CODAL LANGUAGE REFERENCE MANUAL

Version: 6.10.0

Release Date: November 14, 2017

COPYRIGHT AND PROPRIETARY NOTICE

INTEGRATED 3RD PARTY SOFTWARE MODULES AND THEIR LICENSES

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 Release Information

This is the first release of this document.

1.1.3 Product Revisions

Codasip Studio 6.10.0

1.1.4 Typographical Conventions

Table 1: Typographical conventions

Convention	Usage	Example
Capitalised	Standardized terms, defined earlier in the text or in the Glossary	Window, Project
Important	Important text	*Do not forget to ...*
Document ref	Reference to other Codasip and non-Codasip documents	Please refer to the *CodAL Language Reference Manual*.
`Code, filenames etc.`	Code, code values, Unix file names, prompts, etc.	File name `ca_utils.hcodal`
<*abstract name*>	Field for substitution with user data.	<*project*>/`model`
`keyword`	Inline references to CodAL keywords (lower case).	**`element`**, **`event`**

Convention	Usage	Example
IDE_word	Inline references to keywords of the IDE (usually starts in upper case)	The **Project Explorer** window
Option→Suboption	Command path, typically starting from the main toolbar.	**File → New → CodAL Project**
`Example`	Examples - typically snippets of code.	`register bit[DATA_W] test;`
`Syntax explained`	Explanation of syntax	`StartSection: "start" "{"`

1.2 References

1.2.1 Other Codasip Documents

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 ASIP models. The following documents describe this system:

- *Codasip Studio Quick Start Tutorial*
- 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 Studio Installation Guide	How to install the Codasip Studio software package.
Codasip Studio User Guide	Detailed guidance on the use of Codasip Studio and the tools that it contains.

Reference Manuals:

Document	Description
CodAL Language Reference Manual	A complete presentation of the CodAL language and how to use it for writing ASIP models.
Codasip Studio Technical Reference Manual	Reference information on Codasip Studio and the tools that it contains.

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

Tutorials:

Document	**Description**
Codasip Studio Quick Start Tutorial	A step-by-step introduction to the essentials of Codasip Studio.
Codasip Instruction Accurate Model Tutorial	A step-by-step introduction to writing Instruction Accurate ASIP models in CodAL.
Codasip Compiler Generation Tutorial	A step-by-step introduction to generating a C/C++ compiler from an Instruction Accurate ASIP model written in CodAL.
Codasip Cycle Accurate Model Tutorial	A step-by-step introduction to writing a Cycle Accurate CodAL model.
Codasip Interrupts and Peripherals Tutorial	A step-by-step introduction to adding external devices to an ASIP CodAL model.
Codasip JTAG Extension Tutorial	A step-by-step introduction to Codasip's JTAG extension.
Codasip SIMD Extension Tutorial	Tutorial showing the implementation of SIMD extensions in the Codasip uRISC
Codasip Custom Components Verification Tutorial	Tutorial desccribing proccess of adding manually modified UVM test-bench for a component into the ASIP or the top-level UVM test-bench.
Codasip uRISC VLIW Extension Tutorial	Tutorial showing modifications to Codasip uRISC to create a simple VLIW architecture.

1.2.2 Other References

There are no other references.

1.3 Feedback

1.3.1 Feedback on Codasip Products

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

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

1.3.2 Feedback on this Document

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

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

Codasip also welcomes general suggestions for additions and improvements.

2 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

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 CodALdescription have to contain information suitable for a range of tool generation tasks. For example, an object representing an ASIP 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 (pre-ordained 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:

- **`assembler`** - 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 **`assembler`** section. For example:

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

In this example, information from `r3ref`'s own **`assembler`** 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 assembler 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.assembler () or r3ref.binary() calls to clarify what information was sought. This conciseness is possible because CodAL is optimised 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 Design Level Description

CodAL has two aspects. The first is related to design level prototyping. The level describes the near environment around an ASIP or ASIPs. It may contain memories, third-party components, etc. The second concerns the ASIP itself. This separation makes ASIPs easy to reuse across multiple use cases.

A (top) design level is typically used to describe a near environment around ASIP(s). It may include memories, peripherals, and other parts, such as third party components, or even other (lower) design levels. The ASIP being designed is instantiated at a certain design level and connected to its external memory subsystem.

Design level and ASIP descriptions use the same type of project (**CodAL Project**), and this has number of advantages. All actions, such as the creation of simulators, are associated with the CodAL project, so it does not matter whether the project describes design level or ASIP.

The distinction also allows you to easily instantiate the ASIP or lower design level in multiple different use cases, and it provides a convenient mechanism for describing multi-processor systems.

3.3 ASIP Description

The ASIP description is managed separately from the top level description. Even though ASIP descriptions can vary in the terms of their architectural complexity and in the level of detail captured by the model, all of them share the same structure.

ASIP descriptions have 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 ASIP Architecture but which define a particular micro-architectural implementation.

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

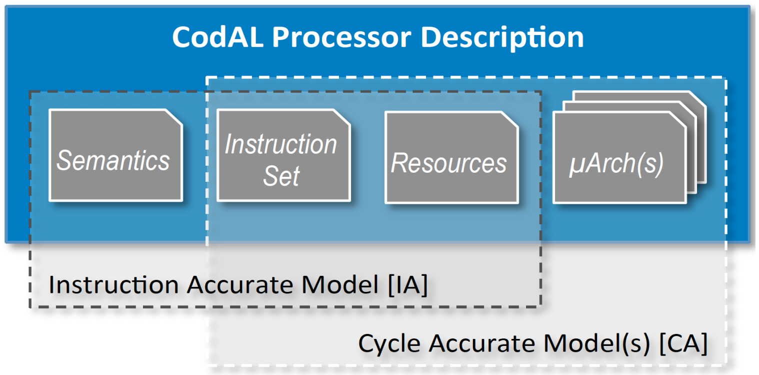
CodAL Processor Description
Semantics
Instruction Set
Resources
µArch(s)
Instruction Accurate Model [IA]
Cycle Accurate Model(s) [CA]

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

3.4 Common Language Constructs

Several language constructs that occur in both ASIP and design level descriptions are presented in the following subsections.

3.4.1 Top Level Constructs

The top-level syntax for a design level or ASIP description is as follows.

```
TranslationUnit
  : TranslationUnit TopConstruct
  | TopConstruct
```

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

```
TopConstruct
  : Resource ';'
  | Element ';'
  | Event ';'
  | Emulation ';'
  | Peephole ';'
  | Set ';'
  | Start ';'
  | ScheduleClass ';'
  | AnsiDeclaration
  | AnsiFunctionDefinition
  | Module ';'
  | Extern ';'
  | Connect ';'
  | SettingsCompound ';'
  | ';'
```

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

3.4.2 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
```

3.4.3 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("Hello World\n");
```

3.4.4 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
```

3.4.5 Compound Id

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

Example 4: CompoundId example

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

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

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

3.4.6 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
```

3.4.7 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 ']'
```

4 DESIGN LEVEL DESCRIPTION

As noted in the CodAL Basics section, any ASIP or Multi-Processor (MP) design may be instantiated in a design level (top level for instance). It contains instances of ASIPs, memories, buses and other components together with their interconnections.

See the following picture that shows two levels. One is TOP LEVEL that contains more instances of a LEVEL. The LEVEL consist of one ASIP and one hardware accelerator. Levels and ASIP are described using CodAL Projects. The hardware accelerator is described using Component Project.

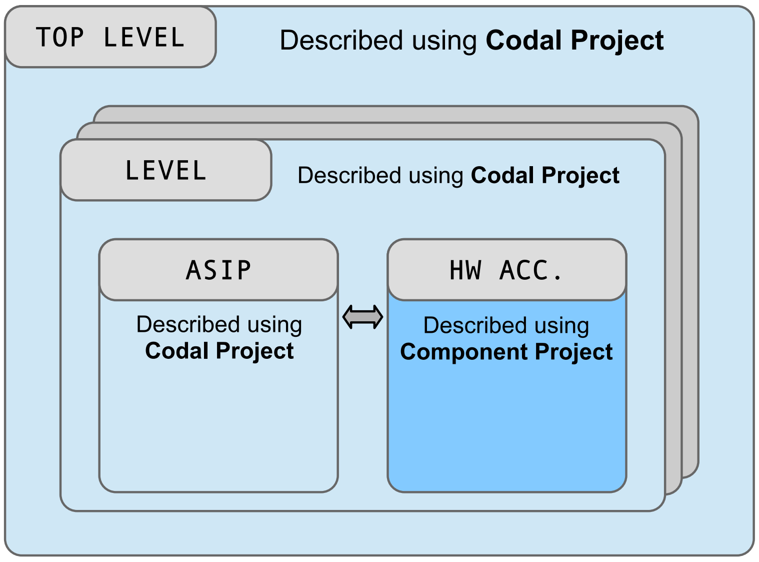
TOP LEVEL
Described using **Codal Project**
LEVEL
Described using **Codal Project**
ASIP
Described using **Codal Project**
HW ACC.
Described using **Component Project**

Figure 2 : Design levels

The files used by CodAL to describe design levels or ASIPs have .codal and .hcodal (**h**eader **codal**) extensions.

The objects that make up a design level 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 design level 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 ASIPs and components and a generated RTL description will contain descriptions of both ASIPs and components.

The motivation for the second case is that a designer can easily instantiate ASIPs 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 7: PIC Component (top.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_clb {
        bits = {32, 32, 8};
        endianness = BIG;
        type = CLB:SLAVE;
        flag = RW;
    };
};
```

In the next example you see instantiation of `codix_lrm` ASIP in an upper design level.

Example 8: ASIP instance (top.codal)

```
// instance of ASIP
extern codix_lrm
{
    interface if_fe
    {
        bits = {32, 32, 8};
        type = CLB:MASTER;
        flag = R;
        endianness = BIG;
    };
    interface if_ldst
    {
        bits = {32, 32, 8};
        type = CLB:MASTER;
        flag = RW;
        endianness = BIG;
    };
    interface if_simd_ldst
    {
        bits = {32, 32, 8};
        type = CLB:MASTER;
        flag = RW;
        endianness = BIG;
    };

    port bit[1]      p_irq       { direction = IN; };
    port bit[32]     p_input     { direction = IN; };
    port bit[1]      p_input_en  { direction = IN; };
    port bit[32]     p_output    { direction = OUT; };
    port bit[1]      p_output_en { direction = OUT; };
    port bit[1]      p_halt      { direction = OUT; };
    port bit[32]     p_error     { direction = OUT; };
};
```

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 6.

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

```
ExternAttribute
  : Interface
  | Port
```

`Port` is used for point-to-point connections. Ports are described in "Ports" on page 26.

`Interface` is used for memory or bus connections. Interfaces are described in "Interfaces" on page 27

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 8Kb (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 9: Memory instance (top.codal)

```
// instance of memory
memory mem {
    // memory size in LAU
    size = 0x10000;
    // only aligned access
    unaligned = false;
    // read/write latency
    latencies = {1, 1};

    // memory has two interfaces
    interface if_fe {
        // address, word size, LAU
        bits = {32, 32, 8};
        endianness = BIG;
        type = CLB:SLAVE;
        flag = R;
    };
    interface if_ldst {
        // address, word size, LAU
        bits = {32, 32, 8};
        endianness = BIG;
        type = CLB: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 6.

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

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

`Interface` is used for memory or bus connections. Interfaces are described in "Interfaces" on page 27

```
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 "CodAL Basics" on page 6.

```
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" '=' CompileExpression
```

`Size` defines size of memories or caches in LAU. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

```
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 "CodAL Basics" on page 6.

```
Unaligned
  : "unaligned" '=' CompileExpression
```

`Unaligned` specifies that a memory or cache supports unaligned accesses. Possible variants are **`true`** and **`false`**. The HDL generated for unaligned memory accesses is more complex than for aligned. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

This attribute is deprecated. Use **`alignment`** attribute in `Interface` instead.

4.3 Cache

Caches are used as fast memories close to the CPU. They may have many attributes including cache associativity or cache line size.

The following example describes a 4Kb instruction cache `l1_i` connected directly to a memory using an interface called `if_sdram`.

Example 10: Cache (top.codal)

```
// L1 instruction cache
cache l1_i
{
    // 8kiB
    size = 0x1000;
    latencies = {2, 2};
    // associativity is 4
    numways = 4;
    // line size in LAUs
    linesize = 16;
    rpl_policy = ROUND_ROBIN;
    // write through to peripheral
    non_cacheable =
    {
            0 .. 0x100
    };
    // interface from ASIP
    interface if_asip
    {
        endianness = BIG;
        bits = {32, 32, 8};
        type = CLB:SLAVE;
        flag = RW;
    };
    // interface to memory
    interface if_sdram
    {
        endianness = BIG;
        bits = {32, 128, 8};
        type = CLB:MASTER;
        flag = RW;
    };
};
```

There are some limitations in the current release of Codasip Studio:

- **`unaligned`** access is not supported - caches accept only aligned addresses.
- There must be exactly two interfaces. The first interface must be of type **`CLB:SLAVE`** and the second of type **`CLB:MASTER`**.
- The word bit size of the master interface must fit with the bit size of a cache line. The cache must be able to read or write the whole cache line using the master interface in only one request in the case of a miss/flush/replace.

- The read and write latency of an instruction cache is always 1 clock cycle in the case of a hit. For data cache it is always 2 clock cycles.

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

```
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 6.

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

```
CacheAttribute
  : Interface
  | Size
  | Linesize
  | Numways
  | RplPolicy
  | Latencies
  | NonCacheable
```

`Interface` is used for memory or bus connections. Interfaces are described in "Interfaces" on page 27

```
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 "CodAL Basics" on page 6.

```
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" '=' CompileExpression
```

`Size` defines size of memories or caches in LAU. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

```
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 "CodAL Basics" on page 6.

```
Linesize
  : "linesize" '=' CompileExpression
```

`LineSize` specifies a size of a cache line in LAU. Only powers of 2 are allowed. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

```
Numways
  : "numways" '=' CompileExpression
```

`NumWays` specifies the associativity of the cache. Only powers of 2 are allowed. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

```
RplPolicy
  : "rpl_policy" '=' Id
```

`RplPolicy` specifies the replacement policy. Possible variants are **`RR`** (round robin), **`LRU`** (least recently used), **`GRR`** (global round robin), **`PLRU`** (pseudo least recently used), and **`NRU`** (not recently used).

```
NonCacheable
  : "non_cacheable" '=' '{' NonCacheableBody '}'
```

```
NonCacheableBody
  : NonCacheableBody ',' NonCacheableRange
  | NonCacheableRange
```

```
NonCacheableRange
  : CompileExpression ".." CompileExpression
```

`NonCacheable` specifies regions which are not handled by cache. The requests are just passed to the upper memory using a connected interface. This is necessary when peripherals are used and are connected to the same bus as the main memory. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

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 Local Bus (CLB) 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 **`CLB`** 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 `clb` bus is defined. Its decoder has one slave. The `mem` memory is mapped to address range `0x10000` to `0x1ffff` and it is connected to the bus through the `if_ldst` interface. The `pic`component is mapped to address range `0x80` to `0x8f` from the relative slave address range `0x0 - 0xf`. Bus address `0x82` is therefore translated to slave relative adress `0x02`.

The width of the address bus is 32 bits and the whole range is connected to slaves. The ASIP instance `codix_lrm` is the only bus master. The word size is 32 bits and the LAU is 8 bits.

Example 11: CLB Bus (top.codal)

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

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

    interface if_ms_2
    {
        endianness = BIG;
        bits = {32, 32, 8};
        type = CLB: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 6 .

```
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 "CodAL Basics" on page 6.

```
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 "CodAL Basics" on page 6.

`CompoundId` refers to a slave and the slave interface used. The syntax of the `CompoundId` construct is described in "CodAL Basics" on page 6.

```
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 "CodAL Basics" on page 6.

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 an ASIP, 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` ASIP using ports. The ASIP 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 12: Connections (top.codal)

```
// connect memory ASIP's interfaces with memory's interfaces
connect codix_lrm.if_fe => mem.if_fe;
connect codix_lrm.if_ldst => mem.if_ldst;
// connect PIC
connect pic.p_irq_out => codix_lrm.p_irq;
// open ports
connect codix_lrm.p_halt => open;
// constant ports
connect 0 => codix_lrm.p_input;
connect 0 => codix_lrm.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 "CodAL Basics" on page 6.

`CompileExpression` denotes a value that is driven to the input port. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

5 PORTS AND INTERFACES DESCRIPTION

Ports as well as interfaces are essential for communication. Using interfaces, a designer may connect ASIPs with memories or buses. Using ports, a designer may connect components and ASIPs in a point-to-point manner.

5.1 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 many places: in ASIPs, components and design levels. 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 13: ASIPCAPTION#39;s ports (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 '{' PortDirection '}'
```

`[CompileExpression]` specifies the bit-width of the port. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

`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 6.

```
PortDirection
  : "direction" '=' Id ';'
```

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

5.2 Interfaces

Interface is mainly used for bus and memory connections. It allows fast prototyping of an ASIP or design level, 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:

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

The protocol of `MEMORY` is the same as the protocol of a BRAM on an FPGA.

The protocol of `CLB` is described in the *Codasip Studio Technical Reference Manual* , chapter "Codasip Local Bus".

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 pre-defined 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 14: Interfaces (interface.codal)

```
// Instruction Bus - FE/DE stage
interface if_fe
{
    bits = { ADDR_W, WORD_W, LAU_W };   // address, word size, LAU
    type = CLB:MASTER;
    flag = R;
    endianness = BIG;
    alignment {
        address = 32;
        data = {32};
    };
};
```

```
// Data Bus - EX/EX2 stage
interface if_ldst
{
    bits = { ADDR_W, WORD_W, LAU_W };   // address, word size, LAU
    type = CLB:MASTER;
    flag = RW;
    endianness = BIG;
    alignment {
        address = 8;
        data = {8, 16, 32};
    };
};
```

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 CLB 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 6.

```
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 "CodAL Basics" on page 6.

```
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 address should be aligned to. **`data`** is a list of values that specify the bitwidth of values which can be accessed in a single transaction. For example if **`data`** alignment is `{8, 32}`, then only 32-bit and 8-bit words can be read, but reading 16-bit words is not allowed.

All alignment values should be multiples of the LAU bitwidth.

5.2.1 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 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);`. Bytes (LAUs) can be accessed with `if_ldst.write(10, addr, 0, 1);`. The first index denotes an offset from the addr in bytes (LAUs), so called *byte index*, it is `0` in this example. The second index denotes the count of bytes (LAUs), so called *byte count*, `1` in this example.

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

- `resource_t read(const codasip_address_t address);`
- `interface_t read(const codasip_address_t address, const unsigned bi, const unsigned bc);`
- `void write(const interface_t data, const codasip_address_t address);`
- `void write(const interface_t data, const codasip_address_t address, const unsigned bi, const unsigned bc);`

To have cycle accurate access, the designer must use the request and finish functions. In other words, the read or write accesses are split into two phases: an address/request phase and a data phase (in the way hardware usually works). The request is performed by `if_ldst.request(CP_CMD_WRITE, addr);`. The data is sent with the `finish` function, `if_ldst.finish(CP_CMD_WRITE, ores);`, where `ores` represents a resource holding data for the write transaction (tipycally a register, a signal, or a local variable). A similar protocol is used for a read. The functions are `if_ldst.request(CP_CMD_READ, addr);` and `if_ldst.finish(CP_CMD_READ, ires);`, where `ires` is a resources that is written to when the data phase finishes (typically a register, a signal, or a local variable).

The cycle-accurate access possibilities are:

- `status_t request(const command_t cmd, const codasip_address_t address, const unsigned bi, const unsigned bc);`
- `response_t finish(const command_t cmd, resource res);`
- `response_t finish(const command_t cmd, resource ires, resource ores);`

Note that **`bus`** and **`memory`** can also be accessed in a cycle-accurate style.

Request function returns a status code (`status_t`) for the destination (cache, memory or bus slave). The user can use these built-in constants in the interface description:

- **`CP_ST_READY`**, destination is ready
- **`CP_ST_BUSY`**, destination is busy
- **`CP_ST_ERROR`**, destination is in an error state

Finish function returns a response code (`response_t`) from the destination (cache, memory or bus slave). Users can use these built-in constants in the interface description:

- **`CP_RS_IDLE`**, slave is in the idle mode
- **`CP_RS_WAIT`**, data/slave is not ready
- **`CP_RS_ACK`**, data/slave is ready
- **`CP_RS_OOR`**, address is out of range
- **`CP_RS_UNALIGNED`**, address is not aligned and memory supports only aligned access
- **`CP_RS_ERROR`**, an error occurred

The `request` as well as `finish` function needs a command that they should request/finish as the first parameter (`command_t`). The following constants are built-in and can be used within a model:

- **`CP_CMD_NONE`**, no request
- **`CP_CMD_READ`**, read request
- **`CP_CMD_WRITE`**, write request
- **`CP_CMD_INVALIDATE`**, invalidate request (cache)
- **`CP_CMD_INVALIDATE_ALL`**, invalidate request (cache)
- **`CP_CMD_FLUSH`**, flush request (cache)
- **`CP_CMD_FLUSH_ALL`**, flush request (cache)

`finish` function has two forms. The first one takes a resource (*resource*) as the second argument. It may be a register, a signal or a local variable. This form is useful for MEMORY interface or CLB interface that is read or write only.

The second form of `finish` function takes two resources. One for read commands and one for write commands. This form is useful only for CLB RW interfaces. It removes a need for a two `finish` functions in a model (one for reading, one for writing). The first argument should be a register holding requested command.

6 ASIP DESCRIPTION

In the following subsections, the use of CodAL to describe ASIPs is organized according the four ASIP parts presented in "CodAL Basics" on page 6 (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 ASIPs have .codal and .hcodal (**h**eader **codal**) extensions.

6.1 Resources

In this section we describe how to define certain structural components of the processor- e.g. registers. An ASIP can contain the following resources:

- *Address space* defines a memory view of the ASIP 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.

The following sections will describe each of these resources with the exception of ports and interfaces, already described in "Ports and Interfaces Description" on page 26.

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

```
Resource
  : AddressSpace
  | Pipeline
  | Register
  | RegisterFile
  | Signal
  | Interface
  | Port
  | Memory
  | Cache
  | Bus
```

6.1.1 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. Just one program counter must 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.

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 doesn't 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 ASIP 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.

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 15: Registers (arch.codal)

```
// program counter, address of bootloader is taked 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];
};
```

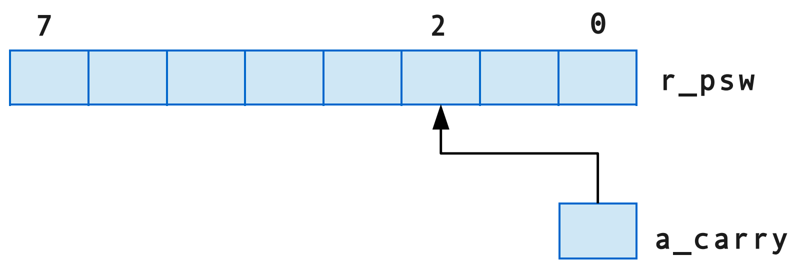

Figure 3: Register alias

The next example shows pipeline registers.

Example 16: Pipeline registers (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 syntax of the `Register` construct is as follows.

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

`[CompileExpression]` specifies the bit-width of the register. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

`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 6.

```
Specifier
  : "arch"
  | "pc"
  | "alias"
  | %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" '=' CompileExpression
  | "pipeline" '=' CompoundId
  | Overlap
  | Dff
  | Reset
  | ClockEnable
```

`reset` enables or disables the reset feature.

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

`default` denotes the value that is assigned to the register during reset or clear.

`pipeline` declares a pipeline register and assigns a pipeline stage.

`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 ']'
```

`CompoundId` is a target identifier.The syntax of the `CompoundId` construct is described in "CodAL Basics" on page 6.

`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 "CodAL Basics" on page 6.

`Dff` defines behavior of a register when an instruction accurate simulator is generated. By default, the register behaves as a variable in the instruction accurate simulator. In some cases, it is useful to have a register that behaves as a D flip-flop (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.

`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.

6.1.2 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`**.

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 (arch.codal)

```
// General Purpose Registers
arch register_file bit[WORD_W] rf_gp32
{
    size = RF_GP_SIZE;
    dataports = {RF_GP_RP, RF_GP_WP};
};
// 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];
};
```

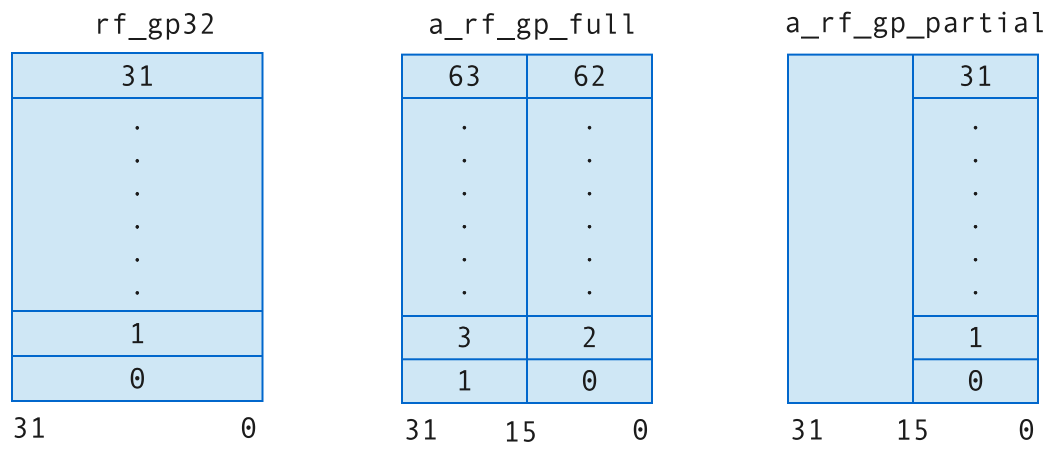

Figure 4: Register file alias

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

```
RegisterFile
  : Specifier "registerfile" "bit" '[' CompileExpression ']' IdList '{' RegisterFileBody '}'
```

`[CompileExpression]` specifies the bit-width of the register file. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

`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 6.

```
Specifier
  : "arch"
  | "pc"
  | "alias"
  | %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
  | "mask" '=' CompileExpression
  | "dataports" '=' '{' CompileExpression ',' CompileExpression '}'
  | "default" '=' CompileExpression
  | Overlap
  | Dff
  | Reset
  | ClockEnable
```

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

`mask` denotes whether a register file supports a masked write.

`dataports` defines a number of read and write data ports.

`default` denotes a value that is assigned to all members during reset.

`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 ']'
```

`CompoundId` is a target identifier.The syntax of the `CompoundId` construct is described in "CodAL Basics" on page 6.

`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 "CodAL Basics" on page 6.

`Dff` defines behavior of a register when an instruction accurate simulator is generated. By default, the register behaves as a variable in the instruction accurate simulator. In some cases, it is useful to have a register that behaves as a D flip-flop (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.

`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.

6.1.3 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.

The following example shows two signals:

Example 18: Signals (ca_resources.codal)

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

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

```
Signal
  : "signal" "bit" '[' CompileExpression ']' IdList
```

`[CompileExpression]` specifies the bit-width of the signal. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

`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 6.

6.1.4 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 ASIP 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 address 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. Each bounded interface can have an address interval that is taken into account when a particular address is accessed. In other words, it specifies addresses which are handled by the interface (see Figure 5).

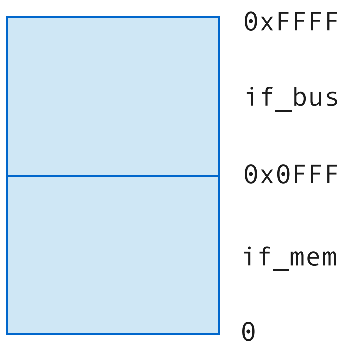

Figure 5: Interface mapping within the address space

When an address range is omitted, then the all address are handled by a bounded interface. If there are more interfaces and addresses are overlapping, 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 19: Von Neumann Address space (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 20: Harvard Address spaces (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
  : "addressspace" Id '{' AddressSpaceBody '}'
```

`Id` is an identifier naming a construct. The syntax of the `Id` construct is described in "CodAL Basics" on page 6 .

```
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 "CodAL Basics" on page 6.

```
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
```

`CompileExpression`s are used for address range specification. The syntax of the `CompileExpression` construct is described in "CodAL Basics" on page 6.

`CompoundId` denotes a bounded interface.The syntax of the `CompoundId` construct is described in "CodAL Basics" on page 6.

6.1.5 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 21: Pipeline (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 6 .

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

`Id` names a pipeline stage. The syntax of the `Id` construct is described in "CodAL Basics" on page 6 .

6.2 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 *cluster*s.

Figure 6 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`**, **`timing`** 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.

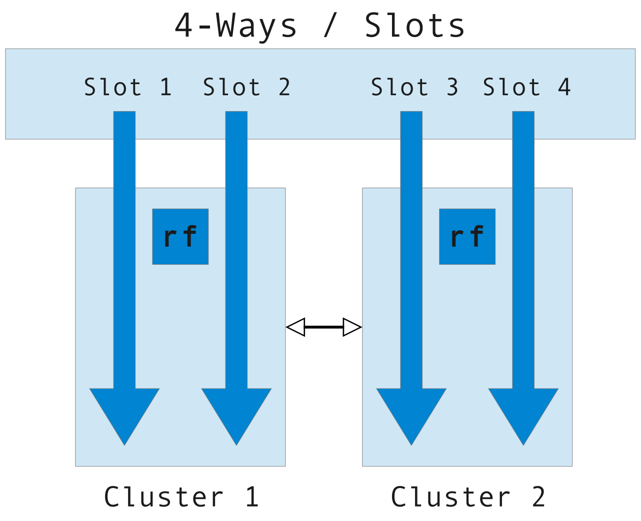

Figure 6: Cluster scheme

Although the modules use different resources, their names inside a **semantics**, **timing** 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 22: Modules declaration (ca_resources.codal)

```
// Modules
module m_0, m_1
{
    // general purpose registers
    register_file bit[16] rf_gp
    {
        size = 16;
        dataports = {3, 1};
    };
    // 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 23: Semantics with modules (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** and **decoders** 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 24: Using modules in the use section (ca_pipe_stage1_id.codal)

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

    decoders
    {
        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 '{' 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 6.

```
ModuleBody
  : ModuleBody TopConstruct
  | TopConstruct
```

`TopConstruct` stands for a subset of all supported constructs that can be used within a module. See "ASIP Description" on page 32.

6.3 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 an ASIP, as shown in the following figure (`el` is abbreviation for **`element`**):

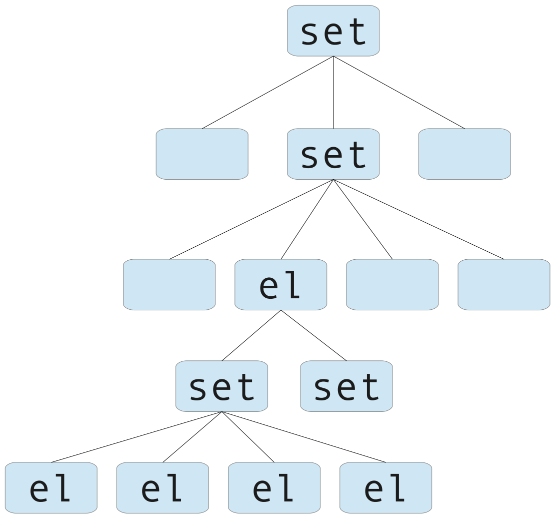

Figure 7: Instruction set description

The root (or roots in the case of VLIW) of the instruction set are specified in a **`start`** section (see "Start Section" on page 48).

The assembler syntax is defined in an **`assembler`** part of **`settings`** (see "ASIP Description" on page 32).

6.3.1 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 ASIP 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 section, 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 65.

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 71.

The following example shows a single instruction set definition.

Example 25: Start section (isa.codal)

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

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

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
```

Were `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.

6.3.2 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`**, **`assembler`** and **`binary`**. The **`use`** sections instantiate other **`element`**s and **`set`**s, and also **`event`**s. The **`assembler`** 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 `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 `COMPILER`, `PROFILER`, `RANDOMGEN`, or `DOCUMENTATION`.
- `register_operand(CompoundId)` - If used, the **`element`** or **`set`** describes a 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`.

- `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 62 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 ASIP itself. Unlike of assembler aliases, semantics may be assigned. See "Pseudo Instructions" on page 62 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 79.
- **`return`** - defines a return value when the **`element`** is used. See "Return Section" on page 81.
- **`timing`** - defines activated **`event`**s in the micro-architecture associated with the **`element`**. See "Timing Section" on page 93.

`Note that the timing section is deprecated.`

Moreover, any element can have a local resources that may be used to pass information from one section to another (e.g. from **`semantics`** to **`timing`**).

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

Example 26: Simple element (isa_operands.codal)

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

    assembler { val };
    binary { val };
    return { val; };
};
```

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

Example 27: Complex element (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
    assembler { 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 28: Assembler alias (isa.codal)

```
element i_nop_alias : assembler_alias(i_3_reg_operands)
{
    assembler { "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
  | SectionAssembler ';'
  | SectionBinary ';'
  | SectionSemantics ';'
  | SectionReturn ';'
  | SectionTiming ';'
```

```
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 6 .

```
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 "CodAL Basics" on page 6.

`String` is used as an argument to the `profiler()` property. The syntax of the String construct is described in "CodAL Basics" on page 6

6.3.2.1 Use Section

The **`use`** section contains instance declarations of other **`element`**s, **`set`**s, or **`event`**s (see "Implementation" on page 91 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 29: Use sections (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 6 .

```
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 6 .

6.3.2.2 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 30: Attributes (isa_operands.codal)

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

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

    assembler { 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 31: Attributes semantics action (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;
    };

    assembler { 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 semantic action within an element for such an instruction will look like this:

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 32: Attributes semantics action (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 = (val << 2) + current_address;
    };

    assembler { val };
    binary { val };
    return { val; };
};
```

This example shows an assembler section for an **element**that describes the address of a jump instruction. It is a signed, 14-bit number. When assembling, the assembler will take the input value, subtract the current absolute address, shift to the right by two and then store the result value as the binary operand.

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 6 .

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

`AttribteSpecisier` denotes signess of an attribute

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

`AttributeBit` specifies 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
```

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

`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, whre `<text>` is an identifier of the user-defined base (e.g. `my_dec`).

An attribute may have several attributes.

6.3.2.3 Assembler Section

An **`assembler`** section defines a textual representation of an **`element`**. The assembler section consists of instances, text constants and attributes. Every assembler 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 assembler 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.

6.3.2.3.1 Syntax

The syntax of the `SectionAssembler` construct is as follows.

```
SectionAssembler
  : "assembler" '{' SectionAssemblerBody '}'
```

```
SectionAssemblerBody
  : SectionAssemblerBody SectionAssemblerElement
  | SectionAssemblerElement
```

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

6.3.2.3.2 Instances

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

6.3.2.3.3 Text Constants

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

6.3.2.3.4 Attributes

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

The attributes are described in "Attributes" on page 54 in greater detail.

6.3.2.4 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.

6.3.2.4.1 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.

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

Example 33: Instance split (isa.codal)

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

    assembler { 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 8.

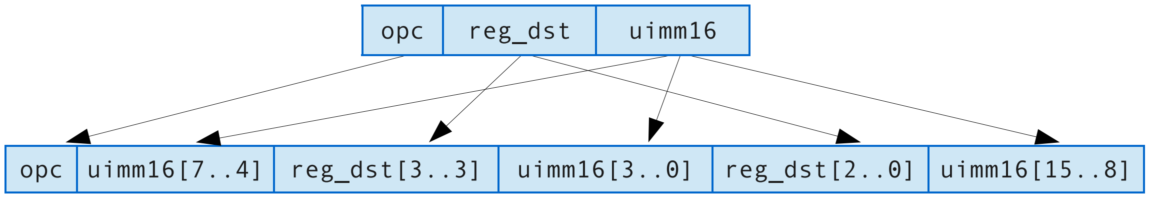

Figure 8: Instance split

6.3.2.4.3 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 34: Binary constants (isa.codal)

```
enum : uint5
{
    OPC_ENUM   = 0x10,
    OPC_ENUM_2 = 0x2
};

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

6.3.2.4.4 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 60).

The attributes are described in "Attributes" on page 54 in greater detail.

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

Example 35: Unnamed attribute (isa.codal)

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

6.3.2.5 Conditional Sections

Sections within an element can 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 36: Conditional sections (isa.codal)

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

    assembler { "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`** variant. Each variant within the **`switch`** must have a **`break`** statement. The **`binary`** section must be included in all variants and it must be unique for each variant. Only the following sections can be conditional: **`binary`**, **`semantics`**, **`return`** and **`timing`**. If the section is declared outside of the **`if`** statement body, then it is used in all variants of the binary encoding (in this way two instructions can have the same **`timing`** 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 '}'
```

6.3.2.6 Pseudo Instructions

Besides the real instructions, an ASIP 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`.

6.3.2.6.1 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*.

6.3.2.6.2 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*.

6.3.3 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 `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 `COMPILER`, `PROFILER`, `RANDOMGEN`, or `DOCUMENTATION`.
- `register_operand(CompoundId)` - If used, the **`element`** or **`set`** describes a 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`.

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 37: Register operands (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
{
    assembler { "r0" };
    binary { 0:bit[2] };
    return { 0; };
};
element reg1
```

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

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

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

Example 38: Defining a set of instructions (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 an ASIP.

Example 39: Top level set (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 6 .

```
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
```

`Id` is an identifier of a `set` member. The syntax of the `Id` construct is described in "CodAL Basics" on page 6 .

```
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 6 .

```
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 "CodAL Basics" on page 6.

`String` is used as an argument to the `profiler()` property. The syntax of the String construct is described in "CodAL Basics" on page 6

6.3.4 Emulations

Emulation 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 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 32bit constant load using two instructions. One instruction is used for a loading of an upper 16bits of the constant and the second one is used for a loading the lower part of the constant.

Example 40: Emulation (isa.codal)

```
element i_lui_16b
{
    use gpreg as reg_dst;
    unsigned attribute bit[16] val;

    assembler { "lui" reg_dst "," val };
    binary { 0b0000:bit[4] reg_dst val };

    // ...
};
element i_ori_16b
{
    use gpreg as reg_dst;
    unsigned attribute bit[16] val;

    assembler { "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 6 .

```
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 6 .

```
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 79 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`.

6.3.5 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 41: Peephole pattern

```
element i_load32
{
	use reg32_any as reg32_dst;
	use reg64_any as reg64_base;
	use simm12 as offset;

	assembler { "load" reg32_dst "," offset "(" reg64_base ")" };

	// ...
};

element i_zero_ext
{
	use reg32_any as reg32_src;
	use reg64_any as reg64_dst;

	assembler { "zex" reg64_dst "," reg32_src };

	// ...
};

element i_load64
{
	use reg64_any as reg64_dst;
	use reg64_any as reg64_base;
	use simm12 as offset;

	assembler { "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 6 .

```
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 patern.

```
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 alsot 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.

6.3.6 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 represents encoded instructions of each slot.

Example 42: 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 32bit instruction. **bundling_int32** works similarly to **uint32**, but do not support all operations, see a type documentation for details.

Example 43: Structure for the 32bit 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 44: 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 45: Debundle function

```
debundle128 fdebundle(const uint128)
{
    ...
}
```

The debundle type (for example debundle128) is the 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 46: Structure for the 128bit debundle

```
struct debundle128
{
    uint128 value;
    uint32 input_size;
};
```

6.3.7 Settings

Settings allows to set different attributes for tools in SDK. Namely, there is a section for assembler, C/C++ compiler, and debugger. The syntax is the same for all kinds of sections. The `SettingsCompound` is a top level rule that encapsulates all settings. If the model of ASIP contains more that one `SettingsCompound` contruct 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"
```

The `SettingSpecifier` construct specifies a SDK tool. There are four options now:

- **`compiler`** - Settings for C/C++ compiler,
- **`debugger`** - Settings for on-chip and software debugger,
- **`simulator`** - Settings for simulator,
- **`assembler`** - Settings for assembler.

```
SettingsBody
  : SettingsBody Setting ';'
  | Setting ';'
```

The `Settings` construct consist of `Setting` construct that describes one singe setting/option for a SDK tool.

```
Setting
  : Id '=' SettingElement
  | Id SetAssign '{' SettingBoby OptionalComma '}'
```

The `Id` specifies attribute/option of a SDK tool. For instance, a register that holds a return address of a function.

```
SettingBoby
  : SettingBoby ',' 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 47: Settings Example

```
settings
{
  compiler
  {
    function_result = { rf_gpr[2], rf_gpr[3], rf_gpr[4], rf_gpr[5] };

    // short vesrion
    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 describes attributes/options that you can set for SDK tools.

6.3.7.1 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. There are several options:

Numeric Attributes

- **`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.

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 defaultly used for certain type. Allowed only once per type.

Data Types 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. Allowed only once in the model.
- `datalayout` - a string. The string is the same as the standard LLVM datalayout (see LangRef.html).

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 1bit register. Carry status flag register. Allowed only once in the model.
- `overflow_flag` - a 1bit register. Overflow status flag register. Allowed only once in the model.
- `sign_flag` - a 1bit register. Sign status flag register. Allowed only once in the model.
- `zero_flag` - a 1bit 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. Allowed only once in the model.
- `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. Allowed only once in the model.
- `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. Allowed only once in the model.
- `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. Allowed only once in the model.

- `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. Allowed only once in the model.

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.

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, which will be defined for every program compiled by the generated C/C++ compiler.

6.3.7.2 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. There are several options:

- `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, tells to the on-chip debugger to be active or not (high means active).
- `hw_break_request` - a signal, it holds a hardware break request from the software debugger to the on-chip debugger (high means a valid request).
- `sw_break_request` - a signal, it holds a software break request from the on-chip debugger to the software (high means a valid request).
- `syscall` - a register, a register in a register file or an interface with an address that holds an address of a syscall pyload. Note that this setting is applied only on the cycle-accurate models and it requries JTAG.

6.3.7.3 Simulator Settings

No attribute/option is allowed here at the moment.

6.3.7.4 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`** and the default is **`P2ALIGN`**.
- **`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.

6.3.8 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 48: Schedule class (isa.codal)

```
schedule_class loads
{
    latency = 2;
};
```

The syntax of the `ScheduleClass` construct is as follows.

```
ScheduleClass
  : "scheduleclass" 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
```

6.4 Semantics

The semantic, or behavioral, part of a CodAL model is to be found in its **`semantics`** and **`return`** sections.

6.4.1 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 initialize 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.

The following example shows the definition of an ALU unit:

Example 49: Semantics section (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 50: Resent event (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 "Timing Section" on page 93), the decoder activation has one or two arguments (see the "Decoders Section." on page 94).

The syntax of the `SectionSemantics` construct is as follows.

```
SectionSemantics
  : "semantics" AnsiCompoundStatement
```

6.4.2 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 79. 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 51: Return section of opcodes (isa_operands.codal)

```
element opc_add
{
    assembler { "add" };
    binary { OPC_ADD };
    return { OPC_ADD; };
};

element opc_sub
{
    assembler { "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;

    assembler { 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 52: Return section of immediate operand (isa_operands.codal)

```
// 5bit Signed Immediate Value
element simm5
{
    signed attribute bit[5] val;

    assembler { val };
    binary { val };
    return { val; };
};
```

The syntax of the `SectionReturn` construct is as follows.

```
SectionReturn
  : "return" '{' AnsiExpression OptionalSemicolon '}'
```

6.4.3 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 53: Function definition (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

6.4.4 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 2: 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 2 can be preceded with qualifiers (`signed`, `unsigned`, `const`) to change the default signedness or add special attributes. 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 3: Extended CodAL basic data types

Type	Bit-width	Signedness	Description
int**N**	N	signed	Bit-width of signed CodAL integer is limited to 2-1024.
int_<**E**>	E	signed	Bit-width is denoted by a compile expression **E**. **E** is limited to 2-1024.
uint**N**	N	unsigned	Bit-width of signed CodAL integer is limited to 1-1024.
uint_<**E**>	E	unsigned	Bit-width is denoted by a compile expression **E**. **E** is limited to 1-1024.
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 3 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 4: CodAL vector data types

Type	Bit-width	Signedness	Description
v**E**i**N**	E*N	signed	Vector with signed integer elements.
v_i_<**Y**, **Z**>	Y*Z	signed	Vector with signed integer elements, where **Y** and **Z** are compile time expressions. **Y** denotes the bit-width of an integer element, **Z** denotes a number of elements.
v**E**u**N**	E*N	unsigned	Vector with unsigned integer elements.
v_u_<**Y**, **Z**>	Y*Z	unsigned	Vector with unsigned integer elements, where **Y** and **Z** are compile time expressions. **Y** denotes the bit-width of an integer element, **Z** denotes a number of elements.
v**E**f**N**	E*N	signed	Vector with floating point elements.

Table 4 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.

6.4.5 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 54: Enumerators (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.

6.4.6 CodAL Operators

CodAL supports all ANSI C operators except pointer operators (`*`, `&`) and the `sizeof` operator. Moreover the language adds a bit-select operator `[..]`. Table 5 summarises all supported operators, their precedence and associativity.

Table 5: 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.	
2	`++`	Prefix increment.	Right to left.
	`--`	Prefix decrement.	
	`clog2`	Bit width of constants.	
	`bitsizeof`	Bit size of types and resources.	
	`+`	Unary plus.	
	`-`	Unary minus.	
	`!`	Logical NOT.	
	`~`	Bitwise NOT.	
	`(type)`	Type cast.	

Precedence	Operator	Description	Associativity
3	`*`	Multiplication.	Left to right.
	`/`	Division.	
	`%`	Remainder.	
	`**`	Power	
	`::*`	Replication	
4	`+`	Binary addition.	
	`-`	Binary subtraction.	
	`::`	Concatenation	
5	`<<`	Left shift.	
	`>>`	Right shift.	
	`<<<`	Rotation left	
	`>>>`	Rotation 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.

6.4.6.1 Bit-select Operator

CodAL defines an operator for bit selection from an expression. The syntax of the operator is:

Example 55: 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 56: 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 `lsb`,

- `msb` and `lsb` must be constant positive integers,
- `msb` and `lsb` must be in range of bit-width of `expr`.

6.4.7 Implicit conversions

Implicit conversions used in CodAL are different from standard ANSI C, as they have been adapted for specifying ASIPs. Compared to ANSI C, there are no exceptions and rules are simpler to remember. Table 6 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 6: 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	Builtin 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 7: 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 6.

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.

6.4.8 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 sufixes is arbitrary. 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 57: Specification of nonstandard bit-width of the integer literal.

```
uint9 d = (uint9)42u;
int65 x = (int65)0x1FFFFFFFFFFFFFFFFll;
```

6.5 Implementation

The implementation part of the ASIP model defines micro-architectural timing and optimal instruction decoding. It allows us to generate a cycle-accurate simulator and RTL, from which real hardware may be synthesised. The main component of the hardware concerned by the implementation part of the ASIP 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 **`decoders`**, and the relationships between them. Each pipeline stage has a set of assigned **`event`**s or **`decoders`**, as shown in Figure 9 .

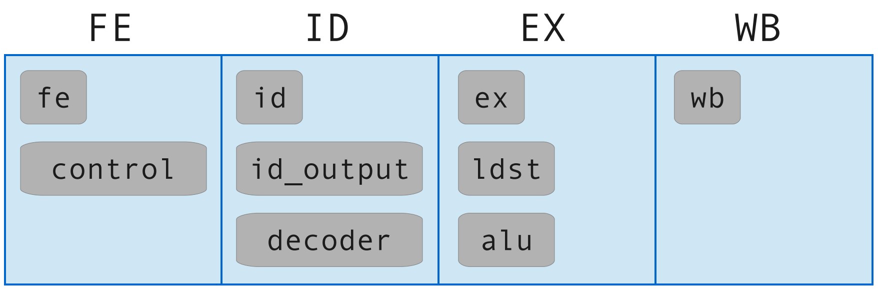

Figure 9: Pipeline stages and events

6.5.1 Event

`event` is a basic construct that is used in the implementation part of the ASIP model. Its functionality is captured in its **`semantics`** section. Each **`event`** can activate other **`event`**s and **`decoder`**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 three sections. The order of execution within a simulation is the same is in the following list:

- **semantics** - defines the semantics of the **event**. See "Semantics Section" on page 79.
- **decoders** - defines the activated decoders. See "Decoders Section." on page 94.
- **timing** - defines other **event**s that are activated when the **element** is used. See "Timing Section" on page 93.

Note that the decoders and timing sections are deprecated.

Moreover, an event can have a local resources. The local resources can be used for advanced RTL optimizations (e.g. re-timing).

In addition, the **start** section is used in the main **event**, as described in "Start Section" on page 48.

The following example shows an **event**, fe, used for instruction fetching:

Example 58: Event description for the fetch stage (ca_pipe_stage0_fe.codal)

```
event fe : pipeline(pipe.FE)
{
    semantics
    {
        // --------------------------------------------------------------------
        // Instruction fetch request
        if_fe.request(CP_CMD_READ, r_pc);

        // --------------------------------------------------------------------
        // 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 ';'
  | SectionTiming ';'
  | SectionDecoders ';'
```

The **use** section in an **event** definition is identical to that in an **element** definition and is described in "Use Section" on page 53.

6.5.1.1 Timing Section

This section is deprecated. You can activate other events directly in the semantics section.

The **timing** section becomes important when we want to model a processor on a cycle-accurate level. Roughly speaking, it contains a list of **event**s that should be activated when the current **timing** section is handled. In the case of simulation, these **event**s are activated in the same order as listed in this section. When realized in 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 without any side effect. The expression can contain a test or a resource value or something like this.

The following example illustrates the **timing** section. The `ex` **event** is assigned to the `pipe.EX` stage. The `cond_compare` and `ex_output` **event**s are assigned to the same stage, so they 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 59: Timing section (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;

    timing
    {
        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();
    };
};
```

The syntax of the `SectionTiming` construct is as follows.

```
SectionTiming
  : "timing" '{' AnsiStatementList '}'
```

`AnsiStatementList` can contain statements with **`event`** activations and selection statements (i.e. **`if`**s and **`switch`**s).

6.5.1.2 Decoders Section.

`This section is deprecated. You can activate instruction decoders directly in the semantics section.`

The **`decoders`** section describes instruction decoders. In contrast to the **`start`** section (which affects only the programming tools), this section has a direct effect on the created hardware and simulation tools. The **`decoders`** section can be placed in one or more **`event`**s. 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 decoder is essentially an **`event`** that 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 60: Decoders activation (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
    decoders
    {
        instr_opcode(s_id_opcode, s_iis_1);
        if (s_id_alu)
        {
            instr_alu(s_id_alu, s_iis_2);
        }
    };
};
```

The syntax of the `SectionDecoders` is as follows.

```
SectionDecoders
  : "decoders" '{' AnsiStatementList '}'
```

`AnsiStatementList` can contain statements with decoders activations and selection statements (i.e. **`if`**s and **`switch`**s).

6.5.1.3 Mandatory events

Each ASIP 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`** - It is activated each clock cycle and it is a top level **`event`** that should contain **`start`** section. It the case of an IA model, it should also contain a **`decoders`** section. In the case of a CA model, it should contain a **`timing`** section that activates **`event`**s within a pipeline.

7 USING BUILTINS IN CODAL MODELS

Builtins are functions that can be included in CodAL descriptions in order to guide one or more of the tools using that description. They affect assembler, disassembler, simulator and compiler generation in ways appropriate to each of the tools.

7.1 Debug Functions

7.1.1 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.

7.1.1.1 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.

7.1.1.2 Return Value

None.

7.1.1.3 Example

Example 61: codasip_assert function

```
semantics
{
    ...
    codasip_assert((r_pc % INSTR_LAU_SIZE) == 0, "PC is unaligned, PC = %d\n", r_pc);
    ...
};
```

7.1.2 codasip_disassembler

Semantic category - simulator

Supported on - IA, CA

`void codasip_disassembler(uint32 verbosity, uint1024 input, uint64 address, text isa)`

Disassemble input data and print the decoded instruction to the output. The print is done in `codasip_print()` way, so no prefix is printed nor the end of the line. You may use it for a pipeline debugging as well (meaning that you can call `codasip_disassembler()` in each pipeline stage to see which instruction if there).

7.1.2.1 Parameters

- input - Input data. The data should be aligned to MSB.
- address (optional) - Address of the decoded instruction (default: 0). It is needed for the correct computation when the decoded instruction uses **`current_address`** within attribute's **`decoding`** section.
- start (optional) - Name of the **`start`** section (default: "default").

7.1.2.2 Return Value

No return value.

7.1.2.3 Example

Example 62: codasip_dissasembler function

```
codasip_print("ID: ");
codasip_disassembler(r_ir, r_pc);
codasip_print("\n");
```

7.1.3 codasip_error

Semantic category - general

Supported on - IA, CA

`void codasip_error(uint32 verbosity, text format, ...)`

Write formatted data to the simulator output. Suitable for error messages that won't stop simulation.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

7.1.3.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

```
%[flags][width][.precision]specifier
```

Table 8: 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 9: 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 63: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

7.1.3.2 Parameters

verbosity

Specifies the message group for simulator output. Must be an unsigned integer value.

`Note: This behaviour is not currently implemented in the simulator.`

format

A C string containing the text to be written to the output.

The output of `codasip_error` function is in the format:

```
error (<verbosity>):  <ASIP  name>@<clock  cycle>:  <formatted
text>\n
```

7.1.3.3 Return Value

None.

7.1.3.4 Example

Example 64: codasip_error function

```
#define ERROR_LEVEL       5

codasip_error(ERROR_LEVEL, "Unknown operation in i_cache_hw!");
```

Example 65: codasip_error function output

```
error(5): codasip_urisc@2500: Unknown operation in i_cache_hw!
```

7.1.4 codasip_fatal

Semantic category - general

Supported on - IA, CA

```
void codasip_fatal(uint32 exit_code, text format, ...)
```

Write formatted data to the simulator output and terminate the simulation.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

7.1.4.1 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 10: Codasip format flags

`flags`	Description
-	Left-justify when specifier is padded to given width. Right justification is the default.
#	Show numeric base prefix - 0b for binary, 0 for octal, 0x for hexadecimal. Valid only for numeric specifiers.
0	Use character '0' for padding to given width. By default padding is performed by space.

The `width` specifies minimum number of characters to be printed for the specifier. If the formatted value including optional base prefix is shorter, it is padded according to `flags` into the minimum width. If the formatted value is longer, no change is performed.

The `precision` specifies minimum number of digits to be printed after the decimal point. This setting is used for floating point specifier only.

Table 11: Codasip format specifiers

`specifier`	Description	Example
d	Decimal integer	123
u	Unsigned decimal integer (deprecated). Use unsigned integer as argument	123

b	Binary integer	1111011
o	Octal integer	173
x	Hexadecimal integer	7b
X	Hexadecimal integer, uppercase	7B
f	Floating point integer	12.3
c	Character	a
s	String	test

No `length` specifier (e.g. `h`, `ll`) is used as in ANSI C `printf`. The type of the variable to be printed is determined from argument of the built-in function after the format string.

Example 66: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

7.1.4.2 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>):  <ASIP  name>@<clock  cycle>:  <formatted
text>\n
```

7.1.4.3 Return Value

None.

7.1.4.4 Example

Example 67: codasip_info function in i_halt (codasip_urisc)

```
codasip_fatal(21, "Unsupported syscall code %d.", code);
```

Example 68: codasip_info function in i_halt output(codasip_urisc)

```
fatal(21): codasip_urisc@666: Unsupported syscall 6.
```

7.1.5 codasip_get_clock_cycle

Semantic category - simulator

Supported on - IA

`uint64 codasip_get_clock_cycle()`

Returns the value of the cycle counter register.

7.1.5.1 Parameters

None.

7.1.5.2 Return Value

Value of the cycle counter register.

7.1.5.3 Example

Example 69: codasip_get_clock_cycle function

```
semantics
{
    uint64 cycles;

    cycles = codasip_get_clock_cycle();
    ...
};
```

7.1.6 codasip_inc_clock_cycle

Semantic category - simulator

Supported on - IA

`void codasip_inc_clock_cycle(uint64 value)`

Can be useful for more precise cycle counting in the IA simulation. In the default settings every instruction takes one cycle.

7.1.6.1 Parameters

<u>value</u>

The number of cycles added to the current cycle counter.

7.1.6.2 Return Value

None.

7.1.6.3 Example

Example 70: codasip_inc_clock_cycle function

```
semantics
{
    ...
    codasip_inc_clock_cycle(2); // two cycles are added to the cycle counter
    ...
};
```

7.1.7 codasip_info

Semantic category - general

Supported on - IA, CA

`void codasip_info(uint32 verbosity, text format, ...)`

Write formatted data to the simulator output. Suitable for the information messages.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

7.1.7.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

`%[flags][width][.precision]specifier`

Table 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 71: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

7.1.7.2 Parameters

verbosity

Specifies the message group for simulator output. Must be an unsigned integer value.

`Note: This behaviour is not currently implemented in the simulator.`

format

A C string containing the text to be written to the output.

The output of the `codasip_info` function is in the format:

`info (<verbosity>): <ASIP name>@<clock cycle>: <formated text>\n`

7.1.7.3 Return Value

None.

7.1.7.4 Example

Example 72: 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 73: codasip_info function in i_halt output(codasip_urisc)

```
info(0): codasip_urisc@2300: Return value = 128
```

7.1.8 codasip_intended_fallthrough

Semantic category - general

Supported on - IA, CA

`void codasip_intended_fallthrough()`

This function suppresses warnings about missing **`break`** in a **`switch case`**. It should be used in the same place where **`break`** would be placed: as the last statement in a **`case`** or **`default`** inside a **`switch`**.

7.1.8.1 Parameters

This function has no parameters.

7.1.8.2 Return Value

This function has no return value.

7.1.8.3 Example

Example 74: 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
        ...
};
```

7.1.9 codasip_print

Semantic category - general

Supported on - IA, CA

`void codasip_print(uint32 verbosity, text format, ...)`

Write formatted data to the simulator output.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers.

7.1.9.1 Format specifiers

Codasip builtin debug functions support formatting of the output using format similar to C standard function `printf`.

The format string can contain embedded format specifiers that are replaced by the values of following function arguments and formatted according to the given specifier.

The format specifier follows this prototype:

`%[flags][width][.precision]specifier`

Table 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 75: codasip_print format specifier

```
codasip_print("hello %#b", (uint4) 6);  // print unsigned 4bit value 6 as binary integer with base
prefix

Output:
hello 0b110
```

7.1.9.2 Parameters

format

A C string that containing the text to be written to the output.

7.1.9.3 Return Value

None.

7.1.9.4 Example

Example 76: codasip_print function

```
codasip_print("pipeline EX clear\n");
```

Example 77: codasip_print function output

```
pipeline EX clear
```

7.1.10 codasip_store_exit_code

Semantic category - simulator

Supported on - IA

`void codasip_store_exit_code(int32 code)`

A call to `codasip_store_exit_code` can be added to the **`semantics`** section of the `HALT` instruction for automatic testing with the instruction accurate simulator. The function creates a file, `sim_exit_code`, containing the return value of the simulated program.

7.1.10.1 Parameters

code

Exit code to be written into file.

7.1.10.2 Return Value

None.

7.1.10.3 Example

Example 78: 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);
};
```

7.1.11 codasip_syscall

Semantic category - general

Supported on - IA

`int32 codasip_syscall(uint64 arguments_address)`

`codasip_ syscall` calls the host operating system function. This function is recognized based on the content of a special structure. The address of this structure is stored in a reserved register and the content of the register is passed to `codasip_ syscall` as a parameter.

The precise description is available in the chapter "Porting Syscalls" in the *Codasip Compiler Generation Tutorial*.

7.1.11.1 Parameters

<u>arguments_address</u>

This argument points to a reserved register which holds the address of the structure with parameters for use by the host operating system syscall.

7.1.11.2 Return Value

It returns zero if the syscall is supported, otherwise, non-zero value is returned. Example of such a syscall is a file removal.

7.1.11.3 Example

Example 79: codasip_syscall function in i_syscall element

```
element i_syscall
{
    assembler { "SYSCALL" };
    binary { OPC_SYSCALL:bit[OPC_W] UNUSED:bit[REMAINING_W(OPC_W)] };
    semantics
    {
        int rc;

        codasip_compiler_unused();
        // Using a reserved register to pass the address of a structure that
        // contains information about the requested syscall.
        rc = codasip_syscall(rf_gpr[30]);
        if (rc != 0)
        {
             codasip_fatal(25, "syscall %d is not supported", rf_gpr[30]);
        }
    };
};
```

7.1.12 codasip_warning

Semantic category - general

Supported on - IA, CA

`void codasip_warning(uint32 verbosity, text format, ...)`

Writes formatted data to the simulator output, suitable for warning messages.

Writes the C string pointed by `format` to the simulator output. If `format` includes format specifiers (which begin with %), the arguments following `format` are formatted and inserted in the resulting string, replacing their respective specifiers. Newline is automatically printed after the formatted text.

7.1.12.1 Parameters

<u>verbosity</u>

Specifies the message group for simulator output. Must be an unsigned integer value.

Note: This behaviour is not currently implemented in the simulator.

<u>format</u>

A C string containing the text to be written to the output.

The output of `codasip_warning` is in the format:

```
warning (<verbosity>): <ASIP name>@<clock cycle>: <formated text>\n
```

7.1.12.2 Return Value

None.

7.1.12.3 Example

Example 80: codasip_warning function

```
#define WARNING_LEVEL       2

codasip_warning(WARNING_LEVEL, "Warning: Default case.");;
```

Example 81: codasip_warning function output

```
warning(2): codasip_urisc@2500: Warning: Default case.
```

7.2 Timing Functions

7.2.1 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 79, when the `HALT` instruction is decoded, the simulation is terminated.

7.2.1.1 Parameters

None.

7.2.1.2 Return Value

None.

7.2.1.3 Example

Example 82: codasip_halt function in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_halt();
    ...
};
```

7.3 Compiler Functions

7.3.1 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/`*`<asip_ name>`*`/ia/compiler/compiler`. This file contains the builtin functions. The compiler generator may remove some complex instructions that cannot be used automatically. This also disables them as builtins. You can use the annotation `codasip_compiler_unused()` together with `codasip_compiler_builtin()` to force builtin generation.

7.3.1.1 Parameters

None

7.3.1.2 Return Value

None.

7.3.1.3 Example

Example 83: codasip_compiler_builtin function

```
semantics
{
    ...
    codasip_compiler_builtin();
    codasip_compiler_unused();
    ...
};
```

7.3.2 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`
- `float80`
- `int8`
- `int16`
- `int32`
- `int64`

7.3.2.1 Parameters

a, b

Operands to compare.

7.3.2.2 Return Value

None.

7.3.2.3 Example

Example 84: codasip_info function in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_compiler_flag_int8(src1, src2);
    ...
};
```

7.3.3 codasip_compiler_hw_loop

Semantic category - compiler

Supported on - IA

```
void codasip_compiler_hw_loop(uint32 loop_count, uint32 loop_end)
```

This function is used in an instruction element to create a hardware loop instruction.

7.3.3.1 Parameters

<u>loop_count</u>

Number of iterations (loops) to be executed.

<u>loop_end</u>

End address of the loop (see Example 86).

7.3.3.2 Return Value

None.

7.3.3.3 Example

Example 85: codasip_compiler_hw_loop function used in i_hw_loop element (HWLOOP instruction)

```
element i_hw_loop
{
    ...
    assembler { "HWLOOP" loop_count "," loop_size};
    ...
    semantics
    {
        ...
        codasip_compiler_hw_loop(loop_count, loop_size);
        ...
    };
};
```

Example 86: 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
    ...
```

7.3.4 codasip_compiler_predicate_false

Semantic category - compiler

Supported on - IA

`void codasip_compiler_predicate_false_<type1>(<type1> predop)`

`void codasip_compiler_predicate_false(uint1 predop)`

If the predicate is zero, the instruction will be executed normally, else `NOP` replaces it. This builtin is used mostly on VLIW architectures, where `NOP` costs less than a jump instruction.

7.3.4.1 Parameters

<u>predop</u>

The predicate.

7.3.4.2 Return Value

None.

7.3.4.3 Example

Example 87: codasip_compiler_predicate_false function

```
semantics
{
    ...
    codasip_compiler_predicate_false(predop); // Instruction will be executed only if predop
                                              // is 0
    ...
};
```

7.3.5 codasip_compiler_predicate_true

Semantic category - compiler

Supported on - IA

`void codasip_compiler_predicate_true_<type1>(<type1> predop)`

`void codasip_compiler_predicate_true(uint1 predop)`

If the predicate is one, the instruction will be executed normally, else `NOP` replaces it. This builtin is used mostly on VLIW architectures, where `NOP` costs less than a jump instruction.

7.3.5.1 Parameters

predop

The predicate.

7.3.5.2 Return Value

None.

7.3.5.3 Example

Example 88: codasip_compiler_predicate_true function

```
semantics
{
    ...
    codasip_compiler_predicate_true(predop); // Instruction will be executed only in case predop
                                             // is 1
    ...
};
```

7.3.6 codasip_compiler_priority

Semantic category - compiler

Supported on - IA

`void codasip_compiler_priority(int32 priority)`

Specifies the priority of an instruction for the compiler instruction selector.

7.3.6.1 Parameters

priority

Sets the instruction priority for instruction selection. The default value is 0, and it can be set to higher values.

If negative, the instruction will not be used during instruction selection.

7.3.6.2 Return Value

None.

7.3.6.3 Example

Example 89: codasip_compiler_priority function

```
semantics
{
    ...
    codasip_compiler_priority(5);
    ...
};
```

7.3.6.4 References

- http://llvm.org/devmtg/2008-08/Gohman_CodeGenAndSelectionDAGs.pdf
- http://llvm.org/devmtg/2008-08/Gohman_CodeGenAndSelectionDAGs_Hi.m4v

7.3.7 codasip_compiler_schedule_class

Semantic category - compiler

Supported on - IA

`void codasip_compiler_schedule_class(schedule_class index)`

This function sets the schedule class property in the instruction semantics. See also chapter "Compiler Generator Guide", section "Instruction Scheduling" in the *Codasip Studio User Guide*.

7.3.7.1 Parameters

index

Schedule class to be set.

7.3.7.2 Return Value

None.

7.3.7.3 Example

Example 90: codasip_compiler_schedule_class function

```
/* schedule class for load instruction

schedule_class loads
{
    latency = 3;
};

*/

semantics
{
    ...
    codasip_compiler_schedule_class(loads);
    ...
};
```

7.3.8 codasip_compiler_undefined

Semantic category - compiler

Supported on - IA

`void codasip_compiler_undefined()`

Used to explicitly specify that the semantics of an instruction are not correct. The compiler will not use this instruction.

For example, some special instructions may read from a FIFO. Such instructions cannot be used by the compiler automatically. Such behavior cannot be captured in the instruction semantics format, therefore the code that describes the behavior has to be enclosed in `#pragma simulator {...}`. For the semantics extractor however, this instruction can behave as a no-operation instruction and can be used by other tools such a Random Assembler Programs as no operation.

7.3.8.1 Parameters

None.

7.3.8.2 Return Value

None.

7.3.8.3 Example

Example 91: codasip_compiler_undefined

```
semantics
{
    ...
    codasip_compiler_undefined();
    ...
};
```

7.3.9 codasip_compiler_unused

Semantic category - compiler

Supported on - IA

`void codasip_compiler_unused()`

Inhibits use of the instruction by the compiler (though Random Assembler Programs can still use it). Typical use of this function is in the instruction semantics of manually entered instructions. e.g. `HALT` and `SYSCALL`.

7.3.9.1 Parameters

None.

7.3.9.2 Return Value

None.

7.3.9.3 Example

Example 92: codasip_compiler_unused in i_halt (codasip_urisc)

```
semantics
{
    ...
    codasip_compiler_unused();
    ...
};
```

7.3.10 codasip_extract_subreg

Semantic category - instructions

Supported on - IA

`void codasip_ extract_ subreg (int32 dst, int32 src, int32 subidx)`

This builtin function is not supported yet. You can contact support@codasip.com if you require more information about this function.

7.3.11 codasip_insert_subreg

Semantic category - instructions

Supported on - IA

`void codasip_insert_subreg(int32 dst, int32 src, int32 subidx)`

This builtin function is not supported yet. You can contact support@codasip.com if you require more information about this function.

7.3.12 codasip_nop

Semantic category - compiler

Supported on - IA

`void codasip_nop()`

Explicitly specifies a `NOP` instruction. This might be useful for architectures that do not handle data or structural hazards, or require that delay slots after jumps.

7.3.12.1 Parameters

None.

7.3.12.2 Return Value

None.

7.3.12.3 Example

Example 93: codasip_nop function in i_nop (codasip_urisc)

```
semantics
{
    codasip_nop();
};
```

7.3.13 codasip_preprocessor_define

Semantic category - compiler

Supported on - IA

`void codasip_preprocessor_define(text define_name)`

If an element that contains a call is directly or indirectly part of the ISA, the preprocessor will then define the given value in all programs compiled by the generated C/C++ compiler.

Having such constant defined is typically useful for checking that a certain instruction extension is enabled.

7.3.13.1 Parameters

id

The constant being defined.

7.3.13.2 Return Value

None.

7.3.13.3 Example

Example 94: 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 95: using the defined constant in a compiled program

```
...
#ifdef ISE_BITCOUNT
   res = bitcount_fast();
#else
   res = bitcount_slow();
#endif
...
```

7.3.14 codasip_undef

Semantic category - general

Supported on - IA

`<type1> codasip_undef_<type1>()`

The function can be used for semantics extractor. It is useful in cases when the compiler should ignore that a result is written into a register, but there should be information that a register is written.

`<type1>` might acquire these values:

- int16
- int32

7.3.14.1 Parameters

None.

7.3.14.2 Return Value

Returns -1.

7.3.14.3 Example

Example 96: codasip_undef function

```
semantics
{
    ...
    int16 undef;
    undef = codasip_undef_int16(); // Contains -1
    ...
};
```

7.4 Arithmetic Functions

7.4.1 codasip_borrow_sub

Semantic category - general

Supported on - IA

`uint1 codasip_borrow_sub_<type1>(<type1> a, <type1> b)`

Subtracts b from a and returns a borrow flag. The result of the subtraction is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.1.1 Parameters

a, b

Operands to subtract.

7.4.1.2 Return Value

1-bit borrow flag.

7.4.1.3 Example

Example 97: codasip_borrow_sub function

```
semantics
{
    uint1 borrow;
    ...
    borrow = codasip_borrow_sub_int32(reg_src1, reg_src2);
    ...
};
```

7.4.2 codasip_borrow_sub_c

Semantic category - general

Supported on - IA

`uint1 codasip_borrow_sub_c_<type1>(<type1> a, <type1> b, uint1 c)`

Subtracts b from a with carry and returns a borrow flag. The result of the subtraction is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.2.1 Parameters

a, b, c

Operands to subtract. `c` is always 1-bit.

7.4.2.2 Return Value

1-bit borrow flag.

7.4.2.3 Example

Example 98: codasip_borrow_sub_c function

```
semantics
{
   uint1 borrow;
   ...
   borrow = codasip_borrow_sub_c_int32(reg_src1, reg_src2, cf);
   ...
};
```

7.4.3 codasip_carry_add

Semantic category - general

Supported on - IA

`uint1 codasip_carry_add_<type1>(<type1> a, <type1> b)`

Adds a to b and returns a carry flag. The result of the addition is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.3.1 Parameters

a, b

Operands to add

7.4.3.2 Return Value

1-bit carry flag

7.4.3.3 Example

Example 99: codasip_carry_add function

```
semantics
{
    uint1 carry;
    ...
    carry = codasip_carry_add_int32(reg_src1, reg_src2);
    ...
};
```

7.4.4 codasip_carry_add_c

Semantic category - general

Supported on - IA

`uint1 codasip_carry_add_c_<type1>(<type1> a, <type1> b, uint1 c)`

Adds a to b with carry and returns a carry flag. The result of the addition is discarded.

Note: The function currently supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.4.1 Parameters

a, b, c

Operands to add. `c` is always a 1-bit integer.

7.4.4.2 Return Value

1-bit carry flag

7.4.4.3 Example

Example 100: codasip_carry_add_c function

```
semantics
{
    uint1 carry;
    ...
    carry = codasip_carry_add_c_int32(reg_src1, reg_src2, carry_old);
    ...
};
```

7.4.5 codasip_ctlo

Semantic category - general

Supported on - IA

`<type1> codasip_ctlo_<type1>(<type1> src)`

Counts the leading (most significant) ones in src. For example, if src = -1, then the result is the size in bits of the type of src.

The compiler cannot use this operation automatically.

Allowed values of `<type1>` are:

- int16
- int32

7.4.5.1 Parameters

src

Operand of type `<type1>`, which will be used for counting the leading ones.

7.4.5.2 Return Value

Count of the most significant ones. Return data type is `<type1>`.

7.4.5.3 Example

Example 101: codasip_ctlo function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = -4; // 1111 1111 1111 1100
    res = codasip_ctlo_int16(src); // res = 14
};
```

7.4.6 codasip_ctlz

Semantic category - general

Supported on - IA

`<type1> codasip_ctlz_<type1>(<type1> src)`

Counts the leading (most significant) zeros in src. For example, if src = 0, then the result is the size in bits of the type of src.

The compiler can use this operation automatically. In C source code, it must be specified with the LLVM/gcc-supported builtin `__builtin_clz`.

Allowed values of `<type1>` are:

- int16
- int32

7.4.6.1 Parameters

src

Operand of type `<type1>`, which will be used for counting the leading zeros.

7.4.6.2 Return Value

Count of the most significant zeroes. Return data type is `<type1>`.

7.4.6.3 Example

Example 102: codasip_ctlz function

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = 2;
    res = codasip_ctlz_int32(src); // res = 30
};
```

7.4.7 codasip_ctpop

Semantic category - general

Supported on - IA

`<type1> codasip_ctpop_<type1>(<type1> src)`

This function counts the 1's in src.

The compiler can use this operation automatically. In C source code, it must be specified with the LLVM/gcc-supported builtin `__builtin_popcount`.

Allowed values of `<type1>` are:

- `int16`
- `int32`

7.4.7.1 Parameters

src

An operand of type `<type1>`.

7.4.7.2 Return Value

Count of 1's. The return data type is `<type1>`.

7.4.7.3 Example

Example 103: codasip_ctpop function)

```
semantics
{
    int32 src;
    int32 res;
    ...
    src = 33; // Bits 0 and 5 are set
    res = codasip_ctpop_int32(src); // res = 2
};
```

7.4.8 codasip_ctto

Semantic category - general

Supported on - IA

`<type1> codasip_ctto_<type1>(<type1> src)`

Counts the trailing (least significant) ones in src. For example, if src = -1, then the result is the size in bits of the type of src.

The compiler cannot use this operation automatically.

Allowed values of `<type1>` are:

- int16
- int32

7.4.8.1 Parameters

src

An operand of type `<type1>`.

7.4.8.2 Return Value

Count of the least significant ones. The return data type is `<type1>`.

7.4.8.3 Example

Example 104: codasip_ctto function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = 7; // 0000 0000 0000 0111
    res = codasip_ctto_int16(src); // res = 3
};
```

7.4.9 codasip_cttz

Semantic category - general

Supported on - IA

`<type1> codasip_cttz_<type1>(<type1> src)`

Counts the trailing (least significant) zeros in src. For example, if src = 0, then the result is the size in bits of the type of src.

The compiler can use this operation automatically. In C source code, it must be specified with the LLVM/gcc-supported builtin `__builtin_ctz`.

Allowed values of `<type1>` are:

- int16
- int32

7.4.9.1 Parameters

src

An operand of type `<type1>`.

7.4.9.2 Return Value

Count of the least significant zeros. The return data type is `<type1>`.

7.4.9.3 Example

Example 105: codasip_cttz function

```
semantics
{
    int16 src;
    int16 res;
    ...
    src = 2; // 0000 0000 0000 0010
    res = codasip_cttz_int16(src); // res = 1
};
```

7.4.10 codasip_overflow_add

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_add_<type1>(<type1> a, <type1> b)`

Adds a to b and returns an overflow flag. The result of the addition is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.10.1 Parameters

a, b

Operands to add.

7.4.10.2 Return Value

1-bit overflow flag.

7.4.10.3 Example

Example 106: codasip_carry_add function

```
semantics
{
    uint1 overflow;


    overflow = codasip_overflow_add_int32(reg_src1, reg_src2);
    ...
};
```

7.4.11 codasip_overflow_add_c

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_add_c_<type1>(<type1> a, <type1> b, uint1 c)`

Adds a to b with the carry flag and returns an overflow flag. The result of the addition is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.11.1 Parameters

a, b, c

Operands to add. c is 1-bit.

7.4.11.2 Return Value

1-bit overflow flag.

7.4.11.3 Example

Example 107: codasip_overflow_add_c function

```
semantics
{
    uint1 overflow;


    overflow = codasip_overflow_add_c_int32(reg_src1, reg_src2, carry);
    ...
};
```

7.4.12 codasip_overflow_sub

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_sub_<type1>(<type1> a, <type1> b)`

Subtracts b from a and returns an overflow flag. The result of the subtraction is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`
- `int16`
- `int32`

7.4.12.1 Parameters

a, b

Operands to subtract.

7.4.12.2 Return Value

1-bit overflow flag.

7.4.12.3 Example

Example 108: codasip_overflow_sub function

```
semantics
{
    uint1 overflow;


    overflow = codasip_overflow_sub_int32(reg_src1, reg_src2);
    ...
};
```

7.4.13 codasip_overflow_sub_c

Semantic category - general

Supported on - IA

`uint1 codasip_overflow_sub_c_<type1>(<type1> a, <type1> b, uint1 c)`

Subtracts b from a with carry and returns an overflow flag. The result of the subtraction is discarded.

Note: The function supports these values of `<type1>`:

- `(u)int4`
- `int8`

- `int16`
- `int32`

7.4.13.1 Parameters

a, b, c

Operands to subtract. c has 1-bit.

7.4.13.2 Return Value

1-bit overflow flag.

7.4.13.3 Example

Example 109: codasip_overflow_sub_c function

```
semantics
{
    uint1 overflow;

    overflow = codasip_overflow_add_c_int32(reg_src1, reg_src2, cf_old);
    ...
};
```

7.4.14 codasip_parity_odd

Semantic category - general

Supported on - IA

`uint1 codasip_parity_odd_<type1>(<type1> a)`

Returns 1 if the parity of the operand is even, else 0.

The computation used is (BIT_WIDTH is the bitwidth of `a`):

```
uint1 res = 1;
for (int i = 0; i < BIT_WIDTH; i++)
    res ^= (a >> i) & 1;
return res;
```

`codasip_parity_odd` is used only by the simulator. It is ignored by semantics extraction and not used for compiler generation.

7.4.14.1 Parameters

a

An operand with the following possible types:

- `int8`
- `int16`
- `int32`

7.4.14.2 Return Value

1-bit parity result.

7.4.14.3 Example

Example 110: 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.
    ...
};
```

7.5 Saturated Arithmetic Functions

7.5.1 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>`.

7.5.1.1 Parameters

a, b

Operands to be added.

7.5.1.2 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>`.

7.5.1.3 Example

Example 111: codasip_usadd function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 196;
    uint8 b = 100;
    uint8 res;

    // Addition generates the carry flag, so the 'res' will
    // contain the maximum 8-bit value, which is 255.
    res = codasip_usadd(a, b);
    ...
};
```

7.5.2 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.

7.5.2.1 Parameters

a, b

Operands to be added.

7.5.2.2 Return Value

Returns 1 if the carry flag has been generated by the addition, else returns 0.

7.5.2.3 Example

Example 112: codasip_usadd_occured function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 196;
    uint8 b = 100;
    uint1 res;

    // Addition generates the carry flag, so the saturation
    // has occured. The 'res' will be set to 1.
    res = codasip_usadd_occured(a, b);
    ...
};
```

7.5.3 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>`.

7.5.3.1 Parameters

a, b

Operands to be subtracted.

7.5.3.2 Return Value

Returns subtraction of a and b. If the subtraction result generates the borrow flag, the result is the maximum value of `<type1>`.

7.5.3.3 Example

Example 113: codasip_ussub function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 100;
    uint8 b = 196;
    uint8 res;

    // Addition generates the borrow flag, so the 'res' will
    // contain the maximum 8-bit value, which is 255.
    res = codasip_ussub(a, b);
    ...
};
```

7.5.4 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.

7.5.4.1 Parameters

a, b

Operands to be subtracted.

7.5.4.2 Return Value

Returns 1 if the borrow flag has been generated by the subtraction, else returns 0.

7.5.4.3 Example

Example 114: codasip_ussub_occured function

```
semantics
{
    ...
    // All operands are 8bit, which means that the maximum is 255
    uint8 a = 100;
    uint8 b = 196;
    uint1 res;

    // Addition generates the borrow flag, so the saturation
    // has occured. The 'res' will be set to 1.
    res = codasip_ussub_occured(a, b);
    ...
};
```

7.6 Floating Point Functions

7.6.1 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
- float80

7.6.1.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

7.6.1.2 Return Value

The smallest integral value not smaller than a.

7.6.1.3 Example

Example 115: codasip_ceil function

```
semantics
{
    float32 a = 1.6f;
    float32 b;
    b = codasip_ceil_float32(a); //b = 2.0f;
    ...
};
```

7.6.2 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`
- `float80`

7.6.2.1 Parameters

a

The argument of `<type1>` representing an angle expressed in radians.

7.6.2.2 Return Value

Cosine of a radians.

7.6.2.3 Example

Example 116: codasip_sin function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.5708f;
    b = codasip_cos_float32(a); //b = 0.0f;
    ...
};
```

7.6.3 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`
- `float80`

7.6.3.1 Parameters

a

Value of the exponent of `<type1>` .

7.6.3.2 Return Value

Exponential of a.

7.6.3.3 Example

Example 117: codasip_exp function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.0f;
    b = codasip_exp_float32(a); //b = 2.718281828f;
    ...
};
```

7.6.4 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
- float80

7.6.4.1 Parameters

a

The argument of `<type1>` whose absolute value is returned.

7.6.4.2 Return Value

The absolute value of a.

7.6.4.3 Example

Example 118: codasip_fabs function

```
semantics
{
    float32 a = -1.0f;
    float32 b;
    b = codasip_fabs_float32(a); //b = 1.0f;
    ...
};
```

7.6.5 codasip_fcmp_oeq

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_oeq_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.5.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.5.2 Return Value

1-bit result.

7.6.5.3 Example

Example 119: codasip_fcmp_oeq function

```
semantics
{
    uint1 feq;

    feq = codasip_fcmp_oeq_float32(src1, src2);
    ...
};
```

7.6.6 codasip_fcmp_oge

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_oge_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.6.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.6.2 Return Value

1-bit result.

7.6.6.3 Example

Example 120: codasip_fcmp_oge function

```
semantics
{
    uint1 fge;

    fge = codasip_fcmp_oge_float32(src1, src2);
    ...
};
```

7.6.7 codasip_fcmp_ogt

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ogt_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater than `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.7.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.7.2 Return Value

1-bit result.

7.6.7.3 Example

Example 121: codasip_fcmp_ogt function

```
semantics
{
    uint1 fgt;

    fgt = codasip_fcmp_ogt_float32(src1, src2);
    ...
};
```

7.6.8 codasip_fcmp_ole

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ole_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is less or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.8.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.8.2 Return Value

1-bit result.

7.6.8.3 Example

Example 122: codasip_fcmp_ole function

```
semantics
{
    uint1 fle;

    fle = codasip_fcmp_ole_float32(src1, src2);
    ...
};
```

7.6.9 codasip_fcmp_olt

Semantic category - general

Supported on - IA

```
uint1 codasip_fcmp_olt_<type1>(<type1> a, <type1> b)
```

Compares two float numbers of the same type. If `a` is lesser than `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.9.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.9.2 Return Value

1-bit result.

7.6.9.3 Example

Example 123: codasip_fcmp_olt function

```
semantics
{
    uint1 feq;

    feq = codasip_fcmp_olt_float32(src1, src2);
    ...
};
```

7.6.10 codasip_fcmp_one

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_one_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is not equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.10.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.10.2 Return Value

1-bit result.

7.6.10.3 Example

Example 124: codasip_fcmp_one function

```
semantics
{
    uint1 fne;

    fne = codasip_fcmp_one_float32(src1, src2);
    ...
};
```

7.6.11 codasip_fcmp_ord

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ord_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. Yields 1 if both operands are not a NaN, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.11.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.11.2 Return Value

1-bit result.

7.6.11.3 Example

Example 125: codasip_fcmp_ord function

```
semantics
{
    uint1 ford;

    ford = codasip_fcmp_ord_float32(src1, src2);
    ...
};
```

7.6.12 codasip_fcmp_ueq

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ueq_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for

representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.12.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.12.2 Return Value

1-bit result.

7.6.12.3 Example

Example 126: codasip_fcmp_ueq function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_ueq_float32(src1, src2);
    ...
};
```

7.6.13 codasip_fcmp_uge

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_uge_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for

representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.13.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.13.2 Return Value

1-bit result.

7.6.13.3 Example

Example 127: codasip_fcmp_uge function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_uge_float32(src1, src2);
    ...
};
```

7.6.14 codasip_fcmp_ugt

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ugt_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is greater than to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for

representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.14.1 Parameters

a, b

Operands to compare. Both must have the same bit-width.

7.6.14.2 Return Value

1-bit result depending float on comparison.

7.6.14.3 Example

Example 128: codasip_fcmp_ugt function

```
semantics
{
    uint1 fugt;

    fugt = codasip_fcmp_ugt_float32(src1, src2);
    ...
};
```

7.6.15 codasip_fcmp_ule

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ule_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is less or equal to `b`, then it returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for

representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.15.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.15.2 Return Value

1-bit result.

7.6.15.3 Example

Example 129: codasip_fcmp_ule function

```
semantics
{
    uint1 fule;

    fule = codasip_fcmp_ule_float32(src1, src2);
    ...
};
```

7.6.16 codasip_fcmp_ult

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_ult_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is lesser then `b`, thenit returns 1, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for

representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.16.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.16.2 Return Value

1-bit result.

7.6.16.3 Example

Example 130: codasip_fcmp_ult function

```
semantics
{
    uint1 fult;

    fult = codasip_fcmp_ult_float32(src1, src2);
    ...
};
```

7.6.17 codasip_fcmp_une

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_une_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. If `a` is not equal to `b`, then it returns 1, else 0.

An unordered comparison is performed, which means that there is a check that either operand is `NaN`.

7.6.17.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.17.2 Return Value

1-bit result.

7.6.17.3 Example

Example 131: codasip_fcmp_une function

```
semantics
{
    uint1 fune;

    fune = codasip_fcmp_une_float32(src1, src2);
    ...
};
```

7.6.18 codasip_fcmp_uno

Semantic category - general

Supported on - IA

`uint1 codasip_fcmp_uno_<type1>(<type1> a, <type1> b)`

Compares two float numbers of the same type. Yields 1 if either operand is a NaN, else 0.

Comparisons between floats are classified as either ordered or unordered and, according to which of these is chosen, operands that are NaN (Not a Number - used for representing extremes such as infinity) are treated differently. To show how this funciton relates to other float comparison builtins and their treatment of NaN, here is a summary:

```
oeq: yields true if both operands are not a NaN and a is equal to b.
ogt: yields true if both operands are not a NaN and a is greater than b.
oge: yields true if both operands are not a NaN and a is greater than or equal to b.
olt: yields true if both operands are not a NaN and a is less than b.
ole: yields true if both operands are not a NaN and a is less than or equal to b.
one: yields true if both operands are not a NaN and a is not equal to b.
ord: yields true if both operands are not a NaN.
ueq: yields true if either operand is a NaN or a is equal to b.
ugt: yields true if either operand is a NaN or a is greater than b.
uge: yields true if either operand is a NaN or a is greater than or equal to b.
ult: yields true if either operand is a NaN or a is less than b.
ule: yields true if either operand is a NaN or a is less than or equal to b.
une: yields true if either operand is a NaN or a is not equal to b.
uno: yields true if either operand is a NaN.
```

7.6.18.1 Parameters

a, b

Operands to compare. They must have the same bit-width.

7.6.18.2 Return Value

1-bit result.

7.6.18.3 Example

Example 132: codasip_fcmp_uno function

```
semantics
{
    uint1 fueq;

    fueq = codasip_fcmp_uno_float32(src1, src2);
    ...
};
```

7.6.19 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.

7.6.19.1 Parameters

a

The allowed types for `a` are:

- `float16`
- `float32`
- `float64`
- `float80`

7.6.19.2 Return Value

The value of a rounded down.

7.6.19.3 Example

Example 133: codasip_floor function (positive value use)

```
semantics
{
    float32 a;
    float32 b;

    a = 2.9f;
    b = codasip_floor_float32(a); // b = 2.0f;
    ...
};
```

Example 134: codasip_floor function (negative value use)

```
semantics
{
    float32 a;
    float32 b;

    a = -2.9f;
    b = codasip_floor_float32(a); // b = -3.0f;
    ...
};
```

7.6.20 codasip_fma

Semantic category - general

Supported on - IA

`<type1> codasip_fma_<type1>(<type1> a, <type1> b, <type1> c)`

This functions computes a*b+c.

7.6.20.1 Parameters

a, b, c

The allowed types of the arguments are:

- `float16`
- `float32`
- `float64`
- `float80`

7.6.20.2 Return Value

The result of the computation.

7.6.20.3 Example

Example 135: 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;
    ...
};
```

7.6.21 codasip_fpu_getround

Semantic category - general

Supported on - IA

`int32 codasip_fpu_getround()`

To retrieve the current rounding mode, call `int codasip_fpu_getround()`. See also `codasip_fpu_setround(int)`.

7.6.21.1 Parameters

None.

7.6.21.2 Return Value

Rounding modes have these values (one can define them as constants in the model):

- 0 - round to nearest even (translated to FE_TONEAREST)
- 3 - round toward zero (translated to FE_TOWARDZERO)
- 1 - round toward +infinity (translated to FE_UPWARD)
- 2 - round toward -infinity (translated to FE_DOWNWARD)

7.6.22 codasip_fpu_setround

Semantic category - general

Supported on - IA

`void codasip_fpu_setround(int32 mode)`

All the floating point operations described using C in CodAL use the simulator's host FPU. Its behavior is based on the current floating point rounding setting (fenv).

To set the host rounding mode from the CodAL model, function void `codasip_fpu_setround(int)` can be used. To retrieve the current rounding mode, call `int codasip_fpu_getround()`.

If the model uses FPU operations, it is suggested to add a call to`codasip_fpu_setround` to the event reset in order to initialize the FPU environment.

7.6.22.1 Parameters

`rounding_mode`

Rounding modes have these values (one can define them as constants in the model):

- 0 - round to nearest even (translated to FE_TONEAREST)
- 3 - round toward zero (translated to FE_TOWARDZERO)
- 1 - round toward +infinity (translated to FE_UPWARD)
- 2 - round toward -infinity (translated to FE_DOWNWARD)

7.6.22.2 Return Value

None.

7.6.22.3 Example

Example 136: Usage and constants definitions

```
#define CFE_TONEAREST 0
#define CFE_TOWARDZERO 3
#define CFE_UPWARD 1
#define CFE_DOWNWARD 2
...

codasip_fpu_setround(CFE_TONEAREST);
```

7.6.23 codasip_ftrunc

Semantic category - general

Supported on - IA

`<type1> codasip_ftrunc_<type1>(<type1> a)`

This function rounds a towards zero, returning the nearest integral value magnitude of which is not greater a .

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

7.6.23.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

7.6.23.2 Return Value

The nearest integral value that is not greater than a in magnitude.

7.6.23.3 Example

Example 137: codasip_ftrunc function

```
semantics
{
    float32 a = 2.3f;
    float32 b;
    b = codasip_ftrunc_float32(a); //b = 2.0f;
    ...
};
```

7.6.24 codasip_log

Semantic category - general

Supported on - IA

`<type1> codasip_log_<type1>(<type1> a)`

Returns the natural logarithm of a, i.e. the base-e logarithm, or the inverse of the natural exponential function (`codasip_exp`)

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

7.6.24.1 Parameters

a

Value of `<type1>` whose logarithm is calculated.

7.6.24.2 Return Value

Natural logarithm of a.

7.6.24.3 Example

Example 138: codasip_log function

```
semantics
{
    float32 a;
    float32 b;

    a = 2.718281828f;
    b = codasip_log_float32(a); //b = 1.0f;
    ...
};
```

7.6.25 codasip_pow

Semantic category - general

Supported on - IA

`<type1> codasip_pow_<type1>(<type1> a, <type1> b)`

Returns the base raised to the power of the exponent: a^b.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

7.6.25.1 Parameters

a

Base value of `<type1>` .

b

Exponent value of `<type1>` .

7.6.25.2 Return Value

The result of raising the base to the power of the exponent.

7.6.25.3 Example

Example 139: 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;
    ...
};
```

7.6.26 codasip_powi

Semantic category - general

Supported on - IA

`<type1> codasip_powi_<type1>(<type1> a, int32 b)`

Returns the base raised to the integer power of the exponent: $\underline{a}^{b}$.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

7.6.26.1 Parameters

<u>a</u>

Base value of `<type1>` .

<u>b</u>

Exponent value of type `int32`.

7.6.26.2 Return Value

The result of raising the base to the power of the exponent.

7.6.26.3 Example

Example 140: codasip_powi function

```
semantics
{
    float32 a;
    int32 b;
    float32 c;

    a = 2.0f;
    b = 3;
    c = codasip_powi_float32(a,b); // c = 8.0f;
    ...
};
```

7.6.27 codasip_rint

Semantic category - general

Supported on - IA

`<type1> codasip_rint_<type1>(<type1> a)`

This function rounds a to an integer value, using the round direction specified by the current rounding mode.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

7.6.27.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

7.6.27.2 Return Value

The nearest integral value to a according to the current rounding mode.

7.6.27.3 Example

Example 141: codasip_rint function

```
semantics
{
    float32 a = 2.3f;
    float32 b;
    b = codasip_rint_float32(a); //b = 2.0f, depends on the current rounding mode
    ...
};
```

7.6.28 codasip_round

Semantic category - general

Supported on - IA

`<type1> codasip_round_<type1>(<type1> a)`

Returns the integral value that is nearest to a, with halfway cases rounded away from zero.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

7.6.28.1 Parameters

a

The argument of `<type1>` whose rounded value is returned.

7.6.28.2 Return Value

The value of a rounded to the nearest integral.

7.6.28.3 Example

Example 142: codasip_round function

```
semantics
{
    float32 a = 3.8f;
    float32 b;
    b = codasip_round_float32(a); //b = 4.0f
    ...
};
```

7.6.29 codasip_sin

Semantic category - general

Supported on - IA

`<type1> codasip_sin_<type1>(<type1> a)`

This function returns the sine of the operand.

Allowed values of `<type1>` are:

- `float16`
- `float32`
- `float64`
- `float80`

7.6.29.1 Parameters

a

The argument of `<type1>` representing an angle expressed in radians.

7.6.29.2 Return Value

Sine of a.

7.6.29.3 Example

Example 143: codasip_sin function

```
semantics
{
    float32 a;
    float32 b;

    a = 1.5708f;
    b = codasip_sin_float32(a); //b = 1.0f;
    ...
};
```

7.6.30 codasip_sqrt

Semantic category - general

Supported on - IA

`<type1> codasip_sqrt_<type1>(<type1> a)`

This function returns the sqrt of the specified operand if it is a nonnegative floating point number.

Allowed values of `<type1>` are:

- float16
- float32
- float64
- float80

7.6.30.1 Parameters

a

The argument of `<type1>` whose square root is computed.

7.6.30.2 Return Value

Square root of a.

7.6.30.3 Example

Example 144: codasip_sqrt function

```
semantics
{
    float64 a;
    float64 b;

    a = 16.0f;
    b = codasip_sqrt_float64(a); //b = 4.0f;
    ...
};
```

7.7 Vector Functions

7.7.1 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.

7.7.1.1 Parameters

cond

Conditions vector.

a, b

Vectors containing elements, which are to be selected.

7.7.1.2 Return Value

Returns vector of `<type1>` with selected elements.

7.7.1.3 Example

Example 145: 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}
    ...
};
```

7.7.2 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>`.

7.7.2.1 Parameters

a

A vector whose elements are to be sign extended.

7.7.2.2 Return Value

Sign extended vector. Each element has the same bit width as `<type2>`.

7.7.2.3 Example

Example 146: 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}
    ...
};
```

7.7.3 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 ith element of the first input vector, `a` (where the index starts at 0). If i>=L, then it indicates the (i-L)th element of the second input vector, `b`.

The characteristics of the input and output vectors must be as follows:

- element count: 2, 4, 8, 16
- element bitwidth: 8, 16, 32, 64

The characteristics of the mask vector must be as follows:

- the element count must be the same as that of the input vectors
- element bitwidth: 8, 16, 32, 64

7.7.3.1 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.

7.7.3.2 Return Value

A vector with the same characteristics as `a` and `b`.

7.7.3.3 Example

Example 147: 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 148: 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)
    ...
};
```

7.7.4 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>`

7.7.4.1 Parameters

a

A vector whose elements are to be truncated.

7.7.4.2 Return Value

Truncated vector. Each element has the same bit width as `<type2>`.

7.7.4.3 Example

Example 149: 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}
    ...
};
```

7.7.5 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>`.

7.7.5.1 Parameters

a

A vector whose elements are to be zero extended.

7.7.5.2 Return Value

Zero extended vector. Each element has the same bit width as `<type2>`.

7.7.5.3 Example

Example 150: 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}
    ...
};
```

7.8 Fixed Point Functions

7.8.1 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`

7.8.1.1 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 thre is no rounding.

7.8.1.2 Return Value

Returns the result of the division.

7.8.1.3 Example

Example 151: codasip_fx_div function

```
semantics
{
    ...
    fx_div_res = codasip_fx_div_int32(reg_src1, reg_src2, frac_bits, round);
    ...
};
```

7.8.2 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`

7.8.2.1 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.

7.8.2.2 Return Value

Returns the result of a conversion. The type is the same as `<type2>`.

7.8.2.3 Example

Example 152: codasip_fx_fptofx_to function

```
semantics
{
    ...
    fx_res = codaip_fx_fptofx_float32_to_int32(src, frac_bits);
    ...
};
```

7.8.3 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`

7.8.3.1 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.

7.8.3.2 Return Value

Returns the result of a conversion. The datatype is the same as `<type2>`.

7.8.3.3 Example

Example 153: codasip_fx_fptofx_to function

```
semantics
{
    ...
    fx_res = codaip_fx_fptofx_float32_to_int32(src, frac_bits);
    ...
};
```

7.8.4 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`

7.8.4.1 Parameters

a

The source fixed point number.

fract_bits

Number of bits representing the decimal part of the result.

7.8.4.2 Return Value

The result of the conversion.

7.8.4.3 Example

Example 154: 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.
    ...
};
```

7.8.5 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`

7.8.5.1 Parameters

a

The source integer number.

fract_bits

The number of bits in the decimal part of the result.

7.8.5.2 Return Value

The result of the conversion.

7.8.5.3 Example

Example 155: 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)
    ...
};
```

7.8.6 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`

7.8.6.1 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.

7.8.6.2 Return Value

Returns the result of a mutiplication.

7.8.6.3 Example

Example 156: codasip_fx_div function

```
semantics
{
    ...
    fx_mul_res = codasip_fx_mul_int32(reg_src1, reg_src2, frac_bits, round);
    ...
};
```

7.9 Complex Numbers Functions

7.9.1 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.

7.9.1.1 Parameters

`a,b`

Complex vector variables.

7.9.1.2 Return Value

Result of addition.

7.9.2 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 157 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;
}
```

7.9.2.1 Parameters

`a,b`

Complex vector variables.

7.9.2.2 Return Value

Result of complex division.

7.9.3 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 158: 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;
}
```

7.9.3.1 Parameters

`a,b`

Complex vector variables.

7.9.3.2 Return Value

Result of complex multiplication.

7.9.4 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`.

7.9.4.1 Parameters

`a,b`

Complex vector variables.

7.9.4.2 Return Value

Result of complex subtraction.

7.10 Miscellaneous Functions

7.10.1 codasip_bitcast_to

Semantic category - general

Supported on - IA

`<type2> codasip_bitcast_<type1>_to_<type2>(<type1> value)`

This function converts the value of an input type, `<type1>`, to a target type, `<type2>`, without changing any bits. The bitsize of the input type must match the size of the target type (the total size in the case of vector). The list below shows the possible combinations of parameter and return value. There are two types of bitcasts:

- Integer to float (and vice versa).
  The following conversions are allowed(`<type1>` to `<type2>`):
  - `uint16` to `float16` (`float16` to `uint16`)
  - `uint32` to `float32` (`float32` to `uint32`)
  - `uint64` to `float64` (`float64` to `uint64`)
  - `uint80` to `float80` (`float80` to `uint80`)
  - `uint16` to `float16` (`float16` to `uint16`)
- Integer to vector of floats (and vice versa).
  The following conversions are allowed(`<type1>` to `<type2>`):
  - `uint64` to `v2f32` (`v2f32` to `uint64`)
  - `uint128` to `v4f32` (`v4f32` to `uint128`)
  - `uint128` to `v2f64` (`v2f64` to `uint128`)
  - `uint256` to `v8f32` (`v8f32` to `uint256`)
  - `uint256` to `v4f64` (`v4f64` to `uint256`)
  - `uint512` to `v16f32` (`v16f32` to `uint512`)
  - `uint512` to `v8f64` (`v8f64` to `uint512`)
  - `uint1024` to `v16f64` (`v16f64` to `uint1024`)

These bitcasts do simple bit-precise copy, an example of internal implementation in C can be seen here:

Example 159: codasip_bitcast

```
float codasip_bitcast_uint32_to_float32(uint32_t uninterpreted_value)
{
	return *((float *)&uninterpreted_value);
}
```

7.10.1.1 Parameters

value

Input value of type `<type1>` for the conversion.

7.10.1.2 Return Value

Returned converted value of type `<type2>`.

7.10.1.3 Example

Example 160: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]);
}
```

8 ANNEX: CODAL GUIDELINES

Please note that suggested rules for organizing files and directories are given in "Annex: File Organization" on page 200. Further, best practices associated with compiler generation are explained in chapter "Compiler Generation Best Practices" in the *Codasip Studio User Guide*.

8.1 Common Guidelines

8.1.1 CL025 Resources Naming Conventions

There are several types of resources which can be used in a design level description or ASIP 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 ASIP description and their prefixes.

Table 16: Resources in the ASIP 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;`

Resources in the design level description and their prefixes.

Table 17: Resources in the design level 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`

extern	`e_`	Optional	`extern fifo`
			`extern e_fifo`

Rationale: The prefix allows immediate identification of the type of resource without requiring examination of its definition.

8.1.2 CL070 Project Naming Conventions

Project names should be all lowercase with underscores "_" as separators.

Example 161: 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.

8.1.3 CL021 Identifier Naming Conventions

All identifiers (i.e. variables, resources and functions) are written in lowercase with underscores "_" as separators between words.

Example 162: 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.

8.1.4 CL035 Indentation

Use four spaces instead of TAB.

Rationale: File can be opened in any editor and it looks the same.

8.1.5 CL040 Maximum Line Length

Maximum line length is 100 characters (see "CL075 Line Wrapping/Splitting" on page 182 for details how to maintain longer lines).

Rationale : Files can be opened in standard editors on standard monitors while maintaining readability.

8.1.6 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 187). The last example defines a macro which generates one operation code definition.

Example 163: 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.

8.1.7 CL050 `#define` Guard

All header files must have `#define` guards to prevent multiple inclusion.

Example 164: 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.

8.1.8 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 165: File naming conventions - valid

```
utilities.hcodal
ca_pipe_stage3_ex.codal
```

Examples of invalid file names:

Example 166: 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 *<project>*`/model/`*<ia>*`/compiler` and/or *<project>*`/model/`*<ca>*`/compiler`. See Codasip Studio Technical Reference Manual chapter "File Organization".

8.1.9 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 167: 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.

8.1.10 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 168: 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.

8.1.11 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 169: 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 170: Line wrapping/splitting - function call parameters

```
res = function(op,
    src1,
    src2);
```

Example 171: Line wrapping/splitting - loop control statements

```
for (ii = 0;
    ii < MAX;
    ++ii)
{
    ...
}
```

Example 172: Line wrapping/splitting - operators (align to '=')

```
// align to '='
result = a + b + c +
         d + e + f +
         g;
```

Example 173: Line wrapping/splitting - ternary if

```
result = condition
         ? return_something
         : return_else;
```

Example 174: 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.

8.1.12 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.

8.1.13 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 175: 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.

8.1.14 CL093 White Spaces in Expressions

Use spaces for separation of operators and operands in expressions.

Example 176: 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.

8.1.15 CL100 Variable Declaration Formatting

Insert an empty line after a declaration block of variables to separate declaration from semantics.

Example 177: 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.

8.1.16 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 178: 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.

8.2 Design Level Guidelines

8.2.1 CL610 Uniform Design Level 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 179: Design level resource declarations - memories and caches

```
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 180: Design level resource declarations - buses

```
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 181: Design level resource declarations - externs

```
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 182: Design level resource declarations - ports

```
port bit[DATA_W] p_data_out { direction = OUT; };
```

Interfaces.

Example 183: Design level resource declarations - interfaces

```
interface if_fe
{
    bits = { ADDR_W, WORD_W, LAU_W };
    type = MEMORY:MASTER;
    flag = R;
    endianness = BIG;
};
```

Rationale: Improved readability and maintainability of code.

8.2.2 CL605 Connections (Rec)

When connecting components, resources or ASIPs within the design level or ASIP a designer should follow data flow from the left (output) to right (input).

The following examples show two connections. The first one shows a data connection from an ASIP to a design level port. The second example shows a connection from a design level port to an ASIP input.

Example 184: Connections

```
connect codasip_urisc.p_data_out => p_data_out;
connect p_data_in => codasip_urisc.p_data_in;
```

Rationale: Improved readability and maintainability of code.

8.3 ASIP Guidelines

8.3.1 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 185: 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.

8.3.2 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 186: 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.

8.3.3 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 Codix Cobalt processor.

Example 187: Set naming convention

```
set isa = i_ctrl, i_alu, i_special, ...;
```

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 188: 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 189: 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 190: 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 191: 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 192: 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.

8.3.4 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 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 193: 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.

8.3.5 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 194: 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.

8.3.6 CL395 Uniform ASIP 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 195: ASIP 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 196: ASIP resource declarations - signals

```
signal bit[OPC_W] s_opc;
```

Ports.

Example 197: ASIP resource declarations - ports

```
port bit[DATA_W] p_data_out { direction = OUT; };
```

Interfaces.

Example 198: ASIP resource declarations - interfaces

```
interface if_fe
{
    bits = { ADDR_W, WORD_W, LAU_W };
    type = MEMORY:MASTER;
    flag = R;
    endianness = BIG;
};
```

Address Spaces.

Example 199: ASIP 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 200: ASIP resource declarations - module

```
module m_1
{
    register bit[DATA_W] r_data;
    ...
};
```

Rationale: Unified formatting across all code.

8.3.7 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 201: Element formatting

```
element el
{
    use ...;

    assembler { ... };
    binary { ... };

    semantics
    {
        ...
    };
    return { ... };
    timing
    {
        ...
    };
};
```

Event

Example 202: Event formatting

```
event ev : pipe.STAGE
{
    use ...;

    start
    {
        ...
    };
    semantics
    {
        ...
    };
    decoders
    {
        ...
    };
    timing
    {
        ...
    };
};
```

Rationale: Unified formatting across all code.

8.3.8 CL378 `Decoder` Section Formatting

`decoder` section should have the following form.

Example 203: 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.

8.3.9 CL387 `if-else` Statement Formatting

An `if-else` statement should have the following form.

Example 204: if-else statement formatting

```
if (cond)
{
    ...
    do_someting();
    ...
}
else
{
    ...
    do_something_else();
    ...
}
```

Rationale: Unified formatting across all code.

8.3.10 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 205: 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.

8.3.11 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 206: 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.

8.3.12 CL390 Loops Formatting

Loops should have the following form.

Example 207: 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.

8.3.13 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 208: if-else cascade formatting

```
if (cond1)
{
    ...
}
else if (cond2)
{
    ...
}
else if (cond3)
{
    ...
}
else
{
    ...
}
```

Rationale: Unified formatting across all code.

8.3.14 CL393 Functions Definitions Formatting

Use the following formatting.

Example 209: Function definition formatting

```
int function()
{
    ...
    do_something();
    ...
}
```

Rationale: Unified formatting across all code.

8.3.15 CL330 Loop Control Statements in the `for()` Construction

Only loop control statements are allowed in the `for()` construction.

Example 210: Control statements in the for() construction

```
sematics
{
    ...
    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.

8.3.16 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 211: 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).

8.3.17 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 212: 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.

8.3.18 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.

8.3.19 CL345 Start Section Placement (Rec)

Start section should be placed in `isa.codal` file.

Example 213: 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.

9 ANNEX: FILE ORGANIZATION

The following subsections describes the best practice for a file organization of CodAL source files.

9.1 Directory Structure of CodAL Project Describing Design Level

9.1.1 Directory *`<project>`*

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

- *`<project>`*`/doc/`
  Optional directory. Contains documentation of the project (e.g. top level description, ...).
- *`<project>`*`/model/`
  Mandatory directory. Contains the design level model (see "Directory <project>/model" on page 200).

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

Table 18: Directory hierarchy summary of ASIP project – directory <project>

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

9.1.2 Directory *`<project>`*`/model`

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

- *`<project>`*`/model/`*`<ca>`*`/`
  Optional directory. Contains the cycle-accurate model. *`<ca>`* is the name of the cycle-accurate model, such as *ca* (default name) or *ca_3stages*.
- *`<project>`*`/model/`*`<ia>`*`/`
  Optional directory. Contains the instruction-accurate model. *`<ia>`* is the name of the instruction-accurate model, such as *ia* (default name) or *ia_functional*.
- *`<project>`*`/model/share/`
  Mandatory directory. Contains shared parts of the model across the instruction-

accurate and cycle-accurate models. For example top level description is usually the same for all models (see "Directory <project>/model/share" on page 201).

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

Table 19: Directory hierarchy summary of ASIP project– directory <project>/model

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

9.1.3 Directory `<project>/model/share`

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

- `<project>/model/share/include/`
  Mandatory directory. Contains common includes, headers of auxiliary functions, and configuration headers.
  - `config.hcodal`
    Mandatory file. Contains configuration of interfaces, register files, address spaces, etc.
  - `utils.hcodal`
    Header file name example.
- `<project>/model/share/level/`
  Mandatory directory. Contains a design level description.
  - `level.codal`
    Mandatory file. This file may be split into more codal files for better readability.
- `<project>/model/share/other/`
  Optional directory. Contains other source files, such as sources of auxiliary functions.
  - `utils.codal`
    Source file name example.

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

Table 20: Directory hierarchy summary of ASIP project– directory <project>/model/share

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

9.2 Directory Structure of CodAL Project Describing ASIP

9.2.1 Directory *<project>*

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

- `<project>/doc/`
  Optional directory. Contains documentation of the project (e.g. ISA description, scheme of the ASIP, ...).
- `<project>/libs/`
  Optional directory. Contains a port of standard C library (e.g. newlib), and startup code.
- `<project>/linker/`
  Optional directory. Contains custom linker script with `.lds` extension.
- `<project>/model/`
  Mandatory directory. Contains the ASIP model (see "Directory <project>/model" on page 200)

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

Table 21: Directory hierarchy summary of ASIP project– directory <project>

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

9.2.2 Directory *<project>*/model

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

- `<project>/model/<ca>/`
  Optional directory. Contains the cycle-accurate model. `<ca>` is the name of the cycle-accurate model, such as *ca* (default name) or *ca_3stages* (see "Directory <project>/model/<ca>" on page 206).
- `<project>/model/<ia>/`
  Optional directory. Contains the instruction-accurate model. `<ia>` is the name of the instruction-accurate model, such as *ia* (default name) or *ia_functional* (see "Directory <project>/model/<ia>" on page 205).
- `<project>/model/share/`
  Mandatory directory. Contains shared parts of the model across the instruction-accurate and cycle-accurate models. For example it includes instruction set architecture (ISA) description (see "Directory <project>/model/share" on page 201).

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

Table 22: Directory hierarchy summary of ASIP project− directory <project>/model

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

9.2.3 Directory *<project>*/model/share

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

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

  - `opcodes.hcodal`
    Mandatory file. Contains defines for operation codes of instructions including bit widths of certain parts of instructions.
  - `utils.hcodal`
    Header file name example.

- `<`*`project`*`>/model/share/isa/`
  Mandatory directory. Contains an ISA description.
  - `isa.codal`
    Mandatory file. This file may be split into more codal files for better readability. (e.g. `isa.codal`, `isa_ops.codal`, `isa_am.codal`, ...)
  - `emulations.codal`
    Optional file. Contains emulations that are needed for C compiler. This file may be split into more codal files for better readability.
  - `peepholes.codal`
    Optional file. Contains peephole patters for C compiler. This file may be split into more codal files for better readability.

- `<`*`project`*`>/model/share/other/`
  Optional directory. Contains other source files, such as sources of auxiliary functions.
  - `utils.codal`
    Source file name example.
  - `settings.codal`
    Settings file name example.
  - `version.codal`
    File holding a version of Codasip Studio that is needed.

- `<`*`project`*`>/model/share/resources/`
  Mandatory directory. Contains a description of the interfaces of the ASIP as well as architectural resources.
  - `arch.codal`
    Mandatory file. Contains architectural registers, architectural resources, address spaces, assembler configuration and schedule classes.
  - `interface.codal`
    Mandatory file. Contains interfaces of ASIP (e.g. interfaces to buses/memories, and ports)

  - `externs.codal`
    Optional file. Contains definitions of used **`extern`**s and their connections.

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

Table 23: Directory hierarchy summary of ASIP project− directory <project>/model/share

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

9.2.4 Directory *<project>*/model/*<ia>*

The directory contains definition of resources, headers and sources of auxiliary functions, and defines, associated only with the instruction-accurate model. The *`<ia>`* is the name of the instruction-accurate model, such as *ia* (default name) or *ia_functional*. The name must have the prefix *ia*, so the name also denotes the type of the model. Each file, except files placed in the `compiler` directory, has *`<ia>`*`_` as the file name prefix. The standard set of directories and files are:

- *`<project>`*`/model/`*`<ia>`*`/compiler/`
  Optional directory. Contains additional source files for compiler generator.
  - `user_semantics.sem`
    Compiler source file name example.
- *`<project>`*`/model/`*`<ia>`*`/events/`
  Mandatory directory. Must contain the definition of *main* and *reset* events, and may contain files with additional event definitions.
  - `<ia>_main_reset.codal`
    Mandatory file. Contains definition of *main* and *reset* events.
- *`<project>`*`/model/`*`<ia>`*`/include/`
  Optional directory. Contains instruction-accurate specific headers or define files.
  - `ia_defines.hcodal`
    Defines file name example.

    - `ia_utils.hcodal`
      Header file name example.
- `<project>/model/<ia>/other/`
  Optional directory. Contains source files for instruction-accurate auxiliary functions.
    - `ia_utils.codal`
      Source file name example.
    - `ia_settings.codal`
      Settings file name example.
- `<project>/model/<ia>/resources/`
  Mandatory directory. Contains instruction-accurate specific resources (e.g. register holding a fetched instruction from a memory).
    - `ia_resources.codal`
      Resource file name example.

The following table summarizes the directory hierarchy of the instruction-accurate model.

Table 24: Directory hierarchy summary of ASIP project− directory <project>/model/<ia>

Directory hierarchy summary	
Directory	**Type**
`<project>/model/<ia>/compiler/`	Optional
`<project>/model/<ia>/events/`	Mandatory
`<project>/model/<ia>/include/`	Optional
`<project>/model/<ia>/other/`	Optional
`<project>/model/<ia>/resources/`	Mandatory

9.2.5 Directory `<project>/model/<ca>`

The directory contains definition of resources, decoders, pipelines, headers, sources of auxiliary functions, and defines associate only with the cycle-accurate model. `<ca>` is the name of the cycle-accurate model, such as *ca* (default name) or *ca_3stages*. The name must have the prefix *ca*, so the name also denotes the type of the model. Each file, except files placed in the `compiler` directory, has `<ca>_` as the file name prefix. The standard set of directories and files are:

- `<project>/model/<ca>/compiler/`
  Optional directory. Contains additional source files for the compiler generator.

   - `CodasipMicroClasses.td`
     Source file name example.

- `<project>/model/<ca>/decoders/`
  Mandatory directory. Contains definition of decoder(s). The naming convention is `<ca>_<name>.codal` where `<name>` is a name of the decoder. If the decoder is too large (e.g. file contains more than 500 lines), it may be split into more smaller files (e.g. `<ca>_<name>.codal`, `<ca>_<name>_am.codal`, `<ca>_<name>_ops.codal`) for readability.

  If there are more than one decoder, each decoder is placed in a sub-directory called `<name>` where <*name*> is a name of the decoder. The files within this directory follows the same naming conventions as in the case of a single decoder.

   - `ca_decoder.codal,ca_decoder_am.codal,ca_decoder_ops.codal`
     Split file name example.
   - `predecoder/ca_predecoder.codal`
     `decoder/ca_decoder.codal`
     Multi-decoder directory and file name example.

- `<project>/model/<ca>/events/`
  Mandatory directory. Contains definition of *main* and *reset* events, and may contain files with additional event definitions.

   - `<ca>_main_reset.codal`
     Mandatory file. Contains definition of *main* and *reset* events.

- `<project>/model/<ca>/include/`
  Optional directory. Contains cycle-accurate specific headers or defines files.

   - `ca_defines.hcodal`
     Defines file name example.
   - `ca_utils.hcodal`
     Header file name example.

- `<project>/model/<ca>/other/`
  Optional directory. Contains source files of cycle-accurate auxiliary functions.

   - `ca_utils.codal`
     Source file name example.
   - `ca_settings.codal`
     Settings file name example.

- `<project>/model/<ca>/pipelines/`
  Mandatory directory for a pipelined architecture. Contains definition of pipeline (s). File naming convention is `<ca>_<pipe>_stage<order>_<stage>.codal`, where `<pipe>` is a name of the pipeline, `<order>` is an order of a pipeline stage starting from zero and `<stage>` is a name of the pipeline stage.

  If there are more pipelines, each pipeline is placed in a sub-directory called `<name>`. The files within this directory follows the same naming convention as in the case of one pipeline architecture.

  Each file is dedicated to the specific stage of the pipeline and contains events assigned to this stage.

  - `ca_pipe_stage0_fe.codal`
    First stage file name example.
  - `ca_pipe_stage1_id.codal`
    Second stage file name example.
- `<project>/model/<ca>/resources/`
  Mandatory directory. Contains cycle-accurate specific resources (e.g. pipeline registers).
  - `ca_resources.codal`
    Resource file name example. The file can be split to smaller files is necessary (e.g. split resources respecting a pipeline).

The following table summarizes the directory hierarchy of the cycle-accurate model.

Table 25: Directory hierarchy summary of ASIP project− directory <project>/model/<ca>

Directory hierarchy summary	
Directory	**Type**
`<project>/model/<ca>/compiler/`	Optional
`<project>/model/<ca>/decoders/`	Mandatory
`<project>/model/<ca>/events/`	Mandatory
`<project>/model/<ca>/include/`	Optional
`<project>/model/<ca>/other/`	Optional
`<project>/model/<ca>/pipelines/`	Mandatory
`<project>/model/<ca>/resources/`	Mandatory

10 ANNEX: CODAL SYNTAX SUMMARY

The following syntax summary is valid for descriptions of ASIPs: that is, .codal source files and files included into them.

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

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

```
MappingStatements
  : MappingStatements ',' CompoundId ':' CompoundId OptionalComma
  | CompoundId ':' CompoundId OptionalComma
```

```
ScheduleClassBodyEncap
  : '{' ScheduleClassBody '}'
  | %empty
```

```
RegisterFile
  : Specifier "registerfile" "bit" '[' CompileExpression ']' IdList '{' RegisterFileBody '}'
```

```
PeepholeSection
  : SectionPattern ';'
  | SectionReplace
  | SectionIfReplace
  | SectionMapping ';'
```

```
SectionAssemblerBody
  : SectionAssemblerBody SectionAssemblerElement
  | SectionAssemblerElement
```

```
AnsiMultiplicativeExpression
  : AnsiCastExpression
  | AnsiMultiplicativeExpression '*' AnsiCastExpression
  | AnsiMultiplicativeExpression '/' AnsiCastExpression
  | AnsiMultiplicativeExpression '%' AnsiCastExpression
  | AnsiMultiplicativeExpression "**" AnsiCastExpression
  | AnsiMultiplicativeExpression "::*" AnsiCastExpression
```

```
ElementSection
  : SectionIf
  | SectionSwitch
  | SectionAssembler ';'
  | SectionBinary ';'
  | SectionSemantics ';'
  | SectionReturn ';'
  | SectionTiming ';'
```

```
AnsiFunctionDeclarator
  : AnsiId '(' ')'
  | AnsiId '(' AnsiParameterList ')'
```

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

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

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

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

```
SetBodyElement
  : CompoundId
```

```
SettingsCompoundBody
  : SettingsCompoundBody Settings ';'
  | Settings ';'
```

```
ExternAttribute
  : Interface
  | Port
```

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

```
TranslationUnit
  : TranslationUnit TopConstruct
  | TopConstruct
```

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

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

```
AnsiLabeledStatementList
  : AnsiLabeledStatementList AnsiLabeledStatement
  | AnsiLabeledStatement
```

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

```
EventSections
  : EventSections EventSection
  | EventSection
```

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

```
SectionIfReplaceBody
  : SectionReplace
  | '{' SectionReplace '}'
```

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

```
AnsiInclusiveOrExpression
  : AnsiExclusiveOrExpression
  | AnsiInclusiveOrExpression '|' AnsiExclusiveOrExpression
```

```
AnsiInitDeclarator
  : AnsiDeclarator
  | AnsiDeclarator '=' AnsiInitializer
```

```
ScheduleClassAttribute
  : "latency" '=' CompileExpression
  | "delay_slot" '=' CompileExpression
  | "allow_in_delay_slot" '=' CompileExpression
  | "custom_schedule" '=' CompileExpression
  | "llvm_class" '=' String
```

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

```
Peephole
  : "peephole" Id '{' PeepholeDeclarations PeepholeSections '}'
  | "peephole" Id '{' PeepholeSections '}'
```

```
SectionUseInstance
  : Id
```

```
SettingsBody
  : SettingsBody Setting ';'
  | Setting ';'
```

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

```
PropertyArgument
  : CompoundId
  | String
```

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

```
Id
  : "identifier"
```

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

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

```
AnsiUnaryOperator
  : '+'
  | '-'
  | '~'
  | '!'
```

```
RegisterFileAttribute
  : "size" '=' CompileExpression
  | "mask" '=' CompileExpression
  | "dataports" '=' '{' CompileExpression ',' CompileExpression '}'
  | "default" '=' CompileExpression
  | Overlap
  | Dff
  | Reset
  | ClockEnable
```

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

```
AnsiAssignmentExpression
  : AnsiConditionalExpression
  | AnsiUnaryExpression AnsiAssignmentOperator AnsiAssignmentExpression
```

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

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

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

```
AnsiInitializer
  : AnsiAssignmentExpression
```

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

```
EmulationSections
  : EmulationSections EmulationSection
  | EmulationSection
```

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

```
PeepholeDeclarations
  : PeepholeDeclarations PeepholeDeclaration ';'
  | PeepholeDeclaration ';'
```

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

```
AnsiAttributeBody
  : AnsiAttributeBody ',' AnsiId
  | AnsiId
```

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

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

```
TopConstruct
  : Resource ';'
  | Element ';'
  | Event ';'
  | Emulation ';'
  | Peephole ';'
  | Set ';'
  | Start ';'
  | ScheduleClass ';'
  | AnsiDeclaration
  | AnsiFunctionDefinition
  | Module ';'
  | Extern ';'
  | Connect ';'
  | SettingsCompound ';'
  | ';'
```

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

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

```
SectionTiming
  : "timing" '{' AnsiStatementList '}'
```

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

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

```
AnsiStatement
  : AnsiCompoundStatement
  | AnsiExpressionStatement
  | AnsiSelectionStatement
  | AnsiIterationStatement
  | AnsiJumpStatement
```

```
AnsiUnaryExpression
  : AnsiPostfixExpression
  | "++" AnsiUnaryExpression
  | "--" AnsiUnaryExpression
  | AnsiUnaryOperator AnsiCastExpression
  | "bitsizeof" AnsiUnaryExpression
  | "bitsizeof" '(' AnsiDeclarationSpecifiers ')'
  | "clog2" AnsiUnaryExpression
```

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

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

```
AnsiTypeSpecifier
  : "bool"
  | "char"
  | "short"
  | "int"
  | "int_" '<' AnsiTypeSpecifierTemplate '>'
  | "long"
  | "signed"
  | "unsigned"
  | "void"
  | "v_[uif]_" '<' AnsiTypeSpecifierTemplate ',' AnsiTypeSpecifierTemplate '>'
  | "v*[uif]*"
  | "float"
  | "double"
  | "bundle*"
  | "debundle*"
  | "typename"
```

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

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

```
AnsiAdditiveExpression
  : AnsiMultiplicativeExpression
  | AnsiAdditiveExpression '+' AnsiMultiplicativeExpression
  | AnsiAdditiveExpression '-' AnsiMultiplicativeExpression
  | AnsiAdditiveExpression "::" AnsiMultiplicativeExpression
```

```
AnsiConditionalExpression
  : AnsiLogicalOrExpression
  | AnsiLogicalOrExpression '?' AnsiExpression ':' AnsiConditionalExpression
```

```
AnsiLabeledStatement
  : "case" AnsiConstantExpression ':' AnsiStatementList
  | "case" AnsiConstantExpression ':'
  | "default" ':' AnsiStatementList
  | "default" ':'
```

```
RplPolicy
  : "rpl_policy" '=' Id
```

```
AnsiEnumId
  : "identifier"
```

```
AnsiVariableDeclarator
  : AnsiId
  | AnsiId '[' AnsiConstantExpression ']'
```

```
AnsiDeclarationSpecifiersBasic
  : AnsiStorageClassSpecifier
  | AnsiDeclarationSpecifiersBasic AnsiStorageClassSpecifier
  | AnsiTypeSpecifier
  | AnsiDeclarationSpecifiersBasic AnsiTypeSpecifier
  | AnsiTypeQualifier
  | AnsiDeclarationSpecifiersBasic AnsiTypeQualifier
  | AnsiAttribute
  | AnsiDeclarationSpecifiersBasic AnsiAttribute
```

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

```
String
  : "string"
```

```
AnsiJumpStatement
  : "continue" ';'
  | "break" ';'
  | "return" ';'
  | "return" AnsiExpression ';'
```

```
AnsiStorageClassSpecifier
  : "typedef"
```

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

```
SettingsCompound
  : "settings" '{' SettingsCompoundBody '}'
```

```
AnsiLogicalAndExpression
  : AnsiInclusiveOrExpression
  | AnsiLogicalAndExpression "&&" AnsiInclusiveOrExpression
```

```
AnsiShiftExpression
  : AnsiAdditiveExpression
  | AnsiShiftExpression "<<" AnsiAdditiveExpression
  | AnsiShiftExpression ">>" AnsiAdditiveExpression
  | AnsiShiftExpression ROL_OP AnsiAdditiveExpression
  | AnsiShiftExpression ROR_OP AnsiAdditiveExpression
```

```
SectionMapping
  : "mapping" '{' MappingStatements '}'
```

```
Emulation
  : "emulation" Id Properties '{' EmulationDeclarations EmulationSections '}'
  | "emulation" Id Properties '{' EmulationSections '}'
```

```
AnsiEnumerator
  : AnsiEnumId
  | AnsiEnumId '=' AnsiConstantExpression
```

```
ScheduleClass
  : "scheduleclass" IdList ScheduleClassBodyEncap
```

```
AnsiDeclaration
  : AnsiDeclarationSpecifiers ';'
  | AnsiDeclarationSpecifiers AnsiInitDeclaratorList ';'
```

```
AnsiTypeQualifier
  : "const"
  | "inline"
```

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

```
CacheAttribute
  : Interface
  | Size
  | Linesize
  | Numways
  | RplPolicy
  | Latencies
  | NonCacheable
```

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

```
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 ']'
```

```
AnsiStatementList
  : AnsiStatement
  | AnsiStatementList AnsiStatement
```

```
SectionReturn
  : "return" '{' AnsiExpression OptionalSemicolon '}'
```

```
SectionReplace
  : "replace" AnsiCompoundStatement ';'
```

```
AnsiId
  : "identifier"
  | '.' "identifier"
```

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

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

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

```
EmulationDeclarations
  : EmulationDeclarations EmulationDeclaration ';'
  | EmulationDeclaration ';'
```

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

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

```
Unaligned
  : "unaligned" '=' CompileExpression
```

```
SetAssign
  : '='
  | "+="
```

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

```
AnsiAssignmentOperator
  : '='
  | "*="
  | "/="
  | "%="
  | "+="
  | "-="
  | "<<="
  | ">>="
  | "&="
  | "^="
  | "|="
  | ">>>="
  | "<<<="
```

```
EmulationDeclaration
  : SectionUse
  | Attribute
```

```
Signal
  : "signal" "bit" '[' CompileExpression ']' IdList
```

```
Linesize
  : "linesize" '=' CompileExpression
```

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

```
ModuleBody
  : ModuleBody TopConstruct
  | TopConstruct
```

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

```
AnsiArgumentList
  : AnsiAssignmentExpression
  | AnsiArgumentList ',' AnsiAssignmentExpression
```

```
AnsiDeclarationList
  : AnsiDeclaration
  | AnsiDeclarationList AnsiDeclaration
```

```
EmulationSection
  : SectionInstructions
  | SectionIfInstructions
  | SectionSemantics ';'
```

```
Event
  : "event" Id Properties '{' EventSections '}'
  | "event" Id Properties '{' EventDeclarations EventSections '}'
```

```
PeepholeSections
  : PeepholeSections PeepholeSection
  | PeepholeSection
```

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

```
SettingBoby
  : SettingBoby ',' SettingElement
  | SettingElement
```

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

```
PortDirection
  : "direction" '=' Id ';'
```

```
AnsiConstantExpression
  : AnsiConditionalExpression
```

```
AnsiExpressionStatement
  : ';'
  | AnsiExpression ';'
```

```
AnsiExclusiveOrExpression
  : AnsiAndExpression
  | AnsiExclusiveOrExpression '^' AnsiAndExpression
```

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

```
EventDeclaration
  : SectionUse
  | Register
  | RegisterFile
  | Signal
```

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

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

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

```
AnsiPostfixExpression
  : AnsiPrimaryExpression
  | AnsiPostfixExpression '[' AnsiExpression ']'
  | AnsiPostfixExpression '[' AnsiExpression ".." AnsiExpression ']'
  | AnsiPostfixExpression '(' ')'
  | AnsiPostfixExpression '(' AnsiArgumentList ')'
  | AnsiPostfixExpression '.' AnsiId
  | AnsiPostfixExpression "++"
  | AnsiPostfixExpression "--"
```

```
AnsiPrimaryExpression
  : AnsiId
  | "integer constant"
  | "boolean constant"
  | "real constant"
  | "char constant"
  | AnsiStringList
  | '(' AnsiExpression ')'
```

```
PeepholeDeclaration
  : SectionUse
```

```
SettingsSpecifier
  : "compiler"
  | "debugger"
  | "simulator"
  | "assembler"
```

```
AnsiFunctionDefinition
  : AnsiDeclarationSpecifiers AnsiFunctionDeclarator AnsiCompoundStatement
```

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

```
EventDeclarations
  : EventDeclarations EventDeclaration ';'
  | EventDeclaration ';'
```

```
SectionSemantics
  : "semantics" AnsiCompoundStatement
```

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

```
SettingElement
  : AnsiConditionalExpression
  | AnsiTypeSpecifier
```

```
AnsiEnumeratorList
  : AnsiEnumerator
  | AnsiEnumeratorList ',' AnsiEnumerator
```

```
Resource
  : AddressSpace
  | Pipeline
  | Register
  | RegisterFile
  | Signal
  | Interface
  | Port
  | Memory
  | Cache
  | Bus
```

```
NonCacheable
  : "non_cacheable" '=' '{' NonCacheableBody '}'
```

```
SectionDecoders
  : "decoders" '{' AnsiStatementList '}'
```

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

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

```
AnsiPragmaModuleId
  : AnsiPragmaModuleId ',' CompoundId
  | CompoundId
```

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

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

```
AnsiStringList
  : "string"
  | AnsiStringList "string"
```

```
Settings
  : SettingsSpecifier '{' SettingsBody '}'
```

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

```
AnsiSelectionStatement
  : "if" '(' AnsiExpression ')' AnsiStatement
  | "if" '(' AnsiExpression ')' AnsiStatement "else" AnsiStatement
  | "switch" '(' AnsiExpression ')' '{' AnsiLabeledStatementList '}'
```

```
AnsiExpression
  : AnsiAssignmentExpression
  | AnsiExpression ',' AnsiAssignmentExpression
```

```
OptionalSemicolon
  : ';'
  | %empty
```

```
AnsiDeclarator
  : AnsiVariableDeclarator
  | AnsiFunctionDeclarator
```

```
AnsiForExpression
  : AnsiExpression
  | %empty
```

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

```
CompileExpression
  : AnsiCompileExpression
```

```
AnsiAndExpression
  : AnsiEqualityExpression
  | AnsiAndExpression '&' AnsiEqualityExpression
```

```
Numways
  : "numways" '=' CompileExpression
```

```
OptionalComma
  : ','
  | %empty
```

```
Size
  : "size" '=' CompileExpression
```

```
AnsiEnum
  : "enum" '{' AnsiEnumeratorList OptionalComma '}'
  | "enum" ':' AnsiDeclarationSpecifiersBasic '{' AnsiEnumeratorList OptionalComma '}'
  | "enum" AnsiEnumId '{' AnsiEnumeratorList OptionalComma '}'
  | "enum" AnsiEnumId ':' AnsiDeclarationSpecifiersBasic '{' AnsiEnumeratorList OptionalComma '}'
  | "enum" AnsiEnumId
```

```
NonCacheableRange
  : CompileExpression ".." CompileExpression
```

```
AnsiDeclarationSpecifiers
  : AnsiDeclarationSpecifiersBasic
  | AnsiEnum
  | AnsiDeclarationSpecifiers AnsiEnum
```

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

```
AnsiParameterDeclaration
  : AnsiDeclarationSpecifiers AnsiId
  | AnsiDeclarationSpecifiers
```

```
SectionInstructions
  : "instructions" AnsiCompoundStatement ';'
```

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

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

```
RegisterAttribute
  : "write_enable" '=' CompileExpression
  | "default" '=' CompileExpression
  | "pipeline" '=' CompoundId
  | Overlap
  | Dff
  | Reset
  | ClockEnable
```

```
AnsiPragmaSpecifier
  : "#pragma simulator"
  | "#pragma compiler"
  | "#pragma module" AnsiPragmaModuleId
  | "#pragma unknown"
  | %empty
```

```
AnsiRelationalExpression
  : AnsiShiftExpression
  | AnsiRelationalExpression '<' AnsiShiftExpression
  | AnsiRelationalExpression '>' AnsiShiftExpression
  | AnsiRelationalExpression "<=" AnsiShiftExpression
  | AnsiRelationalExpression ">=" AnsiShiftExpression
```

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

```
Root
  : TranslationUnit
  | %empty
```

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

```
NonCacheableBody
  : NonCacheableBody ',' NonCacheableRange
  | NonCacheableRange
```

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

```
AnsiEqualityExpression
  : AnsiRelationalExpression
  | AnsiEqualityExpression "==" AnsiRelationalExpression
  | AnsiEqualityExpression "!=" AnsiRelationalExpression
```

```
EventSection
  : SectionSemantics ';'
  | SectionTiming ';'
  | SectionDecoders ';'
```

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

```
AnsiInitDeclaratorList
  : AnsiInitDeclarator
  | AnsiInitDeclaratorList ',' AnsiInitDeclarator
```

```
Setting
  : Id '=' SettingElement
  | Id SetAssign '{' SettingBoby OptionalComma '}'
```

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

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

```
AnsiIterationStatement
  : "while" '(' AnsiExpression ')' AnsiStatement
  | "do" AnsiStatement "while" '(' AnsiExpression ')' ';'
  | "for" '(' AnsiForExpression ';' AnsiForExpression ';' AnsiForExpression ')' AnsiStatement
```

```
SectionIfReplace
  : "if" '(' AnsiExpression ')' SectionIfReplaceBody
```

```
AnsiAttribute
  : "__attribute__(())" '(' '(' AnsiAttributeBody ')' ')'
```

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

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

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

```
AnsiLogicalOrExpression
  : AnsiLogicalAndExpression
  | AnsiLogicalOrExpression "||" AnsiLogicalAndExpression
```

```
SectionAssembler
  : "assembler" '{' SectionAssemblerBody '}'
```

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

```
AnsiParameterList
  : AnsiParameterDeclaration
  | AnsiParameterList ',' AnsiParameterDeclaration
```

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

```
AnsiCastExpression
  : AnsiUnaryExpression
  | '(' AnsiDeclarationSpecifiers ')' AnsiCastExpression
```

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

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

```
ExternSpecifier
  : "component"
  | "codal"
  | %empty
```

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

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

```
SectionPattern
  : "pattern" AnsiCompoundStatement
```

```
ScheduleClassBody
  : ScheduleClassBody ScheduleClassAttribute ';'
  | ScheduleClassAttribute ';'
```

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

```
AnsiCompoundStatement
  : AnsiPragmaSpecifier '{' '}'
  | AnsiPragmaSpecifier '{' AnsiStatementList '}'
  | AnsiPragmaSpecifier '{' AnsiDeclarationList '}'
  | AnsiPragmaSpecifier '{' AnsiDeclarationList AnsiStatementList '}'
```

11 ANNEX: CODAL KEYWORDS

Here is a list of the keywords in CodAL:

```
address
address_space
alias
alignment
allow_in_delay_slot
arbiter
arch
as
assembler
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
dataports
debugger
decoder
decoders
decoding
default
delay_slot
dff
direction
do
double
element
else
emulation
emulations
encoding
endianness
enum
event
extern
flag
float
for
identifier
if
inline
```

```
instructions
int
int_
interface
interfaces
label
latencies
latency
linesize
llvm_class
long
mapping
mask
memory
module
non_cacheable
numways
open
overlap
pattern
pc
peephole
peepholes
pipeline
port
register
register_file
replace
reset
return
roots
rpl_policy
schedule_class
semantics
set
settings
short
signal
signed
simulator
size
start
string
STRINGIZE
switch
symbol
timing
type
typedef
unaligned
unsigned
use
void
while
write_enable
```

12 ANNEX: TABLES, EXAMPLES AND FIGURES

12.1 List of Tables

12.2 List of Examples

12.3 List of Figures

13 GLOSSARY

A

ABI
Application Binary Interface: describes the conventions for which registers are used for return values, the stack pointer etc.

ALU
Arithmetic and Logic Unit

API
Application Programming Interface: a set of routines, protocols, and tools that gives one piece of software access to the functionality of another. For example, a SystemVerilog simulator could access information from a simulation inside Codasip Studio through Studio's API.

ASIP
Application Specific Instruction-set Processor

ASIP Architecture
The programmer-visible aspects of the ASIP.

ASIP Microarchitecture
A specific implementation of an ASIP Architecture. An ASIP may have several Microarchitectures, each of which can have a different size and performance. Each Microarchitecture will have its own CA model, but all of them will share the same IA model.

Assembler
A tool that takes in assembler code (usually .s or .asm format) and outputs object code (.obj).

C

CA
Cycle Accurate

CLI
Command Line Interface.

Co-simulation

The simulation of a system using multiple, independent simulators. The Codasip simulator is encapsulated in a wrapper which allows it to co-simulate with tools that run, for example, SystemC or RTL simulations.

Codasip Commandline

An alternative to the IDE that allows users to run Codasip's tools from a command line.

Codasip Studio

The IDE, the CommandLine and the Tools.

Compiler

A tool that takes in high level code such as C or C++ and outputs assembly code.

H

HDL

Hardware Description Language. Verilog and VHDL are examples of HDLs. They are used to describe electronic hardware (see also “RTL”).

I

IA

Instruction Accurate

IDE

Integrated Development Environment. Codasip's IDE is based on Eclipse and is one component of Codasip Studio.

ISA

Instruction Set Architecture

L

LAU

Least Addressable Unit. The smallest piece of data that the processor system can handle. This is typically a byte, since the smallest addressable unit of memory tends to be a byte.

Linker

A tool that takes in object code and libraries and links them together to create executable code (typically a .exe).

LLVM

The name "LLVM" is not an acronym; it is the full name of a project, described as a collection of modular and reusable compiler and toolchain technologies (http://www.llvm.org).

M

MP

Multi-Processor (in this context, multiple ASIPs in a single Platform)

N

Non-terminal

See the definition of "terminal".

P

Panel

An area within the IDE main Window containing one or more Views (the Views being accessible via tabs).

Perspective

A style for the IDE window designed for a particular purpose, such as Debug or Profiling. The Views available in each Perspective are different. The default Perspective is called "Codasip".

Project

An object in the Codasip Studio that groups together items related to a particular part of a design. Different types of Projects are defined, and each type has a certain set of possible actions associated with it. There are three main types of Projects in Codasip: ASIPs, Platforms and Test.

R

RTL

Register Transfer Level. Used in Hardware Description Languages (HDLs) to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived.

S

SDK

Software Design Kit. The tools needed to generate a test program and simulate an ASIP model. They include, for example, a compiler, an assembler and a simulator.

T

Terminal

Syntax definitions consist of statements that progressively define a language from a root down to the leaves, or "terminals" of the language. They are to be distinguished from rule names (also called non-terminal symbols), which label intermediate branches of the syntax. Terminal symbols are written exactly as they are to be represented, whereas rule names are given an arbitrary name that represents the branch.

TLM

Transaction Level Model. A simulation model where the lowest level of abstraction is a transaction. Hence, signal-level details are hidden in a TLM.

U

UART

Universal Asynchronous Receiver Transmitter

V

View

The lowest level of hierarchy within the IDE Window (the Window contains Panels which contain Views, each of which are accessible through its tab).

W

Window

The IDE top level, synonymous with the Workbench.

Workbench

The interface presented by an IDE to the user.