings

--Wsign
Enable sign warnings

--Wnoalias
Disable ISA alias warnings

--Walias
Enable ISA alias warnings

--Wnostd
Disable standard warnings

--Wstd
Enable standard warnings

--Werror
Treat all warnings as errors.

--Wall
Enable all warnings.

--Wnone
Disable all warnings.

--watermark
Dump watermark of the compiler model.

--dump-codal-macros
Dump preprocessor macros defined internally in codalc.

--analyse
Analyse and report the configuration of the project.

-D <define>, --define <define> (accepted multiple times)
Preprocessor defines. Will be used for preprocessing source files.

--include <file> (accepted multiple times)
Process file as input before processing the regular input file.

-I <dir>, --includedir <dir> (accepted multiple times)
Include directory. Will be used for preprocessing source files.

--temp-path <temp-path>
Path to temporary directory for storing intermediate files.

--preprocessor <command>
Custom preprocessor command.

-e <extern>, --extern <extern> (accepted multiple times)
Links extern constructs, format: <name>,<XML path>.

--design-path <design-path>
Design path for multi-level compilation. Items are project name and model name separated by dot.
e.g. for project top.ca.middle.share.asip.ca3
 project name: asip
 model name: ca3
 design path: top.ca.middle.share.asip.ca3

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.3 copy_sdk_tools

Copyright (C) 2017 Codasip Ltd. Copy tools to SDK.

4.2.3.1 Usage

```
 <dir> [--doc] [--prefix <prefix>] [-q] [-V] ... [--] [--
version] [-h]
```

4.2.3.2 Descriptions

<dir>
(required) Directory where SDK tools will be generated to.

--doc
Copy GNU man and info documentation.

--prefix <prefix>
Prefix for binaries.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.4 extract_semantics

Copyright (C) 2017 Codasip Ltd. Extract instruction semantics of ASIP. Instruction semantics (by default) are used for generation of C compiler backend. Documentation semantics (-d) are used for generation of instruction documentation. Finally simulation semantics (-s) are used for instruction simulator generation.

4.2.4.1 Usage

```
 <xml> [-D <define>] ... [--include <file>] ... [-a <abi>] [--
no-debug] [-t] [-c] [-b] [-r] [-d] [-s] [--no-binary] [-q] [-
V] ... [--] [--version] [-h]
```

4.2.4.2 Descriptions

<xml>
(required) Input XML model file

-D <define>, --define <define> (accepted multiple times)
Preprocessor defines

--include <file> (accepted multiple times)
Process file as input before processing the regular input file.

-a <abi>, --abi <abi>
User specified semantics

--no-debug
Disables creating of debug files. By default files are created.

-t, --decorate
Decorates local variables of user functions and subinstructions with the name of the function or subinstruction in order to make variable names unique when the function or subinstruction is inlined in the instruction semantics.

-c, --compact
Instead of printing instruction semantics in SSA form, print it more like an expression. The resulting semantics description is usually more readable, but cannot be parsed. This is useful mainly for manual semantics checking.

-b, --builtin-generator
Builtin generator semantics are generated.

-r, --randomgen
Randomgen semantics are generated.

-d, --doc
Documentation semantics are generated.

-s, --sim
Semantic actions are considered for generation of simulator semantics.

--no-binary
Disables printing of instruction binary coding. By default printing is enabled.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.5 generate_asm

Copyright (C) 2017 Codasip Ltd. Generator of ASIP assembler.

4.2.5.1 Usage

```
 <file> [-P <number>] [-O <number>] [--include <file>] ... [--
user-parser <yy>] [--user-scanner <l>] [--base-parser <yy>] [-
-base-scanner <l>] [--default-linker-args <args>] [--linker-
script <lds>] ... [--svg] [--version-text <file>] [-q] [-V]
... [--prefix <prefix>] [--] [--version] [-h]
```

4.2.5.2 Descriptions

<file>
(required) Input XML model file.

-P <number>, --parallel-build <number>
Number of CPUs that are used for a build.

-O <number>, --optimization <number>
Optimization level. 0 (default) for no optimization, 3 for maximum.

--include <file> (accepted multiple times)
Process file as input before processing the regular input file.

--user-parser <yy>

Custom user assembler parser implemented in Bison.

--user-scanner <l>
Custom user assembler scanner implemented in Flex.

--base-parser <yy>
Custom main assembler parser implemented in Bison.

--base-scanner <l>
Custom main assembler scanner implemented in Flex.

--default-linker-args <args>
Default arguments that will be always used when linker is run.

--linker-script <lds> (accepted multiple times)
Custom linker script to be copied into the SDK

--svg
Generate debug SVG files.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--prefix <prefix>
Prefix for binaries.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.6 generate_ccompiler

Copyright (C) 2017 Codasip Ltd. Generate backend and compile C compiler.

4.2.6.1 Usage

```
 -a <arch> <semantics> [-O <number>] [--default-clang-args
<args>] [--include <file>] ... [-C <dir>] [-s] [-L <log>] [-H]
[-l] [-u <emulations>] [--version-text <file>] [-q] [-V] ...
[--prefix <prefix>] [--] [--version] [-h] <file> ...
```

4.2.6.2 Descriptions

-a <arch>, --arch <arch>
(required) Name of architecture. Can be used in C/C++ source files to determine type of compiler used.

<semantics>
(required) Input instruction semantics file (.sem).

-O <number>, --optimization <number>
Optimization level. 0 (default) for no optimization, 3 for maximum.

--default-clang-args <args>
Default arguments that will be always used when clang is run.

--include <file> (accepted multiple times)
Process file as input before processing the regular input file.

-C <dir>, --custom-lib-path <dir>
Optional path to custom LLVM libraries. (used only when executed through Codasip commandline)

-s, --svg
Allows inclusion of semantics graphs (in SVG) in the HTML report.

-L <log>, --log <log>
Filename of log of instruction processing. HTML log will be generated as well, in a directory 'instr_report'.

-H, --verbose-help
Enables printing of verbose internal help/hints - deprecated

-l, --lines
Enables printing of line macros

-u <emulations>, --emulations <emulations>
Defines an user emulation file. Default file rulelib.rl will be created if not specified.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--prefix <prefix>
Prefix for binaries.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

<file> (accepted multiple times)
Additional files used for overwriting default generated files.

4.2.7 generate_component

Copyright (C) 2017 Codasip Ltd. Create plugin project.

4.2.7.1 Usage

```
 -o <output> <config> [--uvm-unique-files] [--uvm-func-
coverage] [--uvm-code-coverage] [--uvm-abv] [-f] [--rtl-
testbenches <user|auto|jtag>] ... [--rtl-tck-frequency
<frequency-in-MHz>] [--rtl-clk-frequency <frequency-in-MHz>]
[--rtl-synthesis-tool <cadence_rtl_compiler|cadence_
genus|xilinx_ise|xilinx_vivado>] ... [--rtl-name-suffix-module
<string-id>] [--rtl-name-prefix-module <string-id>] [--rtl-
name-suffix-instance <string-id>] [--rtl-name-prefix-instance
<string-id>] [--rtl-name-suffix-signal <string-id>] [--rtl-
name-prefix-signal <string-id>] [--rtl-name-suffix-port
<string-id>] [--rtl-name-prefix-port <string-id>] [--rtl-name-
case-instance <none|lower|upper>] [--rtl-name-case-signal
<none|lower|upper>] [--rtl-name-case-port <none|lower|upper>]
[--rtl-name-case-module <none|lower|upper>] [--rtl-name-case-
top-port <none|lower|upper>] [--rtl-name-case-top
<none|lower|upper>] [--rtl-name-scan-mode <string-id>] [--rtl-
name-reset2 <string-id>] [--rtl-name-reset <string-id>] [--
rtl-name-clock <string-id>] [--rtl-heap-stack] [--rtl-comments
<length>] [--rtl-reset-synchronous] [--rtl-reset-active-high]
[-l <vhdl|verilog|sverilog>] [--version-text <file>] [-q] [-V]
... [--] [--version] [-h]
```

4.2.7.2 Descriptions

-o <output>, --output <output>
(required) Path to output directory.

<config>
(required) Path to configuration file.

--uvm-unique-files
Generates unique names for files of the UVM to simplify compilation under different HVL simulators.

--uvm-func-coverage
Enables / disables collection of functional coverage during functional verification.

--uvm-code-coverage
Enables / disables collection of code coverage during functional verification.

--uvm-abv
Enables / disables assertion based verification support in generated UVM.

-f, --uvm
Generate UVM functional verification environment.

--rtl-testbenches <user|auto|jtag> (accepted multiple times)
Generates HDL testbenches of specific type.

--rtl-tck-frequency <frequency-in-MHz>
Specify target frequency in MHz for JTAG clock signal (used by synthesis and simulation tools).

--rtl-clk-frequency <frequency-in-MHz>
Specify target frequency in MHz for global clock signal (used by synthesis and simulation tools).

--rtl-synthesis-tool <cadence_rtl_compiler|cadence_genus|xilinx_ise|xilinx_vivado> (accepted multiple times)

Generate RTL synthesis scripts for this tool.

--rtl-name-suffix-module <string-id>
Sets the suffix used for all project internal modules/entities.

--rtl-name-prefix-module <string-id>
Sets the prefix used for all project internal modules/entities.

--rtl-name-suffix-instance <string-id>
Sets the suffix used for all module/entity instance identifiers inside the project.

--rtl-name-prefix-instance <string-id>
Sets the prefix used for all module/entity instance identifiers inside the project.

--rtl-name-suffix-signal <string-id>
Sets the suffix used for all internal signal identifiers.

--rtl-name-prefix-signal <string-id>
Sets the prefix used for all internal signal identifiers.

--rtl-name-suffix-port <string-id>
Sets the suffix used for all internal port identifiers.

--rtl-name-prefix-port <string-id>
Sets the prefix used for all internal port identifiers.

--rtl-name-case-instance <none|lower|upper>
Sets the case level for all module/entity instance identifiers inside the project. The instance identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-signal <none|lower|upper>
Sets the case level for all internal signal identifiers inside the project. The signal identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-port <none|lower|upper>
Sets the case level for all internal port identifiers inside the project sub-hierarchy. The port identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-module <none|lower|upper>

Sets the case level for all project internal modules/entities. The module/entity identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-top-port <none|lower|upper>
Sets the case level for all port identifiers of the project top module/entity. The port identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-top <none|lower|upper>
Sets the case level for the project top module/entity. The project identifier will use upper/lower case English letters only or remain untouched.

--rtl-name-scan-mode <string-id>
Name of the default scan mode input port. This identifier will be used across the whole design.

--rtl-name-reset2 <string-id>
Name of the second reset input port. This identifier will be used across the whole design.The second reset port will be used only when the model contains both synchronous and asynchronous reset configuration.

--rtl-name-reset <string-id>
Name of the default reset input port. This identifier will be used across the whole design.

--rtl-name-clock <string-id>
Name of the default clock input port. This identifier will be used across the whole design.

--rtl-heap-stack
RTL memory initialization procedure skips ".heap" and ".stack" section by default. This flag forces their initialization in RTL source code.

--rtl-comments <length>
Add source file and line comment before every statement in generated RTL files including original C code. [length] limits length of the comment.

--rtl-reset-synchronous
Enable registers with synchronous reset instead of asynchronous (default).

--rtl-reset-active-high
Set reset active level to high, default is low.

-l <vhdl|verilog|sverilog>, --lang <vhdl|verilog|sverilog>
Hardware description language (HDL) that will be used as output for RTL description.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.8 generate_doc

Copyright (C) 2017 Codasip Ltd. Generate documentation of instructions for given ASIP.

4.2.8.1 Usage

```
 -s <semantics> -m <model> [-f <rtf|html|htm>] [-q] [-V] ...
[--] [--version] [-h] <file> ...
```

4.2.8.2 Descriptions

-s <semantics>, --semantics <semantics>
(required) Path to documentation semantics file.

-m <model>, --model <model>
(required) Path to ASIP XML model.

-f <rtf|html|htm>, --format <rtf|html|htm>
Format for generation of documentation. HTML by default.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)

Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

<file> (accepted multiple times)
Additional user file. This file will overwrite generated file with same name.

4.2.9 generate_prof

Copyright (C) 2017 Codasip Ltd. Generator of profiler.

4.2.9.1 Usage

```
 <file> [-P <number>] [-O <number>] [--svg] [--version-text
<file>] [-q] [-V] ... [--prefix <prefix>] [--] [--version] [-
h]
```

4.2.9.2 Descriptions

<file>
(required) Input XML model file.

-P <number>, --parallel-build <number>
Number of CPUs that are used for a build.

-O <number>, --optimization <number>
Optimization level. 0 (default) for no optimization, 3 for maximum.

--svg
Generate debug SVG files.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)

Increases verbosity of output.

--prefix <prefix>
Prefix for binaries.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.10 generate_random_asm

Copyright (C) 2017 Codasip Ltd. Run randomgen script.

4.2.10.1 Usage

```
 -i <instructions> <sem> [--use-smt-solver] [--skip-
constraint-generation] [--include <file>] ... [--constraints
<files>] ... [--smt-reference <file>] [-s <seed>] [-c <count>]
[--batch-scripts <folder>] [--smt-solver-timeout <number>] [--
use-undefined] [--memory-size <number>] [-m <file>] [-q] [-V]
... [--] [--version] [-h]
```

4.2.10.2 Descriptions

-i <instructions>, --instructions <instructions>
(required) Maximum number of instructions in generated ASM program.

<sem>
(required) Path to input simulation semantics file.

--use-smt-solver
Generates tests with usage of SMT solver.

--skip-constraint-generation
Skip generation of constraint files.

--include <file> (accepted multiple times)
Process file as input before processing the regular input file.

--constraints <files> (accepted multiple times)

Constraints files. Either generated or user specified. Order of these files is important.

--smt-reference <file>
SMT reference file, required when use-smt-solver is provided.

-s <seed>, --seed <seed>
Seed used to initialize random number generator. If not set, current time is used.

-c <count>, --count <count>
Count of random assembler files to be generated.

--batch-scripts <folder>
Folder for output batch scripts, required when use-smt-solver is provided.

--smt-solver-timeout <number>
Timeout in milliseconds for SMT solver.

--use-undefined
Use instructions with undefined semantics.

--memory-size <number>
Memory size

-m <file>, --model <file>
Input XML model file.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.11 generate_rtl

Copyright (C) 2017 Codasip Ltd. Generator of HDL description of ASIP or platform.

4.2.11.1 Usage

```
 -l <vhdl|verilog|sverilog> <file> [-a <asip> ... [--nexus-
watchpoints] [--nexus-breakpoints <0-8>] [--nexus-bsc
<resources>] ... [--nexus-interfaces <ifaces>] ... [--jtag-
instr-nexus-control <1-(2^IRSIZE-2)>] [--jtag-instr-nexus-
access <1-(2^IRSIZE-2)>] [--jtag-instr-idcode <1-(2^IRSIZE-
2)>] [--jtag-ir-size <4-64>] [--jtag-id <32-bit value>] [--
init <xexe>]] [--fpga-bscan-dev-before <32-bit value>] [--
fpga-bscan-ir-before <32-bit value>] [--fpga-bscan-ir-size <4-
64>] [--fpga-bscan-user-code <4-64>] [--fpga-bscan-jtag-chain
<32-bit value>] [--fpga-bscan] [--nexus-interfaces <ifaces>]
... [--dft-scan-mode] [--nexus-clock-control] [--rtl-
simulation-tool <questa|riviera|rtlsim>] ... [--disable-res-
optimizing] [-w] [--memory-latencies <memory-read-write-
latencies>] [--memory-size <memory-size>] [--rtl-tck-frequency
<frequency-in-MHz>] [--rtl-clk-frequency <frequency-in-MHz>]
[--rtl-synthesis-tool <cadence_rtl_compiler|cadence_
genus|xilinx_ise|xilinx_vivado>] ... [--rtl-name-suffix-module
<string-id>] [--rtl-name-prefix-module <string-id>] [--rtl-
name-suffix-instance <string-id>] [--rtl-name-prefix-instance
<string-id>] [--rtl-name-suffix-signal <string-id>] [--rtl-
name-prefix-signal <string-id>] [--rtl-name-suffix-port
<string-id>] [--rtl-name-prefix-port <string-id>] [--rtl-name-
case-instance <none|lower|upper>] [--rtl-name-case-signal
<none|lower|upper>] [--rtl-name-case-port <none|lower|upper>]
[--rtl-name-case-module <none|lower|upper>] [--rtl-name-case-
top-port <none|lower|upper>] [--rtl-name-case-top
<none|lower|upper>] [--rtl-name-scan-mode <string-id>] [--rtl-
name-reset2 <string-id>] [--rtl-name-reset <string-id>] [--
rtl-name-clock <string-id>] [--rtl-dff-delay <delay-in-pico-
seconds>] [--rtl-sync-reg-length <2-64>] [--rtl-technology
<asic|xilinx-artix-7|xilinx-virtex-5|xilinx-spartan-3>] [--
rtl-opt-fu-act-aux] [--rtl-opt-fu-act] [--rtl-opt-fu-input-
ops] [--rtl-opt-fu-always-act] [--rtl-opt-mux-select-max <sel-
bits>] [--rtl-cnd-nesting-level <level>] [--rtl-heap-stack] [-
-rtl-comments <length>] [--rtl-reset-synchronous] [--rtl-
reset-active-high] [--hdl-templates <directory>] [-e <extern>]
... [--rtl-testbenches <user|auto|jtag>] ... [--svg] [--
version-text <file>] [-q] [-V] ... [--prefix <prefix>] [--] [-
-version] [-h]
```

4.2.11.2 Descriptions

-l <vhdl|verilog|sverilog>, --lang <vhdl|verilog|sverilog>
(required) Hardware description language (HDL) that will be used as output for RTL description.

<file>
(required) Input XML model file.

-a <asip>, --asip <asip> (accepted multiple times)
Following arguments modifies settings for given ASIP type or instance.

--nexus-watchpoints
Enable watchpoint support in Nexus breakpoint modules.

--nexus-breakpoints <0-8>
Specify how many breakpoint modules to include in Nexus debugger. Max. value is 8.

--nexus-bsc <resources> (accepted multiple times)
Generate boundary scan-chain (JTAG) support for given component instance(s). Specifies resources to access as well.

--nexus-interfaces <ifaces> (accepted multiple times)
Enable Nexus client support for given memory interface(s).

--jtag-instr-nexus-control <1-(2^IRSIZE-2)>
Specify value of JTAG NEXUS_CONTROL instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-instr-nexus-access <1-(2^IRSIZE-2)>
Specify value of JTAG NEXUS_ACCESS instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-instr-idcode <1-(2^IRSIZE-2)>
Specify value of JTAG IDCODE instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-ir-size <4-64>
Specify bit width (IRSIZE) of the instruction register inside JTAG TAP (Test Access Port) controller. Min. value is 4 bits, max. value is 64.

--jtag-id <32-bit value>
Specify identifier of the JTAG TAP (Test Access Port) controller. Note that the least significant bit must be set to 1.

--init <xexe>
For given ASIP instance initializes content of memories from program using ASIP's interfaces. Program's data are directly stored in generated HDL description.

--fpga-bscan-dev-before <32-bit value>
Specifies number of devices before the BSCAN cell.

--fpga-bscan-ir-before <32-bit value>
Specifies number of bits before the BSCAN cell.

--fpga-bscan-ir-size <4-64>
Specifies bit-width of the instruction register in BSCAN cell.

--fpga-bscan-user-code <4-64>
Specifies USER code for FPGA BSCAN cell.

--fpga-bscan-jtag-chain <32-bit value>
Specify JTAG_CHAIN parameter for BSCAN cells, required only when using xilinx-artix-7 or xilinx-virtex-5 target devices.

--fpga-bscan
Use embedded Boundary Scan (BSCAN) cell for connecting internal logic to JTAG interface available on FPGA.

--nexus-interfaces <ifaces> (accepted multiple times)
Enable Nexus client support for given memory interface(s).

--dft-scan-mode
Enable support for DFT (Design For Test) by creating global scan mode input port.

--nexus-clock-control
Generate logic used as global clock buffer with clock enable for ASIP that is used during reset of the ASIP.

--rtl-simulation-tool <questa|riviera|rtlsim> (accepted multiple times)
Specify RTL simulation tool that uses a generated co-simulator. Also RTL testbench and verification startup scripts will be generated for this tool.

--disable-res-optimizing
Disable optimizing of resources, that are never read.

-w, --watchpoints

Enable support for HW watchpoints.

--memory-latencies <memory-read-write-latencies>
Read/Write Latency(ies) of memory. Format: r1,...,rn:w1,...,wn

--memory-size <memory-size>
Size for automatically inferred memory.

--rtl-tck-frequency <frequency-in-MHz>
Specify target frequency in MHz for JTAG clock signal (used by synthesis and simulation tools).

--rtl-clk-frequency <frequency-in-MHz>
Specify target frequency in MHz for global clock signal (used by synthesis and simulation tools).

--rtl-synthesis-tool <cadence_rtl_compiler|cadence_genus|xilinx_ise|xilinx_vivado> (accepted multiple times)
Generate RTL synthesis scripts for this tool.

--rtl-name-suffix-module <string-id>
Sets the suffix used for all project internal modules/entities.

--rtl-name-prefix-module <string-id>
Sets the prefix used for all project internal modules/entities.

--rtl-name-suffix-instance <string-id>
Sets the suffix used for all module/entity instance identifiers inside the project.

--rtl-name-prefix-instance <string-id>
Sets the prefix used for all module/entity instance identifiers inside the project.

--rtl-name-suffix-signal <string-id>
Sets the suffix used for all internal signal identifiers.

--rtl-name-prefix-signal <string-id>
Sets the prefix used for all internal signal identifiers.

--rtl-name-suffix-port <string-id>
Sets the suffix used for all internal port identifiers.

--rtl-name-prefix-port <string-id>
Sets the prefix used for all internal port identifiers.

--rtl-name-case-instance <none|lower|upper>
Sets the case level for all module/entity instance identifiers inside the project. The instance identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-signal <none|lower|upper>
Sets the case level for all internal signal identifiers inside the project. The signal identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-port <none|lower|upper>
Sets the case level for all internal port identifiers inside the project sub-hierarchy. The port identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-module <none|lower|upper>
Sets the case level for all project internal modules/entities. The module/entity identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-top-port <none|lower|upper>
Sets the case level for all port identifiers of the project top module/entity. The port identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-top <none|lower|upper>
Sets the case level for the project top module/entity. The project identifier will use upper/lower case English letters only or remain untouched.

--rtl-name-scan-mode <string-id>
Name of the default scan mode input port. This identifier will be used across the whole design.

--rtl-name-reset2 <string-id>
Name of the second reset input port. This identifier will be used across the whole design.The second reset port will be used only when the model contains both synchronous and asynchronous reset configuration.

--rtl-name-reset <string-id>
Name of the default reset input port. This identifier will be used across the whole design.

--rtl-name-clock <string-id>

Name of the default clock input port. This identifier will be used across the whole design.

--rtl-dff-delay <delay-in-pico-seconds>
Specify propagation delay (in picoseconds) to model Clock-to-Q delay of all flip-flop registers. The value will be hard-coded into generated RTL description. Default value of zero means no delay and no additional HDL code at all.

--rtl-sync-reg-length <2-64>
Specify length of the register used for JTAG clock synchronization (meta-stability treatment).

--rtl-technology <asic|xilinx-artix-7|xilinx-virtex-5|xilinx-spartan-3>
Specifies target technology to be used (ASIC vs. FPGA).

--rtl-opt-fu-act-aux
Reduce usage of activation condition inside functional units using auxiliary signal.

--rtl-opt-fu-act
Optimize away operand isolation logic created for activation of simple combinational C functional units with return value only.

--rtl-opt-fu-input-ops
Optimize away operand isolation logic created for data parameters of functional units.

--rtl-opt-fu-always-act
Removes activation ports in functional units that are always activated. The reset unit has to be empty and the unit must be activated unconditionally.

--rtl-opt-mux-select-max <sel-bits>
Limit the bit width of the selection condition for RTL multiplexer. Do not generate RTL multiplexers (case/switch) with large bit width of the selection condition.

--rtl-cnd-nesting-level <level>
Limit nesting of Verilog conditional operator (?) or VHDL when-else. The value defines maximum allowed number of nesting levels for the operator. Default zero value means no limit.

--rtl-heap-stack

RTL memory initialization procedure skips ".heap" and ".stack" section by default. This flag forces their initialization in RTL source code.

--rtl-comments <length>
Add source file and line comment before every statement in generated RTL files including original C code. [length] limits length of the comment.

--rtl-reset-synchronous
Enable registers with synchronous reset instead of asynchronous (default).

--rtl-reset-active-high
Set reset active level to high, default is low.

--hdl-templates <directory>
Path to hdl templates (e.g. cache templates).

-e <extern>, --extern <extern> (accepted multiple times)
Sources of extern constructs, format: <name>,<source path>.

--rtl-testbenches <user|auto|jtag> (accepted multiple times)
Generates HDL testbenches of specific type.

--svg
Generate debug SVG files.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--prefix <prefix>
Prefix for binaries.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.12 generate_sim

Copyright (C) 2017 Codasip Ltd. Generator of simulator and co-simulator.

4.2.12.1 Usage

```
 <file> [-a <asip> ... [--nexus-watchpoints] [--nexus-breakpoints <0-8>] [--nexus-bsc <resources>] ... [--nexus-interfaces <ifaces>] ... [--jtag-instr-nexus-control <1-(2^IRSIZE-2)>] [--jtag-instr-nexus-access <1-(2^IRSIZE-2)>] [--jtag-instr-idcode <1-(2^IRSIZE-2)>] [--jtag-ir-size <4-64>] [--jtag-id <32-bit value>]] [--fpga-bscan-dev-before <32-bit value>] [--fpga-bscan-ir-before <32-bit value>] [--fpga-bscan-ir-size <4-64>] [--fpga-bscan-user-code <4-64>] [--fpga-bscan-jtag-chain <32-bit value>] [--fpga-bscan] [--nexus-interfaces <ifaces>] ... [--dft-scan-mode] [--nexus-clock-control] [--rtl-simulation-tool <questa|riviera|rtlsim>] ... [--disable-res-optimizing] [-w] [-P <number>] [-O <number>] [--stack-overflow-check] [-d] [--memory-latencies <memory-read-write-latencies>] [--memory-size <memory-size>] [--profiler-rt-ppa] [--profiler-rt-decoders-coverage] [--profiler-rt-codal-coverage] [--profiler-rt-call-stack] [--profiler-rt-resources-coverage] [--codal-debug] [-D] [-p <L|H>] [-c <dpi|systemc|cpp>] [-e <extern>] ... [--svg] [--version-text <file>] [-q] [-V] ... [--prefix <prefix>] [--] [--version] [-h]
```

4.2.12.2 Descriptions

<file>
(required) Input XML model file.

-a <asip>, --asip <asip> (accepted multiple times)
Following arguments modifies settings for given ASIP type or instance.

--nexus-watchpoints
Enable watchpoint support in Nexus breakpoint modules.

--nexus-breakpoints <0-8>
Specify how many breakpoint modules to include in Nexus debugger. Max. value is 8.

--nexus-bsc <resources> (accepted multiple times)

Generate boundary scan-chain (JTAG) support for given component instance(s). Specifies resources to access as well.

--nexus-interfaces <ifaces> (accepted multiple times)
Enable Nexus client support for given memory interface(s).

--jtag-instr-nexus-control <1-(2^IRSIZE-2)>
Specify value of JTAG NEXUS_CONTROL instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-instr-nexus-access <1-(2^IRSIZE-2)>
Specify value of JTAG NEXUS_ACCESS instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-instr-idcode <1-(2^IRSIZE-2)>
Specify value of JTAG IDCODE instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-ir-size <4-64>
Specify bit width (IRSIZE) of the instruction register inside JTAG TAP (Test Access Port) controller. Min. value is 4 bits, max. value is 64.

--jtag-id <32-bit value>
Specify identifier of the JTAG TAP (Test Access Port) controller. Note that the least significant bit must be set to 1.

--fpga-bscan-dev-before <32-bit value>
Specifies number of devices before the BSCAN cell.

--fpga-bscan-ir-before <32-bit value>
Specifies number of bits before the BSCAN cell.

--fpga-bscan-ir-size <4-64>
Specifies bit-width of the instruction register in BSCAN cell.

--fpga-bscan-user-code <4-64>
Specifies USER code for FPGA BSCAN cell.

--fpga-bscan-jtag-chain <32-bit value>
Specify JTAG_CHAIN parameter for BSCAN cells, required only when using xilinx-artix-7 or xilinx-virtex-5 target devices.

--fpga-bscan

Use embedded Boundary Scan (BSCAN) cell for connecting internal logic to JTAG interface available on FPGA.

--nexus-interfaces <ifaces> (accepted multiple times)
Enable Nexus client support for given memory interface(s).

--dft-scan-mode
Enable support for DFT (Design For Test) by creating global scan mode input port.

--nexus-clock-control
Generate logic used as global clock buffer with clock enable for ASIP that is used during reset of the ASIP.

--rtl-simulation-tool <questa|riviera|rtlsim> (accepted multiple times)
Specify RTL simulation tool that uses a generated co-simulator. Also RTL testbench and verification startup scripts will be generated for this tool.

--disable-res-optimizing
Disable optimizing of resources, that are never read.

-w, --watchpoints
Enable support for HW watchpoints.

-P <number>, --parallel-build <number>
Number of CPUs that are used for a build.

-O <number>, --optimization <number>
Optimization level. 0 (default) for no optimization, 3 for maximum.

--stack-overflow-check
Generate simulator with support of stack overflow checking. When stack overflow occurs, error message is printed. When debugger is used simulation is suspended. When simulator is used without debugger (-r flag), simulation is terminated.

-d, --debugger
Enable debugger support.

--memory-latencies <memory-read-write-latencies>
Read/Write Latency(ies) of memory. Format: r1,...,rn:w1,...,wn

--memory-size <memory-size>
Size for automatically inferred memory.

--profiler-rt-ppa
Enable ppa info collection for profiler-rt.

--profiler-rt-decoders-coverage
Enable decoders coverage info collection for profiler-rt.

--profiler-rt-codal-coverage
Enable codal coverage info collection for profiler-rt.

--profiler-rt-call-stack
Enable call stack unwinder for profiler-rt.

--profiler-rt-resources-coverage
Enable resource coverage info collection for profiler-rt.

--codal-debug
Generate simulator with support of CodAL code debugging.

-D, --dump
Enable resource changes dump support.

-p <L|H>, --profiler-rt <L|H>
Generate profiler-rt into simulator, values are N (none), L (low), and H (high).

-c <dpi|systemc|cpp>, --cosim <dpi|systemc|cpp>
Generate co-simulator using given foreign interface.

-e <extern>, --extern <extern> (accepted multiple times)
Sources of extern constructs, format: <name>,<source path>.

--svg
Generate debug SVG files.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--prefix <prefix>
Prefix for binaries.

--, --ignore_rest

Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.13 generate_uvm

Copyright (C) 2017 Codasip Ltd. Generator of functional verification environment.

4.2.13.1 Usage

```
 -l <vhdl|verilog|sverilog> <file> [-a <asip> ... [--uvm-init
<xexe/zip>] [--nexus-watchpoints] [--nexus-breakpoints <0-8>]
[--nexus-bsc <resources>] ... [--nexus-interfaces <ifaces>]
... [--jtag-instr-nexus-control <1-(2^IRSIZE-2)>] [--jtag-
instr-nexus-access <1-(2^IRSIZE-2)>] [--jtag-instr-idcode <1-
(2^IRSIZE-2)>] [--jtag-ir-size <4-64>] [--jtag-id <32-bit
value>] [--init <xexe>]] [--uvm-unique-files] [--uvm-func-
coverage] [--uvm-code-coverage] [--uvm-abv] [--uvm-data-ver]
[--uvm-gm <xml>] [--uvm-auto] [-f] [--fpga-bscan-dev-before
<32-bit value>] [--fpga-bscan-ir-before <32-bit value>] [--
fpga-bscan-ir-size <4-64>] [--fpga-bscan-user-code <4-64>] [--
fpga-bscan-jtag-chain <32-bit value>] [--fpga-bscan] [--nexus-
interfaces <ifaces>] ... [--dft-scan-mode] [--nexus-clock-
control] [--rtl-simulation-tool <questa|riviera|rtlsim>] ...
[--disable-res-optimizing] [-w] [--stack-overflow-check] [-d]
[--memory-latencies <memory-read-write-latencies>] [--memory-
size <memory-size>] [--rtl-tck-frequency <frequency-in-MHz>]
[--rtl-clk-frequency <frequency-in-MHz>] [--rtl-synthesis-tool
<cadence_rtl_compiler|cadence_genus|xilinx_ise|xilinx_vivado>]
... [--rtl-name-suffix-module <string-id>] [--rtl-name-prefix-
module <string-id>] [--rtl-name-suffix-instance <string-id>]
[--rtl-name-prefix-instance <string-id>] [--rtl-name-suffix-
signal <string-id>] [--rtl-name-prefix-signal <string-id>] [--
rtl-name-suffix-port <string-id>] [--rtl-name-prefix-port
<string-id>] [--rtl-name-case-instance <none|lower|upper>] [--
rtl-name-case-signal <none|lower|upper>] [--rtl-name-case-port
<none|lower|upper>]      [-   -   rtl-   name-   case-   module
<none|lower|upper>]     [-   -   rtl-   name-   case-   top-   port
<none|lower|upper>] [--rtl-name-case-top <none|lower|upper>]
```

```
[--rtl-name-scan-mode <string-id>] [--rtl-name-reset2 <string-
id>] [--rtl-name-reset <string-id>] [--rtl-name-clock <string-
id>] [--rtl-dff-delay <delay-in-pico-seconds>] [--rtl-sync-
reg-length <2-64>] [--rtl-technology <asic|xilinx-artix-
7|xilinx-virtex-5|xilinx-spartan-3>] [--rtl-opt-fu-act-aux] [-
-rtl-opt-fu-act] [--rtl-opt-fu-input-ops] [--rtl-opt-fu-
always-act] [--rtl-opt-mux-select-max <sel-bits>] [--rtl-cnd-
nesting-level <level>] [--rtl-heap-stack] [--rtl-comments
<length>] [--rtl-reset-synchronous] [--rtl-reset-active-high]
[--hdl-templates <directory>] [-e <extern>] ... [--version-
text <file>] [-q] [-V] ... [--] [--version] [-h]
```

4.2.13.2 Descriptions

-l <vhdl|verilog|sverilog>, --lang <vhdl|verilog|sverilog>
(required) Hardware description language (HDL) that will be used as output for RTL description.

<file>
(required) Input XML model file.

-a <asip>, --asip <asip> (accepted multiple times)
Following arguments modifies settings for given ASIP type or instance.

--uvm-init <xexe/zip>
For given ASIP instance initializes content of memories from program or ZIP archive containing multiple programs. Program's data are loaded into foreign simulator during functional verification.

--nexus-watchpoints
Enable watchpoint support in Nexus breakpoint modules.

--nexus-breakpoints <0-8>
Specify how many breakpoint modules to include in Nexus debugger. Max. value is 8.

--nexus-bsc <resources> (accepted multiple times)
Generate boundary scan-chain (JTAG) support for given component instance(s). Specifies resources to access as well.

--nexus-interfaces <ifaces> (accepted multiple times)
Enable Nexus client support for given memory interface(s).

--jtag-instr-nexus-control <1-(2^IRSIZE-2)>
Specify value of JTAG NEXUS_CONTROL instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-instr-nexus-access <1-(2^IRSIZE-2)>
Specify value of JTAG NEXUS_ACCESS instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-instr-idcode <1-(2^IRSIZE-2)>
Specify value of JTAG IDCODE instruction (valid range <1 .. (2^IRSIZE-2)>).

--jtag-ir-size <4-64>
Specify bit width (IRSIZE) of the instruction register inside JTAG TAP (Test Access Port) controller. Min. value is 4 bits, max. value is 64.

--jtag-id <32-bit value>
Specify identifier of the JTAG TAP (Test Access Port) controller. Note that the least significant bit must be set to 1.

--init <xexe>
For given ASIP instance initializes content of memories from program using ASIP's interfaces. Program's data are directly stored in generated HDL description.

--uvm-unique-files
Generates unique names for files of the UVM to simplify compilation under different HVL simulators.

--uvm-func-coverage
Enables / disables collection of functional coverage during functional verification.

--uvm-code-coverage
Enables / disables collection of code coverage during functional verification.

--uvm-abv
Enables / disables assertion based verification support in generated UVM.

--uvm-data-ver
Enables / data verification support on ASIP interfaces.

--uvm-gm <xml>
: Path to XML model for golden model generation (ASIP/Level) used in automatic functional verification.

--uvm-auto
: Enables automatic functional verification.

-f, --uvm
: Generate UVM functional verification environment.

--fpga-bscan-dev-before <32-bit value>
: Specifies number of devices before the BSCAN cell.

--fpga-bscan-ir-before <32-bit value>
: Specifies number of bits before the BSCAN cell.

--fpga-bscan-ir-size <4-64>
: Specifies bit-width of the instruction register in BSCAN cell.

--fpga-bscan-user-code <4-64>
: Specifies USER code for FPGA BSCAN cell.

--fpga-bscan-jtag-chain <32-bit value>
: Specify JTAG_CHAIN parameter for BSCAN cells, required only when using xilinx-artix-7 or xilinx-virtex-5 target devices.

--fpga-bscan
: Use embedded Boundary Scan (BSCAN) cell for connecting internal logic to JTAG interface available on FPGA.

--nexus-interfaces <ifaces> (accepted multiple times)
: Enable Nexus client support for given memory interface(s).

--dft-scan-mode
: Enable support for DFT (Design For Test) by creating global scan mode input port.

--nexus-clock-control
: Generate logic used as global clock buffer with clock enable for ASIP that is used during reset of the ASIP.

--rtl-simulation-tool <questa|riviera|rtlsim> (accepted multiple times)
: Specify RTL simulation tool that uses a generated co-simulator. Also RTL testbench and verification startup scripts will be generated for this tool.

--disable-res-optimizing
Disable optimizing of resources, that are never read.

-w, --watchpoints
Enable support for HW watchpoints.

--stack-overflow-check
Generate simulator with support of stack overflow checking. When stack overflow occurs, error message is printed. When debugger is used simulation is suspended. When simulator is used without debugger (-r flag), simulation is terminated.

-d, --debugger
Enable debugger support.

--memory-latencies <memory-read-write-latencies>
Read/Write Latency(ies) of memory. Format: r1,...,rn:w1,...,wn

--memory-size <memory-size>
Size for automatically inferred memory.

--rtl-tck-frequency <frequency-in-MHz>
Specify target frequency in MHz for JTAG clock signal (used by synthesis and simulation tools).

--rtl-clk-frequency <frequency-in-MHz>
Specify target frequency in MHz for global clock signal (used by synthesis and simulation tools).

--rtl-synthesis-tool <cadence_rtl_compiler|cadence_genus|xilinx_ise|xilinx_vivado> (accepted multiple times)
Generate RTL synthesis scripts for this tool.

--rtl-name-suffix-module <string-id>
Sets the suffix used for all project internal modules/entities.

--rtl-name-prefix-module <string-id>
Sets the prefix used for all project internal modules/entities.

--rtl-name-suffix-instance <string-id>
Sets the suffix used for all module/entity instance identifiers inside the project.

--rtl-name-prefix-instance <string-id>

Sets the prefix used for all module/entity instance identifiers inside the project.

--rtl-name-suffix-signal <string-id>
Sets the suffix used for all internal signal identifiers.

--rtl-name-prefix-signal <string-id>
Sets the prefix used for all internal signal identifiers.

--rtl-name-suffix-port <string-id>
Sets the suffix used for all internal port identifiers.

--rtl-name-prefix-port <string-id>
Sets the prefix used for all internal port identifiers.

--rtl-name-case-instance <none|lower|upper>
Sets the case level for all module/entity instance identifiers inside the project. The instance identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-signal <none|lower|upper>
Sets the case level for all internal signal identifiers inside the project. The signal identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-port <none|lower|upper>
Sets the case level for all internal port identifiers inside the project sub-hierarchy. The port identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-module <none|lower|upper>
Sets the case level for all project internal modules/entities. The module/entity identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-top-port <none|lower|upper>
Sets the case level for all port identifiers of the project top module/entity. The port identifiers will use upper/lower case English letters only or remain untouched.

--rtl-name-case-top <none|lower|upper>
Sets the case level for the project top module/entity. The project identifier will use upper/lower case English letters only or remain untouched.

--rtl-name-scan-mode <string-id>

Name of the default scan mode input port. This identifier will be used across the whole design.

--rtl-name-reset2 <string-id>
Name of the second reset input port. This identifier will be used across the whole design.The second reset port will be used only when the model contains both synchronous and asynchronous reset configuration.

--rtl-name-reset <string-id>
Name of the default reset input port. This identifier will be used across the whole design.

--rtl-name-clock <string-id>
Name of the default clock input port. This identifier will be used across the whole design.

--rtl-dff-delay <delay-in-pico-seconds>
Specify propagation delay (in picoseconds) to model Clock-to-Q delay of all flip-flop registers. The value will be hard-coded into generated RTL description. Default value of zero means no delay and no additional HDL code at all.

--rtl-sync-reg-length <2-64>
Specify length of the register used for JTAG clock synchronization (meta-stability treatment).

--rtl-technology <asic|xilinx-artix-7|xilinx-virtex-5|xilinx-spartan-3>
Specifies target technology to be used (ASIC vs. FPGA).

--rtl-opt-fu-act-aux
Reduce usage of activation condition inside functional units using auxiliary signal.

--rtl-opt-fu-act
Optimize away operand isolation logic created for activation of simple combinational C functional units with return value only.

--rtl-opt-fu-input-ops
Optimize away operand isolation logic created for data parameters of functional units.

--rtl-opt-fu-always-act

Removes activation ports in functional units that are always activated. The reset unit has to be empty and the unit must be activated unconditionally.

--rtl-opt-mux-select-max <sel-bits>
Limit the bit width of the selection condition for RTL multiplexer. Do not generate RTL multiplexers (case/switch) with large bit width of the selection condition.

--rtl-cnd-nesting-level <level>
Limit nesting of Verilog conditional operator (?) or VHDL when-else. The value defines maximum allowed number of nesting levels for the operator. Default zero value means no limit.

--rtl-heap-stack
RTL memory initialization procedure skips ".heap" and ".stack" section by default. This flag forces their initialization in RTL source code.

--rtl-comments <length>
Add source file and line comment before every statement in generated RTL files including original C code. [length] limits length of the comment.

--rtl-reset-synchronous
Enable registers with synchronous reset instead of asynchronous (default).

--rtl-reset-active-high
Set reset active level to high, default is low.

--hdl-templates <directory>
Path to hdl templates (e.g. cache templates).

-e <extern>, --extern <extern> (accepted multiple times)
Sources of extern constructs, format: <name>,<source path>.

--version-text <file>
Model version text that will be part of --version output text.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest

Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

4.2.14 list_sdk_tools

Return list of SDK tool names. Can be used to check if all such tools are present in SDK.

4.2.15 parse_config

Parse Codasip configuration at given path and return Python dictionary with its content.

4.2.16 parse_config_check

Parse Codasip configuration at given path and return Python dictionary with its content. Check the configuration is valid Codasip Project configuration.

4.2.17 set_work_dir

Set current project working directory.

5 BUILD SYSTEM

The Codasip Studio projects are built by Codasip build system based on Python DoIt module (http://pydoit.org/). The build system allows users to modify, add, and automate build tasks, such as verification and non-regression testing. The same build system is also used by Codasip Studio.

The build system automatically loads project, its configuration and all referenced projects. It also manages dependencies between tasks, so only necessary tasks are executed. This greatly improves and speeds up usage of Codasip Studio tools.

This chapter is organized as follows:

5.1 Project

The Codasip build system searches current directory and tries to find Codasip project.

Currently there are 2 types of projects the build system can load:

1. CodAL project
    - Contains at least single source file (`.codal` extension) in `models` directory.
    - Optional configuration file `codal.conf`.
2. Component project
    - Contains configuration file `component.conf`.

When any of these projects is found, its loaded. If some error occurs, build system prints error message and terminates.

5.1.1 CodAL project hierarchy

Every CodAL project may consist of multiple models. The directory structure is described in "File Organization" on page 119".

Every model is named after directory the code is stored. The model itself is then denoted by `<project name>.<model name>`. E.g. for model `ia` of `codasip_urisc` project is referenced as `codasip_urisc.ia`.

If current project references another project, its default model for current model type (instruction or cycle accurate) is used:

1. Default models can be set in configuration.
2. If not set, first instruction or cycle accurate model of referenced project is used.

When referenced default model is found, it's added as reference of current model. This creates project-model hierarchy, that is used by build system to execute tasks. Every referenced model is then denoted by full path from top-level project model.

Example 1: Project hierarchy

```
Project: codasip_toplevel
    model: ia, ca, ca_test

Project codasip_urisc
    models: ia, ca, ca_test
    default IA model: ia
    default CA model: ca_test

codasip_urisc models full path:
    codasip_toplevel.ia.codasip_urisc.ia
    codasip_toplevel.ca.codasip_urisc.ca_test
    codasip_toplevel.ca_test.codasip_urisc.ca_test


Model codasip_urisc.ca is not used, because it is not set as default CA model.
```

5.2 Task

Task describes single tool generation or other action. Every task can have dependencies on other tasks or files and specifies its target files. Codasip build system then uses this information to correctly order tasks execution and skip already finished or up-to-date tasks.

All tasks are visible in Codasip Studio **Codasip Tasks** view or using `list` command.

Tasks are loaded from the project file `work/tasks.py`. If it is not present or it is older than Codasip Studio installation, the file is automatically recreated. To create own new tasks, create file `tasks.py` in project root directory. This file can overwrite default tasks or add new custom tasks.

5.3 Building from Command Line

The Codasip build system is implemented as Python module in Codasip Commandline.

To run the build system, use Codasip Commandline and module `codasip.build`. This module controls the build system using command-line arguments.

Example 2: Running Codasip build system

```
<Codasip Studio installation>/tools/bin/cmdline -m codasip.build <command> <arguments>
```

The command in this example will load project in current working directory.

The build system then executes given command with its arguments over the tasks. If no command is specified, `run` command is used.

Arguments:

- `--dir <path>` (or `-d`)
  Load project from given directory, not current working directory.
- `--work <path>`
  Set custom directory as working directory for project. All generated outputs, tools and reports will be generated into the directory. `<project directory>/work` by default.
- `--paths <path>,<path>,...`
  Search all given paths when referenced projects will be searched for. More information in "Project" on page 71. Can be used multiple times.

5.3.1 CodAL Project References

Every CodAL project can reference other CodAL or Component projects by usage of **`extern`** construct that is described in *CodAL Language Reference Manual*. References can also be specified in configuration file `codal.conf`.

Location of referenced project paths can be set by argument `--paths`. This argument can be used multiple times.

Example 3: Setting project locations and listing all tasks

```
cmdline -m codasip.build --paths C:\Projects,D:\Projects\codasip_urisc list
```

The location for every referenced project is tried in the following order:

- Directories specified by the argument `--paths` that have same name as the project.
- Directory inside directories set by the argument `--paths` that has the same name as the referenced project.
- Directory at the same level as the current project directory that has the same name as the referenced project.

When the project directory is found and the project is successfully loaded, its referenced projects are being loaded. This recursive loading continues until all referenced projects are found.

If the referenced project is not found, an error message is printed and the build system terminates.

5.4 Commands

When the project is successfully loaded, the build system will execute user specified command.

Main commands:

- `run`
  Execute the tasks. This command can be omitted at command line.
  - use `-s` to run single task without dependencies.
  - use `-a` to force running of task even if it is up-to-date,
  - use `-c` to continue, when some tasks fails.

  The last argument are IDs of tasks to be run. To select multiple tasks, regular expression can be used (e.g. `sim*` to build all simulators).
- `list`
  List all tasks, theirs IDs and names.
  - use `-s` to show status of every task (`R` = run - task will be executed, `U` = up-to-date - task won't be executed),
  - use `--all` to see complete hierarchy of tasks, not just top-level group tasks.
- `info`
  Print information about single task.
- `help`
  Prints all commands and its arguments including all previous mentioned commands and its arguments.

Example 4: Execution of task sim.ia and generation of all assemblers

```
cmdline -m codasip.build sim.ia asm*
```

Example 5: Command line `list` command with status of tasks

```
cmdline -m codasip.build list -s
U model.ia        Model Compilation
U model.ca        Model Compilation
R asm.ia          Assembler
R dsm.ia          Disassembler
```

```
R prof.ia         Profiler
R prof.ca         Profiler
R sim.ia          Simulator
R sim.ca          Simulator
U compiler.ia     C/C++ Compiler
R libs.ia         SDK Libraries
R sdk.ia          SDK
R sdk.ca          SDK
R publish_sdk     SDK Publication
R cosim.ia        C/C++ Co-simulator
R cosim.ca        C/C++ Co-simulator
R random_asm.ia   Random Assembler Programs
R doc.ia          DocumentationR rtl.ca          RTL
R uvm.ca          UVM Verification (Auto)
R publish_uvm     UVM Verification Publication (Auto)
```

5.5 Default Tasks

Codasip Studio default `work/tasks.py` file contains following tasks:

Model Compilation	Compile CodAL source code and check it for errors.
Assembler Disassembler Profiler Simulator C/C++ Compiler	Generate given SDK tool.
SDK Libraries	Build C/C++ libraries enabled in configuration. If the task is not present, no library is enabled.
SDK	Generate complete SDK for single model.
SDK Publication	Combine SDKs of all models in entire project hierarchy into single folder.
`<type>` Co-simulator	Generate Co-simulator of given `type`.
Random Assembler Programs	Generate random assembler programs. If seed is not set, this task is never up-to-date.
Documentation	Generate instruction set documentation.
RTL	Generate RTL .
UVM Verification (Auto/User)	Generate UVM functional verification environment of given type.
UVM Verification Publication (Auto/User)	Publish generated UVM into folder.
Component Sources	Generate empty source files for all referenced components.

5.6 Custom Tasks

Every task is created by Python function, which is called by Codasip build system. These functions start with `task_` prefix and are called task creators.

To create a custom new task, create file `tasks.py` in the root project directory. This file will overwrite the same named task creators in `work/tasks.py` and can add new task creators to add new custom tasks.

Example 6: Example task to run simulator

```
# import needed Python modules
import subprocess
import codasip
from codasip import info

# filter task-creator for top-level IA models
@codasip.filter(ia=True, top=True)
def task_run_sim(model):
    # location of simulator binary
    simulator = model.get_sdk_tool('isimulator')

    # function to be run when task is executed
    def run():
        # print information using Codasip function info.
        # will automatically format the message
        info('Running simulator for model: {}', model.design_path)
        info('Model project name: {}', model.project.name)
        info('Model sources: {}', model.dir)
        info('Model work: {}', str(model.work_dir))
        info('Model simulator path: {}', simulator)

        # run simulator binary using Python
        subprocess.check_call([simulator, '--version'])

    return {
        # ID of task. Model design path will be appended to create unique ID
        'basename': 'run_sim',
        # actions to be run for task
        'actions': [run],
        # dependencies on other tasks
        'task_dep': [model.get_task_id('sim')],
        # dependencies on files
        'file_dep': [simulator],
        # custom detection if task is up-to-date
        'uptodate': [False],
        # title of task to be shown in IDE
        'title': 'Test simulator'
    }
```

Tasks file in this example contains single task creator `task_run_sim`. This task creator creates task with `basename run_sim` for every IA top-level model in the project hierarchy. The task is dependent on the simulator generation task and the simulator binary file. If the task is executed, it will print information about the model and will run the simulator with argument `--version`.

By default, task-creators are called for every model in the project hierarchy. To filter the models use `@codasip.filter` decorator with Boolean arguments:

- `ia` - Filter task for IA (`True`) or CA (`False`) models only.
- `asip` - Filter task for ASIP (`True`) or Level (`False`) models only.
- `sdk` - Filter task for ASIP IA (`True`) tasks only, alias for `ia=True` and `asip=True`.
- `top` - Filter task for top-level models only.

To access default task file from user defined `tasks.py` file use `import tasks`. This will import default tasks file and all helper functions in it. This approach also enables to overwrite helper functions in default `tasks.py` by assigning new function to them.

Example 7: Overwrite default helper function

```
import tasks

# store original function
sdk_libs_orig = tasks.sdk_libraries

def sdk_libraries(model, print_warning=False):
    """Overwrite sdk_libraries function to disable newlib library"""
    # call original helper function
    libs = sdk_libs_orig(model, print_warning)

    if 'newlib' in libs:
        del libs['newlib']

    return libs


# overwrite the helper function
tasks.sdk_libraries = sdk_libraries
```

5.6.1 Build System Objects

The task-creator has a single argument `model`, which is set to model instance created by Codasip build system. This instance has following main attributes and methods:

- `model.name` - Name of model, e.g. `'ia'`.
- `model.design_path` - Full design path, e.g.. `'codasip_top_level.ia.codasip_urisc.ia'`.
- `model.project` - Instance of project the model belongs to.
- `model.dir` - Model source file directory.
- `model.ia` - `True` for IA model, `False` for CA model.
- `model.asip` - `True` for model describing ASIP, `False` for model describing Level.
- `model.sdk_dir` - Model SDK directory.
- `model.work_dir` - Object managing location of output files in project work directory.

- `model.get_sdk_tool('isimulator')` - Returns full path to SDK binary. This example returns path to simulator.
- `model.get_task_id('sim')` - Returns full task ID for current model and task `basename`. This example returns simulator generation task ID.
- `model.config['general.verbosity']` - Returns value from configuration. This example returns current verbosity level.

The `model.project` returns a project object describing single loaded CodAL project. This instance has the following main attributes and methods:

- `project.name` - Name of project, e.g. `'codasip_urisc'`.
- `project.dir` - Directory of project.
- `project.list_models(ia=True, asip=True)` - Returns list of model instances that are present in this project and its referenced projects. Optional filter can be used on returned models. Current example returns all IA models describing ASIP.

5.6.2 Task Description

Tasks are described by Python dictionary that are returned from task creators. Multiple tasks can be returned using Python `yield` keyword. The keys of the task dictionary are described in Doit documentation (see http://pydoit.org/).

Main task dictionary keys:

- `basename` - Base ID of the task. When task for given model / project is returned from generator, its `basename` is modified so that it is unique in the entire project hierarchy. The modified name is then used as a task ID.
  E.g. task `basename asm` for model `codasip_urisc.ia` will be modified into `asm.ia:codasip_urisc.ia`
- `actions` - List of actions to be executed when a task is executed. Every item of the list can be either name of Python function or a string used as a system command (e.g. `'touch a'`).
  - To add custom arguments to Python function, a tuple of function, arguments and keyword arguments will be used
    Example:
    `task['actions'] = [(callable, ['arg1', 12], {'arg2': 15}), callable2]`
    When task is executed, call function `callable('arg1', 12, arg2 = 15)` and then `callable2()` without any arguments

- `task_dep` - List of task IDs the task is dependent on. When any of these tasks is not up-to-date, this task is not up-to-date as well.
- `file_dep` - List of files the task is dependent on. If any of these files is not present or modified, this task is not up-to-date.
- `targets` - List of files the task is creating. Can be used by other task as dependencies.
- `uptodate` - List containing custom task status handling. If set to `[False]`, the task is never up-to-date.
- `title` - Label of task.

For all tasks in entire project hierarchy with same `basename`, a group top-level tasks is created. This task is named `<basename>.<model name>` and can be used to execute given tasks on every model in entire hierarchy. These tasks are then shown in Codasip Studio or by `list` command. When such task is executed all its sub-tasks are executed.

5.6.3 Hooks

The default tasks are usually very complex because they have to cover all configuration options and possibilities. This can be a problem when user wants to alter the default task, because the task has to be completely copied and then modified.

Therefore Codasip build system enables hooks to be added, that are called when some tasks or action is generated and can modify the task or add new ones based on them.

5.6.3.1 Task Hooks

Task hook is implemented by function with signature `hook_`*`<basename>`* in `tasks.py` file. This task hook will be called when task with given *`<basename>`* is created.

Every hook function has 2 arguments:

1. Item representing build system instance the task was created for. It can be CodAL model or project.
2. The dictionary of the task.

The hook function then returns modified task dictionary or `None` if the task should be discarded. The hook can also generate multiple dictionary tasks using multiple Python `yield` with task dictionary.

Example 8: Hook for default CodAL compilation task with basename `model`

```
def hook_compile(model, task):
    # add new action to be run when task is executed
    def new_action():
        print "New action is run"
    task['actions'].append(new_action)

    # add preprocessor define for CodAL compilation
    action = task['actions'][0]   # get first action
    action_args = action[1]       # action is tuple (callable, args, kwargs), get arguments
    # modify 1st argument = codasip.command_builder object containing arguments for codasip.compile
    action_args[0] += '-DCUSTOM_DEFINE'

    # add dependency on another task
    if 'task_dep' not in task:
        task['task_dep'] = []
    task['task_dep'] = [model.get_task_id('custom_task')]

    # return modified task
    return task
```

The task hook can also be created by decorating a function with Python decorator `@codasip.hook(basename)`. This decorator will associate the decorated function (hook) to task with given `basename`. If multiple task types should be hooked, `basename` argument can be a list.

Example 9: Modify title for assembler and disassembler tasks

```
@codasip.hook(['asm', 'dsm'])
def custom_hook(model, task):
    task['title'] += ' modified'
    return task
```

5.6.3.2 Event Hooks

There are also events in Codasip build system that can be hooked to. This enables to perform one time initialization or configuration parsing.

The hook for event is created by decorating hook function using Python decorator `@codasip.hook(event='event name')`. Multiple events can be hooked by using list of event names for event argument.

Every hook function gets action name as its first argument, so same hook function can differentiate between multiple event types. The other arguments are dependent on the event itself.

Currently supported events:

1. `tasks_imported(action, project, task_generators)`
   Called after the task files are loaded, but the task generator functions are not called.
   The `project` argument is top-level project instance, the `task_generators` is dictionary of task generator name to the generator function.
   This event can be used to perform one time loading, initialization or filtering of task generators.

2. `tasks_finished(action, project, tasks)`
   Called after all task generators are called, all task hooks are processed and the tasks are finished.
   The `project` argument is top-level project instance, the `tasks` is list of final Task instances.
   This event can be used to perform final filtering and customization of tasks that is not possible through hooks. The hooks are a preferred way of altering tasks because they are specific to some task types only and works over task dictionaries.

Example 10: Usage of `tasks_imported`event hook

```
@codasip.hook(event='tasks_imported')
def custom_config(action, project, generators):
    # load custom configuration option - bitwidth of ISA
    isa = project.config['general.isa']
    if isa not in [32, 64]:
        codasip.fatal('Invalid ISA bitwidth {}', isa)

    # store configuration into project instance that will be available in all tasks using mod-
el.project.ia
    project.isa = isa
```

Codasip build system has some differences from Doit library task dictionaries keys:

- `basename` and `name` keys are automatically handled, use `basename` only.
- `title` for task is automatically appended by model name.
- `doc` is automatically generated from `title`.

6 PROJECT CONFIGURATION

Codasip projects have uniform configuration file format based on INI file format.

6.1 Documentation Conventions

The table below defines the syntax conventions used in Codasip configuration documentation.

`Type`	Type of option. Valid types: • `bool` - boolean value, • `int` - number, • `list` - inline list [...], • `string` - string, • `path` - filesystem path - absolute or relative to project directory, • `model_path` - model path in format `project.model.project.model...` • `instance_path` - instance path in format `instance.instance...`
`Global`	If present, given option is valid for every model in entire project hierarchy. If not present, given option is valid only for project where configuration option is used.
`Multiple`	If present, given option can be used multiple times and describes list of options.
`Default value`	Default value that will be used if option is not set in configuration file.
`Values`	Possible values of option.

6.2 Configuration reference

6.2.1 version

Version of the configuration. Is used by Codasip Studio to support backward compatibility and to upgrade the configuration correctly.

6.2.2 general

General options for multiple tools and build system.

`model-version`

`Type: string`

Version of model. Is used when publishing model package.

`enable-references`

`Type: bool`

Enables/disables user specified references of project. If set to false, option `references` is not used.

`references`

`Type: list`

List of project names that are referenced within the project. When not specified, simple detection of references will occur (using regular expression).

`detect-project-type`

`Type: bool`

Enables/disables detection of project type. If set to false, option `asip` is not used.

`project-type`

`Type: string`

`Values:`

`asip`	`The current project describes an ASIP.`
`level`	`The current project describes a design level.`

Determines if current project describes ASIP or level. ASIP projects will contain tasks for SDK generation.

`ia-model`

`Type: string`

Name of model that will be used as IA model when current project is referenced from other one. If not set, first model starting with 'ia' will be used.

`ca-model`

`Type: string`

Name of model that will be used as CA (cycle accurate) model when current project is referenced from other one. If not set, first model starting with 'ca' will be used.

`optimization`

```
Type: int
Default value: 1
Global: yes
Values:
  0       No optimizations.
  1       Moderate optimizations without
          degrading compilation time
          significantly.
  2       Full optimizations.
  3       Full optimizations with aggressive
          inlining.
```

Level of optimization used for building the Codasip assembler, disassembler, simulator and co-simulator. Higher level increases build times and memory usage.

`parallel-build`

```
Type: int
Global: yes
```

Specifies number or CPUs used for a build. The default values are obtained from the system.

`verbosity`

```
Type: int
Default value: 1
Global: yes
Values:
  0        Silent.
  1        Default verbosity
  2        High verbosity
  3        Debug messages.
  4        All messages.
```

Level determines the amount of informative messages printed during tool generation.

`svg`

```
Type: bool
Default value: false
Global: yes
```

Enables generation of debugging SVG images containing internal representation of design.

`version-text`

```
Type: string
Default value:
Global: yes
```

Custom text that is printed when a tool is called with `--version`. The same text is placed in generated RTL or UVM.

`sdk-id`

```
Type: string
Default value:
Global: yes
```

Specifies custom ID that will be used as first part of SDK ID. When not set project name is used. Can be used to register multiple SDKs into Codasip IDE generated from single project.

`hdk-id`

```
Type: string
Default value:
Global: yes
```

Specifies custom ID that will be used as first part of HDK ID. When not set project name is used. Can be used to register multiple HDKs into Codasip IDE generated from single project.

`memory`

`Global: yes`

When memory is automatically instantiated, the size is taken from address bit width. When the address bit width is larger than 19 bits, the set value is used. Memory does not allow unaligned access by default. It may be overwritten.

`size`

`Type: int`

`Default value: 0x80000`

`Global: yes`

Size of default memories.

`latencies`

`Type: list`

`Default value: [[1],[1]]`

`Global: yes`

Sets the read/write latencies. Supported format: '[[R1, ...,Rn], [W1,...,Wn]].

6.2.3 codal

User arguments for CodAL compilation.

`model`

`Type: model_path`

Model path (e.g. codasip_urisc.ia) that specifies for which model arguments will be used.

`args`

`Type: string`

`Default value:`

User arguments that will be used during IA or CA CodAL models compilation. It will be used before `ia-args` and `ca-args`.

6.2.4 simulator

Options for simulator.

`debugger`

```
Type: bool
Default value: true
Global: yes
```

Enables support of C/C++ or assembly code debugging in simulator.

`codal-debugger`

```
Type: bool
Default value: false
Global: yes
```

Enables debugging inside CodAL source code. Option `debugger` has to be enabled.

`watchpoints`

```
Type: bool
Default value: false
Global: yes
```

Enables support for watchpoints during debugging. Option `debugger` has to be enabled.

`dump`

```
Type: bool
Default value: false
Global: yes
```

Enables support for resource writes dumping into VCD or Codasip format.

`stack-overflow-detection`

```
Type: bool
Default value: false
Global: yes
```

Enables detection of stack overflow during simulation.

`enable-profiler`

```
Type: bool
Global: yes
```

Enables/disables collection of profiling information during simulation. If set to false, option `profiler` is not used.

`profiler`

```
Type: string
Global: yes
Values:
  low      Enables basic profiling
           capabilities that are available
           only during debugging.
  high     Full featured profiler capabilities
           including an instruction tracking,
           different kinds of coverage, etc.
           Note that it slows down the
           simulation.
```

Specifies amount of data to be gathered during profiling.

`profiler-resources-coverage`

```
Type: bool
Default value: true
Global: yes
```

Enables resources coverage.

`profiler-call-stack`

```
Type: bool
Default value: true
Global: yes
```

Enables call stack support (has a big impact on simulation speed).

`profiler-codal-coverage`

```
Type: bool
Default value: true
Global: yes
```

Enables CodAL coverage (has a big impact on simulation speed).

`profiler-decoders-coverage`

```
Type: bool
Default value: true
Global: yes
```

Enables instruction and sequence tracking.

`profiler-ppa`

```
Type: bool
Default value: true
Global: yes
```

Enables PPA analysis.

`co-simulator`

```
Type: string
Default value: cpp
Global: yes
Values:
  cpp      Co-simulator with C/C++ interface.
  systemc  Co-simulator with SystemC TLM 2.0
           interface. An access to a SystemC
           libraries and headers is required.
  dpi      Co-simulator with DPI interface. An
           access to a DPI libraries and
           headers is required.
```

Type of Co-simulator to be generated.

`optimization`

```
Type: int
Global: yes
Values:
  0        No optimizations.
  1        Moderate optimizations without
           degrading compilation time
           significantly.
  2        Full optimizations.
  3        Full optimizations with aggressive
           inlining.
```

Level of optimization used for building the Codasip simulator and co-simulator. Overrides default optimization level.

`build-asip-tools`

```
Type: bool
Default value: false
Global: yes
```

Build also ASIPs simulator(s) and profiler(s) for extern constructs used in the project.

6.2.5 sdk

Describes content of generated SDK.

`model`

`Type: model_path`

Model path (e.g. codasip_urisc.ia) that specifies for which model options will be used.

`startup`

`Type: bool`

`Default value: false`

Enables / disables startup library containing startup assembler code to call C main function.

`newlib`

`Type: bool`

`Default value: false`

Enables / disables standard C library newlib. Provides default newlib build.

`newlib_configuration`

`Multiple: yes`

Provides custom newlib build folder name. CMake build options are passed as defines.

`name`

`Type: string`

Newlib configuration name.

`defines`

`Type: string`

Newlib configuration defines.

`compiler-rt`

`Type: bool`

`Default value: false`

Enables / disables LLVM- based runtime library for the software implementation of several operations.

`cxxabi`

`Type: bool`

`Default value: false`

Enables / disables C++ runtime library.

`man`

`Type: bool`

`Default value: false`

Determines if manual and Info documentation pages will be copied into SDK `share` directory.

`prefix`

`Type: string`

The prefix, that is used for all binaries generated in SDK, e.g. "codasip_urisc-ia", consists of two parts. The first part of the prefix is user's prefix. If not set, the project name is used, e.g. "codasip_urisc". The second part of the prefix is the model name (it cannot be changed), e.g. "ia".

`default-linker-args`

`Type: string`

`Default value:`

Default arguments that will be always used when linker is run.

6.2.6 random-assembler-programs

`use-undefined`

```
Type: bool
Default value: false
Global: yes
```

Uses instructions with undefined behavior.

`fix-complex-syntax`

```
Type: bool
Default value: false
Global: yes
```

Uses instructions with complex behavior.

`programs`

```
Type: int
Default value: 10
Global: yes
```

Number of random assembly files to be generated.

`instructions`

```
Type: int
Default value: 100
Global: yes
```

Number of instructions in generated program. Counts only PDM program instructions.

`random-seed`

```
Type: bool
Global: yes
```

Uses random seed value for generation of random programs.

`seed`

`Type: int`

`Global: yes`

If `random-seed` is false or not set, specifies numeric random seed that will be used for generation of random programs.

`override-constraints`

`Type: bool`

`Default value: false`

`Global: yes`

Do not generate constraints from model but use user contraints in .cons file only.

`use-smt-solver`

`Type: bool`

`Global: yes`

Use SMT solver mode for generation of random programs.

`use-smt-solver-timeout`

`Type: bool`

`Global: yes`

Uses SMT solver timeout for generation of random programs.

`smt-solver-timeout`

`Type: int`

`Global: yes`

If `use-smt-solver-timeout` is set, specifies value that will be used for setting of SMT solver timeout for generation of random programs.

6.2.7 compiler

Options for generation of C/C++ compiler.

`model`

`Type: model_path`

Model path (e.g. codasip_urisc.ia) that specifies for which model options will be used.

`backendgen-args`

`Type: string`

`Default value:`

C Compiler backend generator flags.

`semextr-args`

`Type: string`

`Default value:`

Semantics Extractor flags.

`architecture`

`Type: string`

Architecture name. Can be later used in C/C++ source files as preprocessor macro to detect type of C/C++ clang compiler used. If not set, "project.model" (e.g. codasip_urisc.ia) will be used. In C/C++ source file, this option can be used as #define

`default-clang-args`

`Type: string`

`Default value:`

Default arguments that will be always used when C/C++ compiler driver clang is run.

6.2.8 rtl

Options for RTL and UVM generation.

`language`

```
Type: string
Default value: verilog
Global: yes
Values:
  vhdl     Output language is VHDL.
  verilog  Output language is Verilog.
  sverilog Output language is SystemVerilog.
```

Hardware description language (HDL) that will be used as output for RTL description.

`testbenches`

```
Type: list
Default value: []
Global: yes
Values:
  auto     The testbench should be used for
           fast and easy-to-use tests whether
           programs runs on the ASIP or not.
           It instantiates the ASIP design
           with automatically connected
           memories, generates clock and
           reset, loads memory content and
           drives inputs with default values.
  jtag     The testbench should be used to
           test the on-chip debugger if
           available for the ASIP. It
           instantiates the ASIP design with
           automatically connected memories,
           generates clock and reset, loads
           memory content and drives inputs
           with default values.
  user     The testbenches should be used as a
           template for user-defined tests. It
           instantiates the design (DUT),
           generates clock and reset and
           drives inputs with default values.
           Also testbenches for internal units
           will be generated as well.
```

Generates HDL testbenches of specific type.

`comments-length`

`Type: int`

`Default value: 100`

`Global: yes`

Adds source file and line comment before every statement in generated RTL files including original CodAL code. The length limits the original CodAL code comment. Zero value means no comments.

`cnd-nesting-level`

`Type: int`

`Default value: 0`

`Global: yes`

Limit nesting of Verilog conditional operator (?) or VHDL when-else. The value defines maximum allowed number of nesting levels for the operator. Default zero value means no limit.

`opt-mux-select-max`

`Type: int`

`Default value: 8`

`Global: yes`

Limit the bit width of the selection condition for RTL multiplexer. Do not generate RTL multiplexers (case/switch) with large bit width of the selection condition.

`opt-fu-always-act`

`Type: bool`

`Default value: false`

`Global: yes`

Removes activation ports in functional units that are always activated. The reset unit has to be empty and the unit must be activated unconditionally.

`opt-fu-input-ops`

```
Type: bool
Default value: false
Global: yes
```

Optimize away operand isolation logic created for data parameters of functional units.

`opt-fu-act`

```
Type: bool
Default value: false
Global: yes
```

Optimize away operand isolation logic created for activation of simple combinational C functional units with return value only.

`opt-fu-act-aux`

```
Type: bool
Default value: true
Global: yes
```

Reduce usage of activation condition inside functional units using auxiliary signal.

`reset-level`

```
Type: string
Default value: low
Global: yes
Values:
  low      Reset is active in low (0).
  high     Reset is active in high (1).
```

Sets reset active level to be low or high.

`reset-type`

`Type: string`

`Default value: async`

`Global: yes`

`Values:`

`sync`	`A design has synchronous reset (good for FPGA designs).`
`async`	`A design has asynchronous reset (good for ASIC designs).`

Chooses synchronous or asynchronous reset for all registers and FSM in the design. The only exception is related to JTAG modules which are defined by the standard to be asynchronous.

`technology`

`Type: string`

`Default value: asic`

`Global: yes`

`Values:`

`asic`	`Generic ASIC technology.`
`xilinx-artix-7`	`Xilinx Artix7 FPGA family.`
`xilinx-virtex-5`	`Xilinx Virtex5 FPGA family.`
`xilinx-spartan-3`	`Xilinx Spartan3 FPGA family.`

Specifies target technology to be used (ASIC vs. FPGA).

sync-reg-length

```
Type: int
Default value: 2
Global: yes
```

Specifies number of flip-flops used for signal synchronization between asynchronous clock domains (meta-stability treatment). It is the number of flip-flops used in target clock domain. By default the two flip-flop synchronizer will be used.

dff-delay

```
Type: int
Default value: 0
Global: yes
```

Specify propagation delay (in picoseconds) to model Clock-to-Q delay of all flip-flop registers. The value will be hard-coded into generated RTL description. Default value of zero means no delay and no additional HDL code at all.

synthesis-tool

```
Type: list
Default value: []
Global: yes
Values:
```

`xilinx_ise`	`Xilinx ISE Design Suite`
`xilinx_vivado`	`Xilinx Vivado Design Suite`
`cadence_rtl_compiler`	`Cadence Encounter RTL Compiler`
`cadence_genus`	`Genus Synthesis Solution`

Chooses RTL synthesis tool.

`rtl-simulation-tool`

```
Type: list
Default value: [questa]
Global: yes
Values:
  questa   Mentor Graphics Questa Advanced
           Simulator
  riviera  Aldec Riviera-PRO
  rtlsim   Not supported RTL simulator (no
           startup scripts)
```

Specify foreign simulation tool used by DPI co-simulator. Also RTL testbench and verification startup scripts will be generated for this tool.

`clk-frequency`

```
Type: int
Default value: 100
Global: yes
```

Specify target frequency in MHz for global clock signal (used by synthesis and simulation tools).

`tck-frequency`

```
Type: int
Default value: 100
Global: yes
```

Specify target frequency in MHz for JTAG clock signal (used by synthesis and simulation tools).

`memory-init`

```
Type: bool
Global: yes
```

Initializes content of RTL memories using mapped programs.

`force-heap-and-stack`

`Type: bool`

`Default value: false`

`Global: yes`

Chooses whether to skip or include content of .heap and .stack sections during RTL memory initialization procedure.

`mapping`

`Global: yes`

`Multiple: yes`

Program mapping for a memory initializer. The memory initializer is able to initialize program and data memories with mapped programs.

> **`instance`**
>
> `Type: instance_path`
>
> An ASIP instance.
>
> **`program`**
>
> `Type: path`
>
> Path to a program that should be used for a memory initialization.

name-case-top

```
Type: string
Default value: none
Global: yes
Values:
  lower    Convert project identifier to lower
           case English letters.
  upper    Convert project identifier to upper
           case English letters.
  none     Do not convert identifier, use
           original project identifier.
```

Sets the case level for the project top module/entity. The project identifier will use upper/lower case English letters only or remain untouched.

name-case-top-port

```
Type: string
Default value: none
Global: yes
Values:
  lower    Convert all top port identifiers to
           lower case English letters.
  upper    Convert all top port identifiers to
           upper case English letters.
  none     Do not convert identifiers, use
           identifiers created automatically
           by the generator.
```

Sets the case level for all port identifiers of the project top module/entity. The port identifiers will use upper/lower case English letters only or remain untouched.

`name-case-module`

```
Type: string
Default value: none
Global: yes
Values:
  lower    Convert module/entity identifiers
           to lower case English letters.
  upper    Convert module/entity identifiers
           to upper case English letters.
  none     Do not convert identifiers, use
           identifiers created automatically
           by the generator.
```

Sets the case level for all project internal modules/entities. The module/entity identifiers will use upper/lower case English letters only or remain untouched.

`name-case-port`

```
Type: string
Default value: none
Global: yes
Values:
  lower    Convert all top port identifiers to
           lower case English letters.
  upper    Convert all top port identifiers to
           upper case English letters.
  none     Do not convert identifiers, use
           identifiers created automatically
           by the generator.
```

Sets the case level for all internal port identifiers inside the project sub-hierarchy. The port identifiers will use upper/lower case English letters only or remain untouched.

`name-case-signal`

```
Type: string
Default value: none
Global: yes
Values:
  lower    Convert all signal identifiers to
           lower case English letters.
  upper    Convert all signal identifiers to
           upper case English letters.
  none     Do not convert identifiers, use
           identifiers created automatically
           by the generator.
```

Sets the case level for all internal signal identifiers inside the project. The signal identifiers will use upper/lower case English letters only or remain untouched.

`name-case-instance`

```
Type: string
Default value: none
Global: yes
Values:
  lower    Convert all instance identifiers to
           lower case English letters.
  upper    Convert all instance identifiers to
           upper case English letters.
  none     Do not convert identifiers, use
           identifiers created automatically
           by the generator.
```

Sets the case level for all module/entity instance identifiers inside the project. The instance identifiers will use upper/lower case English letters only or remain untouched.

`name-prefix-port`

```
Type: string
Default value:
Global: yes
```

Sets the prefix used for all internal port identifiers.

`name-suffix-port`

```
Type: string
Default value:
Global: yes
```

Sets the suffix used for all internal port identifiers.

`name-prefix-signal`

```
Type: string
Default value:
Global: yes
```

Sets the prefix used for all internal signal identifiers.

`name-suffix-signal`

```
Type: string
Default value:
Global: yes
```

Sets the suffix used for all internal signal identifiers.

`name-prefix-instance`

```
Type: string
Default value:
Global: yes
```

Sets the prefix used for all module/entity instance identifiers inside the project.

`name-suffix-instance`

`Type: string`

`Default value:`

`Global: yes`

Sets the suffix used for all module/entity instance identifiers inside the project.

`name-prefix-module`

`Type: string`

`Default value:`

`Global: yes`

Sets the prefix used for all project internal modules/entities.

`name-suffix-module`

`Type: string`

`Default value: _t`

`Global: yes`

Sets the suffix used for all project internal modules/entities.

`name-clock`

`Type: string`

Name of the default clock input port. This identifier will be used across the whole design. The case level will be changed according to `name-case-top-port` setting.

`name-reset`

`Type: string`

Name of the default reset input port. This identifier will be used across the whole design. The case level will be changed according to `name-case-top-port` setting.

`name-reset2`

`Type: string`

Name of the second reset input port. This identifier will be used across the whole design. The second reset port will be used only when the model contains both synchronous and asynchronous reset configuration. The case level will be changed according to `name-case-top-port` setting.

`name-scan-mode`

`Type: string`

Name of the default scan mode input port. This identifier will be used across the whole design. The case level will be changed according to `name-case-top-port` setting.

6.2.9 uvm

Options for generation of UVM functional verification environment.

`auto`

`Type: bool`

`Default value: true`

`Global: yes`

Chooses mode of functional verification. Automatic mode contains valid golden model, initialization, and launch of the golden model. User mode is a pre-generated template and must be manually modified by the user.

`assertions`

`Type: bool`

`Default value: false`

`Global: yes`

Enables / disables assertion based verification support in generated UVM.

`data-ver`

`Type: bool`

`Default value: false`

`Global: yes`

Enables / disables data verification support on ASIP interfaces in generated UVM.

`code-coverage`

`Type: bool`

`Default value: false`

`Global: yes`

Enables / disables collection of code coverage during functional verification.

`func-coverage`

`Type: bool`

`Default value: false`

`Global: yes`

Enables / disables collection of functional coverage during functional verification.

`unique-files`

`Type: bool`

`Default value: false`

`Global: yes`

Generates unique names for files of the UVM to simplify compilation under different HVL simulators.

`mapping`

`Global: yes`

`Multiple: yes`

Mapping of programs to instances in design to be automatically loaded.

`instance`

`Type: instance_path`

An ASIP instance.

`program`

`Type: path`

Path to an program that should be used for a loading.

6.2.10 jtag

`nexus-clock-control`

`Type: bool`

`Default value: false`

`Global: yes`

Generate logic used as global clock buffer with clock enable for ASIP that is used during reset of the ASIP.

`dft-scan-mode`

`Type: bool`

`Default value: false`

`Global: yes`

Enable support for DFT (Design For Test) by creating global scan mode input port.

6.2.11 nexus

Options for HW debugging using JTAG/Nexus.

`model`

`Type: model_path`

Specifies associated ASIP model this configuration applies to.

`identifier`

```
Type: int
Default value: 0x10001003
Global: yes
```

Specifies identifier of the JTAG TAP (Test Access Port) controller.

`ir-length`

```
Type: int
Default value: 4
Global: yes
```

Specifies bit width of the instruction register inside JTAG TAP (Test Access Port) controller.

`instruction-idcode`

```
Type: int
Default value: 2
Global: yes
```

Specifies value of JTAG IDCODE instruction.

`instruction-nexus-access`

```
Type: int
Default value: 3
Global: yes
```

Specifies value of JTAG NEXUS_ACCESS instruction.

`instruction-nexus-control`

```
Type: int
Default value: 4
Global: yes
```

Specifies value of JTAG NEXUS_CONTROL instruction.

`interfaces`

```
Type: list
Default value: []
```

List of ASIP interfaces enhanced by Nexus client. Such interface could be used to load program or read/write memory content."

`breakpoints`

```
Type: int
Default value: 8
```

Specifies how many HW breakpoint modules to include in Nexus HW debugger.

`watchpoints`

```
Type: bool
Default value: false
```

Enables / disables support for HW watchpoints.

`bscan-resources`

```
Type: list
Default value: []
```

Specifies list of ASIP registers with enabled boundary scan-chain (BSC JTAG) support. Note that program counter is always presented and does not need to be listed.

6.2.12 fpga_bscan

Detailed specification of BSCAN cell/primitive and its connection in the FPGA.

`enable`

`Type: bool`

`Default value: false`

`Global: yes`

Use embedded Boundary Scan (BSCAN) cell for connecting internal logic to JTAG interface available on FPGA.

`jtag-chain`

`Type: int`

`Default value: 1`

`Global: yes`

Specifies JTAG_CHAIN parameter for BSCAN cells (required for `xilinx-artix-7` and `xilinx-virtex-5` only).

`user-code`

`Type: int`

`Default value: 3`

`Global: yes`

Specifies USER code for FPGA BSCAN cell.

`ir-length`

`Type: int`

`Default value: 4`

`Global: yes`

Specifies bit-width of the instruction register in BSCAN cell.

`ir-before`

`Type: int`

`Default value: 0`

`Global: yes`

Specifies sum of instruction registers length before the BSCAN cell.

dev-before

```
Type: int
Default value: 0
Global: yes
```

Specifies number of devices before the BSCAN cell.

6.2.13 documentation

Configuration for document generator.

format

```
Type: string
Default value: html
Global: yes
Values:
  html     HTML output.
  rtf      RTF output.
```

6.2.14 publication

Output paths for publication.

publish-models

```
Type: list
Default value: []
```

Specifies list of models within CodAL project that should be published.

sdk

```
Type: bool
Default value: true
```

Enables / disables SDK publication into IP package.

```
hdk
Type: bool
Default value: true
```

Enables / disables HDK publication into IP package.

```
hdk-uvm
Type: bool
Default value: false
```

Enables / disables UVM publication into IP package.

```
tests
Type: bool
Default value: true
```

Enables / disables tests publication into IP package.

```
doc
Type: bool
Default value: true
```

Enables / disables documentation publication into IP package.

```
archive
Type: bool
Default value: true
```

Enables / disables package archivation

6.2.15 extern

Describes configuration of referenced extern projects.

`id`

`Type: string`

Name of referenced project.

`configuration`

`Type: string`

Configuration of the project.

`options`

`Type: dict`

Contains options for given project.

7 FILE ORGANIZATION

A software project that creates not only programs to run but also the processor(s) to run them on and the tools to assemble and compile them with must manage many files and file types. To facilitate this task, Codasip has defined a methodology and a file organization associated with it.

The Codasip build system then uses this methodology to detect all input files and automatically creates tasks dependencies, so no additional configuration is needed and only necessary tasks are executed.

This chapter is organized as follows:

7.1 Directory Structure of CodAL Project Describing Design Level

7.1.1 Directory *`<project>`*

Each project consists of the following directories at the top level. The *`<project>`* is the project name, such as *codasip_urisc_top* or *codix_vliw_branch_predictor*.

- *`<project>`*`/doc/`
  Optional directory. Contains documentation of the project (e.g. top level description, ...).
- *`<project>`*`/model/`
  Mandatory directory. Contains the design level model (see "Directory <project>/model" on page 120).

The following table summarizes the directory hierarchy of each design level project.

Table 2: Directory hierarchy summary of ASIP project− directory <project>

Directory hierarchy summary	
Directory	**Type**
`<project>``/doc/`	Optional
`<project>``/model/`	Mandatory

7.1.2 Directory *<project>*/model

The project may contain one or more instruction-accurate models and/or one or more cycle-accurate models. The directory contains the following sub-directories:

- `<project>/model/<ca>/`
  Optional directory. Contains the cycle-accurate model. `<ca>` is the name of the cycle-accurate model, such as *ca* (default name) or *ca_3stages*.
- `<project>/model/<ia>/`
  Optional directory. Contains the instruction-accurate model. `<ia>` is the name of the instruction-accurate model, such as *ia* (default name) or *ia_functional*.
- `<project>/model/share/`
  Mandatory directory. Contains shared parts of the model across the instruction-accurate and cycle-accurate models. For example top level description is usually the same for all models (see "Directory <project>/model/share" on page 120).

The following table summarizes the directory hierarchy of the model directory.

Table 3: Directory hierarchy summary of ASIP project– directory <project>/model

Directory hierarchy summary	
Directory	**Type**
`<project>/model/<ca>/`	Optional
`<project>/model/<ia>/`	Optional
`<project>/model/share/`	Mandatory

7.1.3 Directory *<project>*/model/share

The directories contain includes, common configuration files, headers and sources forthe design level. Files at this level have no mandatory prefix, the standard set of directories and files are:

- `<project>/model/share/include/`
  Mandatory directory. Contains common includes, headers of auxiliary functions, and configuration headers.
  - `config.hcodal`
    Mandatory file. Contains configuration of interfaces, register files, address spaces, etc.
  - `utils.hcodal`
    Header file name example.

- `<project>/model/share/level/`
  Mandatory directory. Contains a design level description.
  - `level.codal`
    Mandatory file. This file may be split into more codal files for better readability.
- `<project>/model/share/other/`
  Optional directory. Contains other source files, such as sources of auxiliary functions.
  - `utils.codal`
    Source file name example.

The following table summarizes the directory hierarchy of the share directory.

Table 4: Directory hierarchy summary of ASIP project– directory <project>/model/share

Directory hierarchy summary	
Directory	**Type**
`<project>/model/share/include/`	Mandatory
`<project>/model/share/level/`	Mandatory
`<project>/model/share/other/`	Optional

7.2 Directory Structure of CodAL Project Describing ASIP

7.2.1 Directory *`<project>`*

Each project consists of the following directories at the top level. The `<project>` is the project name, such as *codasip_urisc* or *codix_vliw*.

- `<project>/doc/`
  Optional directory. Contains documentation of the project (e.g. ISA description, scheme of the ASIP, ...).
- `<project>/libs/`
  Optional directory. Contains a port of standard C library (e.g. newlib), and startup code.
- `<project>/linker/`
  Optional directory. Contains custom linker script with `.lds` extension.
- `<project>/model/`
  Mandatory directory. Contains the ASIP model (see "Directory <project>/model" on page 120)

The following table summarizes the directory hierarchy of each ASIP project.

Table 5: Directory hierarchy summary of ASIP project– directory <project>

Directory hierarchy summary	
Directory	**Type**
`<project>/doc/`	Optional
`<project>/libs/`	Optional
`<project>/linker/`	Optional
`<project>/model/`	Mandatory

7.2.2 Directory `<project>/model`

The project may contain a single instruction-accurate model and/or one or more cycle-accurate models. The directory contains the following sub-directories:

- `<project>/model/<ca>/`
  Optional directory. Contains the cycle-accurate model. `<ca>` is the name of the cycle-accurate model, such as *ca* (default name) or *ca_3stages* (see "Directory <project>/model/<ca>" on page 125).
- `<project>/model/<ia>/`
  Optional directory. Contains the instruction-accurate model. `<ia>` is the name of the instruction-accurate model, such as *ia* (default name) or *ia_functional* (see "Directory <project>/model/<ia>" on page 124).
- `<project>/model/share/`
  Mandatory directory. Contains shared parts of the model across the instruction-accurate and cycle-accurate models. For example it includes instruction set architecture (ISA) description (see "Directory <project>/model/share" on page 120).

The following table summarizes the directory hierarchy of the model directory.

Table 6: Directory hierarchy summary of ASIP project– directory <project>/model

Directory hierarchy summary	
Directory	**Type**
`<project>/model/<ca>/`	Optional
`<project>/model/<ia>/`	Optional
`<project>/model/share/`	Mandatory

7.2.3 Directory *<project>*/`model/share`

The directory contains includes, common configuration files, headers and sources for auxiliary functions, and ISA description. Files at this level have no mandatory prefix, the standard set of directories and files are:

- *<project>*`/model/share/include/`
  Mandatory directory. Contains common includes, headers of auxiliary functions, and configuration headers.
  - `config.hcodal`
    Mandatory file. Contains configuration of interfaces, register files, address spaces, etc.
  - `opcodes.hcodal`
    Mandatory file. Contains defines for operation codes of instructions including bit widths of certain parts of instructions.
  - `utils.hcodal`
    Header file name example.
- *<project>*`/model/share/isa/`
  Mandatory directory. Contains an ISA description.
  - `isa.codal`
    Mandatory file. This file may be split into more codal files for better readability. (e.g. `isa.codal`, `isa_ops.codal`, `isa_am.codal`, ...)
  - `emulations.codal`
    Optional file. Contains emulations that are needed for C compiler. This file may be split into more codal files for better readability.
  - `peepholes.codal`
    Optional file. Contains peephole patters for C compiler. This file may be split into more codal files for better readability.
- *<project>*`/model/share/other/`
  Optional directory. Contains other source files, such as sources of auxiliary functions.
  - `utils.codal`
    Source file name example.
  - `settings.codal`
    Settings file name example.
  - `version.codal`
    File holding a version of Codasip Studio that is needed.
- *<project>*`/model/share/resources/`
  Mandatory directory. Contains a description of the interfaces of the ASIP as

well as architectural resources.

- `arch.codal`
  Mandatory file. Contains architectural registers, architectural resources, address spaces, assembler configuration and schedule classes.
- `interface.codal`
  Mandatory file. Contains interfaces of ASIP (e.g. interfaces to buses/memories, and ports)
- `externs.codal`
  Optional file. Contains definitions of used **`extern`**s and their connections.

The following table summarizes the directory hierarchy of the share directory.

Table 7: Directory hierarchy summary of ASIP project– directory <project>/model/share

Directory hierarchy summary	
Directory	**Type**
<project>/model/share/include/	Mandatory
<project>/model/share/isa/	Mandatory
<project>/model/share/other/	Optional
<project>/model/share/resources/	Mandatory

7.2.4 Directory *<project>*/model/*<ia>*

The directory contains definition of resources, headers and sources of auxiliary functions, and defines, associated only with the instruction-accurate model. The *<ia>* is the name of the instruction-accurate model, such as *ia* (default name) or *ia_functional*. The name must have the prefix *ia*, so the name also denotes the type of the model. Each file, except files placed in the `compiler` directory, has *<ia>*_ as the file name prefix. The standard set of directories and files are:

- *<project>*/model/*<ia>*/compiler/
  Optional directory. Contains additional source files for compiler generator.
  - `user_semantics.sem`
    Compiler source file name example.
- *<project>*/model/*<ia>*/events/
  Mandatory directory. Must contain the definition of *main* and *reset* events, and may contain files with additional event definitions.

  - `<ia>_main_reset.codal`
    Mandatory file. Contains definition of *main* and *reset* events.

- `<project>/model/<ia>/include/`
  Optional directory. Contains instruction-accurate specific headers or define files.
  - `ia_defines.hcodal`
    Defines file name example.
  - `ia_utils.hcodal`
    Header file name example.
- `<project>/model/<ia>/other/`
  Optional directory. Contains source files for instruction-accurate auxiliary functions.
  - `ia_utils.codal`
    Source file name example.
  - `ia_settings.codal`
    Settings file name example.
- `<project>/model/<ia>/resources/`
  Mandatory directory. Contains instruction-accurate specific resources (e.g. register holding a fetched instruction from a memory).
  - `ia_resources.codal`
    Resource file name example.

The following table summarizes the directory hierarchy of the instruction-accurate model.

Table 8: Directory hierarchy summary of ASIP project− directory <project>/model/<ia>

Directory hierarchy summary	
Directory	**Type**
`<project>/model/<ia>/compiler/`	Optional
`<project>/model/<ia>/events/`	Mandatory
`<project>/model/<ia>/include/`	Optional
`<project>/model/<ia>/other/`	Optional
`<project>/model/<ia>/resources/`	Mandatory

7.2.5 Directory `<project>/model/<ca>`

The directory contains definition of resources, decoders, pipelines, headers, sources of auxiliary functions, and defines associate only with the cycle-accurate model. `<ca>` is the name of the cycle-accurate model, such as *ca* (default name) or *ca_3stages*. The name must have the prefix *ca*, so the name also denotes the type of the model. Each file, except files placed in the `compiler` directory, has `<ca>_` as the file name prefix. The standard set of directories and files are:

- `<project>/model/<ca>/compiler/`
  Optional directory. Contains additional source files for the compiler generator.
  - `CodasipMicroClasses.td`
    Source file name example.
- `<project>/model/<ca>/decoders/`
  Mandatory directory. Contains definition of decoder(s). The naming convention is `<ca>_<name>.codal` where `<name>` is a name of the decoder. If the decoder is too large (e.g. file contains more than 500 lines), it may be split into more smaller files (e.g. `<ca>_<name>.codal`, `<ca>_<name>_am.codal`, `<ca>_<name>_ops.codal`) for readability.

  If there are more than one decoder, each decoder is placed in a sub-directory called `<name>` where *<name>* is a name of the decoder. The files within this directory follows the same naming conventions as in the case of a single decoder.
  - `ca_decoder.codal,ca_decoder_am.codal,ca_decoder_ops.codal`
    Split file name example.
  - `predecoder/ca_predecoder.codal`
    `decoder/ca_decoder.codal`
    Multi-decoder directory and file name example.
- `<project>/model/<ca>/events/`
  Mandatory directory. Contains definition of *main* and *reset* events, and may contain files with additional event definitions.
  - `<ca>_main_reset.codal`
    Mandatory file. Contains definition of *main* and *reset* events.
- `<project>/model/<ca>/include/`
  Optional directory. Contains cycle-accurate specific headers or defines files.
  - `ca_defines.hcodal`
    Defines file name example.
  - `ca_utils.hcodal`
    Header file name example.
- `<project>/model/<ca>/other/`
  Optional directory. Contains source files of cycle-accurate auxiliary functions.

  - `ca_utils.codal`
    Source file name example.
  - `ca_settings.codal`
    Settings file name example.

- *<project>*`/model/`*<ca>*`/pipelines/`
  Mandatory directory for a pipelined architecture. Contains definition of pipeline (s). File naming convention is *<ca>*`_`*<pipe>*`_stage`*<order>*`_` *<stage>*`.codal`, where *<pipe>* is a name of the pipeline, *<order>* is an order of a pipeline stage starting from zero and *<stage>* is a name of the pipeline stage.

  If there are more pipelines, each pipeline is placed in a sub-directory called *<name>*. The files within this directory follows the same naming convention as in the case of one pipeline architecture.

  Each file is dedicated to the specific stage of the pipeline and contains events assigned to this stage.

  - `ca_pipe_stage0_fe.codal`
    First stage file name example.
  - `ca_pipe_stage1_id.codal`
    Second stage file name example.

- *<project>*`/model/`*<ca>*`/resources/`
  Mandatory directory. Contains cycle-accurate specific resources (e.g. pipeline registers).

  - `ca_resources.codal`
    Resource file name example. The file can be split to smaller files is necessary (e.g. split resources respecting a pipeline).

The following table summarizes the directory hierarchy of the cycle-accurate model.

Table 9: Directory hierarchy summary of ASIP project– directory <project>/model/<ca>

Directory hierarchy summary	
Directory	**Type**
<project>`/model/`*<ca>*`/compiler/`	Optional
<project>`/model/`*<ca>*`/decoders/`	Mandatory
<project>`/model/`*<ca>*`/events/`	Mandatory
<project>`/model/`*<ca>*`/include/`	Optional
<project>`/model/`*<ca>*`/other/`	Optional

`<project>/model/<ca>/pipelines/`	Mandatory
`<project>/model/<ca>/resources/`	Mandatory

8 ASSEMBLER, DISASSEMBLER AND LINKER

The Assembler is a tool that translates human-readable assembly code into a binary object file. Object files are afterward linked by the Codasip Linker to processor-readable binary code.

Basically, the Codasip Assembler is able to parse two languages at once. The first being the instruction set language described by the CodAL model (instruction set syntax). The second being the language that is used to specify assembler directives and symbols.

The generated Assembler supports the compilation of assembly programs in the form commonly used by the GNU assembler. The Codasip C/C++ Compiler, as well as the GCC and LLVM compilers, produces the outputs in this format. Thus, the Codasip Assembler is able to assemble their outputs as well.

The language used to specify directives can be changed in order to be compatible with the company’s standard or to handle some of the compiler's output. Changing the directives and symbols format is relatively straightforward.

The Disassembler is a tool that reads an executable file (created as output of an assembler or a linker) and transforms it into an original assembly code. The Codasip Disassembler was designed in such way that its output is valid input to the Codasip Assembler and can be thus assembled again.

The generated Disassembler automatically treats object file data sections as data and generates constant data definitions to the output assembly code. As for code sections, it is slightly more complex. The Disassembler tries to recognize instructions, if it succeeds, it generates a textual instruction format. Otherwise it generates constant data definition and retries if the following data can be recognized as an instruction.

The Linker is a tool that links together multiple object files and resolves address relocations that were unknown during the assembly time. The Codasip Linker is independent of the target architecture, so it is not necessary to generate it.

This chapter is organized as follows:

8.1 Assembler arguments

When used from command line the Assembler command, `asm.ia`, has a number of options (see "Tool arguments reference" on page 195). However, when used from Codasip Studio or Codasip Codespace, only few options are visible and they are specified through the assembler settings of the **C/C++ Project**. Assembler is then automatically used when project is build.

8.2 Assembler Language Syntax

The input file is processed line-by-line. The format of the input file follows.

```
Program
  : Program Directive "\n"
  | Program Label
  | Program MacroCall "\n"
  | Program SymbolDefinition "\n"
  | Program "\n"
  | Program error "\n"
  | %empty
```

Some C compilers generate multiple instructions on one line. This is not allowed by default.

```
  | Program Instruction "\n"
```

However, the user can specify newline delimiter in the CodAL model using the **`new_line_delimiter`** definition of **`assembler settings`** section. Then the following rule is applicable as well.

```
  | Program Instruction "user specified new_line_delimiter"
```

8.2.1 Identifiers

Syntax of the identifier follows.

```
Identifier
  : "identifier"
  | "."
```

It can be formed with literals, decimal digits, dots, underscores and dollar characters ($), a digit cannot be used as a first character .

```
[$A-Za-z._]([$0-9A-Za-Z._])+
```

By default, the prefix $ is used for the symbols generated by the C-compiler. If there is a **`symbol_prefix`** specified in the CodAL **`assembler settings`** section, then also strings that begin with the specified prefix are taken as valid identifiers. The ssembler removes the symbol prefix from the symbol name when generating an object file. Further details are described in the *CodAL Language Reference Manual* , chapter "ASIP Description", section "Assembler Settings".

8.2.2 Integer constants

Integer constant notation is the same as in the GNU assembler.

Syntax of constant value follows. `IntegerConstantValue` is value of token `IntegerConstant`. `IntegerConstant` is used for example when represented number has to be remembered as part of an expression.

```
IntegerConstantValue
  : IntegerConstant
```

It is possible to use binary, octal, decimal and hexadecimal constant notations.

```
IntegerConstant
  : BinaryConstant
  | OctalConstant
  | DecimalConstant
  | HexadecimalConstant
```

Binary constant has prefix '0b'.

```
BinaryConstant
  : "binary constant"
```

Octal constant is prefixed with '0'.

```
OctalConstant
  : "octal constant"
```

Decimal constant does not have any prefix.

```
DecimalConstant
  : "decimal constant"
```

Similarly to C, the hexadecimal constant has prefix '0x'.

```
HexadecimalConstant
  : "hexadecimal constant"
```

Floating-point constants are also allowed.

```
DoubleConstant
  : MaybeMinus "floating-point constant"
```

```
MaybeMinus
  : %empty
  | "-"
```

8.2.3 Expressions

Expressions are inspired by the C language. Parentheses can be used as is common.

```
Expression
  : ExpressionAll
```

```
ExpressionAll
  : ExpressionAll "|" ExpressionAll
  | ExpressionAll "^" ExpressionAll
  | ExpressionAll "&" ExpressionAll
  | ExpressionAll "<<" ExpressionAll
  | ExpressionAll ">>" ExpressionAll
  | ExpressionAll "+" ExpressionAll
  | ExpressionAll "-" ExpressionAll
  | ExpressionAll "*" ExpressionAll
  | ExpressionAll "/" ExpressionAll
  | ExpressionAll "%" ExpressionAll
  | "!" ExpressionAll
  | "~" ExpressionAll
  | "+" ExpressionAll
  | "-" ExpressionAll
  | ExpressionCore
```

```
ExpressionCore
  : IntegerConstant
  | Identifier
  | "(" ExpressionAll ")"
```

Supported operators follows, they are sorted by priority.

```
+, - , ~, ! (unary),
*, /, %,
+, -,
 <<, >>,
&,
^,
|.
```

Furthermore, expression computable during compile time has to be introduced. Syntax of such expression follows. `CompileExpressionValue` is value of token

`CompileExpression.CompileExpression` is used for example when represented expression has to be remembered as a part of bigger expression and not immediately computed.

```
CompileExpressionValue
  : CompileExpression
```

```
CompileExpression
  : Expression
```

8.2.4 Comments

Comments are the same as in the C language. Line comments start with `//`. Block comments start with `/*` and end with `*/`. The user may also specify a character that starts a one-line comment using **`comment_prefix`** option of CodAL Language Reference Manual **`assembler section`**. Example of line comment and block comment follows.

Example 11: Assembly Language comments

```
// line comment
/* block comment */
```

8.2.5 Symbols

Symbols are defined as labels or using directives. They can be:

- absolute - constant
- relative (labels) - while being processed by the assembler, their value equals the distance from the beginning of the section
- undefined - either they were not defined yet within the current compilation or their definition is contained in another file

Relative symbols are constrained with the same rules as are used for C language pointers: these operations are allowed with given result:

```
abs + rel -> rel
rel + abs -> rel
rel - rel -> abs
```

Expressions work the same as in the C language. These operators are supported:

```
 !, ~, *, /, %, +, -, <<, >>, &, ^,  |
```

They have the same semantics, priority, and associativity as in the C language.

Expressions that cannot be evaluated during the first pass through the source file are substituted with special symbols (so called pseudo-symbols). The following example will provide some illustration:

Example 12: Pseud-symbols

```
MOV AX, const + 10
.equiv const, 5
```

When compiling the `MOV` instruction, the value of the `const` constant is not yet known. Despite this, we still must maintain information that the second operand of this instruction has value of const + 10. We create a pseudo-symbol whose value is the result of the given expression. The created pseudo-symbol will be evaluated during the second pass through the source file and the computed value will be used as the second operand for the `MOV` instruction. Names of pseudo-symbols are composed of the prefix "@PS" followed by the line number.

Identifiers may begin with the dollar ($) character and then continue with a literal, and then with literals, digits, underscores or dots as is specified in this regular expression

```
$?[A-Za-z._]([A-Za-z._]|[0-9A-Za-Z._])+
```

8.2.6 Directives

All assembler directives have names that begin with a period ('.'). The names are case insensitive.

Supported directives are further discussed in section "Assembler, Disassembler and Linker" on page 129. Unsupported GNU directives are contained in section "Unsupported GNU Assembler Directives" on page 164. Ignored GNU directives are shown in section "Ignored GNU Assembler Directives" on page 165 and unknown GNU directives are mentioned in section "GNU Assembler Directives Unknown to Assembler" on page 166.

8.2.7 Labels

Syntax of the labels follows. The first rule stands for a label, the second for a local label.

```
Label
  : Identifier ":"
  | DecimalConstant ":"
```

Label identifies a place in the assembler code. After the program is tranlated into a binary code, all the references on the labels are translated to addresses as well or encoded as relocations.

Local labels can be used in the code similar as normal labels.

Local labels can be referenced as instruction operands in the form [0-9]+(b|f). When the label has suffix 'b', then it's location is searched backwards (lower or equal addresses), if 'f' then the corresponding label is searched forwards (higher addresses).

8.2.8 Macros

Codasip assembler also supports simple macro definitions and subsequent macro calls. Syntax of the macro call follows.

```
MacroCall
  : "macro call" MaybeMacroCallArguments
```

```
MacroCallArguments
  : MacroCallArgument
  | MacroCallArguments "," MacroCallArgument
```

```
MacroCallArgument
  : Identifier
  | "(" Identifier ")"
```

Terminal "`macro call`" refers to the name of the macro definition. Example of macro usage follows.

Example 13: Macro call

```
.macro three_nops
nop
nop
nop
.endm

.text
...
three_nops
...
```

Firstly, macro is defined by using **`.macro`** and **`.endm`** directives. Then call to the macro can be used. Call of the macro `three_nops` puts three `nop` instructions into `.text` section.

8.2.9 Symbol definition

Symbols can be defined also by the following syntax just as by using directives for symbol definition, such an `.equiv` directive.

```
SymbolDefinition
  : Identifier SymbolDefinitionOperator Expression
```

```
SymbolDefinitionOperator
  : "="
```

Symbol is defined and immediately granted value represented by the expression. Note that symbols should not have recursive definition.

Example 14: Symbol definition

```
.text
// both following lines have the same semantics
.equiv a, 1
a = 1
```

8.3 Supported Assembler Directives

Most of the directives listed in this section are inspired by the GNU assembler, they are fully supported, and their syntax and semantics are exactly the same as in the GNU assembler, version 2.25. Whole description of the directives in this section is taken from http://sourceware.org/binutils/docs/as.

List of Codasip directives follows.

- .abs_org
- .address_space
- .bit
- .bundle_align_end
- .codasip_retstruct_reg
- .codasip_retval_regs
- .debug_off
- .debug_on
- .profiler
- .section_adjustable

List of all supported assembler directives follows.

```
Directive
  : DirectiveAbsOrg
  | DirectiveAddressSpace
  | DirectiveAlignGroup
  | DirectiveAltmacro
  | DirectiveAltmacroLocal
  | DirectiveAsciiGroup
  | DirectiveBit
  | DirectiveBundleAlignEnd
  | DirectiveByteGroup
  | DirectiveCfiStartproc
  | DirectiveCfiEndproc
  | DirectiveCfiDefCfa
  | DirectiveCfiDefCfaRegister
  | DirectiveCfiDefCfaOffset
  | DirectiveCfiAdjustCfaOffset
  | DirectiveCfiOffset
  | DirectiveCfiValOffset
  | DirectiveCfiRegister
  | DirectiveCfiSameValue
  | DirectiveCfiReturnColumn
  | DirectiveCodasipRetstructReg
  | DirectiveCodasipRetvalRegs
  | DirectiveCommGroup
  | DirectiveDataGroup
  | DirectiveDebugOff
  | DirectiveDebugOn
  | DirectiveDoubleGroup
  | DirectiveEndm
  | DirectiveEquGroup
  | DirectiveEquivGroup
  | DirectiveErr
  | DirectiveError
  | DirectiveFile
  | DirectiveFill
  | DirectiveGlobalGroup
  | DirectiveIdent
  | DirectiveIncbin
  | DirectiveLine
  | DirectiveList
  | DirectiveLoc
  | DirectiveLocal
  | DirectiveMacro
  | DirectiveNoaltmacro
  | DirectiveNolist
  | DirectiveOrg
  | DirectivePopsection
  | DirectivePrevious
  | DirectiveProfiler
  | DirectivePushsection
  | DirectiveSection
  | DirectiveSectionAdjustable
  | DirectiveSize
  | DirectiveSkipGroup
  | DirectiveSleb128Group
  | DirectiveType
  | DirectiveWarning
  | DirectiveWeak
```

8.3.1 .2byte

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

The directive `.2byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next 2 bytes.

8.3.2 .4byte

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

The directive `.4byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next 4 bytes.

8.3.3 .abs_org

```
DirectiveAbsOrg
  : ".abs_org" CompileExpression
```

The directive slaces section that contains it to the address specified by the expression. It should not be used more than once per section and sections should not overlap with each other.

Example 15: Directive .abs_org

```
.text
.abs_org 0x40000 // moves section .text to the address 0x40000
```

8.3.4 .address_space

```
DirectiveAddressSpace
  : ".address_space" CompileExpressionValue
```

The directive assigns address space index specified by the expression to the section that contains it.

Example 16: Directive .address_space

```
.text
.address_space 1 // assign address space with index 1 to the section .text
```

8.3.5 .align

```
DirectiveAlignGroup
  : DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue "," Com-
pileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue
```

```
DirectiveAlignGroupType
  : ".align"
  | ".balign"
  | ".balignw"
  | ".balignl"
  | ".p2align"
  | ".p2alignw"
  | ".p2alignl"
```

The directive pads the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It may be omitted (and the comma as well). If it is omitted, the padding bytes are by default zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system for GNU assembler. The Codasip Assembler is similar to GNU as it also provides **`.balign`** and **`.p2align`** directives, described later, which have a consistent behaviour across all architectures.

For the Codasip® Assembler, **`.align`** is by default **`.balign`**, its semantics can be changed in the CodAL **`assembler settings`** section.

8.3.6 .altmacro

```
DirectiveAltmacro
  : ".altmacro"
```

Enables alternate macro mode.

Supported feature of alternate macro mode is **`LOCAL`** directive which syntax follows.

```
DirectiveAltmacroLocal
  : "LOCAL" Identifier
```

It enables to define symbols in macros without fear of symbol name collisions.

Example 17: Directive .altmacro

```
.altmacro
.macro jump_over
LOCAL a
jump a
...
a:
.endm

.text
...
jump_over
...
jump_over
...

// generated following code internally
.text
...
jump jump_over_a_0
...
jump_over_a_0:
...
jump jump_over_a_1
...
jump_over_a_1:
...
```

8.3.7 .ascii (8-bit characters)

```
DirectiveAsciiGroup
  : DirectiveAsciiGroupType StringList
```

```
DirectiveAsciiGroupType
  : ".ascii"
  | ".asciz"
```

The directive **`.ascii`** expects zero or more string literals separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

8.3.8 .asciiz (8-bit characters)

```
DirectiveAsciiGroup
  : DirectiveAsciiGroupType StringList
```

```
DirectiveAsciiGroupType
  : ".ascii"
  | ".asciz"
```

The directive `.asciz` is just like `.ascii`, but each string is followed by a zero byte. The character “z” in `.asciz` stands for “zero”.

8.3.9 .balign, .balignw, .balignl

```
DirectiveAlignGroup
  : DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue "," Com-
pileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue
```

```
DirectiveAlignGroupType
  : ".align"
  | ".balign"
  | ".balignw"
  | ".balignl"
  | ".p2align"
  | ".p2alignw"
  | ".p2alignl"
```

The directive pad sthe location counter (in the current subsection) to a particular storage boundary. The first expression is the alignment request in bytes. For example `.balign 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are by default zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two-byte word value. The `.balignl` directives treats the fill pattern as a four-byte longword value. For example, `.balignw`

`4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

8.3.10 .bit

```
DirectiveBit
  : ".bit" CompileExpressionValue "," ExpressionList
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

Defines initialized data of bit-width n. Unsigned numbers are encoded by direct encoding, signed numbers by two's complement.

- bitwidth – Data bitwidth, must be a multiple of current section's byte bitwidth
- values – List of expressions separated by comma

8.3.11 .bbs

```
DirectiveDataGroup
  : DirectiveDataGroupType MaybeDirectiveDataGroupFlags
```

```
DirectiveDataGroupType
  : ".data"
  | ".text"
  | ".bss"
```

```
MaybeDirectiveDataGroupFlags
  : "," String
  | %empty
```

The directive `.bss` tells the assembler to assemble the following statements onto the end of the uninitialized data section.

8.3.12 .bundle_align_end

```
DirectiveBundleAlignEnd
  : ".bundle_align_end"
```

Propagates flag to the bundling function that recently translated instruction has to be aligned. User than could by reading this flag align instruction properly.

Example 18: Directive .bundle_align_end

```
.text
jump aligned_address
slot0
slot1
slot2
slot3_compressable // this will be compressed by bundling function, current address is unaligned now
.bundle_align_end
slot0
slot1
slot2
slot3
aligned_address: // current address is aligned on bundle, because zeroes could be filled by bundling
function after second bundle
...
```

8.3.13 .byte

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

The directive `.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

8.3.14 .cfi_adjust_cfa_offset

```
DirectiveCfiAdjustCfaOffset
  : ".cfi_adjust_cfa_offset" CompileExpressionValue
```

Same as the directive `.cfi_def_cfa_offset` but offset is a relative value that is added/substracted from the previous offset.

8.3.15 .cfi_def_cfa

```
DirectiveCfiDefCfa
  : ".cfi_def_cfa" CompileExpressionValue "," CompileExpressionValue
```

The directive `.cfi_def_cfa` defines a rule for computing CFA asssemler; it take address from register and add offset to it.

8.3.16 .cfi_def_cfa_offset

```
DirectiveCfiDefCfaOffset
  : ".cfi_def_cfa_offset" CompileExpressionValue
```

The directive `.cfi_def_cfa_offset` modifies a rule for computing CFA. The register remains the same, but offset is new. Note that it is the absolute offset that will be added to a defined register to compute CFA address.

8.3.17 .cfi_def_cfa register

```
DirectiveCfiDefCfaRegister
  : ".cfi_def_cfa_register" CompileExpressionValue
```

The directive `.cfi_def_cfa_register` modifies a rule for computing CFA. From now on the new register will be used instead of the old one. Offset remains the same.

8.3.18 .cfi_endproc

```
DirectiveCfiEndproc
  : ".cfi_endproc"
```

The directive `.cfi_endproc` is used at the end of a function where it closes its unwind entry previously opened by `.cfi_startproc`, and emits it to `.debug_frame`.

8.3.19 .cfi_offset

```
DirectiveCfiOffset
  : ".cfi_offset" CompileExpressionValue "," CompileExpressionValue
```

Previous value of register is saved at offset from CFA.

8.3.20 .cfi_register

```
DirectiveCfiRegister
  : ".cfi_register" CompileExpressionValue "," CompileExpressionValue
```

Previous value of first expression is saved in second expression.

8.3.21 .cfi_return_column

```
DirectiveCfiReturnColumn
  : ".cfi_return_column" CompileExpressionValue
```

Changes return column register, i.e. the return address is either directly in register or can be accessed by rules for register.

8.3.22 .cfi_same_value

```
DirectiveCfiSameValue
  : ".cfi_same_value" CompileExpressionValue
```

Current value of register is the same like in the previous frame, i.e. no restoration needed.

8.3.23 .cfi_startproc

```
DirectiveCfiStartproc
  : ".cfi_startproc"
```

The directive **`.cfi_startproc`** is used at the beginning of each function that should have an entry in `.debug_frame`. It initializes some internal data structures. Do not forget to close the function by directive **`.cfi_endproc`**.

8.3.24 .cfi_val_offset

```
DirectiveCfiValOffset
  : ".cfi_val_offset" CompileExpressionValue "," CompileExpressionValue
```

Previous value of register is CFA + offset.

8.3.25 .codasip_retstruct_reg

```
DirectiveCodasipRetstructReg
  : ".codasip_retstruct_reg" CompileExpressionValue
```

Stores the index of the register with address of the structure that contains return value of a function to object file as CIE information.

8.3.26 .codasip_retval_regs

```
DirectiveCodasipRetvalRegs
  : ".codasip_retval_regs" CompileExpressionList
  | ".codasip_retval_regs"
```

```
CompileExpressionList
  : CompileExpression
  | CompileExpressionList "," CompileExpression
```

Stores indexes of the registers that contains return value of a function to object file as CIE information.

8.3.27 .comm

```
DirectiveCommGroup
  : DirectiveCommGroupType Identifier "," CompileExpressionValue MaybeDirectiveCommAlignment
```

```
DirectiveCommGroupType
  : ".comm"
  | ".lcomm"
```

```
MaybeDirectiveCommAlignment
  : "," CompileExpressionValue
  | %empty
```

The directive `.comm` declares a common symbol named symbol. Common symbols are usually used for global variables. Codasip® Linker does not currently merge common symbols, but this feature will be supported soon.

8.3.28 .data

```
DirectiveDataGroup
  : DirectiveDataGroupType MaybeDirectiveDataGroupFlags
```

```
DirectiveDataGroupType
  : ".data"
  | ".text"
  | ".bss"
```

```
MaybeDirectiveDataGroupFlags
  : "," String
  | %empty
```

The directive `.data` tells assembler to assemble the following statements onto the end of the initialized data section.

8.3.29 .debug_off

```
DirectiveDebugOff
  : ".debug_off"
```

The directive `.debug_off` disables debugging of assembler parser.

8.3.30 .debug_on

```
DirectiveDebugOn
  : ".debug_on"
```

The directive `.debug_on` enables debugging of assembler parser.

It can be used to find why is an instruction incorrectly translated and to find collisions in assembly grammar.

8.3.31 .double

```
DirectiveDoubleGroup
  : DirectiveDoubleGroupType CompileDoubleExpressionList
```

```
DirectiveDoubleGroupType
  : ".double"
  | ".float"
  | ".single"
```

```
CompileDoubleExpressionList
  : %empty
  | DoubleConstant
  | CompileExpression
  | CompileDoubleExpressionList "," DoubleConstant
  | CompileDoubleExpressionList "," CompileExpression
```

The directive `.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. The double constant is stored according to the IEEE 754-2008 as a binary64 value.

8.3.32 .dword

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

This directive expects zero or more expressions, of any section, separated by commas.

The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

8.3.33 .endm

```
DirectiveEndm
  : ".endm"
```

The directive `.endm` closes macro definition started by the directive `.macro`.

8.3.34 .equ

```
DirectiveEquGroup
  : DirectiveEquGroupType Identifier "," Expression
```

```
DirectiveEquGroupType
  : ".equ"
  | ".set"
```

The directive `.equ` sets the value of symbol to expression. It may redefine symbol value. It is synonymous with the directive `.set`.

8.3.35 .equiv

```
DirectiveEquivGroup
  : DirectiveEquivGroupType Identifier "," Expression
```

```
DirectiveEquivGroupType
  : ".equiv"
  | ".eqv"
```

The `.equiv` directive is like directives `.equ` and `.set`, except that the assembler will signal an error if symbol is already defined. It is synonymous with thw directive `.eqv`.

8.3.36 .eqv

```
DirectiveEquivGroup
  : DirectiveEquivGroupType Identifier "," Expression
```

```
DirectiveEquivGroupType
  : ".equiv"
  | ".eqv"
```

The `.eqv` directive is like directives `.equ` and `.set`, except that the assembler will signal an error if symbol is already defined. It is synonymous with the directive `.equiv`.

8.3.37 .err

```
DirectiveErr
  : ".err"
```

If assembler assembles a `.err` directive, it will print an error message and, it will not generate an object file. This can be used to signal an error in conditionally compiled code.

8.3.38 .error

```
DirectiveError
  : ".error"
  | ".error" String
```

Similarly to the directive `.err`, this directive emits an error, but you can specify a string that will be emitted as the error message. If you do not specify the message it prints the default error message.

8.3.39 .file

```
DirectiveFile
  : ".file" IntegerConstantValue String
  | ".file" String
```

When emitting DWARF2 line number information, `.file` assigns filenames to the `.debug_line` file name table.

The integer constant should be a unique positive integer to use as the index of the entry in the table. The filename operand is a string literal.

The detail of filename indices is exposed to the user because the filename table is shared with the `.debug_info` section of the DWARF2 debugging information, and thus the user must know the exact indices that table entries will have.

Second variant of `.file` directive is ignored.

8.3.40 .fill

```
DirectiveFill
  : ".fill" CompileExpressionValue
  | ".fill" CompileExpressionValue "," CompileExpressionValue
  | ".fill" CompileExpressionValue "," CompileExpressionValue "," CompileExpressionValue
```

Operands are repeat, size and value by order. This emits repeat copies of size bytes. Repeat may be zero or more. Size may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the byte-order of an integer on the computer as is assembling for. Each size bytes in a repetition is taken from the lowest order size bytes of this number. Again, this bizarre behaviour is compatible with other people's assemblers.

Size and value are optional. If the second comma and value are absent, value is assumed zero. If the first comma and following tokens are absent, size is assumed to be 1.

8.3.41 .float

```
DirectiveDoubleGroup
  : DirectiveDoubleGroupType CompileDoubleExpressionList
```

```
DirectiveDoubleGroupType
  : ".double"
  | ".float"
  | ".single"
```

```
CompileDoubleExpressionList
  : %empty
  | DoubleConstant
  | CompileExpression
  | CompileDoubleExpressionList "," DoubleConstant
  | CompileDoubleExpressionList "," CompileExpression
```

The directive `.float` assembles zero or more flonums, separated by commas. It has the same effect as directive `.single`. The float constant is stored according to the IEEE 754-2008 as a binary32 value.

8.3.42 .global, .globl

```
DirectiveGlobalGroup
  : DirectiveGlobalGroupType DirectiveGlobalGroupSymbols
```

```
DirectiveGlobalGroupType
  : ".global"
  | ".globl"
```

```
DirectiveGlobalGroupSymbols
  : DirectiveGlobalGroupSymbol
  | DirectiveGlobalGroupSymbols "," DirectiveGlobalGroupSymbol
```

```
DirectiveGlobalGroupSymbol
  : Identifier
```

The directive `.global` makes the specified symbols visible to ld. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings, `.global` and `.globl`, are accepted, for compatibility with other assemblers.

8.3.43 .half (16-bit)

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

This directive is a synonym for directive `.hword`.

8.3.44 .hword (16-bit)

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

The directive expects zero or more expressions, and emits a 16-bit number for each.

This directive is a synonym for directive `.short`. Depending on target architecture, it may also be a synonym for the directive `.word`.

8.3.45 .ident

```
DirectiveIdent
  : ".ident" String
```

This directive is used to place tags in object files.

8.3.46 .incbin

```
DirectiveIncbin
  : ".incbin" String
  | ".incbin" String "," CompileExpressionValue
  | ".incbin" String "," CompileExpressionValue "," CompileExpressionValue
```

The **`.incbin`** directive includes file verbatim at the current location. Quotation marks are required around file.

Arguments are file skip and count by order. The skip argument skips a number of bytes from the start of the file. The count argument indicates the maximum number of bytes to read. Note that the data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the **`.incbin`** directive.

8.3.47 .int (32-bit)

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

The directive expects zero or more expressions, of any section, separated by commas. For each expression, it emits a number that, at run time, is the value of the expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

8.3.48 .lcomm

```
DirectiveCommGroup
  : DirectiveCommGroupType Identifier "," CompileExpressionValue MaybeDirectiveCommAlignment
```

```
DirectiveCommGroupType
  : ".comm"
  | ".lcomm"
```

```
MaybeDirectiveCommAlignment
  : "," CompileExpressionValue
  | %empty
```

The directive reserves expression bytes for a local common symbol denoted by identifier. The section and value of the identifier are those of the new local common symbol. The symbol is not declared globally, see the directive `.global`, so is normally not visible to ld.

8.3.49 .line

```
DirectiveLine
  : ".line" CompileExpressionValue
```

The directive changes the logical line number. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number line-number − 1.

8.3.50 .list

```
DirectiveList
  : ".list"
```

The directive increments listing counter. Assembler prints listing in a form address, source file line and binary coding for each processed instruction, if listing counter is greater than zero. Listing counter starts as zero.

8.3.51 .loc

```
DirectiveLoc
  : ".loc" IntegerConstantValue IntegerConstantValue MaybeDirectiveLocColumn May-
beDirectiveLocOptions
```

```
MaybeDirectiveLocColumn
  : %empty
  | IntegerConstantValue
```

```
MaybeDirectiveLocOptions
  : Identifier MaybeDirectiveLocOptions
  | Identifier IntegerConstantValue MaybeDirectiveLocOptions
  | %empty
```

When emitting DWARF2 line number information, the `.loc` directive will add a row to the `.debug_line` line number matrix corresponding to the immediately following assembly instruction. The arguments in the order "fileno, lineno, and optional column arguments" will be applied to the `.debug_line` state machine before the row is added.

The options are a sequence of the following tokens in any order:

```
basic_block
```

This option will set the `basic_block` register in the `.debug_line` state machine to true.

```
prologue_end
```

This option will set the `prologue_end` register in the `.debug_line` state machine to true.

```
epilogue_begin
```

This option will set the `epilogue_begin` register in the `.debug_line` state machine to true.

```
is_stmt value
```

This option is not supported.

```
isa value
```

This option is not supported.

```
discriminator value
```

This directive will set the discriminator register in the `.debug_line` state machine to value, which must be an unsigned integer.

8.3.52 .local

```
DirectiveLocal
  : ".local" Identifier
```

This directive, which is available for ELF targets, marks symbol as a local symbol so that it will not be externally visible. If the symbol does not already exist, it will be created.

For targets where the **`.lcomm`** directive does not accept an alignment argument, which is the case for most ELF targets, the **`.local`** directive can be used in combination with **`.comm`** to define aligned local common data.

8.3.53 .long (32-bit)

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

Directive `.long` is the same as directive `.int`.

8.3.54 .macro

```
DirectiveMacro
  : ".macro" Identifier DirectiveMacroArguments
```

```
DirectiveMacroArguments
  : %empty
  | DirectiveMacroArguments Identifier
```

The directive `.macro` starts macro definition.

8.3.55 .noaltmacro

```
DirectiveNoaltmacro
  : ".noaltmacro"
```

The directive disables alternate macro mode. See directive `.altmacro` for more information.

8.3.56 .nolist

```
DirectiveNolist
  : ".nolist"
```

The directive decrements listing counter. Assembler prints listing in the form "address, source file line and binary coding" for each processed instruction, if listing counter is greater than zero. Listing counter starts as zero.

8.3.57 .org

```
DirectiveOrg
  : ".org" CompileExpression MaybeFill
```

```
MaybeFill
  : "," CompileExpressionValue
  | %empty
```

The directive advances the location counter of the current section to a new address specified by the expression. The new address is either an absolute expression or an expression with the same section as the current subsection. You cannot use directive `.org` to cross sections.

If fill is specified, then the byte value is used to fill the space between the current end of the section and the requested address relative to the beginning of the section. Default fill value is 0.

8.3.58 .p2align

```
DirectiveAlignGroup
  : DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue "," Com-
pileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue
```

```
DirectiveAlignGroupType
  : ".align"
  | ".balign"
  | ".balignw"
  | ".balignl"
  | ".p2align"
  | ".p2alignw"
  | ".p2alignl"
```

The directive pads the location counter (in the current subsection) to a particular storage boundary. The first expression is the number of low-order zero bits the location counter must have after advancement. For example `.p2align 3` advances the location counter until itis a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are by default zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example,

`.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

8.3.59 .popsection

```
DirectivePopsection
  : ".popsection"
```

This directive replaces the current section with the top section on the section stack. This section is popped off the stack.

8.3.60 .previous

```
DirectivePrevious
  : ".previous"
```

This directive swaps the current section with most recently referenced section prior to this one. Multiple **`.previous`** directives in a row will flip between two sections.

8.3.61 .profiler

```
DirectiveProfiler
  : ".profiler" String
  | ".profiler" String "," String
```

Stores profiler information into an object file with relocation. Arguments are flags and optionally block name. Supported flags are `"s"`, start of profiling block, and `"x"`, end of profiling block. Default block name is `""`.

Example 19: Directive .profiler

```
.text
...
.profiler "s", "Block A"
...
.profiler "x", "Block A"
...
```

8.3.62 .pushsection

```
DirectivePushsection
  : ".pushsection" SectionName SectionFlags
```

```
SectionName
  : Identifier
  | String
  | ".data"
  | ".text"
  | ".bss"
  | Identifier "-" Identifier
```

```
SectionFlags
  : %empty
  | "," String SectionType
```

```
SectionType
  : %empty
  | "," SectionTypeSpecifier SectionFlagSpecificArguments
```

```
SectionTypeSpecifier
  : Identifier
```

```
SectionFlagSpecificArguments
  : %empty
  | "," IntegerConstantValue
  | "," IdOrString "," Identifier
  | "," CompileExpressionValue "," IdOrString "," Identifier
```

```
IdOrString
  : String
  | Identifier
```

This is one of the ELF section stack manipulation directives. The others are `.section`, `.subsection` , `.popsection`, and `.previous`.

This directive pushes the current section onto the top of the section stack, and then replaces the current section with name. The optional flags, type and arguments are treated the same as in the `.section` directive.

8.3.63 .quad

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

The directive `.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum. The term “quad” comes from contexts in which a “word” is two bytes; hence quad-word for 8 bytes.

8.3.64 .section

```
DirectiveSection
  : DirectiveSectionKeyword SectionName SectionFlags
```

```
DirectiveSectionKeyword
  : ".section"
```

```
SectionName
  : Identifier
  | String
  | ".data"
  | ".text"
  | ".bss"
  | Identifier "-" Identifier
```

```
SectionFlags
  : %empty
  | "," String SectionType
```

```
SectionType
  : %empty
  | "," SectionTypeSpecifier SectionFlagSpecificArguments
```

```
SectionTypeSpecifier
  : Identifier
```

```
SectionFlagSpecificArguments
  : %empty
  | "," IntegerConstantValue
  | "," IdOrString "," Identifier
  | "," CompileExpressionValue "," IdOrString "," Identifier
```

```
IdOrString
  : String
  | Identifier
```

The directive tells as the assemble the following statements onto the end of the named section.

Currently, these section flags are supported: “`x`” - text (code), “`d`” - initialized read/write data, “`b`” - uninitialized read/write data.

8.3.65 .section_adjustable

```
DirectiveSectionAdjustable
  : DirectiveSectionAdjustableKeyword SectionName "," String "," CompileExpressionValue "," Com-
pileExpressionValue "," DirectiveSectionAdjustableEndianess
```

```
DirectiveSectionAdjustableKeyword
  : ".section_adjustable"
```

```
DirectiveSectionAdjustableEndianess
  : ".little"
  | ".big"
```

The directive ends the creation of a previous section (if applicable), and creates a new section with required properties. In case there was a section with the same name already defined, this section will be reopened. This directive can be used, when special memory organization is needed. It is more convenient to use `.text`. `.data`, `.bss` and `.section` directives that use memory organization read from the memory specification in the CodAL model.

Arguments in this order have the following meaning:

- `name`– Identifier that specifies section name, if a section with the same name was already defined, this directive sets as actual section the already existing section
- `flags`– String that specifies section type flags, currently supported flags are : `"x"` - text (code), `"d"` - initialized read/write data, `"b"` - uninitialized read/write data
- `word_bitwidth` – Bitwidth of one word
- `byte_bitwidth` - Bitwidth of one least addressable unit (LAU), word bitwidth must be divisible by byte bitwidth
- `endianness` – either .`.big`or .`.little`, assembler currently supports both, but program linker and loader in simulator supports only big endianness

8.3.66 .set

```
DirectiveEquGroup
  : DirectiveEquGroupType Identifier "," Expression
```

```
DirectiveEquGroupType
  : ".equ"
  | ".set"
```

This `.set` directive sets the value of symbol to expression. It may redefine symbol value. It is synonymous with the directive `.equ`.

8.3.67 .short (16-bit)

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

The directive `.short` is normally the same as `.word`.

In some configurations, however, `.short` and `.word` generate numbers of different lengths.

8.3.68 .single

```
DirectiveDoubleGroup
  : DirectiveDoubleGroupType CompileDoubleExpressionList
```

```
DirectiveDoubleGroupType
  : ".double"
  | ".float"
  | ".single"
```

```
CompileDoubleExpressionList
  : %empty
  | DoubleConstant
  | CompileExpression
  | CompileDoubleExpressionList "," DoubleConstant
  | CompileDoubleExpressionList "," CompileExpression
```

The directive `.single` assembles zero or more flonums, separated by commas. It has the same effect as the directive `.float`.

8.3.69 .size

```
DirectiveSize
  : ".size" Identifier "," Expression
```

This directive sets the size associated with a symbol name. The size in bytes is computed from expression which can make use of label arithmetic. This directive is typically used to set the size of function symbols.

8.3.70 .skip

```
DirectiveSkipGroup
  : DirectiveSkipGroupType CompileExpressionValue MaybeFill
```

```
DirectiveSkipGroupType
  : ".skip"
  | ".space"
  | ".zero"
```

```
MaybeFill
  : "," CompileExpressionValue
  | %empty
```

This directive emits expression bytes, each of fill value. If the fill value is omitted, fill value is assumed to be zero. It is the same as the directives `.space` and `.zero`.

8.3.71 .sleb128

```
DirectiveSleb128Group
  : DirectiveSleb128GroupType ExpressionList
```

```
DirectiveSleb128GroupType
  : ".sleb128"
  | ".uleb128"
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

The directive `.sleb128` stands for “signed little endian base 128.” This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See the directive `.uleb128`.

8.3.72 .space

```
DirectiveSkipGroup
  : DirectiveSkipGroupType CompileExpressionValue MaybeFill
```

```
DirectiveSkipGroupType
  : ".skip"
  | ".space"
  | ".zero"
```

```
MaybeFill
  : "," CompileExpressionValue
  | %empty
```

This directive is the same as directive `.skip` and `.zero`.

8.3.73 .text

```
DirectiveDataGroup
  : DirectiveDataGroupType MaybeDirectiveDataGroupFlags
```

```
DirectiveDataGroupType
  : ".data"
  | ".text"
  | ".bss"
```

```
MaybeDirectiveDataGroupFlags
  : "," String
  | %empty
```

The directive `.text` tells assembler to assemble the following statements onto the end of the text section.

8.3.74 .type

```
DirectiveType
  : ".type" Identifier "," Identifier DirectiveTypeDescription
```

```
DirectiveTypeDescription
  : "," Identifier
  | %empty
```

The directive sets the type of symbol name to be either a function symbol or an object symbol. There are five different syntaxes supported for the type description field, in order to provide compatibility with various other assemblers.

Supported types are @function, @object, or @ followed by an identifier. Also versions with % instead of @ are supported.

8.3.75 .uleb128

```
DirectiveSleb128Group
  : DirectiveSleb128GroupType ExpressionList
```

```
DirectiveSleb128GroupType
  : ".sleb128"
  | ".uleb128"
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

The directive `.uleb128` stands for “unsigned little endian base 128.” This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. the See directive `.sleb128`.

8.3.76 .warning

```
DirectiveWarning
  : ".warning"
  | ".warning" String
```

Similar to the directive `.error`, but it emits only a warning.

8.3.77 .weak

```
DirectiveWeak
  : ".weak" Identifier
```

This directive sets the weak attribute on a comma-separated list of symbol names. If the symbols do not already exist, they will be created.

8.3.78 .word

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

This directive expects zero or more expressions, of any section, separated by commas.

The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

8.3.79 .zero

```
DirectiveSkipGroup
  : DirectiveSkipGroupType CompileExpressionValue MaybeFill
```

```
DirectiveSkipGroupType
  : ".skip"
  | ".space"
  | ".zero"
```

```
MaybeFill
  : "," CompileExpressionValue
  | %empty
```

This directive is the same as the directives `.skip` and `.space`.

8.4 Unsupported GNU Assembler Directives

When using the following directives, the warning is printed.

- .abort
- .desc
- .end
- .entry
- .fail
- .include
- .linkonce
- .octa
- .reloc
- .string, .string8, .string16, .string32, .string64
- .subsection

When using the following directives, the error is printed and object file is not generated.

- .usect

8.5 Ignored GNU Assembler Directives

When using the following directives, nothing is printed and these directives are silently ignored.

- .arch
- .cfi_lsda
- .cfi_personality
- .cfi_sections
- .cfi_signal_frame
- .cpload
- .cprestore
- .endfunc
- .ent
- .extern
- .fmask
- .func
- .gnu_attribute
- .hidden
- .lflags
- .mask

- .option
- .proc

8.6 GNU Assembler Directives Unknown to Assembler

When using the following directives, syntax errors are printed, because the assembler does not know these directives.

- .def
- .eject
- .else
- .elseif
- .endif
- .exitm
- .if
- .internal
- .irp
- .irpc
- .mri
- .print
- .protected
- .psize
- .purgem
- .rept
- .sbttl
- .sizem
- .stab
- .struct
- .symver

8.7 Annex: Assembly Language Keywords

Here is a list of the keywords in assembly language:

```
.abs_org
.address_space
.align
.altmacro
.ascii
.asciz
```

```
.balign
.balignl
.balignw
.big
.bit
.bss
.bundle_align_end
.byte
.cfi_adjust_cfa_offset
.cfi_def_cfa
.cfi_def_cfa_offset
.cfi_def_cfa_register
.cfi_endproc
.cfi_offset
.cfi_register
.cfi_return_column
.cfi_same_value
.cfi_startproc
.cfi_val_offset
.codasip_retstruct_reg
.codasip_retval_regs
.comm
.data
.debug_off
.debug_on
.double
.dword
.endm
.equ
.equiv
.eqv
.err
.error
.file
.fill
.float
.global
.globl
.half
.hword
.ident
.incbin
.int
.lcomm
.line
.list
.little
.loc
.local
.long
.macro
.noaltmacro
.nolist
.org
.p2align
.p2alignl
.p2alignw
.popsection
.previous
.profiler
.pushsection
.quad
.section
```

```
.section_adjustable
.set
.short
.single
.size
.skip
.sleb128
.space
.text
.type
.uleb128
.warning
.weak
.word
.zero
binary constant
decimal constant
hexadecimal constant
identifier
LOCAL
macro call
octal constant
string constant
```

8.8 Annex: Assembly Language Syntax Summary

The following syntax summary is valid for assembly programs; that is, `.s` or `.S` files.

```
DirectiveLocal
  : ".local" Identifier
```

```
DirectiveEquivGroup
  : DirectiveEquivGroupType Identifier "," Expression
```

```
DirectiveIncbin
  : ".incbin" String
  | ".incbin" String "," CompileExpressionValue
  | ".incbin" String "," CompileExpressionValue "," CompileExpressionValue
```

```
SectionFlags
  : %empty
  | "," String SectionType
```

```
DirectiveSectionAdjustable
  : DirectiveSectionAdjustableKeyword SectionName "," String "," CompileExpressionValue "," Com-
pileExpressionValue "," DirectiveSectionAdjustableEndianess
```

```
DirectiveErr
  : ".err"
```

```
DirectiveType
  : ".type" Identifier "," Identifier DirectiveTypeDescription
```

```
CompileExpressionValue
  : CompileExpression
```

```
Program
  : Program Directive "\n"
  | Program Label
  | Program MacroCall "\n"
  | Program SymbolDefinition "\n"
  | Program "\n"
  | Program error "\n"
  | %empty
```

```
DirectiveEquivGroupType
  : ".equiv"
  | ".eqv"
```

```
DirectiveAbsOrg
  : ".abs_org" CompileExpression
```

```
DirectiveEquGroup
  : DirectiveEquGroupType Identifier "," Expression
```

```
DirectiveDebugOn
  : ".debug_on"
```

```
DirectiveDataGroup
  : DirectiveDataGroupType MaybeDirectiveDataGroupFlags
```

```
MaybeFill
  : "," CompileExpressionValue
  | %empty
```

```
DirectiveAlignGroup
  : DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue "," Com-
pileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue "," CompileExpressionValue
  | DirectiveAlignGroupType CompileExpressionValue
```

```
DirectiveCfiAdjustCfaOffset
  : ".cfi_adjust_cfa_offset" CompileExpressionValue
```

```
DirectiveAltmacroLocal
  : "LOCAL" Identifier
```

```
DirectiveCfiOffset
  : ".cfi_offset" CompileExpressionValue "," CompileExpressionValue
```

```
DirectiveEquGroupType
  : ".equ"
  | ".set"
```

```
DirectiveSleb128GroupType
  : ".sleb128"
  | ".uleb128"
```

```
DirectiveCodasipRetstructReg
  : ".codasip_retstruct_reg" CompileExpressionValue
```

```
OctalConstant
  : "octal constant"
```

```
DirectiveDoubleGroup
  : DirectiveDoubleGroupType CompileDoubleExpressionList
```

```
DirectiveOrg
  : ".org" CompileExpression MaybeFill
```

```
MacroCall
  : "macro call" MaybeMacroCallArguments
```

```
String
  : "string constant"
```

```
Identifier
  : "identifier"
  | "."
```

```
DirectiveWarning
  : ".warning"
  | ".warning" String
```

```
Label
  : Identifier ":"
  | DecimalConstant ":"
```

```
DirectiveSectionKeyword
  : ".section"
```

```
CompileExpression
  : Expression
```

```
Expression
  : ExpressionAll
```

```
BinaryConstant
  : "binary constant"
```

```
DirectiveFill
  : ".fill" CompileExpressionValue
  | ".fill" CompileExpressionValue "," CompileExpressionValue
  | ".fill" CompileExpressionValue "," CompileExpressionValue "," CompileExpressionValue
```

```
DirectiveCfiReturnColumn
  : ".cfi_return_column" CompileExpressionValue
```

```
DirectiveAltmacro
  : ".altmacro"
```

```
HexadecimalConstant
  : "hexadecimal constant"
```

```
Start
  : Program
```

```
DirectiveSize
  : ".size" Identifier "," Expression
```

```
ExpressionCore
  : IntegerConstant
  | Identifier
  | "(" ExpressionAll ")"
```

```
DirectiveSleb128Group
  : DirectiveSleb128GroupType ExpressionList
```

```
DirectiveGlobalGroup
  : DirectiveGlobalGroupType DirectiveGlobalGroupSymbols
```

```
DirectiveCommGroup
  : DirectiveCommGroupType Identifier "," CompileExpressionValue MaybeDirectiveCommAlignment
```

```
DecimalConstant
  : "decimal constant"
```

```
DirectiveCfiRegister
  : ".cfi_register" CompileExpressionValue "," CompileExpressionValue
```

```
DirectiveList
  : ".list"
```

```
SectionType
  : %empty
  | "," SectionTypeSpecifier SectionFlagSpecificArguments
```

```
DirectiveFile
  : ".file" IntegerConstantValue String
  | ".file" String
```

```
MaybeDirectiveCommAlignment
  : "," CompileExpressionValue
  | %empty
```

```
DirectiveDataGroupType
  : ".data"
  | ".text"
  | ".bss"
```

```
DirectiveSection
  : DirectiveSectionKeyword SectionName SectionFlags
```

```
DirectiveLine
  : ".line" CompileExpressionValue
```

```
MaybeDirectiveDataGroupFlags
  : "," String
  | %empty
```

```
DirectiveNolist
  : ".nolist"
```

```
DirectiveBit
  : ".bit" CompileExpressionValue "," ExpressionList
```

```
IdOrString
  : String
  | Identifier
```

```
DirectiveCfiDefCfa
  : ".cfi_def_cfa" CompileExpressionValue "," CompileExpressionValue
```

```
DirectiveCfiDefCfaOffset
  : ".cfi_def_cfa_offset" CompileExpressionValue
```

```
DirectiveGlobalGroupSymbols
  : DirectiveGlobalGroupSymbol
  | DirectiveGlobalGroupSymbols "," DirectiveGlobalGroupSymbol
```

```
MaybeDirectiveLocOptions
  : Identifier MaybeDirectiveLocOptions
  | Identifier IntegerConstantValue MaybeDirectiveLocOptions
  | %empty
```

```
SectionName
  : Identifier
  | String
  | ".data"
  | ".text"
  | ".bss"
  | Identifier "-" Identifier
```

```
DirectiveDoubleGroupType
  : ".double"
  | ".float"
  | ".single"
```

```
DirectivePushsection
  : ".pushsection" SectionName SectionFlags
```

```
DirectiveCfiDefCfaRegister
  : ".cfi_def_cfa_register" CompileExpressionValue
```

```
ExpressionAll
  : ExpressionAll "|" ExpressionAll
  | ExpressionAll "^" ExpressionAll
  | ExpressionAll "&" ExpressionAll
  | ExpressionAll "<<" ExpressionAll
  | ExpressionAll ">>" ExpressionAll
  | ExpressionAll "+" ExpressionAll
  | ExpressionAll "-" ExpressionAll
  | ExpressionAll "*" ExpressionAll
  | ExpressionAll "/" ExpressionAll
  | ExpressionAll "%" ExpressionAll
  | "!" ExpressionAll
  | "~" ExpressionAll
  | "+" ExpressionAll
  | "-" ExpressionAll
  | ExpressionCore
```

```
DirectiveEndm
  : ".endm"
```

```
DirectiveIdent
  : ".ident" String
```

```
DirectiveCommGroupType
  : ".comm"
  | ".lcomm"
```

```
DirectiveProfiler
  : ".profiler" String
  | ".profiler" String "," String
```

```
DirectiveAsciiGroupType
  : ".ascii"
  | ".asciz"
```

```
SectionTypeSpecifier
  : Identifier
```

```
DirectiveWeak
  : ".weak" Identifier
```

```
MaybeDirectiveLocColumn
  : %empty
  | IntegerConstantValue
```

```
DirectiveByteGroup
  : DirectiveByteGroupType ExpressionList
```

```
ExpressionList
  : Expression
  | ExpressionList "," Expression
```

```
CompileDoubleExpressionList
  : %empty
  | DoubleConstant
  | CompileExpression
  | CompileDoubleExpressionList "," DoubleConstant
  | CompileDoubleExpressionList "," CompileExpression
```

```
SectionFlagSpecificArguments
  : %empty
  | "," IntegerConstantValue
  | "," IdOrString "," Identifier
  | "," CompileExpressionValue "," IdOrString "," Identifier
```

```
DirectiveCfiStartproc
  : ".cfi_startproc"
```

```
DirectiveCfiValOffset
  : ".cfi_val_offset" CompileExpressionValue "," CompileExpressionValue
```

```
DirectiveSectionAdjustableEndianess
  : ".little"
  | ".big"
```

```
DirectiveNoaltmacro
  : ".noaltmacro"
```

```
DirectiveAlignGroupType
  : ".align"
  | ".balign"
  | ".balignw"
  | ".balignl"
  | ".p2align"
  | ".p2alignw"
  | ".p2alignl"
```

```
Directive
  : DirectiveAbsOrg
  | DirectiveAddressSpace
  | DirectiveAlignGroup
  | DirectiveAltmacro
  | DirectiveAltmacroLocal
  | DirectiveAsciiGroup
  | DirectiveBit
  | DirectiveBundleAlignEnd
  | DirectiveByteGroup
  | DirectiveCfiStartproc
  | DirectiveCfiEndproc
  | DirectiveCfiDefCfa
  | DirectiveCfiDefCfaRegister
  | DirectiveCfiDefCfaOffset
  | DirectiveCfiAdjustCfaOffset
  | DirectiveCfiOffset
  | DirectiveCfiValOffset
  | DirectiveCfiRegister
  | DirectiveCfiSameValue
  | DirectiveCfiReturnColumn
  | DirectiveCodasipRetstructReg
  | DirectiveCodasipRetvalRegs
  | DirectiveCommGroup
  | DirectiveDataGroup
  | DirectiveDebugOff
  | DirectiveDebugOn
  | DirectiveDoubleGroup
  | DirectiveEndm
  | DirectiveEquGroup
  | DirectiveEquivGroup
  | DirectiveErr
  | DirectiveError
  | DirectiveFile
  | DirectiveFill
  | DirectiveGlobalGroup
  | DirectiveIdent
  | DirectiveIncbin
  | DirectiveLine
  | DirectiveList
  | DirectiveLoc
  | DirectiveLocal
  | DirectiveMacro
  | DirectiveNoaltmacro
  | DirectiveNolist
  | DirectiveOrg
  | DirectivePopsection
  | DirectivePrevious
  | DirectiveProfiler
  | DirectivePushsection
  | DirectiveSection
  | DirectiveSectionAdjustable
  | DirectiveSize
  | DirectiveSkipGroup
  | DirectiveSleb128Group
  | DirectiveType
  | DirectiveWarning
  | DirectiveWeak
```

```
DirectiveTypeDescription
  : "," Identifier
  | %empty
```

```
DirectiveError
  : ".error"
  | ".error" String
```

```
DirectiveDebugOff
  : ".debug_off"
```

```
DirectiveMacroArguments
  : %empty
  | DirectiveMacroArguments Identifier
```

```
DirectiveCodasipRetvalRegs
  : ".codasip_retval_regs" CompileExpressionList
  | ".codasip_retval_regs"
```

```
MaybeMacroCallArguments
  : %empty
  | MacroCallArguments
```

```
DirectiveSkipGroup
  : DirectiveSkipGroupType CompileExpressionValue MaybeFill
```

```
DirectivePopsection
  : ".popsection"
```

```
DirectiveMacro
  : ".macro" Identifier DirectiveMacroArguments
```

```
DirectiveSectionAdjustableKeyword
  : ".section_adjustable"
```

```
MacroCallArgument
  : Identifier
  | "(" Identifier ")"
```

```
DirectiveCfiSameValue
  : ".cfi_same_value" CompileExpressionValue
```

```
DirectiveBundleAlignEnd
  : ".bundle_align_end"
```

```
DirectiveByteGroupType
  : ".byte"
  | ".hword"
  | ".int"
  | ".long"
  | ".quad"
  | ".short"
  | ".word"
  | ".half"
  | ".dword"
  | ".2byte"
  | ".4byte"
```

```
DirectiveAsciiGroup
  : DirectiveAsciiGroupType StringList
```

```
DirectiveLoc
  : ".loc" IntegerConstantValue IntegerConstantValue MaybeDirectiveLocColumn May-
beDirectiveLocOptions
```

```
DirectiveCfiEndproc
  : ".cfi_endproc"
```

```
DirectiveAddressSpace
  : ".address_space" CompileExpressionValue
```

```
DirectiveGlobalGroupSymbol
  : Identifier
```

```
DirectiveSkipGroupType
  : ".skip"
  | ".space"
  | ".zero"
```

```
DoubleConstant
  : MaybeMinus "floating-point constant"
```

```
StringList
  : String
  | StringList "," String
```

```
MacroCallArguments
  : MacroCallArgument
  | MacroCallArguments "," MacroCallArgument
```

```
MaybeMinus
  : %empty
  | "-"
```

```
CompileExpressionList
  : CompileExpression
  | CompileExpressionList "," CompileExpression
```

```
IntegerConstant
  : BinaryConstant
  | OctalConstant
  | DecimalConstant
  | HexadecimalConstant
```

```
DirectiveGlobalGroupType
  : ".global"
  | ".globl"
```

```
DirectivePrevious
  : ".previous"
```

```
IntegerConstantValue
  : IntegerConstant
```

8.9 Disassembler

Guide for disassembler usage can be found in "Assembler, Disassembler and Linker" section in *Codasip Studio User Guide*. List of disassembler arguments can be found here: "Tool arguments reference" on page 195.

8.10 Process of assembling and disassembling

Information about how the Codasip Assembler produces binary output from information provided by the model and how the Codasip Disassembler does the reverse process can be found in this chapter. Description about little-endian conversions, parcel bytes, bundling and debundling functions is provided.

8.10.1 Process parcels in assembler.

Process parcels in assembler is a subprocess in assembling process. Both input and output is big-endian. It is defined as follows:

1. This phase is skipped when a program memory is using big-endian or when parcel bytes are not provided. `le_instruction_parcel_bytes` option is not specified into the `assemblersettings` section.
2. Instruction is split into parcel parts of size determined by the given parcel bytes. Then order of these parts is reverted and the instruction is recreated from the new order.

8.10.2 Process parcels in disassembler.

Process parcels in disassembler is a subprocess in disassembling process. Input is in program memory endianness and output is big-endian. It is defined as follows:

1. This phase is skipped when program memory is using big-endian.
2. Parcel size is determined from parcel bytes and byte size. Raw data size is parcel size, when parcel bytes are not provided, `le_instruction_parcel_bytes` option is not specified into the `assemblersettings` section.
3. Raw data is split into parcels of parcel size. Then order of bytes in these parts are reverted and these parts create the new data.

8.10.3 Assembling process

Assembling process is done iteratively over each parsed instruction as follows.

1. Parse instruction of the input program.
2. Select all elements, sets and attributes, that match parsed instruction with their assembler sections.
3. Binary sections of selected elements and attributes are used for binary output generation. This binary output is encoded as big-endian now. Instructions are split into binary slots, when the model is VLIW.
4. A bundling function can be optionally used. It takes binary slot instructions encoded as big-endian and produce one binary bundle instruction, that is pot-tentionally compressed or filled with padding, encoded as big-endian.
5. Process parcels in binary instruction, described in "Process parcels in assembler." on page 177.
6. Revert all bits of the binary instruction or bundle when program memory is little-endian.

8.10.4 Disassembling process

The disassembling process works in opposite to the assembling process as follows.

1. Determine lowest instruction bit-width from the model.
2. Determine recent decoded instruction bit-width.
   a. This phase is skipped if it is obvious from the model, recent decoded instruction bit-width is then lowest instruction bit-width.
   b. Read lowest instruction bit-width as raw bytes. Raw bytes are read from the object file as is except that bits are reverted if the section is little-endian.
   c. Process parcels in read raw bytes, described in "Process parcels in assembler." on page 177. If function is skipped, revert byte order is used when program memory is little-endian.
   d. Now determine recent decoded instruction bit-width.
3. Read recent decoded instruction bit-width as raw bytes. Raw bytes are read from the object file as is except that bits are reverted if the section is little-endian.

4. Process parcels in read raw bytes, described in "Process parcels in assembler." on page 177. If the function is skipped, revert byte order is used when program memory is little-endian.
5. A debundling function can be optionally used. It takes undecoded data of size of the bundle instruction size + maximum padding, that is pottentionally compressed or filled with padding, encoded as big-endian. It produces debundle structure, which contains undecoded binary slot instructions and size of the read input data.
6. Select all elements, sets and attributes, that match decoded instruction with their binary sections.
7. Assembler sections of selected elements and attributes are used for assembler output generation.

8.10.5 Examples of assembling and disassembling

In this section, some examples will be provided. Each byte is denoted as a couple of same letters. Bit positions are not taken into account.

First example is straightforward. Program memory is big-endian with word bit-width 32, LAU bit-width 8. It has only one size of instructions 32.

Example 20: Big endian, one instruction size

```
Assembler
1) "some instruction"
2) function: "some instruction" => AABBCCDD
3) 5) 6) AABBCCDD

Disassembler
1) 2) 32
3) 4) AABBCCDD
6) function: AABBCCDD => "some instruction"
7) "some instruction"
```

Second example uses a model similar to the model used in the first example. The only difference is that the program memory is little-enidan.

Example 21: Little endian, one instruction size

```
Assembler
1) "some instruction"
2) function: "some instruction" => AABBCCDD
3) 5) AABBCCDD
6) DDCCBBAA

Disassembler
1) 2) 32
3) DDCCBBAA
4) AABBCCDD
```

```
6) function: AABBCCDD => "some instruction"
7) "some instruction"
```

Third example is the same as second example and option `le_instruction_parcel_bytes` is set on 2 provided to the `assembler section`.

Example 22: Little endian, one instruction size, parcel bytes 2

```
Assembler
1) "some instruction"
2) function: "some instruction" => AABBCCDD
3) AABBCCDD
5) CCDDAABB
6) BBAADDCC

Disassembler
1) 2) 32
3) BBAADDCC
4) AABBCCDD
6) function: AABBCCDD => "some instruction"
7) "some instruction"
```

Fourth example is based on the second example, but it features multiple instruction sizes, 16 and 32.

Example 23: Little endian, more instruction sizes

```
Process for the small instruction size

Assembler
1) "small instruction"
2) function: "small instruction" => AABB
3) 5) AABB
6) BBAA

Disassembler
1) 16
2.2) BBAA
2.3) AABB
2.4) 16
3) BBAA
4) AABB
6) function: AABB => "small instruction"
7) "small instruction"

Process for the large instruction size

Assembler
1) "large instruction"
2) function: "large instruction" => AABBCCDD
3) 5) AABBCCDD
6) DDCCBBAA

Disassembler
1) 16
2.2) DDCC
2.3) CCDD
2.4) 32
3) DDCCBBAA
4) AABBCCDD
6) function: AABBCCDD => "large instruction"
7) "large instruction"
```

8.11 Linker

When used from the Codasip Commandline the Linker command, `ld`, has a number of options (see "Tool arguments reference" on page 195). However, when used from Codasip Studio, only a few options are visible and they are specified through the linker settings parameters of the **C/C++ Project**. The Linker is automatically used when the project is built. Gnu linker scripts are used in this case.

9 COMPILER GENERATOR

This chapter describes three aspects of the Compiler Generator:

- Builtin functions of particular relevance to Compiler Generation
- The semantics description file produced by the semantics extractor - the font end of Compiler Generation
- The syntax of the rule file used with the Compiler Generation backend

More details on the use of these items can be found in the *Codasip Studio User Guide*.

This chapter is organized as follows:

9.1 About Builtins

Builtins provide operations that would be hard to describe using the CodAL language alone. Also, some operations such as carry generation from addition can be defined in multiple ways, and using functions from the builtins library allows the compiler generator to identify them. The builtins library also contains functions that annotate instructions for the C/C++ compiler generator, giving it additional guidance.

Builtin functions are mainly intended for use in instruction accurate CodAL models, because they cannot be synthesized into RTL.

The builtins library provides the following types of operations:

- Annotations for the compiler generator
- Wide loads and stores for compiler use
- Flag computation
- Saturated arithmetic
- Floating point support

- Conversions
- Comparisons
- Special operations

All builtins are described in a section "Builtin Functions" on page 208. Only several of them that are specific to compiler generation are listed here.

9.2 Builtins for Semantics Extraction

There are several functions that control the behavior of the Semantics Extractor, used by the Compiler Generator and other tools. In fact, the Compiler Generator is the dominant influence on the Semantics Extractor, and the vast majority of the builtins described in this section are there to serve its needs.

9.2.1 Common Annotations for Compiler Generation

These functions are used to annotate instructions for the semantics extraction process. As far as the simulator is concerned, these builtins are empty and can be ignored. There is no need to encapsulate them in a compiler pragma, for example.

9.2.1.1 codasip_compiler_unused

```
void codasip_compiler_unused()
```

Sometimes, the user wants to specify that the compiler should not use an instruction, even though its semantics can be correctly extracted. You can check the correct usage of this annotation in the instruction semantics file in the instruction usage specification (specifiers can be: ok, unused, undefined, or extractor removed).

9.2.1.2 codasip_compiler_undefined

```
void codasip_compiler_undefined()
```

To explicitly specify that semantics of an instruction are not correct, call to a function must be present in the instruction's semantics. Compiler will not use this instruction.

9.2.1.3 codasip_compiler_priority

```
void codasip_compiler_priority(int32 priority)
```

This annotation specifies priority of an instruction for the compiler instruction selector. Within one phase of compiler generation, the generator sorts semantically equivalent instructions based on some internal rules such as the number of side effects, and binary coding length. For some particular operations such as register moves, it selects the highest ranked instructions. Also for instruction selector rules generation, it sorts the equivalent instructions according to the internal order.

To change this default ordering, instruction priority can be set. Default priority of instructions is 0, it can be set to any higher value such as 5 or 10.

A negative value has a specific meaning, this tell the compiler generator to use these instructions during instruction set analysis, e.g. to find instructions that form a function prologue (stack manipulation), but these instructions won't be used during instruction selection, because their instruction selection pattern will not be generated.

9.2.1.4 codasip_compiler_builtin

```
void codasip_compiler_builtin()
```

This function instructs the compiler generator to generate a builtin function for this instruction. After compiler generation two files inlines.c and inlines.h are generated in the CodAL project's directory `work/<model_name>/compiler/compiler`. These files contain builtin functions with inline assembler. The compiler generator may remove some complex instructions that cannot be used automatically. This also disables them as builtins. You can use the annotation codasip_compiler_unused() together with codasip_compiler_builtin() to force builtin generation.

9.2.1.5 codasip_nop

```
void codasip_nop()
```

This is a specification of an explicit nop instruction. It may be needed for compiler for architectures that do not handle data or structural hazards, opr require delay slots after jumps.

9.2.1.6 codasip_compiler_flag_cmp_datatype

```
void codasip_compiler_flag_cmp_datatype(datatype src1, datatype src2)
```

This is an explicit specification of an compare operation that produces the comparison result in some specific flags. The rulelib matcher from compiler generator can then use this instruction as a comparison instruction.

The allowed variants of datatype are: int8, int16, int32, int64, float, double, and long double (long_double in function name).

An example of such specific compare instruction follows. The example architecture has floating point equal (feq), lower or equal (fle), and ordered flags (ford). You can see that the setting of flags is present in the semantics section and the usage of function codasip_compiler_flag_cmp_ is really used only a guidance for the compiler generator.

Example 24:Specific compare instruction

```
element instr_compare_float
{
    use opcode_fcmp;
    use reg_gpr as reg_src1, reg_src2;
    assembler {opcode_fcmp reg_src1 "," reg_src2};
    binary {opcode_fcmp 0:5 reg_src1 reg_src2 0:11};

    semantics {
        float src1, src2, res;
        src1 = codasip_int32_to_float_bitcast(regs[reg_src1]);
        src2 = codasip_int32_to_float_bitcast(regs[reg_src2]);

        feq = codasip_fcmp_oeq_float(src1, src2);
        fle = codasip_fcmp_ole_float(src1, src2);
        ford = codasip_fcmp_ord_float(src1, src2);

        codasip_compiler_flag_cmp_float(src1, src2);
    };
};
```

Example 25:Specific compare instruction

```
element instr_compare_float
{
    use opcode_fcmp;
    use reg_gpr as reg_src1, reg_src2;
    assembler {opcode_fcmp reg_src1 "," reg_src2};
    binary {opcode_fcmp 0:5 reg_src1 reg_src2 0:11};

    semantics {
        float src1, src2, res;
        src1 = codasip_int32_to_float_bitcast(regs[reg_src1]);
        src2 = codasip_int32_to_float_bitcast(regs[reg_src2]);

        feq = codasip_fcmp_oeq_float(src1, src2);
        fle = codasip_fcmp_ole_float(src1, src2);
        ford = codasip_fcmp_ord_float(src1, src2);

        codasip_compiler_flag_cmp_float(src1, src2);
    };
};
```

Then in the user rulelib file (ia/compiler/user_rules.rl), the compare specification is for example for conditional branch that compare floating point operations used like this:

Example 26: Simple compare pattern

```
(seteq reg:0, op:1):
    (seteq (xor :0, :1), 0) |
    (select (seteq :0, :1), 1, 0) ;
```

The second equivalence rule uses the instruction specified with the codasip_compiler_ flag_cmp_float annotation and the second instruction (that is a conditional branch jumping if flag feq is set to one) is specified by the user manually.

9.2.1.7 codasip_compiler_predicate_true/false

```
void codasip_compiler_predicate_true(uint1 predop)
void codasip_compiler_predicate_false(uint1 predop)
```

These two instructions are used to specify instruction predicates. Instruction predicate can be built only from 1-bit registers. Predication is a little tricky to describe correctly, if this is needed, please contact the support@codasip.com for further information.

9.3 Semantics Description Format

This section describes the structure and semantics of the semantics extraction result file. This instruction semantic s file is used for several purposes: 1) C compiler generation, 2) documentation generatio and 3) random assembler programs generation. For each of these tools, the file is slightly different, because different type of information must be captured. For example, for the C compiler generation, one can find the instruction semantics file under CodAL project in a file `work/ia/report/semextr/compiler.sem`.

9.3.1 Header

The first line of the results file describes the version of the semantics extractor. The second line contains the name of the processor (architecture) for which the semantics were extracte. Finally, the third line contains information about the endianness of the architecture. Endianess may be either big or little.

Example 27: Semantics extraction header

```
version "2.0.0"
processor codasip_urisc.ia
endian big
```

9.3.2 Registers

There are two views on registers from the architectural point of view. First one are physical registers that directly correspond to registers and register fields in a processor. The second view is as instruction operands, register operands are viewed as logical registers from a register class. There can be more than one register class that maps to a physical register field.

Data type of register is not important here because it is defined in definition of instruction, where it is used. Every register from one register field has to have the same bit size.

Example 28:Register class bitwidths

```
//'reg' name, bit_width, address
reg rf_gpr, 32, 0
reg rf_gpr, 32, 1
...
//'regop' class_name, resource_name, bit_width, syntax, address
regop gpreg, rf_gpr, 32, "R0", 0
regop gpreg, rf_gpr, 32, "R1", 1
...
```

9.3.3 Application Binary Interface (ABI)

For the generated C compiler, the call stack formed by the C compiler has a fixed format. The first thing on the stack are arguments that could not fit into registers. Then the return address is stored (when needed), then the previous stack pointer value (also when needed). All function local variables are stored after that.

One can specify the compiler ABI in a CodAL model using section `settings { compiler { ... }; };`.

Below is an example of how the ABI specification looks in the instruction semantics format for the MIPS architecture.

Here are the physical register definitions.

Example 29: Physical register definition

```
reg regs, 32, 0
reg regs, 32, 1
...
reg regs, 32, 31
```

Now in the file user_semantics.sem can the user add the ABI specification:

Example 30: Adding ABI specification

```
// Stack pointer
stack_pointer = regs(1)
// Base pointer
base_pointer = regs(2)
// To save these registers is a responsibility of the callee,
the caller does not need to store them.
callee_saved = { regs(16), regs(17), regs(18), regs(19), regs(20), regs(21), regs(22), regs(23) }
// Registers used to pass arguments, when more values that is here specified
// is needed, the rest of arguments is stored to stack.
function_result = { regs(3), regs(4), regs(5), regs(6) }
// Explicit specification of register to store
// call return address, this is usually automatically determined
// from the instruction semantics
return_address = regs(31)

// An usused register, e.g. a register for exception return address.
// The compiler will never use an unused register.
```

```
unused_registers = { regs(29) }
// Stack can grow either downwards, or upwards,
// the oonly currently supported direction is downwards.
stack_direction = down
```

While all of these directives are optional (when automatic inference is sufficient), some architectures may actually require them. For example, specification of SP (and BP) is required if instructions use SP (and BP) as fixed registers (instead of normal operands).

Furthermore, there may be additional information for the compiler generator present.

Explicit specification of legal types supported by the architecture, this assures that e.g. a 1-bit variables will not be used even if there is partial support for them in the architecture:

Example 31: Legal data types

```
legal_types = { i32 }
```

Specification for architectures whose instruction set provides unaligned loads and stores. When this option is enabled, all instructions that are usable only for aligned access must have their addresses masked to the an aligned access. When there is no mask in address computation, the compiler generator assumes that this may be an unaligned load or store.

Example 32: Representative register class

```
representative_register_class type, reg_class
```

Each data-type can have only one primary (logical) register class for housing it. Use this keyword if automatic detection yields unwanted results.

Example 33: Pointer register class - beware of a possible conflict with explicit data-layout

```
pointer_reg_class = reg_class
```

Very similar to previous keyword but specifically for pointers (that's something a little bit different than an integer of the same width). Can be set to a different class than a representative class for an integer of the same width, if set.

Example 34: Pointer width

```
pointer_size = N
```

Example 35: Setting unaligned memory access

```
may_have_unaligned_loadstore = 1
```

When the architecture has no instruction that can be automatically identified to perform register spilling (store/load to/from stack), an explicit specification is needed. This is an example for the MIPS architecture that uses 1-bit register file cop1_cc_op for floating point conditions. However to spill these registers a complex combination of instruction is needed. With the explicit_copyab specification, the user says that implementation for register spilling is provided in a file CodasipInstrInfo.cpp (see the MIPS model for an example).

Example 36: Setting explicit specification (MIPS)

```
explicit_copyab cop1_cc_op
```

Example 37: Setting stack alignment explicitly (per each stack) - beware of a possible conflict with explicit data-layout

```
stack_alignment(N) = A
```

9.3.4 Subinstructions

If source model does not contain subinstructions, this section will be empty. Subinstructions are used only for instruction set documentation generation. Subinstruction does not have input arguments. But it has operands, syntax and binary section. Semantically, subinstructions are the same as instructions which are described in the next section.

Under the list of subinstructions are sets of subinstructions that define sets of subinstructions. The subinstruction set is then used similarly as an instruction operand.

Example 38: Subinstruction set

```
// subinstr_set defines a set of conditions
subinstr_set cond_all = cond_eq, cond_ne, ..., cond_al;
```

9.3.5 Instructions

Every instruction modeled architecture is described by its name, semantics in SSA form, syntax, binary coding and latency.

9.3.5.1 Data types

The following data types are supported:

```
iX, where X is 1, 2, ... represents integer value with X bit size, signedness depends on the per-
formed operation
float, double, x86_fp80 - floating point data types
vXiY - vectors data types, X is number of elements, Y is bit size basic type
```

9.3.5.2 Operands

This part contains operands which are either registers, immediate values or subinstructions. Bit size for register is known from its register class, bit size for immediate value is know from semantics and bit size for subinstruction is explicitly defined. These operands are used in instruction's semantics by operations.

Example 39: Instruction operands options

```
instr instr__add__reg__uimm12__, ok,
{ imm_1 = immop(), regstd_0 = regop(regstd), update_flag = subinstrop(update_flag) },
...
```

9.3.5.3 Operations

Semantics of instruction is described by operations. In the following list, t may be any data type, f is any floating-point type, and ix is arbitrary data type with bit size x.

Operations are based on LLVM IR, for detailed description, please see a document at llvm.org/docs/LangRef/html.

- reading register operand `t regop(id)`;
- reading immediate operand `t immop(id, sign, real_width/0)`;
- conversion `ix op_conv(iy src, iz)`, where op_conv is one of sext, zext, trunc, bitcast;
- binary arithmetic operations `ix op_arithm_log(ix, ix)`, where op_arithm_log is one of add, sub, mul, sdiv, udiv, srem, urem, and, or, xor;
- binary logical operations `ix op_shift_or_bitcnt(ix, iy)`, where op_shift_or_bitcnt is one of shl, sra, srl, shifter operand (of type iy) is always masked to have value lower than x;
- bit-count operations `ix op_bitcnt(ix src)`,where op_bitcnt is one of op_bitcnt: cttz (count trailing zeros), ctlz (count leading zeros), ctpop (count set bits);
- floating-point arithmetic operations `f op_arithm_float(f, f)`, where op_arithm_floaf is one of op_arithm_float: fadd, fsub, fmul, fdiv, frem;
- load/store operations `t load(t, ix addr), void store(t val, iy addr)`;
- conditional assignment `t select(iy cond, t trueval, t falseval)` is used to choose one value based on a condition, without branching;
- conditional execution `void if (iy cond, void act1; void act2)`;
- compare `i1 op_cmp (ix src1, ix src2)`, where op_cmp is one of seteq, setgt, setugt, setge, setuge, setlt, setult, setle, setule, setne, or one of floating point compare fcmpoeq, fcmpogt, fcmpoge, fcmpolt, fcmpole, fcmpone, fcmpord, fcmpuno, fcmpueq, fcmpugt, fcmpuge, fcmpult, fcmpule, fcmpune;
- jump operation `void brc(ix addr, iy cond)` jump to value given by argument;
- reading value of pc `ix getcurrpc(void)`;

- operation nop `void nop()`, this says that this instruction should be used as no-operation where needed;
- vector operation `extractelement <n x <ty>> <val>, i32 <idx>`, extracts a single scalar element from a vector at a specified index;
- vector operation `insertelement <n x <ty>> <val>, <ty> <elt>, i32 <idx>` inserts a scalar element into a vector at a specified index;
- vector operation `shufflevector <n x <ty>> <v1>, <n x <ty>> <v2>, <m x i32> <mask>` constructs a permutation of elements from two input vectors, returning a vector with the same element type as the input and length that is the same as the shuffle mask ;
- also all special operations as described in the Special operations section may be used in the description (with the same name, but without the codasip_ prefix and data type suffix).

9.4 Rule File Syntax for Compiler Generation Backend

The syntax is described here in the same way as the syntax of CodAL.

Every rule consists of a left side and any number of right sides separated as described:

```
<expr>":" [ <expr> { "|" <expr> } ] ";"
```

The left side is called a *request* while the right sides are called *emulations*. The system will determine whether there is a native support for the request and attempts to emulate as described in emulations if not.

All the needed requests are already provided in the generated skeleton so all that is required is to fill-in an emulation. Hence, the previous syntax definition could be simplified as follows:

```
<expr>":" <expr> ";"
```

The existing requests in the library file cover all operations that can be handled by the rule library, so only the pre-generated requests present in the default file should be used.

The expressions consist of nested abstract and untyped expressions in postfix notation.

```
expr := "(" <operation> <operand> { "," <operand> } ")"
operand := ( <expr> | <parameter> | <constant> | <special> )
```

The list of available operations is provided lower. A constant is an integer number in decimal notation. Parameters provide interface between a request and its emulations. In the request, they are denoted as a specifier and an order number separated by a colon. The specifier can be "**`reg`**", "**`imm`**", "**`op`**" (which stands for both) or "**`bb`**" (which can be used only as a target of a jump). In an emulation, the specifier should be omitted. The numbers of the parameters should begin at 0 and continue incrementally.

The special operands are used in flag architectures to denote flag registers: "cf", "of", "sf", "zf". Another special operand is "bb_skip" which is used in phi constructions (see the examples lower). It should be noted that this kind of operands is unlikely to be used since the system emulations utilizing them should suffice for any architecture. It's advised to conform to commonly recognized standards when designing a flag architecture.

Sometimes (see the next section) it's needed to directly reference certain registers. This is denoted like this (also see the example in the next section):

```
$ <class> ( n )
```

Here is a list of the available operations (with allowed operands):

- brcond - expr, bb

A conditional jump. The first operand describes a condition.

- select - expr, expr, expr

A conditional value select. The operands are a condition, true value and false value.

- set(u/o)eq/ne/ge/gt/le/lt - expr, expr

A compare operation. Can be used both as a root operation or in a condition. The naming of these operations follows internal naming of LLVM which overlaps operation codes used for comparing of integers and those used for floats. Hence, a more detailed list is provided.

Setcc over integers:

- seteq, setne - equal and not equal, sign does not matter
- setge,gt,le,lt - signed greater or equal, greater, less or equal, less
- setuge,ugt,ule,ult - unsigned greater or equal, greater, less or equal, less

Setcc over floats:

- seto, setu - ordered, unordered
- setoeq, one, oge, ogt, ole, olt - ordered comparison
- setueq, une, uge, ugt, ule, ult - unordered comparison
- seteq, ne, ge, gt, le, lt - doesn't matter (which should be the same thing as unordered)

`Note: operation "setcc unordered" is denoted as "setuo" in LLVM. However, the same operation is written as "setu" in rulelib.`

As you can notice, operations for unsigned integers are named the same way as operations for unordered floats (with the exception of eq and ne) and signed integers share naming with "don't care" floats. Only comparison of ordered floats doesn't overlap.

- xor/and/or/shl/srl/sra/add(c)/sub(c) - expr, expr

Various arithmetic operation. They aren't needed usually but some wilder architectures might use them. Please remember that operations "xor", "and" and "or" represent bitwise operations (LLVM doesn't have a logical "xor", "and" or "or"). See the note on top of the default RL file (which is generated when no custom rl file is supplied) for more information.

- zext - expr

Zero extend.

- lptr - imm

A meta-operation signifying load of a big absolute value (pointer or a large constant). Can appear only on a left side.

It's advised to tweak the instruction set instead and let the generator deal with this automatically. When one does use this nonetheless, it's necessary to load a big imm in no more than two steps (due to technical constraints stemming from the implementation of LLVM and compiler generator).

- cmp - reg, op

A flag compare (such as the one utilized by x86). Can be used only on a right side.

- phi expr, expr

Used for emulation of selects on flag architectures (such as x86). Can be used only on a right side. See examples.

C single-line comments are supported in rule-libraries.

For an extensive set of examples of basic (platform independent) usage of rules, see the default rulelib file (see the beginning of this chapter). However, a set of simplified examples was prepared for your convenience right here.

This is a simple compare pattern:

Example 40: Simple compare pattern

```
(seteq reg:0, op:1):
    (seteq (xor :0, :1), 0) |
    (select (seteq :0, :1), 1, 0) ;
```

On the left side (the first line), you can see an equality compare. The first operand is always a register while the second one will cause the pattern to be split into two distinct patterns: one with a register in that place and another one with an immediate operand.

On the right side, there are two emulations. The first one utilizes bitwise XOR and then tests the result on zero (some architectures have one-operand compare instructions which can only test the input operand with a certain constant). The second emulation is for architectures which do not contain compares at all and have to do with selects.

Here is an example of a conditional jump:

Example 41:Conditional jump

```
(brcond reg:0, bb:1):
    (brcond (setne :0, 0), :1) |
    (cmp :0, 0), (brcond (xor zf, -1), :1) ;
```

The first emulation is there for cases when an input condition is wrapped in a test for being non-zero. The other utilizes a flag jump and has two parts. The first one performs the actual compare while the other describes an x86-like jump (NZ in this case).

Here is a conditional move/select:

Example 42: Conditional move/select

```
(select reg:0, op:1, op:2):
    (select (setne :0, 0), :1, :2) |
    (cmp :0, 0), (brcond (xor zf, -1), bb_skip), (phi :1, :2) |
    (brcond :0, bb_skip), (phi :1, :2) ;
```

The first emulation pattern isn't interesting but the other two are. Only the first of them utilizes flag compare but both of them use a phi construction. See further for explanation about phi.

These three examples cover the most usual usage of rulelib. Specialties are described in the following sections. There is only one more possible type of a rule:

Example 43: Additional rule type

```
(lptr imm:0): ;
```

This rule serves for searching for an emulation of a load of a big constant value (or a global address). As mentioned earlier, it's better to leave this rule alone and let the system take care of it automatically. If that fails, use user_semantics to supply instruction (s) which have the expected format. E.g. on an i32 architecture that would an instruction which loads an i16 imm value and shifts it left by 16 bites and an instruction which performs a bitwise OR over a register and i16 imm value.

10 SDK TOOLS ARGUMENTS

During the SDK generation, tool binaries are created, which are located in `.../sdk/bin/` and can be run directly from a command line. This section lists the arguments that can be used along with these tools.

10.1 Documentation Conventions

The table below defines the syntax conventions used in Codasip Commandline commands.

`-l`	Letters starting with one dash indicate letters you enter literally. These letters represent short option names.
`--literal`	Words starting with two dashes indicate keywords you enter literally. These keywords represent long option names.
`|`	Vertical bars (OR-bars) separate possible choices for a single argument.
`<arguments>`	Words inside of angle brackets indicate arguments for which you must substitute a name or a value. When used with OR-bars, they enclose a list of choices from which you can choose one.
`[]`	Brackets indicate non-required options. More options inside of brackets indicate state option. All options inside of brackets are binded to the first one. `[-s <state> ... [-b <binded1>] [-b <binded2>]]`
`{}`	Braces indicate that a choice is required from the list of options separated by OR-bars. Choose one from the list. `{--option1|--option2|--option3}`
`...`	Three dots indicate that you can repeat the previous option.

10.2 Tool arguments reference

10.2.1 Assembler

Copyright (C) 2017 Codasip Ltd. Assemble input assembly files into output object file.

10.2.1.1 Usage

```
 -o <file> [-r] [-g <number>] [-a] [-m] [-t] [-x] [-s] [-w] [-
i <file>] [-q] [-V] ... [--] [--version] [-h] <file>
```

10.2.1.2 Descriptions

-o <file>, --output <file>
(required) Path to output binary executable (.xexe).

-r, --no-retval-info
Do not generate Codasip DWARF extension for return values.

-g <number>, --debug-level <number>
Debug level, default is 1 to generate assembly line info or equivalent 'line-info-only', 0 to disable line info.

-a, --print-address-space-info
Print address space information, all other options except -d are ignored.

-m, --get-memory-info
Print memory information, all other options except -d are ignored.

-t, --no-remove-two-dots-id
Do not change two dots on at the beginning of an id to just one dot.

-x, --comm-to-current-section
Always put common symbols declared with .comm and .lcomm to the current section, by default these symbols are put into .data section

-s, --section-names-suffix
Use input file as suffix of section name for .text and .data sections.

-w, --Wnooverflow
Do not print overflow warnings.

-i <file>, --input <file>
(Deprecated) Path to input assembly file.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help

Displays usage information and exits.

<file>
Path of input assembly file.

10.2.2 Disassembler

Copyright (C) 2017 Codasip Ltd. Disassembler input object files into output assembly file.

10.2.2.1 Usage

```
 <input> [-S] [--no-show-raw-insn] [-E] [-e] [-D] [-d] [--
stop-address <ADDR>] [--start-address <ADDR>] [-o <output>] [-
q] [-V] ... [--] [--version] [-h]
```

10.2.2.2 Descriptions

<input>
(required) Path of input object file.

-S, --source
Intermix source code with disassembly (not supported, only for compatibility)

--no-show-raw-insn
Display objdump output without raw binary coding, used only when -d or -D is used

-E, --disassemble-simple-all
Display assembler contents of all sections in a simplified form

-e, --disassemble-simple
Display assembler contents of executable sections in a simplified form

-D, --disassemble-all
Display assembler contents of all sections

-d, --disassemble
Display assembler contents of executable sections

--stop-address <ADDR>
Only process data whose address is <= ADDR

--start-address <ADDR>
Only process data whose address is >= ADDR

-o <output>, --output <output>
Path to output disassembled file.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

10.2.3 JTAG/Nexus Adapter

Copyright (C) 2017 Codasip Ltd. Codasip jtag/debugger with GNU gdb (GDB) 7.4.1 interface

10.2.3.1 Usage

```
 [--no-cache-region <noncachable regions description>] ... [-
v] [-h] [--checks] [--no-cache] [-c <config>] [--plugin
<plugin>] [--load-stack-heap] [-q] [-V] ... [--error <log>] [-
-warning <log>] [--info <log>] [--no-load] [--args <args>] [--
component <component> ... [--args <args>]] [-a <asip> ... [--
error <log>] [--warning <log>] [--info <log>] [--no-load] [--
args <args>] <exe>] [--timeout <timeout>] [--verify-
executables] [--batch] [-x <file>] [--dump-info] [-p <port>]
[-i <gdb|mi|mi2>] [--] <exe> <plugin/options> ...
```

10.2.3.2 Descriptions

--no-cache-region <noncachable regions description> (accepted multiple times)
Description of noncacheable regions in memory in format "<interface_path>:<low_address>:<high_address>".Addresses can be in hexadecimal, octal or decimal format. Example: "urisc.if_ldst:0x10:2048"

-v, --version

Displays version information and exits.

-h, --help
Displays usage information and exits. Display help of all plugins as well.

--checks
Additional checks of JTAG resources. Will catch some errors but with performance cost.

--no-cache
Disable software caching of registers and memory when ASIP is suspended.

-c <config>, --config <config>
XML configuration file with ASIP/platform description

--plugin <plugin>
Name of plugin for communication with JTAG device. Either built-in "impact" plugin can be used or plugin loaded from shared library. The shared library can be specified by a path to the library it self or by name. In case of a name the tool will try to locate the library file automatically.

--load-stack-heap
Load .stack and .heap sections into simulator memory (by default these sections are skipped)

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--error <log>
Sets enabled types for codasip_error().

--warning <log>
Sets enabled types for codasip_warning().

--info <log>
Sets enabled verbosity levels for codasip_info().

--no-load
Do not load executable into simulator, only load debug info.

--args <args>

Arguments for simulated application. For multiple arguments use quotes. If not used after -a <asip>, arguments are used for all ASIPs in simulator.

--component <component> (accepted multiple times)
Following arguments modify settings for given component.

--args <args>
Arguments for simulated application. For multiple arguments use quotes. If not used after -a <asip>, arguments are used for all ASIPs in simulator.

-a <asip>, --asip <asip> (accepted multiple times)
Following arguments modify settings for given ASIP instance.

--error <log>
Sets enabled types for codasip_error().

--warning <log>
Sets enabled types for codasip_warning().

--info <log>
Sets enabled verbosity levels for codasip_info().

--no-load
Do not load executable into simulator, only load debug info.

--args <args>
Arguments for simulated application. For multiple arguments use quotes. If not used after -a <asip>, arguments are used for all ASIPs in simulator.

<exe>
Input executable application. If not used after -a <asip>, application is loaded into all ASIPs in simulator.

--timeout <timeout>
Maximum number of clock cycle. When this number is reached, the simulation is stopped.

--verify-executables
Verify if all executables were loaded successfully into simulator (if they are suppose to)

--batch
Run debugger in batch mode, execute batch file and exit.

-x <file>, --command <file>

File containing GDB commands to be executed. Executed after binaries set through arguments are loaded. When error occurs during execution of the file, it is stopped. If debugger is run in batch mode, debugger exits after execution of all commands or error.

--dump-info

Dump information about the simulator in XML format.

-p <port>, --port <port>

Simulator's port (remote debugging is enabled).

-i <gdb|mi|mi2>, --interpreter <gdb|mi|mi2>

Select interpreter

--, --ignore_rest

Ignores the rest of the labeled arguments following this flag.

<exe>

Input executable application. If not used after -a <asip>, application is loaded into all ASIPs in simulator.

<plugin/options> (accepted multiple times)

Path to shared libraries (.dll) implementing disassembler or JTAG communication plugin. If argument starts with -- or -, it will be used as option for selected JTAG communication plugin.

10.2.4 Profiler

Copyright (C) 2017 Codasip Ltd. Create and prints statistics from a file dump produced by a simulator.

10.2.4.1 Usage

```
 --executable <xexe> ... [--dump-info] [--ide] [--address-space-histogram-start-end <ashse>] ... [--address-space-histogram-bins <ashb>] ... [--annotate-asm] [--annotate-src] [--merge] [--ppa] [--sequences] [--codal-coverage] [--resources-coverage] [--decoders-coverage] [--call-stack] [--source-code-coverage] [-a] [--ppa-weights-save <>] [--ppa-weights-load <>] [--output-annotate <output-annotate>] [--search-dir <>] [-f <format>] [-o <output>] [-q] [-V] ... [--] [--version] [-h] <> ...
```

10.2.4.2 Descriptions

--executable <xexe> (accepted multiple times)
(required) Executables used during simulation (.xexe)

--dump-info
Dump info about profiler

--ide
Generate Eclipse file for in-editor annotations

--address-space-histogram-start-end <ashse> (accepted multiple times)
Start and stop addresses separated by a comma for an address space histograms (e.g. 0x100,0x200). (Start address is optional.)Optionally, a particular address space histograms can be selected (e.g. as_ data:0x100,0x200). Default is all addresses for all address spaces.

--address-space-histogram-bins <ashb> (accepted multiple times)
Number of bins of any address space histogram. Optionally, a particular address space histograms can be selected (e.g. as_data:64). Default is 32 for all address spaces.

--annotate-asm
Annotate assembly

--annotate-src
Annotate source code files

--merge
Merge dump files

--ppa
Enable PPA analysis

--sequences
Enable decoded sequences

--codal-coverage
Enable CodAL coverage

--resources-coverage
Enable resources coverage

--decoders-coverage
Enable decoders coverage

--call-stack

Enable source code coverage

--source-code-coverage
Enable source code coverage

-a, --all
Enable all options

--ppa-weights-save <>
PPA configuration output file

--ppa-weights-load <>
Path to file with PPA weights.

--output-annotate <output-annotate>
Annotated output source code files directory. The value is append to --output argument

--search-dir <>
Source code files directory

-f <format>, --format <format>
Format of the output [html, text, oprof, cprof]

-o <output>, --output <output>
Path to output directory or file.

-q, --quiet
Suppresses most of output of tool.

-V, --verbose (accepted multiple times)
Increases verbosity of output.

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

<> (accepted multiple times)
Path of input dump file.

10.2.5 Simulator

Copyright (C) 2017 Codasip Ltd. Codasip simulator/debugger with GNU gdb (GDB) 7.4.1 interface

10.2.5.1 Usage

```
 [--error <log>] [--warning <log>] [--info <log>] [--no-load]
[--args <args>] [--component <component> ... [--args <args>]]
[-a <asip> ... [--error <log>] [--warning <log>] [--info
<log>] [--no-load] [--args <args>] <exe>] [--timeout
<timeout>] [--verify-executables] [--batch] [-x <file>] [--
dump-info] [-l <resource> ... ] [-d <resource> ... [--vcd] [-f
<file>]] [--dump-clock-cycles <file>] [--profiler-call-stack]
[--profiler-sequences <sequences>] [--profiler-sampling-rate
<sampling-rate>] [--profiler-output <file>] [-p <port>] [-r]
[-i <gdb|mi|mi2>] [--] [--version] [-h] <exe>
```

10.2.5.2 Descriptions

--error <log>
Sets enabled types for codasip_error().

--warning <log>
Sets enabled types for codasip_warning().

--info <log>
Sets enabled verbosity levels for codasip_info().

--no-load
Do not load executable into simulator, only load debug info.

--args <args>
Arguments for simulated application. For multiple arguments use quotes. If not used after -a <asip>, arguments are used for all ASIPs in simulator.

--component <component> (accepted multiple times)
Following arguments modify settings for given component.

--args <args>
Arguments for simulated application. For multiple arguments use quotes. If not used after -a <asip>, arguments are used for all ASIPs in simulator.

-a <asip>, --asip <asip> (accepted multiple times)

Following arguments modify settings for given ASIP instance.

--error <log>
Sets enabled types for codasip_error().

--warning <log>
Sets enabled types for codasip_warning().

--info <log>
Sets enabled verbosity levels for codasip_info().

--no-load
Do not load executable into simulator, only load debug info.

--args <args>
Arguments for simulated application. For multiple arguments use quotes. If not used after -a <asip>, arguments are used for all ASIPs in simulator.

<exe>
Input executable application. If not used after -a <asip>, application is loaded into all ASIPs in simulator.

--timeout <timeout>
Maximum number of clock cycle. When this number is reached, the simulation is stopped.

--verify-executables
Verify if all executables were loaded successfully into simulator (if they are suppose to)

--batch
Run debugger in batch mode, execute batch file and exit.

-x <file>, --command <file>
File containing GDB commands to be executed. Executed after binaries set through arguments are loaded. When error occurs during execution of the file, it is stopped. If debugger is run in batch mode, debugger exits after execution of all commands or error.

--dump-info
Dump information about the simulator in XML format.

-l <resource>, --load-dump <resource> (accepted multiple times)
Load dumped changes into resource.

-d <resource>, --dump <resource> (accepted multiple times)
Dump writes into resource.

--vcd
Use Value Change Dump (VCD) format. By default Codasip Dump Format (CDF) is used.

-f <file>, --file <file>
File resource changes will be dumped into. If not set, '<resource name>.cdf' will be used.

--dump-clock-cycles <file>
Name of the output file.

--profiler-call-stack
Enable call stack tracking.

--profiler-sequences <sequences>
Enable sequence tracking. Sequence lenghts are separated by a comma (e.g. 3,5)

--profiler-sampling-rate <sampling-rate>
Defines how often a profiler samples data.

--profiler-output <file>
Name of the output file.

-p <port>, --port <port>
Simulator's port (remote debugging is enabled).

-r, --run-only
Run simulation without debugger.

-i <gdb|mi|mi2>, --interpreter <gdb|mi|mi2>
Select interpreter

--, --ignore_rest
Ignores the rest of the labeled arguments following this flag.

--version
Displays version information and exits.

-h, --help
Displays usage information and exits.

<exe>

Input executable application. If not used after -a <asip>, application is loaded into all ASIPs in simulator.

11 BUILTIN FUNCTIONS

Builtins are functions that can be included in CodAL descriptions in order to guide one or more of the tools using that description. They affect assembler, disassembler, simulator and compiler generation in ways appropriate to each of the tools.

This chapter is organized as follows:

11.1 Semantic Categories

All Codasip builtin functions are tagged with the semantic category for the concrete tool. (e.g. compiler, simulator, general). This category serves as a filter for this tool and all functions in the given category are taken into account only within this tool. This filter is equivalent to using `#pragma` preprocessor directive. (`#pragma compiler`, `#pragma simulator`). We distinguish three categories:

1. general – the functions that are used in every tool. (In some cases the tool itself ignores the function)
2. simulator – the functions that are used only in simulator. (This corresponds with wrapping the function with `#pragma simulator`)

3. compiler – the functions that are used only in compiler. (This corresponds with wrapping the function with `#pragma compiler`)

The latter pragmas are described and illustrated in the *Codasip Studio User Guide* and in the *CodAL Language Reference Manual*.

11.2 Limitations of Codasip builtin functions

There are some limitations when using Codasip builtin functions to take into account.

11.2.1 Model type limitations

All the functions can be used in the IA model, but only functions in the list below can be used in the CA model.

- `codasip_print`, `codasip_error`, `codasip_warning`, `codasip_info`, `codasip_fatal`, `codasip_assert`, `codasip_halt`.

11.2.2 Block Placement Limitations

Codasip builtin functions can be placed only in the **`semantics`** section.

11.3 Debug Functions

11.3.1 codasip_assert

Semantic category - general

Supported on - IA, CA

```
void codasip_assert(bool condition, text format, ...)
```

If the argument `condition` is evaluated as zero (i.e., the `condition` is false), a formatted text message and automatic newline is written to the standard error output and simulation is terminated. If CodAL debugger is enabled, the simulation is suspended in the debugger before it is terminated.

11.3.1.1 Parameters

condition

Condition to be evaluated. If this condition evaluates to 0, the assertion fails and the message is written to the standard error output.

text

The message to be written when an assertion failure occurs. The message may contain format strings such as '%s' and '%d', as used in codasip_print(), etc.

11.3.1.2 Return Value

None.

11.3.1.3 Example

Example 44: codasip_assert function

```
semantics
{
    ...
    codasip_assert((r_pc % INSTR_LAU_SIZE) == 0, "PC is unaligned, PC = %d\n", r_pc);
    ...
};
```

11.3.2 codasip_disassembler

Semantic category - simulator

Supported on - IA, CA

`void codasip_disassembler(uint32 verbosity, uint1024 input, uint64 address, text isa)`

Disassemble input data and print the decoded instruction to the output. The print is done in `codasip_print()` way, so no prefix is printed nor the end of the line. You may use it for a pipeline debugging as well (meaning that you can call `codasip_disassembler()` in each pipeline stage to see which instruction if there).

11.3.2.1 Parameters

- input - Input data. The data should be aligned to MSB.
- address (optional) - Address of the decoded instruction (default: 0). It is needed for the correct computation when the decoded instruction uses **`current_address`** within attribute's **`decoding`** section.
- start (optional) - Name of the **`start`** section (default: "default").

11.3.2.2 Return Value

No return value.

11.3.2.3 Example

Example 45: codasip_dissasembler function

```
codasip_print("ID: ");
codasip_disassembler(r_ir, r_pc);
codasip_print("\n");
```

11.3.3 codasip_error

Semantic category - general

Supported on - IA, CA

`void codasip_error(uint32 verbosity, text format, ...)`

Write formatted data to the simulator output. Suitable for error messages that won't stop simulation.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

11.3.3.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

`%[flags][width][.precision]specifier`

Table 10: Codasip format flags

flags	**Description**
-	Left-justify when specifier is padded to given width. Right justification is the default.
#	Show numeric base prefix - 0b for binary, 0 for octal, 0x for hexadecimal. Valid only for numeric specifiers.
0	Use character '0' for padding to given width. By default padding is performed by space.

The `width` specifies minimum number of characters to be printed for the specifier. If the formatted value including optional base prefix is shorter, it is padded according to `flags` into the minimum width. If the formatted value is longer, no change is performed.

The `precision` specifies minimum number of digits to be printed after the decimal point. This setting is used for floating point specifier only.

Table 11: Codasip format specifiers

`specifier`	Description	Example
d	Decimal integer	123
u	Unsigned decimal integer (deprecated). Use unsigned integer as argument	123
b	Binary integer	1111011
o	Octal integer	173
x	Hexadecimal integer	7b
X	Hexadecimal integer, uppercase	7B
f	Floating point integer	12.3
c	Character	a
s	String	test

No `length` specifier (e.g. `h`, `ll`) is used as in ANSI C `printf`. The type of the variable to be printed is determined from argument of the built-in function after the format string.

Example 46: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

11.3.3.2 Parameters

<u>verbosity</u>

Specifies the message group for simulator output. Must be an unsigned integer value.

`Note: This behaviour is not currently implemented in the simulator.`

<u>format</u>

A C string containing the text to be written to the output.

The output of `codasip_error` function is in the format:

`error (<verbosity>): <ASIP name>@<clock cycle>: <formatted text>\n`

11.3.3.3 Return Value

None.

11.3.3.4 Example

Example 47: codasip_error function

```
#define ERROR_LEVEL      5

codasip_error(ERROR_LEVEL, "Unknown operation in i_cache_hw!");
```

Example 48: codasip_error function output

```
error(5): codasip_urisc@2500: Unknown operation in i_cache_hw!
```

11.3.4 codasip_fatal

Semantic category - general

Supported on - IA, CA

`void codasip_fatal(uint32 exit_code, text format, ...)`

Write formatted data to the simulator output and terminate the simulation.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

11.3.4.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

`%[flags][width][.precision]specifier`

Table 12: Codasip format flags

flags	**Description**

-	Left-justify when specifier is padded to given width. Right justification is the default.
#	Show numeric base prefix - 0b for binary, 0 for octal, 0x for hexadecimal. Valid only for numeric specifiers.
0	Use character '0' for padding to given width. By default padding is performed by space.

The `width` specifies minimum number of characters to be printed for the specifier. If the formatted value including optional base prefix is shorter, it is padded according to `flags` into the minimum width. If the formatted value is longer, no change is performed.

The `precision` specifies minimum number of digits to be printed after the decimal point. This setting is used for floating point specifier only.

Table 13: Codasip format specifiers

specifier	**Description**	**Example**
d	Decimal integer	123
u	Unsigned decimal integer (deprecated). Use unsigned integer as argument	123
b	Binary integer	1111011
o	Octal integer	173
x	Hexadecimal integer	7b
X	Hexadecimal integer, uppercase	7B
f	Floating point integer	12.3
c	Character	a
s	String	test

No `length` specifier (e.g. `h`, `ll`) is used as in ANSI C `printf`. The type of the variable to be printed is determined from argument of the built-in function after the format string.

Example 49: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

11.3.4.2 Parameters

<u>exit_code</u>

Exit code of the terminated simulation. Must be greater than 16. (0-16 are reserved) format

A C string containing the text to be written to the output.

The output of `codasip_fatal` function is in the format:

```
fatal (<exit_ code>):  <ASIP  name>@<clock  cycle>:  <formatted
text>\n
```

11.3.4.3 Return Value

None.

11.3.4.4 Example

Example 50: codasip_info function in i_halt (codasip_urisc)

```
codasip_fatal(21, "Unsupported syscall code %d.", code);
```

Example 51: codasip_info function in i_halt output(codasip_urisc)

```
fatal(21): codasip_urisc@666: Unsupported syscall 6.
```

11.3.5 codasip_get_clock_cycle

Semantic category - simulator

Supported on - IA

```
uint64 codasip_get_clock_cycle()
```

Returns the value of the cycle counter register.

11.3.5.1 Parameters

None.

11.3.5.2 Return Value

Value of the cycle counter register.

11.3.5.3 Example

Example 52: codasip_get_clock_cycle function

```
semantics
{
    uint64 cycles;

    cycles = codasip_get_clock_cycle();
    ...
};
```

11.3.6 codasip_inc_clock_cycle

Semantic category - simulator

Supported on - IA

`void codasip_inc_clock_cycle(uint64 value)`

Can be useful for more precise cycle counting in the IA simulation. In the default settings every instruction takes one cycle.

11.3.6.1 Parameters

value

The number of cycles added to the current cycle counter.

11.3.6.2 Return Value

None.

11.3.6.3 Example

Example 53: codasip_inc_clock_cycle function

```
semantics
{
    ...
    codasip_inc_clock_cycle(2); // two cycles are added to the cycle counter
    ...
};
```

11.3.7 codasip_info

Semantic category - general

Supported on - IA, CA

`void codasip_info(uint32 verbosity, text format, ...)`

Write formatted data to the simulator output. Suitable for the information messages.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

11.3.7.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

```
%[flags][width][.precision]specifier
```

Table 14: Codasip format flags

`flags`	Description
-	Left-justify when specifier is padded to given width. Right justification is the default.
#	Show numeric base prefix - 0b for binary, 0 for octal, 0x for hexadecimal. Valid only for numeric specifiers.
0	Use character '0' for padding to given width. By default padding is performed by space.

The `width` specifies minimum number of characters to be printed for the specifier. If the formatted value including optional base prefix is shorter, it is padded according to `flags` into the minimum width. If the formatted value is longer, no change is performed.

The `precision` specifies minimum number of digits to be printed after the decimal point. This setting is used for floating point specifier only.

Table 15: Codasip format specifiers

`specifier`	Description	Example
d	Decimal integer	123
u	Unsigned decimal integer (deprecated). Use unsigned integer as argument	123
b	Binary integer	1111011
o	Octal integer	173
x	Hexadecimal integer	7b
X	Hexadecimal integer, uppercase	7B
f	Floating point integer	12.3
c	Character	a
s	String	test

No `length` specifier (e.g. `h`, `ll`) is used as in ANSI C `printf`. The type of the variable to be printed is determined from argument of the built-in function after the format string.

Example 54: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

11.3.7.2 Parameters

<u>verbosity</u>

Specifies the message group for simulator output. Must be an unsigned integer value.

`Note: This behaviour is not currently implemented in the simulator.`

<u>format</u>

A C string containing the text to be written to the output.

The output of the `codasip_info` function is in the format:

```
info (<verbosity>):  <ASIP  name>@<clock  cycle>:  <formated
text>\n
```

11.3.7.3 Return Value

None.

11.3.7.4 Example

Example 55: codasip_info function in i_halt (codasip_urisc)

```
#define INFO_LEVEL       0

codasip_info(INFO_LEVEL,
             "Return value = %d\n",
             rf_gpr[REG_RETVAL] & EXIT_MASK);
```

Example 56: codasip_info function in i_halt output(codasip_urisc)

```
info(0): codasip_urisc@2300: Return value = 128
```

11.3.8 codasip_intended_fallthrough

Semantic category - general

Supported on - IA, CA

```
void codasip_intended_fallthrough()
```

This function suppresses warnings about missing **break** in a **switch case**. It should be used in the same place where **break** would be placed: as the last statement in a **case** or **default** inside a **switch**.

11.3.8.1 Parameters

This function has no parameters.

11.3.8.2 Return Value

This function has no return value.

11.3.8.3 Example

Example 57: codasip_ceil function

```
semantics
{
    int x;
    x = 1;
    switch (condition)
    {
        case 1:
            x = 3;
            // will get a warning here: missing break
        case 2:
            // doing nothing here, no warning
        case 3:
            x += 1;
            codasip_intended_fallthrough();
            // no warning here, fallthrough is intentional
        ...
};
```

11.3.9 codasip_print

Semantic category - general

Supported on - IA, CA

`void codasip_print(uint32 verbosity, text format, ...)`

Write formatted data to the simulator output.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers.

11.3.9.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

```
%[flags][width][.precision]specifier
```

Table 16: Codasip format flags

`flags`	**Description**
-	Left-justify when specifier is padded to given width. Right justification is the default.
#	Show numeric base prefix - 0b for binary, 0 for octal, 0x for hexadecimal. Valid only for numeric specifiers.
0	Use character '0' for padding to given width. By default padding is performed by space.

The `width` specifies minimum number of characters to be printed for the specifier. If the formatted value including optional base prefix is shorter, it is padded according to `flags` into the minimum width. If the formatted value is longer, no change is performed.

The `precision` specifies minimum number of digits to be printed after the decimal point. This setting is used for floating point specifier only.

Table 17: Codasip format specifiers

`specifier`	**Description**	**Example**
d	Decimal integer	123
u	Unsigned decimal integer (deprecated). Use unsigned integer as argument	123
b	Binary integer	1111011
o	Octal integer	173
x	Hexadecimal integer	7b
X	Hexadecimal integer, uppercase	7B
f	Floating point integer	12.3
c	Character	a
s	String	test

No `length` specifier (e.g. `h`, `ll`) is used as in ANSI C `printf`. The type of the variable to be printed is determined from argument of the built-in function after the format string.

Example 58: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

11.3.9.2 Parameters

<u>format</u>

A C string that containing the text to be written to the output.

11.3.9.3 Return Value

None.

11.3.9.4 Example

Example 59: codasip_print function

```
codasip_print("pipeline EX clear\n");
```

Example 60: codasip_print function output

```
pipeline EX clear
```

11.3.10 codasip_store_exit_code

Semantic category - simulator

Supported on - IA

`void codasip_store_exit_code(int32 code)`

A call to `codasip_store_exit_code` can be added to the **`semantics`** section of the `HALT` instruction for automatic testing with the instruction accurate simulator. The function creates a file, `sim_exit_code`, containing the return value of the simulated program.

11.3.10.1 Parameters

<u>code</u>

Exit code to be written into file.

11.3.10.2 Return Value

None.

11.3.10.3 Example

Example 61: codasip_store_exit_code function in the i_halt of the codasip_urisc model

```
semantics
{
    //codasip_halt is called to terminate the simulation
    codasip_halt();
    codasip_compiler_unused();
    //exit code is stored in the sim_exit_code file
    codasip_store_exit_code(rf_gpr[RF_RET_REG] & MASK_EXIT);
};
```

11.3.11 codasip_syscall

Semantic category - general

Supported on - IA

`int32 codasip_syscall(uint64 arguments_address)`

`codasip_ syscall` calls the host operating system function. This function is recognized based on the content of a special structure. The address of this structure is stored in a reserved register and the content of the register is passed to `codasip_ syscall` as a parameter.

The precise description is available in the chapter "Porting Syscalls" in the *Codasip Compiler Generation Tutorial*.

11.3.11.1 Parameters

<u>arguments_address</u>

This argument points to a reserved register which holds the address of the structure with parameters for use by the host operating system syscall.

11.3.11.2 Return Value

It returns zero if the syscall is supported, otherwise, non-zero value is returned. Example of such a syscall is a file removal.

11.3.11.3 Example

Example 62: codasip_syscall function in i_syscall element

```
element i_syscall
{
    assembler { "SYSCALL" };
    binary { OPC_SYSCALL:bit[OPC_W] UNUSED:bit[REMAINING_W(OPC_W)] };
    semantics
    {
        int rc;

        codasip_compiler_unused();
        // Using a reserved register to pass the address of a structure that
        // contains information about the requested syscall.
        rc = codasip_syscall(rf_gpr[30]);
        if (rc != 0)
        {
             codasip_fatal(25, "syscall %d is not supported", rf_gpr[30]);
        }
    };
};
```

11.3.12 codasip_warning

Semantic category - general

Supported on - IA, CA

`void codasip_warning(uint32 verbosity, text format, ...)`

Writes formatted data to the simulator output, suitable for warning messages.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

11.3.12.1 Parameters

verbosity

Specifies the message group for simulator output. Must be an unsigned integer value.

Note: This behaviour is not currently implemented in the simulator.

format

A C string containing the text to be written to the output.

The output of `codasip_warning` is in the format:

`warning (<verbosity>): <ASIP name>@<clock cycle>: <formated text>\n`

11.3.12.2 Return Value

None.

11.3.12.3 Example

Example 63: codasip_warning function

```
#define WARNING_LEVEL       2
```

```
codasip_warning(WARNING_LEVEL, "Warning: Default case.");;
```

Example 64: codasip_warning function output

```
warning(2): codasip_urisc@2500: Warning: Default case.
```

11.4 Timing Functions

11.4.1 codasip_halt

Semantic category - general

Supported on - IA, CA

`void codasip_halt()`

`codasip_halt` is a built-in function that is used to terminate simulation. The function can be called within any **`element`** or **`event`**. In Example 62, when the `HALT` instruction is decoded, the simulation is terminated.

11.4.1.1 Parameters

None.

11.4.1.2 Return Value

None.

11.4.1.3 Example

Example 65: codasip_halt function in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_halt();
    ...
};
```

11.5 Compiler Functions

11.5.1 codasip_compiler_builtin

Semantic category - compiler

Supported on - IA

`void codasip_compiler_builtin()`

This function instructs the compiler generator to generate a builtin function for this instruction. After compiler generation, the file `inlines.h` is generated in the CodAL project's directory `work/<asip_name>/ia/compiler/compiler`. This file contains the builtin functions. The compiler generator may remove some complex instructions that cannot be used automatically. This also disables them as builtins. You can use the annotation `codasip_compiler_unused()` together with `codasip_compiler_builtin()` to force builtin generation.

11.5.1.1 Parameters

None

11.5.1.2 Return Value

None.

11.5.1.3 Example

Example 66: codasip_compiler_builtin function

```
semantics
{
    ...
    codasip_compiler_builtin();
    codasip_compiler_unused();
    ...
};
```

11.5.2 codasip_compiler_flag_cmp

Semantic category - compiler

Supported on - IA

`void codasip_compiler_flag_cmp_<type1>(<type1> a, <type1> b)`

`codasip_compiler_flag_cmp` is an auxiliary mark for compiler generator that says that this instruction can be used as a compare instruction and that this instruction generates flags.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`
- `int8`

- `int16`
- `int32`
- `int64`

11.5.2.1 Parameters

a, b

Operands to compare.

11.5.2.2 Return Value

None.

11.5.2.3 Example

Example 67: codasip_info function in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_compiler_flag_int8(src1, src2);
    ...
};
```

11.5.3 codasip_compiler_hw_loop

Semantic category - compiler

Supported on - IA

`void codasip_compiler_hw_loop(uint32 loop_count, uint32 loop_end)`

This function is used in an instruction element to create a hardware loop instruction.

11.5.3.1 Parameters

loop_count

Number of iterations (loops) to be executed.

loop_end

End address of the loop (see Example 69).

11.5.3.2 Return Value

None.

11.5.3.3 Example

Example 68: codasip_compiler_hw_loop function used in i_hw_loop element (HWLOOP instruction)

```
element i_hw_loop
{
    ...
    assembler { "HWLOOP" loop_count "," loop_size};
    ...
    semantics
    {
        ...
        codasip_compiler_hw_loop(loop_count, loop_size);
        ...
    };
};
```

Example 69: use of the instruction with codasip_compiler_hw_loop function

```
    ...
    MOV R1, R0
    ...
    HWLOOP 10, LABEL    //Start of the hardware loop (the loop is executed 10 times)
    ADD R3, R4, R3
    ...
    NOP
LABEL:                  //address of the end of the hardware loop
    ...
```

11.5.4 codasip_compiler_predicate_false

Semantic category - compiler

Supported on - IA

`void codasip_compiler_predicate_false_<type1>(<type1> predop)`

`void codasip_compiler_predicate_false(uint1 predop)`

If the predicate is zero, the instruction will be executed normally, else `NOP` replaces it. This builtin is used mostly on VLIW architectures, where `NOP` costs less than a jump instruction.

11.5.4.1 Parameters

predop

The predicate.

11.5.4.2 Return Value

None.

11.5.4.3 Example

Example 70: codasip_compiler_predicate_false function

```
semantics
{
    ...
    codasip_compiler_predicate_false(predop); // Instruction will be executed only if predop
                                              // is 0
    ...
};
```

11.5.5 codasip_compiler_predicate_true

Semantic category - compiler

Supported on - IA

`void codasip_compiler_predicate_true_<type1>(<type1> predop)`

`void codasip_compiler_predicate_true(uint1 predop)`

If the predicate is one, the instruction will be executed normally, else `NOP` replaces it. This builtin is used mostly on VLIW architectures, where `NOP` costs less than a jump instruction.

11.5.5.1 Parameters

<u>predop</u>

The predicate.

11.5.5.2 Return Value

None.

11.5.5.3 Example

Example 71: codasip_compiler_predicate_true function

```
semantics
{
    ...
    codasip_compiler_predicate_true(predop); // Instruction will be executed only in case predop
                                             // is 1
    ...
};
```

11.5.6 codasip_compiler_priority

Semantic category - compiler

Supported on - IA

`void codasip_compiler_priority(int32 priority)`

Specifies the priority of an instruction for the compiler instruction selector.

11.5.6.1 Parameters

priority

Sets the instruction priority for instruction selection. The default value is 0, and it can be set to higher values.

If negative, the instruction will not be used during instruction selection.

11.5.6.2 Return Value

None.

11.5.6.3 Example

Example 72: codasip_compiler_priority function

```
semantics
{
    ...
    codasip_compiler_priority(5);
    ...
};
```

11.5.6.4 References

- http://llvm.org/devmtg/2008-08/Gohman_CodeGenAndSelectionDAGs.pdf
- http://llvm.org/devmtg/2008-08/Gohman_CodeGenAndSelectionDAGs_Hi.m4v

11.5.7 codasip_compiler_schedule_class

Semantic category - compiler

Supported on - IA

`void codasip_compiler_schedule_class(schedule_class index)`

This function sets the schedule class property in the instruction semantics. See also chapter "Compiler Generator Guide", section "Instruction Scheduling" in the *Codasip Studio User Guide*.

11.5.7.1 Parameters

index

Schedule class to be set.

11.5.7.2 Return Value

None.

11.5.7.3 Example

Example 73: codasip_compiler_schedule_class function

```
/* schedule class for load instruction

schedule_class loads
{
    latency = 3;
};

*/

semantics
{
    ...
    codasip_compiler_schedule_class(loads);
    ...
};
```

11.5.8 codasip_compiler_undefined

Semantic category - compiler

Supported on - IA

`void codasip_compiler_undefined()`

Used to explicitly specify that the semantics of an instruction are not correct. The compiler will not use this instruction.

For example, some special instructions may read from a FIFO. Such instructions cannot be used by the compiler automatically. Such behavior cannot be captured in the instruction semantics format, therefore the code that describes the behavior has to be enclosed in `#pragma simulator {...}`. For the semantics extractor however, this instruction can behave as a no-operation instruction and can be used by other tools such a Random Assembler Programs as no operation.

11.5.8.1 Parameters

None.

11.5.8.2 Return Value

None.

11.5.8.3 Example

Example 74: codasip_compiler_undefined

```
semantics
{
    ...
    codasip_compiler_undefined();
    ...
};
```

11.5.9 codasip_compiler_unused

Semantic category - compiler

Supported on - IA

`void codasip_compiler_unused()`

Inhibits use of the instruction by the compiler (though Random Assembler Programs can still use it). Typical use of this function is in the instruction semantics of manually entered instructions. e.g. `HALT` and `SYSCALL`.

11.5.9.1 Parameters

None.

11.5.9.2 Return Value

None.

11.5.9.3 Example

Example 75: codasip_compiler_unused in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_compiler_unused();
    ...
};
```

11.5.10 codasip_extract_subreg

Semantic category - instructions

Supported on - IA

`void codasip_ extract_ subreg (int32 dst, int32 src, int32 subidx)`

This builtin function is not supported yet. You can contact support@codasip.com if you require more information about this function.

11.5.11 codasip_insert_subreg

Semantic category - instructions

Supported on - IA

```
void codasip_insert_subreg(int32 dst, int32 src, int32 subidx)
```

This builtin function is not supported yet. You can contact support@codasip.com if you require more information about this function.

11.5.12 codasip_nop

Semantic category - compiler

Supported on - IA

```
void codasip_nop()
```

Explicitly specifies a `NOP` instruction. This might be useful for architectures that do not handle data or structural hazards, or require that delay slots after jumps.

11.5.12.1 Parameters

None.

11.5.12.2 Return Value

None.

11.5.12.3 Example

Example 76: codasip_nop function in i_nop (codasip_urisc)

```
semantics
{
    codasip_nop();
};
```

11.5.13 codasip_preprocessor_define

Semantic category - compiler

Supported on - IA

```
void codasip_preprocessor_define(text define_name)
```

If an element that contains a call is directly or indirectly part of the ISA, the preprocessor will then define the given value in all programs compiled by the generated C/C++ compiler.

Having such constant defined is typically useful for checking that a certain instruction extension is enabled.

11.5.13.1 Parameters

<u>id</u>

The constant being defined.

11.5.13.2 Return Value

None.

11.5.13.3 Example

Example 77: codasip_preprocessor_define function

```
semantics
{
    ...
    codasip_preprocessor_define("ISE_BITCOUNT"); // If this instruction is part of ISA, constant ABC
                                                    will be defined in all compiled programs
    ...
};
```

Example 78: using the defined constant in a compiled program

```
...
#ifdef ISE_BITCOUNT
   res = bitcount_fast();
#else
   res = bitcount_slow();
#endif
...
```

11.5.14 codasip_undef

Semantic category - general

Supported on - IA

`<type1> codasip_undef_<type1>()`

The function can be used for semantics extractor. It is useful in cases when the compiler should ignore that a result is written into a register, but there should be information that a register is written.

`<type1>` might acquire these values:

- int16
- int32

11.5.14.1 Parameters

None.

11.5.14.2 Return Value

Returns -1.

11.5.14.3 Example

Example 79: codasip_undef function

```
semantics
{
    ...
    int16 undef;
    undef = codasip_undef_int16(); // Contains -1
    ...
};
```

11.6 Arithmetic Functions

11.6.1 codasip_borrow_sub

Semantic category - general

Supported on - IA

`uint1 codasip_borrow_sub_<type1>(<type1> a, <type1> b)`

Subtracts b from a and returns a borrow flag. The result of the subtraction is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.1.1 Parameters

a, b

Operands to subtract.

11.6.1.2 Return Value

1-bit borrow flag.

11.6.1.3 Example

Example 80: codasip_borrow_sub function

```
semantics
{
    uint1 borrow;
    ...
    borrow = codasip_borrow_sub_int32(reg_src1, reg_src2);
    ...
};
```

11.6.2 codasip_borrow_sub_c

Semantic category - general

Supported on - IA

`uint1 codasip_borrow_sub_c_<type1>(<type1> a, <type1> b, uint1 c)`

Subtracts b from a with carry and returns a borrow flag. The result of the subtraction is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.2.1 Parameters

a, b, c

Operands to subtract. `c` is always 1-bit.

11.6.2.2 Return Value

1-bit borrow flag.

11.6.2.3 Example

Example 81: codasip_borrow_sub_c function

```
semantics
{
   uint1 borrow;
   ...
   borrow = codasip_borrow_sub_c_int32(reg_src1, reg_src2, cf);
   ...
};
```

11.6.3 codasip_carry_add

Semantic category - general

Supported on - IA

`uint1 codasip_carry_add_<type1>(<type1> a, <type1> b)`

Adds a to b and returns a carry flag. The result of the addition is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.3.1 Parameters

a, b

Operands to add

11.6.3.2 Return Value

1-bit carry flag

11.6.3.3 Example

Example 82: codasip_carry_add function

```
semantics
{
    uint1 carry;
    ...
    carry = codasip_carry_add_int32(reg_src1, reg_src2);
    ...
};
```

11.6.4 codasip_carry_add_c

Semantic category - general

Supported on - IA

`uint1 codasip_carry_add_c_<type1>(<type1> a, <type1> b, uint1 c)`

Adds a to b with carry and returns a carry flag. The result of the addition is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.4.1 Parameters

a, b, c

Operands to add. `c` is always a 1-bit integer.

11.6.4.2 Return Value

1-bit carry flag

11.6.4.3 Example

Example 83: codasip_carry_add_c function

```
semantics
{
    uint1 carry;
    ...
    carry = codasip_carry_add_c_int32(reg_src1, reg_src2, carry_old);
    ...
};
```

11.6.5 codasip_ctlo

Semantic category - general

Supported on - IA

`<type1> codasip_ctlo_<type1>(<type1> src)`

Counts the leading (most significant) ones in src. For example, if src = -1, then the result is the size in bits of the type of src.

The compiler cannot use this operation automatically.

Allowed values of `<type1>` are:

- int16
- int32

11.6.5.1 Parameters

src

Operand of type `<type1>`, which will be used for counting the leading ones.

11.6.5.2 Return Value

Count of the most significant ones. Return data type is `<type1>`.

11.6.5.3 Example

Example 84: codasip_ctlo function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = -4; // 1111 1111 1111 1100
    res = codasip_ctlo_int16(src); // res = 14
};
```

11.6.6 codasip_ctlz

Semantic category - general

Supported on - IA

`<type1> codasip_ctlz_<type1>(<type1> src)`

Counts the leading (most significant) zeros in src. For example, if src = 0, then the result is the size in bits of the type of src.

The compiler can use this operation automatically. In C source code, it must be specified with the LLVM/gcc-supported builtin `__builtin_clz`.

Allowed values of `<type1>` are:

- int16
- int32

11.6.6.1 Parameters

src

Operand of type `<type1>`, which will be used for counting the leading zeros.

11.6.6.2 Return Value

Count of the most significant zeroes. Return data type is `<type1>`.

11.6.6.3 Example

Example 85: codasip_ctlz function

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = 2;
    res = codasip_ctlz_int32(src); // res = 30
};
```

11.6.7 codasip_ctpop

Semantic category - general

Supported on - IA

`<type1> codasip_ctpop_<type1>(<type1> src)`

This function counts the 1's in src.

The compiler can use this operation automatically. In C source code, it must be specified with the LLVM/gcc-supported builtin `__builtin_popcount`.

Allowed values of `<type1>` are:

- `int16`
- `int32`

11.6.7.1 Parameters

src

An operand of type `<type1>`.

11.6.7.2 Return Value

Count of 1's. The return data type is `<type1>`.

11.6.7.3 Example

Example 86: codasip_ctpop function)

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = 33; // Bits 0 and 5 are set
    res = codasip_ctpop_int32(src); // res = 2
};
```

11.6.8 codasip_ctto

Semantic category - general

Supported on - IA

`<type1> codasip_ctto_<type1>(<type1> src)`

Counts the trailing (least significant) ones in src. For example, if src = -1, then the result is the size in bits of the type of src.

The compiler cannot use this operation automatically.

Allowed values of `<type1>` are:

- int16
- int32

11.6.8.1 Parameters

src

An operand of type `<type1>`.

11.6.8.2 Return Value

Count of the least significant ones. The return data type is `<type1>`.

11.6.8.3 Example

Example 87: codasip_ctto function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = 7; // 0000 0000 0000 0111
    res = codasip_ctto_int16(src); // res = 3
};
```

11.6.9 codasip_cttz

Semantic category - general

Supported on - IA

`<type1> codasip_cttz_<type1>(<type1> src)`

Counts the trailing (least significant) zeros in src. For example, if src = 0, then the result is the size in bits of the type of src.

The compiler can use this operation automatically. In C source code, it must be specified with the LLVM/gcc-supported builtin `__builtin_ctz`.

Allowed values of `<type1>` are:

- int16
- int32

11.6.9.1 Parameters

src

An operand of type `<type1>`.

11.6.9.2 Return Value

Count of the least significant zeros. The return data type is `<type1>`.

11.6.9.3 Example

Example 88: codasip_cttz function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = 2; // 0000 0000 0000 0010
    res = codasip_cttz_int16(src); // res = 1
};
```

11.6.10 codasip_overflow_add

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_add_<type1>(<type1> a, <type1> b)`

Adds a to b and returns an overflow flag. The result of the addition is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`

- `int16`
- `int32`

11.6.10.1 Parameters

a, b

Operands to add.

11.6.10.2 Return Value

1-bit overflow flag.

11.6.10.3 Example

Example 89: codasip_carry_add function

```
semantics
{
    uint1 overflow;


    overflow = codasip_overflow_add_int32(reg_src1, reg_src2);
    ...
};
```

11.6.11 codasip_overflow_add_c

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_add_c_<type1>(<type1> a, <type1> b, uint1 c)`

Adds a to b with the carry flag and returns an overflow flag. The result of the addition is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.11.1 Parameters

a, b, c

Operands to add. c is 1-bit.

11.6.11.2 Return Value

1-bit overflow flag.

11.6.11.3 Example

Example 90: codasip_overflow_add_c function

```
semantics
{
    uint1 overflow;

    overflow = codasip_overflow_add_c_int32(reg_src1, reg_src2, carry);
    ...
};
```

11.6.12 codasip_overflow_sub

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_sub_<type1>(<type1> a, <type1> b)`

Subtracts b from a and returns an overflow flag. The result of the subtraction is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.12.1 Parameters

a, b

Operands to subtract.

11.6.12.2 Return Value

1-bit overflow flag.

11.6.12.3 Example

Example 91: codasip_overflow_sub function

```
semantics
{
    uint1 overflow;
    overflow = codasip_overflow_sub_int32(reg_src1, reg_src2);
    ...
};
```

11.6.13 codasip_overflow_sub_c

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_sub_c_<type1>(<type1> a, <type1> b, uint1 c)`

Subtracts b from a with carry and returns an overflow flag. The result of the subtraction is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

11.6.13.1 Parameters

a, b, c

Operands to subtract. c has 1-bit.

11.6.13.2 Return Value

1-bit overflow flag.

11.6.13.3 Example

Example 92: codasip_overflow_sub_c function

```
semantics
{
    uint1 overflow;

    overflow = codasip_overflow_add_c_int32(reg_src1, reg_src2, cf_old);
    ...
};
```

11.6.14 codasip_parity_odd

Semantic category - general

Supported on - IA

`uint1 codasip_parity_odd_<type1>(<type1> a)`

Returns 1 if the parity of the operand is even, else 0.

The computation used is (BIT_WIDTH is the bitwidth of `a`):

```
uint1 res = 1;
for (int i = 0; i < BIT_WIDTH; i++)
    res ^= (a >> i) & 1;
return res;
```

`codasip_parity_odd` is used only by the simulator. It is ignored by semantics extraction and not used for compiler generation.

11.6.14.1 Parameters

a

An operand with the following possible types:

- `int8`
- `int16`
- `int32`

11.6.14.2 Return Value

1-bit parity result.

11.6.14.3 Example

Example 93: codasip_parity_odd function

```
semantics
{
    int8 number;
    uint1 result;

    number = 30;  // 0001 1110
    result = codasip_parity_odd(number); // result = 1 => parity of 1 0001 1110 is odd.
    ...
};
```

11.7 Saturated Arithmetic Functions

11.7.1 codasip_usadd

Semantic category - general

Supported on - IA

`<type1> codasip_usadd_<type1>(<type1> a, <type1> b)`

The function performs an unsigned saturated addition. This means that the result of such an operation is limited to a range, defined by a minimum and a maximum . If the result of the operation is greater than the maximum, then it is set to the maximum; if it is below the minimum, then it is set to the minimum.

The minimum and maximum values are given by the size of `<type1>`.

11.7.1.1 Parameters

a, b

Operands to be added.

11.7.1.2 Return Value

Returns the addition of a and b. If the addition result generates the carry flag, the result is the maximum value for `<type1>`.

11.7.1.3 Example

Example 94: codasip_usadd function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 196;
    uint8 b = 100;
    uint8 res;

    // Addition generates the carry flag, so the 'res' will
    // contain the maximum 8-bit value, which is 255.
    res = codasip_usadd(a, b);
    ...
};
```

11.7.2 codasip_usadd_occured

Semantic category - general

Supported on - IA

`uint1 codasip_usadd_occured_<type1>(<type1> a, <type1> b)`

The functions performs an unsigned saturated addition and returns 1-bit information to say if saturation occurred or not. Saturation occurs when the addition sets the carry flag.

11.7.2.1 Parameters

a, b

Operands to be added.

11.7.2.2 Return Value

Returns 1 if the carry flag has been generated by the addition, else returns 0.

11.7.2.3 Example

Example 95: codasip_usadd_occured function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 196;
    uint8 b = 100;
    uint1 res;

    // Addition generates the carry flag, so the saturation
    // has occured. The 'res' will be set to 1.
    res = codasip_usadd_occured(a, b);
    ...
};
```

11.7.3 codasip_ussub

Semantic category - general

Supported on - IA

`<type1> codasip_ussub_<type1>(<type1> a, <type1> b)`

The functions performs an unsigned saturated subtraction. Basically it means that the result of such operation is limited to a fixed range between a minimum and maximum value. If the result of an operation is greater than maximum it is set to the maximum; if it is below the minimum it is set to minimum.

Minimum and maximum value is given by the size of `<type1>`.

11.7.3.1 Parameters

a, b

Operands to be subtracted.

11.7.3.2 Return Value

Returns subtraction of a and b. If the subtraction result generates the borrow flag, the result is the maximum value of `<type1>`.

11.7.3.3 Example

Example 96: codasip_ussub function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 100;
    uint8 b = 196;
    uint8 res;

    // Addition generates the borrow flag, so the 'res' will
    // contain the maximum 8-bit value, which is 255.
    res = codasip_ussub(a, b);
    ...
};
```

11.7.4 codasip_ussub_occured

Semantic category - general

Supported on - IA

`uint1 codasip_ussub_occured_<type1>(<type1> a, <type1> b)`

The functions performs an unsigned saturated subtraction and returns 1-bit information if saturation has occurred. Saturation occurs when the subtraction sets the borrow flag.

11.7.4.1 Parameters

a, b

Operands to be subtracted.

11.7.4.2 Return Value

Returns 1 if the borrow flag has been generated by the subtraction, else returns 0.

11.7.4.3 Example

Example 97: codasip_ussub_occured function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 100;
    uint8 b = 196;
    uint1 res;

    // Addition generates the borrow flag, so the saturation
    // has occured. The 'res' will be set to 1.
    res = codasip_ussub_occured(a, b);
    ...
};
```

11.8 Floating Point Functions

11.8.1 codasip_ceil

Semantic category - general

Supported on - IA

`<type1> codasip_ceil_<type1>(<type1> a)`

This function returns the lowest integer value greater than or equal to a.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

11.8.1.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

11.8.1.2 Return Value

The smallest integral value not smaller than a.

11.8.1.3 Example

Example 98: codasip_ceil function

```
semantics
{
    float32 a = 1.6f;
    float32 b;
    b = codasip_ceil_float32(a); //b = 2.0f;
    ...
};
```

11.8.2 codasip_cos

Semantic category - general

Supported on - IA

`<type1> codasip_cos_<type1>(<type1> a)`

This function returns the cosine of the operand.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.2.1 Parameters

a

The argument of `<type1>` representing an angle expressed in radians.

11.8.2.2 Return Value

Cosine of a radians.

11.8.2.3 Example

Example 99: codasip_sin function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.5708f;
    b = codasip_cos_float32(a); //b = 0.0f;
    ...
};
```

11.8.3 codasip_exp

Semantic category - general

Supported on - IA

`<type1> codasip_exp_<type1>(<type1> a)`

Returns the base-e exponential function of a, which is e raised to the power a: e^a.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.3.1 Parameters

a

Value of the exponent of `<type1>` .

11.8.3.2 Return Value

Exponential of a.

11.8.3.3 Example

Example 100: codasip_exp function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.0f;
    b = codasip_exp_float32(a); //b = 2.718281828f;
    ...
};
```

11.8.4 codasip_fabs

Semantic category - general

Supported on - IA

`<type1> codasip_fabs_<type1>(<type1> a)`

This function returns the absolute value of a: |a|.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

11.8.4.1 Parameters

a

The argument of `<type1>` whose absolute value is returned.

11.8.4.2 Return Value

The absolute value of a.

11.8.4.3 Example

Example 101: codasip_fabs function

```
semantics
{
    float32 a = -1.0f;
    float32 b;
    b = codasip_fabs_float32(a); //b = 1.0f;
    ...
};
```

11.8.5 codasip_fcmp_oeq

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_oeq_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.5.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.5.2 Return Value

1-bit result.

11.8.5.3 Example

Example 102: codasip_fcmp_oeq function

```
semantics
{
    uint1 feq;

    feq = codasip_fcmp_oeq_float32(src1, src2);
    ...
};
```

11.8.6 codasip_fcmp_oge

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_oge_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.6.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.6.2 Return Value

1-bit result.

11.8.6.3 Example

Example 103: codasip_fcmp_oge function

```
semantics
{
    uint1 fge;

    fge = codasip_fcmp_oge_float32(src1, src2);
    ...
};
```

11.8.7 codasip_fcmp_ogt

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ogt_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater than `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.7.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.7.2 Return Value

1-bit result.

11.8.7.3 Example

Example 104: codasip_fcmp_ogt function

```
semantics
{
    uint1 fgt;

    fgt = codasip_fcmp_ogt_float32(src1, src2);
    ...
};
```

11.8.8 codasip_fcmp_ole

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ole_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is less or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.8.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.8.2 Return Value

1-bit result.

11.8.8.3 Example

Example 105: codasip_fcmp_ole function

```
semantics
{
    uint1 fle;

    fle = codasip_fcmp_ole_float32(src1, src2);
    ...
};
```

11.8.9 codasip_fcmp_olt

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_olt_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is lesser than `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.9.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.9.2 Return Value

1-bit result.

11.8.9.3 Example

Example 106: codasip_fcmp_olt function

```
semantics
{
    uint1 feq;

    feq = codasip_fcmp_olt_float32(src1, src2);
    ...
};
```

11.8.10 codasip_fcmp_one

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_one_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is not equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.10.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.10.2 Return Value

1-bit result.

11.8.10.3 Example

Example 107: codasip_fcmp_one function

```
semantics
{
    uint1 fne;

    fne = codasip_fcmp_one_float32(src1, src2);
    ...
};
```

11.8.11 codasip_fcmp_ord

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ord_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. Yields 1 if both operands are not a NaN, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.11.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.11.2 Return Value

1-bit result.

11.8.11.3 Example

Example 108: codasip_fcmp_ord function

```
semantics
{
    uint1 ford;

    ford = codasip_fcmp_ord_float32(src1, src2);
    ...
};
```

11.8.12 codasip_fcmp_ueq

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ueq_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.12.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.12.2 Return Value

1-bit result.

11.8.12.3 Example

Example 109: codasip_fcmp_ueq function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_ueq_float32(src1, src2);
    ...
};
```

11.8.13 codasip_fcmp_uge

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_uge_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.13.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.13.2 Return Value

1-bit result.

11.8.13.3 Example

Example 110: codasip_fcmp_uge function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_uge_float32(src1, src2);
    ...
};
```

11.8.14 codasip_fcmp_ugt

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ugt_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater than to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.14.1 Parameters

a, b

Operands to compare. Both must have the same bit-width.

11.8.14.2 Return Value

1-bit result depending float on comparison.

11.8.14.3 Example

Example 111: codasip_fcmp_ugt function

```
semantics
{
    uint1 fugt;

    fugt = codasip_fcmp_ugt_float32(src1, src2);
    ...
};
```

11.8.15 codasip_fcmp_ule

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ule_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is less or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.15.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.15.2 Return Value

1-bit result.

11.8.15.3 Example

Example 112: codasip_fcmp_ule function

```
semantics
{
    uint1 fule;

    fule = codasip_fcmp_ule_float32(src1, src2);
    ...
};
```

11.8.16 codasip_fcmp_ult

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ult_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is lesser then `b`, thenit returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.16.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.16.2 Return Value

1-bit result.

11.8.16.3 Example

Example 113: codasip_fcmp_ult function

```
semantics
{
    uint1 fult;

    fult = codasip_fcmp_ult_float32(src1, src2);
    ...
};
```

11.8.17 codasip_fcmp_une

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_une_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is not equal to `b`, then it returns 1, else 0.

An unordered comparison is performed, which means that there is a check that either operand is `NaN`.

11.8.17.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.17.2 Return Value

1-bit result.

11.8.17.3 Example

Example 114: codasip_fcmp_une function

```
semantics
{
    uint1 fune;

    fune = codasip_fcmp_une_float32(src1, src2);
    ...
};
```

11.8.18 codasip_fcmp_uno

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_uno_<type1>(<type1> a, <type1> b)
```

Compares two float numbers of the same type. Yields 1 if either operand is a NaN, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

11.8.18.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

11.8.18.2 Return Value

1-bit result.

11.8.18.3 Example

Example 115: codasip_fcmp_uno function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_uno_float32(src1, src2);
    ...
};
```

11.8.19 codasip_floor

Semantic category - general

Supported on - IA

```
<type1> codasip_floor_<type1>(<type1> a)
```

Rounds a down, returning the largest integral value that is not greater than a.

11.8.19.1 Parameters

a

The allowed types for `a` are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.19.2 Return Value

The value of a rounded down.

11.8.19.3 Example

Example 116: codasip_floor function (positive value use)

```
semantics
{
    float32 a;
    float32 b;

    a = 2.9f;
    b = codasip_floor_float32(a); // b = 2.0f;
    ...
};
```

Example 117: codasip_floor function (negative value use)

```
semantics
{
    float32 a;
    float32 b;

    a = -2.9f;
    b = codasip_floor_float32(a); // b = -3.0f;
    ...
};
```

11.8.20 codasip_fma

Semantic category - general

Supported on - IA

`<type1> codasip_fma_<type1>(<type1> a, <type1> b, <type1> c)`

This functions computes a*b+c.

11.8.20.1 Parameters

a, b, c

The allowed types of the arguments are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.20.2 Return Value

The result of the computation.

11.8.20.3 Example

Example 118: codasip_fma function

```
semantics
{
    float32 a;
    float32 b;
    float32 c;
    float32 d;

    a = 2.0f;
    b = 3.0f;
    c = 3.0f;
    d = codasip_fma_float32(a,b,c); // d = 9.0f;
    ...
};
```

11.8.21 codasip_fpu_getround

Semantic category - general

Supported on - IA

`int32 codasip_fpu_getround()`

To retrieve the current rounding mode, call `int codasip_fpu_getround()`. See also `codasip_fpu_setround(int)`.

11.8.21.1 Parameters

None.

11.8.21.2 Return Value

Rounding modes have these values (one can define them as constants in the model):

- 0 - round to nearest even (translated to FE_TONEAREST)
- 3 - round toward zero (translated to FE_TOWARDZERO)
- 1 - round toward +infinity (translated to FE_UPWARD)
- 2 - round toward -infinity (translated to FE_DOWNWARD)

11.8.22 codasip_fpu_setround

Semantic category - general

Supported on - IA

`void codasip_fpu_setround(int32 mode)`

All the floating point operations described using C in CodAL use the simulator's host FPU. Its behavior is based on the current floating point rounding setting (fenv).

To set the host rounding mode from the CodAL model, function void `codasip_fpu_setround(int)` can be used. To retrieve the current rounding mode, call `int codasip_fpu_getround()`.

If the model uses FPU operations, it is suggested to add a call to`codasip_fpu_setround` to the event reset in order to initialize the FPU environment.

11.8.22.1 Parameters

`rounding_mode`

Rounding modes have these values (one can define them as constants in the model):

- 0 - round to nearest even (translated to FE_TONEAREST)
- 3 - round toward zero (translated to FE_TOWARDZERO)
- 1 - round toward +infinity (translated to FE_UPWARD)
- 2 - round toward -infinity (translated to FE_DOWNWARD)

11.8.22.2 Return Value

None.

11.8.22.3 Example

Example 119: Usage and constants definitions

```
#define CFE_TONEAREST 0
#define CFE_TOWARDZERO 3
#define CFE_UPWARD 1
#define CFE_DOWNWARD 2
...
```

```
codasip_fpu_setround(CFE_TONEAREST);
```

11.8.23 codasip_ftrunc

Semantic category - general

Supported on - IA

`<type1> codasip_ftrunc_<type1>(<type1> a)`

This function rounds a towards zero, returning the nearest integral value magnitude of which is not greater a .

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

11.8.23.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

11.8.23.2 Return Value

The nearest integral value that is not greater than a in magnitude.

11.8.23.3 Example

Example 120: codasip_ftrunc function

```
semantics
{
    float32 a = 2.3f;
    float32 b;
    b = codasip_ftrunc_float32(a); //b = 2.0f;
    ...
};
```

11.8.24 codasip_log

Semantic category - general

Supported on - IA

`<type1> codasip_log_<type1>(<type1> a)`

Returns the natural logarithm of a, i.e. the base-e logarithm, or the inverse of the natural exponential function (`codasip_exp`)

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.24.1 Parameters

a

Value of `<type1>` whose logarithm is calculated.

11.8.24.2 Return Value

Natural logarithm of a.

11.8.24.3 Example

Example 121: codasip_log function

```
semantics
{
    float32 a;
    float32 b;

    a = 2.718281828f;
    b = codasip_log_float32(a); //b = 1.0f;
    ...
};
```

11.8.25 codasip_pow

Semantic category - general

Supported on - IA

`<type1> codasip_pow_<type1>(<type1> a, <type1> b)`

Returns the base raised to the power of the exponent: a^b.

Allowed values of `<type1>` are:

- `float16`
- `float32`

- `float64`
- `float80`

11.8.25.1 Parameters

<u>a</u>

Base value of `<type1>`.

<u>b</u>

Exponent value of `<type1>`.

11.8.25.2 Return Value

The result of raising the base to the power of the exponent.

11.8.25.3 Example

Example 122: codasip_pow function

```
semantics
{
    float32 a;
    float32 b;
    float32 c;

    a = 2.0f;
    b = 3.0f;
    c = codasip_pow_float32(a,b); // c = 8.0f;
    ...
};
```

11.8.26 codasip_powi

Semantic category - general

Supported on - IA

`<type1> codasip_powi_<type1>(<type1> a, int32 b)`

Returns the base raised to the integer power of the exponent: $\underline{a}^{b}$.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.26.1 Parameters

a

Base value of `<type1>` .

b

Exponent value of type `int32`.

11.8.26.2 Return Value

The result of raising the base to the power of the exponent.

11.8.26.3 Example

Example 123: codasip_powi function

```
semantics
{
    float32 a;
    int32 b;
    float32 c;

    a = 2.0f;
    b = 3;
    c = codasip_powi_float32(a,b); // c = 8.0f;
    ...
};
```

11.8.27 codasip_rint

Semantic category - general

Supported on - IA

`<type1> codasip_rint_<type1>(<type1> a)`

This function rounds a to an integer value, using the round direction specified by the current rounding mode.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

11.8.27.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

11.8.27.2 Return Value

The nearest integral value to a according to the current rounding mode.

11.8.27.3 Example

Example 124: codasip_rint function

```
semantics
{
    float32 a = 2.3f;
    float32 b;
    b = codasip_rint_float32(a); //b = 2.0f, depends on the current rounding mode
    ...
};
```

11.8.28 codasip_round

Semantic category - general

Supported on - IA

`<type1> codasip_round_<type1>(<type1> a)`

Returns the integral value that is nearest to a, with halfway cases rounded away from zero.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

11.8.28.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

11.8.28.2 Return Value

The value of a rounded to the nearest integral.

11.8.28.3 Example

Example 125: codasip_round function

```
semantics
{
    float32 a = 3.8f;
    float32 b;
    b = codasip_round_float32(a); //b = 4.0f
    ...
};
```

11.8.29 codasip_sin

Semantic category - general

Supported on - IA

`<type1> codasip_sin_<type1>(<type1> a)`

This function returns the sine of the operand.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

11.8.29.1 Parameters

a

The argument of `<type1>` representing an angle expressed in radians.

11.8.29.2 Return Value

Sine of a.

11.8.29.3 Example

Example 126: codasip_sin function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.5708f;
    b = codasip_sin_float32(a); //b = 1.0f;
    ...
};
```

11.8.30 codasip_sqrt

Semantic category - general

Supported on - IA

`<type1> codasip_sqrt_<type1>(<type1> a)`

This function returns the sqrt of the specified operand if it is a nonnegative floating point number.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

11.8.30.1 Parameters

a

The argument of `<type1>` whose square root is computed.

11.8.30.2 Return Value

Square root of a.

11.8.30.3 Example

Example 127: codasip_sqrt function

```
semantics
{
    float64 a;
    float64 b;

    a = 16.0f;
    b = codasip_sqrt_float64(a); //b = 4.0f;
    ...
};
```

11.9 Vector Functions

11.9.1 codasip_select

Semantic category - general

Supported on - IA

`<type1> codasip_ select_ <type1> (<type1> cond, <type1> a, <type1> b)`

The function is used to choose a value based on the condition. It requires 3 vectors of the same type. Then it check all cond elements. If the element is non-zero, then the corresponding element of a is selected, else the element of b is selected.

Note: Vector elements must be unsigned.

11.9.1.1 Parameters

cond

Conditions vector.

a, b

Vectors containing elements, which are to be selected.

11.9.1.2 Return Value

Returns vector of `<type1>` with selected elements.

11.9.1.3 Example

Example 128: codasip_select function

```
semantics
{
    ...
    v4u4 cond = { 1, 0, 0, 1};
    v4u4 a = {1, 2, 3, 4};
    v4u4 b = {5, 6, 7, 8};
    v4u4 res;

    // Elements 1 and 4 of vector 'cond' are non-zero, so vector 'a'
    // will be selected for these and vector 'b' for the two remaining
    // elements.
    res = codasip_select_v4u4(cond, a, b);
    // 'res' will contain { 1, 6, 7, 4}
    ...
};
```

11.9.2 codasip_sext

Semantic category - general

Supported on - IA

`<type2> codasip_sext_<type1>_<type2>(<type1> a)`

The function performs a sign extension of all the elements of the input vector, of type `<type1>`, to return a result vector of type `<type2>`. To perform the sign extension, the sign bit (highest order bit) of each element of a is copied into the higher order bits of the output element.

Both `<type1>` and `<type2>` must be vectors with the same element count and the bit width of `<type1>` must be smaller than the bit width of `<type2>`.

11.9.2.1 Parameters

a

A vector whose elements are to be sign extended.

11.9.2.2 Return Value

Sign extended vector. Each element has the same bit width as `<type2>`.

11.9.2.3 Example

Example 129: codasip_sext function

```
semantics
{
    ...
    v2i8 vec_in = {-5, 64}; // In binary {1111 1011, 0100 0000}
    v2i16 vec_out;

    vec_out = codasip_sext_v2i8_v2i16(vec_in);
    // vec_out will contain {1111 1111 1111 1011, 0000 0000 0100 0000}
    ...
};
```

11.9.3 codasip_shufflevector

Semantic category - general

Supported on - IA

`<type1> codasip_ shufflevector_ <type1>_ <type2> (<type1> a, <type1> b, <type2> mask)`

Constructs a permutation of elements of two input vectors. The input vectors must have the same characteristics - see below - and the returned vector will have the same characteristics.

A `mask` vector argument is used to control the shuffle. It does not have to have exactly the same characteristics as the input vectors, but it must have the same length - see below.

Suppose that the length of the vectors is L. Each element of the mask holds the index of the input vector element to include in the corresponding output. If i<L, then it indicates

the ith element of the first input vector, `a` (where the index starts at 0). If i>=L, then it indicates the (i-L)th element of the second input vector, `b`.

The characteristics of the input and output vectors must be as follows:

- element count: 2, 4, 8, 16
- element bitwidth: 8, 16, 32, 64

The characteristics of the mask vector must be as follows:

- the element count must be the same as that of the input vectors
- element bitwidth: 8, 16, 32, 64

11.9.3.1 Parameters

a, b

Input vectors with the same characteristics

mask

Specifies, for each element of the result vector, which element from the two input vectors should be used.

11.9.3.2 Return Value

A vector with the same characteristics as `a` and `b`.

11.9.3.3 Example

Example 130: codasip_shufflevector function (output vector with two different elements)

```
semantics
{
    v2i8 vsrc1, vsrc2, vmask, vdst;

    vsrc1[0] = 25; // element 0
    vsrc1[1] = 63; // element 1
    vsrc2[0] = 19; // element 2
    vsrc2[1] = 88; // element 3
    vmask[0] = 0;
    vmask[1] = 3;
    vdst = codasip_shufflevector(vsrc1, vsrc2,vmask); // vdst = (25,88)
    ...
};
```

Example 131: codasip_shufflevector function (output vector with two same elements)

```
semantics
{
    v2i8 vsrc1, vsrc2, vmask, vdst;

    vsrc1[0] = 25; // element 0
    vsrc1[1] = 63; // element 1
    vsrc2[0] = 19; // element 2
    vsrc2[1] = 88; // element 3
    vmask[0] = 1;
    vmask[1] = 1;
    vdst = codasip_shufflevector(vsrc1, vsrc2,vmask); // vdst = (63,63)
    ...
};
```

11.9.4 codasip_trunc

Semantic category - general

Supported on - IA

`<type2> codasip_trunc_<type1>_<type2>(<type1> a)`

The function truncates its `<type2>` operand to a return value of type `<type2>`. To perform the truncation of each element of the input vector, the excess high order bits are ignored. `<type1>` must therefore have more bits than `<type2>`

11.9.4.1 Parameters

a

A vector whose elements are to be truncated.

11.9.4.2 Return Value

Truncated vector. Each element has the same bit width as `<type2>`.

11.9.4.3 Example

Example 132: codasip_trunc function

```
semantics
{
    ...
    v2u16 vec_in = {15, 320}; // In binary {0000 0000 0000 1111, 0000 0001 0100 0000}
    v2u8 vec_out;

    vec_out = codasip_trunc_v2i16_v2i8(vec_in);
    // vec_out will contain {0000 1111, 0100 0000}
    ...
};
```

11.9.5 codasip_zext

Semantic category - general

Supported on - IA

`<type2> codasip_zext_<type1>_<type2>(<type1> a)`

The function performs a zero extension of all the elements of the input vector, of type `<type1>`, to return a result vector of type `<type2>`. To perform the zero extension, zero is copied into the higher order bits of the output element.

Both `<type1>` and `<type2>` must be vectors with the same element count and the bit width of `<type1>` must be smaller than the bit width of `<type2>`.

11.9.5.1 Parameters

a

A vector whose elements are to be zero extended.

11.9.5.2 Return Value

Zero extended vector. Each element has the same bit width as `<type2>`.

11.9.5.3 Example

Example 133: codasip_zext function

```
semantics
{
    ...
    v2u8 vec_in = {15, 196}; // In binary {0000 1111, 1100 0000}
    v2u16 vec_out;

    vec_out = codasip_zext_v2i8_v2i16(vec_in);
    // vec_out will contain {0000 0000 0000 1111, 0000 0000 1100 0000}
    ...
};
```

11.10 Fixed Point Functions

11.10.1 codasip_fx_div

Semantic category - general

Supported on - IA

`<type1> codasip_fx_div_<type1>(<type1> a, <type1> b, int32 fract_bits, uint1 rounding_flag)`

Divides `a` by `b` using fixed point arithmetic.

The function currently supports these types:

- `int8`
- `int16`

- `int32`
- `int64`

11.10.1.1 Parameters

a

Dividend.

b

Divisor.

fract_bits

Number of bits representing the decimal part.

rounding_flag

If rounding_flag is 1, then the least significant bit is rounded up, else thre is no rounding.

11.10.1.2 Return Value

Returns the result of the division.

11.10.1.3 Example

Example 134: codasip_fx_div function

```
semantics
{
    ...
    fx_div_res = codasip_fx_div_int32(reg_src1, reg_src2, frac_bits, round);
    ...
};
```

11.10.2 codasip_fx_fptofx_to

Semantic category - general

Supported on - IA

`<type2> codasip_fx_fptofx_<type1>_to_<type2>(<type1> a, int32 fract_bits)`

The function converts a number from floating point to fixed point. The number of decimal bits is given by the fract_bits parameter.

Note: Using this function may lead to precision loss as the floating point number might be too small or too large for fixed point.

`<type1>` might be:

- `float16`
- `float32`
- `float64`

`<type2>` might be:

- `int16`
- `int32`
- `int64`

11.10.2.1 Parameters

a

The source floating point number, which is to be converted to a fixed point.

fract_bits

Number of bits representing decimal part of the result.

11.10.2.2 Return Value

Returns the result of a conversion. The type is the same as `<type2>`.

11.10.2.3 Example

Example 135: codasip_fx_fptofx_to function

```
semantics
{
    ...
    fx_res = codaip_fx_fptofx_float32_to_int32(src, frac_bits);
    ...
};
```

11.10.3 codasip_fx_fxtofp_to

Semantic category - general

Supported on - IA

`<type2> codasip_fx_fxtofp_<type1>_to_<type2>(<type1> a, int32 fract_bits)`

The function converts a number from fixed point to floating point. The number of decimal bits of a is given by the fract_bits parameter.

Note: Using this function may lead to precision loss as the floating point number might be too small or too large for fixed point.

`<type1>` might be:

- `int16`
- `int32`
- `int64`

`<type2>` might be:

- `float16`
- `float32`
- `float64`

11.10.3.1 Parameters

a

The source floating point number, which is to be converted to a fixed point.

fract_bits

Number of bits representing decimal part of the result.

11.10.3.2 Return Value

Returns the result of a conversion. The datatype is the same as `<type2>`.

11.10.3.3 Example

Example 136: codasip_fx_fptofx_to function

```
semantics
{
    ...
    fx_res = codaip_fx_fptofx_float32_to_int32(src, frac_bits);
    ...
};
```

11.10.4 codasip_fx_fxtoi_to

Semantic category - general

Supported on - IA

`<type2> codasip_fx_fxtoi_<type1>_to_<type2>(<type1> a, int32 fract_bits)`

The function converts a fixed point number to an integer. The number of decimal bits is given by `fract_bits`. The source is arithmetically shifted right by `fract_bits` bits. The result of the shift is then truncated so that it has the same bit width as `<type2>`.

`<type1>` and `<type2>` may be:

- `int8`
- `int16`
- `int32`
- `int64`

11.10.4.1 Parameters

a

The source fixed point number.

fract_bits

Number of bits representing the decimal part of the result.

11.10.4.2 Return Value

The result of the conversion.

11.10.4.3 Example

Example 137: codasip_fx_fxtoi_to function

```
semantics
{
    ...
    int 8 src, int_res;
    int32 feac_bits;
    // Consider 'src' as a fixed point with 1 sign bit, 2 integer bits and 5 decimal bits
    frac_bits = 5; // 5 bits for decimal part
    src = 80; // In binary 0101 0000

    // The result will be equal to (src / (2^frac_bits)) = 80/32 = 2.5.
    int_res = codasip_fx_int8_fxtoi_to_int8(src, frac_bits); // int_res = 0b0000 0010 = 2
                                                             // Note that decimal part has
                                                             // been lost.
    ...
};
```

11.10.5 codasip_fx_itofx_to

Semantic category - general

Supported on - IA

`<type2> codasip_fx_itofx_<type1>_to_<type2>(<type1> a, int32 fract_bits)`

The function converts an integer number to fixed point. The number of decimal bits is given by `fract_bits` . The source is shifted left by `fract_bits` bits. The result of the shift is then truncated so it has the same bit width as `<type2>`.

`<type1>` and `<type2>` may be:

- `int8`
- `int16`
- `int32`
- `int64`

11.10.5.1 Parameters

<u>a</u>

The source integer number.

<u>fract_bits</u>

The number of bits in the decimal part of the result.

11.10.5.2 Return Value

The result of the conversion.

11.10.5.3 Example

Example 138: codasip_fx_itofx_to function

```
semantics
{
    ...
    int8 src, fx_res;
    int32 frac_bits;
    // 'src' is an integer with 1 sign bit and 7 integer bits
    frac_bits = 4; // 4 bits for (results) decimal part
    src = 6; // In binary 0000 0110

    // The result will be equal to (src * (2^frac_bits)) = 6*16 = 96.
    fx_res = codasip_fx_int8_itofx_to_int8(src, frac_bits); // fx_res = 0b0110.0000 (=96)
    ...
};
```

11.10.6 codasip_fx_mul

Semantic category - general

Supported on - IA

`<type1> codasip_fx_mul_<type1>(<type1> a, <type1> b, int32 fract_bits, uint1 rounding_flag)`

The function multiplies two numbers using fixed point arithmetic.

The function supports these values of `<type1>`:

- `int8`
- `int16`
- `int32`
- `int64`

11.10.6.1 Parameters

a, b

Operands to multiply.

fract_bits

Number of bits representing decimal part.

rounding_flag

If rounding_flag is 1, then the least significant bit is rounded up, else do not round at all.

11.10.6.2 Return Value

Returns the result of a mutiplication.

11.10.6.3 Example

Example 139: codasip_fx_div function

```
semantics
{
    ...
    fx_mul_res = codasip_fx_mul_int32(reg_src1, reg_src2, frac_bits, round);
    ...
};
```

11.11 Complex Numbers Functions

11.11.1 codasip_cplx_add

Semantic category - general

Supported on - IA

`<type1> codasip_cplx_add_<type1>(<type1> a, <type1> b)`

Perform complex addition, either scalar or vector.

Although vector addition is the same for integers and complex numbers, this operation specifies that this is a vector complex addition that is recognized by the compiler generator and can be used by the compiler vectorizer.

The currently supported combinations of vector types for complex numbers have these parameters: element count: 2, 4, 8, 16, and element size: 16, 32, 64. Elements can be either integers (16 - int16_t, 32 - int32_t, 64 - int64_t) or floating point values (16 - half, 32 - float, 64 - double).

The complex numbers are represented as vector data types where always 2 elements contain one complex number. For example v2i32 represents one complex number with 32-bit real and imaginary parts, v8i32 is then a vector of 4 such complex number. This approach to model complex numbers was chosen because it allows vectorization in C.

It is also possible to use standard complex C data type (__complex), but the efficiency may not be so high as when using the vector representation for complex types.

11.11.1.1 Parameters

`a,b`

Complex vector variables.

11.11.1.2 Return Value

Result of addition.

11.11.2 codasip_cplx_div

Semantic category - general

Supported on - IA

`<type1> codasip_cplx_div_<type1>(<type1> a, <type1> b)`

Perform complex division, either scalar or vector.

For more details on complex data type operations see `codasip_cplx_add`.

Result of division by 0 is undefined and will possibly terminate simulation.

Example 140 Implementation of complex division

```
DATA_TYPE codasip_cplx_div_DATA_TYPE(DATA_TYPE a, DATA_TYPE b)
{
  DATA_TYPE res;
  for (int i = 0; i < ELEM_COUNT/2; i+=2)
  {
    res[i] = (a[i] * b[i] + a[i+1] * b[i+1]) / (b[i] * b[i] + a[i+1] * b[i+1]);
    res[i+1] = (a[i+1] * b[i] - a[i] * b[i+1]) / (b[i] * b[i] + a[i+1] * b[i+1]);
  }
  return res;
}
```

11.11.2.1 Parameters

`a,b`

Complex vector variables.

11.11.2.2 Return Value

Result of complex division.

11.11.3 codasip_cplx_mul

Semantic category - general

Supported on - IA

`<type1> codasip_cplx_mul_<type1>(<type1> a, <type1> b)`

Perform complex multiplication, either scalar or vector.

For more details on complex data type operations see `codasip_cplx_add`.

Example 141: Implementation of complex multiplication

```
DATA_TYPE codasip_cplx_div_DATA_TYPE(DATA_TYPE a, DATA_TYPE b)
{
  DATA_TYPE res;
  for (int i = 0; i < ELEM_COUNT/2; i+=2)
  {
       res[i] = a[i] * b[i] - a[i+1] * b[i+1];
       res[i+1] = a[i] * b[i+1] - a[i+1] * b[i];
  }
  return res;
}
```

11.11.3.1 Parameters

`a,b`

Complex vector variables.

11.11.3.2 Return Value

Result of complex multiplication.

11.11.4 codasip_cplx_sub

Semantic category - general

Supported on - IA

`<type1> codasip_cplx_sub_<type1>(<type1> a, <type1> b)`

Perform complex subtraction, either scalar or vector.

For more details on complex data type operations see `codasip_cplx_add`.

11.11.4.1 Parameters

`a,b`

Complex vector variables.

11.11.4.2 Return Value

Result of complex subtraction.

11.12 Miscellaneous Functions

11.12.1 codasip_bitcast_to

Semantic category - general

Supported on - IA

`<type2> codasip_bitcast_<type1>_to_<type2>(<type1> value)`

This function converts the value of an input type, `<type1>`, to a target type, `<type2>`, without changing any bits. The bitsize of the input type must match the size of the target type (the total size in the case of vector). The list below shows the possible combinations of parameter and return value. There are two types of bitcasts:

- Integer to float (and vice versa).
  The following conversions are allowed(`<type1>` to `<type2>`):
  - `uint16` to `float16` (`float16` to `uint16`)
  - `uint32` to `float32` (`float32` to `uint32`)
  - `uint64` to `float64` (`float64` to `uint64`)
  - `uint80` to `float80` (`float80` to `uint80`)
  - `uint16` to `float16` (`float16` to `uint16`)
- Integer to vector of floats (and vice versa).
  The following conversions are allowed(`<type1>` to `<type2>`):

- `uint64` to `v2f32` (`v2f32` to `uint64`)
- `uint128` to `v4f32` (`v4f32` to `uint128`)
- `uint128` to `v2f64` (`v2f64` to `uint128`)
- `uint256` to `v8f32` (`v8f32` to `uint256`)
- `uint256` to `v4f64` (`v4f64` to `uint256`)
- `uint512` to `v16f32` (`v16f32` to `uint512`)
- `uint512` to `v8f64` (`v8f64` to `uint512`)
- `uint1024` to `v16f64` (`v16f64` to `uint1024`)

These bitcasts do simple bit-precise copy, an example of internal implementation in C can be seen here:

Example 142: codasip_bitcast

```
float codasip_bitcast_uint32_to_float32(uint32_t uninterpreted_value)
{
	return *((float *)&uninterpreted_value);
}
```

11.12.1.1 Parameters

<u>value</u>

Input value of type `<type1>` for the conversion.

11.12.1.2 Return Value

Returned converted value of type `<type2>`.

11.12.1.3 Example

Example 143:codasip_bitcast_uint128_to_v4u32 function use

```
// vregs vector read unsigned
v4u32 vec_read_v4u32(const uint4 idx)
{
    return codasip_bitcast_uint128_to_v4u32(rf_vec[idx]);
}
```

12 CODASIP COMPONENTS

This chapter provides information on how to implement external components for Instruction Accurate (IA) and Cycle Accurate (CA) simulators and RTL.

A component is required when an `extern` construct is specified in a CodAL model.

The component is always associated with its type and so it can exist in multiple instances. If the component is using shared resources, it is necessary to ensure mutually exclusive access to these resources. There's no guarantee that components are running in the same thread.

The main interface has to be written in C++ language. When there is a requirement to write the component in another language or to use a specialized library, then the C++ interface becomes just a wrapper providing a bridge between the simulator and the component implementation in question.

This chapter is organized as follows:

12.1 Component Configuration

Each component has its own configuration. The configuration consists of a port and interface definition. Then there are flags denoting whether the component has a reset port or clock port. The configuration is the main source file for a generator that generates all wrapper code as well as testbenches or makefiles.

The configuration may be written by hand or it may be created via Codasip Studio wizard. The following examples shows a component that has no clock or reset and it has two input ports and one output port. Such a component usually represents come combinatorial logic (e.g. multiply-and-accumulate).

Example 144: Component configuration

```
[component]
  has-reset = false
  has-clock = false

  [[component.port]]
    name = p_in1
    bit = 32
    direction = IN

  [[component.port]]
    name = p_in2
    bit = 32
    direction = IN

  [[component.port]]
    name = p_out
    bit = 32
    direction = OUT
```

12.2 C++ namespace

Every constants and classes pertain to C++ namespace `codasip::resources`. To avoid prepending of namespace with the every symbol following directive can be used:

```
using namespace codasip::resources;
```

12.3 Payload

The payload is used to pass information to and from every interface callbacks. The payload encapsulates one transaction and carry all information about command, data with specific type, address and signals. The information and signals stored in the payload depends on specific protocol.

The following description regarding the `CLB` and `MEMORY` protocols which CodAL language uses for internal communication.

Methods to access stored information:

- codasip_address_t **get_address**() const
- DATA_TYPE **get_data**() const
- Index **get_bi**() const - Get byte index of first byte in a word.
- Index **get_bc**() const - Get byte count.
- Command **get_command**() const

- Uid **get_uid**() const
- Response **get_response**() const
- Status **get_status**() const

Methods to set information:

- void **set_address**(const codasip_address_t address)
- void **set_data**(const T data)
- void **set_bi**(const Index bi)
- void **set_bc**(const Index bc)
- void **set_command**(const Command command)
- void **set_uid**(const Uid uid)
- void **set_response**(const Response response)
- void **set_status**(const Status status)

12.4 Interface Command Constants

`codasip::resources::Command`

The following constants represent all commands type currently supported by simulator.

Request command types:

- `CP_CMD_NONE` (0) - No active request.
- `CP_CMD_READ` (1) - Initiate read data from given address.
- `CP_CMD_WRITE` (2) - Initiate write operation.
- `CP_CMD_INVALIDATE` (4) - Invalidate cache line.
- `CP_CMD_INVALIDATE_ALL` (5) - Invalidate the whole cache.
- `CP_CMD_FLUSH` (6) - Flush cache line.
- `CP_CMD_FLUSH_ALL` (7) - Flush the whole cache.

Simulation command types:

- `CP_CMD_LOAD` (100) - Load data to component.
- `CP_CMD_DREAD` (101) - Read data initiated by debugger.
- `CP_CMD_DWRITE` (102) - Write data initiated by debugger.

12.5 Interface Status Constants

`codasip::resources::Status`

This section describes status constants which can be returned from request method. These method are used only for CA models and the return constants represent the state of transaction which should be checked.

Request status constants:

- `CP_ST_BUSY` (0) - The slave cannot accept new request.
- `CP_ST_READY` (1) - The slave has accepted request and is waiting for finish.
- `CP_ST_ERROR` (3) - An unknown error occurred.

12.6 Interface Response Constants

`codasip::resources::Response`

Similarly to interface status constants these constants are used only in CA mode. The only differece is that its value is set in finish method instead of request method.

Finish response constants:

- `CP_RS_IDLE` (0) - The slave does not perform any transfer.
- `CP_RS_ACK` (1) - Transfer successfully ended.
- `CP_RS_WAIT` (2) - Transfer is still running and master should wait.
- `CP_RS_ERROR` (3) - An unknown error occurred.
- `CP_RS_UNALIGNED` (4) - Slave do not support unaligned access.
- `CP_RS_OOR` (5) - Access out of range of the slave.

It is recommended to use these symbolic constants instead of using direct values because these values could be changed in the future.

12.7 Callbacks

If a component has slave interfaces or input ports, a component must implement three mandatory methods for the slave interfaces and two mandatory methods for the input ports.

12.7.1 Callbacks for Slave Interfaces

Each slave interface calls `blocking_cb`, `request_cb`, or `finish_cb` whenever there is a transfer initiated by master interfaces.

Example 145: Interface callbacks

```
void blocking_cb(codasip::resources::Payload<DATA_TYPE>& p, codasip::resources::Uid uid)
void request_cb(codasip::resources::Payload<DATA_TYPE>& p, codasip::resources::Uid uid)
void finish_cb(codasip::resources::Payload<DATA_TYPE>& p, codasip::resources::Uid uid)
```

`p` – Payload with all information for the current transaction.

`uid` – Identifier of an interface.

12.7.2 Callback for Input Ports

Each input port calls `write` or `dwrite` whenever it is written.

Example 146: Asynchronous callback

```
void write_cb(const DATA_TYPE data, codasip::resources::Uid uid)
void dwrite_cb(const DATA_TYPE data, codasip::resources::Uid uid)
```

`data` – Value which has been stored.

`uid` – Identifier of an input port.

12.8 Access Methods

Access methods are provided to let components communicate with its output ports and/or master interfaces.

Some methods access the master interface of components (e.g. cache) or output ports - these are associated with interface object which is part of the component member variables and has the same name as the name defined in component description. For slave interfaces, appropriate callbacks are generated and behaviour is implemented as a response to these methods called from another master component.

The components for IA and CA models have to use different methods to access their interfaces. For the IA model there exists only one method with the name `blocking`. For the CA model the functionality is divided into `request` and `finish` methods. Several special commands have only `request` part (e.g. CP_CMD_INVALIDATE). The `request` method is intended to initiate transaction and `finish` is for processing the result. These two methods are usually called in different clock cycles because transaction may take some time. Response codes from both methods are described in the chapters "Interface Status Constants" on page 293 and "Interface Response Constants" on page 294.

12.8.1 Port read/write Methods

These methods, for accessing the component ports, are typically used in clock or asynchronous callback to alternate behaviour of the component through its ports. Only caveat is that writing to the port can cause recursive call of asynchronous callback of the same component.

Example 147: Ports access methods

```
DATA_TYPE read() const
DATA_TYPE dread() const
DATA_TYPE write(const DATA_TYPE data)
DATA_TYPE dwrite(const DATA_TYPE data)
```

`data` - Data to be written.

There are variants of the read/write command with a `d` prefix `dread` and `dwrite`. They are for access from the debugger and should provide less restrictive access to the interface. Usually, sub-block and unaligned access is provided and warnings are suppressed.

12.8.2 Interface Blocking Method

This method is intended for both IA and CA mode and are used by loader, debugger and IA simulator. Unlike non-blocking methods this type of method responses immediately to the request and so it is not convenient for asynchronous CA simulator. CA simulator uses this callbacks only for loader, debugger and code encapsulated in `#pragma simulator` block which is useful for debugging purposes.

Example 148: Interface blocking method

```
void blocking(codasip::resources::Payload<DATA_TYPE>& p)
```

`p` – Payload with all information for current transaction.

The type of requested command has to be stored in the payload. For blocking access there are supported only these commands:

- `CP_CMD_READ` / `CP_CMD_DREAD`
- `CP_CMD_WRITE` / `CP_CMD_DWRITE`
- `CP_CMD_INVALIDATE` / `CP_CMD_INVALIDATE_ALL`
- `CP_CMD_FLUSH` / `CP_CMD_FLUSH_ALL`
- `CP_CMD_LOAD`

Each command is implemented via a helper methods too. See the folloving subsection.

12.8.2.1 Helper Methods

The following chapter continues with description of helper methods which internally sets the paylod accordingly its input parameters and call correspondent blocking method. These methods can be used optionally and their purpose is only for simpler code description.

12.8.2.1.1 Blocking Read Method

This method is for reading from interface from specific address. This method is intended to be used only in IA model. Method immediately returns read value.

Example 149: Interface blocking read methods

```
DATA_TYPE read(const codasip_address_t addr,
    const int bi = 0, const int bc = BYTES_PER_WORD)
DATA_TYPE dread(const codasip_address_t addr,
    const int bi = 0, const int bc = BYTES_PER_WORD)
```

`addr` - Address from where the data are going to be read.

`bi` - Byte index of first byte in a word.

`bc` - Byte count.

12.8.2.1.2 Blocking Write Method

This method is similar as previous one but is meant to be used for writing to given address. This method should be used only in IA mode too.

Example 150: Interface blocking write methods

```
DATA_TYPE write(const DATA_TYPE data, const codasip_address_t addr,
    const int bi = 0, const int bc = BYTES_PER_WORD)
DATA_TYPE dwrite(const DATA_TYPE data, const codasip_address_t addr,
    const int bi = 0, const int bc = BYTES_PER_WORD)
```

`data` - Value to be stored.

`addr` - Address from where the data are going to be written.

`bi` - Byte index of first byte in a word.

`bc` - Byte count.

12.8.2.1.3 Load Method

This method is useful only for simulation and is used for loading input program to interface.

Example 151: Loading input program method

```
void load(const DATA_TYPE data, const codasip_address_t addr,
    const int bi = 0, const int bc = BYTES_PER_WORD)
```

`data` - Value to be loaded.

`addr` - Address to where the data are going to be loaded.

`bi` - Byte index of first byte in a word.

`bc` - Byte count.

12.8.2.1.4 Cache Methods

Example 152: Cache line invalidate and flush methods

```
Response invalidate(codasip_address_t addr)
Response invalidate_all(codasip_address_t addr)
Response flush(codasip_address_t addr)
Response flush_all(codasip_address_t addr)
```

`addr` - Address to be invalidated or flushed.

The goal of invalidate and flush method is to provide interface for cache support but it is not limited only for caches. The invalidate method marks cache line that the data will have to be invalidated or deleted. The flush method forces the cache line to be written to the higher level of cache hierarchy. There exist method alternatives with suffix `all` in its name where aforementioned operations are applied on the whole cache. Address must be specified in the `all` variant also because of bus interfaces where more then one component can be connected and address specifies which one is to be invalidated or flushed. These methods should be called only in IA model. For CA model is used generic request method with appropriate command type.

12.8.3 Interface Non-blocking Methods

These methods are intended only for CA mode and consists of two methods `request` and `finish`.

Example 153: Interface non-blocking methods

```
void request(codasip::resources::Payload<DATA_TYPE>& p)
void finish(codasip::resources::Payload<DATA_TYPE>& p)
```

`p` – Payload with all information for current transaction.

The type of requested command has to be stored in the payload. For non-blocking access, the following commands are supported :

- `CP_CMD_NONE`
- `CP_CMD_READ`
- `CP_CMD_WRITE`
- `CP_CMD_INVALIDATE` / `CP_CMD_INVALIDATE_ALL`
- `CP_CMD_FLUSH` / `CP_CMD_FLUSH_ALL`

The request method (address phase of the transfer) is for CA models only and is used to initiate a command. The finish method (data phase of the transfer) picks up/stores data itself.

12.8.3.1 Helper Methods

The following chapter continues with description of helper methods which internally sets the paylod accordingly its input parameters and call correspondent non-blocking method. These methods can be used optionally and their purpose is only for simpler code description.

12.8.3.1.1 Request Method

This method is for CA models only and is used to initiate request command. The method returns `CP_ST_READY` state when requested component is ready to accept finish command in the next cycle/cycles. When component is not ready it returns `CP_ST_BUSY` state and request should be repeated until ready state is returned.

Example 154: Interface request command method

```
Status request(const codasip::resources::Command command, const codasip_address_t addr,
    const int bi = 0, const int bc = BYTES_PER_WORD)
```

`command` - Requested command.

`addr` - Address from where the data are going to be (read / written / invalidated / flushed).

`bi` - Byte index of first byte in a word.

`bc` - Byte count.

12.8.3.1.2 Finish Method

This method is called in CA models only after request command and are intended for receiving or passing input/output data. Read and write request should be paired with `finish` method. Other commands are not paired with finish and should be called only with `request` method.

Example 155: Methods called after request command

```
Response finish(const codasip::resources::Command command, RESOURCE& res)
```

`command` - Finish command.

`res` - Resource to be read or to be written.

Return response code should be checked to determine if previous request has been finished properly.

12.8.4 Debugger support

Debugger support is implemented by a component base class `Component`. This base class manages child resources of the component and is used by debugger to read and write to these resources.

To make component's resource visible in the debugger, the `Insert` method with resource object as an argument has to be called in the component constructor.

For implementing read and write access to the component resources, following methods have to be implemented:

- `MaxInt` **`ResourceRead`**`(const Uid uid, const Address addr = 0)`
  Returns value of the resource. If given resource is addressable, returns an entire word.
- `MaxInt` **`ResourceRead`**`(const Uid uid, const Address addr, const unsigned sbi, const unsigned sbc)`
  Returns sub-blocks of the resource at a given address.
- `void` **`ResourceWrite`**`(const Uid uid, const MaxInt& data, const Address addr = 0)`
  Writes data to the resource. If given resource is addressable, an entire word is written.
- `void` **`ResourceWrite`**`(const Uid uid, const MaxInt& data, const Address addr, const unsigned sbi, const unsigned sbc)`
  Writes sub-blocks to the resource at a given address.

All these functions use unique ID (`Uid`) to select the requested resource. These IDs are assigned in the constructor of the component. If these functions are not defined but some resources are registered using `Insert` method, value `0` will be returned and no writes will be performed.

Example 156: Component with port, interface and debugger functions

```
UserComponent::UserComponent(const std::string& id, const unsigned int uid)
  : Component(id, uid),
    port("port", port_UID, this),      // input port
    iface("iface", iface_UID, this)    // interface
{
    Insert(port);
    Insert(iface);
}

MaxInt UserComponent::ResourceRead(const Uid uid, const Address addr)
{
    switch (uid)
    {
    case port_UID:
        return port.dread();
    case iface_UID:
        return iface.dread(addr);            // read entire word
    return 0;
}

MaxInt UserComponent::ResourceRead(const Uid uid, const Address addr, const unsigned sbi, const
unsigned sbc)
{
    switch (uid)
    {
    case iface_UID:
        return iface.dread(addr, sbi, sbc);  // read single LAU
    return 0;
}

void UserComponent::ResourceWrite(const Uid uid, const MaxInt& data, const Address addr)
{
    switch (uid)
    {
    case port_UID:
        port.dwrite(data);
        break;
    case iface_UID:
        iface.dwrite(data, addr);            // write entire word
        break;
    }
}

void UserComponent::ResourceWrite(const Uid uid, const MaxInt& data, const Address addr, const
unsigned sbi, const unsigned sbc)
{
    switch (uid)
    {
    case iface_UID:
        iface.dwrite(data, addr, sbi, sbc);  // write single LAU
        break;
    }
}
```

12.8.5 Source Code Structure

The structure of the source code can be arbitrary with some restrictions. The makefile with list of sources to be compiled must be in the `cpp` root directory. The name of makefile is mandatory and follows the same rules as previously introduced.

Example 157: Source code structure - name prefix

```
TYPE.cmake
codasip_simulator_TYPE.cpp
```

Subdirectories are allowed but the names of the source code must be unique across the whole directory structure in order to avoid overwriting of the generated object files. The list of source files in the makefile must be enumerated with the whole path relatively to root directory of the component source files.

12.8.6 C++ Unit Test

Each component project contains unit test. It is split to two pieces, one for IA and one for CA mode. The generated sources of the unit test is placed in `cpp/unittest/` directory. You should use this unit test to debug the component before you'll use it in the simulator. The unit test uses a CMake build system. It means that if you add a source file, then you should add it to the main component cmake file (`codasip_simulator_TYPE.cmake`, see "Component Build System" on page 302). The build system configures and builds unittests in `work/unittest.ia` and `work/unittest.ca`directory. Build process itself is described in the Codasip Studio User Guide.

12.9 Component Build System

The component build system is based on the standard cmake script for CMake with specific structure to be compatible with the Codasip® Studio. The name of the script is derived from the type name of a component (`codasip_simulator_TYPE.cmake`) and it consists mainly of predefined variables with the specific meaning. All variables are optional but `COMPONENT_SOURCES` variable should contain at least one source file. `COMPONENT_LINK_LIBRARIES` contains a list of third party libraries that components use (e.g. library for math). It is optional. Following script shows all possible variables and commands compatible with the Codasip® build system:

Example 158: Build system - possible variables and commands

```
set(COMPONENT_SOURCE codasip_simulator_TYPE.cpp)
set(COMPONENT_LINK_LIBRARIES m)
```

`COMPONENT_SOURCES` - A list of source files to build. It is required to be provided with at least one file.

`COMPONENT_LINK_LIBRARIES` - A list of third party libraries needed by a component. It is optional.

12.10 Component in HDL

Comonent should have an implementation in HDL (VHDL or Verilog) if a designer plan to generate RTL. Codasip Studio pre-generate a wrapper what the designed should complete. The wrapper will contain all needed interface including mandatory ports, such as *clock* or *reset*.

Example 159: Component in HDL- structure

```

module irq_generator_t(
    // mandatory ports
    // interfaces
    // ports);

       // Place functionality here
endmodule
```

13 CODASIP LOCAL BUS

Codasip Local Bus (CLB) is a high- performance, synchronous bus used for communication between ASIP cores and internal memory, cache or other high-bandwidth peripherals. CLB supports a single-master bus system. Data bus widths of up to 1024 bits are supported, and the address bus can be up to 64 bits wide.

This chapter is organized as follows:

13.1 Bus Wiring

Figure 15 shows a bus with 2 slaves. The bus control logic contains a decoder which decodes addresses and selects the corresponding slaves. Control and data signals from selected slaves are sent back to the master. Other signals are shared between all slaves.

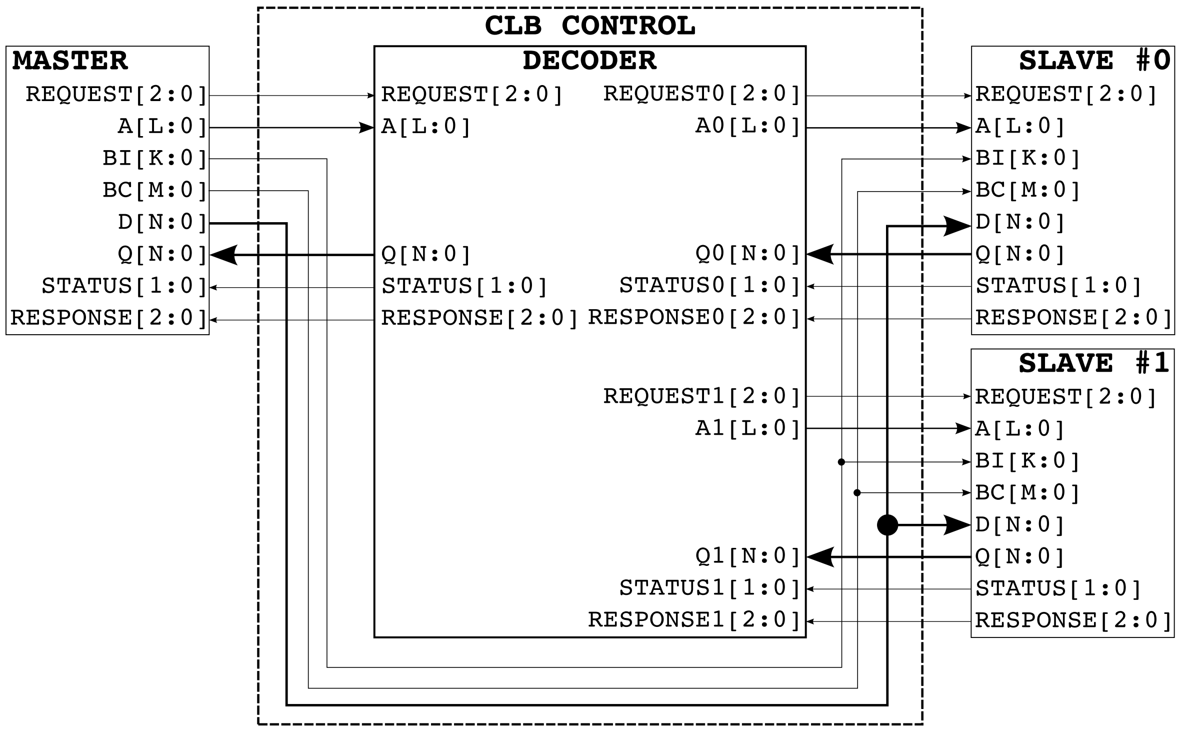


Figure 15: Bus system

13.1.1 Master Interface

The table below describes the CLB signals. K, L, M and N in the bit-width denote that the bit-width is derived from the ASIP description in the CodAL language.

Table 18: Master interface - signals description

Name	Direction	Bit-width	Description
`REQUEST`	Out	3	Requested operation (one of the request constants).
`A`	Out	L+1	Address.
`BI`	Out	K+1	Byte index.
`BC`	Out	M+1	Byte count.
`D`	Out	N+1	Write data.
`Q`	In	N+1	Read data.
`STATUS`	In	2	Status of the previous communication (one of the status constants).
`RESPONSE`	In	3	Slave response on the `REQUEST` command (one of the response constants).

13.1.2 Slave Interface

As you can see in the Figure 15, the slave has the same signals as the master, only in the opposite direction. The table below describes the signals. K, L, M and N in the bit-width denote that the bit-width is derived from the ASIP description in the CodAL language.

Table 19: Slave interface - signals description

Name	Direction	Bit-width	Description
`REQUEST`	In	3	Requested operation (one of the request constants).
`A`	In	L+1	Address.
`BI`	In	K+1	Byte index.
`BC`	In	M+1	Byte count.

`D`	In	N+1	Write data.
`Q`	Out	N+1	Read data.
`STATUS`	Out	2	Status of the slave (one of the status constants). If CP_ST_READY, then the slave must accept any `REQUEST`.
`RESPONSE`	Out	3	Response on the `REQUEST` command (one of the response constants).

13.1.3 Predefined Constants

Commands for `REQUEST` signal:

Table 20: Predefined constants - for REQUEST signal

Command	Name	Binary code	Decimal code
CP_CMD_NONE	NONE	000	0
CP_CMD_READ	READ	001	1
CP_CMD_WRITE	WRITE	010	2
Reserved		011	3
CP_CMD_INVALIDATE	INVALIDATE	100	4
CP_CMD_ INVALIDATE_ALL	INVALIDATE_ ALL	101	5
CP_CMD_FLUSH	FLUSH	110	6
CP_CMD_FLUSH_ALL	FLUSH_ALL	111	7

Constants for `STATUS` signal:

Table 21: Predefined constants - for STATUS signal

Status	Name	Binary code	Decimal code
CP_ST_BUSY	BUSY	00	0
CP_ST_READY	READY	01	1
Reserved		10	2
CP_ST_ERROR	ERROR	11	3

Constants for `RESPONSE` signal

Table 22: Predefined constants - for RESPONSE signal

Response	Name	Binary code	Decimal code
CP_RS_IDLE	IDLE	000	0
CP_RS_ACK	ACK	001	1
CP_RS_WAIT	WAIT	010	2
CP_RS_ERROR	ERROR	011	3
CP_RS_ UNALIGNED	UNALIGNED	100	4
CP_RS_OOR	OOR	101	5
Reserved		110	6
Reserved		111	7

13.1.4 Decoder

The decoder takes an address on the `A` signal and selects a slave which has this address in it's address range. If no slave is selected, the decoder sets the `STATUS` signal to CP_ST_ERROR. Otherwise, the `REQUEST` signal from master and offset of the address are re-sent to the selected slave. Signals `STATUS`, `RESPONSE` and `Q` from the selected slave are switched to the master. Other slaves on the bus get CP_CMD_NONE on the `REQUEST` signal and their output signals don't pass through the decoder to the master.

13.2 Communication Protocol

CLB uses a synchronous bus protocol. A CP_CMD_READ or CP_CMD_WRITE operation takes at least 2 clock cycles. The first clock cycle serves for addressing purposes. In the second clock cycle, data is handled. The basic transfer is shown in the Figure 16. Signals `BI` and `BC` are hidden behind an abstract `CONTROL` signal.

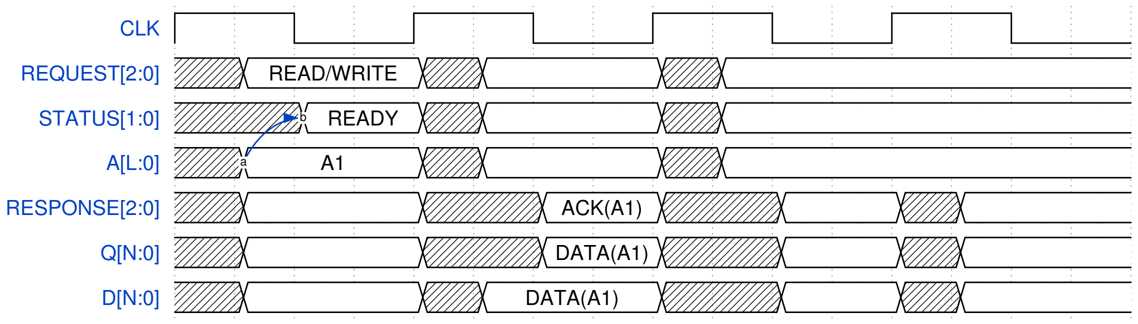


Figure 16: Basic transfer

The protocol is as follows:

- In the first clock cycle the master asserts a CP_CMD_READ or CP_CMD_ WRITE request with address and control signals. Then the CLB selects one of the slaves according to the address and switches its `STATUS` signal to master. A CP_ST_READY value on this signal means that the bus and slave are able to accept the new request. Otherwise, `STATUS` is set to CP_ST_ BUSY.
- In the second clock cycle, the master checks if a slave has finished the master's request. If the slave is done, it asserts data on `Q/D` and asserts CP_ RS_ACK on `RESPONSE` . If the slave hasn't finished the requested operation, it asserts CP_RS_WAIT on `RESPONSE`.
- If the request command is not supported, the slave act as though CP_CMD_ NONE had been asserted.

13.2.1 CP_CMD_READ Request

A basic read transfer is shown in Figure 17. In the simplest case, a CP_CMD_READ request takes 2 clock cycles. In the first clock cycle, the master asserts address and control signals. In the second clock cycle the slave places a response to the request.

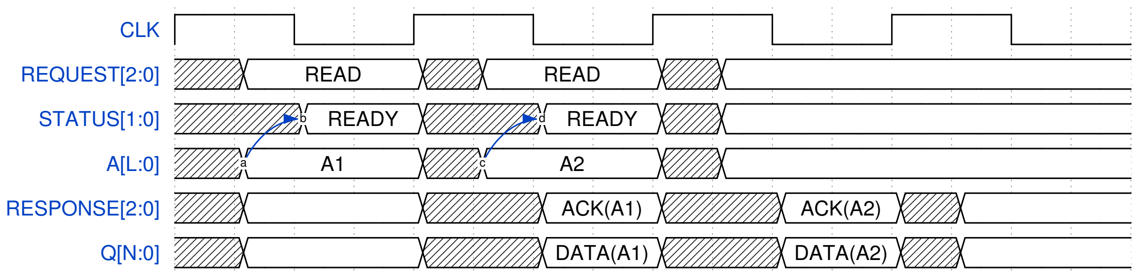


Figure 17: Simple CP_CMD_READ request

In a more complex case, a slave first accepts a CP_CMD_READ request but the slave addressed in the second clock cycle is unable to accept the second request. The master must reissue the second request because the slave has CP_ST_BUSY on `STATUS` . In the next cycle, a slave is CP_ST_READY, so the request is accepted and finished in the fourth clock cycle. See Figure 18.

Note: When CP_ST_BUSY is on STATUS, master can decide to cancel requested operation and request different operation in the next cycle.

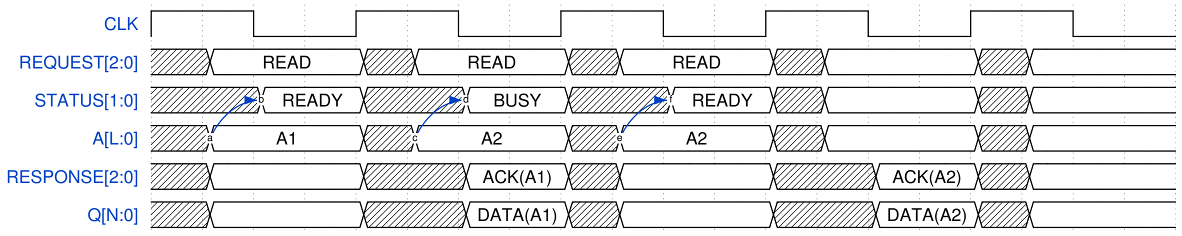


Figure 18: CP_CMD_READ request example with busy slave

Another example is in the Figure 19. In this case, the slave asserts CP_RS_WAIT on `RESPONSE`. This means that the slave needs more time to finish the request. As for the next request, if `STATUS` is CP_ST_BUSY, the request must be reissued by master until the `STATUS` is CP_ST_READY. Only then the next request can be processed.

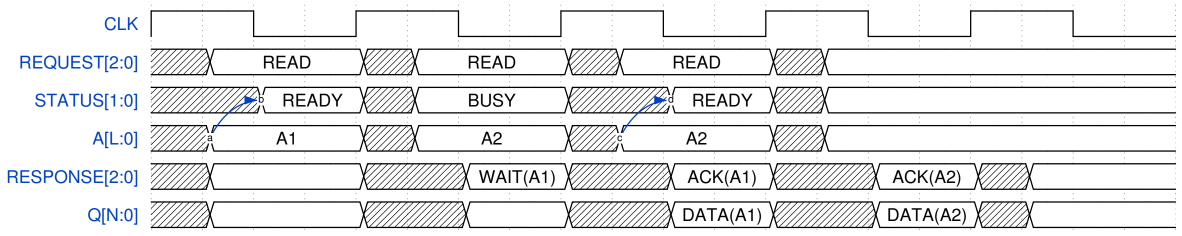


Figure 19: CP_CMD_READ request example with CP_RS_WAIT on RESPONSE

13.2.2 CP_CMD_WRITE Request

The write request uses the same timing as the read request. The slave expects data on the `D` signal in the second clock cycle. See Figure 20.

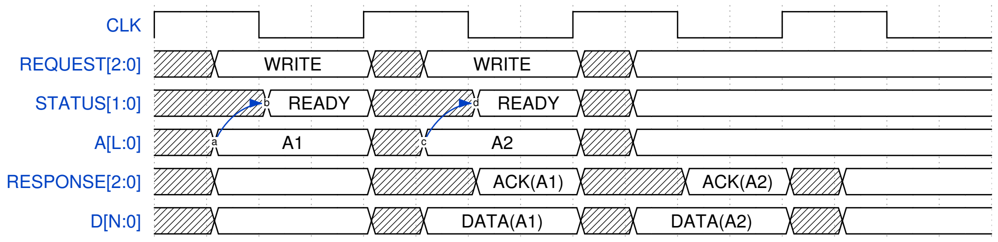


Figure 20: Simple CP_CMD_WRITE request

As you can see in Figure 21, when a slave does not assert CP_RS_ACK on RESPONSE, the master must re-issue the write request until the STATUS is CP_ST_READY. Only then the next request can be processed.

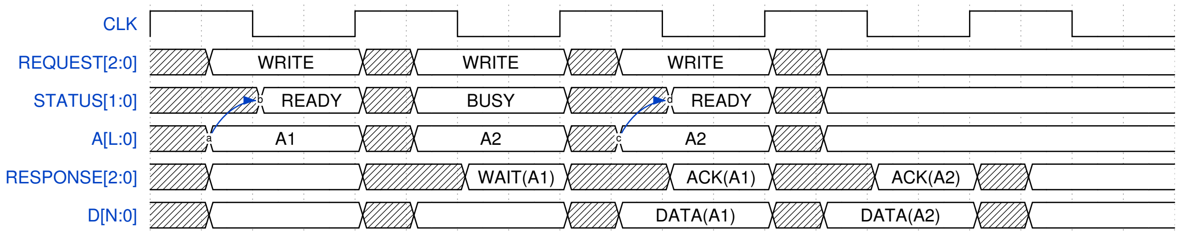


Figure 21: CP_CMD_WRITE request example with CP_RS_WAIT on RESPONSE

13.2.3 CP_CMD_INVALIDATE and CP_CMD_INVALIDATE_ALL Requests

Request CP_CMD_INVALIDATE can be used with a cache. Cache can be connected to a bus or a processor. In the first clock cycle, a cache accepts a request when it has CP_ST_READY on the STATUS signal. In the second cycle, the cache responds with CP_RS_ACK on RESPONSE. The cache must assert STATUS to CP_ST_BUSY during the invalidation. The master can use another request to check whether the invalidation is finished. See the example in the Figure 22. The request variant with the _ALL suffix means that all cache lines must be invalidated.

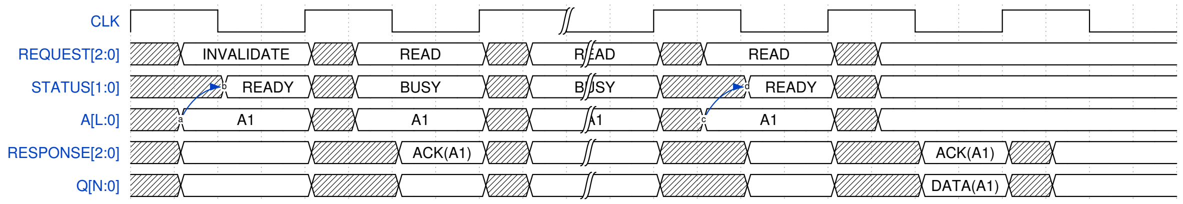


Figure 22: CP_CMD_INVALIDATE example

13.2.4 CP_CMD_FLUSH and CP_CMD_FLUSH_ALL Requests

A CP_CMD_FLUSH request can be used with a cache. In the first clock cycle, the cache shows that it can accept a request by asserting CP_ST_READY on STATUS. In the second cycle, the cache responds with CP_RS_ACK on RESPONSE. The cache must assert CP_ST_BUSY on STATUS during the flushing. The master can use another request to check whether the flush is finished. See the example in Figure 23. A request variant with the _ALL suffix means that all cache lines must be flushed.

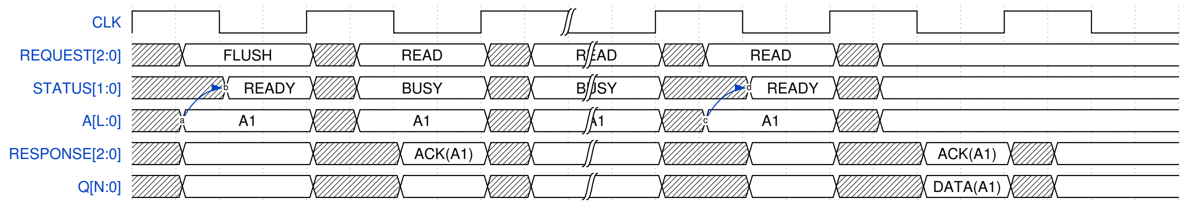


Figure 23:CP_CMD_FLUSH example

13.2.5 CP_CMD_NONE Request

The master can request CP_CMD_NONE. It means that the master does not want to do anything useful, so no operation is started on the bus. This request has no associated finish operation.

13.2.6 Transitions

Table 23: Transitions

REQUEST	STATUS	RESPONSE
t: n		t: n+1
CP_CMD_NONE	CP_ST_READY, CP_ST_ERROR	CP_RS_IDLE
CP_CMD_NONE	CP_ST_BUSY	CP_RS_IDLE, CP_RS_ACK, CP_RS_WAIT, ERRORs[1] (with respect to the previous request)
No request asserted.	CP_ST_READY, CP_ST_ERROR	CP_RS_IDLE
No request asserted.	CP_ST_BUSY	CP_RS_IDLE, CP_RS_ACK, CP_RS_WAIT, ERRORs (with respect to the previous request)

[1]ERRORs can be substituted by CP_RS_ERROR, CP_RS_OOR or CP_RS_UNALIGNED

CP_CMD_READ, CP_CMD_WRITE, CP_CMD_INVALIDATE, CP_CMD_INVALIDATE_ALL, CP_CMD_FLUSH, CP_CMD_FLUSH_ALL	CP_ST_READY	CP_RS_ACK, CP_RS_WAIT, ERRORs
CP_CMD_READ, CP_CMD_WRITE, CP_CMD_INVALIDATE, CP_CMD_INVALIDATE_ALL, CP_CMD_FLUSH, CP_CMD_FLUSH_ALL	CP_ST_BUSY	CP_RS_ACK, CP_RS_WAIT, ERRORs (with respect to the previous request)
CP_CMD_READ, CP_CMD_WRITE, CP_CMD_INVALIDATE, CP_CMD_INVALIDATE_ALL, CP_CMD_FLUSH, CP_CMD_FLUSH_ALL	CP_ST_ERROR	CP_RS_IDLE

13.2.7 Use of the BI and BC Signals

Using the `BI` signal, you can specify the index of a byte of data from the `D/Q` signal that you want to process. For example if you have a 32 bit data bus, you have 4 bytes and you can choose index 0, 1, 2 or 3. Signal `BC` then specifies the number of requested bytes. The sum of `BI` and `BC` can't be higher than the number of bytes of the data word. For example `BI = 2` and `BC = 2` is a valid combination, but `BI = 1` and `BC = 4` is not (for a 32 bit wide data word).

13.2.8 Slave Error Signaling

If the slave detects some error, there are 3 possible scenarios. In the first case we have a memory slave that does not support unaligned data access. If you try to perform an operation on unaligned address, you get CP_RS_UNALIGNED on the `RESPONSE` signal. The second case is when you try to perform an operation on an address that is out of the slave's address range. Then you get an CP_RS_OOR (Out of Range) response. Finally, if an unspecified error occurs, the slave asserts CP_RS_ERROR on `RESPONSE`, because it can't finish the previous request.

14 MEMORY INTERFACE

This is simplified interface for low latency memories like BRAMs. Unlike the CLB, this interface has no support for bus topology and it's intended to use for direct connection between ASIP core and memory.

14.1 Interface Wiring

Figure 24 shows only possible connection between master and slave. There is no other control logic and only one slave can be connected to the master interface.

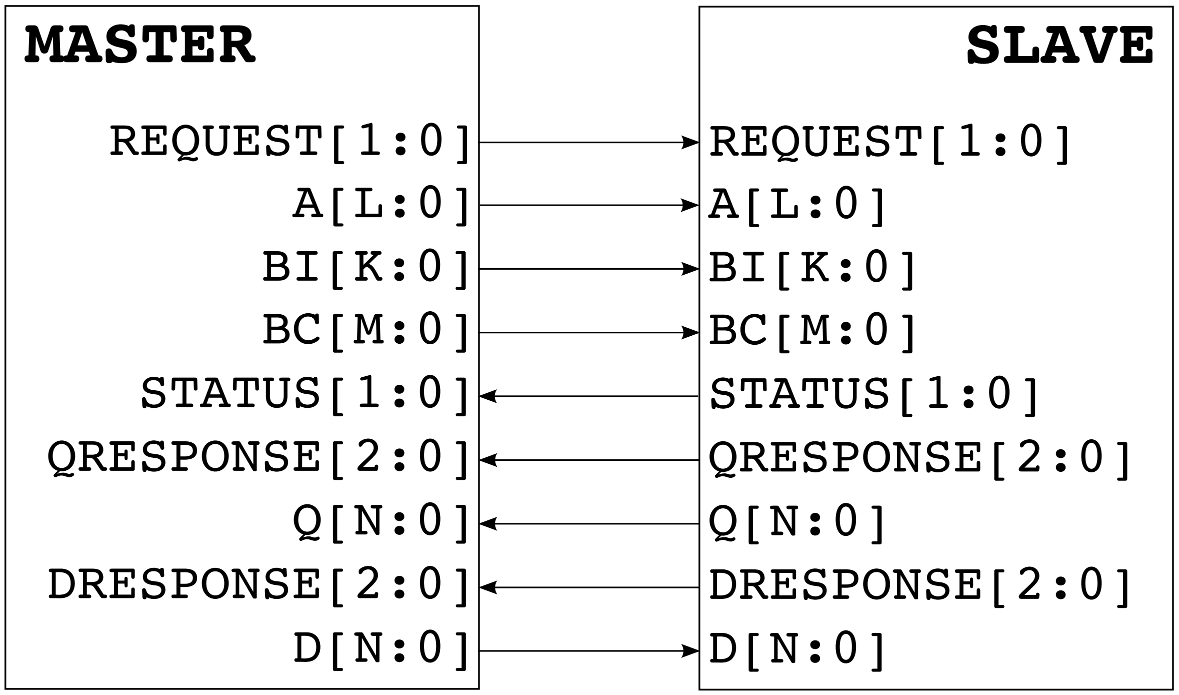


Figure 24: Direct Master <-> Slave connection

14.2 Signals Description

The tables below contains description of all signals. K, L, M and N in the bit-width means that the bit-width is derived from the ASIP description in the CodAL language.

Table 24: Master signals

Name	Direction	Bit-width	Description

`REQUEST`	Out	2	Requested operation (one of the requested constants).
`A`	Out	L+1	Address.
`BI`	Out	K+1	Byte index.
`BC`	Out	M+1	Byte count.
`STATUS`	In	2	Status of the previous communication (one of the status constants).
`QRESPONSE`	In	3	Slave response on the CP_CMD_READ operation (one of the response constants).
`Q`	In	N+1	Read data.
`DRESPONSE`	In	3	Slave response on the CP_CMD_WRITE operation (one of the response constants).
`D`	Out	N+1	Write data.

Table 25: Slave signals

Name	Direction	Bit-width	Description
`REQUEST`	In	2	Requested operation (one of the requested constants).
`A`	In	L+1	Address.
`BI`	In	K+1	Byte index.
`BC`	In	M+1	Byte count.
`STATUS`	Out	2	Status of the previous communication (one of the status constants).
`QRESPONSE`	Out	3	Slave response on the CP_CMD_READ operation (one of the response constants).
`Q`	Out	N+1	Read data.
`DRESPONSE`	Out	3	Slave response on the CP_CMD_WRITE operation (one of the response constants).
`D`	In	N+1	Write data.

14.3 Predefined Constants

Commands for `REQUEST` signal:

Table 26: Predefined constants for REQUEST signal

Command	Name	Binary code	Decimal code
CP_CMD_NONE	NONE	00	0
CP_CMD_READ	READ	01	1
CP_CMD_WRITE	WRITE	10	2
Reserved		11	3

Constants for `STATUS` signal:

Table 27: Predefined constants for STATUS signal

Command	Name	Binary code	Decimal code
CP_ST_BUSY	BUSY	00	0
CP_ST_READY	READY	01	1
Reserved		10	2
CP_ST_ERROR	ERROR	11	3

Constants for `QRESPONSE`/`DRESPONSE` signal:

Table 28: Predefined constants for QRESPONSE/DRESPONSE signal

Command	Name	Binary code	Decimal code
CP_RS_IDLE	IDLE	000	0
CP_RS_ACK	ACK	001	1
Reserved		010	2
CP_RS_ERROR	ERROR	011	3
CP_RS_UNALIGNED	UNALIGNED	100	4
CP_RS_OOR	OOR	101	5
Reserved		110	6
Reserved		111	7

14.4 Communication Protocol

Depending on the slave read latency, the communication can be synchronous or asynchronous. Write latency is always 1 clock cycle. Read latency can be up to 1 clock cycle. This implies that CP_ST_BUSY value on `STATUS` signal can be only after reset and then there will be only CP_ST_READY value. Response for supported request can be CP_RS_ACK or any of error responses. CP_RS_IDLE is valid response for CP_CMD_NONE or unsupported request.

14.4.1 Synchronous Communication

Communication protocol is synchronous when read latency is 1 clock cycles. In this case, the slave behaves as BRAM. See Figure 25 and Figure 26 with timing diagrams of CP_CMD_READ and CP_CMD_WRITE operations. Signals `BI` and `BC` are hidden behind an abstract `CONTROL` signal.

Protocol:

- In the first clock cycle the master asserts CP_CMD_READ or CP_CMD_WRITE request with address and control signals. Then master checks slave's `STATUS` signal. CP_ST_READY value on this signal means that the slave is able to accept new request. Otherwise, `STATUS` is set to CP_ST_BUSY.
- In case of CP_CMD_WRITE operation, master asserts data on `D` signal in the same clock cycle as operation request. Slave responds with CP_RS_ACK, CP_RS_ERROR, CP_RS_UNALIGNED or CP_RS_OOR on `DRESPONSE` signal due to result of the operation and the transaction ends.
- CP_CMD_READ operation continues in the next clock cycle. Slave asserts data on `Q` signal and CP_RS_ACK, CP_RS_ERROR, CP_RS_UNALIGNED or CP_RS_OOR on `QRESPONSE` signal due to result of the operation. Then the transaction is finished.

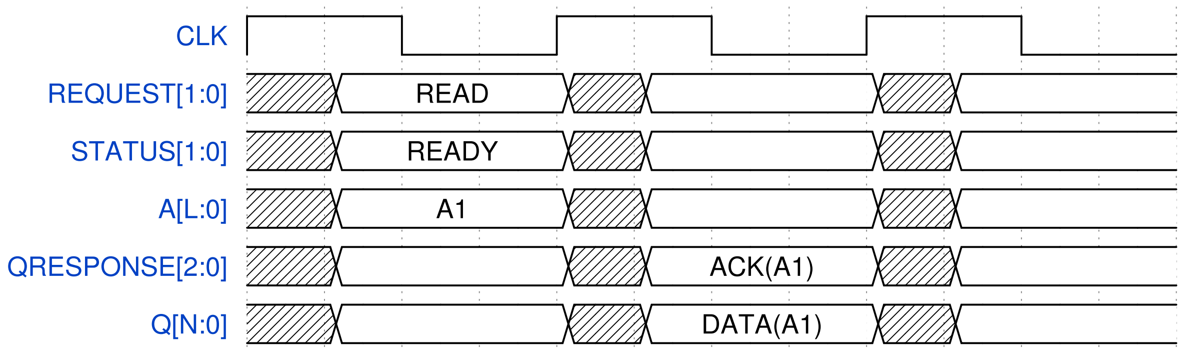


Figure 25: Synchronous CP_CMD_READ operation

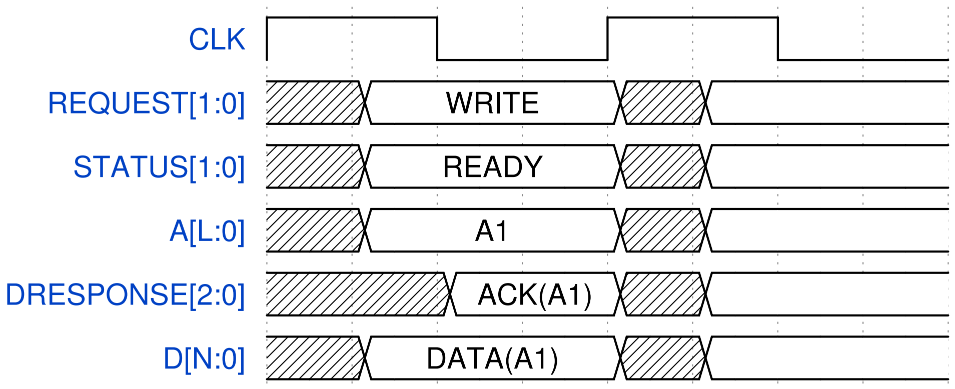


Figure 26: CP_CMD_WRITE operation (same for synchronous and asynchronous communication)

14.4.2 Asynchronous Communication

When the read latency is 0 clock cycles, the communication is asynchronous. Slave behaves as register file in this case. See Figure 27 and Figure 28 with timing diagrams of CP_CMD_READ and CP_CMD_WRITE operations. Note that CP_CMD_WRITE is always synchronous.

Protocol:

- In the first clock cycle the master asserts CP_CMD_READ or CP_CMD_WRITE request with address and control signals. Then master checks slave's `STATUS` signal. CP_ST_READY value on this signal means that the slave is able to accept new request. Otherwise, `STATUS` is set to CP_ST_BUSY.
- Operation is finished in the same clock cycle as operation request. In case of CP_CMD_READ operation, slave asserts data on `Q` signal and CP_RS_ACK, CP_RS_ERROR, CP_RS_UNALIGNED or CP_RS_OOR on `QRESPONSE` signal due to result of the operation. For CP_CMD_WRITE operation, master asserts data on `D` signal and slave responds with CP_RS_ACK, CP_RS_ERROR, CP_RS_UNALIGNED or CP_RS_OOR on `DRESPONSE` signal due to result of the operation.

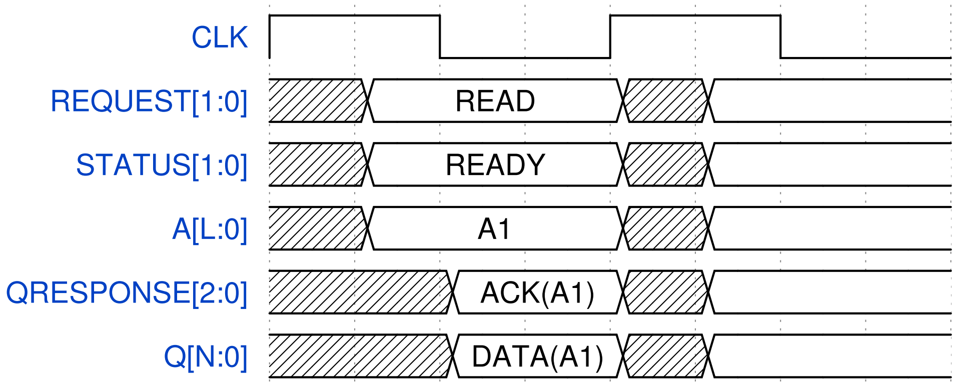


Figure 27: Asynchronous CP_CMD_READ operation

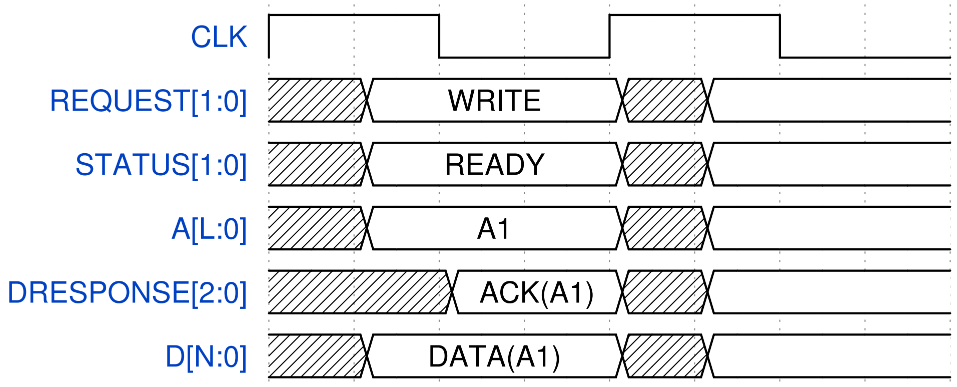


Figure 28: CP_CMD_WRITE operation (same for synchronous and asynchronous communication)

14.4.3 CP_CMD_NONE Request

The master can request CP_CMD_NONE. It means that the master does not want to do anything useful, so none operation is initiated. This request has no associated finish operation.

14.4.4 Use of the BI and BC signals

With `BI` signal, you can specify index of byte of data block from `D/Q` signal you want to process. For example if you have 32 bits wide data bus, you have 4 bytes and you can choose index 0, 1, 2 or 3. This is index of the first byte of data block. Signal `BC` then specifies number of requested bytes. Sum of `BI` and `BC` can't be higher than number of bytes of the data word. For example `BI = 2` and `BC = 2` is valid combination, but `BI = 1` and `BC = 4` is not (for 32 bits wide data word).

14.4.5 Slave Error Signaling

If the slave detects some error, there are 3 possible scenarios. In the first case we have a memory slave thath does not support unaligned data access. If you try to perform an operation on unaligned address, you get CP_RS_UNALIGNED on `QRESPONSE/DRESPONSE` signal. The second case is when you try to perform an operation on an address that is out of slave's address range. Then you get CP_RS_ OOR (Out of Range) response. Finally, if an unspecified error occurs, the slave asserts CP_RS_ERROR on `QRESPONSE/DRESPONSE`, because it can't finish pending request.

15 CO-SIMULATION

One of the usual designer’s needs is to create an ASIP simulator, optionally with debugging features, in the form of a component for the SoC simulation model. Such components can be modeled using the standardized IEEE SystemC simulation platform or other simulation platforms, such as Questa® Advanced Simulator. Hence, the designer can optionally select the SystemC option in the Codasip Studio and a SystemC component is generated. Then it can be easily integrated into the SoC simulation model. Note that components with DPI and C/C++ interfaces can be generated as well.

Each simulation platform needs a compatible plugin, which must have a proper interface. This interface is stored in a header file, which is delivered together with the simulation platform. Paths to these files must be provided to Codasip Studio.

This chapter is organized as follows:

15.1 Overview

Co-simulators (C/C++, SystemC, DPI) are wrappers around the Codasip Simulator (more precisely, around simulator's public interface) and provide for you an interface to control it, obtain and set data from and to it and handle bus transactions.

Due to language specifics, not every construction is available for all co-simulators. For info about supported features, go to the following chapters dealing with language specifics.

The Codasip Simulator has a public interface called `SimulatorInterface`. A description of this interface is in a file called `codasip_interface.h` which is installed together with generated co-simulators. The co-simulators are instancing a simulator and are using this interface for simulation control (loading an executable, starting simulation and more). You will not use this interface directly, since every co-simulator implements its own wrapper around it.

A quick overview of these wrappers:

- **C++** and **SystemC** use a wrapper called `Foreign` (installed header file `*_foreign.h` by means of inheriting it using C++ inheritance system
- **C** and **DPI** use the whole **C++ co-simulator** as a wrapper around simulator interface

15.2 Foreign interface

The generated co-simulation wrappers provide following interface methods.

15.2.1 C++/SystemC

These co-simulation wrappers inherit from the `Foreign` wrapper and thus it's interface methods are available.

`GetName`	Return instance name of the simulator.
`Reset`	Reset the simulator.
`GetCycleCount`	Return current cycle counter value.
`ClockCycle`	Execute clock cycle.
`Run`	Starts the whole simulation and let simulator handle clock cycles.
`ResourceRead`, `ResourceWrite`	Read from or write to a simulator resource. Returns true if given resource was found, otherwise false is returned.
`Register`	Register a custom debugger interface. Prototype of this interface can be found in the installed file `dbg_interface.h`. Method does not return a value.
`GetInterface`	Get the simulator interface. Available methods are listed in installed `codasip_interface.h`.
`LoadExecutable`	Load an executable into the simulator. The `path` parameter takes a relative path to an ASIP into which the application should load. `executable` parameter is a system path to application. `arguments` are parameters for the loaded application.

Interface methods `Reset`, `ClockCycle`, `Run` and `LoadExecutable` return the simulator status code enum (`codasip::simulator::SimulatorReturnCode`) defined in file `codasip_interface.h`.

15.2.2 C/DPI

Since DPI and C co-simulator wrappers do not expose the `Foreign` wrapper, their interface mimics it.

`*_constructor`	Construct new simulator instance. Return it's handle that will be used for following method calls (the `h` parameter).
`*_destructor`	Destruct the simulator instance.
`*_reset`	Resets the simulator.
`*_get_cycle_count`	Return current cycle counter value.
`*_clock_cycle`	Execute one clock cycle.
`*_run`	Start the whole simulation and let simulator handle clock cycles.
`*_get_last_error_msg`	Return last occurred error message from the simulator.
`*_load_executable`	Load an executable into simulator. The `simulator` parameter takes a relative path to an asip into which the application should load. `executable` parameter is a system path to application.
`*_get_name`	Return instance name of the simulator.
`*_get_`***`id`***, `*_set_`***`id`***	Get or set value of the port.

Interface methods from previous table, which have no description about their return value, return the simulator status code (`codasip_sim_codes_e`).

15.2.3 Functional verification specifics

In addition to DPI interface methods, the one for function verification adds following.

`*_get_RST`, `*_set_RST`	Get or set value of the reset signal.
`*_get_`***`address_space`***	Get value from the address space on `addr`.
`*_get_`***`register_file`***	Get value from register file on `addr`.

And for interfaces (all of the following methods return data from the last interface access).

`*_REQUEST0`	Request signal.

`*_A0`	Address signal.
`*_BI0`	Sub-block index.
`*_BC0`	Sub-block count.
`*_RESPONSE0`	Response signal.
`*_Q0`	Data.
`*_D0`	Data.

15.3 Bus transactions

Bus transactions are handled as callbacks that are invoked by the Codasip Simulator (the processor model). So when you call a method of CodAL interface, the callbacks of co-simulator are also called.

Callback definitions are in the generated co-simulator files. For an interface identified as `if_fetch`, there will be methods with prototypes:

```
void if_fetch_blocking_cb(ForeignPayload&);
void if_fetch_request_cb(ForeignPayload&);
void if_fetch_finish_cb(ForeignPayload&);
```

15.3.1 Foreign Payload

`ForeignPayload` represents a payload of a bus. This payload is passed by reference to all bus transaction callbacks (blocking, request and finish). Its interface can be found in `codasip_interface.h`.

15.3.2 Commands description

Table 29: List of payload commands

CodAL	**Co-simulator**	**Description**
`load`	`CP_CMD_LOAD`	Load an application
`dread`	`CP_CMD_DREAD`	Debug read
`dwrite`	`CP_CMD_DWRITE`	Debug write
`read`	`CP_CMD_READ`	Reading
`write`	`CP_CMD_WRITE`	Writing
`invalidate`	`CP_CMD_INVALIDATE`	Mark cache line to be invalidated or deleted

invalidate_all	CP_CMD_INVALIDATE_ALL	Mark whole cache to be invalidated or deleted
flush	CP_CMD_FLUSH	Force cache line to be written to the higher level of cache hierarchy
flush_all	CP_CMD_FLUSH_ALL	Force whole cache to be written to the higher level of cache hierarchy

For more information on interfaces, look at chapter "Codasip Local Bus" on page 304.

15.4 Ports

Output ports have asociated callbacks similar to the callbacks of a bus transaction. When a value of a an output port of a processor is written, a callback is invoked.

Callback definitions are in the generated co-simulator files. For a port identified as `p_output`, there will be methods with prototypes:

```
void p_output_write_cb(const uint8_t* data, const int len);
void p_output_dwrite_cb(const uint8_t* data, const int len);
```

15.5 SystemC specifics

SystemC co-simulation is based on the OSCI Transaction-Level Modeling Standard, version 2.0 (IEEE 1666-2011). A subset of the this standard is used, namely core interfaces, initiator/target socket and a generic payload. A so-called *loosely-timed model* is in place.

Blocking callback's behavior is pre-implemented in generated files. The implementation works with SystemC transaction payload.

15.5.1 load, dread, dwrite

In these methods, pre-generated implementation consists of setting data to a generic payload and a `transport_dbg`.

15.5.2 read, write

Pre-generated implementation consists of synchronization with the Simulator, setting data to a generic payload and a blocking `b_transport`.

15.6 C++ specifics

C++ co-simualtor adopts the same concept as the SystemC one. Implementation of the bus transaction callbacks is left on you.

15.7 C specifics

Since C++ inheritance cannot be used in C co-simulator, bus transaction callbacks and callbacks of output ports are handled differently.

In the `*_c.h` file, there are two declarations of structured types. `payload_t` and `callbacks_t`. The `payload_t` is a bus transaction payload, which has the same attributes as `ForeignPayload`. `callbacks_t` contains function pointer declarations for callbacks of bus transactions and output ports. Set these pointers to your defined functions and to **`NULL`** value for callbacks you do not intend to use.

15.8 DPI specifics

DPI co-simulator does not support bus callbacks.

16 HIGH LEVEL SYNTHESIS

When an architecture design is stable enough, a synthesizable RTL ASIP representation can be generated. It is possible to generate test benchmarks for the architecture, to generate asserts into the RTL description, or optionally, to generate support for a JTAG/Nexus debugging interface. Note that the generated synthesizable RTL is well proven by the 3rd party ASIC and FPGA synthesizers.

Additionally, equivalence between the simulator and the hardware representation has to be ensured. Bear in mind that the behavior of this simulator should be the same as the behavior of the real hardware. In the terms of the Codasip Studio, this equivalence is guaranteed by the fact that all principles and algorithms are based on formal models and are well proven. Moreover, the simulator and hardware generators use the same algorithms for generation.

This chapter is organized as follows:

16.1 Hardware representation generation

When the architecture design is stable enough, the hardware description on the register transfer level (RTL) can be generated. It is also possible to generate test benchmarks for the architecture.

Generated VHDL files (naming for all supported language representations, Verilog/SystemVerilog/VHDL, adopts the same concept, except for file extensions):

- *name-of-project.vhd*: contains the top design entity (VHDL). In case of Design level project, the entity contains all instances and interconnection between them as specified in the model. It can be instances of used components, ASIPs, memories, caches, and buses. The ASIP entity contains ASIP core, bus interfaces, CodAL ports, and optionally Nexus client. Also in case of JTAG/Nexus debugging, the top entity always contains the Nexus port.

- *name-of-asip_core_t.vhd*: contains components of a pipeline control, reset unit, decoders, functional unitsand resources (bus interfaces are excluded and placed in the ASIP top entity). The purpose of this entity is to connect the components with each other.

- *name-of-asip_core_name-of-pipeline_t.vhd*: contains the entity having responsibility of clearing and stalling the pipeline.

- *name-of-asip_core_CodAL-module_name-of-functional-unit_t.vhd*: contains the entity of one functional unit. Each functional unit is described in a distinct file. The functional unit performs the computation described in the `semantics` section of the `event` in the CodAL description. The name of the functional unit may be formed from multiple parts. The first part is the name of the `event`, the second part is the name of instanc, and optionally, the instance history which should create unique id.

- *d_ff_plain_t.vhd*: contains the generic entity of a register. This type of register has the following ports: `Clock`, `D`, `Q`. A write operation is synchronous. `Reset` and `Write enable` ports are optional, based on description in CodAL. Activation level of the `Reset` and whether it is synchronous is set using generics (these two settings are set per-project in configuration).

- *d_ff_pipeline_t.vhd*: contains the generic entity of a pipeline register. This register has ports similar to the plain register type (with equivalent rules applied) and in addition has a `Clear` and `Stall` ports. The `Clear`, `Stall` and write operations are synchronous. `Clear` sets the value of the register to the default value (a state equivalent to the state right after reset). `Stall` forbids the register to write new value into it. `Clear` has higher priority than `Stall`. `Stall` has higher priority than `Write`.

- *name-of-asip_core_name-of-register-file_t.vhd*: contains the entity with a register file. The file name is equal to the register file name declared in the ASIP model.

- *codasip_cache_t.vhd*: contains implementation of Codasip caches.

- *codasip_memory_name-of-memory_t.vhd*: contains the entity with a memory element. The file name is equal to the memory element name declared in the Design Level model.

16.1.1 ASIP entity

The main entity (in *name-of-asip.vhd*) provides communication means with the ASIP. It contains ports explicitly declared in the CodAL model (see CodAL Language Reference Manual). Additional ports in the interface are provided for clock and reset signals. There are also interfaces present (on a hardware level, they are a group of ports).

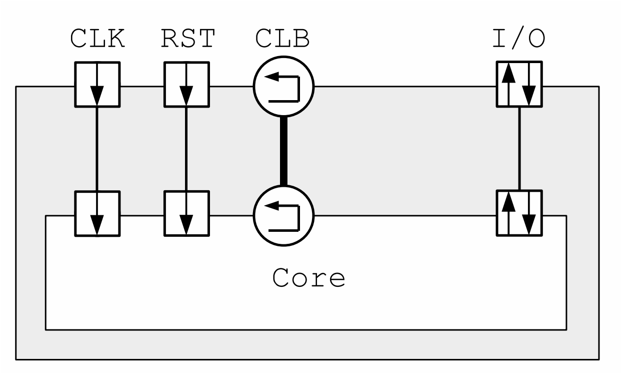


Figure 29: View of processor entity

16.1.2 ASIP core entity

The core entity (in *name-of-asip_core_t.vhd*) contains components of function units, pipeline control, reset unit, CodAL modules, decoders and resources. The interface of this entity contains signals for:

- access to all bus interfaces
- access to all ports declared in the ASIP model
- data transfer from decoder or pipeline control to function units – parameters of computation (e.g. attributes), activation, etc.
- reset
- clock

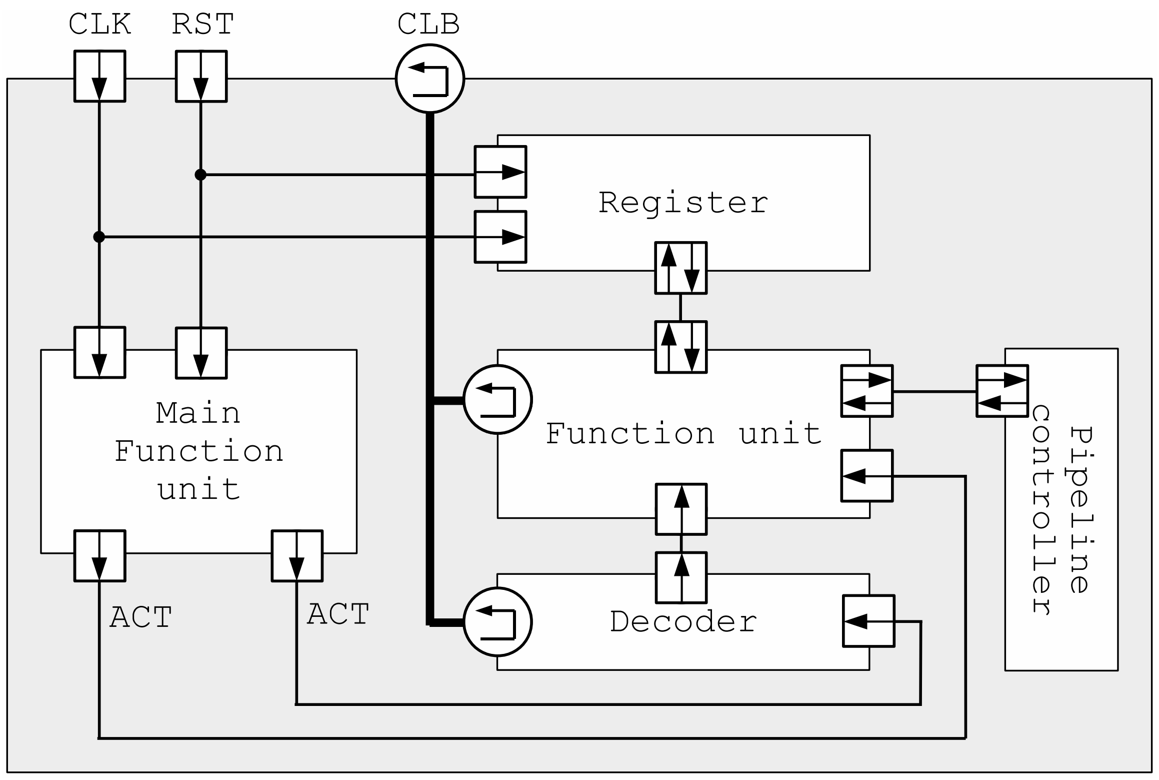


Figure 30 : Processor core in detail

16.1.3 Instruction decoder entities

Instruction decoders are generated on the basis of `binary` and `decoders` sections. The identification of the instruction decoder consists of the name of the decoder's instance and the decoder itself that is used in the `decoders` section. The decoder groups together `element`s (and their `semantics`, `return` and `timing` sections).

16.1.4 Functional units

Functional units (*FU*) are created from functions and events. For each `event` and its instance, one *FU* is created. It means that if you instantiate one `event` in two or more other `events` or `elements` using the same name, then only one *FU* is created. If you use different names for instances, several *FU*s are created. In case of a function, more instances are generated when the function is declared `inline`.

Functional units are one of these types:

- FU for the `semantics` section of a CodAL `event`
- FU for C function

Each function unit is a synchronous finite-state machine. Most of the units have only one state (i.e., they perform computation immediately within current clock cycle, since they contain only combinational logic). If a functional unit needs more clock cycles to finish it's computation (e.g., crc32 computation), then it is in the default state after the restart or after the finish of the computation. The FU does not change state or return any value in the default state. When the *FU* is activated, the state is changed to computational one. The state is changed each rising edge of the clock cycle.

An *FU* consists of the following parts:

- interface: input and output ports from/to resources, pipeline control or decoder
- optionally state register: contains the value of the current state
- optionally state machine: decides the next state according to the current state and the transition logic
- data-path: contains logic and output logic

The following figure shows the schema of a functional unit. The Statements represents data-path. The control unit has state register within itself. Note that the control unit is not present if the functional unit is purely combinational logic.

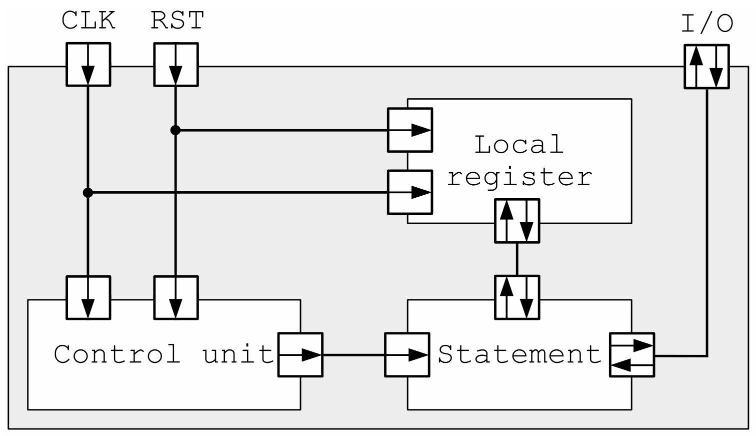


Figure 31: Function unit in detail

16.1.5 C Language Code Transformations

C language code used in `semantics` sections or in C functions is always being transformed (normalized). Here is the list of all transformations.

- `for` to `while` transformation

```
int a;
for (a = 0; a < 10; a++) {
    ...
}
```

```
int a;
a = 0;
while (a < 10) {
    ...
    a++;
}
```

- Postfix and prefix transformation

```
b = a++;
c = ++d + e
```

```
b = a;
a = a + 1;
d = d + 1;
c = d + e;
```

- Multi-assignment

  If the source of an assignment is a constant, then it is propagated. See the following example.

```
a = b = 4 * 5;
```

```
b = 20;
a = 20;
```

- The situation is different, when the target is composed of an expression. Members of the expression can be addressable resources, so the generator cannot perform propagating, but it has to generate assignments in the proper order. If it did the propagation, then there might be a problem with a read data-port's handling.

```
x = y = e + g * f;
```

```
y = e + g * f;
z = y;
```

- Assignment with operator

```
d += a * b[c];
```

```
d = d + (a * b[c])
```

16.1.6 Timing of C Language

Body of a functional unit is created from a description in the C language. Because a C code is sequential, the hardware generator must perform automatic parallelization/timing of that code. There are several rules applied during the timing. Each resource has a predefined read and write latency. The latency information is used for delaying some parts of the C code which are dependent on the results from the previous parts of the C code. Detailed information can be found in the following table.

Table 30: Timing of C Language

Resource	Read latency	Write latency
Signal	0	0
Port	0	0
Register/Register file	0	1
Interface	1	1

Local variables can be transformed to a signal, a register, or a register file. The transformation to a signal is a default behaviour.

Another important feature of resources is a number of read and write ports. If the resource is not addressable, then there is no limit set to the reading within one clock cycle (i.e. the resource can be read by unlimited number of functional units), but there can be only one write per clock cycle. In the case of addressable resources (register file), the number of read and write ports is obtained from the register file description. For CodAL interfaces there is only one data port, and only one call of given access method is allowed per clock cycle. It should be noted that if the local variable is transformed to a register file, then there will be two reading ports and one write port. The number of read and write ports of the transformed register field cannot be modified.

Warning: It should be noted that any resource can have only one writing data-port currently.

The generator checks data dependencies of each access to a resource or local variable. There are several types of data dependencies and conflicts:

- Read dependence (R),
- Write dependence (W),
- Conflict Read After Read (RAR),
- Conflict Read After Write (RAW),
- Conflict Write After Read (WAR),
- Conflict Write After Write (WAW).

The dependencies *R* or *W* mean that the resource or local variable is read by a functional unit or some data is written to this resource. The conflicts *RAR*, *WAR* and *WAW* are important only for register file. The conflict *RAW* is important for all resources. Depending on the set of data dependencies and conflicts, the functional unit can be combinational or sequential. An automatic parallelization starts with a presumption that the time needed for a computation should be as small as possible. Therefore, the generator tries to do as many C statements as possible within one clock cycle. It also tries to do C statements as soon as the theirs dependencies are satisfied, so-called *ASAP* (As Soon As Possible) scheduling.

16.1.6.1 Waiting States

To handle conflicts described in the previous sub-chapter, the generator infers waiting states in appropriate locations. These waiting states are in reality states of a final state machine, effectively making the unit a sequential one.

With the assumption that local variables are handled as signals, the following example shows a *WAW* conflict (Write After Write) to a register named `test`. This conflict exists since there are two statements in which a value of the register `test` is assigned.

```
register bit[2] test { default = 0; };
uint1 test_waiting_states()
{
    uint1 result;
    uint2 prev_test;

    prev_test = test;
    test = 1;
    result = test;
    codasip_assert(result == prev_test, "Result must be of value from previous clock cycle.");
    // Waiting clock cycle inferred here. Following statements executed in next cycle.
    test = 2;
    result = test;
    codasip_assert(result == 1, "Result must be of value 1");

    return result;
}
```

The first assignment sets the value of the register to `1`. After this, the local variable `result` is set to contain value of the register. But since a write latency of the register is 1, the value has not propagated yet. As a result, the local variable has the value as the value written in previous clock cycle. This is displayed using the `codasip_assert` built-in function.

Then there is an assignment setting a value of the register to `2`. That is a conflict, since you cannot write value to a register twice in one clock cycle. This conflict is resolved by adding a new state that will be handled in a following clock cycle. Since now, a new state exists and statements following the first assert will be executed inside of it. In the new state, the value `1` from previous assignment is already stored and another write to a local variable stores value `1`. This is again displayed with a call to `codasip_assert`.

Another conflict occurs when there are reading to/writing from statements more times than the resource can handle within one clock cycle. Waiting cycles are inserted so that only a limited count of reading/writing is performed within the same cycle. Look at the following example. There is a register file with three read data-ports and one write data-port. The functional unit demands four readings and because of that, the last reading is separated from the previous ones by waiting a clock cycle. The computation is therefore performed in two clock cycles.

```
register bit[32] rf {
    dataports = {3, 1};
    size = 32;
};
...
semantics {
    int32 a, b, c, d;
    ...
    a = rf[op1];
    b = rf[op2];
    c = rf[op3];
    // The following statement will be performed in a following clock cycle
    d = rf[op4];
    ...
};
```

Warning: The address of an addressable resource is not checked by the generator, so if there is a read and a write to the same memory location, the result is dependent on a final realization of the memory.

16.1.6.2 Conversion from Variables to Registers

Now, let's look in detail on how the local variables are handled and in which situation additional signals or registers are constructed. As mentioned before, the timing process of each functional unit can change local variables from signals into registers or it can create new local registers and/or signals. Local variables are implicitly handled as signals.

The conversion from local variables into registers can be caused by following cases.

Conflict write after read within the statement

Generally, if the analysis process of the data dependencies finds write after read conflict inside a single statement, it must change the signal into a register. A typical example is an increment of a variable in `for` loops. In the following example the variable `i` will be handled as a register.

```
semantics {
    int i;
    for ( i = 0; i < 32; ++i ) {
        ...
    }
}
```

But it can be any other expression where a value of the variable is used to compute a new one:

```
semantics {
    ...
    x = x * 2;
    ...
}
```

Preserving values among iterations

If a value of a variable, which wasn't previously set inside a loop is read inside the loop, it must, in principle, become a register in order to store its value from previous iteration or before entering the loop itself. In order to use local signals inside loops, their values must be written before it can be read. In the following example the variable `y` must be changed from a signal to a register.

```
semantics {
    int x, y;
    x = 0;
    y = a;
    while ( y >= b ) {
        int tmp;
        tmp = y - b;
        y = tmp;
        ++x;
    }
}
```

Multi-port register file access

If there are more read accesses to a register file inside a single statement than the read data-ports resource actually has, the statement must be transformed. There are local registers created for each access which has no read data-port available to use. For example, when reading four values from a register file with only two read data-ports, two local registers will be created. Look at the following example.

```
// register file definition
register bit [32] regs {
    dataports = {2, 1};
    size = 32;
};

// original statement
semantics {
    a = regs[1] + regs[2] + regs[3] + regs[4];
}

// pseudo-code of the transformation
semantics {
    uint32 codasip_local_register_1;
    uint32 codasip_local_register_2;

    codasip_local_register_1 = regs[3];
    codasip_local_register_2 = regs[4];
    a = regs[1] + regs[2] + codasip_local_register_2 + codasip_local_register_1;
}
```

16.1.7 Resources

The hardware description allows specifying the following resources: registers, register files, memories, caches and buses. The function units can access all types of resources, but the memories, caches and buses are accessible only via the CodAL interface.

The multi-port register fields are also supported. Specific read/write port is assigned during hardware generation based on the **semantics** sections processing. For instance, if the register file has two ports for reading, there are two *FU-S*s, and each of them contains only one reading from the register file, then both ports are used (i.e., two reads are performed in one clock cycle and there is no collision). In the case that one *FU-S* contains two readings from the register file and the second *FU-S* contains one reading, the first *FU-S* occupies both ports and the second *FU-S* is connected to the first reading port again. Thus, if both *FU-S*s are activated at the same time, collision occurs. Note that this collision is detected by CA simulator but not in RTL.

The following example shows a simple register and a pipeline register. Both of them are the same for all models. The VHDL entity takes two generic parameters denoting register bitwidth and its value after reset. The pipeline register, which is shown in the lower part, has two extra ports - `STALL` and `CLEAR` ports, which are used by the pipeline controller. The `CLEAR` has bigger priority than `STALL`.

Example 160: Simple and pipeline register

```
entity d_ff_plain_rst_we_t is
        generic(
               bit_width : natural := 8;
               default_value : std_logic_vector := "00000000";
               reset_level : std_logic := '0';
               reset_sync : boolean := false
        );
        port (
                CLK: instd_logic;
                WE0: instd_logic;
                RST: instd_logic;
                D0: instd_logic_vector(bit_width - 1 downto 0);
                Q0: outstd_logic_vector(bit_width - 1 downto 0)
        );
endentity;

entity d_ff_pipeline_rst_we_t is
       generic (
               bit_width : natural := 8;
               default_value : std_logic_vector := "00000000";
               reset_level : std_logic := '0';
               reset_sync : boolean := false
       );
       port (
               CLEAR : instd_logic;
               CLK : instd_logic;
               D0 : instd_logic_vector(bit_width-1 downto 0);
               Q0 : outstd_logic_vector(bit_width-1 downto 0);
               RST : instd_logic;
               STALL : instd_logic;
               WE0 : instd_logic
       );
end entity d_ff_pipeline_rst_we_t;
```

Register files are generated from the ASIP description (i.e. unlike simple registers, they differ for ASIP descriptions). So, let's assume that we have register file with two read ports and one write port. The next example depicts such a register file. The dataport is denoted by number after name of the port (e.g. `Q0` vs. `Q1`).

Example 161: Register file

```
entity codasip_urisc_ca_core_rf_gpr_t is
    port (
        CLK : in std_logic;
        D0 : in std_logic_vector(15 downto 0);
        Q0 : out std_logic_vector(15 downto 0);
        Q1 : out std_logic_vector(15 downto 0);
        RA0 : in std_logic_vector(3 downto 0);
        RA1 : in std_logic_vector(3 downto 0);
        RE0 : instd_logic;
        RE1 : instd_logic;
        WA0 : in std_logic_vector(3 downto 0);
        WE0 : in std_logic;
);
end entity codasip_urisc_ca_core_rf_gpr_t;
```

Resources like memories, caches, and buses can be accessed using CodAL interfaces only. The interface always contains the same set of signals, regardless of if they are connected to the bus or not.

16.2 JTAG/Nexus On-chip Debugger Generation

Based on project configuration, the hardware generator is capable of generating full-feature on-chip debugger logic with JTAG interface. All modules of the on-chip debugger are located inside of the ASIP with 5-wire IEEE JTAG interface.

Following files are generated on the ASIP's level:

- *name-of-asip_nexus_port_t.vhd*: contains definition of the Nexus port. The Nexus port interconnects all Nexus clients to one JTAG interface and ensures proper clock synchronization between JTAG and ASIP clock domain.
- *name-of-asip_nexus_port_jtag_tap_t.vhd*: contains definition of JTAG TAP (Test Access Port) specified by IEEE Std 1149.1-2001.
- *name-of-asip_nexus_port_jtag_sync_t.vhd*: contains definition of JTAG clock domain synchronization entity.
- *name-of-asip_nexus_port_jtag_tck_domain_t.vhd*: contains definition of resources used in JTAG clock domain of the Nexus port.
- *name-of-asip_nexus_port_jtag_clk_domain_t.vhd*: contains definition of resources used in ASIP clock domain of the Nexus port.
- *name-of-asip_nexus_client_t.vhd*: contains definition of Nexus ASIP client. The purpose of this client is to control ASIP's execution and provide debugging information about current state.
- *name-of-asip_nexus_client_bw_module_t.vhd*: contains definition of Nexus breakpoint/watchpoint module.
- *name-of-asip_name-of-interface-with-nexus-support_t.vhd*: contains a CodAL interface enhanced with Nexus memory access support. The purpose of this Nexus client is to provide read and/or write access to memory through ASIPs interface using JTAG/Nexus interface. Such interface is often used to load application into the design and reading of variables during debug process.

16.2.1 Schema

Following schema shows a simplified generated hierarchy of ASIP with Nexus on-chip debugger and IEEE JTAG interface.

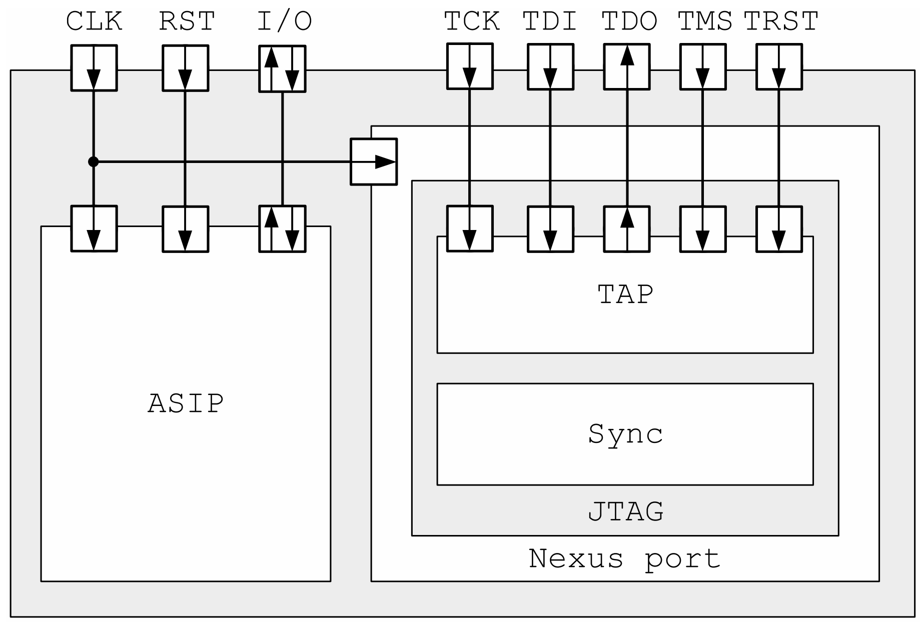


Figure 32 : Simplified JTAG/Nexus hierarchy

If multiple ASIPs with the JTAG interface are used on top level, then all ASIPs are connected in one JTAG chain (TDO of previous ASIP is connected to TDI of next ASIP). The order of ASIPs in the chain is defined by the hardware generator.

16.2.2 JTAG/Nexus settings

User can define all basic information about JTAG TAP in the project configuration. The following JTAG parameters are supported:

- JTAG TAP identification
- length of the instruction registers
- code of the JTAG instructions NEXUS_ACCESS, NEXUS_CONTROL and IDCODE instruction.

There are some extended settings of JTAG/Nexus debugger:

- support of embedded FPGA BSCAN cell
- support of clock gating control logic to avoid setup/hold time issues during resetting of the ASIP through Nexus logic
- support of DFT scan mode for ASIP DFFs

When the embedded FPGA BSCAN cell is enabled, then a special module is placed to the top module (ASIP or platform). The module contains one BSCAN cell and logic that translates signals from the BSCAN cell to the IEEE JTAG interface. A valid target device

(FPGA) must be selected in **RTL** page in the project configuration. Currently the following FPGAs are supported:

- Xilinx Artix-7
- Xilinx Virtex-5
- Xilinx Spartan-3

16.2.3 Clock domains relationship

The ASIP clock domain and JTAG clock domain are asynchronous with each other. The clock domains synchronization logic is based on the parallel synchronization technique. This technique allows to use frequency of JTAG clock slower or equal to ASIP clock frequency.

16.3 Nexus implementation

The Nexus standard defines interface, set of registers and protocol to implement a control instrumentation allowing application debugging.

The debug support is based on the definition and implementation of Nexus standard recommended registers. The Codasip Nexus support implements the following set of registers (Nexus Recommended Registers – NRR):

NRR	Bitwidth [bits]	Index (decimal)	Read/Write
Device ID (DID)	32	0	R
Client Select Control (CSC)	8	1	R/W
Development Control (DC)	32	2	R/W
Development Status (DS)	32	4	R
Read/Write Access Control/Status (RWCS)	32	7	R/W
Read/Write Access Address (RWA)	vendor defined, architecture dependent	9	R/W
Read/Write Access Data (RWD)	vendor defined, architecture dependent	10	R/W
Breakpoint/Watchpoint Control (BWC) (2)	32	22-23	R/W
Breakpoint/Watchpoint Control (Reserved - 6)	32	24-29	—
Breakpoint/Watchpoint Address (BWA) (2)	vendor defined, architecture dependent	30-31	R/W

Breakpoint/Watchpoint Address (Reserved - 6)	vendor defined, architecture dependent	32-37	—
Breakpoint/Watchpoint Data (BWD) (2)	vendor defined, architecture dependent	38-39	R/W
Breakpoint/Watchpoint Data (Reserved - 6)	vendor defined, architecture dependent	40-45	—
Vendor defined[1]	vendor defined	64-127	—

The color distinguish the following cases:

- full implementation
- partial implementation
- vendor defined implementation
- not implemented

The description of registers with supported functionality is as follows:

Table 31:Supported functionality in the Development Control Register (DC).

Bit Number	Field Name	Description
24	RST	User reset of ASIP/platform
13	DBE	DBE - Debug Enable (Class 4)
		0 = Debug mode disabled
		1 = Debug mode enabled
12	DBR	DBR - Debug Request (Class 4)
		0 = Exit debug mode
		1 = Request debug mode
8	SS	SS - Step Enable (Class 4)
		0 = Single-step disabled
		1 = Single-step enabled

Table 32: Supported functionality in the Development Status Register (DS).

Bit Number	Field Name	Description
15-8	BP7-0	BPn - Breakpoint Status
		0 = No breakpoint
		1 = Breakpoint occurred

[1]Defined by the configuration in the Codasip Studio.

6	RSTS	RSTS - Reset Status (detection of external reset)
		0 = Processor not reset
		1 = Processor reset since last DS Register read
5	DBS	DBS - Debug Status
		0 = Processor not halted
		1 = Processor halted in debug mode
2	HWB	HWB - Hardware Breakpoint Status
		0 = No hardware breakpoint
		1 = Hardware breakpoint
1	SWB	SWB - Software Breakpoint Status
		0 = No software breakpoint
		1 = Software breakpoint
0	SSS	SSS - Single-Step Status
		0 = Processor not halted
		1 = Processor halted in debug mode after single step

Table 33: Supported functionality in the Read/Write Access Control/Status Register (RWCS).

Bit Number	Field Name	Description
31	AC	AC - Access Control
		0 = End access
		1 = Start access
30	RW	RW - Read/Write
		0 = Read access
		1 = Write access
15-2	CNT	CNT - Access Count
		hhhh = Number of accesses of word size SZ
1	ERR	Last access generated an error
0	DV	Data valid in RWD

Table 34: Supported functionality in the Breakpoint/Watchpoint Control Register (BWC).

Bit Number	Field Name	Description

31-30	BWE	BWE - Breakpoint/Watchpoint Enable
		00 = Disabled
		01 = Breakpoint enabled
		10 = Reserved
		11 = Watchpoint enabled
29-28	BRW	BRW - Breakpoint/Watchpoint Read/Write Select
		00 = Break on read access
		01 = Break on write access
		10 = Break on any access
		11 = Reserved
17-16	BWO	BWO - Breakpoint/Watchpoint Operand
		00 = Disabled value
		10 = Compare with BWA value
		01 = Compare with BWD
		11 = Compare with BWA and BWD value

The basic commands like enabling debug mode, breaking, stepping, and a reading register value are performed by reading and writing particular registers. The communication protocol is simple — using two data words sent via JTAG. First word represents address (i.e. register index on the most significant bits b7-b1, plus an additional least significant bit b0 determining read or write operation; 0 means read and 1 means write operation). The second word carries the data (read or written).

The following sections depict a couple of use cases. The register index is always followed by the data, so the pair of words is represented by a single table line in the following examples. The length of data is dependent on the particular register accessed.

16.3.1 Setting a HW breakpoint

First JTAG NEXUS_ACCESS instruction is sent (value is defined in the project configuration) followed by a JTAG data.

Table 35: Setting a HW breakpoint (JTAG data sequence).

Register Name (Index)	Data	Description
CSC (0x1)	<defined_by_rtl_generator>	The register index represents register CSC (client select) and the data the client number.
BWA (0x1E)	<breakpoint_address>	The index of BWA register is written (up to 8 registers are supported) followed by a breakpoint address.

BWC (0x16)	BWE: 0b01	The index of BWC register is written (enables breakpoint) and its bits BWE are set to value 0b01.
DC (0x2)	DBE: 0b1	The index of DC register is written and its bit DBE is set to 0b1.

16.3.2 Instruction stepping

First JTAG NEXUS_ACCESS instruction is sent (value is defined in the project configuration) followed by a JTAG data.

Table 36:Instruction stepping (JTAG data sequence).

Register Name (Index)	Data	Description
CSC (0x1)	<defined_by_rtl_generator>	The register index represents register CSC (client select) and the data the client number.
DC (0x2)	DBE: 0b1, SS: 0b1	The index of DC register is written and both its bits DBE and SS are set to 0b1.
DS (0x4)	SSS bit reading	The index of register DS is written and its bit SSS is then read until it is set.

16.3.3 Writing to Memory

First JTAG NEXUS_ACCESS instruction is sent (value is defined in the project configuration) followed by a JTAG data.

Table 37: Writing to memory (JTAG data sequence).

Register Name (Index)	Data	Description
CSC (0x1)	<defined_by_rtl_generator>	The register index represents register CSC (client select) and the data the client number.
RWA (0x9)	<address>	The register index represents register RWA. The particular address follows.
RWD (0xA)	<data_to_be_written>	The register index represents register RWD. The data to be written follows.
RWCS (0x7)	AC: 0b1, RW: 0b1	The register index represents register RWCS, bits AC are set to 0b1 and RW to 0b1.

16.3.4 Reading from Memory

First JTAG NEXUS_ACCESS instruction is sent (value is defined in the project configuration) followed by a JTAG data.

Table 38:Reading from memory (JTAG data sequence).

Register Name (Index)	Data	Description
CSC (0x1)	<defined_by_rtl_generator>	The register index represents register CSC (client select) and the data the client number.
RWA (0x9)	<address>	The register index represents register RWA. The particular address follows.
RWCS (0x7)	AC: 0b1, RW: 0b0	The register index represents register RWCS, bits AC are set to 0b1 and RW to 0b0.
RWCS (0x7)	DV bit reading	The register index represents register RWCS, bit DV is read - if it is set it is possible to read data.
RWD (0xA)	<data_to_be_read>	The register index represents the RWD register, the read data follows.

Access to ports is not supported in the JTAG implementation.

16.4 Testbench generation

The hardware generator can produce simple testbench files (Verilog/SystemVerilog/VHDL). The testbench is well prepared (clock and reset are generated, unit under test is also connected), so user should only change a stimuli. There is also a simple TCL startup script to launch simulation of the testbench using Questa® Advanced Simulator, Riviera-PRO. The script compiles the design, performs some customizations of the wave-form, and starts the simulation itself.

16.5 Synthesis script generation

Another capability of the hardware generator is generation of basic synthesis script templates. This includes basic timing constraints as well. Generated scripts should be prepared to run out-of-box, but they should be completed and enhanced manually by the hardware engineers. For example the technology library should be configured for ASICs, for FPGAs pin location should be specified.

Following synthesizers are supported:

- Cadence Encounter RTL Compiler
- Cadence Genus Synthesis solution

- Xilinx ISE
- Xilinx Vivado

16.6 UVM verification generation

More details about verification, generating, and running of UVM verification environments and their description can be found in the chapter "Verification Guide" in the *Codasip Studio User Guide*.

16.7 Visualisation

The hardware generator can produce several images projecting generated functional units. Each functional unit is visualized separately. The images are stored next to the RTL sources in the `report` directory in the Codasip Studio. The image contains RTL graph with transitions. Each state can have several statements assigned, which can be also conditioned (conditions from **`if`**/**`switch`** statements in **`semantics`** section). You can see which statements are performed in which clock cycle. There is also a file aggregating information about computation requirements of each FU in terms of time, number of local registers or signals. The file is also located next to the RTL sources in a `report` directory. There you can see how many clock cycles are needed by a particular functional unit.

16.8 RTL aware CodAL modeling

There are a few caveats and recommendations regarding cycle-accurate modeling in relation to the hardware generator. As a result of following this guide, you can reach a better and cleaner resulting VHDL/Verilog/SystemVerilog code together with its increased quality (less inferred resources and/or use of more optimal constructions).

16.8.1 Multipliers

There are several ways to describe multiplication operation in CodAL to ensure correct bit-width and sign of operands and result. Following example describes concrete casts for specific use-cases.

Example 162:Multiplication

```
int mult;   // 32-bit result
int aa, bb; // 32-bit operands
short a, b; // 16-bit operands

// 32x32 with 32-bit result
mult = aa * bb;
mult = (int)(int)a * (int)(int)b;
// 16x16 with 16-bit result sign extended to 32-bit
```

```
mult = a * b;
mult = (int)(a * b);
// 16x16 with 32-bit result
mult = (int)a * b;
mult = a * (int)b;
mult = (int)a * (int)b;
```

Do not use shift-and-add for a constant multiplication. Use multiply operator instead. This way it is easier for a synthesis tool to recognize such unit.

16.8.1.1 Signed/Unsigned

When there is a need for both signed and unsigned multiplier, it is preferable to use only one instance of a multiplier that is programmable. Following example is a non-optimal code inferring two instances of multipliers.

Example 163: Non-optimal programmable multiplier

```
uint8 opc;
uint16 a, b;
uint32 mult;

if (opc == EX_MULS)
{   // signed 16x16 with 32-bit result multipler
    mult = (int32)(int16)a * (int32)(int16)b;
}
else
{   // unsigned 16x16 with 32-bit result multipler
    mult = (uint32)(uint16)a * (uint32)(uint16)b;
}
```

To generate only one instance of the multiplier capable of both signed and unsigned multiplication, you need to add one bit to the operands. Following example is implementing the idea.

Example 164: Optimal programmable multiplier

```
uint8 opc;
uint16 a, b;
int17 aa, bb;
uint32 mult;

// sign expand of inputs
aa = (opc == EX_MULS) ? (int17)(int16)a : (int17)a;
bb = (opc == EX_MULS) ? (int17)(int16)b : (int17)b;
// using single 17x17 operator with 32-bit result
mult = (int32)aa * (int32)bb;
```

Be aware that the first (non-optimal) approach might be superior for FPGAs since it only switches between results and also the operation has narrower inputs.

16.8.1.2 Pipelined multiplier

It is recommended to store result of a multiplication into a local register that is to be re-timed into the multiplier (although some synthesizers might cope with a global one). Following example shall infer a pipelined multiplier.

Example 165: Pipelined multiplier

```
// global signal holding value of pipelined multiplier (value from previous clock cycle)
signal bit[WORD_W] s_wb_mult_W;

// pipelined multiplier
event ex_mult : pipeline(pipe.EX)
{   // local register definition, must be without write-enable signal
    // may contain synchronous reset, not asynchronous
    register bit[WORD_W] r_pipe_mult { reset = false; write_enable = false; };

    semantics
    {   // drive global signal to provide access to result from previous clock cycle
        s_wb_mult_W = r_pipe_mult;
        // pipelined multiplier
        r_pipe_mult = r_ex_regA * r_ex_regB;
    };
};
```

16.8.2 Switch/Multiplexer

Use **`switch`** statement to infer multiplexer. You should also solely switch a local variable, since switching a CodAL register may lead to conditioning of not only data, but also of a write-enable signal. Following code leads directly into a multiplexer construction.

Example 166: Multiplexer construction

```
// declare local variable
int rd_mem_rw_int;
// assign a value in switch (in all branches)
switch (opc)
{
    case OPC6_STB:
        rd_mem_rw_int = MEM_STB; break;
    case OPC6_ST:
        rd_mem_rw_int = MEM_ST; break;
    default:
        rd_mem_rw_int = MEM_NO_RW; break;
}
// assign global resource
r_rd_mem_rw = rd_mem_rw_int;
```

Avoid using nested branches. Nested branches break multiplexer pattern matching algorithm. It is then up to a synthesizer to decide, whether or not to infer multiplexer. Following example will not likely be synthesized as a multiplexer.

Example 167:Nested switch

```
int rd_mem_rw_int;
// first condition
if ( r_rd_mem_rw != 0 )
{   // second (nested) condition
    switch (opc)
    {
        case OPC6_STB:
            rd_mem_rw_int = MEM_STB; break;
        case OPC6_ST:
            rd_mem_rw_int = MEM_ST; break;
        default:
            rd_mem_rw_int = MEM_NO_RW; break;
    }
}
```

Solution is to try rewriting the code and moving the **`switch`** out of the **`if`**/**`switch`** conditions.

16.8.3 Fully-specified and functionally specified switches

This section provides more information on how to control generated RTL description in order to get functionally or fully-specified multiplexers. Such control allows you to avoid RTL linter warnings and errors or mismatches between RTL and gate-level simulations. This section provides examples of possible scenarios and a resulting Verilog HDL code.

16.8.3.1 Fully-specified switches

For fully-specified **`switch`** statements, the description is direct and simple. The following CodAL examples show such switches:

Example 168: Fully-specified switch without default statement

```
// does not use default statement, all cases/labels are present
switch ((uint2)cnd)
{
    case 0:
        dst = 0;
        break;
    case 1:
        dst = 1;
        break;
    case 2:
        dst = 2;
        break;
    case 3:
        dst = 3;
        break;
}
```

Example 169: Fully-specified switch with default statement

```
// use default statement, stays for single case/label only
switch ((uint2)cnd)
{
    case 0:
        dst = 0;
        break;
    case 1:
        dst = 1;
        break;
    case 2:
        dst = 2;
        break;
    default:
        dst = 3;
        break;
}
```

When default branch stays for single `case` only, the `default` branch is unrolled/replaced with missing `case` and the result is the same as when used explicitly.

Example 170: Fully-specified Verilog case statement

```
always @(*) begin
    case ( cnd )
        2'd0: dst = 8'd0;
        2'd1: dst = 8'd1;
        2'd2: dst = 8'd2;
        2'd3: dst = 8'd3;
        default: dst = 'x;
    endcase
end
```

The Verilog code lists all possible 2-state values and adds `default` branch driving unknown value. There are no pragmas like for full or parallel case.

16.8.3.2 Functionally-specified switches

To create a functionally-specified multiplexer without any default driver, all legal values must be listed and the `default` branch must be also defined to indicate error/warning during CA simulation.

Example 171: Functionally-specified switch

```
switch ((uint2)cnd)
{
    case 0:
        dst = 0;
        break;
    case 1:
        dst = 1;
        break;
    case 2:
        dst = 2;
        break;
    default:
        codasip_fatal( 128, "Reached unexpected branch, cnd = %d.", cnd );
        break;
}
```

The `default` branch should not contain any drivers. The call of *codasip_fatal*, *codasip_error* and *codasip_warning* builtin functions should indicate completeness of the `switch`. The `break` statement is also mandatory. The RTL description should look like this:

Example 172: Functionally-specified Verilog case statement

```
always @(*) begin
    case ( cnd )
        2'd0: dst = 8'd0;
        2'd1: dst = 8'd1;
        2'd2: dst = 8'd2;
        default: dst = 'x;
    endcase
end
```

Note: There should be no warning about uninitialized access of local variables driven in the `default` branch of a `switch`.

16.8.3.3 Incomplete switches with default

In the next example, only some values are specified and for others the `default` branch is used. Such description is direct and simple:

Example 173: Incomplete switch with default branch

```
switch ((uint2)cnd)
{
    case 0:
        dst = 0;
        break;
    case 1:
        dst = 1;
        break;
    default:
        dst = 3;
        break;
}
```

The `default` branch stays for multiple branches, no unrolling is performed and the code will be exactly as it is specified in CodAL.

Example 174:Incomplete Verilog case statement with default branch

```
always @(*) begin
    case ( cnd )
        2'd0: dst = 8'd0;
        2'd1: dst = 8'd1;
        default: dst = 8'd3;
    endcase
end
```

Note: Such RTL code violates rules of almost all HDL linters.

16.8.3.4 Incomplete switches without default

When the `default` branch is not specified or is empty, default (zero) driver is added automatically.

Example 175: Incomplete switch with empty default branch

```
switch ((uint2)cnd)
{
    case 0:
        dst = 0;
        break;
    case 1:
        dst = 1;
        break;
    case 2:
        dst = 2;
        break;
    default:
        break;
}
```

Example 176: Incomplete switch without default branch

```
switch ((uint2)cnd)
{
    case 0:
        dst = 0;
        break;
    case 1:
        dst = 1;
        break;
    case 2:
        dst = 2;
        break;
}
```

The result should produce code looking like priority encoder.

Example 177: Verilog conditional driver created from incomplete switch

```
assign dst = (cnd == 2'd0) ? 8'd0 :
    (cnd == 2'd1) ? 8'd1 :
    (cnd == 2'd2) ? 2'd2 : 8'd0;
```

16.8.3.5 Fully-specified switches with different set of targets

The last example shows a situation when the `switch` statement does not drive the same set of resources and contains code description for full `switch`:

Example 178: Functionally-specified switch driving different set of targets

```
switch ((uint2)cnd)
{
    case 1:
        dst1 = 1;
        break;
    case 2:
        dst2 = 2;
        break;
    default:
        codasip_fatal( 128, "Reached unexpected branch, cnd = %d.", cnd );
        break;
}
```

The tool produces a note telling that the specification will be ignored because all branches must drive the same set of resources when `switch` is marked as fully-specified. The same message will be generated when nested `if`/`else`/`switch` statements will be used in any branch. Conditional (ternary) operator does not break the pattern.

Example 179: Resulting Verilog drivers for unsupported functionally-specified switch

```
assign dst1 = (cnd == 2'd1) ? 8'd1 : 8'd0;
assign dst2 = (cnd == 2'd2) ? 2'd2 : 8'd0;
```

16.8.4 Constant operands in binary operations

There is no need for casting of a constant in comparison operators. A quality of the resulting code will not be increased. Cast to `uint5` in the example below is not necessary since a comparator will be on five bits even with a 32-bit wide constant.

Example 180:Constant operand

```
uint5 i;
// initialize registers to zero
for (i = 0; i < /*(uint5)*/ 16; i++)
    rf_gpr[i] = 0;
```

16.8.5 Local vs. Global signals

Avoid RAW (Read After Write) data dependencies on global signals inside of a single `events`/`elements`/function. Instead, declare a local variable and assign its value with intended logic. Only afterwards, write a value of the local variable into the global resource. This approach produces cleaner interface of the unit and it also improves a process of functional verification.

16.9 RTL options

16.9.1 Reset options

The RTL generator supports various configuration for default reset. The reset may be synchronous or asynchronous, active low or active high. This affects all registers, register files and finite-state machines (FSM) in the design which run in the default clocking domain (CLK) except for explicit override in CodAL model. Note that this configuration does not affect JTAG clock domain (TCK). The JTAG reset configuration is defined by the JTAG IEEE standard - asynchronous, active low.

16.9.2 Target technology

This option specifies target technology to be used, either ASIC or FPGA. There are predefined FPGA targets. Most of the technology specifics are related to the on-chip debugger and clock domain synchronizations. The on-chip debugger for FPGA can be configured to contain BSCAN device primitive for communication over JTAG, or the on-chip debugger may be configured to contain clock gating control logic that inserts special FPGA-specific clock-enable cell.

The ASIP core it self (described directly in CodAL) is technology-independent.

16.9.3 Synchronization register length

This option allows to specify number of flip-flops used for one-bit signal synchronization between asynchronous clock domains(meta-stability treatment). This is only used for one-bit control signals between default clock domain (CLK) and JTAG clock domain (TCK) when on-chip debugger is enabled. General NDFF synchronizer is used for such signals, for n-bit signals the MUX synchronizer is used.

This number specifies the number of flip-flops used in the target clock domain - there is always one-bit flip-flop in source clock domain. Minimum value is 2.

16.9.4 Flip-flop delays

For non-zero values the tool inserts delays to model clock-to-output behaviour on sequential logic. This affects all registers, register files and FSM definitions. The delay is specified in picoseconds.

Example 181: Verilog register template with delay

```
always @(posedge CLK or negedge RST) begin
    if (RST == 1'b0) begin
        Q <= #CODASIP_DFF_DELAY 0;
    end else begin
        Q <= #CODASIP_DFF_DELAY D;
    end
end
```

16.9.5 DFT Scan mode

The option improves DFT (design for test) support by inserting extra input port that is routed to every internal reset generation module. This allows to control reset directly from the top module input port and can be used to pass DFT-specific HDL linter checks.

Note that this option is necessary only when on-chip debugger is enabled. Otherwise there are no internal reset generation modules and the reset is controlled directly from the top module input port.

16.9.6 Condition nesting level

This option limit nesting of Verilog conditional operator (?:) or VHDL when-else. The value defines maximum allowed number of nesting levels for the operator. Default zero value means no limit.

It can be used to pass related HDL linter warnings (STARC).

16.10 RTL optimizations

There are multiple optimization techniques available in the RTL generator. Most of them are suitable for FPGA as well as ASIC synthesis. Their purpose is to create smaller and faster logic. However there are some optimizations which may have negative impact on power consumption. Therefore, these optimizations are optional.

16.10.1 Always activated units

This optimization removes activation ports in functional units that are always activated. The optimization is available only when the *reset* event is empty and the unit is activated unconditionally.

When the *reset* event does not contain any logic, it can be optimized away, the *main* event can be changed to be always active, and its activation port and its logic can be removed. Recursively, every unconditionally activated unit that is also activated by the always-activated unit can be optimized.

Use of this optimization reduces the data path in functional units, improves speed and area of synthesized designs, and may help the synthesis tools with re-timing.

16.10.2 Operand isolation for data parameters

The RTL generator inserts operand isolation logic for drivers of data parameters of instruction decoder and C function. This logic clears the inputs of a unit with zeros when the unit is not activated or when there are multiple drivers of the inputs - the unit is activated from multiple places (e.g. shared adder). This logic may be optimized away if the unit is activated from a single place. The data inputs may be driven unconditionally. This may improve the speed and the area of synthesized design at the cost of power consumption.

16.10.3 Operand isolation for C function activation

The RTL generator inserts operand isolation logic for drivers of the activation port of all units. For simple C function driving only its returning value, this logic may be optimized away. The unit must be called unconditionally from a single place. The driver for the activation signal will be replaced with a constant driver causing the unit to be always active. This may improve the speed and the area of synthesized design at the cost of power consumption.

The difference between this and *Always activated units* optimization is that this time the caller does not need to be always activated and *reset* event does not need to be empty.

16.10.4 Reduce usage of activation condition using auxiliary signal

This optimization reduces complex conditions for drivers with multiple branches including activation condition. It creates an auxiliary signal driving reduced condition if the activation condition is true.

This can be useful to infer switches/multiplexers when driving global signals.

The following Verilog examples show the non-optimized and optimized version.

Example 182: Verilog example before optimization

```
assign dst = ((ACT == 1'd1) && cnd1) ? 8'd1 :
    ((ACT == 1'd1)) && cnd2) ? 8'd2 :
    ((ACT == 1'd1)) && cnd3) ? 8'd3 :
    ((ACT == 1'd1)) && cnd4) ? 8'd4 : 8'd0;
```

Example 183: Verilog example after optimization

```
assign dst_ACT_wire = cnd1 ? 8'd1 :
    cnd2 ? 8'd2 :
    cnd3 ? 8'd3 :
    cnd4 ? 8'd4 : 8'd0;
assign dst = (ACT == 1'd1) ? dst_ACT_wire :  8'd0;
```

16.11 HDL naming conventions

The RTL generator provides basic support for naming conventions used by hardware engineers in different companies. There are multiple options how to influence generated HDL identifiers as well as generated HDL source files. This section provides basic descriptions of available options and categories.

16.11.1 Top ports style

This style influences the ports of the top module (VHDL entity) of the project. The top level port (created from a CodAL project) will respect the definition in *CodAL Language Reference Manual*. By default these ports will be left unchanged, but they can be converted to all upper or lower case English letters.

16.11.2 Top module style

This style is applied to the top module (VHDL entity) name of the CodAL Project only. It also affects the generated HDL file name, because the generator creates files with the exact same name as the module name. The generator respects the project name, but the letters can be converted to all upper or lower case.

16.11.3 Inner modules, ports, instances and signals

Separate style may be defined for inner modules, inner ports, instances of a HDL module and signals. The style defined for inner modules also affects the generated HDL file names, because the generator creates files with the exact same name as the module name. The identifiers are based on the identifiers used in CodAL Project, but the letters can be converted to all upper or lower case. Also specified prefix and suffix can be added to every identifier.

16.11.4 Clock, reset and scan mode signals

One of the basic options allows to specify name of the default clocking and reset signal as well as scan/test mode signal. These identifiers will be used across the whole design (top or inner ports). However, these identifiers respect the style for top ports.

16.12 Known HDL linter issues

There are known issues reported by some linting tools in generated HDL. They depend on the set of rules the linting tool uses. Most of these violations are related to Cross domain crossing between global clock and JTAG clock. If the design does not use on-chip debugger, these violations will not occur.

16.12.1 STARC: Avoid inverting logic on the same clock line

This rule recommends to use separate clock signal for positive and negative clock edge and a clock generation logic (inverter) moved to separate instance located on the top level.

This issue is related to the on-chip debugger and the register on the output TDO port. The JTAG standard recommends to drive this port on the negative edge of the clock. The register and its driver is located in the TAP (Test Access Port). Because the negative edge is used in this single place only, the extra clock generating module and routing have been considered as inappropriate and therefore there is a direct inverter used for this register.

16.12.2 Avoid internally generated reset

This rule recommends to use internally generated reset only if necessary. The on-chip debugger requires the ability to reset the ASIP core in order to start debugging properly (from a well known state). The reset generation logic is encapsulated inside a separate module and instantiated within the ASIP and only the core of the ASIP is connected to this internal reset. However this may not be recognized properly by some linting tools.

17 UVM VERIFICATION WITH ASSERTIONS AND FUNCTIONAL COVERAGE

Details about verification, generating and running UVM verification environments and their description can be found in the chapter "Verification Guide" in the *Codasip Studio User Guide*).

Coverage points and assertions form a very important part in every generated verification environment, while significant part of them is connected to the Codasip Local Bus (CLB). High level requirements and their implementation specifications are described in the following sections.

17.1 CLB Coverage and Assertions Requirements

General requirements for CLB coverage and assertions are specified here. Furthermore, implementation specification for coverage and for assertions based on these requirements is provided in "Coverage Points and Cross Coverage Implementation Specification" on page 362 and "Assertions Implementation Specification" on page 367.

17.1.1 CLB masters

Cover-points and assertions should not be generated for CLB master components, because then they are duplicated in slave components (the same CLB signals are used on both sides with the same restrictions).

17.1.2 CLB slaves

- GENERAL SLAVES (rules that apply to all slaves)
  - All valid values and combinations of `BI`, `BC` signals should be covered and invalid combinations should be asserted (restrictions can be found directly in the CodAL model).
  - Address ranges should be computed according to the bit width. If a restricted address range is defined (e.g. memory), reasonable ranges should be covered with respect to this definition. Occurrences of addresses out of range should be asserted.
  - `STATUS` can be set to `READY` and `BUSY` (should be covered). `BUSY` is always set during `RESET` sequence and when the latency of memory is higher than 1. `ERROR` and reserved values are considered illegal and should be asserted.

- MEMORY read only (ASIP fetch, separated load)
  - `REQUEST` can be set only to `READ`, `NONE` (should be covered), illegal values are: `WRITE`, `INVALIDATE/ALL`, `FLUSH/ALL` and reserved values (should be asserted).
  - `RESPONSE` can be set to `IDLE`, `ACK`, `WAIT` (should be covered), illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (should be asserted). Response to an unaligned address is `UNALIGNED` – considered illegal. Response to an oor address is `OOR` – considered illegal. Responses to the unsupported requests `INVALIDATE/ALL`, `FLUSH/ALL`, `WRITE`, unsupported combinations of `BI`, `BC` and reserved requests are `IDLE`. If memory with latency = 1 is connected, `WAIT` is illegal.
- MEMORY write only (separated store)
  - `REQUEST` can be set only to `WRITE`, `NONE` (should be covered) , illegal values are `READ`, `INVALIDATE/ALL`, `FLUSH/ALL` and reserved values (should be asserted).
  - `RESPONSE` can be set to `IDLE`, `ACK`, `WAIT` (should be covered), illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (should be asserted). Response to an unaligned address is `UNALIGNED` – considered illegal. Response to the oor address is `OOR` – considered illegal. Responses to an unsupported requests `INVALIDATE/ALL`, `FLUSH/ALL`, `READ`, unsupported combinations of `BI`, `BC` and reserved requests are `IDLE`. If memory with latency = 1 is connected, `WAIT` is illegal.
- MEMORY read/write
  - `REQUEST` can be set only to `READ`, `WRITE`, `NONE` (should be covered), illegal values are, `INVALIDATE/ALL`, `FLUSH/ALL` and reserved values (should be asserted).
  - `RESPONSE` can be set to `IDLE`, `ACK`, `WAIT` (should be covered), illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (should be asserted). Response to an unaligned address is `UNALIGNED` – considered illegal. Response to an oor address is `OOR` – considered illegal. Responses to the unsupported requests `INVALIDATE/ALL`, `FLUSH/ALL`, unsupported combinations of `BI`, `BC` and reserved requests are `IDLE`.

- DCACHE
  - REQUEST can be set to READ, WRITE, NONE, INVALIDATE/ALL, FLUSH/ALL (should be covered), illegal values are just reserved values (should be asserted).
  - RESPONSE can be set to IDLE, ACK, WAIT (should be covered), illegal values are ERROR, UNALIGNED, OOR + reserved (should be asserted). Response to an unaligned address is UNALIGNED – considered illegal. If the memory connected to dcache returns OOR for the requested address, OOR response is propagated from dcache – considered illegal. WAIT occurs when a request to the main memory is issued from cache. ERROR occurs when there is a problem with the connected memory – considered illegal. Unsupported combination of BI and BC is considered illegal, response is ERROR. Responses to the reserved requests are IDLE.
- ICACHE
  - REQUEST can be set to READ, NONE, INVALIDATE/ALL (should be covered), illegal values are WRITE, FLUSH/ALL and reserved values (should be asserted).
  - RESPONSE can be set to IDLE, ACK, WAIT (should be covered), illegal values are ERROR, UNALIGNED, OOR + reserved (should be asserted). Response to an unaligned address is UNALIGNED – considered illegal. If the memory connected to icache returns OOR for the requested address, OOR response is propagated from icache – considered illegal. WAIT occurs when a request to the main memory is issued from cache. ERROR occurs when there is a problem with the connected memory – considered illegal. Unsupported combination of BI and BC is considered illegal, response is ERROR. Responses to the unsupported requests WRITE, FLUSH/ALL and reserved requests are IDLE.
- Bridges to AHB/AXI
  - A address ranges should be computed according to the bit width and covered.
  - BI, BC – for AHB bridges, the only legal value of BI is 0, the only legal values of BC are equal to the power of 2 (should be covered). Other values are considered as illegal and should be asserted. For CLB2AXI, CLB2AXI-Lite bridges, all valid combinations of BI, BC are possible (should be covered).

- `REQUEST` can be set to `READ`, `WRITE`, `NONE`, `INVALIDATE/ALL`, `FLUSH/ALL` (should be covered), illegal values are just reserved values (should be asserted).
- `RESPONSE` can be set to `IDLE`, `ACK`, `WAIT` (should be covered), illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (should be asserted). Response to an unaligned address is `UNALIGNED`/`IDLE` – considered illegal. `OOR` should never occur. `WAIT` occurs when there is a latency at the opposite site of the bridge. `ERROR` occurs when there is a problem at the opposite side of the bridge – considered illegal. Responses to the unsupported requests `INVALIDATE/ALL`, `FLUSH/ALL` and reserved requests and unsupported combination of `BI`, `BC` are `IDLE`.
- `STATUS` can be set to `READY`, `BUSY` (should be covered). `ERROR` is never set – it is considered illegal together with the reserved values (should be asserted).

17.2 Coverage Points and Cross Coverage Implementation Specification

Functional coverage will be generated for registers and instructions of an ASIP. It is also generated for memories, caches and other components that are on the slave side of communication, mainly on the CLB bus – therefore, according to the CLB specification.

For every signal that is important from the coverage point of view, cover-points are generated. Simple cover-points are bound to one particular signal, cross cover-points are bound to at least two signals. In a simple cover-point, we categorize possible values of a signal to bins. They can be specified as:

- `bin` – this value or range of values should be always covered.
- `ignore_bin` – if this value occurs, it is not important from the coverage point of view.
- `illegal_bin` – illegal values that should never occur.

It is important to mention that for every generated cover-point, we provide a generated description of its meaning. These descriptions are in the form of `option.comment` in cover-points implementation.

In the further sub-sections, UVM components called agents are listed. These agents contain several sub-components, among them also coverage monitors which contain simple cover-points and also cross cover-points.

17.2.1 Registers Agent

Register agent is connected to registers or register arrays inside of an ASIP. Registers typically use address (RA = read address, WA = write address), data signals and control signals (WE = write enable, RE = read enable).

- Addresses are monitored with `option.weight = 0`. It means that they do not actively participate in coverage computation. However, they are important in cross coverage where they are coupled with control signals These crosses actively participate in the coverage computation.
- For enable signals like RE, WE, we monitor enabling and also disabling:

Example 184:Enable signals coverage.

```
cvp_regs_RE0 : coverpoint m_transaction_h.RE0 {
       bins enabled  = {1};
       bins disabled = {0};
}
```

- For cross cover-points, addresses are checked only when the control signal is enabled (enable signal = 1):

Example 185: Cross for enable signal.

```
cvp_regs_RA0_valid: cross cvp_regs_RA0, cvp_regs_RE0 {
       ignore_bins ig = binsof(cvp_regs_RE0.disabled);
}
```

17.2.2 Decoder Agent

Decoder agent is connected to an instruction decoder in an ASIP. Therefore, it contains information about which instruction is currently decoded.

- All instructions from the instruction set of an ASIP should be covered.
- Sequences of instructions. This option is not generated by default, because for complex instruction sets there is a huge number of sequences to be covered and not all of them are reachable. Nevertheless, sequences can be easily added - see the following example (checks transition of every instruction to the `jump_addr` instruction):

Example 186:Sequences of instructions.

```
inst_after_inst: coverpoint m_transaction_h.m_instruction {
bins trans1 = ([0:$] => codix_risc_ca_core_main_instr_hw_instr_hw_t_decoder:: jump___
addr__);
}
```

- `UNKNOWN` instructions are illegal, therefore, they belong to the illegal bin:

Example 187: Cross for enable signal.

```
cvp_instructions : coverpoint m_transaction_h.m_instruction{
illegal_bins UNKNOWN = {codix_risc_ca_core_main_instr_hw_instr_hw_t_decoder::UNKNOWN};
}
```

17.2.3 ASIP Agent

The ASIP agent is connected to an ASIP core. From the coverage point of view, only input and output ports of an ASIP are important.

If an ASIP has primary input or output ports, the cover-points for them are generated according to their bit-widths. So we recommend to adjust these cover-points based on their semantics.

If a direct CLB connection exists between an ASIP and a CLB slave, no CLB functional coverage is generated on the ASIP side! It is always generated on the slave side of communication.

17.2.4 Top-level Agent

The top-level agent is connected to an ASIP's top-level architecture, which contains at least the ASIP and memories, and often also caches an other peripherals. From the coverage point of view, only input and output ports of the top-level are interesting. Again, cover-points for them are generated based on their bit-widths automatically, so we recommend to adjust them based on their semantics.

If these ports are the same as in ASIP agent, cover-points in the ASIP agent can be removed.

17.2.5 CLB Agents

Follwing rulles apply to all CBL slaves.

CLB agents are connected to CLB slave components. Cover-points are bound to CLB signals.

- **`STATUS`** - can be set to `READY` and `BUSY` (valid bins). `ERROR` and reserved values are considered illegal (illegal bins).
- **`A`** - address ranges should be computed according to the bit width. If a restricted address range is defined (e.g. memory), reasonable ranges should be set with respect to this definition (valid bins). Addresses are monitored with option.weight = 0, they are important only in cross coverage.

- **`BI,BC`** - all valid combinations (valid bins). `BI` and `BC` values are monitored with option.weight = 0, they are important only in cross coverage.
- **`REQUEST`** - valid values depend on the CLB interface - whether it is read-only, write-only, or read-write. Also, some requests are valid only with a cache connected like INVALIDATE or FLUSH. Values are monitored with option.weight = 0, they are important only in cross coverage.

17.2.5.1 CLB Memory Agent

- READ ONLY INTERFACE (FETCH)
  - Coverpoints

    `REQUEST`: can be set only to `READ`, `NONE` (valid bins)., illegal values are `INVALIDATE/ALL`, `WRITE`, `FLUSH/ALL` and reserved (illegal bins).

    `RESPONSE`: can be set to `IDLE`, `ACK`, `WAIT` (valid bins). If memory with latency = 1 is used, `WAIT` is illegal (illegal bins). Always illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (illegal bins).
  - Cross coverage
    - `READ REQUEST` during `STATUS READY`with different addresses.
    - Optional: Sequences of requests: `RAR`= `READ REQUEST`after `READ REQUEST`.
- WRITE ONLY INTERFACE
  - Cover-points

    `REQUEST`: can be set only to `WRITE`, `NONE` (valid bins), illegal values are `INVALIDATE/ALL`, `READ`, `FLUSH/ALL` and reserved (illegal bins).

    `RESPONSE`: can be set to `IDLE`, `ACK`, `WAIT` (valid bins). If memory with latency = 1 is used, `WAIT` is illegal (illegal bins). Always illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (illegal bins).
  - Cross coverage
    - `WRITE REQUEST` during `STATUS READY` with different addresses.
    - Optional: Sequences of requests: WAW = `WRITE REQUEST` after `WRITE REQUEST`.

- READ/WRITE INTERFACE (LOAD/STORE)
  - Coverpoints

    `REQUEST`: can be set only to `READ`, `WRITE`, `NONE` (valid bins), illegal values are `INVALIDATE/ALL`, `FLUSH/ALL`, reserved values (illegal bins).

    `RESPONSE`: can be set to `IDLE`, `ACK`, `WAIT` (valid bins). Always illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (illegal bins).
  - Cross coverage
    - Different requests during `STATUS READY` with different addresses.
    - Different `BI`/`BC` combinations during `STATUS READY` with different addresses.
    - Optional: Sequences of requests: RAW, RAR, WAW, WAR = `READ REQUEST` after `WRITE REQUEST`, `READ REQUEST` after `READ REQUEST`, `WRITE REQUEST` after `WRITE REQUEST`, `WRITE REQUEST` after `READ REQUEST`.

17.2.5.2 CLB Cache Agent

- ICACHE (slave)
  - Coverpoints

    `REQUEST`: can be set only to `READ`, `NONE`, `INVALIDATE/ALL` (valid bins), illegal values are `WRITE`, `FLUSH/ALL` and reserved (illegal bins).

    `RESPONSE`: can be set to `IDLE`, `ACK`, `WAIT` (valid bins). Always illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (illegal bins).
  - Cross coverage
    - Fetch instructions (`READ`) + `INVALIDATE/ALL` REQUEST) during `STATUS READY`with different addresses.
    - Optional: Sequences of requests: RAR, IAR, IAAR, RAI, RAIA.

- DCACHE (slave)
  - Coverpoints

    `REQUEST`: can be set only to `READ`, `NONE`, `INVALIDATE/ALL`, `WRITE`, `FLUSH/ALL` (valid bins). Illegal values are reserved values (illegal bins).

    `RESPONSE`: can be set to `IDLE`, `ACK`, `WAIT` (valid bins). Always illegal values are `ERROR`, `UNALIGNED`, `OOR` + reserved (illegal bins).
  - Cross coverage
    - Different requests during `STATUS READY` with different addresses.
    - Different `BI`/`BC` combinations during `STATUS READY`with different addresses.
    - Optional: Sequences of requests: RAW, RAR, WAW, WAR, IAR, IAAR, FAR, FAAR, IAW, IAAW, FAW, FAAW, WAI, RAI, WAIA, RAIA, WAF, RAF, WAFA, RAFA.
- CACHE (master) - no coverage collected.

17.3 Assertions Implementation Specification

Similarily as functional coverage, assertions will be also generated for ASIP, its registers, memories, and other components that are on the slave side of communication, mainly on the CLB bus – therefore, according to the CLB specification. All unsupported and illegal scenarios should be asserted.

In the further text, supported `REQUEST` is used to represent the supported commands for specific components, unsupported `REQUEST` represents unsupported commands. Reserved values for `REQUEST` are always considered illegal. `NONE REQUEST` is supported, but no action is done on the CLB bus.

Messages in the text are just approximate. Implemented assertion messages are more accurate and navigate the user to the origin of a problem.

17.3.1 ASIP [ASSERTION_ERROR_ASIP]

- If an ASIP has primary input or output ports, just such assertions which cover undefined values are generated for them. E.g., X value cannot occur on the `PORT_ERROR` port.

  Msg: *“X value occurred on `PORT_ERROR`. This can cause an unexpected* *behavior."*

- In the controllers of an ASIP, there are many constructs like following (signals are multi-bit):

  Example 188: multi-bit signals.

  ```
  assign signal = signal_A  | signal_B | signal_C;
  ```

  The condition here is that just one signal on the right side of the expression can be set to 1. Therefore, an assertion is generated to check this condition:

  Example 189: Generated assertion.

  ```
  logic [2:0] temp;
  assign temp[0] = |(signal_A); // or of internal bits!
  assign temp[1] = |(signal_B);
  assign temp[2] = |(signal_C);

  property one_hot_temp;
        @(posedge CLK)
        $onehot0(temp); // onehot in temp signal -> just one driver
  endproperty

  assert property (one_hot_temp)
  else `uvm_error(ASSERTION_ERROR_SIGNAL_MERGE, $sformatf("More then one driver active for
  temp"));
  ```

  Msg: *"More than one driver is active for `temp`. It this case, only one driver is permitted."*

- Instruction `UNKNOWN` is illegal.

  Msg: *"`UNKNOWN` instruction was encountered. `UNKNOWN` instructions are considered illegal."*

- Instruction cannot contain X value.

  Msg: *"X value occurred in instruction. This can cause an unexpected behavior."*

17.3.2 Registers [ASSERTION_ERROR_REGS]

- X values cannot be written to or read from registers.

  Msg: *"X value is read from the register. This may cause an unexpected behavior. "*; *"X value is written to the register. This may cause an unexpected behavior."*

- During enabled read or write from/to registers, address cannot contain X value.

Msg: *"Address (RA/WA) contains X value during the enabled `READ`/`WRITE` from/to registers (`RE/WE` = 1). This may cause an unexpected behavior."*

- Control enable signals (`RE/WE`) cannot contain X value.

  Msg: *"Control enable signal (RE/WE) contains X value. This may cause an unexpected behavior."*

- Every register should be initiated before read.

  Msg: *"DATA are read from an addressed register (RE is activated) %x, but this register was not initialized before (WE was not activated)."*

17.3.3 Memory [ASSERTION_MEMORY]

- Control signals `REQUEST`, `STATUS`, `QRESPONSE`/`DRESPONSE`, `BI`, `BC` cannot contain X value.

  Msg: *"Control signal `REQUEST`/`STATUS`/`RESPONSE`/`BI`/`BC` contains X value. This may cause an unexpected behavior."*

- `RESERVED` values cannot occur on any of the input and output control signals. Therefore, an assertion is firing if a reserved value is set on any of the following control signals: `REQUEST`, `STATUS`, `QRESPONSE` or `DRESPONSE`.

  Msg: *"A reserved value occurred on `REQUEST`/ `STATUS`/ `QRESPONSE`/ `DRESPONSE`. Reserved values are considered illegal."*

- `ERROR` cannot occur on the `STATUS` control signal.

  Msg: *"Error occurred on `STATUS` control signal. This is considered illegal."*

- `QRESPONSE` and `DRESPONSE` control signals cannot contain values `UNALIGNED`, `ERROR` and `OOR`. This is considered illegal.

  Msg: *"Unaligned occurred on `QRESPONSE`. It means that an unaligned address was set during a valid `READREQUEST`. This is considered illegal."*, *"Unaligned occurred on `DRESPONSE`. It means that an unaligned address was set during a valid `WRITEREQUEST`. This is considered illegal."*

  Msg: *"Error occurred on `QRESPONSE`. This is considered illegal."*, *"Error occurred on `DRESPONSE`. This is considered illegal."*

  Msg: *"`OOR` occurred on `QRESPONSE`. It means that an address bigger than the address space of memory was set during a valid `READREQUEST`. This is considered illegal."*, *"`OOR` occurred on `DRESPONSE`. It means that an address* *bigger than the address space of memory was set during a valid `WRITEREQUEST`. This is considered illegal."*

- When `NONE` occurs on `REQUEST` and `STATUS` is set to `READY`, in the next clock cycle, `RESPONSE` should be `IDLE`.

  Msg: *"`IDLE` is not set on `RESPONSE` one clock cycle after `NONEREQUEST` occurred and `STATUS` was set to `READY`. This is considered illegal."*

17.3.4 Additional Assertions for Specific Memory Interfaces

- Only supported `REQUEST`s can be set on the control signal `REQUEST`.

  Msg: *"Unsupported request occurred on `REQUEST`. The list of unsupported requests for the memory follows: %m."*

- When supported `READREQUEST` occurs, X value cannot be read from the memory. This means that when `ACK` is set on `QRESPONSE`, the value of Q cannot be X.

  Msg: *"X value is read from the memory (Q signal) after a valid `READREQUEST`. This may cause an unexpected behavior."*

- When supported `WRITEREQUEST` occurs, X value cannot be written to the memory. This assertion is raised if `WRITE` occurs on `REQUEST`, `STATUS` is set to `READY`, and the value of D is equal to X.

  Msg: *"X value is written to the memory (D signal) after a valid `WRITEREQUEST`. This may cause an unexpected behavior."*

- Address for supported `REQUEST`s cannot contain X value when `STATUS` is set to `READY`.

  Msg: *"Address contains X value during a valid `REQUEST`. This may cause an unexpected behavior."*

- When supported `REQUEST` occurs and `STATUS` is set to `READY`, the sum of `BI` + `BC` cannot exceed the number of bytes in the word (`WORD_BITS / LAU_BITS`).

  Msg: *"Invalid combination of Index (`BI`) and Counter (`BC`) occurred during a valid `REQUEST`. The summa of Index and Counter cannot exceed the word size."*

- When supported `READ`/`WRITEREQUEST` occurs, `STATUS` is set to `READY` and the sum of `BI` + `BC` exceeds the number of bytes in the word, `ERROR` should be set on `QRESPONSE/DRESPONSE` with the read-operation/write-operation latency.

Msg: *"Invalid combination of Index (`BI`) and Counter (`BC`) during a valid `WRITE REQUEST` is not followed by `ERROR` on `DRESPONSE`. This is considered illegal."*, *"Invalid combination of Index (`BI`) and Counter (`BC`) during a valid `READREQUEST` is not followed by `ERROR` on `QRESPONSE`. This is considered illegal."*

- When supported `READ/WRITEREQUEST` occurs, `STATUS` is set to `READY` and the sum of `BI` + `BC` does not exceed the number of bytes in the word, `ACK` should be set on `QRESPONSE/DRESPONSE` with the read-operation/write-operation latency.

  Msg: *"A valid `WRITEREQUEST` is not followed by `ACK` on `DRESPONSE`. This is considered illegal."*, *"A valid `READ REQUEST` is not followed by `ACK` on `QRESPONSE`. This is considered illegal."*

- When supported `WRITEREQUEST` occurs, the value of D should stay stable while the write operation is processed – i.e. while the `STATUS` remains `BUSY`.

  Msg: *"Data (D signal) are not stable during processing of a valid `WRITEREQUEST`. Processing of `WRITEREQUEST` is signalised by `BUSY` on `STATUS`."*

- When supported `READREQUEST` occurs, `IDLE` should be set on `QRESPONSE` with the proper read-operation latency.

  Msg: *"Unsupported `READ` request is not followed by `IDLE` on `QRESPONSE`. This is considered illegal."*

- When unsupported `WRITEREQUEST` occurs, `IDLE` should be set on `DRESPONSE` with the proper write-operation latency.

  Msg: *"Unsupported `WRITE` request is not followed by `IDLE` on `DRESPONSE`. This is considered illegal."*

- When unsupported `REQUEST` occurs, `IDLE` should be set on both: `QRESPONSE` and `DRESPONSE` with the proper read-operation/write-operation latency. This means that `IDLE` for the same request can occur in a different clock cycle on `QRESPONSE` and `DRESPONSE`.

  Msg: *"Unsupported `REQUEST` is not followed by `IDLE` on both `QRESPONSE` and `DRESPONSE`. This is considered illegal."*

- No other request should be requested while the previous one is still being processed.

  Msg: *"A new request is asserted while the previous request is still being processed. It is considered illegal."*

Restrictions for specific memory interfaces:

1. READ/WRITE INTERFACE
   - Supported `REQUEST`s are: `READ`, `WRITE`.
2. READ ONLY INTERFACE
   - Supported `REQUEST` is: `READ`.
   - Unsupported `REQUEST`is: `WRITE`.
3. WRITE ONLY INTERFACE
   - Supported `REQUEST` is: `WRITE`.
   - Unsupported `REQUEST` is: `READ`.

17.3.5 General CLB [ASSERTION_CLB_]

These assertions are generally applicable in every CLB component.

- Control signals `REQUEST`, `STATUS`, `RESPONSE`, `BI`, `BC` cannot contain X value.

  Msg: *"Control signal `REQUEST`/`STATUS`/`RESPONSE`/`BI`/`BC` contains X value. This may cause an unexpected behavior."*

- `RESERVED` values cannot occur on any of the input and output signals. Therefore, an assertion is fired if a reserved value is set on any of the following control signals: `STATUS`, `REQUEST`, or `RESPONSE`.

  Msg: *"A reserved value occurred on `REQUEST`/ `STATUS`/ `RESPONSE`. Reserved values are considered illegal."*

- `ERROR` cannot occur on the `STATUS` signal.

  Msg: *"Error occurred on `STATUS`. This is considered illegal."*

- `STATUS` is always set to `BUSY` in the same cycle as `RESET` is active.

  Msg: *"`STATUS` is not set to `BUSY` during `RESET`. This is considered illegal."*

- `RESPONSE` signal cannot contain values `UNALIGNED`, `ERROR` and `OOR`. This is considered illegal.

Msg: *"Unaligned occurred on `RESPONSE`. It means that an unaligned address was set during a valid `REQUEST`. This is considered illegal."*

Msg: *"Error occurred on `RESPONSE`. This is considered illegal."*

Msg: *"`OOR` occurred on `RESPONSE`. It means that an address bigger than the address space of memory was set during a valid `REQUEST`. This is considered illegal."*

- When `RESPONSE` is `WAIT`, `STATUS` is always set to `BUSY`.

  Msg: *"`STATUS` is not set to `BUSY` during `WAIT`. This is considered illegal."*

- No other request should be requested while the previous one is still being processed. Therefore, `REQUEST`, `ADDRESS`, `BI` and `BC` should stay stable while `BUSY` is set on `STATUS`.

  Msg: *"`REQUEST` is not stable during `BUSY` on `STATUS`. This is considered illegal."*, *"`ADDRESS` is not stable during `BUSY` on `STATUS`. This is considered illegal."*, *"Index (`BI`) and Counter (`BC`) are not stable during `BUSY` on `STATUS`. This is considered illegal."*

- When `RESERVED` or `NONEREQUEST` occurs and `STATUS` is set to `BUSY`, in the next clock cycle, `RESPONSE` can be set to any value except the reserved values.

  Msg: *"Reserved value occurred on `RESPONSE` one clock cycle after `RESERVED/NONEREQUEST` occurred and `STATUS` was set to `BUSY`. This is considered illegal."*

17.3.6 Additional Assertions for Specific CBL Component Interfaces

- Only supported `REQUEST`s can be set on `REQUEST`.

  Msg: *"Unsupported request occurred on `REQUEST`. The list of unsupported requests for <memory/cache/bridge> follows: %m."*

- When supported `READREQUEST` occurs, X value cannot be read from <memory/cache/bridge>. This means that when the `ACK` is set on `RESPONSE`, the value of Q cannot be X.

  Msg: *"X value is read from <memory/cache/bridge> (Q signal) after a valid `READREQUEST`. This may cause an unexpected behavior."*

- When supported `WRITE/FLUSH/FLUSH_ALLREQUEST`s occur, X value cannot be written to <memory/cache/bridge>. This assertion is raised if

WRITE/FLUSH/FLUSH_ALL is requested on the REQUEST signal, STATUS is set to READY, and the value of D is equal to X.

Msg: *"X value is written to <memory/cache/bridge> (D signal) after a valid WRITE/FLUSH/FLUSH_ ALL REQUEST. This may cause an unexpected behavior."*

- Address for supported REQUESTs (except *_ALLREQUESTs) cannot contain X value when STATUS is set to READY.

  Msg: *"Address contains X value during a valid REQUEST (except *_ ALLREQUESTs). This may cause an unexpected behavior."*

- When a supported REQUEST occurs, the given ADDRESS cannot exceed the size of the memory space, i.e. ADDRESS < MEMORY_SIZE. Otherwise, OOR should be set on RESPONSE.

  Msg: *"Address bigger than the address space of memory was set during a valid REQUEST. This is considered illegal."*

  Msg: *"Address bigger than the address space of memory was set during a valid REQUEST, but OOR did not occur on RESPONSE. This is considered illegal."*

- When supported REQUEST (except *_ALLREQUESTs) occurs, the given ADDRESS should be aligned. ADDRESS is aligned if ADDRESS modulo BC (number of bytes per word) equals zero. Otherwise, UNALIGNED should be set on RESPONSE.

  Msg: *"Unaligned address was set during a valid REQUEST. This is considered illegal. Address is aligned when A modulo WORD_SIZE equals zero."*

  Msg: *"Unaligned address was set during a valid REQUEST, but UNALIGNED did not occur on RESPONSE. This is considered illegal."*

- When supported REQUEST occurs and STATUS is set to READY, after a various number of clock cycles (depends on the latency) during which RESPONSE is set to WAIT, RESPONSE is set to ACK, ERROR, OOR or UNALIGNED. In this case, RESPONSE cannot be set to IDLE before being set to ACK, ERROR, OOR or UNALIGNED.

  Msg: *"RESPONSE is not set to ACK/ERROR/OOR/UNALIGNED after it was set to WAIT during the processing of a valid REQUEST. WAIT is always set one cycle after a valid REQUEST occurs and STATUS was set to READY, but only in the case the processing of the request has some latency."*

- When supported `REQUEST` occurs and `STATUS` is set to `READY`, the sum of `BI` + `BC` cannot exceed the number of bytes in the word (`WORD_BITS / LAU_BITS`).

  Msg: *"Invalid combination of Index (`BI`) and Counter (`BC`) occurred during a valid `REQUEST`. This is considered illegal. The summa of Index and Counter cannot exceed the word size.*

- When supported `REQUEST` occurs, `STATUS` is set to `READY/ERROR`, and an unsupported combination of `BI` and `BC` occurs, in the next clock cycle, `RESPONSE` should be set to `ERROR`.

  Msg: *"Invalid combination of Index (`BI`) and Counter (`BC`) occurred during a valid `REQUEST` and `STATUS` set to `READY/ERROR`, but it is not followed by `ERROR` on `RESPONSE` in the next clock cycle. This is considered illegal."*

- When supported `REQUEST` occurs and `STATUS` is set to `READY` or `BUSY`, in the next clock cycle, `RESPONSE` can be set to any values except the reserved values.

  Msg: *"`RESPONSE` is set to a reserved value one clock cycle after a valid `REQUEST` occurred and `STATUS` was set to `READY/BUSY`. This is considered illegal."*

- When supported `REQUEST` occurs and `STATUS` is set to `ERROR`, in the next clock cycle `RESPONSE` should be set to `IDLE`.

  Msg: *"`RESPONSE` is not set to `IDLE` one clock cycle after a valid `REQUEST` occurred and `STATUS` was set to `ERROR`. This is considered illegal."*

- When unsupported `REQUEST` occurs and `STATUS` is set to `READY` or `ERROR`, in the next clock cycle `RESPONSE` should be set to `IDLE`.

  Msg: *"`RESPONSE` is not set to `IDLE` one clock cycle after an unsupported `REQUEST` occurred and `STATUS` was set to `READY/ERROR`. This is considered illegal."*

- When unsupported `REQUEST` occurs and `STATUS` is set to `BUSY`, in the next clock cycle, `RESPONSE` can be set to any values except the reserved values.

  Msg: *"`RESPONSE` is set to a reserved value one clock cycle after an unsupported `REQUEST` occurred and `STATUS` was set to `BUSY`. This is considered illegal."*

Restrictions for CLB memory interfaces [ASSERTION_CLB_MEMORY]:

1. READ ONLY interface (FETCH)
    - Supported requests are: `READ`
    - Unsupported requests are: `WRITE`, `FLUSH`, `FLUSH_ALL`, `INVALIDATE` and `INVALIDATE_ALL`
2. WRITE ONLY interface
    - Supported requests are: `WRITE`
    - Unsupported requests are: `READ`, `FLUSH`, `FLUSH_ALL`, `INVALIDATE` and `INVALIDATE_ALL`
3. READ/WRITE INTERFACE (LOAD/STORE)
    - Supported requests are: `READ`, `WRITE`
    - Unsupported requests are: `FLUSH`, `FLUSH_ALL`, `INVALIDATE` and `INVALIDATE_ALL`

Restrictions for CLB caches [ASSERTION_CLB_CACHE]:

1. ICACHE
    - Supported requests are: `READ`, `INVALIDATE`, `INVALIDATE_ALL`.
    - Supported requests are: `WRITE`, `FLUSH`, and `FLUSH_ALL`.
2. DCACHE
    - Supported requests are: `READ`, `WRITE`, `INVALIDATE`, `INVALIDATE_ALL`, `FLUSH`, and `FLUSH_ALL`.

18 TESTBENCHES GENERATION

The hardware generator can produce simple testbench files (Verilog/SystemVerilog/VHDL). The testbench is well prepared (clock and reset are generated; unit under test is also connected), so user should only change stimuli. There is also a simple TCL startup script prepared to launch simulation of the testbench using Questa® Advanced Simulator, Riviera-PRO. The script compiles the design, performs some customizations of the wave-form, and starts the simulation itself.

19 CODASIP SDK REFERENCE

An SDK is a collection of programming tools based on the clang compiler driver that can be used for the development of applications. The main goal of the clang compiler driver is gcc command line compatibility, so the majority of existing projects based on the GNU makefile system can be compiled with minimal or no effort.

IN THIS CHAPTER

19.1 Directory structure

The published SDK has a simple directory structure which can vary slightly according to the user's environment. The following directory tree shows the SDK structure and important parts such as libraries. The variations in the tree are marked as optional and further explanation will be provided.

Process of publishing SDK is done into `<project>/work/<ia|ca>/sdk` directory after each tool is generated.

Example 190: Exported toolchain directory structure

```
sdk
|-- bin
|    |-- PREFIX-assembler
|    |-- PREFIX-clang
|    |-- PREFIX-gcc
|    |-- PREFIX-ld
|    |-- PREFIX-llc
|    |-- PREFIX-isimulator
|    +-- ...
|-- lib
|    |-- clang
|    |    +-- x.y
|    |         +-- include
|    |              |-- limits.h
|    |              |-- stdarg.h
|    |              +-- ...
|    +-- libcomp.a (optional)
|-- newlib (optional)
|    |-- lib
|    |    |-- crt1.o
|    |    |-- libc.a
|    |    |-- libnosys.a
|    |    +-- ...
|    +-- include
+-- ...
```

The PREFIX- before the name of every utility corresponds to the user specific prefix which represents a standard pair. The pair is a simple string in form PROJECT-MODEL-. For example the following prefixes are possible:

Example 191: Utility name prefixes

```
codix_cobalt-ia-
codix_cobalt-ca-
codix_urisc-ia-
```

19.2 Programming tools

The published SDK contains all necessary utilities to compile and debug the developed software projects. The utilities mainly consists of the GNU Binutils programming tools, the clang compiler driver and the generated tools build from a model of the target processor.

The following table summarizes all tools contained in the toolchain:

Table 39: Summarization of all tools contained in the toolchain

Tool	**Description**
addr2line *	Converts addresses into file names and line numbers.
ar *	Creates, modifies, and extracts archives.
assembler	Assembles the assembly source file into ELF object files compatible with the GNU Binutils.
c++filt *	Demangles C++ and Java symbols.
clang	The Clang C, C++, and Objective-C compiler and compiler driver.
disassembler	Disassembles the ELF object file into the assembly source file.
elfedit *	Updates the ELF header of the ELF files.
gcc	It's only a script which redirects all inputs the to clang which is compatible with the gcc interface. It is useful for makefile projects which relies on the gcc and it cannot be overridden with the clang driver.
gprof *	Displays call graph profile data. (This utility depends on program which is specifically compiled so it can generate profile file but actual implementation of the toolchain does not support this format.)

isimulator	Simulates the compiled programs in the form of ELF executable file. On top of that it incorporates GDB and GDB/MI2 command interpreters so it can be easily integrated into IDE tools e.g. Eclipse CDT.
ld *	The GNU linker which has been slightly modified to support generic relocations, Harvard architectures, diverse byte and word sizes and multiple address spaces.
llc	Compiles the LLVM source inputs into the assembly language for a target architecture. It is generated compiler back-end and is mainly used by the clang compiler driver.
nm *	Lists symbols from the object files.
objcopy *	Copies and translates the object files.
objdump *	Displays information from the object files. (For Harvard architectures prints same additional information.)
opt	Optimizes the LLVM source files.
ranlib *	Generates index to the archive.
readelf *	Displays information about the ELF files.
size *	Lists section sizes and total size.
strings *	Prints the strings of printable characters in the files.
strip *	Discards symbols from the object files.

* Tools from the GNU Binutils.

19.3 Clang

The clang is a C language front-end for LLVM that also replaces the gcc compiler driver with which it is mostly compatible. Development is completely open-source and involves several major software companies e.g. Google or Apple.

Current state of the development and user's manuals can be obtained from clang http://clang.llvm.org/ website. For description of parameters and usage please refer to the manual pages of the clang or the gcc. This document describes only important or extended parameters.

19.3.1 Mandatory parameters

To support the generated compiler back-end, additional mandatory parameters were added in order to ensure proper functionality of the clang. It would be very inconvenient to add these parameters to every command and so the clang utility is actually a generated script which, alongside with the user input, sets these mandatory parameters and passes them to the real program, which has been renamed to clang.rel.

Table 40: Clang – Mandatory parameters

Parameter	Description
-target	Specifies the target architecture for which is clang configured. The exported toolchain is configured only for the specific codasip target, so the parameter must be in the form of codasip-ARCH. The variable ARCH is set during configuration of the exported toolchain and is used by the clang for implicit macro definition.
-ccc-custom-datalayout	Sets the datalayout according to the generated architecture. It denotes the width of the basic data types.
-ccc-allowed-clobbers	Lists the architectural registers which can be clobbered.

Please don't modify these generated parameters without knowing exactly what these arguments mean.

19.3.2 Useful parameters

The following table shows the recommended clang parameters which can be helpful during the debugging session of the compilation process.

Table 41: Clang – Useful parameters

Parameter	Description
-v	Shows commands to be run with verbose output. Useful for analyzing parameters passed to the external utilities.
-save-temps	Saves the intermediate compilation results. Useful for figuring out which phase of compilation causes an issue.
-ccc-print-bindings	Shows the bindings of the tools to an actions. Useful for the debugging of the clang compiler driver.
-ccc-print-options	Dumps the parsed command line arguments. Can be used to check whether the parameters were properly passed to the clang.

19.3.3 Implicit macro definitions

The clang preprocessor generates several predefined macros. A list of these macros can be obtained with command:

Example 192: Command to obtain clang preprocessor generated macros

```
clang -dM -E - </dev/null
```

The Codasip Studio extended this list with another two implicit macros:

Table 42:Clang extra macros from Codasip Studio

Macro	Description
__codasip__	Can be used for detection of the Codasip Studio or Codasip Codespace.
__ARCH__	User specific implicit macro useful for detection of the target architecture. It's value can be configured with clang -target parameter.

19.4 Standard libraries

The published SDK, except standard programming utilities, is composed of the standard libraries and associated header files. This creates package ready to compile any project written in standard C or C++ language (with the C++ limitations mentioned earlier). The published SDK can be configured not to contain any standard library but it is not very useful for production toolchains.

To simplify invocation of the clang compiler driver, standard paths and libraries are automatically set and passed to the preprocessor and linker. Only minimal set of arguments must be provided on the command line. But in some cases this feature is not required and this default behaviour can be suppressed with standard parameters.

Table 43: Standard libraries overview

Library	Description
crt0.o; crt1.0	Startup routines generally written in assembly language. It usually calls main function written in C.
crti.o	Supplies function prologues for the .init and .fini sections. It is linked in before crtbegin.o.
crtn.o	Supplies function epilogues for the .init and .fini sections. It is linked in after all other files.
crtbegin.o	Provides the body of the constructor functions _init and _finit.

crtend.o	Provides the body of the destructor functions _init and _finit.
libcomp.a	The compiler-rt runtime library that provides an implementation of the low-level target-specific hooks required by the code generation. This is the most important library without which majority of the programs can not be compiled.
libc.a	Implementation of the C standard library. The Codasip Studio provides newlib implementation which is convenient for embedded systems.
libm.a	Provides the standard math library functions. As standard libc library this library is part of the newlib implementation.
libnosys.a	Part of the C standard defines a system dependent function (e.g. fread, fwrite). Newlib provides this library to implement the functions only as a stub so program can be linked correctly without need to implement these functions.

The startup files crti.o, crtn.o, crtbegin.o and crtend.o together create system for invocation of constructors and destructors. These libraries are not currently generated automatically and must be provided by the user. However the most solutions these libraries are not required.

The library libnosys.a is a part of the newlib library Libgloss. This library refers to the things like startup code, and usually I/O support for C library. There is usually a target-specific implementation of system dependent function. Documentation about porting newlib describes how can be added a new user specific library.

19.4.1 Default configuration

The standard configuration of the clang compiler driver assigns default include paths to preprocessor and passes standard libraries to the linker. Default preprocessor paths are as follows:

Example 193: Default preprocessor paths

```
-I$ROOT/lib/clang/x.y/include
-I$ROOT/$STDLIB/include
```

The paths are passed in this order. The symbol `x.y` denotes version of clang and `$ROOT` is a placeholder for the path to the installed toolchain. The `$STDLIB` stands for standard library directory which can be configured with parameter `–stdlib=$STDLIB`. If this parameter is not passed it is treated as if it was newlib which is provided by the Codasip Studio.

Default search paths for linker are set in this order:

Example 194: Default search paths for linker

```
-L$ROOT/$STDLIB/lib
-L$ROOT/lib
```

The startup files `crt0.s`, `crt1.o`, `crti.o`, `crtn.o`, `crtbegin.o` and `crtend.o` can be placed to the directory `newlib/lib` or `lib`. The former location is used first and then latter if the file is missing. These files are not mandatory so it is admissible not to place them in any location.

Alongside with include paths and startup libraries, there is the C standard library and the compiler-rt library linked. However math library is not linked with C library by default. It must be provided on the command line as -lm.

The user libraries placed in the `$ROOT/$STDLIB/lib` (e.g. `libsim.a`) are treated as standard libraries and so placed to the same linker group as libc to bypass possible circular references. However, these libraries must be referred to as -`l$SYSTEM_LIBS`.

For illustration successive sequence is provided for understanding the order in which the libraries are passed to the linker:

Example 195: Sequence of passing the libraries to the linker

```
crt0.o crti.o crtbegin.o
-lm --start-group -lcomp -lc -l$SYSTEM_LIBS --end-group
crtend.o crtn.o
```

For example the following command

Example 196: Command passing libraries

```
./PREFIX-clang main.c -lnosys
```

is passed to the linker as

Example 197: Previous command passed to the linker

```
./PREFIX-ld -o a.out $ROOT/$STDLIB/lib/crt0.o -L$ROOT/lib -L$ROOT/$STDLIB/lib /tmp/main-AV53n8.o --
start-group -lcomp -lc -lnosys --end-group
```

19.4.2 Compiler driver parameters

Default behaviour of the compiler driver can be suppressed with several parameters. Consecutive table summarizes all of them.

Table 44: Compiler driver parameters summarization

Parameter	**Description**

-nodefaultlibs	Do not use the standard system libraries for linking. The standard system startup files (e.g. crt0.o, crti.o) are still linked.
-nostartfiles	Do not use the standard system startup files for linking.
-nostdlib	Do not use the standard system startup files or libraries for linking.
-nostdinc	Do not search the standard system directories for header files.
-nostdlibinc	Do not search the standard library include directories. ($ROOT/$STDLIB/include)
-nobuiltininc	Do not search the builtin include directories. ($ROOT/lib/clang/x.y/include)

Last table shows parameters regarding the standard libraries.

Table 45: Compiler driver parameters regarding the standard libraties

Parameter	**Description**
-stdlib=$STDLIB	Set user specific standard library (e.g. uclibc). Default is newlib.
-print-file-name=$LIBRARY	Prints the full absolute name of the library file $LIBRARY that would be used when linking. This is useful when linking of standard library is disabled but some library is still needed (e.g. -print-file-name=libcomp.a). The obtained absolute name of the library can be manually passed to the linker.

19.5 GNU Binutils

The GNU Binutils are a collection of binary tools. The published SDK uses the Binutils compiled specifically for the Codasip target. The GNU linker (ld) has standard behaviour for Von Neumann architectures. The GNU assembler (as) has been replaced with generated proprietary assembler which has compatible syntax of the input source code but has incompatible command line interface.

19.5.1 Standard linker script

The GNU linker is compiled with standard linker script which is suitable for the most standard architectures. This script can be however overridden with user defined script passed with parameter `-T`.

The standard linker script contains three configurable symbols which are described in subsequent table.

Table 46: Linker script configurable symbols

Symbol	Description
_TEXT_START_ADDR	Base address of executable. (Default is 0)
_HEAP_SIZE	Size of heap. (Default is 0)
_STACK_SIZE	Size of stack in bytes. (Default is 0x2000)

These default values can be overridden with the linker parameter `--defsym=SYMBOL=value`. Note that this parameter can be passed also with clang as `clang -Wl,--defsym=SYMBOL=value`.

The majority of projects compiled with the newlib library also needs correctly configured heap for its proper function, so it is important to pass heap size to final linkage. It can be done as in this example:

Example 198:Heap configuration for newlib compiled projects

```
./PREFIX-clang -Wl,--defsym=_HEAP_SIZE=0x4000 main.c -lnosys
```

Example 199: Default linker script

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT ("elf64-codasip-le", "elf64-codasip-be", "elf64-codasip-le")
ENTRY (_start)
_TEXT_START_ADDR = DEFINED(_TEXT_START_ADDR) ? _TEXT_START_ADDR : 0x0;
_HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x0;
_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x2000;
SECTIONS
{
  . = _TEXT_START_ADDR;
  _ftext = .;
  .text : {
    KEEP (*(SORT(.crt*)))
    *(.text)
    *(.text.*)
    *(.gnu.linkonce.t.*)
  }
  _etext = .;
  .init : { KEEP (*(.init))} =0
  .fini : { KEEP (*(.fini))} =0
  PROVIDE (__CTOR_LIST__ = .);
  PROVIDE (___CTOR_LIST__ = .);
  .ctors : {
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  }
```

```
PROVIDE (__CTOR_END__ = .);
PROVIDE (___CTOR_END__ = .);
PROVIDE (__DTOR_LIST__ = .);
PROVIDE (___DTOR_LIST__ = .);
.dtors : {
  KEEP (*(SORT(.dtors.*)))
  KEEP (*(.dtors))
}
PROVIDE (__DTOR_END__ = .);
PROVIDE (___DTOR_END__ = .);
. = ALIGN(8);
_frodata = .;
.rodata : {
  *(.rodata)
  *(.rodata.*)
  *(.gnu.linkonce.r.*)
}
_erodata = .;
. = ALIGN(8);
_fdata = .;
.data : {
  *(.data)
  *(.gnu.linkonce.d.*)
}
_edata = .;
. = ALIGN(8);
.sdata : {
  *(.sdata)
  *(.sdata.*)
  *(.gnu.linkonce.s.*)
}
. = ALIGN(8);
.sbss : {
  PROVIDE (__sbss_start = .);
  *(.sbss)
  *(.sbss.*)
  *(.gnu.linkonce.sb.*)
  PROVIDE (__sbss_end = .);
}
. = ALIGN(8);
.bss : {
  PROVIDE (__bss_start = .);
  *(.bss)
  *(.bss.*)
  *(.gnu.linkonce.b.*)
  *(COMMON)
  . = ALIGN(8);
  PROVIDE (__bss_end = .);
}
. = ALIGN(8);
.heap : {
  _heap = .;
  _heap_start = .;
  . += _HEAP_SIZE;
  _heap_end = .;
}
. = ALIGN(8);
.stack : {
  _stack_end = .;
  . += _STACK_SIZE;
  . = ALIGN(8);
  _stack = .;
  _end = .;
}
```

```
}
```

19.6 Makefile example

This is a simple example of makefile that allows to build and run an application.

Example 200: Makefile that allows to build and run application with custom toolchain

```
TOOLSPATH=/home/user/codix_cobalt/bin
PREFIX=codix_cobalt-ia
APP=hello-world
CC=$(TOOLSPATH)/$(PREFIX)-clang
SIM=$(TOOLSPATH)/$(PREFIX)-isimulator
CFLAGS=-O0 -g3
LDLIBS=-lsim
SRC=main.c
OBJS=$(SRC:%.c=%.o)

.PHONY: all
all: $(APP)

%.o : %.c
	$(CC) $(CFLAGS) $(INCS) -c $< -o $@

$(APP) : $(OBJS)
	$(CC) $(OBJS) $(LDLIBS) -o $(APP)

.PHONY: clean
clean:
	$(RM) $(APP) $(OBJS)
.PHONY: run
run: all
	$(SIM) -i $(APP) -r $(SIMFLAGS) &2> /dev/null
	echo "Exit code: `cat sim_exit_code`"
```

20 ON-CHIP DEBUGGER

Codasip provides an easy way how to connect to the on-chip debugger (e.g. via JTAG interface using Nexus commands) placed in FPGA/ASIC or RTL simulator from Codasip Studio or Codasip Codespace. Thus, a user can debug software on the real implementation in the same environment and in the same way as he would do on a virtual platform. *JTAG Adapter* enables this behavior.

JTAG (Joint Test Access Group) is a synchronous serial interface used for testing, debugging and programing of a chip. It is described and defined by the IEEE Std. 1149.1-1990. JTAG interface enables controlling a chain of multiple integrated circuits (ICs).

Nexus is debugging interface for debugging of processors and multi-processor systems. It is described and defined by the IEEE-ISTO 5001-2003. It enables run-time control, memory access, breakpoints and more. Nexus usually uses JTAG as underlying transfer protocol.

20.1 JTAG Adapter Description

The JTAG Adapter communicates directly with the FPGA boards/RTL simulators and translates commands from the Codasip debugger into Nexus commands and into underlying JTAG data that are sent via JTAG interface. This enables designer to use the debugger to debug application running directly on the FPGA board.

Currently, there are multiple FPGA boards produced by many companies on the market. The JTAG Adapter is extensible using plugins that enable to add support for new FPGA boards and protocols.

There are two types of plugins. The first one, called *JTAG Adapter Plugin*, is used for different cables. The second one, *RTL Simulator Plugin*, is used for different RTL simulators.

There are two complete examples, which can be modified and used; one is for JTAG adapter plugin (`diligent.zip`) and one for RTL simulator plugin (`questa.zip`). You can find them in `<Codasip Studio/Codasip Codespace_`*`installation_directory`*`>/tools/src/jtagadapter/`.

20.2 JTAG Adapter Plugins

Each JTAG adapter plugin should implement communication using a single FPGA cable (or a family of cables). The following sections describe the plugin interface and usage.

20.2.1 Plugin discovery

JTAG adapter requires specification of the plugin to be used to communicate with the design. The JTAG adapter's `--plugin <name>` argument is used to select the plugin from all loaded plugins. To load any plugin implemented by a shared library (`.dll` or `.so`) add path of such library to the JTAG adapter arguments.

If plugin of given `name` is not found, JTAG adapter will try to find a library implementing the plugin in a shared library named `(lib)`**`name`**`plugin.so` (or `.dll` on Windows). First the working and then the executable directory of JTAG adapter will be searched.

Note that the plugin name is case-insensitive so at first the plugin name is used unchanged and if the plugin is not found the lower-case name will be used.

20.2.2 JTAG Plugin Interface

Each plugin is stored inside a shared library (Windows `.dll`, Linux `.so`) with a predefined interface. The interface is used by the JTAG Adapter to send JTAG data into the FPGA board and to return output data.

Plugin interface is declared in the C/C++ header file `jtag_plugin.h` that can be found in *`<Codasip Studio/Codasip Codespace installation>`*`/tools/include/`. This file also contains helper functions for simplification of plugin implementation.

20.2.2.1 Plugin C API

RegisterCodasipJTAGPlugin

```
CODASIP_EXPORT const CodasipPluginInfo* RegisterCodasipJTAGPlugin()
```

The only mandatory function the plugin (shared library) has to implement. The function is called from the JTAG Adapter during discovery of plugins and returns pointer to the `CodasipPluginInfo` structure that describes a single plugin. The `CODASIP_EXPORT` preprocessor macro is used for correct exporting of the function out of the shared library on Windows.

Returned structure pointer must be valid for entire initialization of the JTAG Adapter until plugin is selected and initialized.

CodasipPluginInfo

```
struct CodasipPluginInfo
```

Structure describing single plugin. Contains following properties:

- `m_Version` - The version of the plugin architecture. Use `CODASIP_PLUGIN_VERSION` preprocessor macro for automatic handling of the version.
- `m_Name` - A unique name of the plugin. User uses this name to select given plugin in JTAG Adapter. Plugin name requested by user is first searched in all registered plugins and if such plugin is not found, case-insensitive search is performed.
- `m_CreateFnc` - A callback for initialization of plugin-specific user data pointer. Returned pointer is then sent to all following plugin callbacks.
- `m_DestroyFnc` - A callback for destruction of plugin user data pointer.
- `m_InitFnc` - A callback for plugin initialization and configuration file loading.
- `m_SendRecvFnc` - A callback for sending binary JTAG data.
- `m_ErrorFnc` - A callback for retrieval of error message text.
- `m_HelpFnc` - A callback for function returning help string.

Plugin API and callback use binary buffer data (`BYTE* data`) for sending and returning binary JTAG data of given bit width. Such buffer has unified memory layout:

Table 47: Memory layout of binary buffer

data[0]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
data[1]	bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8
...								

Developer of plugins can use following helper functions for working with the binary buffer:

CodasipPluginGetBit

```
bool CodasipPluginGetBit(const BYTE* data, const unsigned position
```

- `data` - A pointer to the buffer containing binary data
- `position` - A bit position in the buffer, indexed from 0

Returns a bit value in the given `data` bit buffer at the given `position`.

CodasipPluginStoreBit

```
void CodasipPluginStoreBit(BYTE* data, const unsigned position, const bool bit)
```

- `data` - A pointer to the buffer containing binary data
- `position` - bit position in the buffer, indexed from 0

- `bit` - A new value for a bit at given position

Sets the bit in the given `data` bit buffer at the given `position` to the value `bit`.

20.2.2.2 Plugin lifetime

Main logic of the plugin is called in the callbacks that are registered into the JTAG Adapter as properties of the `CodasipPluginInfo` structure returned by the function `RegisterCodasipJTAGPlugin`. This function should not initialize plugin internal structures but only return a description of the plugin.

When a plugin is selected by the user, the JTAG Adapter initializes the plugin by `CreateFunc`. Afterwards the plugin communication is initialized by `InitFunc`. Then JTAG communication should be established and `DataFunc` callback will be called multiple times to send and receive JTAG data. Finally `DestroyFunc` is called to free all allocated memory and terminate the communication. When some error is reported during callback execution, an error message is retrieved by `ErrFunc` and printed out. When JTAG Adapter needs the help message, the plugin is initialized by `InitFunc` and then the `HelpFunc` is called.

20.2.2.3 Plugin callbacks

CreateFunc

```
void* CreateFunc()
```

Callback that is called to create the plugin. It returns a plugin-specific pointer that is sent to all following plugin callbacks as the `userData` argument. This function should initialize and allocate mandatory plugin structures but should not initialize JTAG communication.

If null pointer (`NULL` or `0`) is returned, plugin creation fails and the JTAG Adapter is terminated. `ErrFunc` callback won't be called because `userData` argument won't be set properly.

DestroyFunc

```
void DestroyFunc(void* userData)
```

- `userData` - A pointer returned from plugin initialization callback.

Callback that is called to destroy the plugin. It should free all allocated memory and terminate the JTAG communication if such communication was initialized using the `InitFunc` callback.

InitFunc

```
bool InitFunc(void* userData, const CodasipPluginConfig* config)
```

- `userData` - A pointer returned from the plugin initialization callback.
- `config` - A pointer to the structure containing configuration of the plugin. Cannot be `NULL`

The callback that is called to initialize JTAG communication. It returns success of initialization. If some error occurs, an error message will be retrieved using the `ErrFunc` callback.

The configuration structure contains the following properties:

- `int m_Argc` - Number of arguments in `m_Argv`
- `CodasipPluginArgument* m_Argv` - Array of arguments. Every item contains a flag and a optional value. The last item of the array is set to `NULL`.
    - `const char* m_Flag` - Name of a flag. Excluding `--` or `-` prefix, e.g. `--device=jtag` will result in the flag `"device"` and the value `"jtag"`.
    - `const char* m_Value` - Optional value for the flag. The value is argument after the flag that doesn't start with a colon and is not separated from the flag by `=`.

Callback should parse all configuration options, print a warning for unknown ones and return false if mandatory ones are not present. Afterwards, the JTAG communication itself should be initialized and success of the initialization returned from the callback.

After loading the configuration, an initialization of the communication will start by calling `DataFunc` repeatedly with JTAG data.

DataFunc

```
bool DataFunc(void* userData, BYTE* response, const BYTE* data, const unsigned bitsize)
```

- `userData` - A pointer returned from the plugin initialization callback.
- `response` - A pointer to the buffer to store `bitsize` bits at minimum that will be used to store JTAG output data from JTAG TDO pin.
- `data` - A pointer to the buffer containing binary JTAG data to be send. Contains pairs of TDI (lsb) and TMS (msb) bits to be sent. The bitwidth of the buffer is 2x `bitsize`.
- `bitsize` - Number of bits to be sent and received.

Callback that is called to send the JTAG data and receive a response. It Returns success of the communication.

The data parameter contains pairs of TDI and TMS bits to be sent. For every bit, TDO response has to be stored into output buffer response. The TDO bit should correspond with JTAG TAP state before new TDI and TMS bits are sent.

Example 201: Data function of JTAG plugin

```
for (unsigned i = 0; i < bitsize; i++)
{
    // retrieve input data
    bool tdi = CodasipPluginGetBit(data, 2*i);
    bool tms = CodasipPluginGetBit(data, 2*i+1);

    bool tdo = Send(tdi, tms);
    CodasipPluginStoreBit(response, i, tdo);
}
```

When an error is returned, the error message will be retrieved using `ErrFunc` callback, the JTAG Adapter destroys the plugin using `DestroyFunc`, and terminates itself.

ErrFunc

```
const char* ErrFunc (void* userData)
```

- `userData` - A pointer returned from the plugin initialization callback.

Returns an error message for the last failed callback execution. The returned pointer must be valid until the plugin is destroyed using `DestroyFunc`.

HelpFunc

```
const char* HelpFunc(void* userData)
```

- `userData` - A pointer returned from the plugin initialization callback.

Returns an help message of the plugin. It should describe every argument parsed in `InitFunc`. The returned pointer must be valid until the plugin is destroyed using `DestroyFunc`.

20.2.2.4 Plugin example

The following source file implements JTAG plugin *Example* that prints JTAG data to standard output.

Compile using following command (MinGW g++):

```
g++ plugin_main.cpp -I c:\Codasip\Studio-6.9.0\tools\include -
shared -fpic -o exampleplugin.dll
```

Example 202: Example plugin implementation file plugin_main.cpp

```
#include <cstdlib>   // NULL
#include <iostream>  // cout

#include "jtag_plugin.h"

///////////////////////////////////////////////////////////////////////////////////////////
struct ExamplePlugin
{
	// Initialization of plugin
	bool Initialize(const CodasipPluginConfig* config)
	{
		// STUB: perform some initialization and loading of configuration file
		return true;
	}

	// Storage for error message
	std::string m_Error;
};

///////////////////////////////////////////////////////////////////////////////////////////
void* PluginCreate()
{
	return static_cast<void*>(new ExamplePlugin());
}

///////////////////////////////////////////////////////////////////////////////////////////
bool PluginInit(void* userData, const CodasipPluginConfig* config)
{
	if (userData == NULL)
		return false;

	return static_cast<ExamplePlugin*>(userData)->Initialize(config);
}

///////////////////////////////////////////////////////////////////////////////////////////
void PluginDestroy(void* userData)
{
	delete static_cast<ExamplePlugin*>(userData);
}

///////////////////////////////////////////////////////////////////////////////////////////
bool PluginData(void* userData, BYTE* response, const BYTE* data, const unsigned bitsize)
{
	if (userData == NULL)
		return false;

	ExamplePlugin* plugin = static_cast<ExamplePlugin*>(userData);

	std::string tdi, tms;
	// binary buffer into binary string
	for (unsigned i = 0; i < bitsize; ++i)
	{
		tdi.insert(0, 1, CodasipPluginGetBit(data, 2*i) ? '1' : '0');
		tms.insert(0, 1, CodasipPluginGetBit(data, 2*i+1) ? '1' : '0');

		// set all bits in response to 0
		CodasipPluginStoreBit(data, i, false);
	}
	std::cout << "Received JTAG data\n: "
		"TDI:"  << tdi << "\n"
		"TMS:" << tms << "\n";
	return true;
}

//////////////////////////////////////////////////////////////////////////////////////////
const char* PluginError(void* userData)
{
	if (userData == NULL)
		return "Example plugin not initialized";


	ExamplePlugin* plugin = static_cast<ExamplePlugin*>(userData);
	return plugin->m_Error.c_str();
}

//////////////////////////////////////////////////////////////////////////////////////////
const char* PluginHelp(void* userData)
{
	if (userData == NULL)
		return "Example plugin not initialized";

	return "Example plugin\n"
		"Arguments:\n"
		"    --flag <string>  Custom flag.\n";
}

//////////////////////////////////////////////////////////////////////////////////////////
CodasipPluginInfo g_Plugin =
{
	CODASIP_PLUGIN_VERSION,
	"Example",
	PluginCreate,
	PluginDestroy,
	PluginInit,
	PluginData,
	PluginError,
	PluginHelp
};

//////////////////////////////////////////////////////////////////////////////////////////
extern "C" CODASIP_EXPORT const CodasipPluginInfo* RegisterCodasipJTAGPlugin()
{
	return &g_Plugin;
}
```

20.2.3 RTL Simulator Plugin

Codasip Studio/Codasip Codespace can be connected with external RTL simulators (for example Questa® Advanced Simulator) using Codasip JTAG adapter and its RTL simulator plugin. This is useful, because user can watch hardware signals when simulating model in the Codasip Studio/Codasip Codespace. The following sections describes API of the plugin and its usage.

20.2.3.1 Plugin interface

Each plugin is stored inside a shared library (Windows `.dll`, Linux `.so`) with a predefined interface.

Plugin interface is declared in the C/C++ header file `jtag_plugin.h` that can be found in `<Codasip Studio/Codasip Codespace` *`installation`*`>/tools/include/`.

20.2.3.2 Plugin C API

CodasipRtlPluginInfo

```
typedef struct
{
	unsigned m_Version;
	CodasipRtlPluginCreateFunc m_CreateFnc;
	CodasipRtlPluginDestroyFunc m_DestroyFnc;
	CodasipRtlPluginSetValueFunc m_SetValueFnc;
	CodasipRtlPluginGetValueFunc m_GetValueFnc;
	CodasipRtlPluginPrintFunc m_PrintFnc;

	CodasipRtlPluginCallbackFunc m_CallbackFnc;
	CodasipRtlPluginSetErrFunc m_SetErrorFnc;
	char* m_PathToJtag;
} CodasipRtlPluginInfo;
```

This structure contains all communication functions between the plugin and the rest of the JTAG adapter (except for "Start" on page 397 function). First six items must be defined in the plugin (see "Output functions" on page 398), the rest will be defined automatically in the "Start" on page 397 function (see "Input functions" on page 398). You should create this struct in the plugin and pass it as a parameter to the `Start` function. This API also defines datatype codasip_sigval_t, which is used for read signal values.

20.2.3.3 Initialization

Start

```
bool Start (CodasipRtlPluginInfo* plugin);
```

This function is automatically imported to the plugin. The parameter `plugin` has to point to a valid structure with assigned plugin function pointers. This structure is described in the "CodasipRtlPluginInfo" on page 397. The function fills the plugin structure with output function pointers and starts a new thread for the debugger and the JTAG adapter. It also instantiates the plugin using its create function. `m_PathToJtag` member of this struct is also initialized during this function. It's value is an empty string or a value of the parameter path-to-jtag if it was used. It must be used by their plugin to correctly detect the CLK signal of the model.

If some error occurred in this function, all allocated resources are freed and all created threads are terminated, so there is no need for additional cleanup. There is also no guarantee that the `plugin` structure is fully initialized, so no functions should be called from JTAG adapter if this happens. The function returns true when there was no error, otherwise it returns false.

20.2.3.4 Input functions

The JTAG adapter provides the following interface functions to the user (inside the `"CodasipRtlPluginInfo" on page 397`). The pointers to these functions will be set in the `Start`initialization function.

CallbackFnc

```
int CallbackFnc();
```

You should call this function from the simulation thread every time the CLK signal of the model changes it's value. It returns 0 if the simulation should continue, 1 if the simulation should stop or 2 if the simulation should finish. Stop or finish is determined by the parameter sim-exit. If it is used, 2 is returned from the function, 1 otherwise. The plugin is not required to implement different behavior for this two values, but it is recommended if possible.

SetErrorFnc

```
int SetErrorFnc(const char* message);
```

This function propagates error message from the plugin to the rest of the JTAG adapter. This causes the debugger thread to terminate before this function returns, freeing correctly all it's resources. It does not call the plugin print function, so error message should be printed in the plugin. Return value and required action to it is the same as in `CallbackFnc`, but 0 will never be returned. This function can't be called before `Start` function and also shouldn't be called if `CallbackFnc` returns a non-zero value, because in that case, debugger thread is already correctly terminated.

20.2.3.5 Output functions

The user have to define five functions callable from the debugger and fill them inside the `"CodasipRtlPluginInfo" on page 397` structure.

CodasipRtlPluginCreateFunc

```
typedef void* (*CodasipRtlPluginCreateFunc)(int* result);
```

This function should create and initialize the plugin. It is called from the `Start` function and returns pointer to any specified user data (usually it is a pointer to a plugin instance). If no user data is necessary, function can return NULL. Integer pointed by result parameter should be set to 0 if no error ocured or to -1 otherwise.

CodasipRtlPluginDestroyFunc

```
typedef void (*CodasipRtlPluginDestroyFunc)(void* userData);
```

This function is called from the debugger thread when an error occurs or when the simulation process was terminated. The parameter `userData` is the pointer that the `CodasipRtlPluginCreateFunc` returned. It should free all resources allocated by the `CodasipRtlPluginCreateFun`.

CodasipRtlPluginPrintFunc

```
typedef void (*CodasipRtlPluginPrintFunc)(void* userData, const char* message);
```

This function is called from the debugger thread when a message is required to be printed in the simulator. Usually this happens during an error or exit and same message is usually printed to the standard error output. The parameter `userData` is the pointer that the `CodasipRtlPluginCreateFunc` returned and the `message` is the text to be printed. This function is not required to be implemented. If that is the case, NULL should be placed instead of this function pointer to the "CodasipRtlPluginInfo" on page 397 struct.

CodasipRtlPluginSetValueFunc

```
typedef int (*CodasipRtlPluginSetValueFunc)(void* userData, const char* name, const int val,const
SetMode mode );
```

This function is called during the "CodasipRtlPluginInfo" on page 397 every time a signal is required to change its value. The parameter `userData` is the pointer that the `CodasipRtlPluginCreateFunc` returned. The`name` is a path to the signal to be set. Signal path is concatenated from a prefix set by the parameter `path-to-jtag` and a name of a requested signal which can also be specified by plugin parameter. The `val` is a new value for the signal. Valid values are 0 and 1 for the logical value of the signal. For success it should return 0, for error it should return -1. The parameter `mode`can have one of three values: `NORMAL`, `FORCE`, `RELEASE`. `NORMAL` mode tells the plugin that new value should be normally stored to signal without any special actions or regard to simulation. `FORCE`mode tells the plugin that it is possible for the signal value to be overriten internally by the simulation itself. From this call, signal should be locked from other changes from simulation until the `RELEASE`mode is applied to the same signal. `RELEASE`mode ignores new signal value, it only makes the signal changeable from simulation again. These modes are typically used when there is a signal, which can be read or set by JTAG adapter according to timing or different conditions.

CodasipRtlPluginGetValueFunc

```
typedef codasip_sigval_t (*CodasipRtlPluginGetValueFunc)(void* userData, const char* name);
```

This function is called during the `"CodasipRtlPluginInfo" on page 397` every time a current signal value is required to be read. The parameter `userData` is the pointer that the `CodasipRtlPluginCreateFunc` returned. The `name` is a path to the signal to be read. Signal path is concatenated from the parameter `path-to-jtag` and a name of requested signal which can also be specified by plugin parameter. For success it should return current value of given signal (0 or 1), for error it should return -1.

20.2.3.6 Example

Example 203: Full example of JTAG Adapter

```
#include "jtag_plugin.h"

#include <string>

CodasipRtlPluginInfo g_PluginInfo;


void* PluginCreate(int* result)
{

    if (result == NULL)
    {
        return NULL;
    }
    try
    {
        *result = 0;// Plugin initialization (none needed in this example)
        return NULL; // No user data
    }
    catch (std::exception& ex)
    {
        *result = -1;
        return NULL;
    }
}

void PluginDestroy(void* userData)
{
       // Destruction of plugin instance, resource dealocation
}
```

```
void PluginPrint(void* userData, const char* message)
{
       // Print message to simulator
}

int PluginSetValue (void* userData, const char* name, const int val, const SetMode mode)
{
       // Find signal by given name
       // Set value val into it according to mode
       // Return 0 or -1 on error
}

int PluginGetValue (void* userData, const char* name)
{
       // Find signal by given name
       // Read value from signal
       // Return this value or -1 on error
}

CodasipRtlPluginInfo g_PluginInfo =
{
       CODASIP_RTL_PLUGIN_VERSION,
       PluginCreate,
       PluginDestroy,
       PluginSetValue,
       PluginGetValue,
       PluginPrint,
       NULL,
       NULL,
       NULL
};

void StopSim (const int exit)
{
       if (exit == 1)
       {
               // Stop simulation
       }
       else // exit == 2
       {
               // Finish simulation
       }
}

void Callback ()
{
       int result = g_PluginInfo.m_CallbackFnc();
       if (result != 0)
       {
               StopSim(result);
       }
}

void FirstCalledFunction ()
{
       if (!Start(&g_PluginInfo))
       {
               // g_PluginInfo not completly initialize
               // Print error
               // Finish simulation
               return;
       }
```

```
	try
	{

		// Find signal with path g_PluginInfo.m_PathToClk
		// Create callback, which activates everytime this signal changes
		// Make function Callback run when this happens
	}
	catch (std::exception& e)
	{
		// Print error message (recomended e.what())
		int result = g_PluginInfo.m_SetErrorFnc("Initialization error");
		StopSim(result);
	}
}
```

21 CODASIP TESTSUITE

One of the most important tasks in the processor design is testing. Therefore, Codasip Studio comes with a testsuite that is able to stress the generated compiler or certain features of the processor. Integrated Testsuite consists of a large number of compiler tests. It can be extended by the user by adding specific tests for the processor.

Codasip Testsuite is based on the Python *pytest* framework. For complete information about the pytest and it usage, you can see Pytest documentation.

You can find more information about the testsuite and examples of use in *Codasip Studio User Guide*, chapter "Codasip Testsuite".

21.1 Usage of Codasip Testsuite

21.1.1 Execution using Command Line

Testsuite can be executed by the following command:

Example 204: Testsuite execution

```
bin/cmdline -m codasip.testsuite <test suite arguments>
```

Arguments:

- `--work-dir` - working directory of testsuite; by default working directory of process
- `--sdk` - list of directories with SDK.
- `--hdk` - list of directories with HDK.
- `--opt` - (`0,1,2,3,s,z`) - optimization levels to run
- `--design-path` - (e.g. `codasip_urisc.ia,codasip_urisc.ca` - simulators to use for simulation; if not set, all found will be used.
- `--sim-timeout` - default timeout for simulator runs in seconds; default is 60
- `--save-passed` - save the temporary files of passed tests

Some standard pytest arguments that can be used along with custom arguments:

- `-n=auto` - number of parallel executions; by default all available; or sequential build `-n0` can be used
- `-m <mark>` - run only tests with given mark
- `-k <regex>` - run only tests matching given regex
- `--help` - show help screen

Complete list of arguments can be found in Pytest documetation.

21.1.2 Execution through Codasip Build System Task

The build system contains Tasks **Testing (ia)** and **Testing (ca)**. These Tasks use testsuite Python API to run the testsuite with *ia* or *ca* tools (e.g. simulator, profiler).

The testsuite running with this Task uses `<model>/work/tests/` folders in the project directory. It also automatically uses the default compiler testsuite.

21.1.3 Execution from Python

The testsuite can be run directly form Python. This approach may be used when you want to integrate the Codasip testuite into a bigger testsuite.

Example 205: Execution of testsuite with Python

```
from codasip.testsuite import Testsuite
suite = Testsuite(
    # directory with tests
    test_dir=os.path.join(model_dir, 'test'),
    # list of directories with SDKs, containing sdk.xml
    sdk_dir=...,
    # list of directories with HDKs, containing sdk.xml
    hdk_dir=...,
    # working directory of testsuite, will contain report.xml
    work)dir=...)

# run testsuite
suite.run()

# run testsuite with custom arguments
suite.run('--sim=codix_berkelium_top.ia')
```

21.2 Configuration of Testsuite Execution

Testsuite can be configured using arguments directly or using a configuration file. Testsuite uses default Pytest configuration file called `pytest.ini`. The testsuite looks for it in `<model>/tests/` folder. The argument `-c` can be used to set the file directly.

Example 206: Example of testsuite configuration in `<model>/tests/pytest.ini`

```
[pytest]

# run all tests with double substring on optimization -O0 and -O3 and do not run any uvm tests.
addopts = -k double -m "not uvm" --opt=0,3
```

21.3 Tests, Marks, and Fixtures

One of the common tasks is to extend the current testsuite by additional tests. Each test has the following form.

Example 207: Test structure

```
<marks>
def test_<name>(<fixture_1>, ..., <fixture_n>)
    <test itself>
```

The following example shows a test that uses simulator and compiler fixtures. Marks limits the test to a simple optimization level -O3.

Example 208: Test example

```
@pytest.mark.compiler(optimization=[3])
def test_basics(self, compiler, simulator):
    f = compiler.run('basics.c')

    print('Assembly: ', f.assembly)
    print('Output: ', f.output)

    sim = simulator.run(f.output)
    print('Simulator exit code: ', sim.exit_code)
    print('Application exit code: ', sim.app_exit_code)
    print('Cycles: ', sim.cycles)
    print('Profiler: ', sim.profiler_output)
```

21.3.1 Tests

Tests are functions with prefix `test_` located in python files with the same prefix `test_` (e.g. `test_setjmp.py`). The test cases can be grouped into classes.

Every test function may have multiple arguments. Every such argument is dependency of the test, called *fixture* (see "Fixtures" on page 407). The test case can be instantiated multiple times, e.g. for every simulator, optimization level etc. This depends on used fixtures. The test case can also be marked so not all the possibilities of fixtures are used. E.g. only IA simulator, only single optimization, etc.

Example 209: Created tests

```
@pytest.mark.compiler(optimization=[0,3,'s'])
def test_basics(self, compiler, simulator):
    f = compiler.run('basics.c')

    print('Assembly: ', f.assembly)
    print('Output: ', f.output)

    sim = simulator.run(f.output)
    print('Simulator exit code: ', sim.exit_code)
    print('Application exit code: ', sim.app_exit_code)
    print('Cycles: ', sim.cycles)
    print('Profiler: ', sim.profiler_output)

def test_sims_only(self, simulator):
    print str(simulator)

def test_class_options(self, compiler, simulator):
    f = compiler.run('basics.c')
    sim = simulator.run(f.output)
    print sim
```

```
@pytest.mark.compiler(optimization=1)
def test_mark_on_class(self, compiler):
    assert compiler.optimization == 1

@pytest.mark.xfail
def test_fail(self, compiler, simulator):
    f = compiler.run('basics.c')
    simulator.run(f.output + '-error')
```

21.3.2 Marks

Testsuite (Pytest) uses marks to annotate testcases so they are run only if some conditions succeed. The mark can be selected on a command line or in `pytest.ini` file. Both approaches use the argument `-m"<expression>"`, e.g. `-m "not compiler"` will deselect all compiler testsuite tests.

Each mark has the following syntax.

```
@pytest.mark.<mark>(optional arguments)
```

There are several build-in marks that you can use.

Mark	**Description**
`@pytest.mark.xfail`	Used when a test should always fail

`@pytest.mark.compiler`	A test is run when compiler has certain features or for a given optimization levels. The test below is run when a copiler has newlib as well as compiler-rt. Furthermore, the test is run on -O0, -O3, and -Oz.
	`@pytest.mark.compiler(newlib=True,`
	`                      compiler_rt=True,`
	`                      optimization=[0,3,'z'])`
	`def test_basics(compiler, simulator):`
	`    ....`

`@pytest.mark.uvm`	Automatically marked to all tests that uses **uvm** fixture. Can be used to disable all UVM tests.

`@pytest.mark.simulator`	Run the test when a simulator has certain features. Example below select an IA simulator with a dumping and profiling feature.
	`@pytest.mark.simulator(ia=True`
	`                       dump=True,`
	`                       profiler=True)`
	`def test_basics(compiler, simulator):`
	`    ...`

`@pytest.mark.model_configuration`	When a test requires specific model configuration, then this mark is used. Note that all specified options have to match for a test execution.
	`@pytest.mark.model_configuration(OPTION_UARCH_NAME='bk3', OPTION_XLEN=[32,64])`
	`def test_basics(compiler, simulator):`
	`    ....`

`@pytest.mark.find_files`	Collect files matching the pattern and run test for each file. File search will be performed in the directory where source file is. The file path is available as a fixture named `file`.
	`@pytest.mark.find_files(pattern='\.c$')`
	`def test_basics(self, file):`
	`    ....`

`@pytest.mark.directory_collect`	Collect all files in the leaf nodes of a file tree matching the pattern and run test for each file. Files will be matched only if there is no subdirectory File search will be performed in the directory where source file is. The file path is available as a fixture named `file`.
	`@pytest.mark.find_files(pattern='\.c$')`
	`def test_basics(self, file):`
	`    ....`

21.3.3 Fixtures

Fixture is a prerequisite for a test case. In general, a fixture is a tool that is used for testing. So, for instance, a fixture can be compiler or profiler. There are also fixtures that collects files or do other useful work. The test case may use more than one fixture at the time. Fixtures are specified as arguments of a test case.

21.3.3.1 work_dir

Working directory for the test case. All temporary files for the test should be put inside it. It is automatically cleaned before test-case execution. If the test fails, it will be copied into `failed` folder under testsuite working directory.

Example 210: work_dir Fixture

```
def test_work_file(work_dir):
   # full path to file
   ifile = os.path.join(work_dir, 'test.input')
   # generate input of the file
   with open(ifile, 'w') as f:
       f.write('123')
```

21.3.3.2 compiler

Instantiate a compiler that matches filters specified by the compiler mark. The instantiated compiler can be executed using `run()` method. It has the following arguments:

- `sources` - sources to compile
- `cflags` - additional flags for the compiler
- `ldflags` - additional arguments for a linker
- `optimization` - optimization level
- `temps` - if true, temporal files are also stored

It returns a result structure that has the following attributes:

- `output` - output executable
- `assembly` - output assembly (only if `temps=True`)

The method throws `ToolError` on error. The sources will be linked into an output binary. If it is used in the testsuite, the sources can be a relative path to the test location..

Example 211: compiler Fixture

```
def test_basics(self, compiler):
    f = compiler.run('basics.c')
```

21.3.3.3 assembler

Instantiate an assembler for running the assembler. The instantiated assembler can be run using `run()` method. It has the following arguments:

- `sources` - sources for a compilation
- `flags` - additional flags
- `ldflags` - additional flags for a linker
- `temps` - if true, temporal files are also stored
- `link` - if `True`, then the linker is used after assembling, so the executable is created. If link is not `True`, only object files will be created.

It returns a result structure that has the following attributes:

- `output` - output executable
- `obj` - output object files

It throws `ToolError` on error. The sources will be linked into an output binary. If used in testsuite, the sources can be a relative path to the test location.

Example 212: assembler Fixture

```
def test_basics(self, assembler):
    f = assembler.run('basics.s')
```

21.3.3.4 disassembler

Instantiate disassembler. The instantiated disassmebler can be run using `run()` method. It has the following arguments:

- `input` - input file
- `flags` - additional flags for disassembler

It returns the output file. It throws `ToolError` on error. If used in a testsuite, the sources can be a relative path to the test location.

Example 213: disassembler Fixture

```
def test_basics(self, assembler, disassembler):
    ar = assembler.run('basic.s')
    dr = disassembler.run(ar.output)
```

21.3.3.5 simulator

Instantiate a simulator that matches filters specified by the simulator mark. The instantiated simulator can be executed using `run()` method. It has the following arguments:

- `apps`- one application or a dictionary where a key is a processor name and value is an application
- `args`- list of arguments to be sent to the simulator
- `cwd`- curent working directory for a simulator
- `ignore_app_exit_code` - if ture, then there is no exception when a simulated application has exit code different than zero

It throws `ToolError` on failure. It returns a result structure that has the following attributes:

- `exit_code` - exit code of the simulation.
- `app_exit_code` - exit code of the simulated application.
- `cycles` - number of cycles of the simulation
- `profiler_output` - file with profiling data or `None`

Example.

Example 214: simulator Fixture

```
def test_basics(self, compiler, simulator):
    rc = compiler.run('basics.c')
    rs = simulator.run(rc.output)
```

21.3.3.6 debugger

Instantiate a simulator with debugger interface. Underlying simulator must be built with debugger support. Debugger is started using the `start()` method.

Example 215: debugger Fixture

```
def test_basics(self, compiler, debugger):
    rc = compiler.run('basics.c')
    rs = debugger.start(executable=c.output, source_file='basics.c', line=5)
    debugger.check('-var-set-format var1 binary',
                   '^done,format="binary",value="1"')
    debugger.exit()
```

Debugger `start()` method has the following arguments:

- `executable`- Binary file to be executed
- `source_file`- optional source file to be loaded in debugger
- `line`- initial breakpoint

Using the `check()` method user can send a command to debugger. First argument is a command. Second argument is list of regular expressions the output of debugger will be matched to.

In case of failure(timeout, pattern mismatch) the `ToolError` is raised.

21.3.3.7 uvm

Instantiate an UVM testbench. The instantiated UVM testbench can be run using `run()` method. It has the following arguments:

- `apps` - path to a single application or a dictionary where the processor name is a key and the path to a single application is value
- `args` - additioanl arguments for UVM testbench

It throws `ToolError` on failure or if any of them fails. It returns a dictionary of an application (key) to the result of verification (value) for the given application. The value is a dictionary with given items:

- `result` - string `"OK"` if everything was ok, otherwise `"FAIL"` or other error messages
- `gm_cycles` - exit code of the simulated application

- `cycles` - number of cycles of the simulation
- `report` - additional report data, contains mismatches or failed assertions.

Example 216: uvm Fixture

```
def test_basics(self, compiler, uvm):
    rc = compiler.run('basics.c')
    ru = uvm.run(rc.output)
```

21.3.4 Helpers

The testsuite contains custom helpers to be used in the test cases.

```
from codasip.testsuite.helpers import *
```

```
list = find_files(path, pattern, exclude=DEFAULT_EXCLUDE)
```

Helper return all files in a given path whose path matches a given regular expression (Python `re` expressions) with given exclusions (Python `re` expressions).

22 CONFIGURABLE MODELS

A CodAL project can be set up as configurable or static. The default configuration of a project is static but it can be changed later. A configurable project contains a file with declarative meta-language where model options and their dependencies are defined.

The current model configuration is done in a separate file where model options are set or a configuration string is used. The configuration string is a symbolic options representation which is alternative to direct use of options. Some options may have an alias which is one or more symbols at a specific position in the configuration string.

From the current model configuration, a header file is then generated where all supported options are stored in the form of preprocessor macro definitions. This header is then automatically included in the model source files which are supported by the Codasip Studio. Standard preprocessor constructs can be used in this source files to alter the code with current configuration.

22.1 Configuration Files

The project configuration is represented by several files. The mandatory file `options.conf` describes list of available options and the others are used for current configuration. The list of files used for configuration is following:

- `options.conf` - Mandatory file with declarative meta-language which describes available model options and their default values. Without this file the CodAL project cannot be configured and is treated as static.
- `ip.conf` (optional) - A file with current model configuration. The configuration can be set by direct use of options defined in the `options.conf` or by using configuration string which is represented by list of option aliases. If all options in the `options.conf` have defined default value the `ip.conf` can be omitted.
- `codal.conf` - A configuration of external models (ASIP, Level) is done in the `extern` section. The body of this section(when `codal.conf` is opened in a text editor)is the same as in the `ip.conf`. In the case that an external model is instantiating the settings, the settings from the `extern` section is used instead of settings stored in the `ip.conf` of the current project.

22.2 Options Metalanguage (options.conf)

All options supported by the model are described in the file `options.conf`. This file is written in a TOML format (Github:TOML) with some Codasip extensions. In this format the declarative meta-language for options description with their dependencies and constraints is stored. Moreover, this file can optionally provide a pattern for definition of configuration string which can be used as an alternative for direct setting of options.

`options.conf` contains list of all supported options and their attributes. These options can be grouped in a pattern which is the list of options. The following example shows a description of very simple configurable model to provide basic overview of the meta-language structure:

Example 217: options.conf structure

```
pattern = [
       UARCH_NAME, # (uarch1|uarch2)
       DELIMITER,  # -
       WSIZE       # (32|64)
       ]

[DELIMITER]
       values = [ "-" ]
       type = delimiter
       default = "-"

[UARCH_NAME]
       values = [ "ur1", "ur2" ]
       type = string

[UARCH_UR2]
       type = expression
       default = "UARCH_UR2 == 'ur2'"

[WSIZE]
       values = { 32 = "32", 64 = "64" }
       type = uint
       format = "%d"
       default = 32
       dependency = "UARCH_UR2"
       help = “Word size”
```

22.2.1 Options Description

The options are represented in the TOML as a named table with several attributes. The name of the table corresponds to the name of the option. Its attributes are intended for the option description where some attributes are mandatory and some are optional. The description includes e.g. a type of the option, relations among other options, enumeration of allowed values, and a message with the option specification that can be used in a configuration dialog as help for users.

22.2.1.1 Attributes

type

The attribute `type` is mandatory and represents the type of an option value. Standard and special types are supported. Their enumeration is in the following list:

- `bool` - A standard boolean type which can be true or false. This type is useful for switches that mark if an option is enabled or disabled.
- `int/uint` - Signed or unsigned integer. It is useful e.g. for configuration of memory size, bus addresses, ...
- `string` - This type serves mainly for auxiliary options used for evaluation of more complex expressions in other options. It can also be used for options used in the `codal.conf`. For a CodAL model it can be used only for printing some messages.
- `expression` - The type is retrieved after evaluation of an expression from a default attribute. The expression is a string with Python expression syntax.
- `delimiter` - It is a special type used only for the configuration string. More details are given in the "Pattern Description" on page 418 section.

default (optional)

This optional attribute is used for setting a default value for an option. This value will be used when user does not specify a different value. The type of default value has to correlate with the type specified in the `type` attribute.

For the `expression` type, the default value is a string with the Python expression. In this expression it is possible to use other options referenced by its name. When the expression raised the exception the default value is set to `None`. If the expression has been successfully evaluated, the type of the result is used as a type of the whole option.

values (optional)

The attribute `values` has several purposes. The main is to limit the number of possible values that the option can be set with. These values can be further limited by some condition. The second purpose is to set a value alias which is used in pattern for the configuration string.

This attribute can be written in two forms. The basic form is a simple list of possible values. The type of these values have to correspond with the type of this option. When the option has the string type then values are automatically treated as aliases too. In this simple form of values, it is not possible add any other conditions. The following example shows its usage:

Example 218: Use of default, type and values attributes

```
# this option can be set only with values “ur1” and “ur2”
# the `ip.conf` has to specify one of them because this option
# does not have any default value set
# this option has defined aliases too because it has the string type
# these aliases will appear in configuration string e.g. `ur1-32`
[UARCH_NAME]
      values = [ "ur1", "ur2" ]
      type = string

# this option can be set only with values `128` and `256`
# the default value is 128
[CACHE_LINE_SIZE]
      values = [ 128, 256 ]
      type = uint
      default = 128
```

The more complex form is values definition with conditions and its aliases. In the TOML format it is written as a table/dictionary where the key is a possible option value and the value pair represents an alias with a condition. The alias with the condition has the two following forms:

```
"alias [if `condition`]"
```

```
"[if `condition`]"
```

The condition is optional, but if alias is not provided, it is treated as an empty string in the configuration string. The empty alias is useful when it is needed to express a disabled feature. For this feature no character exists in the configuration string. This will be shown in the following example:

Example 219: Use of conditions in values attribute

```
# for short record of values inline table is used
# the numbers are aliased with equivalent strings
# the option can be set with value `64` when `UARCH_UR2` is `true`
[WSIZE]
      values = { 32 = "32", 64 = "64 if UARCH_UR2" }
      type = uint

# this option can be enabled or disabled
# when alias `A` is used in the configuration string this option is
# enabled if no character on the specific position is entered then
# this option is disabled
# e.g. `ur1-32A` - enabled, `ur1-32` - disabled
[FEATURE1]
      values = { false = "", true = "A" }
      type = bool

# it is more convenient to use standard TOML table for multiple
# values which does not fit one line
# when alias is empty string for value `B` it is still possible
# to set the condition
[FEATURE2.values]
      B = "if not UARCH_UR1"
      C = "U"
      D = "S if UARCH_UR2
[FEATURE2]
      type = string
```

Note: The limitation of the TOML format causes that the key in the dictionary for options with the string type have to be written as an identifier and not as a string.

The values and aliases must be unique and none of them can be used more than once in the same list. It denotes that only one value can be mapped to an empty alias.

If options have the `default` value set and the `values` attribute is set too then this value have to from the same domain. It is not possible to set a default value with a value which is not specified in the `values` attribute.

dependency (optional)

The `dependency` attribute is used for an expression definition that a specifies condition in which it s possible to change the attribute `default` . In order to ensure correct behavior, it is also necessary to set a default value for the option.

The dependency is usually expressed as a logical expression of related options but it accepts any Python expression whose result can be interpreted as a boolean value. When a dependency condition is true the option can be set to any relevant value. So when a dependency is fulfilled and the option type is bool then the value of the opiton can be set to true or false in the `ip.conf` or in `extern` section of `codal.conf`. When a dependency is not met then a default value is used and it cannot be overridden in any way.

select (optional)

When some option needs to enable other dependent options then the attribute `select` can be used. It is possible to select only options with the boolean type because its value is set to true when selected.

The `select` attribute is a list of other dependent options which are set to true when the option `value` with the select attribute is also set to true. When this option is set to false then these related options are left unchanged.

Whenever some option is selected by other option then it is not possible to set this value to false in the `ip.conf` or in `extern` section of `codal.conf`. These options are forced to be enabled in any case.

The following example shows how the `select` attribute works:

Example 220: Use of select attribute

```
[UARCH_UR1]
      type = expression
      default = "UARCH_UR2 == 'ur1'"
      select = [ ENABLE_FEATURE1, ENABLE_FEATURE2 ]

# when `UARCH_UR1` is `true` then this option is set to `true`
# otherwise the value is left to `false` or can be changed
# in the `ip.conf`
[ENABLE_FEATURE1]
```

```
       type = bool
       default = false

# this option does not have any default value and must be
# specified in the `ip.conf`
# when `UARCH_UR1` is `true` then this option is set to `true`
# and cannot be specified in the `ip.conf`
[ENABLE_FEATURE2]
       type = bool
```

format (optional)

The `format` attribute can be used only in conjunction with the integer `type` option. It defines how the integer is printed to the generated header file. E.g. when the CodAL model expects unsigned long long literal the format can specify suffix `ull` to satisfy it.

Another use is for generated include files which are placed in the model directory `/libs/include/` as a template. The header file can contain predefined macros with the `OPTION_` macros on the right side. All occurrences of these `OPTION_` macros are replaced during export of this header file to the SDK with actual values. The format can specify how these values look like. E.g. the number can be printed as a hexadecimal value prefixed with 0x. The following example shows this format:

Example 221:Use of format attribute

```
[COMPONENT_BASE]
       Type = uint
       format = "0x%x"
       default = 0x1000
```

An expression used for `format` is the same as format used for `printf` or old format style for the Python operator %. Use Python2.7: String Formating documentation to correctly set the format string.

help (optional)

The `help` attribute is set with a string which can be a single line or multiline. This string is then used as a help information in the GUI dialog. It is recommended to use this field instead of simple comment associated with the option.

A single line string starts and ends with single quote character ":

Example 222: Use of help attribute (single-line)

```
help = "single line string"
```

A multi-line string starts and ends with three consecutive quotes """. A text inside the quotes should start at a separate line because whitespaces from the first line are removed from all following lines. This allow creation of a multi-line string which is properly formatted on the source level:

Example 223: Use of help attribute (multi-line) help

```
help = “””
       four spaces before this text specify how are following lines cut
       * this line starts with asterix
         - this line starts with two spaces
“””
```

22.2.2 Pattern Description

The pattern description is a simple list of options with value aliases. This list represents matching pattern for parsing and reverse reconstruction of a configuration string. The position in this list represents position of the aliased value in the configuration string. The representation of this list in a options meta-language is following:

```
pattern = [ OPTION1, OPTION2, …, OPTIONn ]
```

The pattern list can also contain special option with a delimiter type. It is possible to use the same delimiter more than once or create more delimiters with various characters. If delimiter does not separate proper options, it is removed from the configuration string. It eliminates side-by-side delimiters or delimiters at the beginning or at the end of the configuration string when every proper options are reduced to an empty string.

Help for GUI is generated from the pattern description and some rules should be respected to get the expected result. Every position in the pattern is represented by one option with aliases that can be categorized to the following groups:

1. Mandatory list of aliases (one of possibilities must be selected)
    a. List of two: `(ALIAS1|ALIAS2)`
    b. General list: `(ALIAS1|ALIAS2|ALIAS3|...)`
2. Optional list of aliases (none or one of possibilities can be selected)
    a. List of two (switch): `[ALIAS1]`
    b. General list: `[ALIAS1|ALIAS2|...]`

Every group has two special categories because an alias can be only a string but the option has usually bool type and it needs to be converted to the value *true* (enabled) or *false* (disabled). The optional list is a special case of mandatory list where one value has an alias set to empty string. It is the reason why the optional list of two aliases(2a.) has only one alias in the list. The second alias is represented by square brackets which stands for optional selection.

The following examples shows all individual groups:

Example 224: Use of mandatory and optional aliases

```
# 1a/ Mandatory list of two values
[FEATURE1A]
       values = { false = “ALIAS1”, true = “ALIAS2” }
       type = bool
```

```
# 1b/ Unbounded mandatory list of values
[FEATURE1B]
       values = [ “ALIAS1”, “ALIAS2”, “ALIAS3” ]
       type = string

[FEATURE1B_ALIAS1]
       # conversion `ALIAS1` to bool option
       type = expression
       default = “FEATURE1B == ‘ALIAS1’”

# 2a/ Optional list of two values
[FEATURE2A]
       values = { false = “”, true = “ALIAS1” }
       type = bool

# 2b/ Unbounded optional list of values
[FEATURE2B.values]
       REMOVE = “”
       VALUE1 = “ALIAS1”
       VALUE2 = “ALIAS2”
       [FEATURE2B]
       type = string

[FEATURE2B_ALIAS1]
# conversion `ALIAS1` to bool option
       type = expression
       default = “FEATURE2B == VALUE1”
```

22.3 Configuration of Current Model (ip.conf)

Current model configuration is set in the file `ip.conf` which is placed in the root directory of the CodAL project. This configuration is valid only in this project but when the project is used as an external reference in another project (usually Level project) the configuration file is ignored and the setting form `extern` section of `codal.conf` is used instead.

The `ip.conf` consists of two parts which are both optional. The first is configuration attribute which can contain configuration string if a pattern with aliased options exists in the `options.conf` file. The second part is a list of options with their value which overrides default values or sets new values when no default value is set in the `options.conf` file.

The configuration file is stored in a standard TOML format:

Example 225: Current configuration (ip.conf)

```
# comment
configuration = “configuration-string”

[options]
       INT = 0x1000
       BOOL = true
       STR = “string”
       LIST = [ “item1”, “item2” ]
       # in the ‘ip.conf’ it is not possible to use option value as
       # a right side of the assignment (issue with recursion)
       # FOO = OPTION_FOO
```

When the `ip.conf` contains incorrect configuration an error is shown with extended information. The wrong configuration can be caused by:

- Missing value for an option without a default value.
- Incorrect combination of incompatible options. The options may have some dependencies or conditions and not all combinations are correct.
- The `options` section is in conflict with configuration string. It is not possible to override configuration string by changing a specific option in the `options` section.
- The configuration string is incorrect:
    - Invalid order of aliases. The order is specified by the pattern list of options in the `options.conf`.
    - Use of delimiter string which is followed by another delimiter or by an empty string. The delimiter has to separate consecutive option aliases.
    - Use of an unsupported alias.
    - Incorrect combination of incompatible aliases. It is the same as use of incompatible options because aliases represents these options.

22.4 Configuration of Externs (codal.conf)

A Level project can contain another external projects (ASIP, Level) which can be configurable. The project which uses external configurable projects can be static and needn't be converted to a configurable one. But it is possible that configurable Level project can use configurable external projects. These external projects can be even affected by configuration of the top project itself.

The configuration is done in the standard `codal.conf` in special `extern` section. This section has the same content as the `ip.conf` file but it adds a new attribute `id` with the name of the imported configurable project. The following example is similar to the one from the section "Configuration of Current Model (ip.conf)" on page 419:

Example 226:extern configuration (codal.conf)

```
[[extern]]
       configuration = “configuration-string”
       id = “extern_project_name”

[options]
       INT = 0x1000
       BOOL = true
       STR = "string"
       LIST = [ "item1", "item2" ]
       # in the 'codal.conf' it is possible to use option value as
       # a right side of the assignment
       FOO = OPTION_EXTERN_FOO
```

22.5 Generated Header Files

For current configuration of the top project and for all configurations of its external projects a corresponding header file is generated. These files are stored in the root of the work directory of the project. The name of these files has a form of project name plus `_options.hcodal` suffix:

```
project_name_options.hcodal
```

The generated file is a standard C header file which contains preprocessor macros. Each line represents one option which has been defined in the `options.conf` file. Every macro name starts with the prefix `OPTION_` and is followed by the name of the corresponding option. This prefix cannot be altered and is used to distinguish the option macros from the standard ones defined in the model.

The form of generated option macro is affected by the type of option and also by its value. Following example shows all variants:

Example 227: Option macros

```
// 'BOOL' option with 'bool' type and 'true' value
#define OPTION_BOOL
// 'BOOL' option with 'bool' type and 'false' value
#undef OPTION_BOOL
// 'INT' option with 'uint' type, format "0x%x" and '0x1000' value
#define OPTION_INT 0x1000
// 'STR' option with 'string' type and 'string' value
#define OPTION_STR "string"
// 'LIST' option with 'list' type and '[ "item1", "item2" ]' value
#define OPTION_LIST "item1", "item2"
```

22.6 List of Configurable Files

The generated header file is automatically included in the most source files of the CodAL project. Because this header file contains standard macro definitions, they can be used in a preprocessor conditionals and a macro replacement to affect the model source code. The list of configurable source files supported by the Codasip Studio is following:

- `codal.conf` - Every attribute in the `codal.conf` can be set by option for current configuration. In this case the value on the right side of the expression has to start with the prefix `OPTION_`. In this file the preprocessor directives cannot be used.

- `*.codal`, `*.hcodal` - The CodAL source codes for model description.
- `*.cpp`, `*.td`, `*.TD` - Custom source files for the LLVM backend generator.
- `*.c`, `*.h` - Source code for the newlib and other libraries.
- `*.s`, `*.S` - Assembler source codes for the startup library or the newlib.
- `*.lds`, `*.LDS` - Custom linker script.
- `*.cons` - User constraints for the Random Assembler Generator

22.7 Scripting

The core of model configuration is written in Python and can be accessed through the interface of the Codasip build system. The classes and their attributes regarding configurable model are described in this section.

22.7.1 Classes

Project

The `Project` class is intended for loading the CodAL or the Component Project. Standard behaviour for the configurable CodAL projects is to use the file `ip.conf` for the current configuration. This behaviour can be changed with several parameters in the load method.

```
load(dir, work_dir, project_paths, configuration_file, configuration, options)
```

- `configuration_file` - a path to a configuration file with the format same as in the `ip.conf` ,
- `configuration` - a configuration string,
- `options` - a dictionary where the key represents a name of the option and the value is setting of the the option.

CodalProject

For a CodAL project, the method `Project.load()` returns instance of the `CodalProject` class. Several attributes are set in this class regarding the configurable model:

- `options_metadata` - instance of the `OptionsMetadata` class with an internal representation of the `options.conf` for the current project,

- `options` - instance of the `Options` class with current settings of the configurable project.

OptionsMetadata

The `OptionsMetadata` class has only several attributes useful in scripts:

- `pattern` - the list of options representing the configuration string,
- `options` - the dictionary where the key is a name of the option and the value is an instance of the `OptionMetadata` class,
- `get_pattern_values()` - this method returns an OrderedDict where the key is a name of the option and the value is the list of all values which the option can be set with.

OptionMetadata

The `OptionMetadata` class represents one record stored in the `options` attribute in the instance of the `OptionsMetadata` class:

- `name` - a string representing the name of the option,
- `type` - a type of the option which is one of enumerators stored in the `OptionMetadata.Type` class.

Options

The `Options` class represents the current configuration of the CodAL project. The instance of this class stores the options with the actual value. This class provides several methods how to access these options or how to store current state to the configuration file. The following methods are useful in the scripts:

- `__iter__()/iteritems` - the `Options` class can be iterated as a dictionary where the key represents a name of the option and the value is actual value of this option,
- `__getitem__(name)` - the options can be accessed by its name with `[]`,
- `get_configuration()` - returns current configuration string or `None` when the option metadata does not have any pattern,
- `save_ip_conf(stream)` - saves current configuration in a form of `ip.conf` to the stream (the stream is any instance with write method).

22.7.2 Modules

codasip.options

All classes dealing with the options metadata and the option configuration are implemented in the module `codasip.options`. Useful helper functions which can be used in the user scripts ale also stored there.

```
product(metadata, options)
```

- `metadata` - instance of the class `OptionsMetadata`,
- `options` - OrderedDict or standard dict where the key is a name of the option and the value is the list of all values which the option can be set with.

Cartesian product of all possible configurations can be expressed by generator function `product(metadata, options)`. This function takes as a parameter the option `metadata` which can be obtained from the loaded CodAL project and options with the option name and its values which should be used for the product. The generator returns instances of the `Option` class with current configuration as a result.

The options dictionary does not need to contain all options from the option `metadata`. The default behavior is used for options that are not in this dictionary. For options with the `values` attribute, all possible values are used from this list. For options without this attribute, the default value is used which produces only one configuration and does not affect the number of combinations.

The default behavior can be overridden by changing the number of values in the parameter `options` for this option. If there are some options with the `int` type (e.g. an option with a memory size) and various sizes of this attribute should be used then it is possible to add this option to the dictionary with this predefined sizes.

The followng example shows how to print all possible configuration strings:

Example 228: Printing all possible configuration strings

```
from codasip import Project, options

# load the project from the current working directory
project = Project.load()
# get options metadata from this project
metadata = project.options_metadata

# get options constraints based on the options from a pattern
constraints = metadata.get_pattern_values()

# iterate through all possible configurations
or current_options in options.product(metadata, constraints):
       # print the configuration string for current configuration
       print current_options.get_configuration()
```

23 ANNEX: TABLES, EXAMPLES AND FIGURES

23.1 List of Tables

23.2 List of Examples

23.3 List of Figures

24 GLOSSARY

A

ABI

Application Binary Interface: describes the conventions for which registers are used for return values, the stack pointer etc.

ALU

Arithmetic and Logic Unit

API

Application Programming Interface: a set of routines, protocols, and tools that gives one piece of software access to the functionality of another. For example, a SystemVerilog simulator could access information from a simulation inside Codasip Studio through Studio's API.

ASIP

Application Specific Instruction-set Processor

ASIP Architecture

The programmer-visible aspects of the ASIP.

ASIP Microarchitecture

A specific implementation of an ASIP Architecture. An ASIP may have several Microarchitectures, each of which can have a different size and performance. Each Microarchitecture will have its own CA model, but all of them will share the same IA model.

Assembler

A tool that takes in assembler code (usually .s or .asm format) and outputs object code (.obj).

C

CA

Cycle Accurate

CLI

Command Line Interface.

Co-simulation
The simulation of a system using multiple, independent simulators. The Codasip simulator is encapsulated in a wrapper which allows it to co-simulate with tools that run, for example, SystemC or RTL simulations.

Codasip Commandline
An alternative to the IDE that allows users to run Codasip's tools from a command line.

Codasip Studio
The IDE, the CommandLine and the Tools.

Compiler
A tool that takes in high level code such as C or C++ and outputs assembly code.

H

HDL
Hardware Description Language. Verilog and VHDL are examples of HDLs. They are used to describe electronic hardware (see also “RTL”).

I

IA
Instruction Accurate

IDE
Integrated Development Environment. Codasip's IDE is based on Eclipse and is one component of Codasip Studio.

ISA
Instruction Set Architecture

L

LAU
Least Addressable Unit. The smallest piece of data that the processor system can handle. This is typically a byte, since the smallest addressable unit of memory tends to be a byte.

Linker

A tool that takes in object code and libraries and links them together to create executable code (typically a .exe).

LLVM

The name "LLVM" is not an acronym; it is the full name of a project, described as a collection of modular and reusable compiler and toolchain technologies (http://www.llvm.org).

M

MP

Multi-Processor (in this context, multiple ASIPs in a single Platform)

N

Non-terminal

See the definition of "terminal".

P

Panel

An area within the IDE main Window containing one or more Views (the Views being accessible via tabs).

Perspective

A style for the IDE window designed for a particular purpose, such as Debug or Profiling. The Views available in each Perspective are different. The default Perspective is called "Codasip".

Project

An object in the Codasip Studio that groups together items related to a particular part of a design. Different types of Projects are defined, and each type has a certain set of possible actions associated with it. There are three main types of Projects in Codasip: ASIPs, Platforms and Test.

R

RTL

Register Transfer Level. Used in Hardware Description Languages (HDLs) to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived.

S

SDK

Software Design Kit. The tools needed to generate a test program and simulate an ASIP model. They include, for example, a compiler, an assembler and a simulator.

T

Terminal

Syntax definitions consist of statements that progressively define a language from a root down to the leaves, or "terminals" of the language. They are to be distinguished from rule names (also called non-terminal symbols), which label intermediate branches of the syntax. Terminal symbols are written exactly as they are to be represented, whereas rule names are given an arbitrary name that represents the branch.

TLM

Transaction Level Model. A simulation model where the lowest level of abstraction is a transaction. Hence, signal-level details are hidden in a TLM.

U

UART

Universal Asynchronous Receiver Transmitter

V

View

The lowest level of hierarchy within the IDE Window (the Window contains Panels which contain Views, each of which are accessible through its tab).

W

Window

The IDE top level, synonymous with the Workbench.

Workbench

The interface presented by an IDE to the user.