codasip

CODAL LANGUAGE REFERENCE MANUAL

Studio Version 9.4.0
Release Date November 23, 2022

COPYRIGHT AND PROPRIETARY NOTICE

INTEGRATED 3RD PARTY SOFTWARE MODULES AND THEIR LICENSES

In addition to the Codasip License Agreement, integrated 3rd party software is also licensed for use under the terms and conditions of the license agreements set forth in the document titled *Integrated 3rd Party Software Modules and their Licenses* which is located on the Codasip Support.

TABLE OF CONTENTS

1 PREFACE

1.1 About

This manual describes the CodAL™ language, which was developed for prototyping Application Specific Instruction-set Processors (ASIPs) and multiprocessor designs.

1.1.1 Intended Audience

This reference manual is written for engineers who wish to understand CodAL's syntax and semantics.

1.1.2 Product Revisions

Codasip Studio v. 9.4.0

1.1.3 Typographical Conventions

Table 1: Typographical conventions

Convention	Usage	Example
Italics	Important text, term, document reference, or additional information.	See *CodAL Language Reference Manual*.
Bold	Inline references to keywords of the IDE (usually starts in upper case).	The **Project Explorer** window
`Monospace; <abstract name>`	Code, code values, Unix file names, prompts or instructions; field for substitution with user data.	Double-click the installation file `codasip-studio-<version>`
`keyword`	Inline references to CodAL™ keywords (lower case).	`element`, `event`
`Example`	Examples - typically snippets of code.	`register bit[DATA_W] test;`
`Syntax`	Explanation of syntax.	`StartSection: "start" "{"`
NOTICE:	Important notice for the user.	**NOTICE:** Changing this value can cause an error.

1.2 References

This reference manual can be read standalone, but in order to use CodAL it is necessary to also understand Codasip Studio, an Eclipse-based environment and set of tools for developing processor models. The following documents describe this system:

- *Codasip Tutorial 00 - Studio Quick Start*
- Other Codasip tutorials
- *Codasip Studio Technical Reference Manual*
- *Codasip Studio User Guide*

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

Guides

Document	Description
Codasip License Setup and Installation Guide	Detailed description of licenses setup.
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.
Codasip Studio SDK User Guide	A complete reference and guide to usage of SDKs generated by Codasip Studio.

Reference Manuals

Document	Description
Codasip CodAL Language Reference Manual	A complete presentation of the CodAL language and how to use it for writing processor models.
Codasip Instruction Semantics Language Manual	Description of Instruction Semantics that can be used with Codasip C/C++ Compiler and other tools.
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.
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 Technical Reference Manual	Reference information on Codasip Studio and the tools that it contains.

Tutorials

Document	Description
Codasip Tutorial 00 - Studio Quick Start	A step-by-step introduction to the essentials of Codasip Studio.
Codasip Tutorial 01 - CodAL Instruction Accurate Model	A step-by-step introduction to writing Instruction Accurate processor models in CodAL.
Codasip Tutorial 02 - CodAL Compiler Generation	A step-by-step introduction to generating a C/C++ compiler from an Instruction Accurate processor model written in CodAL.
Codasip Tutorial 03 - CodAL Cycle Accurate Model	A step-by-step introduction to writing a Cycle Accurate CodAL model.
Codasip Tutorial 04 - JTAG Extension	A step-by-step introduction to Codasip's JTAG extension.
Codasip Tutorial 05 - CodAL Interrupts and Peripherals	A step-by-step introduction to adding external devices to an processor CodAL model.
Codasip Tutorial 06 - SIMD Extension	Tutorial showing the implementation of SIMD extensions in the Codasip uRISC.
Codasip Tutorial 07 - Custom Components Verification	Tutorial describing process of adding manually modified UVM test-bench for a component into the processor UVM test-bench.
Codasip Tutorial 08 - uRISC VLIW Extension	Tutorial showing modifications to Codasip uRISC to create a simple VLIW architecture.
Codasip Tutorial 09 - uRISC Pipeline Extension	This tutorial provides the basic procedures necessary for modifying the pipeline of Codasip uRISC with additional stage and creating a pipelined multiplier.

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. Please provide:

- The product name
- The product 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. Please provide:

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

Codasip also welcomes general suggestions for additions and improvements.

2 READ ME FIRST

2.1 Prerequisites

This manual can be used to gain an understanding of CodAL by anyone who has a basic understanding of processors and their design.

2.2 Purpose and Goals

This manual describes the CodAL™ language, which was developed for prototyping Application Specific Instruction-set Processors (ASIPs) and multiprocessor designs.

3 CODAL BASICS

This chapter is organized as follows:

3.1 About CodAL

The CodAL language is a mixed-architecture description language, designed for the concurrent design of hardware and software for single or multiprocessor (MP) systems. Related tools for programming and simulation are automatically generated from the processor's CodAL description. The same description also enables the automatic generation of a micro-architecture implementation in hardware, using the VHDL or Verilog languages.

As you might expect from this description, CodAL's programming paradigm is significantly different from that of standard programming languages (such as C and C++) and HDLs (such as Verilog and VHDL). Having said this, its syntax style will be familiar to C and C++ programmers, and the Eclipse-based Codasip Studio™ environment makes it easy to experiment with the language. A novice CodAL programmer is in for a few surprises, but we hope that they will be pleasant ones!

A CodAL description has to be understood together with the decoder model upon which Codasip's tools are built. This makes it different to standard programming languages (e.g. C) and HDLs, where the code contains a more complete picture of the assembler or gates that are desired. One can, either mentally or with a debugger, step through the instructions of a C program in order to understand how it works. With Verilog, this is more tricky, but not unreasonably so. With CodAL, it is difficult to predict what will happen next without an understanding of the decoder model that is implicit in the language, and also an idea of the tools that are needed to support that model (the assembler, disassembler, simulator and compiler).

For these reasons, the objects in a CodAL description have to contain information suitable for a range of tool generation tasks. For example, an object representing a processor instruction has to have associated with it information on:

- What text and binary to use when representing the instruction to the assembler
- How to recognize the instruction and what actions to take when it is encountered during simulation
- What semantics to associate with it for the purposes of compilation
- What hardware to associate it with (resources, such as registers together with signals and their timing)

To express this information in a compact form, a CodAL description is built up in an object-oriented manner. That is simple objects are instantiated into more complex objects which are then instantiated into still more complex objects, and so on. This is true in standard, object-oriented languages also. However, unlike in C++, for example, the propagation of object-related information through the hierarchy is far more structured (preordained might be a better word).

CodAL's highly optimized structure makes it possible to pass the information without recourse to complex function calls (of the type `x = obj.subobj.subsubobj.thing`). The price to pay for this compactness and elegance is to learn the models associated with the CodAL system, so that one can easily understand which information is implicitly passed when an object is used. Fortunately, this is an easy undertaking for anyone familiar with processors.

To get a first idea of the nature of CodAL, consider one of its key object types: the **`element`**. Among the types of information associated with such an object are:

- **`assembly`** – the text rules associated with the **`element`** (for use primarily by an assembler tool, but by several others also)
- **`binary`** – encoding information (for use primarily by an assembler tool, but by several others also)

- **semantics** – behaviour associated with the **object** (for use by simulator and compiler tools, for example)
- **return** – reference information on the **object**, typically used for linking it with other objects

Suppose that we have an **element** called "r3ref", representing a register (or maybe several registers). Many examples of how to code such an element will be given in this document. For the moment, however, let us focus on the various ways in which it may be used. It may appear, for example, in a C-type expression, such as:

```
if(r3ref == 3) { ...
```

or :

```
x = y + r3ref;
```

Tools interested in this type of information will find it in the **element**'s **return** section. Other types of semantic information - describing the behaviour of the **element** to a simulator, for example - can be found in the **element**'s **semantics** section. Each tool knows where to find the **element**-associated information that it needs.

Another way to use `r3ref` is to reference it from within another **element**'s **assembly** section. For example:

```
assembly{ "LOAD" r3ref immediate };
```

In this example, information from `r3ref`'s own **assembly** section will be used to elaborate the text string possibilities. Such information is needed by assemblers, disassemblers and compiler generators, for example. When this example **assembly** section is elaborated, it may be found that `r3ref` represents a simple string, such as `"R3"`, or that it has a range of possibilities - `"R0"` to `"R31"` plus some specific names such as `"Rsp"`, `"Rsr"`, and so on. It depends on the complexity of the `r3ref` **element** and the **elements** from which it is built.

One more way to use `r3ref` is to reference it from within another **element**'s **binary** section. For example:

```
binary{ 0x45:bit[12] r3ref 0x11:bit[4] };
```

In this case, the tool that is interested in the binary instruction information will automatically elaborate information from `r3ref`'s own **binary** section in order to complete this binary description.

Note that in all the above examples - which are by no means exhaustive - the actions associated with the reference to the `r3ref` **element** was implied by the context of the reference. There were no `r3ref.index()`, `r3ref.assembly()` or `r3ref.binary()` calls to clarify what information was sought. This conciseness is possible because CodAL is optimized for a specific domain and has a certain implementation model associated with it - a processor regularly fetching, decoding and executing instructions from memory.

3.2 Project Description

Although each processor may differ in many aspects, every description has four parts:

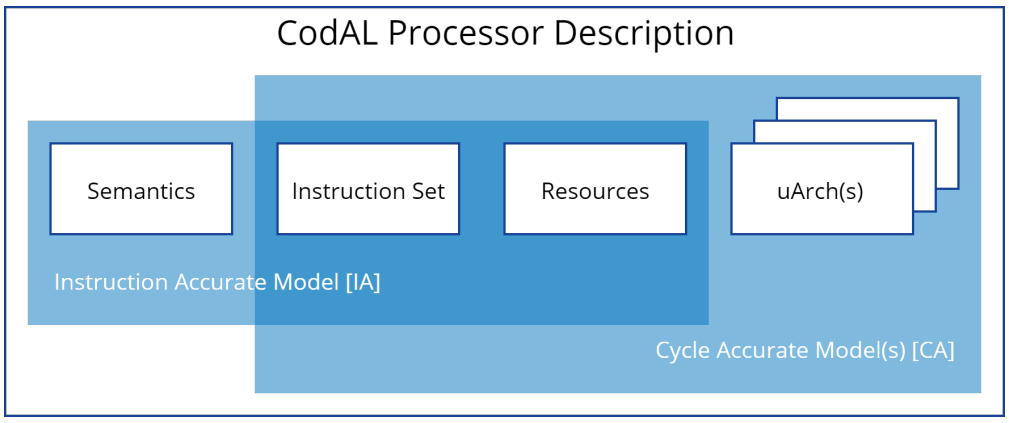
- Architectural Resources: For example, a program counter and registers.
- Instruction Set: The names of instructions, their operands and their binary form (opcodes).
- Semantics: The behaviour of each instruction and exception - how they affect the architecturally-visible resources.
- Implementation: Resources and behaviour (esp. timing) that are not visible in the Processor Architecture but which define a particular micro-architectural implementation.

Each micro-architecture inherits the Architectural Resources and the Instruction Set of its Processor Architecture and defines a particular Implementation. This gives us the following different types of processor model:

- Architectural Model: also known as the Instruction Accurate Model (IAM)
- Micro-architecture Model: also known as the Cycle Accurate Model (CAM).

The Architectural Model (or Instruction Accurate Model) has Architectural Resource, Instruction Set and Semantics parts. A Micro-architecture Model (or Cycle Accurate Model), on the other hand, has Architectural Resource, Instruction Set and Implementation parts. This is illustrated in the diagram below:

Figure 1: The parts of a CodAL processor description



Functional verification is used to ensure that the behaviour of the CAM Implementation (i.e. the micro-architecture) matches that of the IAM Semantics.

For each processor, there should be one IAM and as many CAMs as there are alternative micro-architectural implementations.

Processor description also has the **`testbench`** construct at disposal which can be used to describe a near environment around processor(s) – e.g. memories, peripherals, and other parts, such as third party components, or even other (lower) design levels.

3.3 Common Language Constructs

Several language constructs that occur in processor description are presented in the following subsections.

3.3.1 Top Level Constructs

The top-level syntax for a processor description is as follows:

```
TranslationUnit
   : TranslationUnit TranslationUnitBody
   | TranslationUnitBody
```

`TranslationUnit` represents a `.codal` file. It may contain any number of `TopNode` constructs.

```
TranslationUnitBody
   : Resource ';'
   | Element ';'
   | Event ';'
   | Emulation ';'
   | Peephole ';'
   | Set ';'
   | Start ';'
   | ScheduleClass ';'
   | AnsiDeclaration
   | AnsiFunctionDefinition
```

```
    | Module ';'
    | Extern ';'
    | Connect ';'
    | SettingsCompound ';'
    | Testbench ';'
    | ';'
```

For an example of a processor description, please download and inspect the codasip_urisc reference design, available on the Codasip internet site.

3.3.2 Number

A `Number` is a decimal, hexadecimal, octal or binary constant. For example:

Example 1: Number examples

```
10
0x10
0b1010
010

0xffffLU
1.12
```

The syntax of `Number` is as follows:

```
D           [0-9]           // decimal digits
H           [a-fA-F0-9]     // hexa digits
B           [01]            // binary digits
K           [0-7]           // Octal digit
IS          (u|U|l|L)*      // Type specifier for integer constants

E           [Ee][+-]?{D}+   // Exponent for reals
FS          (f|F|l|L)       // Type specifier for real constants

0[bB]{B}+       // Binary constant
0[xX]{H}+{IS}?  // Hexa constant
0{K}+{IS}?      // Octal constant
{D}+{IS}?       // Decimal constant

{D}+{E}{FS}?             // Real constant
{D}*"."{D}+({E})?{FS}?   // Real constant
{D}+"."{D}*({E})?{FS}?   // Real constant
```

3.3.3 String

The `String` non-terminal is text composed of any printable ASCII characters enclosed in quotation marks ('"'). Escape sequences \", \\ and \n may also be used. For example:

Example 2: String example

```
codasip_print(1, "Hello World\n");
```

3.3.4 Id

An identifier, `Id`, names a construct, such as a resource or an ISA construct. It can be formed of literals, decimal digits, underscores and dollar characters ($). However, a digit may not be used as the first character. For example:

Example 3: Id examples

```
register bit[32] r_pc;
int _tmp;
register_file bit[16] rf_gp32;
```

The syntax of the `Id` construct is as follows:

```
D           [0-9]           // decimal digits
L           [a-zA-Z_]       // Letters

({L}|$)({L}|{D})*           // Identifier
```

3.3.5 Compound Id

The `CompoundId` (compound identifier) construct defines a structure access to a member. It may be access to a port of a processor or a component. The following example shows a port access where the port is owned by a processor:

Example 4: CompoundId example

```
connect codix_lrm.p_output => open;
```

The syntax of the `CompoundId` construct is as follows:

```
CompoundId
   : CompoundId '.' Id
   | Id
   | '.' Id
```

3.3.6 Id List

The `IdList` (identifier list) construct is a list of identifiers, separated by commas, naming the instances of the construct. The following example shows a list of identifiers with a `Register` construct:

Example 5: IdList example

```
register bit[DATA_W] r_data_fe, r_data_id;
```

The syntax of the `IdList` construct is as follows:

```
IdList
   : IdList ',' Id
   | Id
```

3.3.7 Compile Expression

The `CompileExpression` construct is evaluated during compilation and so its operands must be constants (either numbers or enum identifiers). Every standard C operator can be used. CodAL introduces the following additional operators.

- Power. `a ** b` means that `a` is powered by `b`. `b` must be a compile time constant (e.g. `2 ** 3` equals 8).
- Bit-select. `[a..b]` extracts bit filed starting at the index `a` and ending at the index `b`. `a` and `b` must be compile time expressions (e.g. `0xA[2..1]` equals 1).

- Concatenation. `a :: b` concatenates two numbers into one (e.g. `2 :: 2:bit[2]` equals 10).
- Left and right rotation. `a <<< b` and `a >>> b` rotates `a` by `b`.
- `clog2`. It computes number of bits that are needed for a compile time expression (e.g. `clog2 (4)` equals 3).
- `bitsizeof`. It computes number of bits that are used by any resource or compile time expression (e.g. `bitsizeof (rf_gp)` equals 32).

Example 6: Compile expression examples

```
10 + 20       // addition
OPC_ADD[2..0] // bit select
10 * (DEFINE_CONSTANT + 20) // expression
```

The syntax of the `CompileExpression` construct is as follows:

```
CompileExpression
   : AnsiCompileExpression
```

```
AnsiCompileExpression
   : AnsiPrimaryExpression
   | '+' AnsiCompileExpression
   | '-' AnsiCompileExpression
   | '~' AnsiCompileExpression
   | '!' AnsiCompileExpression
   | "bitsizeof" AnsiCompileExpression
   | "bitsizeof" '(' AnsiDeclarationSpecifiers ')'
   | "clog2" AnsiCompileExpression
   | AnsiCompileExpression '|' AnsiCompileExpression
   | AnsiCompileExpression '&' AnsiCompileExpression
   | AnsiCompileExpression '^' AnsiCompileExpression
   | AnsiCompileExpression '+' AnsiCompileExpression
   | AnsiCompileExpression "::" AnsiCompileExpression
   | AnsiCompileExpression '-' AnsiCompileExpression
   | AnsiCompileExpression '*' AnsiCompileExpression
   | AnsiCompileExpression "**" AnsiCompileExpression
   | AnsiCompileExpression "::*" AnsiCompileExpression
   | AnsiCompileExpression '/' AnsiCompileExpression
   | AnsiCompileExpression '%' AnsiCompileExpression
   | AnsiCompileExpression ">>" AnsiCompileExpression
   | AnsiCompileExpression "<<" AnsiCompileExpression
   | AnsiCompileExpression ROR_OP AnsiCompileExpression
   | AnsiCompileExpression ROL_OP AnsiCompileExpression
   | AnsiCompileExpression "&&" AnsiCompileExpression
   | AnsiCompileExpression "||" AnsiCompileExpression
   | AnsiCompileExpression "==" AnsiCompileExpression
   | AnsiCompileExpression "!=" AnsiCompileExpression
   | AnsiCompileExpression '>' AnsiCompileExpression
   | AnsiCompileExpression '<' AnsiCompileExpression
   | AnsiCompileExpression "<=" AnsiCompileExpression
   | AnsiCompileExpression ">=" AnsiCompileExpression
   | AnsiCompileExpression '?' AnsiCompileExpression ':' AnsiCompileExpression
   | AnsiCompileExpression '[' AnsiCompileExpression ".." AnsiCompileExpression ']'
```

3.3.8 Parameters

CodAL parameters provide a mechanism to change the value of certain constant expressions. The user has to first define parameters, their data types and default values. Then they can change their value when starting the simulator or in the RTL code without having to modify their model.

The places where parameters can be used are specified with the `ParameterExpression` construct, which is identical to the `CompileExpression` construct, except that:

- the grammar additionally admits identifiers that name a parameter, and
- binary logical operators (&& and ||) and relational operators (==, !=, <, <=, >, and >=) are forbidden.

The `ParameterExpressionList` construct is used for comma-separated parameter expressions.

Example 7: Parameter definition example

```
parameter bit[32] CACHE_BASE = OPTION_CACHE_BASE;
```

The syntax of a parameter definition is as follows:

```
Parameter
       : "parameter" "bit" '[' BitWidth ']' Name '=' DefaultValue

BitWidth : CompileExpression
DefaultValue : CompileExpression
Name : Id
```

`Name` is the name of the parameter.

`BitWidth` defines the data type of the parameters.

`DefaultValue` specifies the value the parameter will have if no explicit value is provided by the user.

4 TESTBENCH DESCRIPTION

This chapter is organized as follows:

The **testbench** construct may be used to describe a near environment around processor(s) - e.g. memories, peripherals, and other parts, such as third party components, or even other (lower) design levels.

The motivation is that the processor may need to communicate with other components, such as a programmable interrupt controller or UART device. The designer can add such devices to a processor description using the **testbench** construct.

The testbench describing the 3rd party component takes simulation behaviour and a hardware description from the **Component Project** that describes a 3rd party component. A generated simulator will then simulate both processors and components and a generated RTL description will contain descriptions of both processors and components.

Example 8: Testbench example (model/share/other/testbench.codal)

```
testbench
{

    // instance of memory
    memory mem {
        // memory size in LAU
        size = 0x10000;

        // memory has two interfaces
        interface if_fe {
            // address, word size, LAU
            bits = {32, 32, 8};
            endianness = BIG;
            type = AHB3_LITE:SLAVE;
            flag = R;
        };
        interface if_ldst {
            // address, word size, LAU
            bits = {32, 32, 8};
            endianness = BIG;
            type = AHB3_LITE:SLAVE;
            flag = RW;
        };
    };

    // connect memory ASIP's interfaces with memory's interfaces
    connect if_fe => mem.if_fe;
    connect if_ldst => mem.if_ldst;
    // open ports
    connect p_halt => open;
    // constant ports
    connect 0 => p_input;
    connect 0 => p_input_en;

    connect p_error => open;
    connect p_output => open;
    connect p_output_en => open;
};
```

The syntax of the `Testbench` construct is as follows:

```
Testbench
   : "testbench" '{' TestbenchBody '}'
```

```
TestbenchBody
   : TestbenchBody TestbenchBodyConstruct ';'
   | TestbenchBodyConstruct ';'
```

```
TestbenchBodyConstruct
   : Memory
   | Cache
   | Bus
   | Extern
   | Connect
```

The objects that make up a testbench and/or processor description are described in the following sections.

4.1 Extern

The Extern construct may be used for 3rd party component instantiation or other CodAL Project instantiation .

The motivation for the first case is that the processor may need to communicate with other components, such as a programmable interrupt controller or UART device. The designer can add such devices to a processor description using the `Extern` construct.

The extern describing the 3rd party component takes simulation behaviour and a hardware description from the Component Project that describes the 3rd party component. A generated simulator will then simulate both processors and components and a generated RTL description will contain descriptions of both processors and components.

The motivation for the second case is that a designer can easily instantiate processors or low design levels in upper or top design level. In this case, there is no need to provide simulation model or hardware implementation, because everything is handled by Codasip Studio automatically.

In the following example, the component called `pic` is defined. It has one interface called `if_clb`. The component has two ports: one is for input (`p_irq_in`) and the second for output (`p_irq_out`).

Example 9: PIC Component (model/share/other/testbench.codal)

```
    // PIC component
    extern pic_t as pic
    {
        // output ports
        port bit[1] p_irq_out { direction = OUT; };
        // input ports
        port bit[4] p_irq_in { direction = IN; };
        // interface to the outside word
        interface if_ahb {
            bits = {32, 32, 8};
            endianness = BIG;
            type = AHB3_LITE:SLAVE;
            flag = RW;
        };
    };
```

The syntax of the `Extern` construct is as follows:

```
Extern
   : "extern" ExternSpecifier Id '{' ExternBody '}'
   | "extern" ExternSpecifier Id "as" IdList '{' ExternBody '}'
```

`Id` is a type name of the extern. It is a project name (directory name).

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

```
ExternBody
   : ExternBody ExternAttribute ';'
   | ExternAttribute ';'
```

```
ExternAttribute
   : Interface
   | Port
```

`Port` is used for point-to-point connections. Ports are described in Ports on page 37.

`Interface` is used for memory or bus connections. Interfaces are described in Interfaces on page 37.

4.2 Memory

The Memory construct is used for simple memory devices, such as BRAMs on FPGAs. Memories work in two ways. When an instruction-accurate simulator is created, then the memory has no latency - all requests are handled immediately. In the case of a cycle accurate simulation, the latency is taken into account.

Memory is accessed only using its interfaces. The accesses may be restricted by an interface type and flag and by address alignment (i.e. the memory may accept only aligned addresses).

In the following example, a RAM memory element `mem` is defined. This memory consists of 8-kB (0x1000) cells, each of which is 8 bits wide. The operations read and write are allowed on the interface `if_ldst`but only the read operation is allowed on the interface `if_fe`. `mem` is a big-endian memory and its read and write latency is one clock cycle. The memory does not support unaligned access.

Example 10: Memory instance (model/share/other/testbench.codal)

```
    // instance of memory
    memory mem {
        // memory size in LAU
        size = 0x10000;

        // memory has two interfaces
        interface if_fe {
            // address, word size, LAU
            bits = {32, 32, 8};
            endianness = BIG;
            type = AHB3_LITE:SLAVE;
            flag = R;
        };
        interface if_ldst {
            // address, word size, LAU
            bits = {32, 32, 8};
            endianness = BIG;
            type = AHB3_LITE:SLAVE;
            flag = RW;
        };
    };
```

The syntax of the `Memory` construct is as follows:

```
Memory
   : "memory" IdList '{' MemoryBody '}'
```

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

```
MemoryBody
   : MemoryBody MemoryAttribute ';'
   | MemoryAttribute ';'
```

```
MemoryAttribute
   : Interface
   | Size
   | Latencies
```

`Interface` is used for memory or bus connections. Interfaces are described in Interfaces on page 37.

```
Bits
   : "bits" '=' '{' CompileExpression ',' CompileExpression ',' CompileExpression '}'
```

`Bits` specifies the number of bits for the *address*, the *word* and *LAU*. Word must be divisible by LAU without a remainder. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Endianness
   : "endianness" '=' Id
```

`Endianness` defines the endianness of the interface or memory module. The default value is **`LITTLE`** and the only alternative is **`BIG`**.

```
Size
   : "size" '=' ParameterExpression
```

`Size` defines size of memories or caches in LAU. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

```
Latencies
   : "latencies" '=' '{' LatenciesItem ',' LatenciesItem '}'
```

```
LatenciesItem
   : CompileExpression
   | '{' LatenciesItemBody '}'
```

```
LatenciesItemBody
   : LatenciesItemBody ',' CompileExpression
   | CompileExpression
```

`Latencies` specifies how many clock cycles it takes to read (first `LatencyItem`) and write (second `LatencyItem`). The read and write latency must be greater than 0. More than one value is possible for read and write. If the more than one value is used, then the latencies are changed every time a read or write request is asserted. A read request is handled as a synchronous read and a write request is handled as a synchronous write. The latency information is taken into account only when the access is cycle-accurate (request/finish functions available via interfaces) and the simulator is working at the cycle-accurate level. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

4.3 Cache

Caches are used as fast memories close to the CPU. Codasip provides two types of cache:

- Read only (RO) – *Instruction cache*
- Read/write (RW) – *Data cache*

Both cache types have three attributes in common, i.e., **`size`**, **`numways`** (number of ways), and **`linesize`**. These are configured during generation of each cache. Data cache has one additional attribute, **`write-through`**, configurable during runtime. The replacement policy is fixed to a random algorithm.

Timings of instruction and data caches' states and operations are described in sections Instruction Cache on page 1 and Data Cache on page 1 respectively. The caches can be used in three configurations – *I Cache, D Cache and ID Cache* – with different cores and both *Harvard* and *Von-Neumann architectures*. The three configurations are depicted in Figure 2. Control and status registers are present in all three configurations and are described in more detail in section Control Status Unit (CSU) on page 1.

4.3.1 Configuration

Each cache contains two main interfaces:

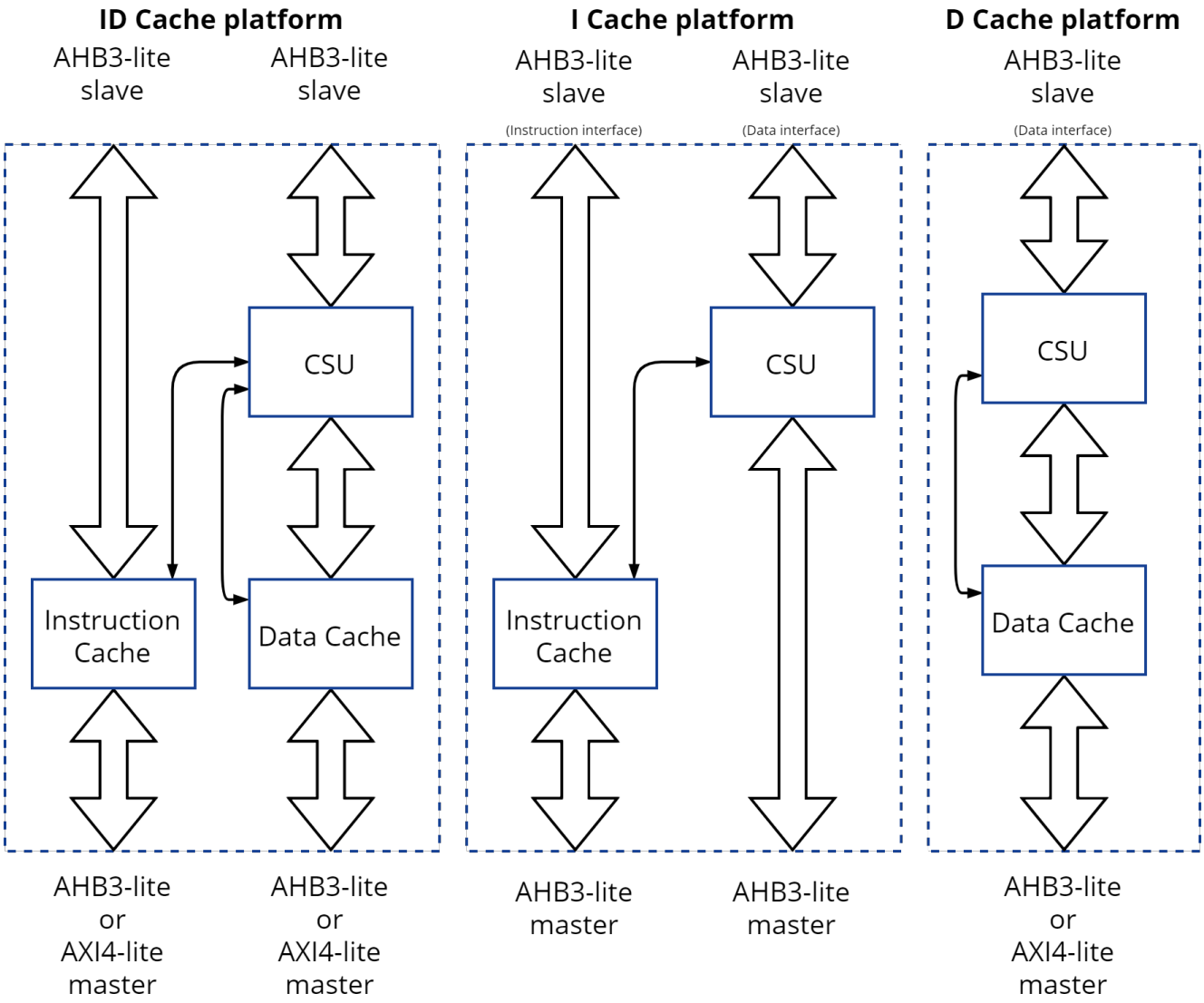
- Slave interface – accepts request from the core, only AHB3-Lite option.
- Master interface – interface to lower-level memory, AHB3-Lite, AXI4-Lite or AXI4.

Caches can be used in the following configurations:

- AHB3-Lite master interface
  - I Cache only – data interface is still present because of memory-mapped registers
  - D Cache only
  - ID Cache
- AXI4-Lite master interface
  - D Cache only
  - ID Cache
- AXI4 master interface
  - D Cache only
  - ID Cache

> **NOTICE:** Configuration with Instruction cache (I Cache) only is not possible with the AXI4-Lite or AXI4 master interface as the cache needs to have the AHB3-Lite slave interface on both the instruction and data interfaces. However, if no data cache (D Cache) is present, the slave and master data interfaces have to be of the same type, i.e., AHB3-Lite.

Figure 2: Cache configurations



More detailed description of cache configurations follows:

ID Cache configuration contains both Instruction and Data cache and has four interfaces. Two AHB3-Lite slave interfaces connect caches to ASIP or higher level memory – Instruction cache is connected directly, Data cache through control and status registers. Remaining two interfaces connect internal caches to the lower level memory. These interfaces can utilize either AHB3-Lite, AXI4-Lite or AXI4 master protocol.

I Cache configuration contains only Instruction cache and has four interfaces. The interfaces are similar to the ID Cache. The exception is that with the missing Data cache, control and status registers are connected directly to the lower level memory. Also, the interfaces to the lower level memory support only AHB3-Lite protocol.

D Cache configuration contains only Data cache. Unlike the previous two configurations, this configuration has only two interfaces. Similarly to the ID Cache, the interface to the lower level memory can utilize either AHB3-Lite, AXI4-Lite or AXI4 protocol.

The following example describes a 4 kB ID Cache `l1` connected directly to the ASIP and memory through four AHB3-Lite interfaces – `if_cpu_fetch`, `if_sdram_fetch`, `if_cpu_ldst`, and `if_sdram_ldst`.

Example 11: Cache (model/share/other/testbench.codal)

```
// L1 Cache
cache l1
{
    base = 0x0;
    type = ALL;

    program
    {
        // 4KB
        size = 4096;
        // associativity is 4
        numways = 4;
        // line size in LAUs/bytes
        linesize = 16;

        // interface from a CPU core
        interface if_cpu_fetch
        {
            bits = {32, 32, 8};
            type = AHB3_LITE:SLAVE;
            flag = R;
            endianness = LITTLE;
        };

        // interface to the upper memory subsystem
        interface if_sdram_fetch
        {
            bits = {32, 128, 8};
            type = AHB3_LITE:MASTER;
            flag = R;
            endianness = LITTLE;
        };
    };

    data
    {
        // 4KB
        size = 4096;
        // associativity is 4
        numways = 4;
        // line size in LAUs/bytes
        linesize = 16;

        // interface from a CPU core
        interface if_cpu_ldst
        {
            bits = {32, 32, 8};
            type = AHB3_LITE:SLAVE;
            flag = RW;
            endianness = LITTLE;
        };

        // interface to the upper memory subsystem
        interface if_sdram_ldst
        {
            bits = {32, 128, 8};
            type = AHB3_LITE:MASTER;
            flag = RW;
            endianness = LITTLE;
        };
    };
};
```

For more details about cache semantics, see chapter Testbench Description, section Cache in *Codasip CodAL Language Reference Manual*.

4.3.2 Configurable Parameters

Caches have several key parameters that allow to configure number of unique designs. Each cache has its own parameters. Therefore, the I Cache and D Cache configurations may differ. Generally, the parameter values must be set as powers of two,

with some exceptions.

4.3.2.1 Common

Table 2: Common configurable parameters

Parameter name	Description	Supported Values	Default Value
RESET_LEVEL	Active level of reset.	Low (0), High (1)	0
RESET_SYNC	Synchronous reset.	Async (0), Sync (1)	1
CSU_BASE_ADDR	Base address of memory-mapped CSU.	0 to 2^64 - CSU_SIZE[1]	0

4.3.2.2 Storage

Table 3: Storage configurable parameters

Parameter name	Description	Supported Values	Default Value
CACHE_NUM_WAYS	Associativity (number of ways).	1, 2, 4, 8, 16	4
CACHE_SIZE	Total cache size in bytes.	2048 (2 kB) to 262144 (256 kb)	16384 (16 kB)
CACHE_LINE_SIZE	Size of one cache line in bytes.	16 to 20481[2]	32
CACHE_IS_LITTLE	Endianness[3].	Big (0), Little (1)	0

4.3.2.3 Interfaces

In the case of the ID Cache configuration, there is a restriction that `DCACHE_DATA_BITS` must be equal or higher than both `ICACHE_ADDR_BITS` and `DCACHE_ADDR_BITS` to be able to reach the whole address space. Master and slave interfaces can have different data bit width, with the master interface equal or wider than the slave interface—except for data interface on the I Cache-only configuration, where they have to be equal. The slave data interface is not allowed to be wider than the master interface. The master address has to be equal or wider than the slave address to be able to reach the whole address space.

Table 4: Interface configurable parameters

Parameter name	Description	Supported Values	Default Value
CACHE_ADDR_BITS	Address bit width for the slave interface.	32 to 64	32
CACHE_DATA_BITS	Data bit width for the slave interface.	32 to 1024	32
CACHE_MASTER_ADDR_BITS	Address bit width for the master interface.	32 to 64	32
CACHE_MASTER_DATA_BITS	Data bit width for the master interface.	32 to 1024[4]	32
CACHE_SRAM_ADDR_BITS	Address bit width for the SRAM interfaces.	1 to 14[5]	7
CACHE_SRAM_TAG_BITS	Data bit width for the TAG SRAM interface.	15 to 60[6]	21

The control and status registers of the cache can be found in Codasip Studio Technical Reference Manual, *chapter "Cache"*.

The syntax of the `Cache` construct is as follows:

[1] `CSU_BASE_ADDR` must be aligned to `max(1024, CSU_SIZE*BYTES_PER_WORD)`.

[2] Size of the cache line must be at least 4 slave interface words and a multiple of the word size, up to 16 words.

[3] BIG endian is not supported with AXI4 and AXI4-Lite master interface.

[4] For the AXI4-Lite master interface, only 32 and 64 bits are possible. For the AHB3-Lite and AXI4 master interfaces, the ration of width to `CACHE_LINE_SIZE` must be equal to 1, 4, 8 or 16 because only `SINGLE` and `WRAP` burst types are supported.

[5] `clog2((CACHE_SIZE / CACHE_LINE_SIZE) / CACHE_NUM_WAYS)`

[6] `CACHE_ADDR_BITS - clog2(CACHE_LINE_COUNT) - clog2(CACHE_WORDS_PER_LINE) - clog2(CACHE_BYTES_PER_WORD) + 1`, where `+ 1` is for a valid bit

```
Cache
   : "cache" IdList '{' CacheBody '}'
```

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

```
CacheBody
   : CacheBody CacheAttribute ';'
   | CacheAttribute ';'
```

```
CacheAttribute
   : "program" '{' CacheDefinitionBody '}'
   | "data" '{' CacheDefinitionBody '}'
   | "base" '=' ParameterExpression
   | "type" '=' Id
```

`Base` defines an address on which control and status registers are accessible. For instance, you can read out the size of caches. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

`Type` defines whether a cache contains instruction and data cache, `ALL`, or it contains only an instruction cache, `PROGRAM`, or it contains only a data cache, `DATA`. If the cache has the instruction cache only, the **`data`** definition has to be there as well. One, a connection to the upper memory subsystem has to use a proper interface. Two, the control and status registers are accessed using interface in the **`data`** definition.

Each type of cache, either the instruction cache or data cache, has separate definition of sizes, interface etc.

```
CacheDefinitionBody
   : CacheDefinitionBody CacheDefinitionAttribute ';'
   | CacheDefinitionAttribute ';'
```

```
CacheDefinitionAttribute
   : Interface
   | Size
   | Linesize
   | Numways
```

`Interface` is used for memory or bus connections. Interfaces are described in Interfaces on page 37.

Each cache has to have two interfaces, one that is controller by processor and the second one that goes to the upper memory.

```
Size
   : "size" '=' ParameterExpression
```

`Size` defines size of memories or caches in LAU. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

```
Linesize
   : "linesize" '=' ParameterExpression
```

`LineSize` specifies a size of a cache line in LAU. Only powers of 2 are allowed. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

```
Numways
   : "numways" '=' ParameterExpression
```

`NumWays` specifies the associativity of the cache. Only powers of 2 are allowed. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

4.4 Bus

Modeling buses is possible via the `Bus` construct. Each bus has an arbiter and a decoder. The decoder consists of one or more interfaces to the slave devices connected through their interfaces. The arbiter handles master interface, only single master system is supported. Each slave device has a dedicated address range. Currently, only the Codasip Processor Bus (CPB) bus is supported. Detailed information about the bus implementation, timing, and behaviour can be found in the *Codasip Studio Technical Reference Manual*. For a single master **CPB** bus, only a single master interface can be specified.

All slaves described in the decoder has to be connected to the bus through a specified slave interface. In other words, you should create an explicit connection to the bus. The same approach is taken in the case of masters and arbiter.

Interval range is translated to the slave relative address range starting at 0x0 to the length of the interval (see the example of a bus for more details).

In the following example, the `cpb` bus is defined. Its decoder has two slaves. One slave is connected to the interface `if_ms_2` and it is mapped to address range `0x10000` to `0x1ffff`. Another slave component is mapped to address range `0x80` to `0x8f` from the relative slave address range `0x0 .. 0xf`. The bus address `0x82` is therefore translated to the slave relative address `0x02`.

The width of the address bus is 32 bits and the whole range is connected to slaves. The bus master is connected to the interface `if_mm`. The word size is 32 bits and the LAU is 8 bits.

Example 12: Bus (model/share/other/testbench.codal)

```
// Bus
bus clb
{
    interface if_mm
    {
        endianness = BIG;
        bits = {32, 32, 8};
        type = AHB3_LITE:MIRRORED_MASTER;
        flag = RW;
    };

    interface if_ms_1
    {
        endianness = BIG;
        bits = {32, 32, 8};
        type = AHB3_LITE:MIRRORED_SLAVE;
        flag = RW;
    };

    interface if_ms_2
    {
        endianness = BIG;
        bits = {32, 32, 8};
        type = AHB3_LITE:MIRRORED_SLAVE;
        flag = RW;
    };

    // slaves
    decoder =
    {
           0x80 ..    0x8f : if_ms_1,
        0x10000 .. 0x1ffff : if_ms_2
    };
    // arbiters
    arbiter =
    {
        if_mm
    };
};
```

The syntax of the `Bus` construct is as follows:

```
Bus
   : "bus" IdList '{' BusBody '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
BusBody
   : BusBody BusAttribute ';'
   | BusAttribute ';'
```

```
BusAttribute
   : Interface
   | BusDecoder
   | BusArbiter
```

```
Bits
   : "bits" '=' '{' CompileExpression ',' CompileExpression ',' CompileExpression '}'
```

`Bits` specifies the number of bits for the *address*, the *word* and *LAU*. Word must be divisible by LAU without a remainder. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Endianness
   : "endianness" '=' Id
```

`Endianness` defines the endianness of the interface or memory module. The default value is **`LITTLE`** and the only alternative is **`BIG`**.

```
BusDecoder
   : "decoder" '=' '{' BusDecoderBody OptionalComma '}'
```

`BusDecoder` describes slaves connections to the bus.

```
BusDecoderBody
   : BusDecoderBody ',' BusDecoderSlave
   | BusDecoderSlave
```

```
BusDecoderSlave
   : CompileExpression ".." CompileExpression ':' Id
```

`CompileExpression`s denotes an address interval (in the form *from* - *to*). The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`CompoundId` refers to a slave and the slave interface used. The syntax of the `CompoundId` construct is described in chapter CodAL Basics, section Compound Id.

```
BusArbiter
   : "arbiter" '=' '{' BusArbiterBody OptionalComma '}'
```

`BusArbiter` describes master connections to the bus.

```
BusArbiterBody
   : BusArbiterBody ',' Id
   | Id
```

`CompoundId` refers to a bus master and the master interface used. The syntax of the `CompoundId` construct is described in chapter CodAL Basics, section Compound Id.

4.5 Connections

Connections are made using the `Connect` construct. The two ends of a connection must be compatible. In other words, the bits, endianness and type information must be the same. The bus interfaces are connected to the bus (a direct connection between slave and master is allowed).

Since the construct is symmetrical, it does not matter which connection is placed on the left and right of the `=>` symbol. Each side can be formed by a single identifier for a processor, a memory, a component, a bus, a port or an interface placed on the top level.

Output ports can be explicitly marked as unconnected by using the **`open`** keyword on one side of the connect description. Unconnected input ports may be driven by a constant value.

In the following example, a component called `pic` is connected to the `codix_lrm` processor using ports. The processor is also connected to the `mem` memory. Port `p_halt` is an output port, but we do not care about its value. The `p_data_in` port is always driven to zero.

Example 13: Connections (model/share/other/testbench.codal)

```
    // connect memory ASIP's interfaces with memory's interfaces
    connect if_fe => mem.if_fe;
    connect if_ldst => mem.if_ldst;
    // connect PIC
    connect pic.p_irq_out => p_irq;
    // open ports
    connect p_halt => open;
    // constant ports
    connect 0 => p_input;
    connect 0 => p_input_en;
```

The syntax of the `Connection` construct is as follows:

```
Connect
   : "connect" AnsiConditionalExpression "->" AnsiConditionalExpression
   | "connect" AnsiConditionalExpression "->" "open"
   | "connect" "open" "->" AnsiConditionalExpression
```

`CompoundId` represents a target or a source of a connection. The syntax of the `CompoundId` construct is described in chapter CodAL Basics, section Compound Id.

`CompileExpression` denotes a value that is driven to the input port. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

5 PROCESSOR DESCRIPTION

This chapter is organized as follows:

In the following subsections, the use of CodAL to describe processors is organized according the four processor parts presented in CodAL Basics on page 5 (Architectural Resources, Instruction Set, Semantics and Implementation) and the language syntax itself. Please note, however, that this division into four parts, while useful, is not rigorous in the mathematical sense. For example, **`event`**s are described under the Implementation section, below, but a few **`event`**s are also needed in an Instruction Accurate Model. Likewise, both architectural and non-architectural resources are described in the Resources section (this is more convenient than having a separate section for the non-architectural resources needed by Cycle Accurate Models).

The file types used by CodAL to describe processors have `.codal` and `.hcodal` (**h**eader **codal**) extensions.

5.1 Resources

In this section we describe how to define certain structural components of processors, e.g. registers, register files, pipelines, etc. A processor may contain the following resources:

- *Address space* defines a memory view of the processor through its interfaces.
- *Pipeline* defines the pipeline stages and their implementation.
- *Register* defines storage.
- *Register File* defines a group of registers.
- *Signal* defines a state but no memory.
- *Interface* represents an connection to the memory or bus.
- *Port* represents point-to-point connection.
- *Tcm* represents tightly coupled memory.

The following sections describe each of these resources in more detail.

5.1.1 Register

Registers are essential components of any processor. CodAL supports the declaration of common registers as well as registers with a special meaning.

The special meaning is assigned using *specifiers*. The following types are supported:

- **`pc`** – Specifier denoting a program counter. A program counter is automatically recognized as an architectural register. Exactly one register construct with this specifier has to be defined in each processor model.
- **`arch`** – Specifier denoting that the register is architectural (i.e. visible to the programmer). It is necessary to identify architectural registers for compiler generation.

- **alias** – Specifier denoting a register alias. It adds another logical view and access to the overlapped resource. The aliasing of a register or register file is used when we need to access certain parts of a register or register file via different names.
- **dont_touch** – Specifier denoting that the register should not be optimized out.

A read request is done within the current clock cycle (asynchronous read, latency is zero) while a written value is visible in the next clock cycle (synchronous write, latency is one). This behaviour is enabled for CA models only. For an IA model, the registers behave in the same way as C variables, i.e. the write is immediately visible.

A register may have a reset—defined by the **reset** attribute, whose default value is `true`. If the register does not have a reset, i.e. the **reset** attribute is set to `false`, then the default value is undefined. If the register has a reset, then the default value is automatically assigned during the processor reset phase. In other words, such a register does not need to be reset explicitly. The default value that is assigned during reset is zero. The value may be changed by user using the **default** attribute.

Registers can be assigned to a pipeline stage. Such a register is called as pipeline register and it has advanced functionality. It may be cleared or stalled. The clear means that the value in the following clock cycle will be zero (or the default value). The stall means that no matter what is written to the register, the register keeps its value. This functionality is needed for hazard handling in pipeline systems.

Arrays of both architectural and microarchitectural registers can be created. Register aliases to the arrays of registers are also supported. For more information about the arrays, see section 5.3 Arrays.

In the following example, the `r_pc` program counter is defined. The bit-width of this register is `ADDR_W`. When reset occurs, it is reset to `BOOT_START`. The next resource is the architectural register `r_psw`, which has an alias . The register alias `a_carry` is one bit wide and it accesses the bit at the index 2. The example is illustrated in Figure 3.

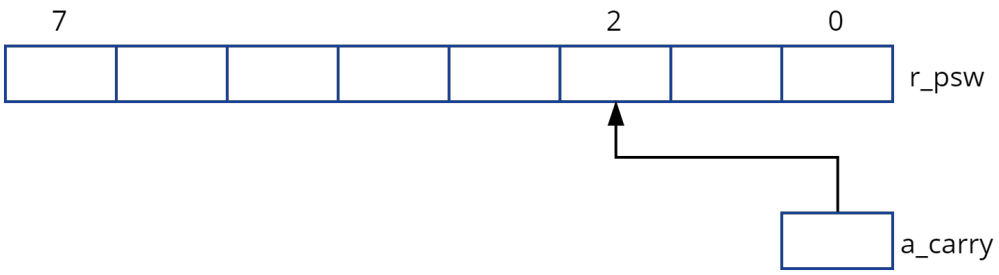
Example 14: Registers (model/share/resources/arch.codal)

```
// Program counter, address of bootloader is taken as default
pc register bit[ADDR_W] r_pc { default = BOOT_START; };

// Program status word
arch register bit[PSW_W] r_psw;

// Alias on the second bit of psw
alias register bit[1] a_carry
{
    // Specify the second bit
    overlap = r_psw[2..2];
};
```

Figure 3: Register alias



The next example shows pipeline registers.

Example 15: Pipeline registers (model/ca/resources/ca_resources.codal)

```
register bit[MEM_OP_W]   r_rd_mem_rw      { pipeline = pipe.RD; };
register bit[ALU_OP_W]   r_rd_alu_op      { pipeline = pipe.RD; };

register bit[MEM_OP_W]   r_ex_mem_rw      { pipeline = pipe.EX; };
register bit[ALU_OP_W]   r_ex_alu_op      { pipeline = pipe.EX; };
```

The next example shows arrays of registers.

Example 16: Arrays of registers

```
// Microarchitectural array of registers
register bit[1] r_flags [FLAGS_COUNT]
{
    pipeline = pipe.EX;
};

// Architectural array of registers
arch register bit[32] r_instr_addr [THREAD_COUNT];

// Alias to array of registers
alias register bit[24] a_instr_addr_block [THREAD_COUNT]
{
    overlap = r_instr_addr[31..8];
};
```

The syntax of the `Register` construct is as follows:

```
Register
   : Specifier "register" "bit" '[' CompileExpression ']' IdList RegisterBodyEncap
   | Specifier "register" "bit" '[' CompileExpression ']' IdList '[' CompileExpression
']' RegisterBodyEncap
```

The first mandatory `[CompileExpression]` specifies the bit-width of the register. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

The second optional `[CompileExpression]` specifies that this construct should represent an array of register resources. The value of the compile expression describes how many registers there should be in the array. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Specifier
   : "arch"
   | "pc"
   | "alias"
   | "dont_touch"
   | %empty
```

`Specifier` adds a special meaning to the register. The `Specifier` is optional.

```
RegisterBodyEncap
   : '{' RegisterBody '}'
   | %empty
```

```
RegisterBody
   : RegisterBody RegisterAttribute ';'
   | RegisterAttribute ';'
```

```
RegisterAttribute
    : "write_enable" '=' CompileExpression
    | Default
    | "pipeline" '=' CompoundId
    | "write_mask" '=' CompileExpression
    | Overlap
    | Dff
    | Reset
    | ClockEnable
    | Dwarf
    | Aacr
```

write_enable enables or disables a write enable port in a register.

Default denotes the value or values that are assigned to the register or array of registers during the reset or clear operation. If the `Register` construct describes a single register, only the first syntax option is available describing one default value of the register with the `ParameterExpression` construct. If the `Register` construct describes an array of register resources, there are two different ways to specify default values of each member of the array. The first is to use one default value for all members of the array. In such a case, the same syntax as for a single register can be used. The second is to specify a list of values with a length matching the size of the array by using the `ParameterExpressionList` construct. This assigns a specific default value to each member of the array. The syntax of the `ParameterExpression` and `ParameterExpressionList` constructs is described in chapter CodAL Basics, section Parameters.

```
Default
   : "default" = ParameterExpression
   | "default" = '{' ParameterExpressionList '}'
```

pipeline declares a pipeline register and assigns a pipeline stage.

write_mask can be optionally used to define a so called "sparse" register, i.e. a register in which certain bits cannot be changed. These fixed bits always retain their default value regardless of the data that is being written to the register. The register write semantics with the **write_mask** is as follows: `new_register_value := (writen_value & write_mask) | (default_value & ~write_mask)`. This feature allows to reduce the usage of hardware resources (number of flip-flops) in the resulting design. If the **write_mask** is omitted, all bits of the register are mutable (writable).

Overlap defines a target register that is overlapped by a register alias. Note that this construct does not support array indexing. However, arrays of alias registers with the **overlap** attribute can be created. In this case, the **overlap** attribute must reference another array of registers. It is not possible to create an array of register aliases that overlaps a single register or a register alias that overlaps an array of registers.

```
Overlap
   : "overlap" '=' CompoundId
   | "overlap" '=' CompoundId '[' CompileExpression ".." CompileExpression ']'
   | "overlap" '=' CompoundId '[' CompileExpression ']'
   | "overlap" '=' CompoundId '[' CompileExpression ']' '[' CompileExpression ".." Com-
pileExpression ']'
```

`CompileExpression` denotes an index within a target register file or it is used for a bit-select operation when only some bits of the target are aliased. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`ParameterExpression` indicates which attributes can be defined using CodAL parameters. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

`CompoundId` is a target identifier. The syntax of the `CompoundId` construct is described in chapter CodAL Basics, section Compound Id.

Reset enables or disables the reset features. It also defines the type of the reset.

```
Reset
   : "reset" '=' CompileExpression
   | "reset" '=' '{' CompileExpression ',' Id '}'
```

`CompileExpression` denotes whether the register or register file has a reset port.

`Id` defines the type of the reset, it may be `ASYNCHRONOUS` or `SYNCHRONOUS`.

`ClockEnable` defines a source of clock enable signal. If the signal is `High`, the clock is not gated. If the signal is `Low`, then the clock is gated and the value is not written.

```
ClockEnable
   : "clock_enable" '=' CompoundId
```

`CompoundId` is an identifier of a signal or a register that holds the clock enable signal.

`Dff` defines behavior of a register when an IA simulator is generated. By default, the register behaves as a variable in the IA simulator. In some cases, it is useful to have a register that behaves as a list of D flip-flops (e.g. VLIW architectures). To enable the D flip-flop behavior, **`dff`** has to be set to `true`.

```
Dff
   : "dff" '=' CompileExpression
```

`CompileExpression` denotes whether **`dff`** behavior is on or off.

`Dwarf` specifies the DWARF index used for a register or multiple DWARF indexes used for an array of registers. If the **`Dwarf`** construct is not used, the DWARF indexes are assigned automatically from the pool of unused indexes. The first mandatory `CompileExpression` value describes the start of the range of DWARF indexes. The second optional `CompileExpression` value describes the end of the range. If the end index is missing, the start index is used, creating a range of a single value. For a single register or architecture array of registers, the range must contain exactly one value. For microarchitecture array of registers, the range must contain number of values equal to the array size. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Dwarf
    : "dwarf" '=' CompileExpression
    | "dwarf" '=' CompileExpression ".." CompileExpression
```

`Aacr` specifies the RISC-V AACR index or indexes of the register or an array of registers. The principle of operation is almost identical to the **`Dwarf`** construct. The only difference is that if the **`Aacr`** construct is not used, the AACR indexes are not automatically generated and the register or array of registers have no AACR indexes assigned to them.

```
Aacr
    : "aacr" '=' CompileExpression
    | "aacr" '=' CompileExpression ".." CompileExpression
```

5.1.2 Register File

Register files are essential components of any processor and CodAL supports the declaration of common registers files well as register files with a special meaning.

The special meaning is assigned using *specifiers*. The following types are supported:

- **`arch`** – Specifier denoting that the register file is architectural (i.e. visible to the programmer). It is necessary to identify architectural register files for compiler generation.

- **alias** – Specifier denoting a register file alias. It adds another logical view and access to the overlapped resource. The aliasing of a register or register file is used when we need to access certain parts of a register or register file via different names.

As for simple registers, a register file read request is done within the current clock cycle (asynchronous read, latency is zero) while a written value is visible in the next clock cycle (synchronous write, latency is one). This behaviour is enabled for CA models only. For an IA model, the registers behave in the same way as C variables (i.e. the write is immediately visible).

The register file is accessed using a standard C style, using brackets. The number within the bracket denotes the index within the register file. The access style is the same in both IA and CA models.

For a CA model, register files must have a number of read and write data-ports defined. The data-ports are automatically bound to the statements within the **semantics** sections of **elements** or **events**.

Arrays of both architectural and microarchitectural register files can be created. Register file aliases to the arrays of register files are also supported. For more information about the arrays, see section 5.3 Arrays.

In the following example, architectural register file `rf_gp32` is defined. It has `RF_GP_SIZE` elements. Data-ports are also defined, so the register file can be used in CA simulation and for generating hardware. It has two aliases. The first one is `a_rf_gp16_full`. It has twice as many items as the target register file, so the target file is fully overlapped. The second alias is `a_rf_gp16_partial`. The overlap definition also contains a bit select information, so the items within the alias overlap bits 15..0. The upper bits are not used by this alias. The code is illustrated in Figure 4.

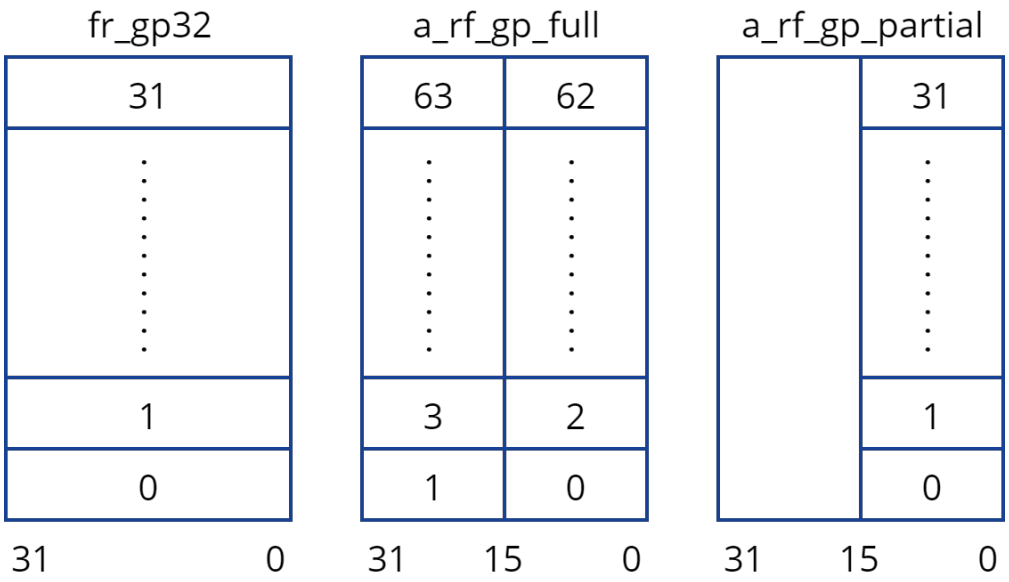
Example 17: Register files (model/share/resources/arch.codal)

```
// General Purpose Registers
arch register_file bit[WORD_W] rf_gp32
{
    size = RF_GP_SIZE;
    dataport r0, r1 { flag = R;};
    dataport w0 { flag = W;};
};

// 16bit alias on register file (full overlap)
alias register_file bit[WORD_W / 2] a_rf_gp16_full
{
    size = RF_GP_SIZE * 2;
    overlap = rf_gp32[0];
};

// 16bit alias on register file (partial overlap)
alias register_file bit[WORD_W / 2] a_rf_gp16_partial
{
    size = RF_GP_SIZE;
    overlap = rf_gp32[0][15..0];
};
```

Figure 4: Register file alias



The next example shows arrays of register files:

Example 18: Arrays of register files

```
// Microarchitectural array of register files
register_file bit[32] rf_fu_storage [FU_COUNT]
{
    size = 4;
    dataport r0, r1 { flag = R;};
    dataport w0 { flag = W;};
}

// Architectural array of register files
arch register_file bit[32] rf_gpr [THREAD_COUNT]
{
    size = 32;
    dataport r0, r1 { flag = R;};
    dataport w0 { flag = W;};
}

// Alias to array of registers
alias register_file bit[16] a_gpr_partial [THREAD_COUNT]
{
    overlap = rf_gpr[31..16];
};
```

The syntax of the `RegisterFile` construct is as follows:

```
RegisterFile
   : Specifier "register_file" "bit" '[' CompileExpression ']' IdList '{' Register-
FileBody '}'
   | Specifier "register_file" "bit" '[' CompileExpression ']' IdList '[' Com-
pileExpression ']' '{' RegisterFileBody '}'
```

The first mandatory `[CompileExpression]` specifies the bit- width of the register file. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

The second optional `[CompileExpression]` specifies that this construct should represent an array of register file resources. The value of the compile expression describes how many registers there should be in the array. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Specifier
   : "arch"
   | "pc"
   | "alias"
   | "dont_touch"
   | %empty
```

`Specifier` adds a special meaning to the register. The `Specifier` is optional. Note that, although the `Specifier` grammar is identical for registers and register files, register files cannot be used with the **`pc`** specifier.

```
RegisterFileBody
   : RegisterFileBody RegisterFileAttribute ';'
   | RegisterFileAttribute ';'
```

```
RegisterFileAttribute
    : "size" '=' CompileExpression
    | "dataport" IdList '{' DataportBody '}'
    | Default
    | Overlap
    | Dff
    | Reset
    | ClockEnable
    | Dwarf
    | Aacr
```

`ParameterExpression` indicates which attributes can be defined using CodAL parameters. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

`size` denotes a number of registers within a register file.

`dataport` defines either read or write dataports.

```
DataportBody
   : DataportBody DataportAttribute ';'
   | DataportAttribute ';'
```

```
DataportAttribute
   : InterfaceFlag
```

```
InterfaceFlag
   : "flag" '=' Id
```

`InterfaceFlag` specifies the read (R) or write (W) capabilities of the dataport

`Default` denotes the value or values that are assigned to all registers of a register file or a register file array during the reset operation. If the `RegisterFile` construct describes a single register file, only the first syntax option is available, assigning one default value to all register file members with the `ParameterExpression` construct. If the `RegisterFile` construct describes an array of register file resources, there are two different ways to specify default values of each member of the array. The first is to use one default value for all members of the array. In such a case, the same syntax as for a single register file can be used. The second is to specify a list of values with a length matching the size of the array by using the `ParameterExpressionList` construct. This assigns a specific default value to each member of the array. The syntax of the `ParameterExpression` and `ParameterExpressionList` constructs is described in chapter CodAL Basics, section Parameters.

```
Default
   : "default" = ParameterExpression
   | "default" = '{' ParameterExpressionList '}'
```

Overlap defines a target register or a register file that is overlapped by a register or a register file alias.

```
Overlap
   : "overlap" '=' CompoundId
   | "overlap" '=' CompoundId '[' CompileExpression ".." CompileExpression ']'
   | "overlap" '=' CompoundId '[' CompileExpression ']'
   | "overlap" '=' CompoundId '[' CompileExpression ']' '[' CompileExpression ".." Com-
pileExpression ']'
```

`CompileExpression` denotes an index within a target register file or it is used for a bit-select operation when only some bits of the target are aliased. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`ParameterExpression` indicates which attributes can be defined using CodAL parameters. The syntax of the `ParameterExpression` construct is described in chapter CodAL Basics, section Parameters.

`CompoundId` is a target identifier. The syntax of the `CompoundId` construct is described in chapter CodAL Basics, section Compound Id.

Reset enables or disables the reset features. It also defines the type of the reset.

```
Reset
   : "reset" '=' CompileExpression
   | "reset" '=' '{' CompileExpression ',' Id '}'
```

`CompileExpression` denotes whether the register or register file has a reset port.

`Id` defines the type of the reset, it may be `ASYNCHRONOUS` or `SYNCHRONOUS`.

ClockEnable defines a source of clock enable signal. If the signal is `High`, the clock is not gated. If the signal is `Low`, then the clock is gated and the value is not written.

```
ClockEnable
   : "clock_enable" '=' CompoundId
```

`CompoundId` is an identifier of a signal or a register that holds the clock enable signal.

Dff defines behavior of a register file when an IA simulator is generated. By default, the register file behaves as a list of variables. In some cases, it is useful to have a register file that behaves as a list of D flip-flops (e.g. VLIW architectures). To enable the D flip-flop behavior, **dff** has to be set to `true`.

```
Dff
   : "dff" '=' CompileExpression
```

`CompileExpression` denotes whether **dff** behavior is on or off.

Dwarf specifies the DWARF index used for a register file or an array of register files. If the **Dwarf** construct is not used, the DWARF indexes are assigned automatically from the pool of unused indexes. The first mandatory `CompileExpression` value describes the start of the range of DWARF indexes. The second optional `CompileExpression` value describes the end of the range. If the end index is missing, the start index is used, creating a range of a single value. Each member of the register file contains its own DWARF index. For a single register file or architecture array of register files, the range must contain the same number of values as is the size of the register file. For micorarchitecture array of register files, the range must contain number

of values equal to the register file size times the array size. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Dwarf
    : "dwarf" '=' CompileExpression
    | "dwarf" '=' CompileExpression ".." CompileExpression
```

`Aacr` specifies the RISC-V AACR index or indexes of the register file or an array of register files. The principle of operation is almost identical to the **`Dwarf`** construct. The only difference is that if the **`Aacr`** construct is not used, the AACR indexes are not automatically generated and the register file or array of register files have no AACR indexes assigned to them.

```
Aacr
    : "aacr" '=' CompileExpression
    | "aacr" '=' CompileExpression ".." CompileExpression
```

5.1.3 Signal

Signals are used in cycle-accurate models with pipelines. In such models, the registers behave like D-type flip-flops (i.e. the write is visible in the next clock cycle). If the result of computation needs to be passed in the same clock cycle, the signal is used instead of the register.

Only one write is allowed to a given signal each clock cycle.

Arrays of signals can be created. For more information about the arrays, see section 5.3 Arrays.

The following example shows two signals and an array of signals:

Example 19: Signals (model/ca/resources/ca_resources.codal)

```
signal bit[RF_W] s_rd_rA;
signal bit[RF_W] s_rd_rB;

signal bit[32] s_blocks [4];
```

The syntax of a `Signal` construct is as follows:

```
Signal
   : SignalSpecifier "signal" "bit" '[' CompileExpression ']' IdList
   | SignalSpecifier "signal" "bit" '[' CompileExpression ']' IdList '[' Com-
pileExpression ']'
```

The first mandatory `[CompileExpression]` specifies the bit-width of the signal. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

The second optional `[CompileExpression]` specifies that this construct should represent an array of signal resources. The value of the compile expression describes how many signals are in the array. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
SignalSpecifier
   : "dont_touch"
   | %empty
```

`SignalSpecifier` adds a special meaning to the signal. The `SignalSpecifier` is optional. The following values are supported:

- **`dont_touch`** – Specifier denoting that the signal should not be optimized out.

5.1.4 Address Space

The address space defines the view of the memory subsystem seen by the processor. It is used by the loader to know which memory should be used for program and data.

The processor description must contain address spaces for program and data. If the processor has a Von Neumann architecture, then an address space that can be used for program and data is defined. If the processor has a Harvard architecture, then at least two address spaces are defined – one for program and one or more for data. The type of architecture is defined by a **`type`** attribute. Possible values are **`ALL`**, **`PROGRAM`**, and **`DATA`**. The type **`ALL`** denotes that the processor has a Von Neumann architecture. Only one such address space can be defined in the processor description. The types **`PROGRAM`** and **`DATA`** are used in the case of a Harvard architecture.

There can be more than one data address space defined, but only one can be marked as default using the **`default`** attribute. More data address spaces are needed if multiple stacks are required.

The address space definition must contain a list of bounded interfaces that are used for memory access. This information is used by the loader and other tools. One address space can have multiple bounded interfaces. If there are more interfaces and addresses, then the first interface from the left is used for an address handling.

In the following example one address space for a program and data is defined, `as_all`. It defines a Von Neumann architecture and it uses interfaces `if_fe` and `if_ldst`. Since there is no address interval information, a simulator will use the `if_fe` interface for program loading (it is the first interface from the left). The interface `if_ldst` must also be enumerated in the address space, so that the C/C++ compiler and linker are aware of it. The second address space is used for SIMD data access and it uses the `if_simd_ldst` interface. The **`default`** attribute is set to **`false`**.

Example 20: Von Neumann Address space (model/share/resources/arch.codal)

```
// main address space
address_space as_all
{
    bits = { ADDR_W, WORD_W, LAU_W };   // address, word size, LAU
    interfaces = { if_fe, if_ldst };
    type = ALL;
    endianness = BIG;
};
// SIMD data address space
address_space as_simd
{
    bits = { ADDR_W, SIMD_WORD_W, SIMD_LAU_W };
    interfaces = { if_simd_ldst };
    type = DATA;
    endianness = BIG;
    default = false;
};
```

An example of a Harvard architecture follows. In this example, two address spaces are defined. The address space `as_program` uses interface `if_fe`. It is used when the program is loaded into the simulator. The second address space, `as_data`, uses interface `if_ldst` and is used for data handling. Again, the additional address space for SIMD data is defined.

Example 21: Harvard Address spaces (model/share/resources/arch.codal)

```
// program address space
address_space as_program
{
    bits = { ADDR_W, WORD_W, LAU_W };
    interfaces = { if_fe };
    type = PROGRAM;
    endianness = BIG;
};
// data address space
address_space as_data
{
    bits = { ADDR_W, WORD_W, LAU_W };
    interfaces = { if_ldst };
    type = ALL;
    endianness = BIG;
    default = true;
};
// SIMD data address space
address_space as_simd
{
    bits = { ADDR_W, SIMD_WORD_W, SIMD_LAU_W };
    interfaces = { if_simd_ldst };
    type = DATA;
    endianness = BIG;
    default = false;
};
```

The syntax of the `AddressSpace` construct is as follows:

```
AddressSpace
   : "address_space" Id '{' AddressSpaceBody '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
AddressSpaceBody
   : AddressSpaceBody AddressSpaceAttribute ';'
   | AddressSpaceAttribute ';'
```

```
AddressSpaceAttribute
   : Bits
   | Endianness
   | AddressSpaceType
   | "interfaces" '=' '{' AddressSpaceInterfaces OptionalComma '}'
   | "default" '=' CompileExpression
```

`interfaces` contains a list of bounded interfaces.

`default` denotes the default address space for program and/or data. There can be only one default address space for program and/or data.

```
Bits
   : "bits" '=' '{' CompileExpression ',' CompileExpression ',' CompileExpression '}'
```

`Bits` specifies the number of bits for the *address*, the *word* and *LAU*. Word must be divisible by LAU without a remainder. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Endianness
   : "endianness" '=' Id
```

`Endianness` defines the endianness of the interface or memory module. The default value is **`LITTLE`** and the only alternative is **`BIG`**.

```
AddressSpaceType
   : "type" '=' Id
```

type denotes the type of address space (see the text above).

```
AddressSpaceInterfaces
   : AddressSpaceInterfaces ',' AddressSpaceInterface
   | AddressSpaceInterface
```

```
AddressSpaceInterface
   : Id ':' CompileExpression ".." CompileExpression ':' CompoundId
   | CompileExpression ".." CompileExpression ':' CompoundId
   | Id ':' CompoundId
   | CompoundId
```

CompileExpressions are used for address range specification. The syntax of the CompileExpression construct is described in chapter CodAL Basics, section Compile Expression.

CompoundId denotes a bounded interface. The syntax of the CompoundId construct is described in chapter CodAL Basics, section Compound Id.

5.1.5 Pipeline

Pipelines are structures that divide processor tasks into stages. These stages are autonomous and they allow a certain type of parallelism. The stages can be assigned in **register** and **event** definitions.

Each pipeline stage has two built-in functions:

- stall() – when the stall is called within a **semantics** section (e.g. pipe.ID.stall()), all registers assigned to this stage keep their values even if a write occurs in the current clock cycle.
- clear() – when the clear is called within a **semantics** section (e.g. pipe.ID.clear()), all registers assigned to this stage clear their values (i.e. writes a default value) even if a write occurs in the current clock cycle.

The example below shows a pipeline declaration with six stages.

Example 22: Pipeline (model/ca/resources/ca_resources.codal)

```
pipeline pipe { FE, ID, RD, EX, EX2, WB };
```

The syntax of the Pipeline construct is as follows:

```
Pipeline
   : "pipeline" Id '{' PipelineBody OptionalComma '}'
```

Id is an identifier naming a construct. The syntax of the Id construct is described in CodAL Basics on page 5.

```
PipelineBody
   : PipelineBody ',' Id
   | Id
```

Id names a pipeline stage. The syntax of the Id construct is described in CodAL Basics on page 5 .

5.1.6 Ports

Ports are essential for point-to-point communication. A port connection is often used for interrupt handling or for reading/writing from/to buffers (e.g. input/output FIFOs). The ports can be instantiated in two places: in processors as part of the **testbench** construct), and components. A port has one of the following directions:

- Input, denoted by identifier **IN**
- Output, denoted by identifier **OUT**
- Input/output denoted by identifier **INOUT** (Note that this direction is not supported in the current release.)

The following example shows some port definitions:

Example 23: ASIP's ports (model/share/resources/interface.codal)

```
port bit[1]      p_irq       { direction = IN; };         // interrupt request input port
port bit[WORD_W] p_input     { direction = IN; };         // input 32b port
port bit[1]      p_input_en  { direction = IN; };         // input enable signal
port bit[WORD_W] p_output    { direction = OUT; };        // output 32b port
port bit[1]      p_output_en { direction = OUT; };        // output enable signal
port bit[1]      p_halt      { direction = OUT; };        // output indication of halt instruction
port bit[32]     p_error     { direction = OUT; };        // output 32b port with error status
```

The syntax of the `Port` construct is as follows:

```
Port
   : "port" "bit" '[' CompileExpression ']' IdList '{' PortBody '}'
```

`[CompileExpression]` specifies the bit-width of the port. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

```
PortBody
   : PortBody PortAttribute ';'
   | PortAttribute ';'
```

```
PortAttribute
   : "direction" '=' Id
   | "type" '=' Id
```

direction is a mandatory attribute to specify the direction of the port.

type is an optional attribute to specify the type of the port. It can be:

- `DATA` – default, the port is used to pass data, e.g., value of GPIO or value obtained from a sensor.
- `CONTROL` – the port is used for control, e.g., full/empty ports on a FIFO.
- `INTERRUPT` – the port is used as a source of interrupts.

5.1.7 Interfaces

Interface is mainly used for bus and memory connections. It allows fast prototyping of a processor, because it hides implementation details. The latter are specified by a type of interface and by a flag.

Each type consists of two parts. The first denotes a protocol and the second a role. The predefined types of interface are:

- **CPB_LITE:MASTER** and **CPB_LITE:SLAVE** denote direct connection to a standard memory (such as a BRAM on an FPGA).
- **CPB:MASTER** can be used only in a processor, component or cache description and it denotes that the interface has additional master ports.
- **CPB:SLAVE** can be used in memories, caches and components.
- **CPB:MIRRORED_MASTER** and **CPB:MIRRORED_SLAVE** can be used for buses or components only. They have similar behaviour to their **CPB** counterparts but the directions of the ports are opposite.
- **AHB3_LITE:MASTER** can be used only in a processor and component
- **AHB3_LITE:SLAVE** can be used in memories and components.
- **AHB3_LITE:MIRRORED_MASTER** and **AHB3_LITE:MIRRORED_SLAVE** can be used for buses or components only. They have similar behaviour to their **AHB3_LITE** counterparts but the directions of the ports are opposite.
- **AXI4_LITE:MASTER** can be used only in a processor and component
- **AXI4_LITE:SLAVE** can be used in memories and components.
- **AXI4_LITE:MIRRORED_MASTER** and **AXI4_LITE:MIRRORED_SLAVE** can be used for buses or components only. They have similar behaviour to their **AXI4_LITE** counterparts but the directions of the ports are opposite.

The protocol of **CPB** is described in the *Codasip Studio Technical Reference Manual* , chapter "*Codasip Processor Bus*".

The protocol of **AHB3_LITE** is described in *AMBA 3 AHB-Lite Protocol Specification*

The protocol of **AXI4_LITE** is described in *AMBA AXI and ACE Protocol Specification*.

The flag denotes allowed operations. In the case of a fetch unit, writing is not needed, so the interface can be restricted. In other words, the flag denotes which data ports are available on the interface. There are three predefined flags:

- **R** means read-only (data, address and control signals).
- **W** means write-only.
- **RW** means shared address and control signals -data signals are separated. If the interface is a bus interface, only **RW** is possible. For cache interfaces only **R** and **RW** are possible. There is no write-only cache.

Here is an interface example:

Example 24: Interfaces (model/share/resources/interface.codal)

```
// Instruction Bus - FE/DE stage
interface if_fe
{
    bits = { ADDR_W, WORD_W, LAU_W };   // address, word size, LAU
    type = AHB3_LITE:MASTER;
    flag = R;
    endianness = BIG;
};

// Data Bus - EX/EX2 stage
interface if_ldst
{
    bits = { ADDR_W, WORD_W, LAU_W };   // address, word size, LAU
    type = AHB3_LITE:MASTER;
    flag = RW;
    endianness = BIG;
};
```

In this example, two **interface**s are defined. `if_fe` defines a read interface for a fetch unit. `if_ldst` defines a master interface, which can be connected to the AHB3_LITE bus. The address bit width as well as other parts of **bits** are the same for both **interface**s. The endianness is **BIG**.

The syntax of the `Interface` construct is as follows:

```
Interface
```



```
   : "interface" IdList '{' InterfaceBody '}'
```

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

```
InterfaceBody
   : InterfaceBody InterfaceAttribute ';'
   | InterfaceAttribute ';'
```

```
InterfaceAttribute
   : Bits
   | Endianness
   | InterfaceType
   | InterfaceFlag
   | InterfaceAlignment
```

```
Bits
   : "bits" '=' '{' CompileExpression ',' CompileExpression ',' CompileExpression '}'
```

`Bits` specifies the number of bits for the *address*, the *word* and *LAU*. Word must be divisible by LAU without a remainder. The syntax of the `CompileExpression` construct is described in chapter CodAL Basics, section Compile Expression.

```
Endianness
   : "endianness" '=' Id
```

`Endianness` defines the endianness of the interface or memory module. The default value is **`LITTLE`** and the only alternative is **`BIG`**.

```
InterfaceType
   : "type" '=' Id ':' Id
```

`InterfaceType` defines a communication protocol and a role.

```
InterfaceFlag
   : "flag" '=' Id
```

`InterfaceFlag` specifies the read, write or read/write capabilities of the interface.

```
InterfaceAlignment
   : "alignment" '=' '{' InterfaceAlignmentBody '}'
```

```
InterfaceAlignmentBody
   : InterfaceAlignmentAttribute ';'
   | InterfaceAlignmentBody InterfaceAlignmentAttribute ';'
```

```
InterfaceAlignmentAttribute
   : "data" '=' '{' CompileExpressionList '}'
   | "address" '=' CompileExpression
```

```
CompileExpressionList
   : CompileExpressionList ',' CompileExpression
   | CompileExpression
```

`InterfaceAlignment` specifies allowed data and address alignments on data and address bus. In other words, it may contains two parts: **`address`** and **`data`**.

The **address** value specifies the number of bits each word address should be aligned to, it specifies word address boundaries. For example on CPB bus if **address** value specifies 32-bits, reading the 32-bit word can be done only from addresses aligned to 4 bytes (0, 4, 8, ...). But when 8-bits are specified, it enables unaligned access and reading 32-bit word can be done from every address (0, 1, 2, 3, ...).

> **NOTICE:** Codasip memory does not support unaligned access. Simulation and RTL models support word-wide access from word aligned addresses only.

data is a list of values that specify the bitwidth of values which can be accessed in a single transaction within the word boundaries. For example if **data** alignment is `{8, 32}`, then only 32-bit and 8-bit wide transfers can be read, but reading 16-bits is not allowed.

All alignment values should be multiples of the LAU bitwidth.

5.1.8 Usage of Interfaces

An interface can be accessed in a functional or cycle-accurate manner. The functional access is written using a C style. For example: `if_ldst[addr] = 10;`. This statement writes a value to a particular address. The whole word is written (e.g. `10` is expanded to 32bit value). The write is performed immediately without any information about clock-cycles needed for the access. The same access can be written as `if_ldst.write(10, addr);`. If only half-word or byte should be written, the method takes one additional argument, the *byte count*. So, to write one byte, the following statement should be used `if_ldst.write(10, addr, 1);`. The *byte count* is `1` in this example, which means that only a single byte is written to the memory. Reading is done in the same way. The functional access does not depend on any particular memory protocol. It is the same for all of them. Note that functional access may do an address shifting or data masking. So, for instance, there is no need to do any byte shift for **AXI4_LITE** protocol in the model. If an error occurs (e.g., out of range access), the ISS stops and informs an user via an error message. To suppress the stop of the simulator, the error has to be handled. The error code is passed as the last argument of the call (e.g., `ldst.write(10, addr, err);`). If this call fails, the simulator is not stopped and an user should decide next steps based on the error code in `err`. `err` can be a local variable or global resource.

Functional access possibilities are as follows (`data_t` is the type of data handled by interface):

- `data_t read(const codasip_address_t address);`
- `data_t read(const codasip_address_t address, const unsigned bc);`
- `data_t read(const codasip_address_t address, const unsigned bc, error_t& error);`
- `void write(const data_t data, const codasip_address_t address);`
- `void write(const data_t data, const codasip_address_t address, const unsigned bc);`
- `void write(const data_t data, const codasip_address_t address, const unsigned bc, error_t& error);`

To have cycle accurate access, the designer must use the transport functions. In other words, the read or write accesses are split into two or more phases. For instance, an address phase and a data phase (in the way hardware usually works). The phase is the first and only mandatory argument of the transport function. The following arguments of the transport function depends on the memory protocol. In other words, transport function for **CPB** is different from the transport function for **AHB3_LITE**. Let's use **AHB3_LITE** protocol for the following illustration of the transport function. **AHB3_LITE** memory protocol has two phases, one is the address phase and the second one is the data phase. So, there is a transport method for each of the phases. To request data from a memory, first the address phase has to be issued by this transport function `if_ldst.transport(CP_PHS_ADDRESS, CP_AHB_NONSEQ, 0, haddr, CP_AHB_SIZE_32);`. The data phase is handled by the following transport method: `if_ldst.transport(CP_PHS_DATA, hready, hresp, rdata);`.

The following sections describes the transport methods for all supported memory protocols.

5.1.8.1 CPB LiteTransports

`CPB_LITE` protocol has two phases, so two transport methods are available. For the detailed timing description see *Codasip Studio Technical Reference Manual*. `data_t` is the type of data handled by interface.

- `void transport(CP_PHS_ADDRESS, const avalid_t avalid, const write_t write, const codasip_address_t addr, const data_t wdata, const wstrb_t wsrtb)`, where:
  - `avalid` transport is valid,
  - `write` current transaction is write,
  - `addr` is an address,
  - `wdata` write data,
  - `wstrb` write strobe.

  This function is used to initiate either a read or write transaction.
- `void transport(CP_PHS_DATA, response_t& response, data_t& rdata)`, where:
  - `response` is information whether the data/slave is ready,
  - `rdata` is a resource (usually signal or register) that contains data from the slave if the transport finished correctly.

The user can use these built-in constants for `avalid_t`:

- `CP_CPB_VALID`, transaction is valid
- `CP_CPB_INVALID`, transaction is invalid

The user can use these built-in constants for `response_t`:

- `CP_CPB_OKAY`, transaction was successfully completed
- `CP_CPB_ERROR`, an error occurred

The user can use these built-in constants for `write_t`:

- `CP_CPB_READ`, read request
- `CP_CPB_WRITE`, write request

5.1.8.2 CPB Transports

`CPB` protocol has two phases, so two transport methods are available. For the detailed timing description see the *Codasip Studio Technical Reference Manual*. `data_t` is the type of data handled by interface.

- `void transport(CP_PHS_ADDRESS, aready_t& aready, const avalid_t avalid, const write_t write, const codasip_address_t addr, const data_t wdata, const wstrb_t wsrtb)`, where:
  - `aready` signals slave/bus state, i.e. whether a new transaction can be made,
  - `avalid` transport is valid,
  - `write` current transaction is write,
  - `addr` is an address,
  - `wdata` write data,
  - `wstrb` write strobe.

  This function is used to initiate either a read or write transaction.
- `void transport(CP_PHS_DATA, valid_t& valid, response_t& response, data_t& rdata)`, where:
  - `valid` signals that the previous transaction is completed,
  - `response` is information whether the data/slave is ready,
  - `rdata` is a resource (usually signal or register) that contains data from the slave if the transport finished correctly.

The user can use these built-in constants for `aready_t`:

- **CP_CPB_READY**, slave can accept a new transaction
- **CP_CPB_WAIT**, slave cannot accept a new transaction

The user can use these built-in constants for `avalid_t`:

- **CP_CPB_VALID**, transaction is valid
- **CP_CPB_INVALID**, transaction is invalid

The user can use these built-in constants for `valid_t`:

- **CP_CPB_VALID**, result/data is valid
- **CP_CPB_INVALID**, result/data is invalid

The user can use these built-in constants for `response_t`:

- **CP_CPB_OKAY**, transaction was successfully completed
- **CP_CPB_ERROR**, an error occurred

The user can use these built-in constants for `write_t`:

- **CP_CPB_READ**, read request
- **CP_CPB_WRITE**, write request

5.1.8.3 AHB3_LITE Transports

AHB3_LITE protocol has two phases, so two transport methods are available. For the detailed timing and signal description see the *AMBA 3 AHB-Lite Protocol Specification*. `data_t` is the type of data handled by interface.

- `void transport(CP_PHS_ADDRESS, const htrans_t htrans, const hwrite_t hwrite, const codasip_address_r haddr, const hsize_t hsize, const hprot_t hprot = CP_AHB_PROT_DEFAULT, const hmastlock_t hmastlock = CP_AHB_UNLOCK, const hburst_t hburst = CP_AHB_SINGLE)`, where:
  - `htrans` denotes type of a transaction, currently only **CP_AHB_IDLE** and **CP_AHB_NONSEQ** are supported.,
  - `hwrite` denotes whether the transaction is reading or writing one,
  - `haddr` is an address,
  - `hsize` denotes a size of a transfer, it can be one or more bytes,
  - `hprot` denotes a mode of a transaction, currently only **CP_AHB_PROT_DEFAULT** is supported by Codasip IPs (a user may take an advantage of this signal in his IPs),
  - `hmastlock` is used for an atomic transactions (not supported by Codasip IPs),
  - `hburst` denotes burst mode.

  This function is used to initiate either a read or write transaction.
- `void transport(CP_PHS_DATA, hread_t& hready, hresp_t& hresp, data_t& hrdata, const data_t hwdata)`, where:
  - `hready` signalizes that the slave is ready to accept a transaction,
  - `hresp` signalizes that the slave finished the transfer successfully or with a failure,
  - `hrdata` is data from a slave,
  - `hwdata` is data for a slave.

  This function is used to finish either a read or write transaction. Note that if the interface is read-only or write-only, then `hrdata` or `hwdata` are not in the list of arguments.

There are two helper functions that shifts bytes to LSB or to a proper byte line.

- `data_t encode_data(const data_t hwdata, constunsigned byte_line, const hsize_t hsize)`, where:

- `hwdata` is data to be encoded,
- `byte_line` is an index of a byte line (i.e. an address offset),
- `hsize` denotes a size of `hwdata`.

This function returns an encoded data to a proper byte line. If the value `0xff` should be encoded to an address `0x3`, and `hsize` is **`CP_AHB_SIZE_8`**, then `encode_data` creates either `0x00ff0000` or `0x0000ff00` (depending on the endiannes).

- `data_t decode_data(const data_t hrdata, constunsigned byte_line, const hsize_t hsize)`, where:
  - `hrdata` is data to be decoded,
  - `byte_line` is an index of a byte line (i.e. an address offset),
  - `hsize` denotes a size of `hrdata`.

  This function returns a decoded data from a given byte line. For instance, if the `byte_line` is `3` and `hsize` is **`CP_AHB_SIZE_8`** and if a value `0x00ff0000` or `0x0000ff00` (depending on the endianness), then the function returns `0xff`.

The user can use these built-in constants for `htrans_t`:

- **`CP_AHB_IDLE`**
- **`CP_AHB_BUSY`**
- **`CP_AHB_NONSEQ`**
- **`CP_AHB_SEQ`**

The user can use these built-in constants for `hready_t`:

- **`CP_AHB_READY`**
- **`CP_AHB_WAIT`**

The user can use these built-in constants for `hresp_t`:

- **`CP_AHB_OKAY`**
- **`CP_AHB_ERROR`**

The user can use these built-in constants for `hwrite_t`:

- **`CP_AHB_READ`**
- **`CP_AHB_WRITE`**

The user can use these built-in constants for `hsize_t`:

- **`CP_AHB_SIZE_8`**
- **`CP_AHB_SIZE_16`**
- **`CP_AHB_SIZE_32`**
- **`CP_AHB_SIZE_64`**
- **`CP_AHB_SIZE_128`**
- **`CP_AHB_SIZE_256`**
- **`CP_AHB_SIZE_512`**
- **`CP_AHB_SIZE_1024`**

The user can use these built-in constants for `hprot_t`:

- **`CP_AHB_DATA_ACCESS`**
- **`CP_AHB_PRIVILEGED`**
- **`CP_AHB_BUFFERABLE`**

- **CP_AHB_CACHEABLE**
- **CP_AHB_PROT_DEFAULT**

The user can use these built-in constants for `hmastlock_t`:

- **CP_AHB_UNLOCK**
- **CP_AHB_LOCK**

The user can use these built-in constants for `hburst_t`:

- **CP_AHB_SINGLE**
- **CP_AHB_INCR**
- **CP_AHB_WRAP4**
- **CP_AHB_INCR4**
- **CP_AHB_WRAP8**
- **CP_AHB_INCR8**
- **CP_AHB_WRAP16**
- **CP_AHB_INCR16**

5.1.8.4 AXI4_LITE Transports

AXI4_LITE protocol has five phases, so five transport methods are available. For the detailed timing and signal description see the *AMBA AXI and ACE Protocol Specification*. `data_t` is the type of data handled by interface.

- **void** `transport(CP_PHS_ADDRESS_READ, const avalid_t arvalid, aready_t& arready, const codasip_adress_t araddr, const aprot_t arprot)`, where:
  - `arvalid` denotes whether the master is ready for a read transaction,
  - `arready` denotes whether the slave is ready to accept a read transaction,
  - `araddr` is an address,
  - `arprot` is a protection mode, Codasip IPs ignores this signal (a user may take an advantage of this signal in his IPs).

  This function is used to initiate a read transaction.
- **void** `transport(CP_PHS_ADDRESS_WRITE, const avalid_t awvalid,aread_t& awready, const codasip_adress_t awaddr, const aprot_t awprot)`, where:
  - `awvalid` denotes whether the master is ready for a write transaction,
  - `awready` denotes whether the slave is ready to accept a write transaction,
  - `awaddr` is an address,
  - `awprot` is a protection mode, Codasip IPs ignores this signal (a user may take an advantage of this signal in his IPs).

  This function is used to initiate a write transaction.
- **void** `transport(CP_PHS_DATA_READ, avalid_t& rvalid, const aready_t rready, aresp_t& rresp, data_t& rdata)`, where:
  - `rvalid` denotes whether a slave is able to finish the read transaction and that it sent data to master,
  - `rready` denotes whether a master is ready and that it is able to accept the data,
  - `rresp` denotes whether the slave successfully completed the read transaction,
  - `rdata` is data from the slave.

  This function is used to finish a read transaction.
- **void** `transport(CP_PHS_DATA_WRITE, const avalid_t wvalid, aready_t& wready, aresp_t& wresp, const data_t wdata)`, where:
  - `wvalid` denotes whether master it is able to finish the write transaction and that it sent data to the slave,

  - `wready` denotes whether a slave it is able to accept the data,
  - `wresp` denotes whether the slave successfully completed the write transaction,
  - `wdata` is data for the slave.

  This function is used to give data for a write transaction.
- **void** `transport(CP_PHS_RESPONSE_WRITE, avalid_t& bvalid,` **const** `aready_t bready, aresp_t& bresp)`, where:
  - `bvalid` denotes whether the slave finished the write,
  - `bready` denotes whether the master is ready to finish the write transaction,
  - `bresp` denotes whether the slave finished the write transaction successfully.

  This function is used to finish a write transaction.

There are three helper functions that shifts data or write strobe to LSB or a proper byte line.

- `data_t encode_data(`**const** `data_t wdata,` **constunsigned** `byte_line,` **const** `asize_t asize)`, where:
  - `wdata` is data to be encoded,
  - `byte_line` is a index of a byte line (i.e. an address offset),
  - `asize` denotes a size of data.

  This function returns encoded data to a proper byte line. If the value `0xff` should be written to an address `0x3`, and `asize` is **`CP_AHB_SIZE_8`**, then `encode_data` creates either `0x00ff0000` or `0x0000ff00` (depending on the endianness).
- `data_t decode_data(`**const** `data_t rdata, const unsigned byte_line, const asize_t asize)`, where:
  - `rdata` is data to be decoded,
  - `byte_line` is a index of a byte line (i.e. an address offset),
  - `asize` denotes a size of data.

  This function returns decoded data from a given byte line. For instance, if the `byte_line` is `3` and `asize` is **`CP_AHB_SIZE_8`** and if a value `0x00ff0000` or `0x0000ff00` (depending on the endianness), then the function returns `0xff`.
- `data_t encode_wstrb(`**const** `data_t wstrb,` **constunsigned** `byte_line,` **const** `asize_t asize)`, where:
  - `wstrb` is a value of a write strobe,
  - `byte_line` is a index of a byte line (i.e. an address offset),
  - `asize` denotes a size of data.

  This function returns encoded write strobe signal. For instance, if the `byte_line` is `0x2` and `asize` is **`CP_AXI_SIZE_16`** and if a value of write strobe is `0x3`, then the function returns `0xC`.

The user can use these built-in constants for `avalid_t`:

- **`CP_AXI_VALID`**
- **`CP_AXI_INVALID`**

The user can use these built-in constants for `aready_t`:

- **`CP_AXI_READY`**
- **`CP_AXI_WAIT`**

The user can use these built-in constants for `aprot_t`:

- **`CP_AXI_PROT_INSTR`**
- **`CP_AXI_PROT_NON_SECURE`**
- **`CP_AXI_PROT_PRIVILEGED`**

The user can use these built-in constants for `asize_t`:

- **`CP_AXI_SIZE_8`**
- **`CP_AXI_SIZE_16`**
- **`CP_AXI_SIZE_32`**
- **`CP_AXI_SIZE_64`**
- **`CP_AXI_SIZE_128`**
- **`CP_AXI_SIZE_256`**
- **`CP_AXI_SIZE_512`**
- **`CP_AXI_SIZE_1024`**

The user can use these built-in constants for `aresp_t`:

- **`CP_AXI_OKAY`**
- **`CP_AXI_EXOKAY`**
- **`CP_AXI_SLVERR`**
- **`CP_AXI_DECERR`**

5.1.9 Arbiter

Arbiters can be modeled with the usage of **`arbiter`** construct. Arbiter is a component that connects multiple `MIRRORED_MASTER` input interfaces to a single `MASTER` output interface.

Arbiter is defined by its interfaces and by a priority list. This list contains all of the arbiter input interfaces sorted by a priority order. As arbiter has only a single output, input transactions within a single clock cycle must be prioritized. Only the active transaction with the high input priority can be send to the output within the same cycle. The rest of transactions is buffered and forwarded in subsequent clock cycles.

Arbiter supports maximum of 255 input AHB interfaces.

Example 25: Arbiter example (model\share\resources\interface.codal)

```
// ASIP interface connected ín most cases to the memory
interface if_ldst {
       bits = { ADDR_W, WORD_W, LAU_W };
       type = AHB3_LITE:MASTER;
       flag = RW;
       endianness = ENDIAN;
       alignment = {
              address = WORD_W;
              data = { LAU_W, 2*LAU_W, WORD_W };
       };
};

arbiter a_neuman {
       // order defines priority of the input interfaces
       priority = { if_ldst, if_fetch };

       // core's interface to data memory (input)
       interface if_ldst {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:MIRRORED_MASTER;
              flag = RW;
              endianness = ENDIAN;
       };

       // core's interface to code memory (input)
       interface if_fetch {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:MIRRORED_MASTER;
              flag = R;
```

```
                endianness = ENDIAN;
                // Specification allowed data and address alignments.
                alignment = {
                        // Data is a list of bitwidth which can be accessed in a single transaction.
                        data = { WORD_W };
                };
        };

        // arbiter output interface
        interface if_neumann {
                bits = { ADDR_W, WORD_W, LAU_W };
                type = AHB3_LITE:MASTER;
                flag = RW;
                endianness = ENDIAN;
        };
};

connect a_neumann.if_neumann => if_ldst;

/* Memory access
        a_neumann.if_ldst.transport/read/write(...);
        a_neumann.if_fetch.transport/read/write(...);
*/
```

The syntax of the **`Arbiter`** construct is as follows:

```
Arbiter
        : "arbiter" Id '{' ArbiterBody '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
ArbiterBody
        : ArbiterBody ArbiterAttribute ';'
        | ArbiterAttribute ';'
```

```
ArbiterAttribute
        : Interface
        | ArbiterTcmPriority
```

```
ArbiterTcmPriority
        : "priority" '=' '{' ArbiterTcmPriorityBody '}'
```

`ArbiterTcmPriority` specifies a list of arbiter input interfaces. The leftmost interface in the list has the highest priority.

```
ArbiterTcmPriorityBody
        : Id
        | ArbiterTcmPriorityBody ',' Id
```

5.1.10 Interconnect

An interconnect can be modeled using the **`interconnect`** construct.

In order to define an interconnect, we must define its interfaces, specify decoders and control ports. For each of the decoders we must specify its input and output interfaces. Those can be defined either by an address range (base address and size) or defined as a default interface. Additionally, we can specify an enable port for each output interface. That port will control

whether the interface is enabled and available, or disabled, in which case any access to its address range will fall through to the default interface.

In the following example, we have an interconnect `i1` connected to a TCM interface and an interface that is routed outside of the ASIP. Its decoders `dec_fetch` and `dec_ldst` connect the ASIP inputs to either to the TCM (`if_itcm_fetch`, `if_itcm_ldst`, `if_dtcm_ldst`) or to one of the ASIP output interfaces. Routing is controlled by parametrized address ranges (`ITCM_BASE_ADDR`, `ITCM_SIZE`, etc.) and enable ports (`p_itcm_en`, `p_dtcm_en`).

Example 26: Interconnect example (model\share\resources\interconnect.codal)

```
interconnect i1 {
	// interfaces controlled by the ASIP core
	interface if_cpu_fetch {
		type = AHB3_LITE:MIRRORED_MASTER;
		flag = R;
			bits = { ADDR_W, WORD_W, LAU_W };
			endianness = ENDIAN;
			alignment = {
				address = WORD_W;
				data = { LAU_W, 2*LAU_W, WORD_W };
			};
	};
	interface if_cpu_ldst {
		type = AHB3_LITE:MIRRORED_MASTER;
		flag = RW;
			bits = { ADDR_W, WORD_W, LAU_W };
			endianness = ENDIAN;
			alignment = {
				address = WORD_W;
				data = { LAU_W, 2*LAU_W, WORD_W };
			};
	};

	// TCM/memory connections
	interface if_itcm_fetch {
		type = AHB3_LITE:MASTER;
		flag = R;
			bits = { ADDR_W, WORD_W, LAU_W };
			endianness = ENDIAN;
			alignment = {
				address = WORD_W;
				data = { LAU_W, 2*LAU_W, WORD_W };
			};
	};
	interface if_itcm_ldst, if_dtcm_ldst {
		type = AHB3_LITE:MASTER;
		flag = RW;
			bits = { ADDR_W, WORD_W, LAU_W };
			endianness = ENDIAN;
			alignment = {
				address = WORD_W;
				data = { LAU_W, 2*LAU_W, WORD_W };
			};
		};

	// TCM enable signals
	port bit [1] p_itcm_en { direction = IN; type = CONTROL; };
	port bit [1] p_dtcm_en { direction = IN; type = CONTROL; };

	// interfaces routed outside of the ASIP
	interface if_fetch {
		type = AHB3_LITE:MASTER;
		flag = R;
			bits = { ADDR_W, WORD_W, LAU_W };
			endianness = ENDIAN;
			alignment = {
				address = WORD_W;
				data = { LAU_W, 2*LAU_W, WORD_W };
			};
	};
	interface if_ldst {
		type = AHB3_LITE:MASTER;
```

```
                flag = RW;
                        bits = { ADDR_W, WORD_W, LAU_W };
                        endianness = ENDIAN;
                        alignment = {
                                address = WORD_W;
                                data = { LAU_W, 2*LAU_W, WORD_W };
                        };
        };

        // decoders for fetch unit
        decoder dec_fetch {
                interface = if_cpu_fetch;
                interfaces = {
                        // base : interface + configuration
                        ITCM_BASE_ADDR :
                                if_itcm_fetch
                                {
                                        enable = p_itcm_en;
                                        size = ITCM_SIZE;
                                },
                        default : if_fetch
                };
        };
        // decoders for ldst unit
        decoder dec_ldst {
                interface = if_cpu_ldst;
                interfaces = {
                        ITCM_BASE_ADDR : if_itcm_ldst { size = ITCM_SIZE; enable = p_itcm_en; },
                        DTCM_BASE_ADDR : if_dtcm_ldst { size = DTCM_SIZE; enable = p_dtcm_en; },
                        default : if_ldst
                };
        };
};

// TCM connections
connect i1.if_itcm_fetch => idtcm.if_itcm_fetch;
connect i1.if_itcm_ldst => idtcm.if_itcm_ldst;
connect i1.if_dtcm_ldst => idtcm.if_dtcm_ldst;
// connections outside the core
connect i1.if_fetch => if_fetch;
connect i1.if_ldst => if_ldst;
```

The syntax of the **`interconnect`** construct is as follows:

```
Interconnect
        : "interconnect" Id '{' InterconnectBody '}'
```

`Id` is an identifier naming the interconnect. The syntax of the **`Id`** construct is described in CodAL Basics on page 5.

```
InterconnectBody
        : InterconnectBody InterconnectAttribute ';'
        | InterconnectAttribute ';'


InterconnectAttribute
        : Interface
        | Port
        | InterconnectDecoder
```

`InterconnectDecoder` defines a decoder component inside the interconnect. The syntax is as follows:

```
InterconnectDecoder
        : "decoder" Id '{' InterconnectDecoderBody '}'
```

`Id` is an identifier naming the decoder. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
InterconnectDecoderBody
        : InterconnectDecoderBody InterconnectDecoderAttribute ';'
```

```
       | InterconnectDecoderAttribute ';'

InterconnectDecoderAttribute
       : "interface" '=' Id
       | "interfaces" '=' '{' InterconnectDecoderInterfacesBody OptionalComma '}'
       ;
```

The `interface` attribute defines the input interface of the decoder. `Id` refers to one of the interfaces defined in the interconnect. The syntax of the Id construct is described inCodAL Basics on page 5.

The `interfaces` attributes define output interfaces of the decoder.

```
InterconnectDecoderInterfacesBody
       : InterconnectDecoderInterfacesBody ',' InterconnectDecoderInterfacesDeclaration
       | InterconnectDecoderInterfacesDeclaration
       ;

InterconnectDecoderInterfacesDeclaration
       : "default" ':' Id
       | ParameterExpression ':' Id '{' InterconnectDecoderInterfacesDeclarationBody '}'
```

`ParameterExpression` defines the base address of the interface, the **`default`** construct is used to define a fallback interface. `Id` refers to one of the interfaces defined in the interconnect. The syntax of the `Id` and `ParameterExpression` constructs is described inCodAL Basics on page 5.

```
InterconnectDecoderInterfacesDeclarationBody
       : InterconnectDecoderInterfacesDeclarationBody Inter-
connectDecoderInterfacesDeclarationAttribute ';'
       | InterconnectDecoderInterfacesDeclarationAttribute ';'
       ;

InterconnectDecoderInterfacesDeclarationAttribute
       : Size
       | "enable" '=' Id
       ;
```

The `enable` attribute specifies which port defined in the interconnect is used to enable or disable this output interface. The syntax of the `Id` construct is described inCodAL Basics on page 5.

5.1.11 Tightly Coupled Memory

Tightly coupled memory (TCM) can be modeled using the **`tcm`** construct. The TCM is a small memory that provides fast access. The TCM has one `SLAVE` interface that is used by the SoC, and one or more `MIRRORED_MASTER` interfaces that provide access from the core. There is an arbiter between the interfaces which controls an access to the TCM's SRAM memory.

Each TCM has its own address range defined by a base address and size. The base address is specified by a constant. The value of the base address must be a multiple of the size. This allows for efficient addressing of the TCM. It is also possible to define more than one TCM with different address ranges, but using a common `SLAVE` interface.

The TCM definition must contain a priority list of the interfaces that can access the TCM. The priority list must contain the `SLAVE` interface and at least one of the `MIRRORED_MASTER` interfaces. In the example below, the priority list of program `itcm` contains all the interfaces: *if_slave*, *if_fetch*, *if_ldst*, but the priority list of data `dtcm` only contains *if_slave* and *if_ldst*.

Only AHB interfaces are supported.

Example 27: TCM example (model\share\resources\tcm.codal)

```
// ASIP slave interface connected in most cases to the interconnect
interface if_slave {
       bits = { ADDR_W, WORD_W, LAU_W };
       type = AHB3_LITE:SLAVE;
       flag = RW;
       endianness = ENDIAN;
};

tcm idtcm {
       // slave interface
       interface if_slave {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:SLAVE;
              flag = RW;
              endianness = ENDIAN;
       };
       // fetch interface used by the core
       interface if_fetch {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:MIRRORED_MASTER;
              flag = R;
              endianness = ENDIAN;
              alignment = {
                     // Data is a list of bitwidth which can be accessed in a single transaction.
                     data = { WORD_W };
              };
       };
       // ldst interface used by the core
       interface if_ldst {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:MIRRORED_MASTER;
              flag = RW;
              endianness = ENDIAN;
       };

       // ITCM
       program itcm {
              size = TCM_SIZE;
              base = TCM_BASE;
              priority = {if_ldst, if_fetch, if_slave};
       };
};

connect if_slave => idtcm.if_slave;
```

Example 28: Separate program and data TCM (model\share\resources\tcm.codal)

```
// ASIP slave interface connected in most cases to the interconnect
interface if_slave {
       bits = { ADDR_W, WORD_W, LAU_W };
       type = AHB3_LITE:SLAVE;
       flag = RW;
       endianness = ENDIAN;
};

tcm idtcm {
       // slave interface
       interface if_slave {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:SLAVE;
              flag = RW;
              endianness = ENDIAN;
       };
       // fetch interface used by the core
       interface if_itcm_fetch {
              bits = { ADDR_W, WORD_W, LAU_W };
              type = AHB3_LITE:MIRRORED_MASTER;
              flag = R;
              endianness = ENDIAN;
              alignment = {
                     // Data is a list of bitwidth which can be accessed in a single transaction.
```

```
                        data = { WORD_W };
                };
        };
        // ldst interface used by the core
        interface if_itcm_ldst, if_dtcm_ldst {
                bits = { ADDR_W, WORD_W, LAU_W };
                type = AHB3_LITE:MIRRORED_MASTER;
                flag = RW;
                endianness = ENDIAN;
        };

        // ITCM
        program itcm {
                size = ITCM_SIZE;
                base = ITCM_BASE_ADDR;
                priority = {if_itcm_ldst, if_itcm_fetch, if_slave};
        };

        // DTCM
        data dtcm {
                size = DTCM_SIZE;
                base = DTCM_BASE_ADDR;
                priority = {if_dtcm_ldst, if_slave};
        };
};

connect if_slave => idtcm.if_slave;
```

The syntax of the **`tcm`** construct is as follows:

```
Tcm
        : "tcm" IdList '{' TcmBody '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
TcmBody
        : TcmBody TcmAttribute ';'
        | TcmAttribute ';'
```

```
TcmAttribute
        : TcmDefinition
        | Interface
```

```
TcmDefinition
        : "data" Id '{' TcmDefinitionBody '}'
        | "program" Id '{' TcmDefinitionBody '}'
```

`TcmDefinition` defines a TCM memory with its size, base address and specifies which interfaces have an access to it.

```
TcmDefinitionBody
        : TcmDefinitionBody TcmDefinitionAttribute ';'
        | TcmDefinitionAttribute ';'
```

```
TcmDefinitionAttribute
        : Size
        | "base" '=' ParameterExpression
        | ArbiterTcmPriority
```

```
ArbiterTcmPriority
       : "priority" '=' '{' ArbiterTcmPriorityBody '}'
```

`ArbiterTcmPriority` specifies a list of interfaces that access the TCM definition. The leftmost interface in the list has the highest priority.

```
ArbiterTcmPriorityBody
       : Id
       | ArbiterTcmPriorityBody ',' Id
```

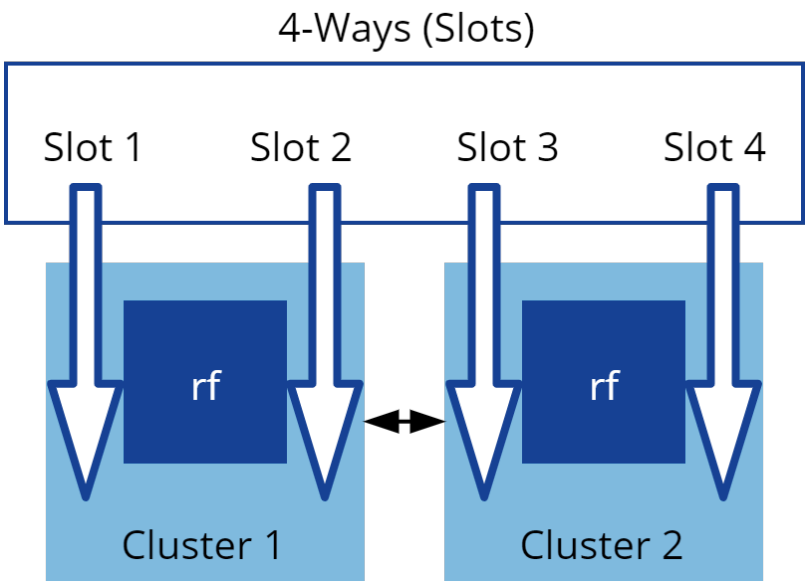
5.2 Module

Basic idea behind the module construct is to have a way how to collect resources, elements, sets, and event into the same name space. There is several places where this is helpful. For instance a designer may place a pipeline stage into one module, create a clusters in VLIW architectures, etc. Placing those constructs into one module influences a high level synthesis as well as other areas. Module definition can be spread over several files. The final definition consists of all locally used definitions. Module can be placed in another module as well (hierarchical modeling). Let's look at VLIW architectures into more details, so the usage of modules will be more obvious.

VLIW micro-architectures use several instruction pipelines (slots) to support instruction-level parallelism. These slots allow the execution of instructions on different sets of resources (used by particular slots), called *clusters*.

Figure 5 illustrates this concept. For such an architecture, two variants of certain instruction have to be described (i.e. one for each of the two clusters). Clusters are modeled using **`module`** construct. Any resource, an **`element`**, **`set`**, or **`event`** can be assigned to a module. When this is done, the code within the **`semantics`** and **`return`** sections is handled individually for each module. Hence, in the end, there is only one description of the instruction (i.e. the assembler code is the same for any module), but the resources and implementation for a given instruction may be different for a different modules.

Figure 5: Cluster scheme



Although the modules use different resources, their names inside a **`semantics`** and **`return`** sections can be the same. When a compiler tries to find a resource, it starts in the closest module. If the resource is not found there, it searches in the one level above the current one until a global namespace is hit. In some cases, the resources can be identified by a full name (i.e. a

name with slot name as a prefix). This can be useful in the case that all slots contain an instruction which accesses the same resource in a particular module. There is also a way how to address an item within a module absolutely (the identifier starts with a dot).

There is usually an instruction which can load data from any set of resources. This can be also modeled in CodAL.

In the following example, we declare two modules, `m_0` and `m_1`. Both modules contain a register file of eight, 16-bit registers `rf_gp`. There is also an 8-bit register called `r_flags`. Moreover there is a local event `mult` that is available in both modules. In fact, the event is instantiated in each module.

Example 29: Modules declaration (model/ca/resources/ca_resources.codal)

```
// Modules
module m_0, m_1
{
    // general purpose registers
    register_file bit[16] rf_gp
    {
        size = 16;
        dataport r0, r1, r2 { flag = R; };
        dataport w0 { flag = W; };
    };
    // flags (carry, overflow, ...)
    register bit[8] r_flags;

    // event in module
    event mult : pipeline (pipe.EX)
    {
        // local resource used for re-timing
        register bit[32] r_tmp;

        semantics
        {
            r_tmp = r_ex_op1 * r_ex_op2;
            r_wb_in = r_tmp;
        };
    };
};
```

The next example shows how to access the register within a module, in the **`semantics`** section of an **`event`** or **`element`**. Let's assume that the **`element`** is instantiated twice, once for `m_0` and a second time for `m_1`. There are three statements in the **`semantics`** section. The first statement does exactly the same thing in both instances (i.e. it accesses the same `r_flags` register). The semantics of the second statement depends on the instance (i.e. one time it accesses `m_0.r_flags` and the second time is accesses `m_1.r_flags`).

In some cases, we need to specify that part of a **`semantics`** section is specific to a particular module. This is done using `#pragma module` statement. In this example, the effect is that the last statement is present only for `m_1`. Note that multiple modules can be specified in the `#pragma module` statement (e.g. `#pragma m_0, m_1`).

Example 30: Semantics with modules (model/ca/decoders/ca_decoder.codal)

```
semantics
{
    // Access r_flags in m_0 in all instances of the element
    .m_0.r_flags = 0;
    // Access r_flags in m_0 or m_1
    r_flags = 0;
    #pragma module m_1
    {
        // Access r_flags only in the instance of element in m_1
        r_flags = 0;
    }
};
```

The last example shows modules in **use** section. Within the use section, two decoders are instantiated. Each one is from a different module. Also resources used during the decoding are taken from a different modules.

Example 31: Using modules in the use section (model/ca/pipelines/ca_pipe_stage1_id.codal)

```
event decode : pipeline(pipe.ID)
{
    use m_0.dec as dec_0;
    use m_1.dec as dec_1;

    semantics
    {
        dec_0(m_0.s_id_opcode, m_0.s_iis);
        dec_1(m_1.s_id_opcode, m_1.s_iis);
    };
};
```

The syntax of the `Module` construct is as follows:

```
Module
   : "module" IdList Properties '{' ModuleBody '}'
```

`IdList` is a list of identifiers, separated by a comma, naming the instances of the construct. The syntax of the `Idlist` construct is described in CodAL Basics on page 5.

```
ModuleBody
   : ModuleBody TranslationUnitBody
   | TranslationUnitBody
```

`TopConstruct` stands for a subset of all supported constructs that can be used within a module. See Processor Description on page 24.

5.3 Arrays

CodAL supports arrays for several types of resources. Local variables inside semantic sections or functions can also be arrays. Arrays in the form of local variables behave in the same way as arrays in ANSI C.

The following resources can be used as arrays:

- *Register*, see section 5.1.1 Register for more details.
- *Register File*, see section 5.1.2 Register File for more details.
- *Signal*, see section 5.1.3 Signal for more details.

An array of resources can be either architectural or microarchitectural. All arrays of signals and in local variables are defined as microarchitectural. The arrays of registers and register files are defined as microarchitectural unless **arch** or **pc** specifier is used.

The microarchitectural arrays have no specific semantics for the architecture and there are no restrictions on their use.

The architectural arrays are more specific and are used in creation of multi-thread designs. Each architecture array must have its size equal to the number of threads in the design (see section 5.4.8 Settings).

Another difference between the architectural and microarchitectural arrays is in the DWARF and AACR indexes used by the debugger. For the microarchitectural arrays, every member of the array has a different index, while for architectural arrays each member of the array has the same index and the debugger distinguishes them based on the selected thread. The range of the DWARF and AACR indexes specified by the user must reflect these rules.

The arrays can be used in several places:

- Program counter register, see section 5.1.1 Register.
- Several compiler settings, see section 5.4.8.1 C/C++ Compiler Settings.
- Several debugger settings, see section 5.4.8.2 Debugger Settings.
- Excluded resources in WFI settings, see 5.4.8.5 WFI Settings.
- Bodies of semantics sections, functions and emulations. In these parts of code, the arrays can be used in the following ways:
    - An assignment between two arrays of the same size and member type.
    - An access to an array member using the "`[]`" operator. Only expressions of scalar unsigned integer types can be used to index the array.

Example 32: Arrays usage

```
settings
{
    architecture
    {
        thread_count = 2;
    };
    compiler
    {
        stack_pointer = rf_gpr[8];
        base_pointer = rf_gpr[9];
    };
    debugger
    {
        instruction_valid = r_instr_valid;
    };
};

pc register bit[32] r_pc [2]
{
    default = { 0x0, 0x1000 };
};

arch register_file bit[32] rf_gpr [2]
{
    size = 16;
    dataport r0, r1 { flag = R;};
    dataport w0 { flag = W;};
    dwarf = 0x100 .. 0x10f;
    aacr = 0x1000 .. 0x100f;
};

register bit[1] r_instr_valid [2]
{
    default = 1;
    dwarf = 0x110 .. 0x111;
}

event main
{
    r_pc[1] = r_pc[0];
    r_pc[0] += 4;
    rf_gpr[0].w0[10] = 0; // CA model
    rf_gpr[1][10] = 0; // IA model
};
```

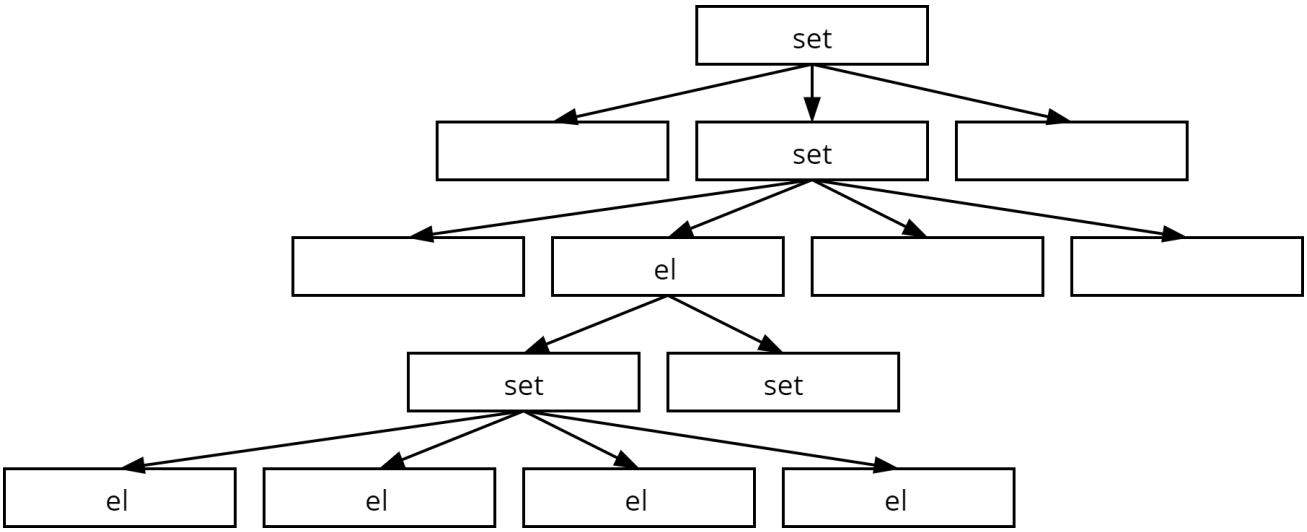
5.4 Instruction Set

CodAL supports many types of architectures. RISC, CISC, DSP and VLIW instruction sets are fully supported and can be easily described.

Each instruction set is defined using two basic constructs: **`element`** and **`set`**. Basic **`element`**s (e.g. describing individual operands and opcodes) are defined first, then more complex **`element`**s and **`set`**s assemble the simpler ones into real

instructions. This creates a tree structure that captures the instruction set of a processor, as shown in the following figure (`el` is an abbreviation for **`element`**):

Figure 6: Instruction set description



The root (or roots in the case of VLIW) of the instruction set are specified in a **`start`** section (see Start Section on page 57).

The assembler syntax is defined in the **`assembler`** part of the **`settings`** (see Assembler Settings on page 84).

5.4.1 Start Section

The **`start`** section represents the top level construct for the grammar describing the assembly and binary language for the modeled processor. In other words, this section represents the *root***`element`**s or **`set`**s for all the instructions implemented in the processor. In the case of RISC, CISC or DSP, only one root is specified. In the case of VLIW, multiple roots are specified, one per slot of the VLIW. Each root is encapsulated using brackets.

When the generated instruction decoder (or instruction encoder in the case of the assembler) processes an instruction, it starts from the root(s) and searches for instances of **`set`**s or **`element`**s that are present as the sources. It is therefore used mainly during the generation of the programming tools.

In the most cases, only one **`start`** section is needed in a model. In some special cases, when processor supports multiple instructions sets, more **`start`** sections are allowed. Each **`start`** section describes a single instruction set. Programming tools are able to recognize instruction sets automatically. Simulation tools and RTL should contain a hand written logic (e.g. check on a particular register) that handles the correct instruction set mode during a runtime.

In the case of multiple start sections, each **`start`** section has to have a name. If only one **`start`** section is needed, the name is optional.

In some cases, emulations are needed for for C/C++ compiler generator. Because the emulations are always tightly coupled with the instruction set, the **`start`** section allows to connect one or more emulations to the instruction set. Emulations are described in details in Emulations on page 72.

VLIW architectures often use code compression and NOPs somehow removes to be able to achieve low code density. This task is also know as *bundling/debundling*. Each instruction set may have separate rules how to do it, so bundling and debundling is tightly coupled with the instruction set (similarly to emulations). Bundling and debundling is described in details in Bundling/Debundling on page 77.

The following example shows a single instruction set definition.

Example 33: Start section (model/share/isa/isa.codal)

```
start
{
    roots = { isa };
    emulations = { e_li };
};
```

`isa` is a reference to the **`set`** that includes all of the instructions of the processor.

The syntax of the `Start` construct is as follows:

```
Start
   : "start" Id '{' StartBody '}'
   | "start" '{' StartBody '}'
```

`Id` represents a name of an instruction set that the **`start`** section denotes.

```
StartBody
   : StartBody StartAttribute ';'
   | StartAttribute ';'
```

```
StartAttribute
   : "roots" '=' '{' StartRootsBody OptionalComma '}'
   | "emulations" '=' '{' StartEmulationsBody OptionalComma '}'
   | "peepholes" '=' '{' StartPeepholesBody OptionalComma '}'
   | "bundling" '=' '{' CompoundId ',' CompoundId '}'
```

`bundling` denotes two entry points for VLIW code compression. `CompoundId`s represent two functions. The first function is used for code compression (*bundling* function). The second function is used for code decompression (*debundling* function).

```
StartRootsBody
   : StartRootsBody ',' CompoundId
   | CompoundId
```

Where `CompoundId` represents an **`element`** or **`set`**.

```
StartEmulationsBody
   : StartEmulationsBody ',' CompoundId
   | CompoundId
```

Where `CompoundId` represents an **`emulation`** or **`set`** of **`emulation`**s.

```
StartPeepholesBody
   : StartPeepholesBody ',' CompoundId
   | CompoundId
```

Where `CompoundId` represents a **`peephole`** or **`set`** of **`peephole`**s.

5.4.2 Element

Element is a basic construction for the instruction set definition. It describes either the whole instruction or its parts (e.g. operands or operation codes). Each element consists of several sections and those used for the instruction set description are introduced by the keywords **`use`**, **`assembly`** and **`binary`**. The **`use`** sections instantiate other **`element`**s and **`set`**s, and also **`event`**s. The **`assembly`** section contains a textual specification of an instruction or its parts. The **`binary`** section defines the binary representation of an instruction or its parts.

`element`s can have several properties that are assigned using `Properties` construct. The supported properties are:

- `operand(<TOOL>, String)` – The `String` argument defines how the <*TOOL*> sees the **`element`** or **`set`**. For instance, during profiling the **`set`** is not expanded. This has a huge effect on the number of instructions that are tracked by the profiler (the number is reduced). <*TOOL*> can be one of `COMPILER`, `PROFILER`, `RANDOMGEN`, or `DOCUMENTATION`.
- `ignore(<TOOL>)` – If used, the **`element`** or **`set`** are not evaluated when the <*TOOL*> is generated or used. <*TOOL*> can be one of `COMPILER`, `PROFILER`, `RANDOMGEN`, or `DOCUMENTATION`.
- `register_operand(CompoundId)` – If used, the **`element`** or **`set`** describes register operands. It implies the following limitations: **`binary`** sections of all instantiated **`element`**s and/or **`set`**s have to have the same bit width. **`return`** section has to return a compile time expression. `CompoundId` is an identifier of a register file.
- `register_class(CompoundId)` – If used, the **`element`** or **`set`** describes a register class for the C compiler generator. It knows which registers from a register file C Compiler may use when issuing assembler instructions. The `CompoundId` is a identifier of the register file. `register_class` is always a subset of `register_operand`.
- `subtarget(IdList)` – If used, the **`element`** or **`set`** is part of a subtarget. It is valid only for **`element`**s describing instructions and **`set`**s that collect such **`element`**s. Instruction(s) described in the **`element`** or **`set`** will be used only if one of the subtargets named in the property is selected in the compiler.
- `instruction(Id)` – Informs the C compiler generator that the **`element`** represents an instruction that a C compiler can use. The `Id` can be:
  - `SLOT` – The **`element`** contains instructions for a VLIW slot.
  - `SEQUENCE_PART` – The **`element`** contains instructions for a sequence.
- `assembler_alias(...)` – The specified **`element`**s or **`set`**s are treated as an assembler alias. See Pseudo Instructions on page 68 for more details.
- `compiler_alias(...)` – This property is for the compiler generator and is ignored by all the other tool generators. The compiler alias can overlap multiple instructions (i.e. **`element`**s or **`set`**s), and so the property takes arbitrary number of aliased **`element`**s or **`set`**s as arguments. This is useful for defining new instructions for the compiler generator, but not for the processor itself. Unlike of assembler aliases, semantics may be assigned. See Pseudo Instructions on page 68 for more details.

Any element can have three additional sections. The order of execution within a simulation is the same is in the following list:

- **`semantics`** – defines the semantics of the **`element`**. See Semantics Section on page 85.
- **`return`** – defines the return value when the **`element`** is used. See Return Section on page 86.

Moreover, any element can have a local resources that may be used within **`semantics`** section.

The following example shows a simple **`element`**:

Example 34: Simple element (model/share/isa/isa_operands.codal)

```
// 16bit Signed Immediate Value
element simm16
{
    signed attribute bit[16] val;

    assembly { val };
    binary { val };
    return { val; };
};
```

The next example shows a more complex **`element`**:

Example 35: Complex element (model/share/isa/isa.codal)

```
element i_3_reg_operands
```

```
{
    // Opcodes
    use opc_arithmetic_logic as opc;
    // Local register instances
    use gpreg as reg_dst, reg_src1, reg_src2;

    // The textual form is the instr. mnemonic name followed by three register operands
    assembly { opc reg_dst "," reg_src1 "," reg_src2 };
    // The binary coding follows the arithmetic and logic instruction format
    binary { opc reg_dst reg_src1 reg_src2 };
};
```

The next example shows `nop` as an assembler alias.

Example 36: Assembler alias (model/share/isa/isa.codal)

```
element i_nop_alias : assembler_alias(i_3_reg_operands)
{
    assembly { "nop" };
    binary { OPC_NOP:bit[32] };
};
```

The syntax of the `Element` construct is as follows:

```
Element
   : "element" Id Properties '{' ElementSections '}'
   | "element" Id Properties '{' ElementDeclarations ElementSections '}'
```

```
ElementDeclarations
   : ElementDeclarations ElementDeclaration ';'
   | ElementDeclaration ';'
```

```
ElementDeclaration
   : SectionUse
   | Register
   | RegisterFile
   | Signal
   | Attribute
```

```
ElementSections
   : ElementSections ElementSection
   | ElementSection
```

```
ElementSection
   : SectionIf
   | SectionSwitch
   | SectionAssembly ';'
   | SectionBinary ';'
   | SectionSemantics ';'
   | SectionReturn ';'
```

```
Properties
   : ':' PropertiesBody
   | %empty
```

```
PropertiesBody
   : PropertiesBody ',' Property
   | Property
```

```
Property
   : Id '(' PropertyArguments ')'
   | "pipeline" '(' PropertyArguments ')'
```

`Property` modifies or adds additional functionality to the **`element`** or **`set`**. `Id` is a name of a property. The syntax of the `Id` construct is described in CodAL Basics on page 5 .

```
PropertyArguments
   : PropertyArguments ',' PropertyArgument
   | PropertyArgument
```

```
PropertyArgument
   : CompoundId
   | String
```

`CompoundId` is the identifier of a resource or other part of an instruction set that is used by a property. The syntax of the `CompoundId` construct is described in chapter CodAL Basics, section Compound Id.

`String` is used as an argument to the `operand()` property. The syntax of the `String` construct is described in CodAL Basics on page 5.

5.4.2.1 Use Section

The **`use`** section contains instance declarations of other **`element`**s, **`set`**s, or **`event`**s (see Implementation on page 96 for details about **`event`**s). The **`use`** section is used in **`element`**s and **`event`**s. The declaration creates one or more instances. The instances can be used within all sections of **`element`**s and **`event`**. For instance, when an instance is used in and **`element`**'s **`binary`** section, then it means that the part of an instruction is formed by the instance.

The following example shows two forms of **`use`**. The first one is for when the name of the **`element`**, **`set`** or **`event`** being instantiated is suitable for the instance name. The second form is used when the local instance should have a different name or when multiple instances are needed.

Example 37: Use sections (model/share/isa/isa.codal)

```
element i_2_reg_operands
{
    use opc_mov_neg;
    use gpreg as reg_dst, reg_src;
```

The syntax of the `SectionUse` construct is as follows:

```
SectionUse
   : "use" SectionUseInstance
   | "use" CompoundId "as" SectionUseAsBody
```

`CompoundId` is the relative or absolute name of the referenced **`element`**, **`set`** or **`event`**. The syntax of the `CompoundId` construct is described in CodAL Basics on page 5 .

```
SectionUseAsBody
   : SectionUseAsBody ',' SectionUseInstance
   | SectionUseInstance
```

```
SectionUseInstance
   : Id
```

`Id` is the local name of an instance or the name of the referenced **`element`**, **`set`** or **`event`**. The syntax of the `Id` construct is described in CodAL Basics on page 5.

5.4.2.2 Attributes

An attribute (`Attribute`) is a numerical immediate operand (e.g. target of a jump). The attribute can be signed (two's complement format is used) or unsigned. If an attribute is used as an address specification, it should be marked as label.

The following example demonstrates two possible specifications of attributes.

Example 38: Attributes (model/share/isa/isa_operands.codal)

```
// 14bit Signed Immediate Value
element simm14
{
    unsigned attribute bit[14] val;

    assembly { val };
    binary { val };
    return { val; };
};
// 20bit Absolute Address
element abs_addr20
{
    unsigned attribute bit[20] val
    {
        label = true;
    };

    assembly { val };
    binary { val };
    return { val; };
};
```

The attribute can have a semantics action specified on the attribute value. This semantic action is used only by programming tools. Using the limited C-like expression, the user may specify how the operand value should be changed by the assembler in order to be stored in the binary coding. Semantic actions on attributes are usually used for address operands. For example, when instructions are always aligned to 4-byte boundaries, it is not necessary to store the two lowest address bits. These two bits are always zero and throwing them away saves 2 bits in the binary coding. A semantic action for such a attribute looks like this: `{ addr = addr >> 2; }`.

Example 39: Attributes semantics action (model/share/isa/isa_operands.codal)

```
// 26bit Absolute Address (shifted by 2 bits)
element abs_addr26
{
    unsigned attribute bit[26] val
    {
        label = true;
        encoding = val >> 2;
        decoding = val << 2;
    };

    assembly { val };
    binary { val };
    return { val; };
};
```

The second usage of semantic actions is to define relative addresses. The main reason for relative addresses is again the need to save bits in the binary coding without losing the ability to jump anywhere we need (even on micro-architectures with 32-bit or larger memory address spaces). A special keyword, **`current_address`**, may be used in a semantic action to specify the

current location of the instruction in the object file. For example, we may need to have an address in the binary coding to be relative to the following instruction while each instruction occupies 4 bytes.

The keyword **current_address** serves only to specify relative addresses and may only be subtracted from the absolute address (input attribute value). The addition of the current address to the absolute address would not result in any useful value.

When the result of the expression specified by a semantic action is not constant at assembly-time (as is usual for addresses), then the information is stored as a relocation to the resulting object file, and then used by the linker.

Here is another simple example of a semantic action:

Example 40: Attributes semantics action (model/share/isa/isa_operands.codal)

```
// 14bit Relative Address (shifted by 2 bits)
element rel_addr14
{
    signed attribute bit[14] val
    {
        label = true;
        // address is relative to the address of the following instruction
        encoding = val - current_address >> 2;
        decoding = ((int16)val << 2) + current_address;
    };

    assembly { val };
    binary { val };
    return { val; };
};
```

This example shows an **assembly** section for an **element** that describes the address of a jump instruction. It is a signed 14-bit number. When assembling, the assembler takes the input value and subtracts the current absolute address from it. Then, it shifts it to the right by two and stores the resulting value as the binary operand.

When disassembling, the disassembler takes the input value, a signed 14-bit number, and sign-extends it to 16-bit. Then, it shifts it to the left by two, adds the current absolute address, and prints the resulting value. Notice that without the sign-extend of `val`, the `val` would remain a 14-bit number; hence, the two highest bits would be lost after shifting to the left.

When attribute has a fixed base all the time, it may be specified using **base** attribute. Supported bases are `DEC`, `HEX`, `BIN`, and `OCT`.

The syntax of the `Attribute` construct is as follows:

```
Attribute
   : AttributeSpecifier "attribute" AttributeBit Id AttributeBodyEncap
```

`Id` is an identifier of a **set** member. The syntax of the `Id` construct is described in CodAL Basics on page 5 .

```
AttributeSpecifier
   : "signed"
   | "unsigned"
```

`AttributeSpecifier` denotes the signedness of an attribute.

```
AttributeBit
   : "bit" '[' CompileExpression ']'
   | %empty
```

`AttributeBit` specifies the bit-width of an attribute. If it is omitted, the bit-width is taken from the **binary** section.

```
AttributeBodyEncap
   : '{' AttributeBody '}'
   | %empty
```

```
AttributeBody
   : AttributeBody AttributeAttribute ';'
   | AttributeAttribute ';'
```

```
AttributeAttribute
   : "encoding" '=' AnsiExpression
   | "decoding" '=' AnsiExpression
   | "base" '=' '{' AttributeBaseBody OptionalComma '}'
   | "symbol" '=' CompileExpression
   | "label" '=' CompileExpression
   | "illegal" '=' '{' AttributeIllegalBody '}'
```

```
AttributeBaseBody
   : AttributeBaseBody ',' AttributeBaseId
   | AttributeBaseId
```

```
AttributeBaseId
   : Id
   | "binary"
```

`Id` defines a base of an immediate. It can be `decimal`, `hexadecimal`, `octal`, `binary`, or `user_<text>`. `user_<text>` is reserved for a user-defined base, where `<text>` is an identifier of the user-defined base (e.g. `my_dec`).

```
AttributeIllegalBody
   : AttributeIllegalBody ',' CompileExpression
   | CompileExpression
   | AttributeIllegalBody ',' CompileExpression ".." CompileExpression
   | CompileExpression ".." CompileExpression
```

`AttributeIllegalBody` defines one or more illegal values of the attribute. The illegal values can be used as opcodes in other instructions.

An attribute may have several attributes.

5.4.2.3 Assembly Section

An **`assembly`** section defines a textual representation of an **`element`**. The **`assembly`** section consists of instances, text constants and attributes. Every **`assembly`** section must contain at least one instance, text constant, or attribute.

In the case of using multiple instances, text constants or attributes, these can be separated by a space or by the ~ (concatenation) operator. A space means that in the actual assembly code, the two elements may be separated by an arbitrary number of spaces and tabulators, including zero. The concatenation operator specifies that there must not be any white-space between the two adjacent elements.

A special feature of the CodAL **`assembly`** section is the `STRINGIZE` function. It takes one parameter and adds quotes to form a string literal. For instance the expression `STRINGIZE(r2)` is the same as the string `"r2"`. The purpose is to use this function combined with the C preprocessor. For example, this function can be used while generating a set of registers that differ only in their numbering.

5.4.2.3.1 Syntax

The syntax of the `SectionAssembly` construct is as follows:

```
SectionAssembly
   : "assembly" '{' SectionAssemblyBody '}'
```

```
SectionAssemblyBody
   : SectionAssemblyBody SectionAssemblyElement
   | SectionAssemblyElement
```

```
SectionAssemblyElement
   : Id
   | String
   | '~'
   | "STRINGIZE" '(' Id ')'
```

5.4.2.3.2 Instances

An instance is an identifier (`Id`) that is declared in a **`use`** section. The instance can be used more than once.

5.4.2.3.3 Text Constants

The text constant (`String`) is a string encapsulated in quotes, e.g. `"nop"`.

5.4.2.3.4 Attributes

An attribute is an identifier (`Id`) that is declared using `Attribute` construct.

The attributes are described in Attributes on page 62 in greater detail.

5.4.2.4 Binary Section

The **`binary`** section defines the binary representation of an instruction (or their parts, such as operands). The binary section can consists of instances, binary constants or attributes. Every binary section must contain at least one instance, binary constant, or attribute. In the case of using multiple instances, binary constants or attributes, these can be separated by a space.

5.4.2.4.1 Syntax

The syntax of the `SectionBinary` is as follows:

```
SectionBinary
   : "binary" '{' SectionBinaryBody '}'
```

```
SectionBinaryBody
   : SectionBinaryBody SectionBinaryElement
   | SectionBinaryElement
```

```
SectionBinaryElement
   : CompileExpression
   | CompileExpression ':' "bit" '[' CompileExpression ']'
   | "bit" '[' CompileExpression ']'
```

This formal syntax description is not very helpful for understanding the meaning of the items to be found in a binary section since, syntactically speaking, they are all described by the `CompileExpression` construct. This is used to describe instances, binary constants and attributes, as will now be explained.

5.4.2.4.2 Instance

Instance is an identifier that is declared in a **use** section. The instance can be used more than once.

The instance can be split. This feature is particular useful for an optimal binary encoding of instructions. An instance can be split only if it has a uniform bit-width. For example, if an instance represents a **set** that has two members where one is a 16-bit instruction and the second one is a 32-bit instruction, then the split is not allowed. Moreover, the binary coding must be fully split, i.e. no part of the instance can be omitted.

The following example shows how the split is performed: `reg_dst` and `uimm16` are split. `reg_dst` is split into two parts where the first has only one bit and the second has three bits. `uimm16` is split into three parts and they are shuffled within the instruction binary coding.

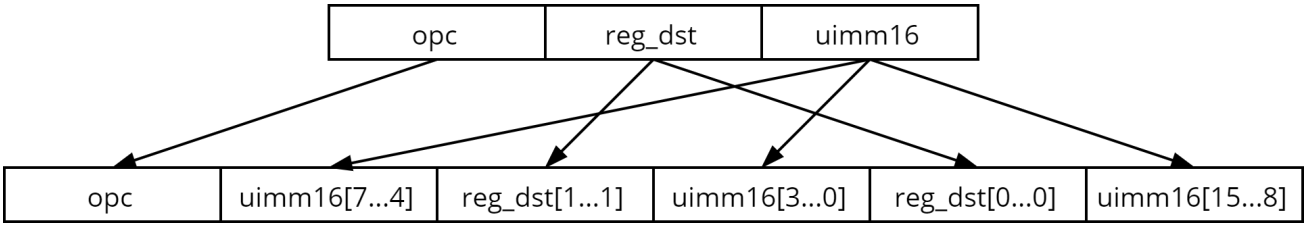
Example 41: Instance split (model/share/isa/isa.codal)

```
element i_lui
{
    use opc_lui as opc;
    use gpreg as reg_dst;
    use uimm16;

    assembly { opc reg_dst "," uimm16 };
    binary { opc uimm16[7..4] reg_dst[1..1] uimm16[3..0] reg_dst[0..0] uimm16[15..8] };
};
```

The previous instance split is illustrated in Figure 7.

Figure 7: Instance split



5.4.2.4.3 Binary Constant

A binary constant is used for opcodes or other fixed parts of the instruction encoding. There are several ways to write a constant within a binary section. You may write a single constant in decimal, hex, octal or binary format. The default bit-width of such a constant is 32-bit, though you may change this bit-width using the **bit** construct.

An enum item can be used within a binary section also. In this case, the bit-width is taken from the enum definition, so the **bit** construct is optional.

The following example shows all possible variants in one element. The number `10` is expanded to 32 bits. The number `12` is restricted to be on four bits. A binary constant can be also an expression `(1+1)` (we have seen that the syntax construct `CompileExpression` is used to represent the binary constant). The expression must be followed by a **bit** construct. The bit-width of the enum identifier `OPC_ENUM` is five, so there is no need to specify it explicitly, but you can use it when necessary (e.g. as shown for `OPC_ENUM_2` in the **binary** section of the **element**).

Example 42: Binary constants (model/share/isa/isa.codal)

```
enum : uint5
{
```

```
    OPC_ENUM  = 0x10,
    OPC_ENUM_2 = 0x2
};

element i_abc
{
    assembly { "abc" };
    binary { 10 12:bit[4] (1+1):bit[1+1] OPC_ENUM OPC_ENUM_2:bit[2]  };
};
```

5.4.2.4.4 Attribute

An attribute is an identifier (`Id`) that is declared using `Attribute` construct. When `Attribute` construct does not have information about bit-width of the attribute, then this information has to be written in the **`binary`** section. This is useful when conditional sections are used (see Conditional Sections on page 67).

The attributes are described in Attributes on page 62 in greater detail.

The **`binary`** section may contain unnamed attributes (they do not have to be included in the corresponding **`assembly`** section). Such attributes are used to represent the *don't-care* bits.

Example 43: Unnamed attribute (model/share/isa/isa.codal)

```
element i_xyz
{
    assembly { "xyz" };
    binary { OPC_XYZ bit[10]  };
};
```

5.4.2.5 Conditional Sections

Sections within an element may be conditional. This is needed when two different instructions have the same textual form but different encoding. For example, `"mov r1, imm"`. Based on the `imm` the move can be encoded on 16 bits (small `imm`) or on 32 bits (big `imm`). The example below shows such a case.

`OPC_MOV` is the opcode for a "move" instruction and it is declared in a global binary section. The global binary section is optional and there can be only one such section in an **`element`**. Based on the value of `attr`, either a small or a big version of `MOV` is used. The final **`binary`** section is then created by concatenating the **`binary`** sections. The order of the concatenation is *from the general case to the specific case*. The final **`binary`** section for the small variant is `{ OPC_MOV MOV_SMALL reg_dst attr:bit[8] }`.

Example 44: Conditional sections (model/share/isa/isa.codal)

```
element i_small_big_move
{
    use gpreg as reg_dst;
    unsigned attribute attr;

    assembly { "mov" reg_dst "," attr };
    binary { OPC_MOV };
    if (attr <= 0xff)
    {
        binary { MOV_SMALL reg_dst attr:bit[8] };
    }
    else
    {
        binary { MOV_BIG reg_dst attr:bit[16] };
    }
};
```

The body of the condition statement can contain constants or attributes with standard operators without side effects. Each **if** part must have an **else** part, and each **switch** must have a **default** branch. Each branch within the **switch** must have a **break** statement. The **binary** section must be included in all branches and it must be unique for each branch. Only the following sections can be conditional: **binary**, **semantics** and **return**. If the section is declared outside of the **if** statement body, then it is used in all branches of the binary encoding (in this way two instructions can have the same **semantics** section, for example).

The syntax of the `SectionIf` construct is as follows:

```
SectionIf
   : "if" '(' AnsiExpression ')' SectionIfBody
```

```
SectionIfBody
   : SectionIfSwitchCompoundSection "else" SectionIfSwitchCompoundSection
```

```
SectionIfSwitchCompoundSection
   : ElementSection
   | '{' ElementSections '}'
   | '{' ElementDeclarations ElementSections '}'
```

The syntax of `SectionSwitch` construct is as follows:

```
SectionSwitch
   : "switch" '(' AnsiExpression ')' '{' SectionSwitchBody '}'
```

```
SectionSwitchBody
   : SectionSwitchBody SectionSwitchBodyElememt "break" ';'
   | SectionSwitchBodyElememt "break" ';'
```

```
SectionSwitchBodyElememt
   : "case" AnsiConstantExpression ':' SectionIfSwitchCompoundSection
   | "default" ':' SectionIfSwitchCompoundSection
```

```
SectionIfSwitchCompoundSection
   : ElementSection
   | '{' ElementSections '}'
   | '{' ElementDeclarations ElementSections '}'
```

5.4.2.6 Pseudo Instructions

Besides real instructions, a processor description may contain so-called *emulated instructions* (also called pseudo-instructions or instruction aliases). Two types are supported: assembler and compiler aliases, using the **assembler_alias** and **compiler_alias** attributes respectively.

The aliases can be used to fill holes in the instruction set. For example, the instruction `nop` may be emulated as `mov r0, r0`.

Another use is a simplification of the syntax generated by the compiler for better readability of assembly code. Let's say that the ISA defines a no-operation instruction alias and by the means of an element with both **compiler_alias** and **assembler_alias** attributes, the user can say to the compiler to generate instruction with syntax `nop` instead of the instruction that is aliased such as `mov r0, r0`.

5.4.2.6.1 Assembler Alias

An assembler alias is the instruction or its part that shares its binary representation (and functionality) with another instruction or its part, but has a different assembly notation. In the `nop` example given above, whenever we use `nop` in our

assembly code, the processor perform a move. Only the assembler can use these aliases. No other tools use aliases.

More information on assembler aliases can be found in the chapter on Compiler Generation in the *Codasip Studio User Guide*.

5.4.2.6.2 Compiler Alias

Specifying an compiler alias is useful to define new instructions for the compiler generator. The user has complete freedom in specifying syntax and semantics. The compiler aliases can overlap one or more instructions. This way can be for example a combinations of instructions such as for branch-if-equal operation defined.

More information on compiler aliases can be found in the chapter on Compiler Generation in the *Codasip Studio User Guide*.

5.4.3 Sets

A **set** groups related **element**s and/or **set**s so that they can be declared inside **element**s and **event**s using a single reference (with the **use** keyword).

Each **set** can have several properties that are assigned using the `Properties` construct. The supported properties are:

- `operand(<TOOL>, String)` – The `String` argument defines how the <*TOOL*> sees the **element** or **set**. For instance, during profiling the **set** is not expanded. This has a huge effect on the number of instructions that are tracked by the profiler (the number is reduced). <*TOOL*> can be one of `COMPILER`, `PROFILER`, `RANDOMGEN`, or `DOCUMENTATION`.
- `ignore(<TOOL>)` – If used, the **element** or **set** are not evaluated when the <*TOOL*> is generated or used. <*TOOL*> can be one of `COMPILER`, `PROFILER`, `RANDOMGEN`, or `DOCUMENTATION`.
- `register_operand(CompoundId)` – If used, the **element** or **set** describes register operands. It implies the following limitations: **binary** sections of all instantiated **element**s and/or **set**s have to have the same bit width. **return** section has to return a compile time expression. `CompoundId` is an identifier of a register file.
- `register_class(CompoundId)` – If used, the **element** or **set** describes a register class for the C compiler generator. It knows which registers from a register file C Compiler may use when issuing assembler instructions. The `CompoundId` is a identifier of the register file. `register_class` is always a subset of `register_operand`.
- `subtarget(IdList)` – If used, the **element** or **set** is part of a subtarget. It is valid only for **element**s describing instructions and **set**s that collect such **element**s. Instruction(s) described in the **element** or **set** will be used only if one of the subtargets named in the property is selected in the compiler.

The following examples illustrate the basic principles of the `Set` construct.

The first example shows a **set**, `gpreg`, that represents a register operand. It has five members. The first four members represent register operands `r0` – `r3`. The fifth member is an alias on `r0` (e.g. `zero`). The alias adds another textual form for the same register operand. In other words, the first item of the register file can be accessed using `"r0"` or `"zero"` at the assembly level. Note that in the case of profiling or disassembling, `"r0"` is always used.

The set has two properties assigned. The first one says that the **set** represents a register class that uses the `rf_gp32`. The second property is used during profiling. The profiler does not expand the **set** and uses just the `"reg"` string when the **set** is used during instruction decoding.

Example 45: Register operands (model/share/isa/isa_operands.codal)

```
// Register operand
set gpreg : register_class(rf_gp32), operand(PROFILER, "reg");
// Members
set gpreg += reg0, reg1, reg2, reg3;
set gpreg += reg0_alias;

// basic elements
element reg0
```

```
{
    assembly { "r0" };
    binary { 0:bit[2] };
    return { 0; };
};
element reg1
{
    assembly { "r0" };
    binary { 1:bit[2] };
    return { 1; };
};
element reg2
{
    assembly { "r2" };
    binary { 2:bit[2] };
    return { 2; };
};
element reg3
{
    assembly { "r3" };
    binary { 3:bit[2] };
    return { 3; };
};

// alias
element reg0_alias : assembler_alias(reg0)
{
    assembly { "zero" };
    binary { 0:bit[2] };
};
```

The next example shows the `opc_mov_neg` **`set`** that contains two opcodes:

Example 46: Defining a set of instructions (model/share/isa/isa.codal)

```
set opc_mov_neg = opc_mov,
                  opc_neg;
```

The last example shows a top level **`set`** called `isa`. It contains all the instructions within the instruction set of a processor.

Example 47: Top level set (model/share/isa/isa.codal)

```
set isa += i_2_reg_operands,
    i_3_reg_operands,
    i_lui,
    i_ori,
    i_movsi,
    i_load_store,
    i_jump_call_imm,
    i_jump_call_reg,
    i_setcmp,
    i_jump_cond,
    i_nop_alias;
```

The syntax of the `Set` construct is as follows:

```
Set
   : "set" Id SetAssign SetBody OptionalComma
   | "set" Id Properties
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
SetAssign
   : '='
```



```
   | "+="
```

These two operators (which are equivalent) are used for adding members incrementally to a **set**. For readability, the += operator is recommended.

```
SetBody
   : SetBody ',' SetBodyElement
   | SetBodyElement
```

```
SetBodyElement
   : CompoundId
```

CompoundId is an identifier of a **set** member. The syntax of the CompoundId construct is described in CodAL Basics on page 5.

```
Properties
   : ':' PropertiesBody
   | %empty
```

```
PropertiesBody
   : PropertiesBody ',' Property
   | Property
```

```
Property
   : Id '(' PropertyArguments ')'
   | "pipeline" '(' PropertyArguments ')'
```

Property modifies or adds additional functionality to the **set**. Id is a name of a property. The syntax of the Id construct is described in CodAL Basics on page 5 .

```
PropertyArguments
   : PropertyArguments ',' PropertyArgument
   | PropertyArgument
```

```
PropertyArgument
   : CompoundId
   | String
```

CompoundId is an identifier of a resource or other part of an instruction set that is used by a property. The syntax of the CompoundId construct is described in chapter CodAL Basics, section Compound Id.

String is used as an argument to the operand() property. The syntax of the String construct is described in CodAL Basics on page 5.

5.4.4 Subtarget

The Subtarget construct is used to specify groups of instructions that can be selectively turned on or off in the C compiler.

Each subtarget must have an Id that is used in the subtarget property to assign **element**s or **set**s to it. Subtargets must also specify a command line argument (argument) for the C compiler. This argument is used to enable or disable the subtarget when compiling applications.

Optional attributes are:

- `depends` – defines a list of subtargets on which the subtarget depends. All dependencies are automatically activated when the subtarget is enabled.
- `enabled_by_default` – if `True`, the subtarget is enabled in the compiler by default, but can be disabled. If `False`, the subtarget must be activated explicitly by its command line argument.

The syntax of the `Subtarget` construct is as follows:

```
Subtarget
   : "subtarget" Id '{' SubtargetBody '}'

SubtargetBody
   : SubtargetBody SubtargetAttribute ';'
   | SubtargetAttribute ';'

SubtargetAttribute
   : "argument" '=' String
   | "depends" '=' '{' IdList '}'
   | "enabled_by_default" '=' CompileExpression
```

For more information, see *Codasip Studio Technical Reference Manual*, chapter Compiler Generator, section Subtargets.

5.4.5 Emulations

Emulations are used only by C/C++ generator. The purpose of the emulations is to map instructions to LLVM operation when there is not 1:1 mapping between instructions and LLVM operations.

Each emulation consists of three parts.

Declarations provide information about used elements and attributes. Used elements in most cases are root elements that forms some instruction. Attributes should be used mainly for declaration of new immediate operands that are larger than immediate operands contained in used instructions as in the following example. Size has to be given to emulation attribute.

Instructions section is a ordered list of instructions, which have same effect as emulation, when they are executed in that order. Each instruction, one instruction is one line in example, should have unique identifier. Instance specializations, they are described as assignments, are within each instruction. Left side stands for instance, that is specialized, and right side is specialization value. For example attributes can be specialized on number or linked with emulation attribute. Sets with instructions operation codes could be specialized on one contained operation code. Register classes could be specialized on one contained register or linked with output register from one of preceding instructions.

Semantics section describes behavior of emulation just like semantics sections in other CodAL constructs. However, there is some difference. Links on elements or sets of used instructions can be used. Emulation from following example has one output register and one input immediate operand. Output register is linked into semantics from second instruction and that linked him from first instruction. Emulation input is linked to the both instructions. Attribute could be linked partially to multiple instructions immediate operands.

The following example shows an emulation for the 32-bit constant load using two instructions. One instruction is used for a loading of an upper 16-bits of the constant and the second one is used for a loading the lower part of the constant.

Example 48: Emulation (model/share/isa/isa.codal)

```
// Load upper 16bit to a register
element i_lui_16b
{
    use gpreg as reg_dst;
    unsigned attribute bit[16] val;

    assembly { "lui" reg_dst "," val };
```

```
    binary { 0b0000:bit[4] reg_dst val };

    // ...
};
// Or of lower 16bit to a register
element i_ori_16b
{
    use gpreg as reg_dst;
    unsigned attribute bit[16] val;

    assembly { "ori" reg_dst "," val };
    binary { 0b0001:bit[4] reg_dst val };

    // ...
};
// emulation the uses lui and ori to load 32bit constant to a register
emulation e_li
{
    use i_lui_16b;
    use i_ori_16b;

    // 32bit value of a constant
    unsigned attribute bit[32] attr;

    instructions
    {
        i_lui_16b(val = attr[31..16] );
        i_ori_16b(reg_dst = i_lui_16b(reg_dst), val = attr[15..0] );
    };

    semantics
    {
        rf_gp32[i_ori_16b.reg_dst] = attr;
    };
};
```

The syntax of the `Emulation` construct is as follows:

```
Emulation
   : "emulation" Id Properties '{' EmulationDeclarations EmulationSections '}'
   | "emulation" Id Properties '{' EmulationSections '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
Properties
   : ':' PropertiesBody
   | %empty
```

```
PropertiesBody
   : PropertiesBody ',' Property
   | Property
```

```
Property
   : Id '(' PropertyArguments ')'
   | "pipeline" '(' PropertyArguments ')'
```

`Property` modifies or adds or limits the functionality to the **`emulation`**. `Id` is a name of a property. The syntax of the `Id` construct is described in CodAL Basics on page 5 .

```
EmulationDeclaration
   : SectionUse
   | Attribute
```

```
EmulationSections
   : EmulationSections EmulationSection
```

```
   | EmulationSection
```

```
EmulationSection
   : SectionInstructions
   | SectionIfInstructions
   | SectionSemantics ';'
```

```
SectionIfInstructions
   : "if" '(' AnsiExpression ')' SectionIfInstructionsBody
```

```
SectionIfInstructionsBody
   : SectionInstructions
   | '{' SectionInstructions '}'
```

`SectionSemantics` describes semantics of an emulation (see Semantics Section on page 85 for more details).

```
SectionInstructions
   : "instructions" AnsiCompoundStatement ';'
```

`SectionInstructions` describes which instructions are used by an emulation . It also describes a specializations of the instructions or connections among the instructions. This section may be conditional and placed in `SectionIfInstruction`.

5.4.6 Peephole

`peephole` patterns are used only by C/C++ compiler. The purpose of the **`peephole`** patterns is to replace several machine instructions with smaller amount of different machine instructions that do the same, when machine code generator is not able to pick fewer instructions himself.

Each peephole pattern consists of four parts:

- Declaration provides information about instances of elements representing instructions. These instances describe instructions that should be replaced and instructions that should replace them.
- The **`pattern`** section contains ordered list of instructions, that optimizer tries to find in a program. Each instruction should have a unique identifier and should be on a separate line. An instruction is represented by its identifier declared in the declare section. An operand within the instances of instructions can be specialized. Specialization is described as an assignment after the instruction identifier and it is enclosed in brackets (similarly to emulations). It is used to link attributes of an instruction with attributes of other instructions to represent data dependencies.
- The **`replace`** section is similar to the **`pattern`** section. It is an ordered list of instructions, that should replace the original pattern. This list consists of instances of elements, that can be specialized. In this case, every instance of every instruction, that represents an operand, must be specialized. In case of an immediate operand, specialization value can be constant. In case of a register operand, the specialization value can be a link to some operand or result of a pattern instruction, or a prior replace instruction. Operands representing a constant immediate value can be specialized directly by the value. This section can be enclosed in if statement which means that found instructions will be replaced only if specified conditions are met.
- The last section is the **`mapping`**. It is an optional section. It specifies results of the instructions of the new pattern and their mapping to the old results in the old pattern. This section is needed for an automatic casting when type of the original result is different from the new result.

Example 49: Peephole pattern

```
element i_load32
{
	use reg32_any as reg32_dst;
	use reg64_any as reg64_base;
	use simm12 as offset;

	assembly { "load" reg32_dst "," offset "(" reg64_base ")" };

	// ...
};

element i_zero_ext
{
	use reg32_any as reg32_src;
	use reg64_any as reg64_dst;

	assembly { "zex" reg64_dst "," reg32_src };

	// ...
};

element i_load64
{
	use reg64_any as reg64_dst;
	use reg64_any as reg64_base;
	use simm12 as offset;

	assembly { "load" reg64_dst "," offset "(" reg64_base ")" };

	// ...
};

// pattern that replaces load and extension of 32-bit value to 64-bit value
// with load of 64-bit value from the same address
peephole p1
{
	use i_load32;
	use i_zero_ext;
	use i_load64;

	pattern
	{
		i_load32();
		i_zero_ext(
			reg32_src = i_load32.reg32_dst
		) ;
	};

	// this pattern will only be replaced if value of address offset is less than 50
	// and base register for load is stack pointer
	if (i_load32.reg64_base == GPR_SP && i_load32.offset.val < 50)
	{
		replace
		{
			// after replacing the pattern, the resulting instruction i_load64 will use
			i_load64(
				// the original destination register of i_zero_ext as a new destination
				reg64_dst = i_zero_ext.reg64_dst,
				// for the base and the offset, we will use the original load arguments
				reg64_base =  i_load32.reg64_base,
				offset.val = i_load32.offset.val
			);
		};
	}

	// if the intermediate result of the pattern
	// i_load32. reg32_dst is needed, the patter can be still replaced and instead
	// of the 32-bit result of the original load
	// can be a subregister (lowest 32 bits) of the new result register used
	mapping
	{
		i_load64.reg64_dst : i_load32.reg32_dst
```

```
      };
};
```

The syntax of the `Peephole` construct is as follows:

```
Peephole
   : "peephole" Id '{' PeepholeDeclarations PeepholeSections '}'
   | "peephole" Id '{' PeepholeSections '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in CodAL Basics on page 5.

```
PeepholeDeclarations
   : PeepholeDeclarations PeepholeDeclaration ';'
   | PeepholeDeclaration ';'
```

```
PeepholeDeclaration
   : SectionUse
```

```
PeepholeSections
   : PeepholeSections PeepholeSection
   | PeepholeSection
```

```
PeepholeSection
   : SectionPattern ';'
   | SectionReplace
   | SectionIfReplace
   | SectionMapping ';'
```

```
SectionPattern
   : "pattern" AnsiCompoundStatement
```

`SectionPattern` describes an instruction pattern, usually consisting of more than one instruction, that is replaced by another instruction pattern.

```
SectionReplace
   : "replace" AnsiCompoundStatement ';'
```

`SectionReplace` describes an instruction pattern, usually consisting of only one instruction, that is used for replacement when a valid pattern is found.

```
SectionIfReplace
   : "if" '(' AnsiExpression ')' SectionIfReplaceBody
```

`SectionIfReplace` allows using a `SectionReplaceBody` section only when `AnsiExpression` is evaluated as true. It also allows having more replace patterns for a single source pattern.

```
SectionMapping
   : "mapping" '{' MappingStatements '}'
```

`SectionMapping` is optional and it specifies operands from the instructions of the original pattern and their mapping to the new operands in the new pattern.

```
MappingStatements
   : MappingStatements ',' CompoundId ':' CompoundId OptionalComma
   | CompoundId ':' CompoundId OptionalComma
```

`MappingStatements` defines a single operand mapping where `CompoundId`s define a source operand and a destination operand.

5.4.7 Bundling/Debundling

Support of instruction bundling is achieved by implementing a *bundling* function and a *debundling* function.

The bundling function is used by assembler. With such function, the assembler tracks bit movements of source instructions and computes valid relocations for them. Interface of bundling function consists of:

- bundle return type,
- function name and,
- input arguments, namely a structure **`bundle_state_t`** followed by arguments with encoded instructions. A number of these arguments must match with a number of bundle slots. Bit width of each argument must match maximum bit width of instructions in a given slot.

The following example shows a declaration of a bundle function for 4-slot VLIW processor. Arguments `i0`, `i1`, `i2`, and `i3` represent encoded instructions of each slot.

Example 50: Bundle Function

```
bundle128 fbundle(const bundle_state_t state,
    const bundle32 i0,
    const bundle32 i1,
    const bundle32 i2,
    const bundle32 i3)
{
    ...
}
```

The bundle type (for example **`bundle32`**) is a special structure type, which consists of two members. The **`value`** member stores encoded instruction at first and then in result it could store a bundled instruction and some extra bits, for instance a stop bit. The **`size`** member tells a bit width which is used by the bundle type. The input encoded instruction could have less bits than the maximum length of the instruction in a given bundle. It also tells to the assembler how many bits of a resulting bundle type are really used.

The following example shows the structure for the 32-bit instruction. **`bundling_int32`** works similarly to **`uint32`**, but do not support all operations, see a type documentation for details.

Example 51: Structure for the 32-bit bundle

```
struct bundle32
{
    bundling_int32 value;
    uint32 size;
};
```

Structure **`bundle_state_t`** stores the assembler state. It has member **`is_align_end`**. Semantics of this member is that the next bundle is a target of a jump instruction and this information can be used to align the actual compressed bundle as wanted. So, the target of the jump instruction starts on the aligned address. **`current_address`**, the second member of the structure stores the current address of the currently crafted section in the assembler.

Example 52: Bundle state

```
struct bundle_state_t
{
    bool is_align_end;
    uint64 current_address;
};
```

The debundling function is used by the disassembler. Interface of the debundling function consists of

- debundle return type,
- function name and,
- one input argument. Input argument's type is uint and its size is maximum size of bits, which must be read for the successful decoding of VLIW bundle.

When an alignment is added after a bundle in the bundling function, then there must be a space for this alignment. The result type must have a length as the maximum size of all instruction slots.

Example 53: Debundle function

```
debundle128 fdebundle(const uint128)
{
    ...
}
```

The debundle type (for example **`debundle128`**) is a special structure type, which consists of two members. The **`value`** member stores a debundled bits. It is always assumed by the disassembler that all bits in the resulting debundle type are valid and usable. The **`input_size`** member tells to the disassembler how many bits are really read from the input argument.

Example 54: Structure for the 128bit debundle

```
struct debundle128
{
    uint128 value;
    uint32 input_size;
};
```

5.4.8 Settings

Settings allows to set different attributes for tools in SDK. Namely, there is a section for assembler, C/C++ compiler, simulator, debugger, WFI and architecture. The syntax is the same for all kinds of sections. The `SettingsCompound` is a top level rule that encapsulates all settings. If the processor model contains more than one `SettingsCompound` construct then they are merged.

```
SettingsCompound
   : "settings" '{' SettingsCompoundBody '}'
```

```
SettingsCompoundBody
   : SettingsCompoundBody Settings ';'
   | Settings ';'
```

The `Setting` construct describes settings for SDK tools.

```
Settings
   : SettingsSpecifier '{' SettingsBody '}'
```

```
SettingsSpecifier
   : "compiler"
   | "debugger"
   | "simulator"
   | "assembler"
   | "wfi"
   | "architecture"
```

The `SettingSpecifier` construct specifies an SDK tool. There are several options:

- **`compiler`** – settings for C/C++ compiler,
- **`debugger`** – settings for on-chip and software debugger,
- **`simulator`** – settings for simulator,
- **`assembler`** – settings for assembler,
- **`wfi`** – settings for WFI (wait-for-interrupt) mode,
- **`architecture`** – settings for the processor architecture.

```
SettingsBody
   : SettingsBody Setting ';'
   | Setting ';'
```

The `Settings` construct consist of `Setting` construct that describes one singe setting/option for an SDK tool.

```
Setting
   : Id '=' SettingElement
   | Id SetAssign '{' SettingBody OptionalComma '}'
```

The `Id` specifies attribute/option of an SDK tool. For instance, a register that holds the return address of a function.

```
SettingBody
   : SettingBody ',' SettingElement
   | SettingElement
```

```
SettingElement
   : AnsiConditionalExpression
   | AnsiTypeSpecifier
```

The `SettingElement` construct is a value or values assigned to `Id` (attribute/option). For instance, a particular register in a register file that holds a return address of a function.

The following example shows some compiler settings.

Example 55: Settings Example

```
settings
{
  compiler
  {
    function_result = { rf_gpr[2], rf_gpr[3], rf_gpr[4], rf_gpr[5] };

    // short version
    stack_pointer = rf_gpr[SP];
    // or
    stack_pointer = {rf_gpr[SP], as_all};

    base_pointer = {rf_gpr[BP], as_all};
  };
};
```

The following subsections describe each `SettingSpecifier` variant. In other words, the following sections describe attributes/options that you can set for SDK tools.

5.4.8.1 C/C++ Compiler Settings

The compiler setting section allows the user to specify the behaviour of the C/C++ compiler generated from a CodAL model. This section is mandatory when the C/C++ compiler is generated.

Several of these settings may contain arrays of resources as their values. In such cases, the array indexing is omitted and the array resources are written in the same way as the individual resources. The following settings allow the resource arrays to be used as values: **`stack_pointer`**, **`base_pointer`**, **`function_params`**, **`function_result`**, **`callee_saved`** and **`callee_saved_leaf`**. For more information about the arrays, see section 5.3 Arrays.

There are several options in the compiler settings:

Numeric Attributes

- **`interrupt_handler_alignment`** – a number that has to be a power of 2. It denotes the minimum required interrupt handler alignment. Allowed only once in the model.
- **`pointer_size`** – a number. It denotes a pointer size. Currently, it have to be one of 16, 32, or 64. Allowed only once in the model.
- **`function_alignment`** – a number. It denotes a required function alignment. Allowed only once in the model.
- **`stack_alignment`** – a number, optionally with an address space. It denotes a required stack alignment. Once per address space.
- **`max_data_address_bits`** – a number. It tells the compiler that global data addresses will be max *X* bits wide. Allowed only once in the model.
- **`max_program_address_bits`** – a number. It tells the compiler that program addresses will be max *X* bits wide. Allowed only once in the model.
- **`bundle_align_jump_target`** – a boolean. Allowed only once in the model.
- **`code_align_fill_value`** – a number in the range 0..255 decimal (or equivalent in other numeric bases). Specifies the value inserted in the program when aligning instructions or blocks of instructions. Default value is 0. Allowed only once in the model.

Register Class Attributes

- **`pointer_register_class`** – an element or set with the **`register_class`** definition. Register class that will be used for pointers. Allowed only once in the model.
- **`representative_register_class`** – a pair, CodAL data type and an element or set with the **`register_class`** definition. Register class that will be used for certain type by default. Allowed only once per type.

Data Type Attributes

- **`type_space_mapping`** – CodAL data type, optionally with an address space). Maps the CodAL data type to the address space. Allowed only once per type.
- **`stack_register_space_mapping`** – an architectural register file or an alias to it, optionally with an address space. Map registers from the register file to the address space.
- **`legal_types`** – strings. Specifies which data types will be legal. A legal data type is one of the native machine types and the compiler requires a certain list of operations over this type to work correctly. Can be used multiple times in the model. The resulting value of the attribute is then a compound of the individual values.
- **`datalayout`** – a string. The string is the same as the standard LLVM datalayout (see LangRef.html).
- **`boolean_int_contents`** – a value, which determines how boolean true is represented in integer variable. False is always zero.

- True is 0b1*1 when set on value **zero_or_negative_one**, otherwise 0b0*1 (**zero_or_one** is default).
- **boolean_float_contents** – a value, which determines how boolean true is represented in float variable. See **boolean_int_contents** for more information.
- **boolean_vector_contents** – a value, which determines how boolean true is represented in vector variable. See **boolean_int_contents** for more information.

Register Attributes

- **stack_pointer** – an architectural register or a certain register form an architectural register file or an alias to them, optionally with an address space. Stack pointer register. Allowed only once per the address space.
- **base_pointer** – an architectural register or a certain register form an architectural register file or an alias to them, optionally with an address space. Base pointer register. Allowed only once per the address space.
- **return_address** – an architectural register or a certain register form an architectural register file or an alias to them. Allowed only once in the model.
- **carry_flag** – a 1-bit register. Carry status flag register. Allowed only once in the model.
- **overflow_flag** – a 1-bit register. Overflow status flag register. Allowed only once in the model.
- **sign_flag** – a 1-bit register. Sign status flag register. Allowed only once in the model.
- **zero_flag** – a 1-bit register. Zero status flag register. Allowed only once in the model.
- **callee_saved** – a list of architectural registers or a certain registers form an architectural register file or an aliases. Callee-saved registers. Can be used multiple times in the model. The resulting value of the attribute is then a compound of the individual values.
- **callee_saved_leaf** – a list of architectural registers or a certain registers form an architectural register file or an aliases. Callee-saved registers for leaf functions. Can be used multiple times in the model. The resulting value of the attribute is then a compound of the individual values.
- **function_result** – a list of architectural registers or a certain registers form an architectural register file or an aliases. Registers used to return values from a function. Can be used multiple times in the model. The resulting value of the attribute is then a compound of the individual values.
- **function_params** – a list of architectural registers or a certain registers form an architectural register file or an aliases. Registers used to pass values to a function. Can be used multiple times in the model. The resulting value of the attribute is then a compound of the individual values.
- **unused_registers** – a list of architectural registers or a certain registers form an architectural register file or an aliases. When a register appears in unused registers, such register cannot be used anywhere else. Can be used multiple times in the model. The resulting value of the attribute is then a compound of the individual values.
- **reg_alloc_order** – a logical register class, a string condition and list of architectural registers accessible from given logical register class. Specifies custom register allocation order of registers of given logical register class if given condition is met.
- **shadow_call_stack_pointer** – an architectural register, a certain register form an architectural register file, or an alias to them. Register used as Shadow Call Stack pointer. Needs to be properly initialized in the startup code. Allowed only once in the model.
- **alternate_return_address** – an architectural register, a certain register from an architectural register file, or an alias to them. Register used to hold return address value when calling functions for save/restore of registers in function prologue/epilogue. Used when compiled with argument *-msave-restore*. When not specified the register is selected automatically and printed to `ABI.txt`.
- **reg_cost_per_use** – an architectural register, a certain register form an architectural register file, or an alias to them and a number that specifies the cost of using this register. Allowed only once per register.

Example 56: reg_alloc_order example

```
settings
{
    compiler
    {
```

```
        reg_alloc_order = { gpr_all, "true",
                            rf_gpr[8], rf_gpr[9], rf_gpr[15], rf_gpr[14], rf_gpr[13],
                            rf_gpr[12], rf_gpr[11], rf_gpr[10], rf_gpr[1], rf_gpr[3],
                            rf_gpr[4], rf_gpr[5], rf_gpr[6], rf_gpr[7],
                            rf_gpr[16], rf_gpr[17], rf_gpr[18], rf_gpr[19], rf_gpr[20],
                            rf_gpr[21], rf_gpr[22], rf_gpr[23], rf_gpr[24], rf_gpr[25],
                            rf_gpr[26], rf_gpr[27], rf_gpr[29], rf_gpr[30], rf_gpr[31]
                          };
    };
};
```

Hazards

- **`eliminate_data_hazards`** – a bool. If set to true, C/C++ compiler handles data hazards.
- **`eliminate_structural_hazards`** – a bool. If set to true, C/C++ compiler handles structural hazards.

VLIW Attributes

- **`slot_prefix`** – a slot position specification and number. All instructions that match given slot position are filled with zero bits prefix. The bitwidth of the instruction inside particular slot then matches given number.
- **`slot_suffix`** – a slot position specification and number. All instructions that match given slot position are filled with zero bits suffix. The bitwidth of the instruction inside particular slot then matches given number.

Auxiliary

- **`verbatim_text`** – a string. Useful in case when I need to pass something that is not supported by CodAL yet. Allowed multiple times in the model.
- **`preprocessor_defines`** – a list of strings defined for every program compiled by the generated C/C++ compiler. Each string can either a standalone valid identifier (for example `__riscv`) or a valid identifier followed by '=' and a number (for example `__riscv_xlen=32`). The first method only defines a symbol, the second one is used to define a symbol with a numeric value.

5.4.8.2 Debugger Settings

The debugger settings section allows the user to specify the behaviour of the on-chip debugger as well as software debugger generated from a CodAL model. This section is optional. The most important value in this section is the **`type`**.

- **`type`** – enum value, selects the on-chip debugger type (`NEXUS`, `RISC_V`).

There are several options, most of which are exclusive depending on the type of the on-chip debugger selected:

Nexus

- **`instruction_address`** – a register, it holds an address of an instruction that is being executed.
- **`instruction_valid`** – a register, it denotes that the address on **`instruction_address`** is valid.
- **`enable`** – a signal indicating to the on-chip debugger whether to be active or not (`HIGH` means active).
- **`hw_break_request`** – a signal holding a hardware break request from the software debugger to the on-chip debugger (`HIGH` means a valid request).
- **`sw_break_request`** – a signal holding a software break request from the on-chip debugger to the software (`HIGH` means a valid request).
- **`trace_jump_address`** – optional; a 2-bit wide signal or register indicating whether a jump is executed and what kind of jump it is:
  - No jump (`CP_JUMP_NONE`, `0b00`),
  - Direct jump (`CP_JUMP_DIRECT`, `0b10`) – destination address is encoded in the instruction,
  - Indirect jump (`CP_JUMP_INDIRECT`, `0b11`) – destination address is stored in the register.

- **trace_jump_type** – optional; an *n*-bit wide signal or register storing the destination address of the jump.
- **sw_break_instruction** – optional in NEXUS; a list of opcodes of supported software break instructions.

RISC_V

- **dm_active** – a 1-bit signal indicating that the debug mode is active.
- **dm_halt** – a 1-bit signal indicating that the debug module wants the hart to go into the halted state.
- **hart_halted** – a 1-bit signal or register indicating that the core/hart is in the debug mode.
- **progbuf_base** – program buffer base address, must be aligned with respect to the total size of the program buffer.
- **progbuf_size** – the capacity of the program buffer (in 32-bit instruction words): 2, 4, 8, or 16.
- **progbuf_access** – interface(s) through which the program buffer becomes accessible, the bitwidth of the interface must not exceed the total size of the program buffer.
- **progbuf_exception** – a 1-bit signal or register indicating that an exception occurred during the execution of the program buffer.
- **progbuf_start** – a 1-bit signal indicating the start of the execution of the program buffer.
- **progbuf_done** – a 1-bit signal or register indicating that the execution of the program buffer ended.
- **sw_break_instruction** – required in RISC_V; a list of opcodes of supported software break instructions.

Several of the debugger settings are also used for purposes other than the description of a debug module. Most notably they are used for executing syscalls, profiling, instruction tracing and other advanced features. All of these settings are optional if not required by the selected debug module and can be use with any debug module type.

- **instruction_address** – see the NEXUS debug module settings.
- **instruction_valid** – see the NEXUS debug module settings.
- **trace_instruction_address** – a register, it overrides the **instruction_address** setting for profiling and tracing purposes, allowing different data to be used for profiling and debugging.
- **trace_instruction_valid** – a register, it overrides the **instruction_valid setting** for profiling and tracing purposes, allowing different data to be used for profiling and debugging.
- **trace_instruction_value** – optional; a register or signal containing the binary value of the currently executed instruction. It is used for the instruction trace in the IA and CA simulators. For CA models, it should contain an instruction at the address given by the **instruction_address** setting. For IA models, it should contain an instruction at the address given by the program counter at the beginning of the clock cycle.
- **trace_jump_address** – see the NEXUS debug module settings.
- **trace_jump_type** – see the NEXUS debug module settings.
- **syscall** – a register, a register in a register file or an interface with an address holding the address of the syscall payload. Note that this setting is applied only to CA models and requires JTAG.

When modeling a multi-thread processor design (see section 5.3 Arrays), some of the above-listed settings can be set differently for each thread, ensuring that the debugger and other tools can function correctly. These settings also support resource arrays or lists of multiple resources if more than one thread is specified in the model. The array size or number of resources in such cases must be the same as the number of threads in the design. All of these settings can still specify a single non-array resource, in which case all of the threads will use the same resource. Multi-thread designs do not support the use of the NEXUS debug module. The following debugger settings are affected: **dm_halt**, **hart_halted**, **progbuf_exception**, **progbuf_done**, **progbuf_start**, **instruction_valid**, and **instruction_address**.

5.4.8.3 Simulator Settings

No attribute/option is allowed here at the moment.

5.4.8.4 Assembler Settings

The assembler settings section allows the user to specify the behaviour of the assembler generated from a CodAL model. This section is optional. There are several options:

- **`new_line_delimiter`** – strings or characters that will have the same meaning in the assembly source code as a newline character. `;` is usually used for this purpose.
- **`comment_prefix`** – strings or characters that start a one-line comment in the assembly language. `//` is used by default. The **`comment_prefix`** option can add more one-line comment prefixes.
- **`code_section_byte_alignment`** – a value that specifies the default code sections byte alignment, e.g. when set to 16, each start of section of code type will be aligned to 16 bytes if not specified otherwise. If no value is specified then the program memory word size is used.
- **`symbol_prefix`** – a string consisting of characters `$`, `]`, `;`, `'`, `{`, `}` that are used by the generated C compiler and assembler as prefixes for symbols from any C source files. The default value is `$`.
- **`align_behavior`** – specifies the behaviour of the `.align` directive. Possible variants are `P2ALIGN` or `BALIGN`, with `BALIGN` being the default.
- **`le_instruction_parcel_bytes`** – specifies the number of bits that are reversed in the case of using the little endian system. The default value is evaluated as the shortest instruction length.

5.4.8.5 WFI Settings

The WFI settings provide options for configuring the wait-for-interrupt mode in the core. When configured, the generated core gates the clock on all resources using the register specified by the **`clock_enable`** option. Additionally, the user can define which ports will automatically wake the system up or explicitly exclude resources from the process.

This section is optional and has the following options:

- **`clock_enable`** – a 1-bit writable register. When this register is asserted, the core is in run mode. Otherwise, it is in sleep mode. Cannot be a pipeline register.
- **`exclude`** – a list of clocked resources that should be excluded from the process. The list may contain arrays of resources, or modules. The **`clock_enable`** register is excluded automatically. Optional.
- **`wakeup`** – a list of ASIP resources (input ports or component output ports, signals and registers) that will wake up the core when asserted (on any bit) and set the **`clock_enable`** register to 1. When the list is empty, the assertion has to be handled manually by the user. Optional, all ports in the list have to be of type `INTERRUPT`.

5.4.8.6 Architecture Settings

The architecture settings provide options that change or influence the architecture of the whole design. Currently the only settings available in this category are related to multi-thread architectures, see section 5.3 Arrays for more information.

- **`thread_count`** – a positive integer value specifying the number of threads in the processor design. The default value of this setting is 1, resulting in a single-thread architecture.
- **`thread_index`** – a register containing the index of the currently selected thread. This setting is used only in IA models to allow the debugger to distinguish between different thread operations without using the `RISC_V` debug module in CA models.

5.4.9 Schedule Class

A schedule class or a micro-architecture class specifies how an instruction should be scheduled by the C compiler. This involves handling data hazards caused by instruction latency (e.g. a long load should be put earlier to hide its latency). The compiler can also reorder instructions in order to minimize structural hazards. Finally control hazards can be handled automatically by the C compiler by introducing jump delay slots.

The following attributes apply:

- **latency** – specifies the latency of the class.
- **delay_slot** – specifies the number of delay slots in clock cycles. It is used for jump instructions.
- **llvm_class** – specifies a user-specific schedule class. This class cannot be used with other attributes.

The following example shows a schedule class for a load instructions.

Example 57: Schedule class (model/share/isa/isa.codal)

```
schedule_class loads
{
    latency = 2;
};
```

The syntax of the `ScheduleClass` construct is as follows:

```
ScheduleClass
   : "schedule_class" IdList ScheduleClassBodyEncap
```

```
ScheduleClassBodyEncap
   : '{' ScheduleClassBody '}'
   | %empty
```

```
ScheduleClassBody
   : ScheduleClassBody ScheduleClassAttribute ';'
   | ScheduleClassAttribute ';'
```

```
ScheduleClassAttribute
   : "latency" '=' CompileExpression
   | "delay_slot" '=' CompileExpression
   | "allow_in_delay_slot" '=' CompileExpression
   | "custom_schedule" '=' CompileExpression
   | "llvm_class" '=' String
```

5.5 Semantics

The semantic, or behavioral, part of a CodAL model is to be found in its **semantics** and **return** sections.

5.5.1 Semantics Section

A **semantics** section describes the behaviour of its containing **element** or **event**. It uses a restricted variant of the ANSI C language. Typically, the **element** or **event** behaviour is to move values from one resource to another, perhaps transforming the value in the process.

The restrictions with respect to the ANSI C language are:

- Pointers are not allowed in any form.
- Structures are not allowed.
- The `goto` statement is not allowed.
- A variable cannot be declared and initialized in one statement (e.g. `int a = 0;` is not allowed).
- Normal C functions are allowed as usual, but they cannot use pointers. All parameters have to be passed by value and can return only standard data types.
- There is a set of loop constraints to ensure correct loop unrolling. For more information, see section 5.6 Loops.

The following example shows the definition of an ALU unit:

Example 58: Semantics section (model/ca/pipelines/ca_pipe_stage3_ex.codal)

```
event alu_arith : pipeline(pipe.EX)
{
    semantics
    {
         switch (r_ex_alu_op)
         {
             case EX_AND:
                 s_ex_alu_logic = r_ex_aluA & r_ex_aluB; break;
             case EX_OR:
                 s_ex_alu_logic = r_ex_aluA | r_ex_aluB; break;
             case EX_NOR:
                 s_ex_alu_logic = ~(r_ex_aluA | r_ex_aluB); break;
             case EX_XOR:
                 s_ex_alu_logic = r_ex_aluA ^ r_ex_aluB; break;

             case EX_SEXT8:
                 s_ex_alu_logic = (uint32)(int8)(r_ex_aluB & 0xFF); break;
             case EX_SEXT16:
                 s_ex_alu_logic = (uint32)(int16)(r_ex_aluB & 0xFFFF); break;

             case EX_LUI:
                 s_ex_alu_logic = ((uint32)r_ex_aluB) << 16; break;

             case EX_GETSTATUS:
                 s_ex_alu_logic = r_int_enabled; break;

             default:
                 s_ex_alu_logic = 0x0000;
                 codasip_error(1, "alu_logic() - unknown switch (0x%02X)!", r_ex_alu_op);
                 break;
         }
    };
};
```

The following example of a semantic section is using a loop construction:

Example 59: Resent event (model/ia/events/ia_main_reset.codal)

```
event reset
{
    semantics
    {
        uint6 i;

        // initialize registers to zero
        for (i = 0; i < RF_GP_SIZE; i++)
            rf_gp32[(uint5)i] = 0;
    };
};
```

The semantics section may contain activation of an event or a decoder. The activation is in a form of a function call. The event activation has no argument (see the Activation of Events on page 98), the decoder activation has one or two arguments (see the Activation of Decoders on page 99).

The syntax of the `SectionSemantics` construct is as follows:

```
SectionSemantics
   : "semantics" AnsiCompoundStatement
```

5.5.2 Return Section

This section contains a single expression that represents the return value of the **`element`**. This is used when instancing elements to access their operands. In situations where a **`set`** of **`element`**s shares a common group of operand combinations,

the **return** section can be used to determine which one was used during an instruction decoding. The expression is a regular ANSI C expression with constraints stated in Semantics Section on page 85. Moreover, the expression cannot have any side effect.

The following example shows two elements that have **return** sections. There is also a set instantiated in the **element**. The instance can be used within **semantics** as well as **return** section.

Example 60: Return section of opcodes (model/share/isa/isa_operands.codal)

```
element opc_add
{
    assembly { "add" };
    binary { OPC_ADD };
    return { OPC_ADD; };
};

element opc_sub
{
    assembly { "sub" };
    binary { OPC_SUB };
    return { OPC_SUB; };
};

set opc_alu += opc_add, opc_sub;

element i_alu
{
    use opc_alu;
    use gpreg as rd, rs;

    assembly { opc_alu rd "," rs };
    binary { opc_alu rd rs };

    semantics
    {
        switch (opc_alu)
        {
        case OPC_ADD:
            rf_gp32[rd] += rf_gp32[rs];
            break;
        case OPC_SUB:
            rf_gp32[rd] -= rf_gp32[rs];
            break;
        default:
            break;
        }
    };
};
```

The next example shows an immediate operand. The value of the immediate operand is returned to the element that instantiated the **element**.

Example 61: Return section of immediate operand (model/share/isa/isa_operands.codal)

```
// 5bit Signed Immediate Value
element simm5
{
    signed attribute bit[5] val;

    assembly { val };
    binary { val };
    return { val; };
};
```

The syntax of the `SectionReturn` construct is as follows:

```
SectionReturn
   : "return" '{' AnsiExpression OptionalSemicolon '}'
```

5.5.3 Function

Functions may have any number of arguments. They are passed by value, not by reference. The function may be marked as `inline`. In this case, the hardware generator and the simulator generate several copies of the function depending on number of uses. There is no need for a function declaration.

The following example shows a function that takes opcode and operands as arguments. It perform an arithmetic or logical operation, then returns a result.

Example 62: Function definition (model/ia/other/ia_utils.codal)

```
int32 alu(int opc9, int32 op1, int32 op2)
{
    int32 tmp32;

    switch (opc9)
    {
        case OPC9_ADD:
            return op1 + op2;
        case OPC9_SUB:
            return op1 - op2;
        case OPC9_MUL:
            return op1 * op2;
        case OPC9_AND:
            return op1 & op2;
        case OPC9_OR:
            return op1 | op2;
        case OPC9_NOR:
            return ~(op1 | op2);
        case OPC9_XOR:
            return op1 ^ op2;
        case OPC9_SHL:
            return op1 << op2;
        case OPC9_ASHR:
            return op1 >> op2;
        case OPC9_LSHR:
            return ((uint32)op1) >> op2;
        default:
            return 0;
    }
}
```

The syntax of the `AnsiFunctionDefinition` can be found in Annex: CodAL Syntax Summary on page 209

5.5.4 Data Types

CodAL supports all basic types defined in ANSI C but, unlike ANSI C, it exactly specifies the bit-width and sign of every type. Moreover, it adds several types that are useful for processor descriptions.

Table 5: Basic ANSI C data types

Type	Bit-width	Signedness	Description
void	0	unknown	Special type which can be used only as return type or parameter of function e.g. void foo(void).
bool	1	unsigned	Equal to uint1.
char	8	signed	Equal to int8.
short	16	signed	Equal to int16.
int	32	signed	Equal to int32.
long	64	signed	Equal to int64.
long long	128	signed	Equal to int128.
float	32	signed	
double	64	signed	
long double	128	signed	This type is not supported now.

All types described in Table 5 can be preceded with qualifiers (`signed`, `unsigned`, `const`, `dont_touch`) to change the default signedness or add special attributes. The `dont_touch` qualifier denotes that the variable should not be optimized out. Not all possible combination are correct.

The `bool` type has the same size as `uint1` but with different semantics for assignment operation. Instead of simple value truncation the `bool` type converts the value to `true` when it is nonzero or to `false` when it is zero.

Table 6: Extended CodAL basic data types

Type	Bit-width	Signedness	Description
int**N**	N	signed	Bit-width of signed CodAL integer is limited to 2-2048.
int_<**E**>	E	signed	Bit-width is denoted by a compile expression **E**. **E** is limited to 2-2048.
uint**N**	N	unsigned	Bit-width of signed CodAL integer is limited to 1-2048.
uint_<**E**>	E	unsigned	Bit-width is denoted by a compile expression **E**. **E** is limited to 1-2048.
float**N**	16, 32, 64	signed	Half, standard and double precision float type. **N** can be only 16, 32, or 64.

Integer and floating point types described in Table 6 are just extensions of basic ANSI C types with an explicit specification of bit-width. No special restrictions are applied to these types, which can be used with standard operators. Signedness is encoded in the type name and cannot be changed with any qualifier.

Table 7: CodAL vector data types

Type	Bit-width	Signedness	Description
v**E**i**N**	E*N	signed	Vector with signed integer elements. **E** is the number of elements and must be greater than zero. **N** is the bit-width of an integer element and is limited to 2-2048.
v_i_<**Y**, **Z**>	Y*Z	signed	Vector with signed integer elements, where **Y** and **Z** are compile time expressions. **Y** denotes the number of elements and must be greater than zero. **Z** denotes the bit-width of an integer element and is limited to 2-2048.
v**E**u**N**	E*N	unsigned	Vector with unsigned integer elements.**E** is the number of elements and must be greater than zero. **N** is the bit-width of an integer element and is limited to 1-2048.
v_u_<**Y**, **Z**>	Y*Z	unsigned	Vector with unsigned integer elements, where **Y** and **Z** are compile time expressions. **Y** denotes the number of elements and must be greater than zero. **Z** denotes the bit-width of an integer element and is limited to 1-2048.
v**E**f**N**	E*N	signed	Vector with floating point elements. **E** is the number of elements and must be greater than zero. **N** is the bit-width of floating point element and can be only 16, 32, or 64.
v_f_<**Y**, **Z**>	Y*Z	signed	Vector with floating point elements, where **Y** and **Z** are compile time expressions. **Y** denotes the number of elements and must be greater than zero. **Z** denotes the bit-width of floating point element and can be only 16, 32, or 64.

Table 7 describes the extension of basic scalar data types to vector types. Only some basic operations are defined on these types and no implicit conversions are performed, so only compatible types (types with the same scalar type and the same number of elements) can be used in expressions.

Note that the vector type with only one element is not equivalent to the scalar type of the same type.

5.5.5 Enumerations

Enumerations consists of named constant values called an enumerator. The enumerator can be used in many places including almost any **`element`** or **`event`** section or function. It can have a value or a type explicitly defined. When the value is not specified, the value taken is that of the of previous enumerator increased by one. If there is no previous enumerator, zero is taken as the default value. If the type is not explicitly specified, then a type is automatically computed. The bit-width of the type is the minimum that is need to store any value within the enum.

The following example shows two **`enums`**. The first one has no type, so the type is automatically computed as **`uint3`**. The second enum has the type explicitly specified.

Example 63: Enumerators (model/share/include/opcodes.hcodal)

```
enum
{
    AM_SHL0,
    AM_SHL1,
    AM_SHL2,
    AM_SHL3,
    AM_CONST = 0x7
};

enum condition_t : uint4
{
    COND_SLT = 0x2,
    COND_SGE,
    COND_SLE,
    COND_SGT
};
```

The syntax of the `Enum` construct can be found in Annex: CodAL Syntax Summary on page 209.

5.5.6 CodAL Operators

CodAL supports all ANSI C operators except pointer operators (`*`, `&`) and the `sizeof` operator. Moreover the language adds a bit-select operator `[..]`. Table 8 summarizes all supported operators, their precedence and associativity.

Table 8: CodAL operators and its precedence

Precedence	Operator	Description	Associativity
1	`++`	Postfix increment	Left to right
	`--`	Postfix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`[..]`	Bit-select	
	`.`	Structure member access	
	`++`	Prefix increment	
	`--`	Prefix decrement	
	`clog2`	Bit width of constants	
	`bitsizeof`	Bit size of types and resources	
2	`+`	Unary plus	Right to left
	`-`	Unary minus	
	`!`	Logical NOT	
	`~`	Bitwise NOT	
	`(type)`	Type cast	
3	`*`	Multiplication	
	`/`	Division	
	`%`	Remainder	
	`**`	Power	
	`::*`	Replication	
4	`+`	Binary addition	
	`-`	Binary subtraction	
	`::`	Concatenation	
5	`<<`	Left shift.	
	`>>`	Right shift.	
	`<<<`	Rotation left	
	`>>>`	Rotation right	Left to right
6	`<`	Relational operator less.	
	`>`	Relational operator greater	
	`<=`	Relational operator less or equal	
	`>=`	Relational operator greater or equal	
7	`==`	Relational operator equal	
	`!=`	Relational operator not equal	
8	`&`	Bitwise AND	
9	`|`	Bitwise OR	
10	`^`	Bitwise XOR	
11	`&&`	Logical AND	
12	`||`	Logical OR	

Precedence	Operator	Description	Associativity
13	`?:`	Ternary conditional	Right to left
14	`=`	Simple assignment	
	`+=`	Assignment by sum	
	`-=`	Assignment by difference	
	`*=`	Assignment by product	
	`/=`	Assignment by quotient	
	`%=`	Assignment by remainder	
	`<<=`	Assignment by left shift	
	`>>=`	Assignment by right shift	
	`&=`	Assignment by bitwise AND	
	`|=`	Assignment by bitwise OR	
	`^=`	Assignment by bitwise XOR	
15	`,`	Comma	Left to right

5.5.6.1 Bit-select Operator

CodAL defines an operator for bit selection from an expression. The syntax of the operator is:

Example 64: Syntax of bit-select operator

```
expr[msb..lsb]
```

- `expr` represents a general expression,
- `msb` is an expression that defines the most significant bit,
- `lsb` is an expression that defines the least significant bit.

The semantics of this operator is:

Example 65: Semantics of bit-select operator

```
(uintX)(expr >> lsb)
```

where `X = (msb - lsb) + 1`.

The type of the result of this operation is unsigned integer and its bit-width is equal to the number of selected bits. Restrictions on operands are:

- `expr` can be only integer expression (constant or variable),
- `msb` must be greater than or equal to `lsb`,
- `msb` and `lsb` must be constant positive integers,
- `msb` and `lsb` must be in range of bit-width of `expr`.

5.5.7 Implicit conversions

Implicit conversions used in CodAL are different from standard ANSI C, as they have been adapted for specifying processors. Compared to ANSI C, there are no exceptions and rules are simpler to remember. Table 9 describes how the bit-width of a result is determined after the application of an operation to a variable of a given type. This table is valid for integer types as well as for floating point types. Some operators are not supported for floating point types.

Table 9: Implicit conversion of types when used in expressions

Expression	Bit-width	Description
Constant number	32, 64, 128, .., 1024	Integer or its power of two.
`clog2`, `bitsizeof`	32	Built-in functions return signed integer numbers.
`x op y`, where `op` is: `+ - * / % & | ^`	`max(W(x), W(y))`	
`op x`, where op is: `++ -- + - ~`	`W(x)`	Unary operators.
`!x`	1 (bool type)	
`x op y`, where `op` is: `== != > >= < <=`	1 (bool type)	Operands are converted to `max(W(x), W(y))`
`x op y`, where `op` is: `&& ||`	1 (bool type)	
`x op y`, where `op` is: `>>, <<, >>>, <<<, **`	`W(x)`	`y` is not converted.
`x = y`	`W(x)`	
`x op y`, where `op` is: `+= -= *= /= %= <<= >>= &= ^= |=`	`W(x)`	Operands are converted to `max(W(x), W(y))`
`x ? y : z`	`max(W(y), W(z))`	`x` is not converted.
`x[y..z]`	`(y - z) + 1`	`y` and `z` are not converted.
`x :: y`	`W(x) + W(y)`	
`x ::* y`	`W(x) * y`	

`W` stands for the bit-width of the operand.

Table 10: Implicit conversion of different types with binary operators

op	intN	uintM	float	double	vector
intN	intN	uintX	float	double	-
uintM	uintX	uintM	float	double	-
float	float	float	float	double	-
double	double	double	double	double	-
vector	-	-	-	-	vector

`X` in newly converted type `uintX` denotes the bit-width determined by the rule specified in Table 9.

The rules for the signedness of number are as follows:

- Comparison operator results are unsigned (bool type), regardless of the operands.
- If any operand is unsigned, the result is unsigned, regardless of the operator.
- If all operands are signed, the result will be signed, regardless of the operator.
- If any operand is real, the result is real and therefore signed.

5.5.8 Integer Literal

CodAL supports all integer literals from standard ANSI C and adds new literals for binary coding. The supported literal prefixes are:

- `[1-9]` for decimal literals,
- `0` for octal literals,
- `0x` or `0X` for hexadecimal literals,
- `0b` or `0B` for binary literals.

The type of the integer literal is determined from the minimal size to which the value can fit or from the suffix added to the integer literal. The minimum bit-width of the integer literal is 32 (type `int`) and the subsequent bit-widths are 64, 128, ..., 1024 (powers of two). The sign of number is always signed when it can fit into the minimal bit-width or unsigned when the value is too big to fit this bit-width (the most significant bit is set to 1). This default behaviour can be overridden with suffixes:

- `u` or `U` for unsigned integer,
- `l` or `L` for long int (64-bit).

The order of the suffixes is arbitrary. The suffix `L` can be repeated up to five times to describe the maximal bit-width (1024). Each use of the `L` suffix doubles the bit-width:

- `ll` or `LL` for long long int (128-bit),
- `lll` or `LLL` for long long long int (256-bit),
- ...
- `lllll` or `LLLLL` for long long long long long int (1024-bit).

In order to specify other bit-widths, an explicit type cast must be made:

Example 66: Specification of nonstandard bit-width of the integer literal.

```
uint9 d = (uint9)42u;
int65 x = (int65)0x1FFFFFFFFFFFFFFFFll;
```

5.5.9 Bundling integer

Bundling integer is derived from Codasip integer and it is an essential part of the bundle type structure, see chapter Bundling/Debundling on page 77 for more information. It supports only the following operators and conversions:

Table 11: Bundling integer operators

Operator	Description	Type support
`[..]`	Bit-select	`b_intX[intX..intX]`
`bitsizeof`	Bit size of types and resources	`bitsizeof(b_intX)`
`~`	Bitwise NOT	`~b_intX`
`(type)`	Type cast	`(b_intX)b_intY`, where `X` > `Y`
`::*`	Replication	`b_intX ::* intY`
`::`	Concatenation	`b_intX :: b_intY`
`<<`	Left shift	`b_intX << intY`
`>>`	Right shift	`b_intX >> intY`
`<<<`	Rotation left	`b_intX <<< intY`
`>>>`	Rotation right	`b_intX >>> intY`
`==`	Relational operator equal	`b_intX == b_intY` or `b_intX == intY`
`!=`	Relational operator not equal	`b_intX != b_intY` or `b_intX != intY`
`&`	Bitwise AND	`b_intX & b_intY` or `b_intX & intY`
`|`	Bitwise OR	`b_intX | b_intY` or `b_intX | intY`
`?:`	Ternary conditional	`intX ? b_intY : b_intY`
`=`	Simple assignment	`b_intX = b_intY` or `b_intX = intY`
`<<=`	Assignment by left shift	`b_intX <<= intY`
`>>=`	Assignment by right shift	`b_intX >>= intY`
`&=`	Assignment by bitwise AND	`b_intX &= b_intY` or `b_intX &= intY`
`|=`	Assignment by bitwise OR	`b_intX |= b_intY` or `b_intX |= intY`

Bundling integer is used as Codasip integer for these operators, but, furthermore, it tracks changes of bit positions. Subsequently, information about bit positions is used to correctly compute relocations after bundling function is completed.

5.6 Loops

CodAL language supports `for`, `while` and `do-while` loops in the same manner as ANSI C. However, there are several limitations to loops that are automatically unrolled by the CodAL compiler.

Loop unrolling is only performed on CA models. All loops in such a model are unrolled by default, except for loops located under the `#pragma simulator` statement. The loop unrolling is necessary for correct generation of RTL. All loops to be unrolled must meet certain constraints for the unrolling to be performed:

- A loop can have one or more control variables. Control variables are all local variables in the `semantics` section that are used in the loop condition.
- The loop condition section may only contain constant expressions and control variables. No resources, non-local variables, function calls or other constructs are allowed.
- Every control variable must be initialized unconditionally (from the loop perspective) to a constant value before the loop statement. Otherwise, the control variable can be used normally before or after the loop.
- Updates of control variables within a loop must be unconditional, i.e. they must not occur in the if or switch statements, and must contain only constant expressions or local variables (same as the loop condition).
- It is possible to have one loop inside the other. In this case, the control variables of the outer loop are treated as constants in the inner loop.

Example 67: Loop unrolling

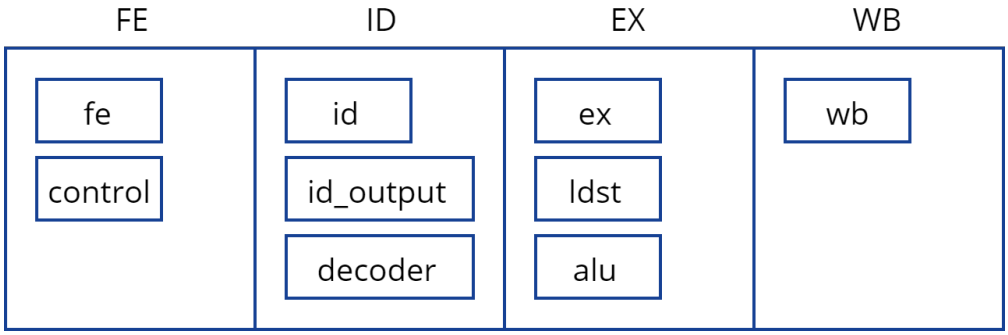
```
semantics
{
    uint32 i;
    for (i = 0; i < 5; i++)
    {
        codasip_print(1, "%d\n", i);
    }
};
```

5.7 Implementation

The implementation part of the processor model defines micro-architectural timing and optimal instruction decoding. It allows us to generate a CA simulator and RTL, from which real hardware may be synthesized. The main component of the hardware concerned by the implementation part of the processor model is a *pipeline controller*, and it is generally the most difficult part to specify.

The pipeline controller is captured by a set of **`event`**s, and the relationships between them. Each pipeline stage has a set of assigned **`event`**s, as shown in Figure 8 .

Figure 8: Pipeline stages and events



5.7.1 Event

event is a basic construct that is used in the implementation part of the processor model. Its functionality is captured in its **semantics** section. Each **event** can activate other **event**s.

An **event** can be assigned to a pipeline stage using a `Properties` construct. The supported property is:

- `pipeline(CompoundId)` – the `CompoundId` argument is the identifier of pipeline stage.

In most cases, there are several **event**s assigned to each pipeline stage. The simulation goes from the last pipeline stage to the first one. In other words, **event**s assigned to the last pipeline stage are executed first.

An event can have a **semantics** section which defines the semantics of the **event**. See Semantics Section on page 85.

Moreover, an event can have a local resources. The local resources can be used for advanced RTL optimizations (e.g. re-timing).

The following example shows an **event**, `fe`, used for instruction fetching:

Example 68: Event description for the fetch stage (model/ca/pipelines/ca_pipe_stage0_fe.codal)

```
event fe : pipeline(pipe.FE)
{
    semantics
    {
        // ------------------------------------------------------------------
        // Instruction fetch request
        if_fe.transport(CP_PHS_ADDRESS, CP_AHB_IDLE, CP_AHB_READ, r_pc, CP_AHB_SIZE_32);

        // ------------------------------------------------------------------
        // Program Counter
        r_pc = (s_ex_pc_we)
             ? s_ex_result & PC_MASK
             : r_pc + INSTRUCTION_SIZE;
    };
};
```

The syntax of the **event** construct is as follows:

```
Event
   : "event" Id Properties '{' EventSections '}'
   | "event" Id Properties '{' EventDeclarations EventSections '}'
```

```
EventDeclarations
   : EventDeclarations EventDeclaration ';'
   | EventDeclaration ';'
```

```
EventDeclaration
   : SectionUse
   | Register
   | RegisterFile
   | Signal
```

```
EventSections
   : EventSections EventSection
   | EventSection
```

```
EventSection
   : SectionSemantics ';'
```

The **use** section in an **event** definition is identical to that in an **element** definition and is described in Use Section on page 61.

5.7.1.1 Activation of Events

The activation of events becomes important when we want to model a processor on a cycle-accurate level. Roughly speaking, it defines a list of **event**s that should be activated when the current **semantics** section is handled. In the case of simulation, these **event**s are activated in the same order as listed in this section. When realized by the hardware, all **event**s are activated simultaneously.

The activation of an **event** can be conditional. The condition is an expression in the ANSI C language. It should not have any side effect. The expression can contain a test or a resource value or something similar.

The following example illustrates the **semantics** section. The `ex` **event** is assigned to the `pipe.EX` stage. The `cond_compare` and `ex_output` **event**s are activated in the same clock cycle. The `alu_logic`, `alu_arith` and `alu_mult` **event**s are also assigned to the same clock cycle and they are conditional, so the condition must be true for them to be activated. The `wb_output` **event** is assigned to the following pipeline stage, so it has to be activated as the first one for a simulation purposes.

Example 69: Timing section (model/ca/pipelines/ca_pipe_stage3_ex.codal)

```
event ex : pipeline(pipe.EX)
{
    use alu_logic;
    use alu_arith;
    use alu_mult;
    use cond_compare;
    use ex_output;
    use wb_output;

    semantics
    {
        wb_output();

        cond_compare();

        if (r_ex_act_alu_logic)
            alu_logic();

        if (r_ex_act_alu_arith)
            alu_arith();

        if (r_ex_act_alu_mult)
            alu_mult();

        ex_output();
    };
};
```

5.7.1.2 Activation of Decoders

The **semantics** section may describe an activation of instruction decoders. In contrast to the **start** section (which affects only the programming tools), this activation has a direct effect on the created hardware and simulation tools. The activation of decoder may be conditional within the **semantics** section. As for the **start** section, multiple slots can be described.

Having different descriptions for the programming tools and for the implementation allows optimal hardware. In many cases, operands can be just passed to the next pipeline stage, so there is no need to have any logic for their decoding. In this case, the decoder section has to deal with opcodes only.

The decoder is activated in the same manner as an **event**. In fact, the activation of decoder is very similar to an activation of an **event**. The only difference is that the decoder activation takes arguments. The activation has one mandatory argument: the place where the source data for instruction decoding is stored. It can be a register or signal. The second argument is optional. This argument can be a signal or register and it is set by an instruction decoder when an invalid instruction is decoded.

The decoder activation can be conditional too. The rules are the same as for **event**s.

The following example shows decoder activations. There is only one slot. The first decoder, `instr_opcode`, is activated all the time. The second decoder, `instr_alu`, is activated only if the control signal `s_id_alu` is set. Signals `s_id_opcode` and `s_id_alu` hold source data for decoding. Signals `s_iis_1` and `s_iis_2` are set by the decoders when an invalid instruction is detected (i.e. the signals `s_id_opcode` and `s_id_alu` contains data that cannot be decoded).

Example 70: Decoders activation (model/ca/pipelines/ca_pipe_stage1_id.codal)

```
event id : pipeline(pipe.ID)
{
    use instr_opcode;
    use instr_alu;

    signal bit[32] s_id_opcode;
    signal bit[32] s_id_alu;

    semantics
    {
        // Opcode
        s_id_opcode = s_id_instruction >> (I_W - OPC_W);
        s_id_alu = s_id_instruction & MASK_ALU;

        // HW decoder
        instr_opcode(s_id_opcode, s_iis_1);
        if (s_id_alu)
        {
            instr_alu(s_id_alu, s_iis_2);
        }
    };
};
```

5.7.1.3 Mandatory Events

Each processor description (i.e. IA and/or CA description) must have the following **event**s:

- `reset` **event** – contains only **semantics** section and defines actions that are performed during reset.
- `main` **event** – activated each clock cycle and it is a top level **event**. It usually contains **semantics** section that activated other events or decoders.

5.7.1.4 Synthesis Attributes

Pipeline events can be further modified by RTL synthesis attributes. The following attributes are currently supported:

- `RETIME` – enables retiming of registers in a defined **`event`**/**`module`**.
- `UNGROUP` – ungroups content of a defined **`event`**/**`module`** during synthesis, moves the content of the specified unit into the parent of the unit, and removes one design hierarchy level.
- `FLATTEN` – recursively ungroups all child instances of defined **`event`**/**`module`**, and reduces the hierarchy level(s) of the specified **`event`**/**`module`** to one.

The syntax can be updated if needed.

> **NOTICE:** These attributes are valid for CA models only. Combination of attributes `RETIME` and `UNGROUP` is not recommended.

The code example bellow shows possible syntax:

Example 71: : Pipelined multiplier modified by the RETIME synthesis attribute

```
event ex_mult : pipeline(pipe.EX), synthesis(RETIME)
{
    // ...
};
```

6 USING BUILTINS IN CODAL MODELS

This chapter is organized as follows:

Built-ins 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.

6.1 Debug Functions

6.1.1 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.

6.1.1.1 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.

6.1.1.2 Return Value

None.

6.1.1.3 Example

Example 72: codasip_assert function

```
semantics
{
    ...
    codasip_assert((r_pc % INSTR_LAU_SIZE) == 0, "PC is unaligned, PC = %d\n", r_pc);
    ...
};
```

6.1.2 codasip_break

Semantic category - simulator

Supported on - IA, CA

```
void codasip_break()
```

This function denotes an instruction used for SW break. When this builtin is executed on an IA simulator, the IA simulator stops at the next instruction following the SW break. On a CA simulator, it behaves as an interrupt that takes core into the debug mode. This builtin is ignored for RTL and UVM.

6.1.2.1 Parameters

None.

6.1.2.2 Return Value

None.

6.1.2.3 Example

Example 73: codasip_break function

```
semantics
{
    codasip_compiler_undefined();

    #pragma simulator
    {
        codasip_break();
    }
};
```

6.1.3 codasip_disassembler

Semantic category - simulator

Supported on - IA, CA

```
void codasip_disassembler(uint32 verbosity, uint2048 input, uint64 address, text isa, uint32 alignment)
```

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).

6.1.3.1 Parameters

- verbosity - Specifies the message group for simulator output. Must be an unsigned integer value.
- 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.
- isa(optional) - Name of the **`start`** section (default: "default").
- alignment - Maximum length of an output. If the output is shorter, then whitespaces (' ') are added. If the output is longer, then it's truncated.

6.1.3.2 Return Value

No return value.

6.1.3.3 Example

Example 74: codasip_dissasembler function

```
codasip_print(1, "ID: ");
codasip_disassembler(r_ir, r_pc);
codasip_print(1, "\n");
```

6.1.4 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.

6.1.4.1 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 75: codasip_print format specifier

```
codasip_print(1, "hello %#b", (uint4) 6);       // print unsigned 4-bit value 6 as binary integer with
base prefix

Output:
hello 0b110
```

6.1.4.2 Parameters

verbosity

Specifies the message group for simulator output. Must be an unsigned integer value.

> **NOTICE:** 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_error` function is in the format:

`error(<verbosity>): <processor name>@<clock cycle>: <formatted text>\n`

6.1.4.3 Return Value

None.

6.1.4.4 Example

Example 76: codasip_error function

```
#define ERROR_LEVEL       5

codasip_error(ERROR_LEVEL, "Unknown operation in i_cache_hw!");
```

Example 77: codasip_error function output

```
error(5): codasip_urisc@2500: Unknown operation in i_cache_hw!
```

6.1.5 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.

6.1.5.1 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 78: codasip_print format specifier

```
codasip_print(1, "hello %#b", (uint4) 6);       // print unsigned 4-bit value 6 as binary integer with
base prefix

Output:
hello 0b110
```

6.1.5.2 Parameters

exit_code

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>): <processor name>@<clock cycle>: <formatted text>\n
```

6.1.5.3 Return Value

None.

6.1.5.4 Example

Example 79: codasip_info function in i_halt (codasip_urisc)

```
codasip_fatal(21, "Unsupported syscall code %d.", code);
```

Example 80: codasip_info function in i_halt output(codasip_urisc)

```
fatal(21): codasip_urisc@666: Unsupported syscall 6.
```

6.1.6 codasip_get_clock_cycle

Semantic category - simulator

Supported on - IA

```
uint64 codasip_get_clock_cycle()
```

Returns the value of the cycle counter register.

6.1.6.1 Parameters

None.

6.1.6.2 Return Value

Value of the cycle counter register.

6.1.6.3 Example

Example 81: codasip_get_clock_cycle function

```
semantics
{
    uint64 cycles;

    cycles = codasip_get_clock_cycle();
    ...
};
```

6.1.7 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.

6.1.7.1 Parameters

value

The number of cycles added to the current cycle counter.

6.1.7.2 Return Value

None.

6.1.7.3 Example

Example 82: codasip_inc_clock_cycle function

```
semantics
{
```

```
    ...
    codasip_inc_clock_cycle(2); // two cycles are added to the cycle counter
    ...
};
```

6.1.8 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.

6.1.8.1 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 83: codasip_print format specifier

```
codasip_print(1, "hello %#b", (uint4) 6);       // print unsigned 4-bit value 6 as binary integer with
base prefix

Output:
hello 0b110
```

6.1.8.2 Parameters

verbosity

Specifies the message group for simulator output. Must be an unsigned integer value.

format

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>): <processor name>@<clock cycle>: <formated text>\n
```

6.1.8.3 Return Value

None.

6.1.8.4 Example

Example 84: 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 85: codasip_info function in i_halt output(codasip_urisc)

```
info(0): codasip_urisc@2300: Return value = 128
```

6.1.9 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`**.

6.1.9.1 Parameters

This function has no parameters.

6.1.9.2 Return Value

This function has no return value.

6.1.9.3 Example

Example 86: 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
        ...
};
```

6.1.10 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.

6.1.10.1 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 18: 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 19: 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 87: codasip_print format specifier

```
codasip_print(1, "hello %#b", (uint4) 6);       // print unsigned 4-bit value 6 as binary integer with
base prefix

Output:
hello 0b110
```

6.1.10.2 Parameters

format

A C string that containing the text to be written to the output.

6.1.10.3 Return Value

None.

6.1.10.4 Example

Example 88: codasip_print function

```
codasip_print(1, "pipeline EX clear\n");
```

Example 89: codasip_print function output

```
pipeline EX clear
```

6.1.11 codasip_store_exit_code

Semantic category - simulator

Supported on - IA, CA

```
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 containing the return value of the simulated program.

The default file for storing the exit code is `sim_exit_code` in the current execution directory. This can be overridden through simulator options.

6.1.11.1 Parameters

code

Exit code to be written into file.

6.1.11.2 Return Value

None.

6.1.11.3 Example

Example 90: 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);
};
```

6.1.12 codasip_symbol_address

Semantic category - simulator

Supported on - IA, CA

```
uint64 codasip_symbol_address(text name)
```

This function returns the address of the symbol in the `.xexe` file (e.g., codasip_argv).

6.1.12.1 Parameters

None.

6.1.12.2 Return Value

Address of the symbol in the `.xexe` file (e.g., codasip_argv).

6.1.12.3 Example

Example 91: codasip_symbol_address function

```
semantics
{
    ...
    #pragma simulator
    {
        uint32 start_addr, end_addr, address;

        start_addr = codasip_symbol_address("codasip_signature_start");
        end_addr = codasip_symbol_address("codasip_signature_end");

        for (address = start_addr; address < end_addr; address = address + 16)
        {
            // do something with current memory address
        }
    }
    ...
};
```

6.1.13 codasip_syscall

Semantic category - general

Supported on - IA, CA

```
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 Standard C Library*" in the *Codasip Tutorial 02 - CodAL Compiler Generation*.

6.1.13.1 Parameters

arguments_address

This argument points to a reserved register which holds the address of the structure with parameters for use by the host operating system syscall.

6.1.13.2 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.

6.1.13.3 Example

Example 92: codasip_syscall function in i_syscall element

```
element i_syscall
{
    assembly{ "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]);
        }
    };
};
```

6.1.14 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.

6.1.14.1 Parameters

verbosity

Specifies the message group for simulator output. Must be an unsigned integer value.

> **NOTICE:** 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>): <processor name>@<clock cycle>: <formatted text>\n`

6.1.14.2 Return Value

None.

6.1.14.3 Example

Example 93: codasip_warning function

```
#define WARNING_LEVEL       2

codasip_warning(WARNING_LEVEL, "Warning: Default case.");
```

Example 94: codasip_warning function output

```
warning(2): codasip_urisc@2500: Warning: Default case.
```

6.2 Timing Functions

6.2.1 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 92, when the `HALT` instruction is decoded, the simulation is terminated.

6.2.1.1 Parameters

None.

6.2.1.2 Return Value

None.

6.2.1.3 Example

Example 95: codasip_halt function in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_halt();
    ...
};
```

6.3 Compiler Functions

6.3.1 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/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.

6.3.1.1 Parameters

None

6.3.1.2 Return Value

None.

6.3.1.3 Example

Example 96: codasip_compiler_builtin function

```
semantics
{
    ...
    codasip_compiler_builtin();
    codasip_compiler_unused();
    ...
};
```

6.3.2 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`
- `int8`
- `int16`
- `int32`
- `int64`

6.3.2.1 Parameters

a, b

Operands to compare.

6.3.2.2 Return Value

None.

6.3.2.3 Example

Example 97: codasip_info function in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_compiler_flag_int8(src1, src2);
    ...
};
```

6.3.3 codasip_compiler_has_side_effects

Semantic category - compiler

Supported on - IA

```
void codasip_compiler_has_side_effects()
```

This function tells the compiler generator that this instruction has some unmodeled side effects. The compiler will then not attempt to move this instruction around in the code because that might affect observation of these side effects. Typically, this could be an exception thrown for certain values of the input operands. For example, a division instruction may throw an exception if its operand representing the divider is zero. In such case, the instruction should be modeled using this builtin as having the side effects.

6.3.3.1 Parameters

None

6.3.3.2 Return Value

None.

6.3.3.3 Example

Example 98: codasip_compiler_has_side_effects function

```
semantics
{
    ...
    codasip_compiler_has_side_effects();
    ...
};
```

6.3.4 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.

6.3.4.1 Parameters

loop_count

Number of iterations (loops) to be executed.

loop_end

End address of the loop (see Example 100).

6.3.4.2 Return Value

None.

6.3.4.3 Example

Example 99: codasip_compiler_hw_loop function used in i_hw_loop element (HWLOOP instruction)

```
element i_hw_loop
{
    ...
    assembly { "HWLOOP" loop_count "," loop_size};
    ...
    semantics
    {
        ...
        codasip_compiler_hw_loop(loop_count, loop_size);
        ...
    };
};
```

Example 100: 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
    ...
```

6.3.5 codasip_compiler_interrupt_return

Semantic category - compiler

Supported on - IA

`void codasip_compiler_interrupt_return(string mode)`

This function defines which instruction should be used to return from an interrupt in a certain mode – i.e. machine, user, or supervisor.

6.3.5.1 Parameters

`mode`

An optional argument that, if used, has to take one of the following values: `machine`, `user`, or `supervisor`. If not set, the default value is `machine`.

6.3.5.2 Return Value

None.

6.3.5.3 Example

Example 101: codasip_compiler_interrupt_return() for return from an interrupt in the machine mode

```
element i_mret
{
    use opc_mret as opc;

    assembly { opc };
    binary { opc[21..10] 0:bit[RF_GPR_ADDR + RF_GPR_PAD] opc[OPC_FRAG1] 0:bit[RF_GPR_ADDR + RF_GPR_
PAD] opc[OPC_FRAG0] };

    semantics
    {
        // equivalent to codasip_compiler_interrupt_return();
        codasip_compiler_interrupt_return("machine");
    };
};
```

By default, the interrupt handler takes no arguments and returns no value. However, it can be marked with the `_attribute_((interrupt()))`, which takes an optional string argument whose value has to be either `machine`, `user` or `supervisor`, and which determines the instruction to be returned from the handler. The value `machine` is the default when no argument is specified.

Example 102: codasip_compiler_interrupt_return() usage in C

```
#include <stdio.h>
#include <stdlib.h>

int a = 1;

void __attribute__((interrupt("machine"))) _int_handler()
{
    a = 42;

    // resume execution on the instruction following the one that raised the exception
    intptr_t mepc;
    __asm__("csrr %0, mepc" : "=r"(mepc));
    __asm__("csrw mepc, %0" : : "r"(mepc+4));
}

int main()
{
    __asm__("csrw mtvec, %0" : : "r"(_int_handler)); // set handler
    __asm__("ecall"); // raise exception

    if (a != 42)
        abort();
    return 0;
}
```

6.3.6 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.

6.3.6.1 Parameters

predop

The predicate.

6.3.6.2 Return Value

None.

6.3.6.3 Example

Example 103: codasip_compiler_predicate_false function

```
semantics
{
    ...
    codasip_compiler_predicate_false(predop); // Instruction will be executed only if predop
                                              // is 0
    ...
};
```

6.3.7 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.

6.3.7.1 Parameters

predop

The predicate.

6.3.7.2 Return Value

None.

6.3.7.3 Example

Example 104: codasip_compiler_predicate_true function

```
semantics
{
    ...
    codasip_compiler_predicate_true(predop); // Instruction will be executed only in case predop
```

```
                                        // is 1
    ...
};
```

6.3.8 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.

6.3.8.1 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.

6.3.8.2 Return Value

None.

6.3.8.3 Example

Example 105: codasip_compiler_priority function

```
semantics
{
    ...
    codasip_compiler_priority(5);
    ...
};
```

6.3.8.4 References

- http://llvm.org/devmtg/2008-08/Gohman_CodeGenAndSelectionDAGs.pdf
- http://llvm.org/devmtg/2008-08/Gohman_CodeGenAndSelectionDAGs_Hi.m4v

6.3.9 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*.

6.3.9.1 Parameters

index

Schedule class to be set.

6.3.9.2 Return Value

None.

6.3.9.3 Example

Example 106: codasip_compiler_schedule_class function

```
/* schedule class for load instruction

schedule_class loads
{
    latency = 3;
};

*/

semantics
{
    ...
    codasip_compiler_schedule_class(loads);
    ...
};
```

6.3.10 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.

6.3.10.1 Parameters

None.

6.3.10.2 Return Value

None.

6.3.10.3 Example

Example 107: codasip_compiler_undefined

```
semantics
{
    ...
    codasip_compiler_undefined();
    ...
```

```
};
```

6.3.11 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`.

6.3.11.1 Parameters

None.

6.3.11.2 Return Value

None.

6.3.11.3 Example

Example 108: codasip_compiler_unused in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_compiler_unused();
    ...
};
```

6.3.12 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.

6.3.13 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.

6.3.14 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.

6.3.14.1 Parameters

None.

6.3.14.2 Return Value

None.

6.3.14.3 Example

Example 109: codasip_nop function in i_nop (codasip_urisc)

```
semantics
{
    codasip_nop();
};
```

6.3.15 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.

6.3.15.1 Parameters

id

The constant being defined.

6.3.15.2 Return Value

None.

6.3.15.3 Example

Example 110: 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 111: using the defined constant in a compiled program

```
...
#ifdef ISE_BITCOUNT
   res = bitcount_fast();
#else
   res = bitcount_slow();
#endif
...
```

6.3.16 codasip_subreg_to_reg

Semantic category - instructions

Supported on - IA

```
void codasip_subreg_to_reg(int32 dst, int32 implsrc, int32 src, int32 subidx)
```

Represents instruction that takes value from source register, writes it in subregister of a destination register and assumes that the rest of the destination register is filled with value provided as the second operand. It can be used only in pattern and replace sections of peephole patterns.

6.3.16.1 Parameters

dst

Result register that has subregisters. It will contain source value concatenated with bits from second operand.

implsrc

Source value will be copied to subregister of the destination register. The rest of the destination register is presumed to contain this value.

src

Source register that contains source value.

subidx

Index of subregister that the value is copied into.

6.3.16.2 Return Value

None.

6.3.16.3 Example

Example 112: codasip_subreg_to_reg builtin

```
pattern
{
    shift_pat(opc = opc_srliw, dst = regs32, src = regs32);
    zext(srl0.dst = regs, sll0.src = shift_pat.dst);
};
replace
{
    ...
    codasip_subreg_to_reg(zext.srl0.dst, 0, shift_pat.dst, 1);
};
```

6.3.17 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

6.3.17.1 Parameters

None.

6.3.17.2 Return Value

Returns -1.

6.3.17.3 Example

Example 113: codasip_undef function

```
semantics
{
    ...
    int16 undef;
    undef = codasip_undef_int16(); // Contains -1
    ...
};
```

6.4 Arithmetic Functions

6.4.1 codasip_bitreverse

Semantic category - general

Supported on - IA, CA

`<type1> codasip_bitreverse_<type1>(<type1> src)`

Reverses bits in `src`. For example, if `src = 0x1111111111111100`, then the result after using this built-in will be `0x0011111111111111`.

The compiler cannot use this operation automatically.

Allowed values of `<type1>` are:

- Scalar integer with a bitwidth of 1-2048 bits, both signed and unsigned.
- Vector of integers with a bitwidth of 1-2048 bits, both signed and unsigned.

6.4.1.1 Parameters

`src`

Type `<type1>` operand whose value will be bit-reverted.

6.4.1.2 Return Value

Bit-reverted value. Return data type is `<type1>`.

6.4.1.3 Example

Example 114: codasip_bitreverse function

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = -4; // 1111 1111 1111 1100
    res = codasip_bitreverse_int32(src); // res = 0011 1111 1111 1111
};
```

6.4.2 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`

6.4.2.1 Parameters

a, b

Operands to subtract.

6.4.2.2 Return Value

1-bit borrow flag.

6.4.2.3 Example

Example 115: codasip_borrow_sub function

```
semantics
{
    uint1 borrow;
    ...
    borrow = codasip_borrow_sub_int32(reg_src1, reg_src2);
    ...
};
```

6.4.3 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`

6.4.3.1 Parameters

a, b, c

Operands to subtract. `c` is always 1-bit.

6.4.3.2 Return Value

1-bit borrow flag.

6.4.3.3 Example

Example 116: codasip_borrow_sub_c function

```
semantics
{
   uint1 borrow;
   ...
   borrow = codasip_borrow_sub_c_int32(reg_src1, reg_src2, cf);
   ...
};
```

6.4.4 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`

6.4.4.1 Parameters

a, b

Operands to add

6.4.4.2 Return Value

1-bit carry flag

6.4.4.3 Example

Example 117: codasip_carry_add function

```
semantics
{
    uint1 carry;
    ...
    carry = codasip_carry_add_int32(reg_src1, reg_src2);
    ...
};
```

6.4.5 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`

6.4.5.1 Parameters

a, b, c

Operands to add. `c` is always a 1-bit integer.

6.4.5.2 Return Value

1-bit carry flag

6.4.5.3 Example

Example 118: codasip_carry_add_c function

```
semantics
{
    uint1 carry;
    ...
    carry = codasip_carry_add_c_int32(reg_src1, reg_src2, carry_old);
    ...
};
```

6.4.6 codasip_ctlo

Semantic category - general

Supported on - IA, CA

`<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

6.4.6.1 Parameters

src

Operand of type `<type1>`, which will be used for counting the leading ones.

6.4.6.2 Return Value

Count of the most significant ones. Return data type is `<type1>`.

6.4.6.3 Example

Example 119: codasip_ctlo function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = -4; // 1111 1111 1111 1100
    res = codasip_ctlo_int16(src); // res = 14
};
```

6.4.7 codasip_ctlz

Semantic category - general

Supported on - IA, CA

```
<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

6.4.7.1 Parameters

src

Operand of type `<type1>`, which will be used for counting the leading zeros.

6.4.7.2 Return Value

Count of the most significant zeroes. Return data type is `<type1>`.

6.4.7.3 Example

Example 120: codasip_ctlz function

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = 2;
    res = codasip_ctlz_int32(src); // res = 30
};
```

6.4.8 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`

6.4.8.1 Parameters

src

An operand of type `<type1>`.

6.4.8.2 Return Value

Count of 1's. The return data type is `<type1>`.

6.4.8.3 Example

Example 121: codasip_ctpop function)

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = 33; // Bits 0 and 5 are set
    res = codasip_ctpop_int32(src); // res = 2
};
```

6.4.9 codasip_ctto

Semantic category - general

Supported on - IA, CA

`<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

6.4.9.1 Parameters

src

An operand of type `<type1>`.

6.4.9.2 Return Value

Count of the least significant ones. The return data type is `<type1>`.

6.4.9.3 Example

Example 122: codasip_ctto function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = 7; // 0000 0000 0000 0111
    res = codasip_ctto_int16(src); // res = 3
};
```

6.4.10 codasip_cttz

Semantic category - general

Supported on - IA, CA

```
<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

6.4.10.1 Parameters

src

An operand of type `<type1>`.

6.4.10.2 Return Value

Count of the least significant zeros. The return data type is `<type1>`.

6.4.10.3 Example

Example 123: codasip_cttz function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = 2; // 0000 0000 0000 0010
    res = codasip_cttz_int16(src); // res = 1
};
```

6.4.11 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`

6.4.11.1 Parameters

a, b

Operands to add.

6.4.11.2 Return Value

1-bit overflow flag.

6.4.11.3 Example

Example 124: codasip_carry_add function

```
semantics
{
    uint1 overflow;


    overflow = codasip_overflow_add_int32(reg_src1, reg_src2);
    ...
};
```

6.4.12 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`

6.4.12.1 Parameters

a, b, c

Operands to add. c is 1-bit.

6.4.12.2 Return Value

1-bit overflow flag.

6.4.12.3 Example

Example 125: codasip_overflow_add_c function

```
semantics
{
```

```
    uint1 overflow;

    overflow = codasip_overflow_add_c_int32(reg_src1, reg_src2, carry);
    ...
};
```

6.4.13 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`

6.4.13.1 Parameters

a, b

Operands to subtract.

6.4.13.2 Return Value

1-bit overflow flag.

6.4.13.3 Example

Example 126: codasip_overflow_sub function

```
semantics
{
    uint1 overflow;

    overflow = codasip_overflow_sub_int32(reg_src1, reg_src2);
    ...
};
```

6.4.14 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`

6.4.14.1 Parameters

a, b, c

Operands to subtract. c has 1-bit.

6.4.14.2 Return Value

1-bit overflow flag.

6.4.14.3 Example

Example 127: codasip_overflow_sub_c function

```
semantics
{
    uint1 overflow;

    overflow = codasip_overflow_add_c_int32(reg_src1, reg_src2, cf_old);
    ...
};
```

6.4.15 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.

6.4.15.1 Parameters

a

An operand with the following possible types:

- `int8`
- `int16`
- `int32`

6.4.15.2 Return Value

1-bit parity result.

6.4.15.3 Example

Example 128: 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.
    ...
};
```

6.4.16 codasip_onehot

Semantic category - simulator

Supported on - IA, CA

`uint1 codasip_onehot_<type1>(<type1> a)`

Determines whether exactly one bit is set in `value`. Equivalently, determines whether `value` is an integer power of 2.

6.4.16.1 Parameters

value

The input value.

6.4.16.2 Return Value

A boot indicating whether `value` has exactly one bit set.

6.4.16.3 Example

Example 129: codasip_onehot function

```
semantics
{
    codasip_assert(codasip_onehot_uint32(uinstr));
    // ...
};
```

6.4.17 codasip_onehot0

Semantic category - simulator

Supported on - IA, CA

`uint1 codasip_onehot0_<type1>(<type1> a)`

Determines whether at most one bit is set in `value`. Equivalently, determines whether `value` is an integer power of 2, or 0.

6.4.17.1 Parameters

value

The input value.

6.4.17.2 Return Value

A boot indicating whether `value` has at most one bit set.

6.4.17.3 Example

Example 130: codasip_onehot0 function

```
semantics
{
    codasip_assert(codasip_onehot0_uint32(uinstr));
    // ...
};
```

6.5 Saturated Arithmetic Functions

6.5.1 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>`.

6.5.1.1 Parameters

a, b

Operands to be added.

6.5.1.2 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>`.

6.5.1.3 Example

Example 131: codasip_usadd function

```
semantics
{
    ...
    // All operands are 8-bit, 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);
    ...
};
```

6.5.2 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.

6.5.2.1 Parameters

a, b

Operands to be added.

6.5.2.2 Return Value

Returns 1 if the carry flag has been generated by the addition, else returns 0.

6.5.2.3 Example

Example 132: codasip_usadd_occured function

```
semantics
{
    ...
    // All operands are 8-bit, which means that the maximum is 255
    uint8 a = 196;
    uint8 b = 100;
    uint1 res;

    // Addition generates the carry flag, so the saturation
    // has occurred. The 'res' will be set to 1.
    res = codasip_usadd_occured(a, b);
    ...
};
```

6.5.3 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>`.

6.5.3.1 Parameters

a, b

Operands to be subtracted.

6.5.3.2 Return Value

Returns subtraction of a and b. If the subtraction result generates the borrow flag, the result is the maximum value of `<type1>`.

6.5.3.3 Example

Example 133: codasip_ussub function

```
semantics
{
    ...
    // All operands are 8-bit, 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);
    ...
};
```

6.5.4 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.

6.5.4.1 Parameters

a, b

Operands to be subtracted.

6.5.4.2 Return Value

Returns 1 if the borrow flag has been generated by the subtraction, else returns 0.

6.5.4.3 Example

Example 134: codasip_ussub_occured function

```
semantics
{
    ...
    // All operands are 8-bit, which means that the maximum is 255
```

```
    uint8 a = 100;
    uint8 b = 196;
    uint1 res;

    // Addition generates the borrow flag, so the saturation
    // has occurred. The 'res' will be set to 1.
    res = codasip_ussub_occured(a, b);
    ...
};
```

6.6 Floating Point Functions

6.6.1 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.1.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

6.6.1.2 Return Value

The smallest integral value not smaller than a.

6.6.1.3 Example

Example 135: codasip_ceil function

```
semantics
{
    float32 a = 1.6f;
    float32 b;
    b = codasip_ceil_float32(a); //b = 2.0f;
    ...
};
```

6.6.2 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.2.1 Parameters

a

The argument of `<type1>` representing an angle expressed in radians.

6.6.2.2 Return Value

Cosine of a radians.

6.6.2.3 Example

Example 136: codasip_sin function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.5708f;
    b = codasip_cos_float32(a); //b = 0.0f;
    ...
};
```

6.6.3 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.3.1 Parameters

a

Value of the exponent of `<type1>` .

6.6.3.2 Return Value

Exponential of a.

6.6.3.3 Example

Example 137: codasip_exp function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.0f;
    b = codasip_exp_float32(a); //b = 2.718281828f;
    ...
};
```

6.6.4 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.4.1 Parameters

a

The argument of `<type1>` whose absolute value is returned.

6.6.4.2 Return Value

The absolute value of a.

6.6.4.3 Example

Example 138: codasip_fabs function

```
semantics
{
    float32 a = -1.0f;
    float32 b;
    b = codasip_fabs_float32(a); //b = 1.0f;
    ...
};
```

6.6.5 codasip_fcmp_oeq

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_oeq_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.5.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.5.2 Return Value

1-bit result.

6.6.5.3 Example

Example 139: codasip_fcmp_oeq function

```
semantics
{
    uint1 feq;

    feq = codasip_fcmp_oeq_float32(src1, src2);
    ...
};
```

6.6.6 codasip_fcmp_oge

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_oge_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.6.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.6.2 Return Value

1-bit result.

6.6.6.3 Example

Example 140: codasip_fcmp_oge function

```
semantics
{
    uint1 fge;

    fge = codasip_fcmp_oge_float32(src1, src2);
    ...
};
```

6.6.7 codasip_fcmp_ogt

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ogt_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.7.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.7.2 Return Value

1-bit result.

6.6.7.3 Example

Example 141: codasip_fcmp_ogt function

```
semantics
{
    uint1 fgt;

    fgt = codasip_fcmp_ogt_float32(src1, src2);
    ...
};
```

6.6.8 codasip_fcmp_ole

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ole_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.8.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.8.2 Return Value

1-bit result.

6.6.8.3 Example

Example 142: codasip_fcmp_ole function

```
semantics
{
    uint1 fle;

    fle = codasip_fcmp_ole_float32(src1, src2);
    ...
};
```

6.6.9 codasip_fcmp_olt

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_olt_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.9.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.9.2 Return Value

1-bit result.

6.6.9.3 Example

Example 143: codasip_fcmp_olt function

```
semantics
{
    uint1 feq;

    feq = codasip_fcmp_olt_float32(src1, src2);
    ...
};
```

6.6.10 codasip_fcmp_one

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_one_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.10.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.10.2 Return Value

1-bit result.

6.6.10.3 Example

Example 144: codasip_fcmp_one function

```
semantics
{
    uint1 fne;

    fne = codasip_fcmp_one_float32(src1, src2);
    ...
};
```

6.6.11 codasip_fcmp_ord

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ord_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.11.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.11.2 Return Value

1-bit result.

6.6.11.3 Example

Example 145: codasip_fcmp_ord function

```
semantics
{
    uint1 ford;

    ford = codasip_fcmp_ord_float32(src1, src2);
    ...
};
```

6.6.12 codasip_fcmp_ueq

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ueq_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.12.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.12.2 Return Value

1-bit result.

6.6.12.3 Example

Example 146: codasip_fcmp_ueq function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_ueq_float32(src1, src2);
    ...
};
```

6.6.13 codasip_fcmp_uge

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_uge_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.13.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.13.2 Return Value

1-bit result.

6.6.13.3 Example

Example 147: codasip_fcmp_uge function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_uge_float32(src1, src2);
    ...
};
```

6.6.14 codasip_fcmp_ugt

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ugt_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.14.1 Parameters

a, b

Operands to compare. Both must have the same bit-width.

6.6.14.2 Return Value

1-bit result depending float on comparison.

6.6.14.3 Example

Example 148: codasip_fcmp_ugt function

```
semantics
{
    uint1 fugt;

    fugt = codasip_fcmp_ugt_float32(src1, src2);
    ...
};
```

6.6.15 codasip_fcmp_ule

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ule_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.15.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.15.2 Return Value

1-bit result.

6.6.15.3 Example

Example 149: codasip_fcmp_ule function

```
semantics
{
    uint1 fule;

    fule = codasip_fcmp_ule_float32(src1, src2);
    ...
};
```

6.6.16 codasip_fcmp_ult

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_ult_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

Compares two float numbers of the same type. If `a` is lesser then `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 function 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.
```

6.6.16.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.16.2 Return Value

1-bit result.

6.6.16.3 Example

Example 150: codasip_fcmp_ult function

```
semantics
{
    uint1 fult;

    fult = codasip_fcmp_ult_float32(src1, src2);
    ...
};
```

6.6.17 codasip_fcmp_une

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_une_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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`.

6.6.17.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.17.2 Return Value

1-bit result.

6.6.17.3 Example

Example 151: codasip_fcmp_une function

```
semantics
{
    uint1 fune;

    fune = codasip_fcmp_une_float32(src1, src2);
    ...
};
```

6.6.18 codasip_fcmp_uno

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_uno_<type1>(<type1> a, <type1> b)
```

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

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 function 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.
```

6.6.18.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

6.6.18.2 Return Value

1-bit result.

6.6.18.3 Example

Example 152: codasip_fcmp_uno function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_uno_float32(src1, src2);
    ...
};
```

6.6.19 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.

6.6.19.1 Parameters

a

The allowed types for a are:

- `float16`
- `float32`

- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.19.2 Return Value

The value of a rounded down.

6.6.19.3 Example

Example 153: codasip_floor function (positive value use)

```
semantics
{
    float32 a;
    float32 b;

    a = 2.9f;
    b = codasip_floor_float32(a); // b = 2.0f;
    ...
};
```

Example 154: codasip_floor function (negative value use)

```
semantics
{
    float32 a;
    float32 b;

    a = -2.9f;
    b = codasip_floor_float32(a); // b = -3.0f;
    ...
};
```

6.6.20 codasip_fma

Semantic category - general

Supported on - IA

`<type1> codasip_fma_<type1>(<type1> a, <type1> b, <type1> c)`

This functions computes a*b+c.

6.6.20.1 Parameters

a, b, c

The allowed types of the arguments are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.20.2 Return Value

The result of the computation.

6.6.20.3 Example

Example 155: 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;
    ...
};
```

6.6.20.4 Usage

To automatically use an instruction marked with this builtin in an application, the application needs to be compiled with the argument `-ffp-contract=on`, or `-ffp-contract=fast`.

6.6.21 codasip_fpu_clear_exception

Semantic category - simulator

Supported on - IA, CA

```
int32 codasip_fpu_clear_exception(int32 exception)
```

This function attempts to clear the floating point exceptions specified by the exception argument.

6.6.21.1 Parameters

`exception`

Mask of exceptions to be cleared.

Exceptions have these values (can be defined as constants in the model):

- 1 - CODASIP_FPU_INVALID
- 2 - CODASIP_FPU_DIVBYZERO
- 4 - CODASIP_FPU_OVERFLOW
- 8 - CODASIP_FPU_UNDERFLOW
- 16 - CODASIP_FPU_INEXACT
- 31 - CODASIP_FPU_ALL_EXCEPT (all previous flags in one value)

6.6.21.2 Return Value

Zero if all exceptions in the exception argument were successfully cleared (or if the exception argument was zero). A non-zero value otherwise.

6.6.21.3 Example

Example 156: codasip_fpu_clear_exception function

```
semantics
{
    codasip_compiler_undefined();

    #pragma simulator
    {
        if (codasip_fpu_clear_exception(CODASIP_FPU_INVALID))
        {
            // error occurred
        }
    }
};
```

6.6.22 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)`.

6.6.22.1 Parameters

None.

6.6.22.2 Return Value

Rounding modes have these values (one can define them as constants in the model):

- 0 - round to the nearest even
- 1 - round to +infinity
- 2 - round to -infinity
- 3 - round to zero
- 4 - round to the nearest
- 5 - round to the nearest odd

6.6.23 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.

6.6.23.1 Parameters

`rounding_mode`

Rounding modes have these values (one can define them as constants in the model):

- 0 - round to the nearest even
- 1 - round to +infinity
- 2 - round to -infinity
- 3 - round to zero
- 4 - round to the nearest
- 5 - round to the nearest odd

6.6.23.2 Return Value

None.

6.6.23.3 Example

Example 157: Usage and constants definitions

```
#define TO_NEAREST_EVEN 0
#define UPWARDS 1
#define DOWNWARDS 2
#define TOWARDS_ZERO 3
#define TO_NEAREST 4
#define TO_NEAREST_ODD 5

...

codasip_fpu_setround(TO_NEAREST_EVEN);
```

6.6.24 codasip_fpu_test_exception

Semantic category - simulator

Supported on - IA, CA

`int32 codasip_fpu_test_exception(int32 exception)`

This function returns the floating point exceptions currently set, among those specified by the exception argument. The value returned is the bitwise OR representation of the subset of exceptions that are currently set in the floating point environment. If none of the exceptions in the exception argument are currently set, zero is returned.

6.6.24.1 Parameters

`exception`

Mask of exceptions to be tested.

Exceptions have these values (can be defined as constants in the model):

- 1 - CODASIP_FPU_INVALID
- 2 - CODASIP_FPU_DIVBYZERO
- 4 - CODASIP_FPU_OVERFLOW
- 8 - CODASIP_FPU_UNDERFLOW

- 16 - CODASIP_FPU_INEXACT
- 31 - CODASIP_FPU_ALL_EXCEPT (all previous flags in one value)

6.6.24.2 Return Value

Zero if none of the exceptions in the exception argument are set. Otherwise, the exceptions (among those of exception argument) currently set.

6.6.24.3 Example

Example 158: codasip_fpu_test_exception function

```
semantics
{
    codasip_compiler_undefined();

    #pragma simulator
    {
        if (codasip_fpu_test_exception(CODASIP_FPU_INVALID))
        {
            // exception CODASIP_FPU_INVALID is set
        }
    }
};
```

6.6.25 codasip_frem

Semantic category - general

Supported on - IA

`<type1> codasip_frem_<type1>(<type1> a, <type1> b)`

This function returns the floating point remainder of 'a/b' (rounded to zero).

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.25.1 Parameters

a

Numerator

b

Denominator

6.6.25.2 Return Value

The remainder of dividing the arguments. If denominator is zero, the function may either return zero or cause a domain error (depending on the library implementation).

6.6.25.3 Example

Example 159: codasip_frem function

```
semantics
{
    float64 res, a, b;
    a = 5.3;
    b = 2;
    res = codasip_frem_float64(a, b);
    // res == 1.3
    ...
};
```

6.6.25.4 Usage

The C/C++ operator used for computing the remainder (`%`) is not directly supported by floating point types. Instead, functions `fmod` (for double precision) and `fmodf` (for single precision) defined in `math.h` need to be used. When an application calling these functions is compiled with the `-ffast-math` argument, the calls will be automatically mapped to instructions marked with the `codasip_frem` builtin.

6.6.26 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.26.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

6.6.26.2 Return Value

The nearest integral value that is not greater than a in magnitude.

6.6.26.3 Example

Example 160: codasip_ftrunc function

```
semantics
{
    float32 a = 2.3f;
    float32 b;
```

```
    b = codasip_ftrunc_float32(a); //b = 2.0f;
    ...
};
```

6.6.27 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.27.1 Parameters

a

Value of `<type1>` whose logarithm is calculated.

6.6.27.2 Return Value

Natural logarithm of a.

6.6.27.3 Example

Example 161: codasip_log function

```
semantics
{
    float32 a;
    float32 b;

    a = 2.718281828f;
    b = codasip_log_float32(a); //b = 1.0f;
    ...
};
```

6.6.28 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.28.1 Parameters

a

Base value of `<type1>` .

b

Exponent value of `<type1>` .

6.6.28.2 Return Value

The result of raising the base to the power of the exponent.

6.6.28.3 Example

Example 162: 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;
    ...
};
```

6.6.29 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: a^b.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.29.1 Parameters

a

Base value of `<type1>` .

b

Exponent value of type `int32`.

6.6.29.2 Return Value

The result of raising the base to the power of the exponent.

6.6.29.3 Example

Example 163: codasip_powi function

```
semantics
{
    float32 a;
    int32 b;
    float32 c;

    a = 2.0f;
    b = 3;
    c = codasip_powi_float32(a,b); // c = 8.0f;
    ...
};
```

6.6.30 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.30.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

6.6.30.2 Return Value

The nearest integral value to a according to the current rounding mode.

6.6.30.3 Example

Example 164: codasip_rint function

```
semantics
{
    float32 a = 2.3f;
    float32 b;
```

```
    b = codasip_rint_float32(a); //b = 2.0f, depends on the current rounding mode
    ...
};
```

6.6.31 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.31.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

6.6.31.2 Return Value

The value of a rounded to the nearest integral.

6.6.31.3 Example

Example 165: codasip_round function

```
semantics
{
    float32 a = 3.8f;
    float32 b;
    b = codasip_round_float32(a); //b = 4.0f
    ...
};
```

6.6.32 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`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.32.1 Parameters

a

The argument of `<type1>` representing an angle expressed in radians.

6.6.32.2 Return Value

Sine of a.

6.6.32.3 Example

Example 166: codasip_sin function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.5708f;
    b = codasip_sin_float32(a); //b = 1.0f;
    ...
};
```

6.6.33 codasip_sqrt

Semantic category - general

Supported on - IA

`<type1> codasip_sqrt_<type1>(<type1> a)`

This function returns the square root (sqrt) of the specified operand if it is a non-negative floating point number.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- vector of floating point values of one of the above mentioned types (e.g. `v4f32`)

6.6.33.1 Parameters

a

The argument of `<type1>` whose square root is computed.

6.6.33.2 Return Value

Square root of a.

6.6.33.3 Example

Example 167: codasip_sqrt function

```
semantics
{
    float64 a;
    float64 b;

    a = 16.0f;
    b = codasip_sqrt_float64(a); //b = 4.0f;
    ...
};
```

6.7 Vector Functions

6.7.1 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.

6.7.1.1 Parameters

cond

Conditions vector.

a, b

Vectors containing elements, which are to be selected.

6.7.1.2 Return Value

Returns vector of `<type1>` with selected elements.

6.7.1.3 Example

Example 168: 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}
    ...
};
```

6.7.2 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>`.

6.7.2.1 Parameters

a

A vector whose elements are to be sign extended.

6.7.2.2 Return Value

Sign extended vector. Each element has the same bit width as `<type2>`.

6.7.2.3 Example

Example 169: 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}
    ...
};
```

6.7.3 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 *i*-th element of the first input vector, a (where the index starts at 0). If *i*>=*L*, then it indicates the *i* or *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

6.7.3.1 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.

6.7.3.2 Return Value

A vector with the same characteristics as a and b.

6.7.3.3 Example

Example 170: 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 171: 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)
    ...
};
```

6.7.4 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>`

6.7.4.1 Parameters

a

A vector whose elements are to be truncated.

6.7.4.2 Return Value

Truncated vector. Each element has the same bit width as `<type2>`.

6.7.4.3 Example

Example 172: 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}
    ...
};
```

6.7.5 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>`.

6.7.5.1 Parameters

a

A vector whose elements are to be zero extended.

6.7.5.2 Return Value

Zero extended vector. Each element has the same bit width as `<type2>`.

6.7.5.3 Example

Example 173: 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}
    ...
};
```

6.8 Fixed Point Functions

6.8.1 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`

6.8.1.1 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 there is no rounding.

6.8.1.2 Return Value

Returns the result of the division.

6.8.1.3 Example

Example 174: codasip_fx_div function

```
semantics
{
    ...
    fx_div_res = codasip_fx_div_int32(reg_src1, reg_src2, frac_bits, round);
    ...
};
```

6.8.2 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`

6.8.2.1 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.

6.8.2.2 Return Value

Returns the result of a conversion. The type is the same as `<type2>`.

6.8.2.3 Example

Example 175: codasip_fx_fptofx_to function

```
semantics
```

```
{
    ...
    fx_res = codaip_fx_fptofx_float32_to_int32(src, frac_bits);
    ...
};
```

6.8.3 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`

6.8.3.1 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.

6.8.3.2 Return Value

Returns the result of a conversion. The datatype is the same as `<type2>`.

6.8.3.3 Example

Example 176: codasip_fx_fptofx_to function

```
semantics
{
    ...
    fx_res = codaip_fx_fptofx_float32_to_int32(src, frac_bits);
    ...
};
```

6.8.4 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`

6.8.4.1 Parameters

a

The source fixed point number.

fract_bits

Number of bits representing the decimal part of the result.

6.8.4.2 Return Value

The result of the conversion.

6.8.4.3 Example

Example 177: 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.
    ...
};
```

6.8.5 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`

6.8.5.1 Parameters

a

The source integer number.

fract_bits

The number of bits in the decimal part of the result.

6.8.5.2 Return Value

The result of the conversion.

6.8.5.3 Example

Example 178: 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)
    ...
};
```

6.8.6 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`

6.8.6.1 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.

6.8.6.2 Return Value

Returns the result of a mutiplication.

6.8.6.3 Example

Example 179: codasip_fx_div function

```
semantics
{
    ...
    fx_mul_res = codasip_fx_mul_int32(reg_src1, reg_src2, frac_bits, round);
    ...
};
```

6.9 Complex Numbers Functions

6.9.1 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.

6.9.1.1 Parameters

a, b

Complex vector variables.

6.9.1.2 Return Value

Result of addition.

6.9.2 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 180: 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;
}
```

6.9.2.1 Parameters

a, b

Complex vector variables.

6.9.2.2 Return Value

Result of complex division.

6.9.3 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 181: 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;
}
```

6.9.3.1 Parameters

a, b

Complex vector variables.

6.9.3.2 Return Value

Result of complex multiplication.

6.9.4 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`.

6.9.4.1 Parameters

a, b

Complex vector variables.

6.9.4.2 Return Value

Result of complex subtraction.

Semantic category - general

Supported on - IA, CA

`<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`)
  - `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 182: codasip_bitcast

```
float codasip_bitcast_uint32_to_float32(uint32_t uninterpreted_value)
{
       return *((float *)&uninterpreted_value);
}
```

6.9.4.3 Parameters

value

Input value of type `<type1>` for the conversion.

6.9.4.4 Return Value

Returned converted value of type `<type2>`.

6.9.4.5 Example

Example 183: 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]);
}
```

6.10 Snippets

6.10.1 codasip_sverilog

Semantic category - general

Supported on - CA

Passes *System Verilog* snippet into RTL generated code. It will be ignored if other RTL language is selected in the `codal.conf`. Simulator or other targets ignore it as well.

6.10.1.1 Parameters

- `text` - System verilog code, with parameters being parsed as `%{number}`, starting with `0`. To pass `%` character, it must be escaped as `%%`.
- Optional: `arg`, ... - CodAL variables passed in code through `%{number}`.

6.10.1.2 Return Value

None.

6.10.1.3 Example

Example 184: codasip_sverilog function

```
semantics
{
  uint32 foo, bar;

  foo = r_foo + 1;
  bar = r_bar + 2;

  // embed display statement
  // %0 is foo, %1 is bar
  codasip_sverilog( "always @(*) $display(\"foo is %%d, bar+1 is %%d.\\n\", %0, (%1)+1);", foo, bar
);

   // coverpoint definition, multi-line code example
  codasip_sverilog( "covergroup FunctionalCoverage();\n"
                    "  cvp_foo: coverpoint %0 {\n"
                    "  /* ... */ \n"
                    "  }\n"
                    "endgroup\n", foo );
```

6.10.2 codasip_sverilog_assert

Semantic category - general

Supported on - CA

Passes *System Verilog Assertion* snippet into RTL generated code. It will be ignored if other RTL language is selected in the `codal.conf`. Simulator or other targets ignore it as well.

6.10.2.1 Parameters

- `text` - System Verilog Assertion, with parameters being parsed as `%{number}`, starting with `0`. To pass `%` character, it must be escaped as `%%`.
- Optional: `arg`, ... - CodAL variables passed in code through `%{number}`.

6.10.2.2 Return Value

None.

6.10.2.3 Example

Example 185: codasip_sverilog_assert function

```
semantics
```

```
{
  uint32 foo, bar;

  foo = r_foo + 1;
  bar = r_bar + 2;

  // embed SVA assertion into SystemVerilog RTL code
  codasip_sverilog_assert( "assert ##1 $onehot(%0) |-> $onehot(%1);", foo, bar );
```

6.10.3 codasip_verilog

Semantic category - general

Supported on - CA

Passes *Verilog* snippet into RTL generated code. It will be ignored if other RTL language is selected in the `codal.conf`. Simulator or other targets ignore it as well.

6.10.3.1 Parameters

- `text` - Verilog code, with parameters being parsed as `%{number}`, starting with `0`. To pass `%` character, it must be escaped as `%%`.
- Optional: `arg`, ... - CodAL variables passed in code through `%{number}`.

6.10.3.2 Return Value

None.

6.10.3.3 Example

Example 186: codasip_verilog function

```
semantics
{
  uint32 foo, bar;

  foo = r_foo + 1;
  bar = r_bar + 2;

  // embed display statement
  // %0 is foo, %1 is bar
  codasip_verilog( "  always @(*) begin\n"
                   "    $display(\"foo is %%d, bar+1 is %%d.\\n\", %0, (%1)+1);\n"
                   "  end", foo, bar );
```

6.10.4 codasip_cpp

Semantic category - simulator

Supported on - IA, CA

Injects C++ code into the simulator. This can be used to implement custom features, e.g. advanced instruction tracing, monitoring, etc. C++ snippets can access routines from the standard library and external component projects.

6.10.4.1 Parameters

- `text` - C++ code, with parameters being parsed as `%{number}`, starting with `0`. To pass `%` character, it must be escaped as `%%`.

- Optional: `arg`, ... - CodAL *variables*, *signals*, *ports*, *registers*, *register files* or constant expressions of integer or floating-point type. Arguments are inserted into the snippet in place of the corresponding `%{number}`. The only supported operations on resource objects are `read()` and `write()`.

6.10.4.2 Return Value

None.

6.10.4.3 Example

Example 187: codasip_cpp function

```
signal bit[32] s_my_ret, s_my_signal;

semantics
{
  ...
  codasip_cpp("%0.write(my_function(%1.read()))", s_my_ret, s_my_signal);
  codasip_cpp("printf(\"s_my_signal value: %%d\", %0.read())", s_my_signal);
  codasip_cpp("printf(\"s_my_signal width: %%d\", %0);", bitsizeof(s_my_signal));
  ...
  // multi-line example
  codasip_cpp("%0.write(0);\n"
              "%1.write(0);", s_my_ret, s_my_signal);
}
```

6.11 Miscellaneous Functions

6.11.1 codasip_bitcast_to

Semantic category - general

Supported on - IA, CA

`<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`)
  - `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 188: codasip_bitcast

```
float codasip_bitcast_uint32_to_float32(uint32_t uninterpreted_value)
{
       return *((float *)&uninterpreted_value);
}
```

6.11.1.1 Parameters

value

Input value of type `<type1>` for the conversion.

6.11.1.2 Return Value

Returned converted value of type `<type2>`.

6.11.1.3 Example

Example 189: 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]);
}
```

7 ANNEX: CODAL GUIDELINES

Please note that suggested rules for organizing files and directories are given in Annex: File Organization on page 205. Further, best practices associated with compiler generation are explained in *chapter "Compiler Generation Best Practices"* in the *Codasip Studio User Guide*.

7.1 Common Guidelines

7.1.1 CL025 Resources Naming Conventions

There are several types of resources which can be used in a processor description. Each resource type has a specific prefix denoting the type of resource. Some prefixes are optional and prefix can be omitted.

Resources in the processor description and their prefixes.

Table 20: Resources in the processor description and their prefixes

Type	Prefix	Type	Example
register	`r_`	Mandatory	`register bit[WORD_W] r_ir;`
register file	`rf_`	Mandatory	`register_file bit[DATA_W] rf_gp`
interface	`if_`	Mandatory	`interface if_ldst`
module	`m_`	Mandatory	`module m_1`
signal	`s_`	Mandatory	`signal bit[ALU_OP_W] s_alu_op;`
address_space	`as_`	Mandatory	`address_space as_data`
port	`p_`	Mandatory	`port bit[IRQS] p_irqs;`
schedule class	`sc_`	Mandatory	`schedule_class sc_loads;`

The `testbench` construct is used to describe an environment around a processor and it is a part of the CodAL processor project. Below, there are resources used for construct description together with their prefixes.

Table 21: Resources in the testbench construct description and their prefixes

Type	Prefix	Type	Example
interface	`if_`	Optional	`interface if_ldst; interface ldst`
port	`p_`	Optional	`port bit[1] fifo_out_wr; port bit[IRQS] p_irqs;`
memory	`m_`	Optional	`memory sram; memory m_sram`
bus	`b_`	Optional	`bus dbus; bus b_data`
cache	`c_`	Optional	`cache l1; cache c_l1`

Rationale: The prefix allows immediate identification of the type of resource without requiring examination of its definition.

7.1.2 CL070 Project Naming Conventions

Project names should be all lowercase with underscores "_" as separators.

Example 190: Project Naming Conventions

```
codasip_urisc
codix_helium
```

Rationale: Some operating systems are case insensitive (e.g. MS Windows). Therefore, the usage of lowercase prevents file names collisions.

7.1.3 CL021 Identifier Naming Conventions

All identifiers (i.e. variables, resources and functions) are written in lowercase with underscores "_" as separators between words.

Example 191: Identifier naming conventions

```
register bit[WORD_W] r_id_opcode;
...
semantics
{
    int opcode;

    opcode = r_id_opcode >> OPC_W;
};
```

Rationale: The code is unified, readable, and identifiers can be recognized from constants/macros/defines.

7.1.4 CL035 Indentation

Use four spaces instead of TAB.

Rationale: File can be opened in any editor and it looks the same.

7.1.5 CL040 Maximum Line Length

Maximum line length is 100 characters (see CL075 Line Wrapping/Splitting on page 189 for details how to maintain longer lines).

Rationale: Files can be opened in standard editors on standard monitors while maintaining readability.

7.1.6 CL045 Macros/Constants/Defines Naming Conventions

All macros, constants and defines should use uppercase with underscores "_" used as separators.

The first example shows a define enabling debug prints. The second example is an operation code definition (see CL315 Operation Codes Definition on page 192). The last example defines a macro which generates one operation code definition.

Example 192: Macros/Constants/Defines naming conventions

```
#define DEBUG

#define OPC_RR_ADD 0x123

#define DEF_OPC_RR(id, opcode) \
    element opc_##id \
    { \
```

```
        assembler { #id }; \
        binary { opcode:OPC_RR_W }; \
        \
        return { opcode; }; \
    };
```

Rationale: The code is unified, readable, and macros/constants/defines can be recognized from other identifiers.

7.1.7 CL050 `#define` Guard

All header files must have `#define` guards to prevent multiple inclusion.

Example 193: Define guards

```
#ifndef IA_UTILS_HCODAL
#define IA_UTILS_HCODAL
...
#endif  // IA_UTILS_HCODAL
```

Rationale: The guard is needed to prevent multiple inclusion, otherwise redeclaration of content of a header file may occur.

7.1.8 CL055 File Naming Conventions

File names should be lowercase and use underscores "_" as separators.

The file name extension of header files should be *.hcodal*, the file name extension of CodAL source files should be *.codal*.

Examples of valid file names:

Example 194: File naming conventions - valid

```
utilities.hcodal
ca_pipe_stage3_ex.codal
```

Examples of invalid file names:

Example 195: File naming conventions - invalid

```
ca-stg3ex.codl      // dash not allowed in the name
codixcobaltutils.codal // no separator between words, hard to read
CodixCobaltUtils.codal // uppercase in codal file name
ca_utils.c          // incorrect file suffix
```

Rationale: File names are consistent and easy to read and using the correct suffix enables CodAL compiler to recognize the origin of the file.

Exceptions: The files used for C compiler generation should follow LLVM naming convention (i.e. files placed in *<codal_project>*`/model/ia/compiler` and/or *<codal_project>*`/model/ca/compiler`. See Codasip Studio Technical Reference Manual chapter "*File Organization*".

7.1.9 CL060 Comments

Every function, resource, event, element, set or variable declaration should have Doxygen comments immediately preceding it that describe what the function, resource, event, element, set or variable does and how to use it.

Example 196: Comments

```
/**
 *  \brief  Brief description of function.
 *  \param  opc Brief description of parameter opc.
 *  \param  src1 Brief description of parameter src1.
 *  \param  src2 Brief description of parameter src2.
 *  \return Description of return value.
 */
uint32 function(const uint5 opc, const uint32 src1, const uint32 src2);

/// General purpose registers
register_file bit[DATA_W] rf_gp { size = RF_GP_SIZE; };
```

Rationale: Comments are essential for understanding of functionality and code reuse or code enhancement.

7.1.10 CL065 Preprocessor Directives

The hash mark "#" that starts a preprocessor directive should always be at the beginning of the line. Even when preprocessor directives are within the body of indented code, the directives should start at the beginning of the line.

Only `#pragma` may be aligned with code if it increases readability.

Example 197: Preprocessor directives

```
if (expression)
{
#ifdef ISA_EXTENSION_1
    isa_extension_1();
#else // ISA_EXTENSION_1
    no_isa_extension();
#ifdef DEBUG
    #pragma simulator
    {
        codasip_info(BASIC, ",,,\n");
    }
#endif // DEBUG
#endif // ISA_EXTENSION_1
}
```

Rationale: Placing the hash at the beginning of the line is easily recognized and ensures conditional code is easily visible.

7.1.11 CL075 Line Wrapping/Splitting

The incompleteness of split lines must be made obvious and it's done when a line limit (i.e. 100 characters) is exceeded. In general:

- Break after a comma.
- Break after a semicolon.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line, or use one indent (four spaces).
- Ternary operator is split using "?" and ":" symbols if the allowed line length is crossed.

Example 198: Line wrapping/splitting - function definition parameters

```
/**
 * \brief Perform arithmetic operations denoted by 'op'.
 * \param op Operation type
 * \param src1 First operand
 * \param src2 Second operand
 * \return Result of the operation. If the operation type is not supported, 0 is returned.
 */
int function(const uint32 op,
    const uint16 src1,
    const uint16 src2)
{
    ...
}
```

Example 199: Line wrapping/splitting - function call parameters

```
res = function(op,
    src1,
    src2);
```

Example 200: Line wrapping/splitting - loop control statements

```
for (ii = 0;
    ii < MAX;
    ++ii)
{
    ...
}
```

Example 201: Line wrapping/splitting - operators (align to '=')

```
// align to '='
result = a + b + c +
         d + e + f +
         g;
```

Example 202: Line wrapping/splitting - ternary if

```
result = condition
         ? return_something
         : return_else;
```

Example 203: Line wrapping/splitting - operators (tab space]

```
// or use tab
result = a + b + c +
    d + e + f +
    g;
```

Rationale: Split lines prevent code exceeding the line limit and significantly improve readability. While it is difficult to give exhaustive rules for split lines, the examples demonstrate the intent.

7.1.12 CL085 Include Statements Placement

Include statements must be located at the top of a file only.

Rationale: Avoids unwanted compilation side effects by *hidden* include statements deep into a source file.

7.1.13 CL090 Magic Numbers

The use of magic numbers in the code should be avoided. Only when a number is self-explaining (e.g. bit width of immediate operand) and used only at one place, the constant is allowed. Use `#defines` for all other constants.

Example 204: Constants and magic numbers

```
semantics
{
       rf_gp[31] = pc;    // Wrong

       rf_gp[RA_IDX] = pc;  // Correct
};
```

Rationale: Readability and maintainability are enhanced by introducing a named constant. An alternative approach is usage of a method from which the constant can be accessed.

7.1.14 CL093 White Spaces in Expressions

Use spaces for separation of operators and operands in expressions.

Example 205: White spaces in expressions

```
res = a * b * c;            // NOT res=a*b*c;
res = alu(a, b, c);         // NOT res=alu(a,b,c);
for (ii = 0; ii < 10; ++ii) // NOT for(ii=0;ii<0;++ii)
while (cond)                // NOT while(cond)
if (cond)                   // NOT if(cond)
switch (variable)           // NOT switch(variable)
```

Rationale: Code readability.

7.1.15 CL100 Variable Declaration Formatting

Insert an empty line after a declaration block of variables to separate declaration from semantics.

Example 206: Variable declaration formatting

```
semantics
{
       // store result
       int data;
       // operation code
       int4 opc;

       // perform computation
       opc = x >> REG_OFFSET;
       data = opc == OPC_ADD ? y : z;
};
```

Rationale: The code is more readable and semantics is separated from variable declarations.

7.1.16 CL080 Include Statements Formatting (Rec)

Include statements should be sorted and grouped. They should be sorted by their hierarchical position in the system with low level files included first. Leave an empty line between groups of include statements. `#pragma include_tools` should be used as the last one.

Include file paths must never be absolute.

The following example shows the low level includes as the top, then instruction-accurate includes follow.

Example 207: Include statements formatting

```
#include "utils.hcodal"
#include "defines.hcodal"

#include "ia_utils.hcodal"
#include "ia_defines.hcodal"
```

Rationale: Include files are more readable and provides information about the modules that are involved.

7.2 Processor Guidelines

7.2.1 CL310 Constants in the Binary Sections

The binary section should contain as few constants as possible. The bit-widths of parts of the instruction encoding must be given by `#define` or computed from other `#define`. They must not be hard coded.

The following example shows a binary statement that starts with an operation code, followed by two register operands. The rest of the binary encoding is filled with zeros (note that an expression is used to compute the number of trailing zeros required).

Example 208: Constants in binary section

```
binary { OPC_RR_ADD:bit[OPC_RR_W] dst src PAYLOAD(INSTR_W - (2 * RI_W) - OPC_RR_W) };
```

Rationale : Information on bit field widths is maintained in a file with operation codes that is common to multiple source files significantly improving maintainability. Further, when the width of an operation code field is changed, there is no need to change the binary section.

7.2.2 CL315 Operation Codes Definition

All operation codes should be placed in the share folder in the file `opcodes.hcodal`.

Each operation code has the following form: OPC_<*CLASS*>_<*NAME*>, where <*CLASS*> represents a class of an instruction denoted by an operation code <*NAME*>. Classes usually represent binary instruction formats used in the processor. If there is only one class, the <*CLASS*> can be omitted. Each <*CLASS*> has an additional `#define` holding its bit-width OPC_*CLASS*_W.

The following example shows two classes. The first one, RR (register-register), defines two operation codes and a bit-width of this class. The second one, RI (register-immediate), defines two operation codes too and the bit-width is different.

Example 209: Definition of operation codes

```
// Using defines

#define OPC_RR_ADD 0x12
#define OPC_RR_SUB 0x13
#define OPC_RR_W      5

#define OPC_RI_ADD 0x22
#define OPC_RI_SUB 0x23
#define OPC_RI_W      6

// Using enums

enum opcs : uint4
{
   OPC_RR_ADD = 0x12,
   OPC_RR_SUB,
   OPC_RI_ADD = 0x22,
   OPC_RI_SUB
};
```

Rationale: Improved readability and maintainability of code.

7.2.3 CL380 Element/Set Naming Conventions

Names of element/set must conform to the following rules.

Top Level Elements/Sets

Non-VLIW processors have the set called `isa` as the top level set/element used in the *start* section. VLIW processors have the sets called `isa_slot_<X>` as the top-level sets/elements used in *start* section, where `<X>` is an order of a slot.

The following example shows the top-level set of Codasip uRISC processor.

Example 210: Set naming convention

```
set isa = i_nop, i_halt, i_alu, ...;
```

Elements/Sets Representing Operation Codes

Prefixes `opc_` or `opc_<X>_` are used for elements and sets representing operation codes of instructions. The second prefix should be used when a designer needs to distinguish special operation codes (e.g. <X> could represent SPECIAL or SPECIAL2 operation codes in MIPS).

The following example shows a definition of an element and a set.

Example 211: Elements/Sets representing operation codes

```
element opc_add
{
    assembler { "add" };
    binary { ADD_OPC:bit[OPC_W] };

    return { ADD_OPC; };
};

set opc_alu = opc_add, opc_sub, ...;
```

Elements/Sets Representing Addressing Modes

Prefix `am_`is used for elements/sets representing address modes.

The following example shows an addressing mode with an immediate shift.

Example 212: Elements/Sets representing addressing modes

```
element am_shl1
{
    use reg;

    assembler { reg "shl1" };
    binary { reg OPC_AM_SHL1:bit[OPC_AM_W] };

    return { gp_rf_read(reg) << 1 };
};
```

Elements Representing Attributes

- Element representing an unsigned immediate using `<X>` bits is named as `uimm<X>`.
- Element representing a signed immediate using `<X>` bits is named as `simm<X>`.
- Element representing an absolute address using `<X>` bits is named as `abs_addr<X>`.
- Element representing a relative address using `<X>` bits is named as `rel_addr<X>`.

The following example shows a signed attribute on 11 bits.

Example 213: Elements representing attributes

```
element simm11
{
    signed attribute bit[11] attr;

    assembler { attr };
    binary { attr };

    return { attr; };
};
```

Elements/Sets Representing Complex Instructions

Prefix `i_` is used for complex elements/sets representing an instruction or a set of instructions.

The following example shows a complex instruction with two register operands.

Example 214: Elements/sets representing complex instructions

```
element i_rr
{
    use reg as rs, rd;
    use opc_alu;

    assembler { opc_alu rd "," rs };
    binary { opc_alu rd rs };

    semantics
    {
        ...
    };
};
```

Elements Representing Aliases

Aliases end with `_alias` suffix.

The following example shows an alias.

Example 215: Element representing alias

```
element i_nop_alias : assembler_alias(i_and) { ... };
```

Rationale: Using this unified approach the semantics of elements/sets is clear without going into their implementation.

7.2.4 CL333 Debug Prints

For debug prints, error messages etc. a designer should use build-in printing functions. There are several types of them:

- `void codasip_print(const int type, const char* fmt, ...);`
- `void codasip_info(const int type, const char* fmt, ...);`
- `void codasip_warning(const int type, const char* fmt, ...);`
- `void codasip_error(const int type, const char* fmt, ...);`
- `void codasip_fatal(const int rc, const char* fmt, ...);`

The first parameter enables an automatic message sorting, the second parameter is a formatting string and then the data parameters follows.

Example 216: Debug print

```
semantics
{
     // output: "info(1): codasip_urisc@101: pc 0x123"
     codasip_info(1, "pc %x\n", pc);
};
```

Rationale: Built-in printing functions will produce unified output and each stream can be disabled.

7.2.5 CL375 Type Conversions

Type conversions must always be done explicitly. Never rely on implicit type conversion.

The following examples firstly cast `uint3` (`r_data`) to `int3` and then extend sign to `int5`.

Example 217: Type conversions

```
register bit[SPEC_DATA_W] r_data;
...
semantics
{
    int5 res;

    // sign extend from 3 bits to 5 bits
    res = (int5)(SPEC_DATA_TYPE)r_data;
};
```

Rationale: Ensures consistency and indicates that any mixing of types is intentional.

7.2.6 CL395 Uniform Processor Resource Declarations

Resources should have unified formatting.

The following examples show a preferred formatting (order of parameters used in resource's declarations is not mandatory).

Register and register file.

Example 218: Processor resource declarations - register and register file

```
// register with no parameter
register bit[DATA_W] r_data;
// register with one parameter
register bit[DATA_W] r_id_data { pipeline = pipe.ID; };
// register with more parameters
register bit[DATA_W] r_id_data
{
    pipeline = pipe.ID;
    reset = false;
};

// register file with one parameter
register_file bit[DATA_W] rf_gp { size = RF_GP_SIZE; };
// register file with two parameter
register_file bit[DATA_W] rf_gp
{
    size = RF_GP_SIZE;
    dataports = { RF_GP_RP, RF_GP_WP };
};
```

Signals.

Example 219: Processor resource declarations - signals

```
signal bit[OPC_W] s_opc;
```

Ports.

Example 220: Processor resource declarations - ports

```
port bit[DATA_W] p_data_out { direction = OUT; };
```

Interfaces.

Example 221: Processor resource declarations - interfaces

```
interface if_fe
{
    bits = { ADDR_W, WORD_W, LAU_W };
    type = MEMORY:MASTER;
    flag = R;
    endianness = BIG;
};
```

Address Spaces.

Example 222: Processor resource declarations - address spaces

```
address_space as_all
{
    bits = { ADDR_W, WORD_W, LAU_W };
    type = ALL;
    endianness = BIG;

    interfaces =
    {
        if_ldst,
        if_fe,
```

```
        ...
    };
};
```

Modules.

Example 223: Processor resource declarations - module

```
module m_1
{
    register bit[DATA_W] r_data;
    ...
};
```

Rationale: Unified formatting across all code.

7.2.7 CL377 Element or Event Formatting

`element` or `event` should have the following form. An empty new line should follow `use` and `binary` section.

Element

Example 224: Element formatting

```
element el
{
    use ...;

    assembler { ... };
    binary { ... };

    semantics
    {
        ...
    };
    return { ... };
    timing
    {
        ...
    };
};
```

Event

Example 225: Event formatting

```
event ev : pipe.STAGE
{
    use ...;

    start
    {
        ...
    };
    semantics
    {
        ...
    };
    decoders
    {
        ...
```

```
    };
    timing
    {
        ...
    };
};
```

Rationale: Unified formatting across all code.

7.2.8 CL378 `Decoder` Section Formatting

`decoder` section should have the following form.

Example 226: Decoder and start section formatting

```
// one slot, one decoder
decoders
{
    dec(r_ir);
};
// one slot, more decoders
decoders
{
    {
        dec1(r_ir1);
        dec2(r_ir2);
    }
};
// more slots
decoders
{
    // slot 1
    { ... };
    // slot 2
    { ... };
    ...
};
```

Rationale: Unified formatting across all code.

7.2.9 CL387 `if-else` Statement Formatting

An `if-else` statement should have the following form.

Example 227: if-else statement formatting

```
if (cond)
{
    ...
    do_someting();
    ...
}
else
{
    ...
    do_something_else();
    ...
}
```

Rationale: Unified formatting across all code.

7.2.10 CL388 `Switch` Statement Formatting

A `switch` should have the following form. Any `case` without the `break` with body must include a specific comment as the last line. The `default` should be present in any case.

Example 228: Switch statement formatting

```
switch (variable)
{
    case A:
    case B:
        ...
        // fall-through
    case C:
        ...
        break;
    case D:
    {
        uint5 res;
        ...
        break;
    }
    default:
        ...
        break;
}
```

Rationale: Unified formatting across all code.

7.2.11 CL389 Single Statement `if-else`, `for()` or `while()` Statements Formatting

Single statement blocks can have the following form. Note that `do-while` construction always uses brackets even for single statements.

Example 229: Statements formatting

```
if (condition)
    statement;

while (condition)
    statement;

for (initialization; condition; update)
    statement;
```

Rationale: When only one statement is used, then brackets can be omitted because of a code reduction.

7.2.12 CL390 Loops Formatting

Loops should have the following form.

Example 230: Loops formatting

```
for (ii = 0; ii < MAX; ++ii)
{
    ...
    do_something();
    ...
```

```
}

while (cond)
{
    ...
    do_something();
    ...
}

do
{
    ...
    do_sometning();
    ...
} while (cond);
```

Rationale: Unified formatting across all code.

7.2.13 CL392 `if-else` Cascade Formatting

The `if-else` cascade should have the following form. Note that if the`if` contains only one statements, the brackets can be omitted.

Example 231: if-else cascade formatting

```
if (cond1)
{
    ...
}
else if (cond2)
{
    ...
}
else if (cond3)
{
    ...
}
else
{
    ...
}
```

Rationale: Unified formatting across all code.

7.2.14 CL393 Functions Definitions Formatting

Use the following formatting.

Example 232: Function definition formatting

```
int function()
{
    ...
    do_something();
    ...
}
```

Rationale: Unified formatting across all code.

7.2.15 CL330 Loop Control Statements in the `for()` Construction

Only loop control statements are allowed in the `for()` construction.

Example 233: Control statements in the for() construction

```
semantics
{
    ...
    for (ii = 0, all = 0; ii < MAX; ++ii) // Incorrect
    {
        ...
    }

    all = 0;
    for (ii = 0; ii < MAX; ++ii)         // Correct
    {
        ...
    }
    ...
};
```

Rationale: Increase maintainability and readability. Make a clear distinction of what controls and what is contained in the loop.

7.2.16 CL320 Register File Access (Rec)

Within an instruction-accurate model, each register file should be accessed using a dedicated function or macro only.

- `<data_t><rf>_read(const uint addr);`
- `void <rf>_write(const <data_t> data, const uint addr);`

Where `<rf>` is a name of the register file and `<data_t>` is a data type of the register file.

Example 234: Register file access

```
// resource
register_file bit[DATA_W] rf_gp { size = RF_GP_SIZE; };
...
uint16 rf_gp_read(const uint addr)
{
    return rf_gp[addr & 0xf];
}
...
semantics
{
    uint16 src1, src2, res;

    src1 = rf_gp_read(i1);
    ...
    rf_gp_write(i3, res);
};
```

Rationale: When a design is changed, access to the register files remains consistent (e.g. register zero is always zero).

7.2.17 CL325 Variable Declarations (Rec)

Variables should be declared in the smallest scope possible.

The following example shows two scopes. The outer one contains `res` defined and it does not uses `tmp` variable. `tmp` is used in the inner scope only.

Example 235: Variable declarations

```
semantics
{
    uint16 res;

    ...
    for (ii = 0; ii < MAX; ++ii)
    {
        uint5 tmp;

        ...
    }
};
```

Rationale: Keeping the operations on a variable within a small scope makes it easier to control the effects and side effects of the variable.

7.2.18 CL335 Use of `break` and `continue` (Rec)

The use of `break` and `continue` in loops should be avoided.

Rationale: Increase readability and reduce errors introduced by overlooking statements.

Exceptions: Should be used if they give higher readability than structured counterparts.

7.2.19 CL345 Start Section Placement (Rec)

Start section should be placed in `isa.codal` file.

Example 236: Start placement

```
start
{
   roots ={ isa; }
};
```

Rationale: *Start* section is not tied to any particular event, so placing it in *main* event is the best practice.

7.2.20 Uniform Testbench Resource Declarations

Resources declarations should have the following unified formatting.

The following examples show the preferred formatting (order of parameters used in resource's declarations is not mandatory).

Memories and caches.

Example 237: Testbench construct resource declarations - memories and caches

```
testbench
{
    memory/cache mem
    {
        size = MEM_SIZE;
        latencies = { MEM_READ_LATENCY, MEM_WRITE_LATENCY };

        interface if_fe
        {
            bits = { ADDR_W, WORD_W, LAU_W };
```

```
            endianness = BIG;
            type = MEMORY:SLAVE;
            flag = R;
        };

        interface ldst
        {
            bits = { ADDR_W, WORD_W, LAU_W };
            endianness = BIG;
            type = MEMORY:SLAVE;
            flag = RW;
        };
        ...
    };
};
```

Buses.

Example 238: Testbench construct resource declarations - buses

```
testbench
{
    bus dbus
    {
        interface if_mem
        {
            bits = { ADDR_W, WORD_W, LAU_W };
            endianness = BIG;
            type = MEMORY:MIRROR_SLAVE;
            flag = RW;
        };

        interface if_ldst
        {
            bits = { ADDR_W, WORD_W, LAU_W };
            endianness = BIG;
            type = MEMORY:MIRROED_MASTER;
            flag = RW;
        };

        decoder =
        {
            0..0x400000 : if_mem;
            ...
        };

        arbiter =
        {
            if_ldst;
        };
    };
};
```

Externs.

Example 239: Testbench construct resource declarations - externs

```
testbench
{
    extern fifo
    {
        type = "fifo_t";

        port bit[DATA_W] p_data_out { direction = OUT; };

        interface if_dbus
        {
            type = CLB:SLAVE;
```

```
            flag = RW;
            bits = { ADDR_W, WORD_W, LAU_W };
            endianness = BIG;
        };
    };
};
```

Ports.

Example 240: Testbench construct resource declarations - ports

```
testbench
{
    port bit[DATA_W] p_data_out { direction = OUT; };
};
```

Interfaces.

Example 241: Testbench construct resource declarations - interfaces

```
testbench
{
    interface if_fe
    {
        bits = { ADDR_W, WORD_W, LAU_W };
        type = MEMORY:MASTER;
        flag = R;
        endianness = BIG;
    };
};
```

Rationale: Improved readability and maintainability of code.

7.2.20.1 CL605 Connections (Rec)

When connecting components, resources or processors within the processor, a designer should follow data flow from the left (output) to right (input).

The following example show two connections. The first one shows a data connection from a processor to a testbench port. The second example shows a connection from a testbench port to a processor input.

Example 242: Connections

```
connect p_data_out => pic.p_data_in;
connect pic.p_data_out => p_data_in;
```

Rationale: Improved readability and maintainability of code.

8 ANNEX: FILE ORGANIZATION

The following subsections describes the best practice for a file organization of CodAL source files.

8.1 Directory Structure of CodAL Project Describing Processor

8.1.1 Directory *<codal_project>*

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

- *<codal_project>*/doc/ – Optional directory. Contains documentation of the project (e.g. ISA description, scheme of the processor, ...).
- *<codal_project>*/libs/ – Optional directory. Contains ported standard C library (e.g. newlib), and startup code.
- *<codal_project>*/linker/ – Optional directory. Contains custom linker script with .lds extension.
- *<codal_project>*/model/ – Mandatory directory. Contains the processor model (see Directory <codal_project>/model on page 205)

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

Table 22: Directory hierarchy summary of processor project– directory <codal_project>

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

8.1.2 Directory *<codal_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:

- *<codal_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 <codal_project>/model/ca on page 207).
- *<codal_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 <codal_project>/model/ia on page 207).
- *<codal_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 <codal_project>/model/share on page 206).

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

Table 23: Directory hierarchy summary of processor project– directory <*codal_project*>/model

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

8.1.3 Directory <*codal_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:

- <*codal_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.
- <*codal_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 -v Optional file. Contains peephole patters for C compiler. This file may be split into more codal files for better readability.
- <*codal_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.
  - testbench.codal – File used for **testbench** construct trough which components are described in a processor project.
  - version.codal – File holding a version of Codasip Studio that is needed.
- <*codal_project*>/model/share/resources – Mandatory directory. Contains a description of the interfaces of the processor 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 processor (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 24: Directory hierarchy summary of processor project– directory <*codal_project*>/model/share

Directory hierarchy summary	
Directory	**Type**
<*codal_project*>/model/share/include	Mandatory
<*codal_project*>/model/share/isa	Mandatory
<*codal_project*>/model/share/other	Optional
<*codal_project*>/model/share/resources	Mandatory

8.1.4 Directory *<codal_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:

- *<codal_project>*`/model/ia/compiler` – Optional directory. Contains additional source files for compiler generator.
  - `user_semantics.sem` – Compiler source file name example.
- *<codal_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.
- *<codal_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.
- *<codal_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.
- *<codal_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 25: Directory hierarchy summary of processor project– directory <*codal_project*>/model/ia

Directory hierarchy summary	
Directory	**Type**
<codal_project>`/model/ia/compiler`	Optional
<codal_project>`/model/ia/events`	Mandatory
<codal_project>`/model/ia/include`	Optional
<codal_project>`/model/ia/other`	Optional
<codal_project>`/model/ia/resources`	Mandatory

8.1.5 Directory *<codal_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:

- *<codal_project>*`/model/ca/compiler` – Optional directory. Contains additional source files for the compiler generator.
  - `CodasipMicroClasses.td` – Source file name example.
- *<codal_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 the name of the decoder. The files within this directory follow 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.
- *<codal_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.
- *<codal_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.
- *<codal_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.
- *<codal_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 follow 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.
- *<codal_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 if necessary (e.g. split resources respecting a pipeline).

The following table summarizes the directory hierarchy of the cycle-accurate model.

Table 26: Directory hierarchy summary of processor project– directory *<codal_project>*/model/ca

Directory hierarchy summary	
Directory	**Type**
<codal_project>/model/ca/compiler	Optional
<codal_project>/model/ca/decoders/	Mandatory
<codal_project>/model/ca/events	Mandatory
<codal_project>/model/ca/include	Optional
<codal_project>/model/ca/other	Optional
<codal_project>/model/ca/include	Mandatory
<codal_project>/model/ca/resources	Mandatory

9 ANNEX: CODAL SYNTAX SUMMARY

The following syntax summary is valid for descriptions of processors: that is, .codal source files and files included into them.

```
Aacr
    : "aacr" '=' CompileExpression
    | "aacr" '=' CompileExpression ".." CompileExpression
```

```
AddressSpace
   : "address_space" Id '{' AddressSpaceBody '}'
```

```
AddressSpaceAttribute
   : Bits
   | Endianness
   | AddressSpaceType
   | "interfaces" '=' '{' AddressSpaceInterfaces OptionalComma '}'
   | "default" '=' CompileExpression
```

```
AddressSpaceBody
   : AddressSpaceBody AddressSpaceAttribute ';'
   | AddressSpaceAttribute ';'
```

```
AddressSpaceInterface
   : Id ':' CompileExpression ".." CompileExpression ':' CompoundId
   | CompileExpression ".." CompileExpression ':' CompoundId
   | Id ':' CompoundId
   | CompoundId
```

```
AddressSpaceInterfaces
   : AddressSpaceInterfaces ',' AddressSpaceInterface
   | AddressSpaceInterface
```

```
AddressSpaceType
   : "type" '=' Id
```

```
AnsiAdditiveExpression
   : AnsiMultiplicativeExpression
   | AnsiAdditiveExpression '+' AnsiMultiplicativeExpression
   | AnsiAdditiveExpression '-' AnsiMultiplicativeExpression
   | AnsiAdditiveExpression "::" AnsiMultiplicativeExpression
```

```
AnsiAndExpression
   : AnsiEqualityExpression
   | AnsiAndExpression '&' AnsiEqualityExpression
```

```
AnsiArgumentList
   : AnsiAssignmentExpression
   | AnsiArgumentList ',' AnsiAssignmentExpression
```

```
AnsiAssignmentExpression
   : AnsiConditionalExpression
   | AnsiUnaryExpression AnsiAssignmentOperator AnsiAssignmentExpression
```

```
AnsiAssignmentOperator
```

```
   : '='
   | "*="
   | "/="
   | "%="
   | "+="
   | "-="
   | "<<="
   | ">>="
   | "&="
   | "^="
   | "|="
   | ">>>="
   | "<<<="
```

```
AnsiAttribute
   : "__attribute__(())" '(' '(' AnsiAttributeBody ')' ')'
```

```
AnsiAttributeBody
   : AnsiAttributeBody ',' AnsiId
   | AnsiId
```

```
AnsiCastExpression
   : AnsiUnaryExpression
   | '(' AnsiDeclarationSpecifiers ')' AnsiCastExpression
```

```
AnsiCompileExpression
   : AnsiPrimaryExpression
   | '+' AnsiCompileExpression
   | '-' AnsiCompileExpression
   | '~' AnsiCompileExpression
   | '!' AnsiCompileExpression
   | "bitsizeof" AnsiCompileExpression
   | "bitsizeof" '(' AnsiDeclarationSpecifiers ')'
   | "clog2" AnsiCompileExpression
   | AnsiCompileExpression '|' AnsiCompileExpression
   | AnsiCompileExpression '&' AnsiCompileExpression
   | AnsiCompileExpression '^' AnsiCompileExpression
   | AnsiCompileExpression '+' AnsiCompileExpression
   | AnsiCompileExpression "::" AnsiCompileExpression
   | AnsiCompileExpression '-' AnsiCompileExpression
   | AnsiCompileExpression '*' AnsiCompileExpression
   | AnsiCompileExpression "**" AnsiCompileExpression
   | AnsiCompileExpression "::*" AnsiCompileExpression
   | AnsiCompileExpression '/' AnsiCompileExpression
   | AnsiCompileExpression '%' AnsiCompileExpression
   | AnsiCompileExpression ">>" AnsiCompileExpression
   | AnsiCompileExpression "<<" AnsiCompileExpression
   | AnsiCompileExpression ROR_OP AnsiCompileExpression
   | AnsiCompileExpression ROL_OP AnsiCompileExpression
   | AnsiCompileExpression "&&" AnsiCompileExpression
   | AnsiCompileExpression "||" AnsiCompileExpression
   | AnsiCompileExpression "==" AnsiCompileExpression
   | AnsiCompileExpression "!=" AnsiCompileExpression
   | AnsiCompileExpression '>' AnsiCompileExpression
   | AnsiCompileExpression '<' AnsiCompileExpression
   | AnsiCompileExpression "<=" AnsiCompileExpression
   | AnsiCompileExpression ">=" AnsiCompileExpression
   | AnsiCompileExpression '?' AnsiCompileExpression ':' AnsiCompileExpression
   | AnsiCompileExpression '[' AnsiCompileExpression ".." AnsiCompileExpression ']'
```

```
AnsiCompoundStatement
   : AnsiPragmaSpecifier '{' '}'
   | AnsiPragmaSpecifier '{' AnsiStatementList '}'
   | AnsiPragmaSpecifier '{' AnsiDeclarationList '}'
   | AnsiPragmaSpecifier '{' AnsiDeclarationList AnsiStatementList '}'
```

```
AnsiConditionalExpression
   : AnsiLogicalOrExpression
   | AnsiLogicalOrExpression '?' AnsiExpression ':' AnsiConditionalExpression
```

```
AnsiConstantExpression
   : AnsiConditionalExpression
```

```
AnsiDeclaration
   : AnsiDeclarationSpecifiers ';'
   | AnsiDeclarationSpecifiers AnsiInitDeclaratorList ';'
```

```
AnsiDeclarationList
   : AnsiDeclaration
   | AnsiDeclarationList AnsiDeclaration
```

```
AnsiDeclarationSpecifiers
   : AnsiDeclarationSpecifiersBasic
   | AnsiEnum
   | AnsiDeclarationSpecifiers AnsiEnum
```

```
AnsiDeclarationSpecifiersBasic
   : AnsiStorageClassSpecifier
   | AnsiDeclarationSpecifiersBasic AnsiStorageClassSpecifier
   | AnsiTypeSpecifier
   | AnsiDeclarationSpecifiersBasic AnsiTypeSpecifier
   | AnsiTypeQualifier
   | AnsiDeclarationSpecifiersBasic AnsiTypeQualifier
   | AnsiAttribute
   | AnsiDeclarationSpecifiersBasic AnsiAttribute
```

```
AnsiDeclarator
   : AnsiVariableDeclarator
   | AnsiFunctionDeclarator
```

```
AnsiEnum
   : "enum" '{' AnsiEnumeratorList OptionalComma '}'
   | "enum" ':' AnsiDeclarationSpecifiersBasic '{' AnsiEnumeratorList OptionalComma '}'
   | "enum" AnsiEnumId '{' AnsiEnumeratorList OptionalComma '}'
   | "enum" AnsiEnumId ':' AnsiDeclarationSpecifiersBasic '{' AnsiEnumeratorList Option-
alComma '}'
   | "enum" AnsiEnumId
```

```
AnsiEnumId
   : "identifier"
```

```
AnsiEnumerator
   : AnsiEnumId
   | AnsiEnumId '=' AnsiConstantExpression
```

```
AnsiEnumeratorList
   : AnsiEnumerator
   | AnsiEnumeratorList ',' AnsiEnumerator
```

```
AnsiEqualityExpression
   : AnsiRelationalExpression
   | AnsiEqualityExpression "==" AnsiRelationalExpression
   | AnsiEqualityExpression "!=" AnsiRelationalExpression
```

```
AnsiExclusiveOrExpression
   : AnsiAndExpression
   | AnsiExclusiveOrExpression '^' AnsiAndExpression
```

```
AnsiExpression
   : AnsiAssignmentExpression
   | AnsiExpression ',' AnsiAssignmentExpression
```

```
AnsiExpressionStatement
   : ';'
   | AnsiExpression ';'
```

```
AnsiForExpression
   : AnsiExpression
   | %empty
```

```
AnsiFunctionDeclarator
   : AnsiId '(' ')'
   | AnsiId '(' AnsiParameterList ')'
```

```
AnsiFunctionDefinition
   : AnsiDeclarationSpecifiers AnsiFunctionDeclarator AnsiCompoundStatement
```

```
AnsiId
   : "identifier"
   | '.' "identifier"
```

```
AnsiInclusiveOrExpression
   : AnsiExclusiveOrExpression
   | AnsiInclusiveOrExpression '|' AnsiExclusiveOrExpression
```

```
AnsiInitDeclarator
   : AnsiDeclarator
   | AnsiDeclarator '=' AnsiInitializer
```

```
AnsiInitDeclaratorList
   : AnsiInitDeclarator
   | AnsiInitDeclaratorList ',' AnsiInitDeclarator
```

```
AnsiInitializer
   : AnsiAssignmentExpression
```

```
AnsiIterationStatement
   : "while" '(' AnsiExpression ')' AnsiStatement
   | "do" AnsiStatement "while" '(' AnsiExpression ')' ';'
   | "for" '(' AnsiForExpression ';' AnsiForExpression ';' AnsiForExpression ')'
AnsiStatement
```

```
AnsiJumpStatement
   : "continue" ';'
   | "break" ';'
   | "return" ';'
   | "return" AnsiExpression ';'
```

```
AnsiLabeledStatement
```

```
   : "case" AnsiConstantExpression ':' AnsiStatementList
   | "case" AnsiConstantExpression ':'
   | "default" ':' AnsiStatementList
   | "default" ':'
```

```
AnsiLabeledStatementList
   : AnsiLabeledStatementList AnsiLabeledStatement
   | AnsiLabeledStatement
```

```
AnsiLogicalAndExpression
   : AnsiInclusiveOrExpression
   | AnsiLogicalAndExpression "&&" AnsiInclusiveOrExpression
```

```
AnsiLogicalOrExpression
   : AnsiLogicalAndExpression
   | AnsiLogicalOrExpression "||" AnsiLogicalAndExpression
```

```
AnsiMultiplicativeExpression
   : AnsiCastExpression
   | AnsiMultiplicativeExpression '*' AnsiCastExpression
   | AnsiMultiplicativeExpression '/' AnsiCastExpression
   | AnsiMultiplicativeExpression '%' AnsiCastExpression
   | AnsiMultiplicativeExpression "**" AnsiCastExpression
   | AnsiMultiplicativeExpression "::*" AnsiCastExpression
```

```
AnsiParameterDeclaration
   : AnsiDeclarationSpecifiers AnsiId
   | AnsiDeclarationSpecifiers
```

```
AnsiParameterList
   : AnsiParameterDeclaration
   | AnsiParameterList ',' AnsiParameterDeclaration
```

```
AnsiPostfixExpression
   : AnsiPrimaryExpression
   | AnsiPostfixExpression '[' AnsiExpression ']'
   | AnsiPostfixExpression '[' AnsiExpression ".." AnsiExpression ']'
   | AnsiPostfixExpression '(' ')'
   | AnsiPostfixExpression '(' AnsiArgumentList ')'
   | AnsiPostfixExpression '.' AnsiId
   | AnsiPostfixExpression "++"
   | AnsiPostfixExpression "--"
```

```
AnsiPragmaModuleId
   : AnsiPragmaModuleId ',' CompoundId
   | CompoundId
```

```
AnsiPragmaSpecifier
   : "#pragma simulator"
   | "#pragma compiler"
   | "#pragma module" AnsiPragmaModuleId
   | "#pragma unknown"
   | %empty
```

```
AnsiPrimaryExpression
   : AnsiId
   | "integer constant"
   | "boolean constant"
   | "real constant"
```

```
   | "char constant"
   | AnsiStringList
   | '(' AnsiExpression ')'
```

```
AnsiRelationalExpression
   : AnsiShiftExpression
   | AnsiRelationalExpression '<' AnsiShiftExpression
   | AnsiRelationalExpression '>' AnsiShiftExpression
   | AnsiRelationalExpression "<=" AnsiShiftExpression
   | AnsiRelationalExpression ">=" AnsiShiftExpression
```

```
AnsiSelectionStatement
   : "if" '(' AnsiExpression ')' AnsiStatement
   | "if" '(' AnsiExpression ')' AnsiStatement "else" AnsiStatement
   | "switch" '(' AnsiExpression ')' '{' AnsiLabeledStatementList '}'
```

```
AnsiShiftExpression
   : AnsiAdditiveExpression
   | AnsiShiftExpression "<<" AnsiAdditiveExpression
   | AnsiShiftExpression ">>" AnsiAdditiveExpression
   | AnsiShiftExpression ROL_OP AnsiAdditiveExpression
   | AnsiShiftExpression ROR_OP AnsiAdditiveExpression
```

```
AnsiStatement
   : AnsiCompoundStatement
   | AnsiExpressionStatement
   | AnsiSelectionStatement
   | AnsiIterationStatement
   | AnsiJumpStatement
```

```
AnsiStatementList
   : AnsiStatement
   | AnsiStatementList AnsiStatement
```

```
AnsiStorageClassSpecifier
   : "typedef"
```

```
AnsiStringList
   : "string"
   | AnsiStringList "string"
```

```
AnsiTypeQualifier
   : "const"
   | "inline"
   | "static"
   | "dont_touch"
```

```
AnsiTypeSpecifier
   : "bool"
   | "char"
   | "short"
   | "int"
   | "int_" '<' AnsiTypeSpecifierTemplate '>'
   | "long"
   | "signed"
   | "unsigned"
   | "void"
   | "v_[uif]_" '<' AnsiTypeSpecifierTemplate ',' AnsiTypeSpecifierTemplate '>'
   | "v*[uif]*"
   | "float"
```

```
   | "double"
   | "bundle*"
   | "debundle*"
   | "typename"
```

```
AnsiTypeSpecifierTemplate
   : AnsiShiftExpression
   | '(' AnsiConditionalExpression ')'
```

```
AnsiUnaryExpression
   : AnsiPostfixExpression
   | "++" AnsiUnaryExpression
   | "--" AnsiUnaryExpression
   | AnsiUnaryOperator AnsiCastExpression
   | "bitsizeof" AnsiUnaryExpression
   | "bitsizeof" '(' AnsiDeclarationSpecifiers ')'
   | "clog2" AnsiUnaryExpression
```

```
AnsiUnaryOperator
   : '+'
   | '-'
   | '~'
   | '!'
```

```
AnsiVariableDeclarator
   : AnsiId
   | AnsiId '[' AnsiConstantExpression ']'
```

```
Attribute
   : AttributeSpecifier "attribute" AttributeBit Id AttributeBodyEncap
```

```
AttributeAttribute
   : "encoding" '=' AnsiExpression
   | "decoding" '=' AnsiExpression
   | "base" '=' '{' AttributeBaseBody OptionalComma '}'
   | "symbol" '=' CompileExpression
   | "label" '=' CompileExpression
   | "illegal" '=' '{' AttributeIllegalBody '}'
```

```
AttributeBaseBody
   : AttributeBaseBody ',' AttributeBaseId
   | AttributeBaseId
```

```
AttributeBaseId
   : Id
   | "binary"
```

```
AttributeBit
   : "bit" '[' CompileExpression ']'
   | %empty
```

```
AttributeBody
   : AttributeBody AttributeAttribute ';'
   | AttributeAttribute ';'
```

```
AttributeBodyEncap
   : '{' AttributeBody '}'
   | %empty
```

```
AttributeIllegalBody
   : AttributeIllegalBody ',' CompileExpression
   | CompileExpression
   | AttributeIllegalBody ',' CompileExpression ".." CompileExpression
   | CompileExpression ".." CompileExpression
```

```
AttributeSpecifier
   : "signed"
   | "unsigned"
```

```
Bits
   : "bits" '=' '{' CompileExpression ',' CompileExpression ',' CompileExpression '}'
```

```
Bus
   : "bus" IdList '{' BusBody '}'
```

```
BusArbiter
   : "arbiter" '=' '{' BusArbiterBody OptionalComma '}'
```

```
BusArbiterBody
   : BusArbiterBody ',' Id
   | Id
```

```
BusAttribute
   : Interface
   | BusDecoder
   | BusArbiter
```

```
BusBody
   : BusBody BusAttribute ';'
   | BusAttribute ';'
```

```
BusDecoder
   : "decoder" '=' '{' BusDecoderBody OptionalComma '}'
```

```
BusDecoderBody
   : BusDecoderBody ',' BusDecoderSlave
   | BusDecoderSlave
```

```
BusDecoderSlave
   : CompileExpression ".." CompileExpression ':' Id
```

```
Cache
   : "cache" IdList '{' CacheBody '}'
```

```
CacheAttribute
   : "program" '{' CacheDefinitionBody '}'
   | "data" '{' CacheDefinitionBody '}'
   | "base" '=' ParameterExpression
   | "type" '=' Id
```

```
CacheBody
   : CacheBody CacheAttribute ';'
   | CacheAttribute ';'
```

```
CacheDefinitionAttribute
   : Interface
   | Size
   | Linesize
   | Numways
```

```
CacheDefinitionBody
   : CacheDefinitionBody CacheDefinitionAttribute ';'
   | CacheDefinitionAttribute ';'
```

```
ClockEnable
   : "clock_enable" '=' CompoundId
```

```
CompileExpression
   : AnsiCompileExpression
```

```
CompileExpressionList
   : CompileExpressionList ',' CompileExpression
   | CompileExpression
```

```
CompoundId
   : CompoundId '.' Id
   | Id
   | '.' Id
```

```
Connect
   : "connect" AnsiConditionalExpression "->" AnsiConditionalExpression
   | "connect" AnsiConditionalExpression "->" "open"
   | "connect" "open" "->" AnsiConditionalExpression
```

```
DataportAttribute
   : InterfaceFlag
```

```
DataportBody
   : DataportBody DataportAttribute ';'
   | DataportAttribute ';'
```

```
Default
   : "default" = ParameterExpression
   | "default" = '{' ParameterExpressionList '}'
```

```
Dff
   : "dff" '=' CompileExpression
```

```
Dwarf
    : "dwarf" '=' CompileExpression
    | "dwarf" '=' CompileExpression ".." CompileExpression
```

```
Element
   : "element" Id Properties '{' ElementSections '}'
   | "element" Id Properties '{' ElementDeclarations ElementSections '}'
```

```
ElementDeclaration
   : SectionUse
   | Register
```

```
   | RegisterFile
   | Signal
   | Attribute
```

```
ElementDeclarations
   : ElementDeclarations ElementDeclaration ';'
   | ElementDeclaration ';'
```

```
ElementSection
   : SectionIf
   | SectionSwitch
   | SectionAssembly ';'
   | SectionBinary ';'
   | SectionSemantics ';'
   | SectionReturn ';'
```

```
ElementSections
   : ElementSections ElementSection
   | ElementSection
```

```
Emulation
   : "emulation" Id Properties '{' EmulationDeclarations EmulationSections '}'
   | "emulation" Id Properties '{' EmulationSections '}'
```

```
EmulationDeclaration
   : SectionUse
   | Attribute
```

```
EmulationDeclarations
   : EmulationDeclarations EmulationDeclaration ';'
   | EmulationDeclaration ';'
```

```
EmulationSection
   : SectionInstructions
   | SectionIfInstructions
   | SectionSemantics ';'
```

```
EmulationSections
   : EmulationSections EmulationSection
   | EmulationSection
```

```
Endianness
   : "endianness" '=' Id
```

```
Event
   : "event" Id Properties '{' EventSections '}'
   | "event" Id Properties '{' EventDeclarations EventSections '}'
```

```
EventDeclaration
   : SectionUse
   | Register
   | RegisterFile
   | Signal
```

```
EventDeclarations
   : EventDeclarations EventDeclaration ';'
```

```
   | EventDeclaration ';'

EventSection
   : SectionSemantics ';'

EventSections
   : EventSections EventSection
   | EventSection

Extern
   : "extern" ExternSpecifier Id '{' ExternBody '}'
   | "extern" ExternSpecifier Id "as" IdList '{' ExternBody '}'

ExternAttribute
   : Interface
   | Port

ExternBody
   : ExternBody ExternAttribute ';'
   | ExternAttribute ';'

ExternSpecifier
   : "component"
   | "codal"
   | %empty

Id
   : "identifier"

IdList
   : IdList ',' Id
   | Id

Interface
   : "interface" IdList '{' InterfaceBody '}'

InterfaceAlignment
   : "alignment" '=' '{' InterfaceAlignmentBody '}'

InterfaceAlignmentAttribute
   : "data" '=' '{' CompileExpressionList '}'
   | "address" '=' CompileExpression

InterfaceAlignmentBody
   : InterfaceAlignmentAttribute ';'
   | InterfaceAlignmentBody InterfaceAlignmentAttribute ';'

InterfaceAttribute
   : Bits
   | Endianness
   | InterfaceType
   | InterfaceFlag
   | InterfaceAlignment
```

```
InterfaceBody
   : InterfaceBody InterfaceAttribute ';'
   | InterfaceAttribute ';'

InterfaceFlag
   : "flag" '=' Id

InterfaceType
   : "type" '=' Id ':' Id

Latencies
   : "latencies" '=' '{' LatenciesItem ',' LatenciesItem '}'

LatenciesItem
   : CompileExpression
   | '{' LatenciesItemBody '}'

LatenciesItemBody
   : LatenciesItemBody ',' CompileExpression
   | CompileExpression

Linesize
   : "linesize" '=' ParameterExpression

MappingStatements
   : MappingStatements ',' CompoundId ':' CompoundId OptionalComma
   | CompoundId ':' CompoundId OptionalComma

Memory
   : "memory" IdList '{' MemoryBody '}'

MemoryAttribute
   : Interface
   | Size
   | Latencies

MemoryBody
   : MemoryBody MemoryAttribute ';'
   | MemoryAttribute ';'

Module
   : "module" IdList Properties '{' ModuleBody '}'

ModuleBody
   : ModuleBody TranslationUnitBody
   | TranslationUnitBody

Numways
   : "numways" '=' ParameterExpression

OptionalComma
   : ','
   | %empty
```

```
OptionalSemicolon
   : ';'
   | %empty
```

```
Overlap
   : "overlap" '=' CompoundId
   | "overlap" '=' CompoundId '[' CompileExpression ".." CompileExpression ']'
   | "overlap" '=' CompoundId '[' CompileExpression ']'
   | "overlap" '=' CompoundId '[' CompileExpression ']' '[' CompileExpression ".." Com-
pileExpression ']'
```

```
ParameterExpression
   : AnsiCompileExpression
```

```
ParameterExpressionList
   : ParameterExpression
   | ParameterExpressionList ',' ParameterExpression
```

```
Peephole
   : "peephole" Id '{' PeepholeDeclarations PeepholeSections '}'
   | "peephole" Id '{' PeepholeSections '}'
```

```
PeepholeDeclaration
   : SectionUse
```

```
PeepholeDeclarations
   : PeepholeDeclarations PeepholeDeclaration ';'
   | PeepholeDeclaration ';'
```

```
PeepholeSection
   : SectionPattern ';'
   | SectionReplace
   | SectionIfReplace
   | SectionMapping ';'
```

```
PeepholeSections
   : PeepholeSections PeepholeSection
   | PeepholeSection
```

```
Pipeline
   : "pipeline" Id '{' PipelineBody OptionalComma '}'
```

```
PipelineBody
   : PipelineBody ',' Id
   | Id
```

```
Port
   : "port" "bit" '[' CompileExpression ']' IdList '{' PortBody '}'
```

```
PortAttribute
   : "direction" '=' Id
   | "type" '=' Id
```

```
PortBody
   : PortBody PortAttribute ';'
```

```
   | PortAttribute ';'
```

```
Properties
   : ':' PropertiesBody
   | %empty
```

```
PropertiesBody
   : PropertiesBody ',' Property
   | Property
```

```
Property
   : Id '(' PropertyArguments ')'
   | "pipeline" '(' PropertyArguments ')'
```

```
PropertyArgument
   : CompoundId
   | String
```

```
PropertyArguments
   : PropertyArguments ',' PropertyArgument
   | PropertyArgument
```

```
Register
   : Specifier "register" "bit" '[' CompileExpression ']' IdList RegisterBodyEncap
   | Specifier "register" "bit" '[' CompileExpression ']' IdList '[' CompileExpression
']' RegisterBodyEncap
```

```
RegisterAttribute
    : "write_enable" '=' CompileExpression
    | Default
    | "pipeline" '=' CompoundId
    | "write_mask" '=' CompileExpression
    | Overlap
    | Dff
    | Reset
    | ClockEnable
    | Dwarf
    | Aacr
```

```
RegisterBody
   : RegisterBody RegisterAttribute ';'
   | RegisterAttribute ';'
```

```
RegisterBodyEncap
   : '{' RegisterBody '}'
   | %empty
```

```
RegisterFile
   : Specifier "register_file" "bit" '[' CompileExpression ']' IdList '{' Register-
FileBody '}'
   | Specifier "register_file" "bit" '[' CompileExpression ']' IdList '[' Com-
pileExpression ']' '{' RegisterFileBody '}'
```

```
RegisterFileAttribute
    : "size" '=' CompileExpression
    | "dataport" IdList '{' DataportBody '}'
    | Default
```

```
    | Overlap
    | Dff
    | Reset
    | ClockEnable
    | Dwarf
    | Aacr

RegisterFileBody
   : RegisterFileBody RegisterFileAttribute ';'
   | RegisterFileAttribute ';'

Reset
   : "reset" '=' CompileExpression
   | "reset" '=' '{' CompileExpression ',' Id '}'

Resource
   : AddressSpace
   | Pipeline
   | Register
   | RegisterFile
   | Signal
   | Interface
   | Port
   | Memory
   | Cache
   | Bus

Root
   : TranslationUnit
   | %empty

ScheduleClass
   : "schedule_class" IdList ScheduleClassBodyEncap

ScheduleClassAttribute
   : "latency" '=' CompileExpression
   | "delay_slot" '=' CompileExpression
   | "allow_in_delay_slot" '=' CompileExpression
   | "custom_schedule" '=' CompileExpression
   | "llvm_class" '=' String

ScheduleClassBody
   : ScheduleClassBody ScheduleClassAttribute ';'
   | ScheduleClassAttribute ';'

ScheduleClassBodyEncap
   : '{' ScheduleClassBody '}'
   | %empty

SectionAssembly
   : "assembly" '{' SectionAssemblyBody '}'

SectionAssemblyBody
   : SectionAssemblyBody SectionAssemblyElement
   | SectionAssemblyElement

SectionAssemblyElement
   : Id
```

```
   | String
   | '~'
   | "STRINGIZE" '(' Id ')'


SectionBinary
   : "binary" '{' SectionBinaryBody '}'


SectionBinaryBody
   : SectionBinaryBody SectionBinaryElement
   | SectionBinaryElement


SectionBinaryElement
   : CompileExpression
   | CompileExpression ':' "bit" '[' CompileExpression ']'
   | "bit" '[' CompileExpression ']'


SectionIf
   : "if" '(' AnsiExpression ')' SectionIfBody


SectionIfBody
   : SectionIfSwitchCompoundSection "else" SectionIfSwitchCompoundSection


SectionIfInstructions
   : "if" '(' AnsiExpression ')' SectionIfInstructionsBody


SectionIfInstructionsBody
   : SectionInstructions
   | '{' SectionInstructions '}'


SectionIfReplace
   : "if" '(' AnsiExpression ')' SectionIfReplaceBody


SectionIfReplaceBody
   : SectionReplace
   | '{' SectionReplace '}'


SectionIfSwitchCompoundSection
   : ElementSection
   | '{' ElementSections '}'
   | '{' ElementDeclarations ElementSections '}'


SectionInstructions
   : "instructions" AnsiCompoundStatement ';'


SectionMapping
   : "mapping" '{' MappingStatements '}'


SectionPattern
   : "pattern" AnsiCompoundStatement


SectionReplace
   : "replace" AnsiCompoundStatement ';'
```

```
SectionReturn
   : "return" '{' AnsiExpression OptionalSemicolon '}'

SectionSemantics
   : "semantics" AnsiCompoundStatement

SectionSwitch
   : "switch" '(' AnsiExpression ')' '{' SectionSwitchBody '}'

SectionSwitchBody
   : SectionSwitchBody SectionSwitchBodyElememt "break" ';'
   | SectionSwitchBodyElememt "break" ';'

SectionSwitchBodyElememt
   : "case" AnsiConstantExpression ':' SectionIfSwitchCompoundSection
   | "default" ':' SectionIfSwitchCompoundSection

SectionUse
   : "use" SectionUseInstance
   | "use" CompoundId "as" SectionUseAsBody

SectionUseAsBody
   : SectionUseAsBody ',' SectionUseInstance
   | SectionUseInstance

SectionUseInstance
   : Id

Set
   : "set" Id SetAssign SetBody OptionalComma
   | "set" Id Properties

SetAssign
   : '='
   | "+="

SetBody
   : SetBody ',' SetBodyElement
   | SetBodyElement

SetBodyElement
   : CompoundId

Setting
   : Id '=' SettingElement
   | Id SetAssign '{' SettingBody OptionalComma '}'

SettingBody
   : SettingBody ',' SettingElement
   | SettingElement

SettingElement
   : AnsiConditionalExpression
   | AnsiTypeSpecifier
```

```
Settings
   : SettingsSpecifier '{' SettingsBody '}'
```

```
SettingsBody
   : SettingsBody Setting ';'
   | Setting ';'
```

```
SettingsCompound
   : "settings" '{' SettingsCompoundBody '}'
```

```
SettingsCompoundBody
   : SettingsCompoundBody Settings ';'
   | Settings ';'
```

```
SettingsSpecifier
   : "compiler"
   | "debugger"
   | "simulator"
   | "assembler"
   | "wfi"
   | "architecture"
```

```
Signal
   : SignalSpecifier "signal" "bit" '[' CompileExpression ']' IdList
   | SignalSpecifier "signal" "bit" '[' CompileExpression ']' IdList '[' Com-
pileExpression ']'
```

```
SignalSpecifier
   : "dont_touch"
   | %empty
```

```
Size
   : "size" '=' ParameterExpression
```

```
Specifier
   : "arch"
   | "pc"
   | "alias"
   | "dont_touch"
   | %empty
```

```
Start
   : "start" Id '{' StartBody '}'
   | "start" '{' StartBody '}'
```

```
StartAttribute
   : "roots" '=' '{' StartRootsBody OptionalComma '}'
   | "emulations" '=' '{' StartEmulationsBody OptionalComma '}'
   | "peepholes" '=' '{' StartPeepholesBody OptionalComma '}'
   | "bundling" '=' '{' CompoundId ',' CompoundId '}'
```

```
StartBody
   : StartBody StartAttribute ';'
   | StartAttribute ';'
```

```
StartEmulationsBody
```

```
   : StartEmulationsBody ',' CompoundId
   | CompoundId

StartPeepholesBody
   : StartPeepholesBody ',' CompoundId
   | CompoundId

StartRootsBody
   : StartRootsBody ',' CompoundId
   | CompoundId

String
   : "string"

Subtarget
   : "subtarget" Id '{' SubtargetBody '}'

SubtargetBody
   : SubtargetBody SubtargetAttribute ';'
   | SubtargetAttribute ';'

SubtargetAttribute
   : "argument" '=' String
   | "depends" '=' '{' IdList '}'
   | "enabled_by_default" '=' CompileExpression

Testbench
   : "testbench" '{' TestbenchBody '}'

TestbenchBody
   : TestbenchBody TestbenchBodyConstruct ';'
   | TestbenchBodyConstruct ';'

TestbenchBodyConstruct
   : Memory
   | Cache
   | Bus
   | Extern
   | Connect

TranslationUnit
   : TranslationUnit TranslationUnitBody
   | TranslationUnitBody

TranslationUnitBody
   : Resource ';'
   | Element ';'
   | Event ';'
   | Emulation ';'
   | Peephole ';'
   | Set ';'
   | Start ';'
   | ScheduleClass ';'
   | AnsiDeclaration
   | AnsiFunctionDefinition
   | Module ';'
   | Extern ';'
   | Connect ';'
   | SettingsCompound ';'
   | Testbench ';'
```

```
    | ';'
```

10 ANNEX: CODAL KEYWORDS

Here is a list of the keywords in CodAL:

```
STRINGIZE
address
address_space
alias
alignment
allow_in_delay_slot
arbiter
arch
as
assembler
assembly
attribute
base
binary
bit
bits
bitsizeof
bool
break
bundling
bus
cache
case
char
clock_enable
clog2
codal
compiler
component
connect
const
continue
custom_schedule
data
dataport
debugger
decoder
decoding
default
delay_slot
dff
direction
do
dont_touch
double
element
else
emulation
emulations
encoding
endianness
enum
event
extern
flag
float
for
identifier
if
illegal
inline
instructions
int
```

```
int_
interface
interfaces
label
latencies
latency
linesize
llvm_class
long
mapping
memory
module
numways
open
overlap
pattern
pc
peephole
peepholes
pipeline
port
program
register
register_file
replace
reset
return
roots
schedule_class
semantics
set
settings
short
signal
signed
simulator
size
start
static
string
switch
symbol
testbench
type
typedef
typename
unsigned
use
void
while
write_enable
write_mask
```

11 ANNEX: TABLES, EXAMPLES AND FIGURES

11.1 List of Tables

11.2 List of Examples

11.3 List of Figures