ADL Language Reference

This document describes the architectural description language ADL. The purpose of ADL is to describe the architecture of a processor and allow for the creation of various tools directly from this description, such as an assembler, a verification ISS, a high-speed ISS, etc. This language is only for architectural description and thus specifically does not encompass any cycle-accurate features. The intent is that the ADL may be used to generate a model which would work in conjunction with a micro-architectural description in order to produce a cycle-accurate model.

Author:	Brian Kahne
Contact:	bkahne@freescale.com

Table of Contents

- ADL Language Reference
  - 1 General Overview
  - 2 C++ Extensions
  - 3 File Format
    - 3.1 Keys/Commands Valid For All Defines/Modifications
    - 3.2 Attributes
    - 3.3 Definition Types And Their Keys
      - 3.3.1 System
      - 3.3.2 Core
      - 3.3.3 Architecture
        - 3.3.3.1 Instruction Field
        - 3.3.3.2 Sub-Instruction
        - 3.3.3.3 Instruction
        - 3.3.3.4 Register
        - 3.3.3.5 Register File
        - 3.3.3.6 Context
        - 3.3.3.7 Parameter
        - 3.3.3.8 Exception
        - 3.3.3.9 MMU
        - 3.3.3.10 Cache
        - 3.3.3.11 Memory
        - 3.3.3.12 Event Bus
        - 3.3.3.13 Group
        - 3.3.3.14 Assembler
        - 3.3.3.15 Real-Address Mask
        - 3.3.3.16 Effective-Address Mask
        - 3.3.3.17 External Resource
        - 3.3.3.18 Prefix Counter
        - 3.3.3.19 Relocation

            - 3.3.3.20 Other Definitions
- 4 Predefined Global Resources
    - 4.1 Global Variables
    - 4.2 Registers
        - 4.2.1 Reading and Writing Registers
        - 4.2.2 Delayed Register Writes
        - 4.2.3 Direct Register Accesses
    - 4.3 Memory
        - 4.3.1 Delayed Writes
    - 4.4 Local Memory
        - 4.4.1 External Register Functions
    - 4.5 MMU API
        - 4.5.1 MMU Lookup Identifiers
    - 4.6 Cache API
    - 4.7 Context API
    - 4.8 Global Functions
        - 4.8.1 External Pipeline Model API
        - 4.8.2 Rollback API
        - 4.8.3 Ignore-Mode API
    - 4.9 System-Call Support
- 5 Usage Notes
    - 5.1 32/64 Mode Behavior
    - 5.2 64-Bit Only Instructions
    - 5.3 Simd Registers and Instructions
    - 5.4 MMU Modeling
    - 5.5 Delay Slots
    - 5.6 Split Branches
    - 5.7 VLIW instructions
    - 5.8 Prefix Instructions
    - 5.9 Assembler Instructions
    - 5.10 Assembler Programming Rules and Post-Packet Assembler Hook
    - 5.11 Variable Width Instructions
- 6 Generators
    - 6.1 Assembler/Disassembler Generation
        - 6.1.1 Multiple Instruction Tables
        - 6.1.2 Debug Prints
    - 6.2 ISS Generation
        - 6.2.1 Usage
        - 6.2.2 Configuration File Syntax
        - 6.2.3 Time Tagged ISSs
        - 6.2.4 Graphite Integration
        - 6.2.5 Graphite Usage With LSF
        - 6.2.6 Graphite Peripheral Models
            - 6.2.6.1 The UVP Testwriter

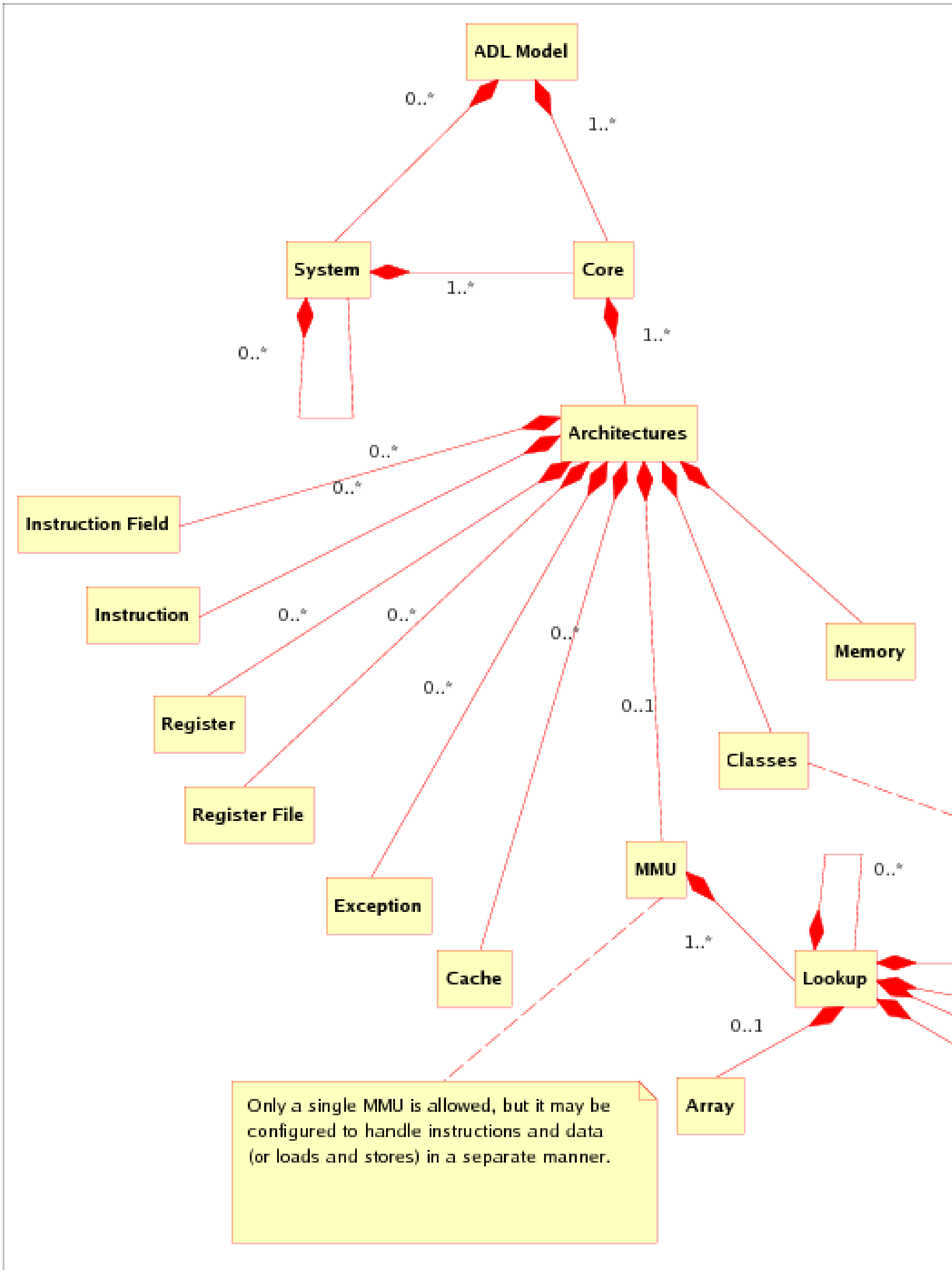
        - 6.3 Documentation Generation
            - 6.3.1 Usage
            - 6.3.2 Influential ADL Keys and Attributes
            - 6.3.3 Configuration File
                - 6.3.3.1 Special Transform Functions
    - 7 Example Code

1 General Overview

An ADL model consists of a system or core object. Systems may be made up of one or more cores or other systems and are used to describe multi-processor models. Core objects describe a single processor core and are made up of one or more architecture blocks. Each architecture block describes resources that it has (registers, instructions, exceptions, etc.) or which should be removed. Each type of object in the model may have a documentation string associated with it which may contain human-readable text. The idea is to use this for the purpose of literate programming: A documentation tool could extract this text, as well as other elements of the design, to create documentation of the model.

Each architecture block can be thought of as a *mix-in*: It does not have to be complete in and of itself and can thus describe a fragment of an architecture. For example, the Power SPE SIMD instructions would be described within an architecture block but a core using it would not be complete without also including the main Power architecture.

Graphically, a model looks like:


ADL Model
0..*
1..*
System
Core
1..*
0..*
1..*
Architectures
0..*
0..*
Instruction Field
Instruction
0..*
0..*
0..*
Register
0..*
0..1
Register File
Memory
Classes
Exception
MMU
0..*
Cache
1..*
Lookup
0..1
Array
Only a single MMU is allowed, but it may be configured to handle instructions and data (or loads and stores) in a separate manner.

ADL Model Organization

The ADL language is declarative, using blocks of extended C++ for describing the semantics of various resources, such as the actions of an instruction. The C preprocessor is used to build a description from multiple files using the include-file mechanism.

Declarations of resources in ADL take the form:

```
[define|defmod] (<type> = [name]) {
  <statement> | <key> = <value> | <define block> ;
}
```

The *type* specifier is required and specifies what is being declared, e.g. *reg*, *instr*, *removes*, etc. The name is optional in general, but is required for some items, such as instructions and registers. It may be either an identifier, a quoted string, or a list of identifiers or strings. If the name is a list, then the behavior is the same as if the same block were replicated for each element in the list.

Within a define body will be either statements, key/value pairs, or other define/ defmod blocks. Statements exist to allow for literate programming and to define various action hooks. As an extension of the C++ grammar, triple-quoted strings will allow embedded newlines. For example:

```
define (instr = bcctrl) {
  """
  Let the branch target effective address (BTEA) be calculated as follows:

  * For bcctrl[l], let BTEA be 32 0s concatenated with the contents of bits
    32:61 of the Count Register concatenated with 0b00.

  The BO field of the instruction specified the condition or conditions that
  must be met in order for the branch to be taken, as defined in `Branch
  Instructions`_.  The sum BI+32 specified the bit of the Condition Register
  that is to be used.
  """;
  ...
}
```

The contents of the string, from the point-of-view of the ADL front-end, are not important, but a documentation back-end might use reStructured Text as a mark-up format.

Generally, define blocks have key/value pairs for specifying information about a resource. For example, a register has a *fields* key for describing register fields, a *size* key for describing its width, etc. Instructions have a key for specifying the opcode, the instruction behavior, the format used by an assembler, etc.

Instructions' behavior is specified using blocks of C++ code, e.g.:

```
define (instr = addi) {
  "If RA=0, the sign extended value of the SI field is placed into GPR(RT)";
  size=32;
  bits=0x38000000;
  fields=(RT,RA,SI);
  action= {
    if ( RA == 0 ) {
      GPR(RT) = signExtend(SI,32);
    } else {
      GPR(RT) = GPR(RA) + signExtend(SI,32);
    }
  };
}
```

2 C++ Extensions

The intent of the ADL extensions is to add as little as possible to C++. Most of these features are needed for the purpose of implementing the declarative syntax used by ADL to describe the resources of the processor. Currently, the following extensions exist:

- Define blocks. Adds the keyword *define*. These are the primary means for describing architectural elements such as registers and instructions. These may appear at the top-level, may be nested, and may be interspersed with normal functions, classes, etc.
- Define-modify blocks. Same as above, except that we are issuing a modify request, rather than a define request. The keyword is *defmod*. Generally, a *define* with the same type and name as a previously encountered *define* will replace the prior definition, whereas a *defmod* will merge its contents with the existing definition.
- An anonymous function syntax exists using the keyword *func*. For example:

  ```
  write = func(unsigned value) { ... };
  ```

  An anonymous function taking no arguments is equivalent to a plain block of code. In other words:

  ```
  action = { ... };
  ```

  is equivalent to:

  ```
  action = func() { ... };
  ```

  If the constant `0` is used in place of a function, it has the effect of removing that entry. In other words:

  ```
  action = func() { ... };
  ```

  has the effect of setting `action` to the specified function. If the following is then specified:

  ```
  action = 0;
  ```

This clears out that function; the `action` key is then considered to be empty.

Note that this syntax is only allowed as the value for a key/value pair and may not be used within a block of action code.

- Multi-line quotes to support literate programming. The use of comments is not a good idea because of the fact that they are *comments* and thus should not be considered to hold important information. Instead, a define or defmod block may have a multi-line quote as the first element after the declaration. These quotes use `"""` as a delimiter, i.e. three double-quote marks.
- A bit vector class with arithmetic operators, called *bits*. The size of the bit vector is specified as a template parameter. This allows a partial specialization to be used for objects of 32-bits or less, resulting in the use of native machine arithmetic instructions. However, the same class can then be used for much larger integers with no difference in the interface.

  Individual bits and slices may be accessed by using the function call operator: *reg(20)* or *reg(20,21)*. Concatenation of bit vectors is handled using the `concat` function:

  ```
  concat(a,zero(10),b);
  ```

  A signed version of the bit vector class is also available, called *sbits*. These vectors behave like signed integers in C. This mean that sign extension is done automatically, for example, in assignment operation when the left value is larger in size than the right value. In addition, arithmetic and comparison operations are also performed using signed semantics.

  In order to refer to bit positions using little endian bit order one can assign to the core parameter *bit_endianness* value *little* (default is *big*). In this ordering bit posistions are counted from the right, so the bit number *0* is the least significant bit.
- Simple type inferencing:

  - *var <ident> = <expr>*: *<ident>* will be declared with the same type as that of the initialization expression.
- Labels are used as code-injection points via the aspect-oriented features of the language. This is not new C++ syntax, but its appearance may be unfamiliar to many programmers. For example:

  ```
  Loop1 : for ( ...) { ... }
  ```

  In this example, `Loop1` is a label that may be used to insert code via an *aspect*.

3 File Format

The C-preprocessor is used to combine together multiple ADL files into a single description. The *#import* directive is preferred in the general case because it automatically prevents the inclusion of a file multiple times. A description, then, after preprocessing, will consist of a series of *define* or *defmod* blocks, possibly interspersed by C++ code. The *define* and *defmod* blocks may be nested. For example, an architecture contains within it all of the defines specifying instructions, registers, etc.

A valid description must have at least one *core* define and may also have a *system* define to describe an MP system. A *core* block will list an architecture which it implements. These architecture defines (*arch* blocks) contain the bulk of the information, describing instructions, registers, etc.

Plain C++ functions are allowed within *arch* or *core* blocks. These are available as helper functions for code blocks specified within defines and may manipulate architected resources. For example, the following code sets a PowerPC condition register:

```
define (arch = power) {

  void setCrField(bits<3> field,bits<32> x,bits<32> y) {
    var cr = CR;
    bits<4> r =
      ( (x.signedLT(y)) ? 0x8 : 0) |
      ( (x.signedGT(y)) ? 0x4 : 0) |
      ( (x == y)        ? 0x2 : 0) ;

    cr.set(4*field,4*field+3,r);
    CR = cr;
  }

  ...

}
```

Enumerated type declarations are also allowed. The `enum` values may then be used within action code, or to initialize field values, such as cache, MMU, or event-bus fields. For example:

```
enum MsgType { Read, Write, Invalidate };
```

This can then be used within action code:

```
msg_t data;
data.type = Read;
msg.send(data);
```

Processing of the description occurs in two main phases: A first pass scans in all elements and does initial syntax checking. Next, a data model is formed by combining all architectures which are instantiated by each core. An architecture will be scanned in a linear fashion, with defines and modifies taking effect sequentially. A later define will replace an earlier one and a modify operation must follow the define of the object it wants to modify. In other words, a modification of the *MSR* register must follow its original declaration. Modifications differ from a duplicate define in that they merge their values rather than replacing. Also, any define or modify block may contain a special key of *remove*, which will remove the entire definition.

Due to the fact that this is a two-pass parser, the order of different types of resources does not matter. For example, while an instruction, in its encoding description, must reference valid instruction fields, those fields may be located, physically in the file, after the declaration of the instruction. This allows the user to structure a file as desired to aid in readability, rather than due to parsing requirements.

The following are the types of defines that ADL supports, along with the allowed keys. The front-end should ignore, possibly with a warning, unknown declarations and keys in order to enable forward compatibility. Keys may occur multiple times within the same block.

3.1 Keys/Commands Valid For All Defines/Modifications

- *remove* = <*int*>: Non-zero means to remove the definition.
- *removable* = <*true* | *false*>: If false, then the object may not be removed by subsequent architecture blocks. The default is *true*.
- *modifiable* = <*true* | *false*>: If false, then the object may not be modified by subsequent architecture blocks. The default is *true*.
- *remove* <*name*>: This command allows for the removal of a key in the current *define/defmod* and all prior instances. Subsequent *defines* or *defmods* will not be modified. For example:

  This will define a register and set an offset value:

  ```
  define (reg=foo) {
    offset = 32;
  }
  ```

  This will then remove the offset key, so that the register will not have an offset value set:

  ```
  defmod(reg=foo) {
    remove offset;
  }
  ```
- Definition: *aspect*: Define a code aspect. This allows a block of code to be inserted immediately after a label, or other defined insertion points, within a block of action code. This define block is valid within any define that contains hooks, e.g. instructions, registers, mmu lookups, caches, etc., or enclosing scopes of such blocks.

The usual code injection point is just a C label. Whenever a label is encountered, the system looks for aspects which contain a matching label. The search begins in the current define block and works upwards to enclosing blocks. If a match is made and any of the matching aspects have the **exclusive** flag set, then the search stops. Otherwise, the search continues to the outer-most scope.

All code for all matching aspects are inserted immediately following the label. The code within an aspect is only evaluated for correctness after it has been inserted and is evaluated within the context of the code into which it is inserted. Thus, an aspect can refer to resources, such as local variables, instruction fields, etc., which are valid within the code containing the label.

Additional insertion points include the start and end of instruction action code.

For example, This defines an instruction with a label of `Update`:

```
define (instr=add1) {
  fields=(OPCD(12),RT,RA,SI);

  action = {
    var carry = Carry(GPR(RA),SI,0);
    Update:
    GPR(RT) = GPR(RA) + SI;
  };
}
```

This modifies the instructions `add1` and `add2` to include an aspect which targets the `Update` label:

```
defmod (instr=(add1,add2)) {
  define (aspect=A) {
    labels = Update;
    action = { GPR(RA) = 0; };
    exclusive = true;
  }
}
```

The aspect defined above would match against the two specified instructions and would preclude aspects in the enclosing architecture block from being expanded because the **exclusive** flag is set.

- *action = func { }*: The code to insert.
- *labels = <ident>* | *<list(ident)>*: Specify target labels for this aspect.
- *attrs = <ident*|*ident(str*|*int[,...])>* | *<list(ident*|*ident(str*|*int[,...]))>*: Expand the aspect only if the parent object has one or more of the specified attributes. A functional form may be used to specify that the attribute must have the specified value(s).
- *if_defined = <ident>* | *<list(ident)>*: Specify symbols to check before expanding the aspect. If the symbols are not defined, then the aspect is not

expanded. For example, the user may list instruction fields and exception names; only if the exception exists in the current core and the instruction fields exist for the current instruction will the aspect be expanded.

Symbol types currently checked: Instruction fields and `blk` types (if the aspect is being expanded within an instruction) and core-level resources, e.g. exception names, register names, etc.

- *instr_pre_exec* = <*bool*>: Insert aspect at the start of instruction action code.
- *instr_post_exec* = <*bool*>: Insert aspect at the end of instruction action code.
- *exclusive* = <*bool*>: If true, on a match, do not proceed to the next outermost scope.
- *priority* = <*int*>: Specifies the priority of this aspect vs. other aspects, if multiple aspects are expanded at the same insertion point. An aspect without a priority is considered to have the lowest priority vs. other aspects that have priorities. The highest priority is 0. The order of expansion of aspects with equal priorities is undefined.

3.2 Attributes

Most definitions support attributes via the `attrs` key. An attribute is simply an arbitrary, user-defined symbol with an optional integer, string, integer-list or string-list value. These may be used by some generators to alter behavior or may be used by clients of the ADL database.

All attributes must be declared at the architecture, core, or system level before they may be used within a resource. This is done via the `attrs` key, e.g.:

```
attrs = (a,b,c);
```

This declares three attributes: **a**, **b**, and **c**.

To assign an attribute to a resource, use the `attrs` key. The value may be either a single attribute or a list of attributes. Values are assigned using a function call notation. For example:

```
define (instr=A) {
  attrs = (a,b(1),c("xyz"));
}
```

The type of the attribute's parameter is determined automatically. It is valid to use parametrized attributes without a parameter.

Some attributes are predefined and required by certain generators:

- *cia*: Specify that a register is the current-instruction-address register.

- *nia*: Specify that a register is the next-instruction-address register. Note that a single register may have both the *nia* and *cia* attributes.
- *condition*: Specify that a register is a condition register.
- *other*: Used to indicate the default instruction table.

Any object which supports attributes also supports the following keys:

- *inherit_attrs* = <*bool*>: True means that an object will inherit its attributes from a parent object, if applicable. For example, nested instructions inherit from the outer instruction and assembler shorthands inherit from their target instructions. Setting this to false means that no inheritance takes place.
- *remove_attrs* = <*ident* | *list(ident)*>: Remove an attribute. Reports an error if the attribute does not exist. This key may be used within any define/defmod object which supports attributes (has the `attrs` key).
- *remove_attrs_nc* = <*ident* | *list(ident)*>: Remove an attribute. No error is reported if the attribute does not exist. This key may be used within any define/defmod object which supports attributes (has the `attrs` key).

3.3 Definition Types And Their Keys

3.3.1 System

- Definition: *sys* = <*name*>: Specify a system. This is only needed for a description of an MP system.

  - Declarations: Declarations for systems or cores which constitute this system. For example, to declare two cores of type *Zen*:

    ```
    define(sys=ZenSystem) {
      Zen core0;
      Zen core1;
    }
    ```

    An array syntax may be used as well, when the type of the declaration is the same:

    ```
    define(sys=ZenSystem) {
      Zen core[2];
    }
    ```

    When an array syntax is used, the specified number of instances of the core or system are created and named by appending each item's index to its declaration name. In the example above, `core0` and `core1` will be created, of type `Zen`.
  - *type* = <*str*>: The type key's use is user definable. It is accessible within the ISS and may be used by test writers, such as the UVP test writer.
  - Definition: *shared*: Specify shared resources. Note that aliased registers and register files may not be shared.

    - *regs* = <*list*>: List shared registers.
    - *regfiles* = <*list*>: List shared register files.
    - *mmulookups* = <*list*>: List shared MMU lookup objects.
    - *parms* = <*list*>: List shared parameters.
    - *caches* = <*list*>: List shared caches. Once a given cache has become shared, all subsequent cache levels must be shared.
    - *mems* = <*list*>: List shared memories.
    - *eventbuses* = <*list*>: List shared event buses.
    - *ext_resources* = <*list*>: List of shared external resources.
  - Definition: *sys* = <*name*>: Systems may be nested. These are treated as local systems, visible only to the containing system block.

3.3.2 Core

- Definition: *core* = <*name*>: Specify a core. A valid description contains at least one core.

  - *archs* = <*list* **|** *id*>: One or more architectures which this core implements.
  - *type* = <*str*>: The type key's use is user definable. It is accessible within the ISS and may be used by test writers, such as the UVP test writer.
  - *instrtables* = <*list*>: List classes of instructions to be grouped into individual instruction tables. By default, only a single decoder is created. However, it is possible to create multiple decode tables using this field. Each element of the list is an instruction class declared using the *attrs* define, or the special keyword *other*, which will take all remaining instructions.

    For example:

    ```
    instrtables = (other,vle);
    ```

    This means that all instructions belonging to the *vle* class will be grouped into a table and all other instructions will be grouped into another table. The tables can be switched using the function *setCurrentInstrTable(<class>)* in an action block or a read or write hook.

    If instead the following line were specified:

    ```
    instrtables = vle;
    ```

    Then only the *vle* decoder would be created. The other instructions could be referenced via aliases but would not exist within the ISS's decoder.

    Note that the first element of the `instrtables` list will be used as the default instruction table when the core is reset.
  - In addition, all keys/defines for architectures may be used directly within the core. This can be useful when a minor modification must be made

to an existing architecture. Rather than creating an additional one-off architecture, the specification can be done directly within the core.

3.3.3 Architecture

- Definition: *arch* = <*name*>: Specify an architecture.

3.3.3.1 Instruction Field

- Definition: *instrfield* = <*name*>: Define an instruction field.
    - *allow_conflict* = <*bool*>: It allows the user to specify that a given non-opcode field may overlap with this field, such that normally a conflict in encoding would be present. However, due to how the field will be set, no conflict would result, and thus this should be allowed. For example, a set of instructions might have a field named **X**, which aligns with a 0 bit for the rest of the instructions which do not include **X**. Thus, if **X** is 1, no conflict results. However, since this is a non-opcode field, the encoder cannot know that **X** will always be 1, and thus normally a conflict would be detected.

      This parameter should be used with caution! It is not normally needed.
    - *assembler* = <*bool*>: Indicates that this is a field processed indirectly by the assembler. Generally, post-packet and post-instruction assembly hooks will set this field, rather than it being set directly through the syntax string.
    - *disassemble* = <*true*|*false*|*prefix*>: Indicate how to handle an instruction field by default, unless otherwise specified by an instruction's syntax string.

      true:
      : Disassemble normally.

      false:
      : Do not disassemble. This is overridden by an instruction's syntax, if specified.

      prefix:
      : Disassemble in front of an instruction's mnemonic. This occurs even if an instruction's syntax string also specifies it. This is useful for implementing instruction prefix syntax without explicit `%p` fields in all syntax strings. To support assembly, the user would normally mark this field as an `assembler` field and then set it via assembler instructions and the `pre_asm` hook.

- *doc_title* = <*string*>: Used only in documentation; a custom title for this instruction field.
- *doc_title_add* = <*string*>: Used only in documentation; extra title information for the instruction field. This is appended to the title, be it the standard one or a custom one specified via *doc_title*.
- *bits* = <*list*>|<*int*>: A list of integers representing the bit indices. A split field is represented as a list of lists, e.g. *SPRN = ((16,20), (11,15))*.
- *type* = <*ident*>: Specifies the type of this instruction field. Valid options are:

  - *regfile*: Specifies that the instruction field is associated with a register file.
  - *memory*: Specifies that the instruction field is associated with a specific memory space.
  - *instr*: Specifies that the field is instruction-type, i.e., associated with an instruction within the instruction table specified by a *ref* key.
  - *immed*: Specifies that the field is an immediate value (default). If *ref* is specified, the *type* can be infered from the resource referenced.
  - *ident*: where *ident* is another prefix instruction field. Only for prefix instruction fields, it denotes that a field implements another field.
- *ref* = <*ident*>: If the type is one which refers to another resource, such as *regfile*, *memory*, or *instr*, this key specifies the association. For example, a register file type must contain a *ref* key which specifies with which register file it is associated, and an instruction type must contain a *ref* key which specifies within which instruction table it is located. For prefix fields, this key refers to the pseudo field this field implments.
- *enumerated* = <*list(string*|*ident*|*list(string*|*ident))*>: Specify that the field is enumerated. Enumerations map to immediate values starting with 0. The list is a series of strings, identifiers or lists of strings or identifiers specifying the enumerated value. An empty string is allowed. The identifier "reserved" may be used to indicate a gap in the sequence. In a nested list all strings specify the same value, e.g. in *enumerated = ("a", ("b1", "b2"), "c")) a* specifies *0*, *b1* and *b2* specify *1* and *c* specifies *2*.
- *overlay* = <*bool*>: If true, then this field may overlap. The field is not decoded for execution, but is provided for assembler/ disassembler purposes. Its value is set after the underlying field and may thus be used to set additional bits, such as optional flags.
- *is_signed* = <*bool*>: If an immediate field, this specifies whether it is a signed quantity. The default is *false*. Within an instruction's

action code, the field's value will be expanded to the size of the *nia* register and sign-extended if this flag is set.

- *unsigned_upper_bound* = <*bool*>: If a signed immediate field, then this specifies that the allowed upper bound should be treated as an unsigned number, when performing range checking, such as by the assembler. This means that a value of 0xdeadbeef, for example, is allowed within a 32-bit field, even though, treated as a signed number, it exceeds the maximum signed value of 0x7fffffff. This feature is useful for allowing users to specify large constants, without the need to calculate the equivalent negative value.
- *is_inverted* = <*bool*>: If an immediate field, this specifies whether the value is inverted prior to being encoded. The default value is *false*. Within an instruction's action code, the value for this field will be inverted.
- *pseudo* = <*bool*>: Specifies that this field is not directly mapped to explicit bits. It may be used within syntax strings, by aliases, or by instructions in which their encoding is defined either by aliases or by the bit-mapped fields notation. If they appear directly within an instruction's encoding, then bit mappings will be assigned implicitly based upon the ordering of the fields. Default is false.
- *parm* = <*bool*>: Specifies that this field can not be mapped to explicit bits and can not be used within syntax strings. It can be used only by aliases and in action code. Default is false.
- *width* = <*int*>: Field width, in bits. Optional, if the field's bits are specified; mandatory for pseudo fields. The keyword *reserved* denotes a variable width field which is associated with all bits not associated with fields in the bit-mapped fields notation. Only one variable width field is allowed in an instruction.
- *size* = <*int*>: Field computed value, in bits. Optional, by default a field computed value is its bits; and its size equals its width.
- *prefix* = <*bool*>: If true, then this field may be used in prefix instructions. Prefix fields may be used only in prefix instructions, and all fields of a prefix must have this key set to true. Default, false. Prefix field inforamtion which is position and block independent can be accessed in action code.
- *indexed* = <*int*>: This *pseudo prefix* field is an indexed field of a given width. The field can be thought of as an array of elements of the specified width, where the specific field is selected based on the instruction's position within the VLIW packet. The width of the field, if not specified, is *paralle_execution* x *indexed*.
- *index_endianness = (big*|*little*|*)*: Endianness for automatically retrieving slices for indexed instruction fields. If not specified (the default), then the same endianness as for the core is used. If set to *big*, then the first instruction indexes from the left-most (most-significant) bits of the packet).

- *blk* = <*ident*> **|** <*list*>: Optional, specifies the logical blocks this field is associated with. If specified, must include the block of any instruction that uses it.
- *addr* = <*ident*>: Specifies that this field represents an address. Value values are:

  pc: PC-relative address. Within an instruction's action code, the value for the field will be the sum of the field's encoded value and the *cia* register.

  abs: Absolute address.

  none: Specify that this field does not represent an address.
- *offset* = <*int*>: Specify an implicit offset. Within an instruction's action code, the value for the field will be the field's encoded value plus the offset.
- *shift* = <*int*>: Specify a shift value for the field. Within an instruction's action code, the value for the field will be the field's encoded value shifted left by the specified number of bits. In the assembler and linker, the value will be shifted right by the specified number of bits when encoding the operand value.
- *display* = <*ident*>: Specifies the formatting to be used when displaying the instruction field. Allowed values are:

  - *hex*: Display the field in hexadecimal notation with a 0x prefix.
  - *dec*: Display the field in decimal notation.
  - *signed_dec*: Display the field in decimal notation with an explicit sign prefix: + if the value is positive, - if the value is negative.
  - *name*: If this is a register file field, display it using the register file's prefix, followed by the value of the field in decimal.
  - *def*: Use the default behavior.

  The default is to use *name* if the instruction field has a register file, otherwise to use *dec* if the instruction field is five bits or less in size, otherwise use *hex*.
- *table* = <*list(k-tuple***|***"reserved")*>: Specify that the field is an enumerated list of admissible resource tuples. Tuples map to indices starting with 0. The identifier "reserved" is used to indicate a gap in the sequence. The tuple elements may be accessed in the instruction *syntax* specifier or in the *action* code by using a function call notation. An instruction field X with a table of pairs of integer may be referenced in the action code by: tmp = X(0) + X(1), where X is a pair index and X(0), X(1) are its first and second elements, respectively.

- *fields* = *<list(int|ident|idnet(int)>*: Specify the value returned by the instruction field. By default, it equals the field's *bits*. This can be combined with nested fields to yield a larger, generated value or a nested field may use this to take a slice of a parent field. If the nested field itself contains a lookup table, then the lookup will be performed automatically when the field is referenced. In this situation, the table may only be one dimensional. For example:

```
define (instrfield = DaDb) {
  width = 5;    // number of  input  bits
  pseudo = true;

  // Various allowed combinations of registers.  Note that shortcuts exist,
  // demonstrating multiple enumeration strings for the same value.
  enumerated = (("d0::d2","d0:d1:d2"),
                ("d1::d3","d1:d2:d3"),
                ("d2::d4","d2:d3:d4"),
                ("d3::d5","d3:d4:d5"),
                ...
                "d0:d7:d15"
                );

  define(instrfield=Da) {
    size = 5;
    fields = (4,0);
    table = (0,1,2,3,4,5,6,7,8,9,10,11,12,12,0,2,4,6,8,10,1,3,5,7,9,11,0,3,6,9,0);
  }
  define(instrfield=Db) {
    size = 5;
    fields = (4,0);
    table = (1,2,3,4,5,6,7,8,9,10,11,12,13,14,2,4,6,8,10,12,3,5,7,9,11,13,3,6,9,12,7);
  }
  define(instrfield=Dc) {
    size = 5;
    fields = (4,0);
    table = (2,3,4,5,6,7,8,9,10,11,12,13,14,15,4,6,8,10,12,14,5,7,9,11,13,15,6,9,12,15,15
  }
}
```

In this example, the parent field is enumerated. Each nested field then refers to a one dimensional encoding slice which corresponds to the enumerated values. When the action code of an instructions references the nested field, e.g. **DaDb.Da**, then the result returned is the value of **DaDb** indexed into the table stored in **Da**.

Another method for using nested fields is to refer to a parent's table for lookup purposes. For example:

```
define (instrfield = DaDb) {
  width = 5;    // Number of  input  bits
  size = 5;   // Necessary so that the default width isn't taken from the
              // nested fields.
  ref = D;
  pseudo = true;

  table = ((0,1,2),
           (1,2,3),
           (2,3,4),
           ...
           (9,12,15),
           (0,7,15)
           );

  // Extra enumerations which act as shortcuts.
  enumerated = ("d0::d2",
                "d1::d3",
                "d2::d4",
                "d3::d5",
                "d4::d6",
                "d5::d7",
                "d6::d8",
                "d7::d9",
                "d8::d10",
                "d9::d11",
                "d10::d12",
                "d11::d13",
                "d12::d14",
                "d12::d15"
                );

  // This has the effect of enumerating the table using the specified formatting.
  syntax = ("%f:%f:%f",DaDb(0),DaDb(1),DaDb(2));

  define(instrfield=Da) {
    width = 4;
    fields = (DaDb(0));
  }
  define(instrfield=Db) {
    width = 4;
    fields = (DaDb(1));
  }
  define(instrfield=Dc) {
    width = 4;
    fields = (DaDb(2));
  }
}
```

In this example, each nested field reference's a particular slice of the parent's table. Some enumerations are specified explicitly, but the use of the syntax string with table-lookup references is used to implicitly create extra enumerations from the field's table. For example, the enumerations "d0:d1:d2" and "d1:d2:d3" are added (the *d* prefix is added implicitly due to the field referencing the **D** register file, which, is presumed to have a prefix of *d*. These are then combined with any existing enumerations, so that explicitly specified enumerations may act as shortcuts, e.g. "d0::d2", as the first enumeration, maps to a value of 0, as does the implicitly created "d0:d1:d2". Gaps in the explicit enumerations may be created by using the **reserved** keyword.

- Definition: *instrfield* = <*name*>: Nested instruction field. These are treated as local fields, and can be accessed from the containing field by their name, or by any instruction using the '.' hierarchical notation, e.g., DaDb.Db. Nested fields inherit some of the properties of the containing instruction. Therefore, can not redefine the keywords, pseudo, prefix, blk or set a reserved width. The *bits* and *fields* keys in a nested instruction field are relative to the containing instruction. For instance:

```
define(instrfield=DaDb) {
  bits = ((8,9),(0,1));

        define(instrfield=Tbl) {
    bits = (3,4);
          table = ((0,1),reserved,(2,3),reserved);
  }
  fields = (0,Db(0),1,Db(1));

        define(instrfield=Db) {
    fields = (0,2);
  }
}
```

The nested field *Tbl* uses bits *9* and *0* of the instruction. The nested field *Db* returns the upper three bits of the *DaDb* field.

- *value* = *<string>*: Optional, specifies default value for the field. For numeric fields, the default value is zero.
- *valid_ranges* = *list(<int>,<int>)*: Optional, specifies a list of inclusive ranges for allowed values. Currently, only unsigned fields are supported.
- *valid_masks* = *list(<uint>,<uint>)*: Optional, specifies a list of allowed masks. Syntax is ((pattern1,value1),...). Value of field is valid if *pattern_i & field == pattern_i & value_i* for some *i*.
- *action* = *<code>*: Optional. The value returned by the instruction field. The computed value should be assinged to the identifier *ThisField*, and expressions of the form *bits(hi,lo)* refering to the field's bits. Note, for most usages the *fields* suffices for describing this value. For indexed fields action code must take one parameter, representing the index, all other fields may have an action of arity 0.
- *syntax = (<string>, <fields ...>)*: Specifies how an instruction field is to be parsed by an assembler or printed by a disassembler. The syntax is a list of nested fields with their *fields* key defined. For example, *syntax = ("%f:%f", Da, Db)*. Either a nested field may be specified or a table-reference, e.g. *syntax = ("%f:%f", DaDb(0), DaDb(1))*. The two forms may not be mixed. If the latter is used, then the syntax string is used to generate enumerations based upon the table data, which may be combined with extra enumerations already specified.

The only valid control field is:

%f: Specifies an nested instruction field.

Expressions are not supported for instruction field syntax.

See definition of syntax for instruction for more details.

- *alias* = <*name*>: Specifies that this field is an alias to another instruction field. Useful when instructions in assembler are distinguished by instruction field syntax. Can be used in syntax similar to how expressions are used -- usign functional notation. Example:

```
define(instrfield=DaDb) {
  ref=D;
  define(instrfield=Da) { ... };
  define(instrfield=Db) { ... };
  syntax = ("%f,+%f",Da,Db);
}

define(instrfield=DaDb2) {
  alias = DaDb;
  syntax = ("%f,-%f",Da,Db);
}

define (instr=add_p) {
  ...
  syntax = ("add %f",DaDb);
}

define (instr=add_m) {
  ...
  syntax = ("add %f",DaDb2,DaDb(DaDb2));
 }
```

In this example the assembly line: `add d0,+d1` is *add_p* instruction and *add d0,-d1* is *add_m*.

3.3.3.2 Sub-Instruction

- Definition: *subinstr* = <*name*>: Define a sub-instruction. A sub-instruction allows you to decompose an instruction into a series of orthogonal sub-components. For example, suppose an architecture has a series of arithmetic operation, e.g. add, multiply, subtract, and a series of addressing modes, e.g. register, memory, indirect memory, etc. Writing by hand the cartesian product of all operations with all addressing modes would be tedious. Instead, you can create a set of sub-instructions for addressing and a set of sub-instructions for the operations, then group these together within an instruction. ADL will automatically generate the cross-product of these two groups.

  - *attrs* = <*list*>: List any attributes that this sub-instruction is associated with. Attributes will be inherited by all instruction built from this sub-instruction.

- *fields* = <*list*>: List of fields. Hard-coded values for fields are specified using the functional notation, e.g., *Op1(12)*. Fields may be specified using the bit-mapped notation, e.g., *fields = ((bits(0,5),RA), (bits(6),0xa), (bits(7,12),RB), (bits(13), reserved), (bits(14,15),b11));*, where are RA, RB are pseudo instruction fields, and the keyword *reserved* denotes that the bit is not used for encoding. The notation *b11* denotes the binary representation of *3*. Pseudo-fields may be repeated across sub-instructions.
- *syntax = (<string>, <fields ...>)*: Specifies how an instruction is to be parsed by an assembler or printed by a disassembler. Takes the form of a list, where the first element is a format string and the following items are opcode fields. See definition of syntax for instruction for more detail.
- *action* = <*code*>: The semantics of the sub-instruction. Instruction fields are accessible using their names and registers are also accessible using their names. The action may take additional arguments which are passed to it by the instanciating instruction, e.g., for subinstr X:

  ```
  action = func(bits<16> &rval) {

  }
  ```

  it is used in the instruction's action code:

  ```
  bits<16>   Y0;
  X(Y0);
  ```

  Note that the fields should not be specified in the action code definition, they are inherited from the instruction. Action code without arguments is specified using:

  ```
  action = {

  }
  ```

  If the action contains a *return* statement, it will abort instruction execution.

  When using defmod construct, one can type:

  ```
  defmod (instr = subinstr1) {
  ...
  }
  ```

  in order to refer to any instruction that is constructed using this subinstruction. Note, that usual syntax:

  ```
  defmod (subinstr = subinstr1) {
  ...
  ```

```
}
```

has different meaning, it modifies parameters of the single subinstruction.

3.3.3.3 Instruction

- Definition: *instr* = <*name*>: Define an instruction.
    - *doc_title* = <*string*>: Used only in documentation; a custom title for this instruction, to appear in any headings or links that refer to the instruction. By default, all documentation uses the instruction's syntax as the title, but it can be useful to override this for the parents of nested instructions (to name the instruction family). Note that all instruction titles will automatically append the block name, if any.
    - *doc_title_add* = <*string*>: Used only in documentation; extra title information for the instruction. This is appended to the title, be it the standard one or a custom one specified via *doc_title*.
    - *dsyntax = (<string>, <fields ...>)*: Specifies how an instruction is to be printed by a disassembler. This is normally not needed, as the disassembly can be derived from the `syntax` key. However, this may be required for some complex instructions.

      Takes the form of a list, where the first element is a format string and the following items are opcode fields or expressions. Expressions may invoke helper functions, but these helper functions must be self-contained and may not reference registers or other model resources.

      Valid control codes are:

      %f: Specify an instruction field.

      %i: Insert the name of the instruction.

      If an expression is used as an argument to a field, in the field list, then this is interpreted to mean that the field's formatting will be used to format the output. For example:

```
 define (instrfield = SCI8) {
   """
   A placeholder for the assembly representation of an SCI8-format constant.
   """;
   width = 32;
   pseudo = true;
   display = hex;
}
```

```
...
dsyntax = ("%i %f,%f,%f",RT,RA,SCI8(sci8(E_F,E_SCL,E_UI8)));
```

In the example above, **E_F**, **E_SCL**, and **E_UI8** are actual encoded fields, **sci8** is a helper function used to calculate the value to be displayed, and **SCI8** is a pseudo-field. Since the **SCI8** definition specifies hex formatting, the resulting disassembly will display the third field using hexadecimal notation.

- *width* = <*int* | *variable*>: Instruction width, in bits. The default value, if not specified, is determined from the max bit positions of the fields rounded up to nearest byte.

  An instruction with *variable* width indicates that all instructions' encoding should be ignored and ADL should internally encode instructions.
- *attrs* = <*list*>: Lists any attributes that this instruction is associated with. If it is a succint instruction then attribures will be inherited by newly built instructions. If it is a nested instruction, then it will inherit its attributes from its parent instruction. Attributes will also be inherited by assembler shorthands from the target instruction.
- *exclude_sources* = <*ident*|*ident(int*|*ident)*|*list*>: The user may use this key to explicitly exclude register resources from being considered as sources. This is primarily used when generating a transactional ISS.

  Normally, the ADL parser will automatically determine sources and targets. However, in some situations, the parser will be overly conservative. This will not affect functional correctness in a transactional ISS coupled to a performance model, but may result in erroneous stalls.

  For example, the following code will list `GPR(RT)` as a source and a target:

  ```
  var m = mode(true/*reg*/,false/*addr*/);
  var carry = Carry(GPR(RA),GPR(RB),0);
  GPR(RT)(m,regSize-1) = GPR(RA) + GPR(RB);
  setXerField(false/*ov*/,false/*so*/,true/*ca*/,carry);
  setCrField(m,0,GPR(RT),0);
  ```

  However, the `setCrField` function may have been written such that it only cares about the least-significant-half of GPR(RT), and thus GPR(RT) is not truly a source. In such a situation, the instruction may need to have this source explicitly excluded:

  ```
  exclude_sources = GPR(RT);
  ```
- *exclude_targets* = <*ident*|*ident(int*|*ident)*|*list*>: This is equivalent to *exclude_sources*, except that it applies to target resources.

- *fields* = <*list*>: A list of fields, sub-instructions, or bit-mapped fields. The sub-instruction fields are added to instruction. A sub-instruction group is substituted by each of its members. Hard-coded values for fields are specified using a functional notation, e.g. *Op1(19)*. Bit-mapped fields are specified using the notation:

  ```
  fields = ((bits(0,5),RA),(bits(6),0xa),(bits(7,12),RB),(bits(13),reserved),(bits(14,15),b1
  ```

  where are RA, RB are pseudo instruction fields. and the keyword *reserved* denotes that the bits is not not used for encoding. The notation *b11* denotes the binary representation of *3*. For this type of encoding, a value may be specified for a field using the following syntax:

  ```
  (bits(<src-start>,<src-end>),<field-name>(<field-start>,<field-stop>),<value>)
  ```

  The size of the instruction is derived from this value by simply examining the highest index and rounding up to the nearest byte. Any bits which are not specified in the *fields* line are set to 0.
- *allow_conflict* = <*bool*>: Allow encoding conflicts. This is similar to **allow_conflict** in instruction fields, but is useful when instruction fields are complex and conflicts do not map neatly to a single instruction field. Normally, ADL requires that, out of a set of instructions, that one instruction contains opcode bits which are a superset of all other opcode bits, or else that of the opcode bits which do overlap, there are different opcode values. This way, a straight-forward tree can be constructed looking at the smallest mask, making a decision on the opcode values, then descending to the relevant next level of the tree.

  In some complex encodings, this might not be the case. For example, imagine a situation where there are three instruction A, B, and C. All three have the same common opcode, but A contains certain fixed bits which do not overlap with the others, C contains certain fixed bits which do not overlap with the others, and B contains only the common opcode. Normally, this would be a conflict, but by setting this flag in instruction A or C, then the decision tree will split the opcode bits of A or'd with C, so that the decode tree will first check to see if A's specified bits are set. If not, then we will check to see if B's bits are set. If not, the C is selected.
- *blk* = <*ident*> | <*list*>: Optional, specifies the logical block this instruction is associated with; or, when present in a parent instruction, the list of all possible blocks used by the nested instructions. The list form is only needed when some nested instructions belong to different blocks. If blk is specified for an

instruction, the value must be identical to the block of all the instruction's fields; and, for nested instructions, must be equal to one of the blocks in the parent's blk list. **Important:** The list form is currently only meaningful in documentation; the rest of ADL will assume that the first block in a parent's list is "the" block for that instruction.

- *next_table* = <*code*>: Specify action to be taken to change the current decode table. This function should be very simple, as it is used by both the dissasembler and the simulator and thus does not have access to registers. It may examine instruction fields and call a limited set of functions, such as setting a new instruction table using `setCurrentInstrTable` and indicating the end of a packet via `setPacketPosition()`.
- *prefix* = <*bool*>: Optional, specifies that this instruction econdes a prefix and should be interpreted as such only if it is first instruction in VLES, or second instrction in VLES and the first one was a prefix. Note, the same encoding may be used for an instruction and a prefix. refix instruction action code can not modify registers or memory and are used to encode information used by other instructions in the VLES.
- *prefix_counters = ((<prefix-counter>,<field>) [, ...])*: Bind prefix counters to prefix fields. This means that the specified field will be used to index into the specified field during decode. This parameter is often used within a parent-instruction type, e.g. a StarCore instruction of type **OneWord** which specifies how prefix bits and actual fetched-bits are arranged within an instruction.
- *prefix_counter_incr = (<prefix-counter> [, ...])*: List prefix counters to be incremented when the specified instruction is decoded.
- *prefix_counter_decr = (<prefix-counter> [, ...])*: List prefix counters to be decremented when the specified instruction is decoded.
- *pseudo* = <*bool*>: The instruction defines a template instruction with no syntax or

  action code. Template instructions define how the full instruction encoding is composed from prefix fields and the instruction's encoding.
- *reserved_bits_value* = <*0*|*1*>: Specify the value to use for instruction reserved bits. The default is 0, but this flag allows this to be changed to a value of 1. If not set in the instruction then the default value from the core is used.
- *type* = <*ident*>: The type of *pseudo* instruction this instruction implements.
- Definition: *instr* = <*name*>: Define a nested instruction. Nested instruction use to group together an instruction and its aliases.

Therefore, a nested instruction can use only the fields, syntax and alias keywords. The alias should refer to the encapsulating instruction. The name of a nested instruction is global and is not affected by the nesting.

- Definition: *subinstrs* = <*name*>: Define a group of sub-instructions.

  - *subs* = <*list*>: List any sub-instructions in this group.
- *names* = <*list*>: List of names for instructions generated from the *subinstrs*.

  It may only be used with the *subinstrs*. The identifer "reserved" may be used to indicate that an instruction should not to be generated from the corresponding combination. The instruction are generated lexicographically by their occurrence in the fields entry, left-to-right within a *subinstrs* group. If this key is omitted, then names for the generated instructions are created by concatenating the instruction name with each sub-instruction name, separated by "_".
- *assemble* = <*bool*>: This instructs ADL to not include the instruction in the assembler. The instruction may only be generated as a result of a macro.
- *disassemble* = <*bool*>: This is a hint which tells ADL whether to exclude this instruction when attempting to disassemble an opcode. It may only be applied to aliases.
- *syntax = (<string>, <fields ...>)*: Specifies how an instruction is to be parsed by an assembler or printed by a disassembler. Takes the form of a list, where the first element is a format string and the following items are opcode fields, sub-instructions, or subinstrs. Nested instructions will inherint the syntax in case they do not have their own syntax definition. for example, *syntax = ("bc %d, %d", BO, BI)*.

  Valid control codes are:

  %f: Specify an instruction field.

  %p: Specify an order independent enumerated instruction field.

  %i: Insert the name of the instruction.

  Fields marked by %p can be specified in any order.For example: Suppose that F1 can be ".right" or ".left" and F2 can be ".up" or ".down" and *syntax = ("%i%p%p %f,%f,F1,F2,R1,R2);* Then, both *foo.up.left r4,r5* and *foo.left.up r4,r5* are valid and equivalent. Note that there can be up to one such squence of "%p" codes which must (if present) follow immediately after the instruction name.

Nested instruction fields are referred as in the following example:

```
syntax = ("%i %f,%f",DaDb.Da,DaDb.Db);
```

where the field DaDb has nested instruction fields `Da` and `Db`.

A group of nested fields can encode a single field value. The field `DaDb` could encode multiple nested fields. For example, a pair of nested fields `Da` and `Db` which are explicit in the syntax of the instruction:

```
add d0,d1,d3
```

while the register pair d0,d1 is encoded by a single field. This instruction syntax is:

```
sytnax = ("%i %f,%f", DaDb, Res);
```

while the syntax of instruction field `DaDb` is:

```
syntax = ("%f,%f", Da, Db);
```

describing how the `DaDb` field is assembler or disassembler.

The field `DaDb` can only encode part of all possible pairs, for example only pairs `Da, Db` such that `Da` is less than or equal `Db`. The following expressions handle this case:

- `reverse(%f)`: tuples encoded by the field in reverse order.
- `symmetric(%f)`: tuples encode by the field and its reverse tuples.

Instruction fields which are table based use a function call notation to specify the element to be encoded. For example, if `X` is a two-element table-based field:

```
syntax = ("foo %f,%f",X(0),X(1));
```

The optional <*expressions*> list, where each expression can be used to specify value for instruction fields that are not mapped directly by the syntax string. For example:

```
syntax = ("foo %f", X,Y(X >> 2 - cia()));
```

where X and Y are instruction fields, X may be a pseudofield but Y must be a real instruction field. The special function `cia()` stands for the program counter. **Note:** Expressions are not supported for nested instruction fields.

Fields that do not appear in the *<fields ...>* list must be defined by an expression. Expressions can be simple arithmetic expression consisting of constants or field names and is written in a functional form. For example, instruction "foo" has three instruction fields and the following syntax:

```
syntax = ( "foo %f,%f", A, B, C(A + B));
```

This means that value of field 'C' is not explicitly defined by the assembly string but derived from values of 'A' and 'B'.

The ternary `a ? b : c` construct is also supported. One pseudo-field may implicitly define two or more fields. For example:

```
syntax = ( "foo %f", P,
           A(P > 0 ? 0 : 1),
                       B(P > 0 ? (P >> 1) : ~(P >> 1) ))
```

If sub-instructions are used and a syntax string is specified, then all sub-instructions must have a syntax definition. In this case, the sub-instruction in the syntax format is replaced by the syntax of the sub-instruction. For example, if X is a sub-instruction with syntax:

```
syntax = ("%f,%f", BO, BI)
```

and instruction Y uses, and has the syntax:

```
syntax = ("foo %f,%f", X, RA)
```

it will be expanded to:

```
syntax = ("foo %f,%f,%f", BO, BI, RA)
```

Subinstructions will be substituted by each of their members.

Another possibility is to write a list of syntaxes. For example:

```
define(instr = foo) {
      names = ("aa","bb",reserved,"cc");
      syntax = ("aa %f %f",A,B,
                 "bb %f %f",A,B,
                     "cc %f"   ,A);
}
```

This example demonstrates the case where the third instruction (`cc`) the field `B` is an opcode, so that the first approach will not work.

Note that the name of the instruction is not implicit; it must be included within the string, as shown in the example. The default syntax string, if one is not specified, is to have the name of the instruction, followed by the instruction fields listed in order.

A few words on how the disassembler generation algorithm works. It tries to invert expression based on binary encoding.

First of all if the expression maps one variable to one field, the algorithm will try to invert it. For example `X(P >> 2)` will produce `P = (X<<2)`. Currently '+', '-', '<<', '>>' and '~' are supported.

The case of ternary operator is more complicated. Here is an example:

```
syntax = ("do %f,%f", A, P, BS(P>0xFFFF ? 1 : 0), IMM(P>0xFFFF ? P>>16 : P));
```

Instruction "do" has three instruction fields: A, BS and IMM. P is a pseudo field.

First, for each expression we determine its type. The possible types are:

1. *To number*: This means that leaves of the tree built by the ternary operator are numbers and do not need to be inverted. For example: `BS(P>0xFFFF ? 1 : 0)`.
2. *Simple function*: For example `A(A+1)`. This expression can be easily inverted.
3. *Complex functions*: This means that the original value may be mapped to several possible simple functions. From our example: `IMM(P>0xFFFF ? P>>16 : P)`. In this case it is not possible to understand the original value of `P` based only on the value of `Y`.

   The algorithm traverses the tree built by the ternary operators (they can be nested) and saves all possible function outcomes that can occur. In the example for the third type we would save the list consisting only of two elements: `P>>16` and `P`.

   Each expression of a such list is a simple function that can be inverted. Now, given the binary encoding of the field we have several possibilities for the original value. In our case it is `IMM` or `IMM<<16`. To select the right one we use expressions of the first type. We check one by one all possible values and the one that is consistent with values of the "checker" fields is chosen to be the right one. In our example we put both "IMM" and "IMM<<16" into expression for "BS" and check it against actual value of "BS".

If no "to number" checker field is available then we take all the expressions where this field is used as input and use those expressions as checkers. This algorithm will not work correctly for every possible syntax expressions but usually it is enough for most assembly constructions.

- *action* = <*code*>: The semantics of the instruction. Instruction fields are accessible using their names and registers are also accessible using their names.

  If the action contains a *return* statement, it will abort instruction execution.

  If the instruction is defined in terms of sub-instructions. Their action code can be called using function call notation, e.g.:

  ```
  bits<16>  z;
  X(z);
  ```

  Note that the sub-instruction action needs to be called explicitly. The sub-instruction fields are implicitly passed to the action code.

  If the instruction contains micro-operations (instruction fields mapped to instruction tables), then those may be either explicitly invoked with the **run** or **run_commit** functions, or implicitly by omitting the action function, in which case the micro-operations will be executed sequentially. In this latter case, a prior instruction, such as a branch, may control which micro operations execute by calling **setMicroOpOffset**.
- *assembler* = <*code*>: Defines code that directs assembler instruction handling by the asembler. The code can access assembler parameters and prefix variables using their names. If *action* is defined, the *assembler* code can access the instruction's fields using their names. It can be used to handle assembler instruction side-effects, and to define assembler directives. If the *queue_size* defined in the assembler configuration is larger than *1*, prefix variables are accessed as array elements where index *0* denotes current packet/instruction.
- *asm_rank* = <*int*>: Define the relative ordering of this instruction versus other instructions of the same name. By default, the assembler generator uses various heuristics to try and order instructions, such as preferring more enumerated fields vs other

types of fields. However, sometimes these heuristics are incorrect. The **asm_rank** key allows the user to override this behavior. A lower value means a higher priority; the default is 100.

- *asm_rules* = <*list*>: List any programming rules to be checked after the instruction is assembled.
- *alias* = <*call* **|** *list(call)*>: The function name (or names) must be that of another instruction already defined. Each argument of the function call is a mapping of the alias's fields to the target's fields, which uses function-call notation of the form `target-field(alias-field)`.

  For example:

  ```
  define (instr=se_add) {
    fields = {Op8(4),RY,RX};
    alias = add(RT(RX),RA(RX),RB(RY));
  }
  ```

  If the name of the alias target is not a simple identifier, then it must be quoted, e.g. `alias = "addic."(RT(RX),RA(RX),RB(RY));"`.

  Valid values for the arguments of the mappings are:

  - Constant integer: The target's field will always receive this value. For example: `RA(0)`. The `RA` field will always be set to 0.
  - An identifier: The identifier must name a field in the alias, or a target's field, in the case of a shorthand. For example: `RB(RS)`. The `RB` field will be set to the value of the `RS` field.
  - An expression. The expression may consist of simple arithmetic operators, constant integers, and field names. The value for the field will be computed using this field. For example: `ME(0-SH)`. The `ME` field will be set to the result of the expression `0-SH`.

    Several functions are provided for supporting various operations such as bit mask encodings, where an alias source field is specified as a bit mask, but the target field is expected to be an index:

    - *count_leading_zeros(n,s)*: Return a value counting the number of bits in `n`, where `n` is of size `s` bits, before the first 0 -> 1 transition, starting with the most-significant bit.
    - *count_trailing_zeros(n)*: Return a value counting the number of bits in `n` after the last 1 -> 0 transition, starting with the least-significant bit.

  - *count_ones(n)*: Returns the number of 1 bits in n.
  - *cia()*: Return the current program counter. For assemblers, this corresponds to a simple line count.

  In addition, user-defined helper functions may be called in order to implement complex algorithms.

Automatically inverting expressions using these operators for disassembling is not currently supported.

If the *fields* parameter is omitted, the instruction is considered a short-hand notation for another instruction. The fields will be extracted from the alias definition. For example:

```
define (instr=li) {
  alias = addi(RD(RD),RA(0),D(D));
}
```

The opcode fields will be those of *addi* and the remaining fields will be *RD* and *D*. In this case, a value of 0 will be used for the *RA* parameter.

Short-hand instructions are meant mainly for use by the assembler and other such tools, rather than ISSs. That is, they are not meant to be disassembled and executed by a model. Aliases which do define fields, however, are meant to be executed. The use of helper functions in alias expressions is only supported for short-hand instructions.

A list of instructions may be specified if this is an assembler shorthand, which allows for the simple assembler macros to be created. For example:

```
define(instr=add2) {
  syntax = ("add %f,%f,%f,%f",R1,R2,R3,SI);
  alias = ( addi(RT(R3),RA(R3),SI(SI)), add(RT(R1),RA(R2),RB(R3)) );
}
```

Instructions using subinstrs are expanded to multiple instructions, therefore, one can not define a short-hand to them.

- *alias_action = func(const InstrArgs &) {}*: If this instruction contains no fields, then the behavior is slightly different. In that case, this function acts as a complex alias expansion. The function must take a constant reference to an **InstrArgs** object, which is a vector of the instruction's values. Currently, only numeric operands are supported. The function must then return an IntrBundle object containing the actual instructions to be assembled. Use createInstr() to create these instructions.

- *dependencies* = <*block*>: Define resource dependencies. This allows an ISS generated with **--dep-tracking** to correlate resource reads with resource writes. Dependency data is described as a series of C++ statements within a brace-delimited block:

  ```
  define (instr=...) {
    dependencies = {
      GPR(RT) = ( GPR(RA), GPR(RB) );
    };
  }
  ```

  Each expression in the dependencies block consists of an expression with the lhs identifying the target and the rhs listing all sources as a comma separated list.

  The syntax for an item in the expression may be:

  - Register or register
  - Register-file(index)
  - Memory

  If a register-file is specified without an index, then this means that any read of that register file is a dependency. Otherwise, an index may be specified. The index takes the form of an expression consisting of constants and instruction fields. Likewise, referencing a register on the lhs means that any read is a dependency.

  For memory, each memory operation is taken to refer to the next register operation, e.g. GPR = Mem means that each memory read will be a dependency for the next GPR write.

  For a POWER condition-register operation, the dependency list would simply be:

  ```
  dependencies = {
          CR = CR;
  }
  ```

  Basically, the write to the condition register depends upon any read of the condition register.

  For a POWER add operation:

  ```
  dependencies = {
          GPR = GPR;
  }
  ```

  Again, any read of the GPRs, GPR(RA) and GPR(RB), is a dependency for GPR(RT).

For a POWER load with update, more must be specified:

```
dependencies = {
        GPR(RT) = Mem
  GPR(RA) = GPR(RA)
}
```

In other words,the target GPR depends upon memory, but the base register depends only upon the base register.

In some cases it is necessary to indicate which register or memory read or write is being targeted. To describe this, sequence numbers are used, denoted as $n, where n is the sequence number. For example:

```
dependencies = {
        LR = Mem($1);
  CTR = Mem($2);
  CR = Mem($3);
}
```

This describes a situation in which a series of memory reads target specific registers.

After a hit, i.e. a write occurs to a target, the dependency list is saved and cleared. This means that repeated operations, such as for POWER's lmw instruction, will report only the last memory read as a dependency, as expected.

- *enable_post_fetch* = *<bool* | *func() {}>*: Enable or disable post-fetch logic for this instruction. The default is **true**. By default, this logic is enabled and generally increments the next-instruction-address register (program counter). The logic may be overridden by defining a post_fetch hook. Setting this flag to false disables the generation of this code. Alternatively, an enable predicate may be specified, which allows for dynamic control over the post-fetch logic. The function should return a boolean value indicating whether to call the post-fetch logic (true) or not (false).

3.3.3.4 Register

- Definition: *reg* = *<name>*: Define a register. The register name must be a valid C++ identifier and may be referred to within action code by using its name.

    - *attrs* = *<list>*: Lists any attributes that this register is associated with. This is either a single identifier or a list of identifiers.
    - *width* = *<int>*: Specifies the register width in bits.

- *offset* = <*int*>: Specifies a starting bit offset. For example, this lets the user express the fact that the register is 32-bits wide, but its starting bit should be 32, rather than 0. The bits for fields are relative to this offset.
- *alias* = <*same as for read or write alias*>: If a register is a simple, direct alias of another register file or register-file element, then the *alias* key may be used. This is equivalent to defining a read and write hook which both have an *alias* key.
- *alias_slice* = <*same as for read or write slice*>: This just provides a shorthand way of defining an alias slice, as opposed to creating a read and write hook and using the *slice* key.
- Definition: *field* = <*name*>: Define or modify a register field. A register field is just a subset of bits within a register which is assigned a name and may be directly read or written. Refer to Registers for information about how to use fields within action code.

  - *attrs* = <*list*>: Lists any attributes that this field is associated with.
  - *bits* = <*list*>|<*int*>: A list of integers representing the bit indices which constitute the field. A split field is represented as a list of lists, e.g. `bits = (1,(11,15),(16,20))`. The first element of the pair must be less than the second element and must be greater-than-or-equal to the offset parameter (or 0, if no offset is specified). Indexed field cannot be split.
  - *indexed* = <*int*>: As an alternative to specifying a bit range, the user may specify that this is an indexed field of a given width, where the size of the register is a multiple of this width. The register can then be thought of as an array of these fields, where the specific field can be selected dynamically. Multiple indexed fields of varying widths can be used to represent the fact that a vector register is a union of different data types.
  - *readonly* = <*bool*>: The field is read-only. On a normal write, i.e. no write hook is defined for this register, this field will not be modified.
  - *reserved* = <*bool*>: The field is reserved, meaning that it is always 0. This can be used to override a definition found in a parent architecture, such as for when a particular design does not implement the field.
  - *writable* = <*bool*>: Only relevant for read-only fields. If a field is marked as *readonly*, then it normally cannot be written. Setting *writable* to true means that a specific write to this field is allowed, while full writes to the register will mask the bits

for this field. For example, for field X marked as *readonly* and *writable*, the following will set register's FOO's X field:

```
FOO.X = 1;
```

whereas the following will mask field X:

```
FOO = 0xdeadbeef
```

If fields are defined and they do not cover all bits in the register, then the bits not mentioned are regarded as reserved and will be tied to 0.

As currently envisioned, accessing fields will still utilize the read and write hooks. The entire register will be accessed, then the relevant slice will be returned.

The *remove* key may be used to delete a field. A modify operation may be used to alter a field. A duplicate definition will generate an error.

- *log_name* = <*ident*|*str*>: Specifies the name to be used when the register is logged via the various trace outputs and how the register may be set using the debugging interface. This name overrides the default, which is simply the register's name as specified in the definition.
- *pseudo* = <*bool*>: If non-zero, indicates that this is an aliased register and does not actually represent an actual storage element. This item necessitates a read and write hook.
- Definition: *read*: Specify an action to be taken on a read of this register. The *remove* key may be used to remove a read hook.

  - *alias* = <*ident* | *ident(int)* | *ident(int).ident* | *ident(int).ident(int)*>: If specified, then a read will access the specified register instead. If the alias is to an element of a register file, then the `ident(int|'child_id')` syntax must be used, where *ident* is the name of the register-file and *int* is the index of the element to be aliased.

    Also supported is the use of the special identifier `child_id`. This means that the index of the alias will be equal to the core's index within the parent system's list of children. This is useful when modeling a simultaneous multi-threaded system. For example, each core's program counter might be an alias into a register-file containing all program counters in the system.

    If referencing a specific element of a context, then the `ident(int).<rest>` syntax is used: The `ident` specifies the name of the context and the `int` specifies the element of the context.

  - *slice = (<int>,<int>)* | *((<int>,<int>),<int>)*: Only valid if *alias* is specified. Indicate a slice of the aliased register to use on a read or write. If the second form is specified, then the third integer specifies a shift. The slice is taken from the alias target, then left-shifted to form the value for this register.
  - *action = func([unsigned start,unsigned stop]) { }*: If a special action is to be taken when a value is read from the register, the code can be placed here. The underlying register is accessible using its name, unless *pseudo* was specified, in which case no underlying register exists. If the optional *start* and *stop* parameters are present, then the precise bits of the read action are specified, such as when a slice or field is being read.

- Definition: *write*: Specify an action to be taken on a write of this register. The *remove* key may be used to remove a write hook.

  - *alias = <same as for read>*: Same as *alias* for *read*, except that this applies to writes.
  - *slice = (<int>,<int>)*: Only valid if *alias* is specified. Indicate a slice of the aliased register to use on a write.
  - *immediate = <bool>*: If true, then if this is a parallel architecture, then the write-hook executes immediately, while the actual update of the register is delayed until the commit-phase, at the end of the packet. If false (the default), then the write-hook executes during the commit-phase. If the `ValueBundle` syntax is used, then a write-hook cannot have the immediate flag set to true. If not specified, then the hook inherits its value from the architecture-level flag `immediate_writehooks`.
  - *ignore = <bool>*: If true, this designates a read-only register (any write is ignored).
  - *action = func(bits<width> value [,unsigned start,unsigned stop]) { }*: If a special action is to be taken when a value is written to the register, the code can be placed here. The underlying register is accessible using its name, unless *pseudo* was specified, in which case no underlying register exists.

    For parallel architectures packet-based register update is supported. This allows for multiple register updates by instructions in the same execution packet. The write hook determines the value actually written based on all updates. The argument is such a case if a *ValueBundle*. A ValueBundle has an STL vector interface, and its elements have the interface:

- *mask()*: the write mask associated with element.
- *value()*: the value written expanded to register width.
- *addr()*: the address of the insturction issuing this write.
- *index()*: the register file element's index, zero for registers.

Within the hook, the following function is valid:

```
void updateReg(const ValueBundle &);
```

This takes a ValueBundle object and applies all of the updates to the hook's register. Use this helper function in order to avoid having the static usage-tracking system think that a full register update is being performed, when in fact only partial accesses are being made, as specified by the instruction which made the write.

If two extra unsigned parameters are supplied, then the inclusive bit-indexes of the range to update are supplied. For example, if the user has updated a field, and the write hook contains these indexes, then the field's bit range will be supplied to the write hook.

- *reset* = *<int>*| *func([unsigned context_index]) { }*: Specify a reset value for the register. If not specified, the power-on-reset value is 0 but the value will remain unchanged for reset operations generated by the core (caused by calling `resetCore` within action code).

  This may be a code sequence to handle more complex reset sequences, such as those that depend upon external values. The function should not return a value; rather, it should write to the register to set a value. If the register belongs to a context, then the context ID is supplied as a parameter to the function in order to allow for context-dependent values.
- *external_write* = *func(bits<width> value) {}*: Define side-effects to occur when a value is written to the register via the `IssNode::setRegExt` function. This is useful, for example, for system models, where an interface of some sort may set a register value, with various side effects.

- *serial* = <*bool*>: In parallel execution mode this register is updated sequentially, i.e., immediately. Default is false.
- *enumerated* = <*list(ident)*>: Specifies that the register is enumerated. Enumerations map strings to values starting with 0. The list is a series of identifiers specifying enumerated values. The identifier "reserved" may be used to indicate a gap in the sequence. Enumerated and corresponding numerical values are interchangeable. Numbers can be used for values that aren't enumerated. An enumeration string can be used in the register scope; other scopes require that it be quantified with a register name. For instance:

```
define(reg=R) {
  width = 32;

  enumerated = ("Normal","DelaySlot");

      reset = Normal;
}

post_fetch = func(unsigned s) {

      if (R == R::DelaySlot) {
    R = R::Normal;
  }
}
```

3.3.3.5 Register File

- Definition: *regfile* = <*name*>: Define a register file. This basically follows the format of a register, with a few modifications. The register name must be a valid C++ identifier and may be referred to within action code by using its name.

  A register file can be thought of in two ways, depending upon whether it is sparse (contains **entry** definitions) or non-sparse (contiguous). A sparse register-file is essentially an interface to a set of real resources (registers or elements of a register-file), whereas a non-sparse register-file is essentially an array of registers.

  In some applications, it is important to hide from an external application whether a set of storage elements in a processor are modeled as a series of individual registers or as a register-file. A sparse register-file never logs accesses itself, but rather defers that to the actual element being accessed, whereas a non-sparse register-file does lot accesses to itself. If a register-file is marked with the **log_as_reg** attribute, however, then it is logged as a register, with the index appended to the name.

For example, a design might contain a series of registers which are very similar, e.g. ABC0, ABC1, ABC2, ABC3. While documented as being individual registers normally accessed through a sparse register-file, the modeler might wish to model these as a register-file named ABC with a size of 4, rather than a series of registers. However, this fact can be hidden from an application using the model through the use of the **log_as_reg** attribute. Accessing the second element of the file will result in a register-write logging call with a name of "ABC1".

- Definition: *entry* = <*int*>: Specifies that this is a sparse register file and that the following register is a member of this register file. Sparse register files are collections of registers which are defined elsewhere. Currently, sparse register files, i.e. any register file with *entry* definitions, may not have per-register-file read or write hooks. The integer value is the index of the register in this register file.

    - *reg* = <*ident* | *ident(int)*>: Specify the name of the member register or a register file. If a register file, then a single element of the register file may be specified using the `ident(index)` syntax. The special identifier `child_id` may be used for the index, to support child-relative indexing.

      If a register-file is specified with no index, then each register-file element is mapped to consecutive entries, starting with the entry which specifies the register-file. For example:

      ```
      define (entry=30) { reg=FOO; };
      ```

      Assuming that **FOO** is a register-file of 4 elements, then entry 30 will map to **FOO(0)**, entry 31 will map to **FOO(1)**, entry 32 will map to **FOO(2)** and entry 33 will map to **FOO(3)**.
    - Definition: *read*: This definition allows the user to apply a read hook to a read from this register when accessed through the register file.

        - *action* = *func() { }*: If a special action is to be taken when a value is read from the register, the code can be placed here. The underlying register is accessible using its name or by using the special name of *ThisReg*. The hook should return a value that is the result of the read operation.
    - Definition: *write*: Specify an action to be taken on a write to this register, when accessed through this register file.

        - *immediate* = <*bool*>: If true, then if this is a parallel architecture, then the write-hook executes immediately, while the actual update of the register is delayed until the commit-phase, at the end of the packet. If false (the default), then the write-hook executes during the commit-phase. If the `ValueBundle` syntax is used, then a write-hook cannot have the immediate flag set to true. If not specified, then the hook inherits its value from the architecture-level flag `immediate_writehooks`.
        - *ignore* = <*bool*>: If true, this designates a read-only register (any write is ignored).
        - *action = func(bits<width> value) { }*: If a special action is to be taken when a value is written to the register, the code can be placed here. The underlying register is accessible using its name or by using the special name of *ThisReg*.
    - *syntax* = <*string*>: Specify a string to be printed in disassembler for instruction field referring to this regfile.
- *attrs* = <*list*>: Same as for *reg*.
- *invalid_entry_read = func(unsigned index) {}*: For sparse register files only. If a read occurs and no entry exists for the specified index, then the default behavior is to abort. This hook allows the user to specify an alternate behavior. It should return a value that is the result of the read operation.
- *invalid_entry_write = func(unsigned index,bits<width> value) {}*: For sparse register files only. If a write occurs and no entry exists for the specified index, then the default behavior is to abort. This hook allows the user to specify an alternate behavior.
- *log_name* = <*ident*|*str*>: Specifies the name to be used when the register-file is logged via the various trace outputs and how the register may be set using the debugging interface. This name overrides the default, which is simply the register-file's name as specified in the definition.
- *prefix* = <*ident*>: A prefix string, to be used for disassembly and assembly.
- *size* = <*int*>: The number of entries in the register file.
- *width* = <*int*>: Same as for *reg*.
- *offset* = <*int*>: Specifies a starting bit offset. For example, this lets the user express the fact that the register is 32-bits wide, but its starting bit should be 32, rather than 0. The bits for fields are relative to this offset.
- *reset* = <*int*> | *func([unsigned context_index]) { }*: Specify a reset value for the register. If not specified, the power-on-reset value is 0 but the value will remain unchanged for reset operations

generated by the core (caused by calling `resetCore` within action code).

This may be a code sequence to handle more complex reset sequences, such as those that depend upon external values. The function should not return a value; rather, it should write to the register to set a value. If the register belongs to a context, then the context ID is supplied as a parameter to the function in order to allow for context-dependent values.

This may not be specified if the register file is sparse. In that case, the registers which constitute the sparse register file will reset themselves.

- *external_write = func(unsigned index,bits<width> value) {}*: Define side-effects to occur when a value is written to the register-file element via the `setRegExt` function. This is useful, for example, for system models, where an interface of some sort may set a register value, with various side effects.
- *alias* = <*ident*>: If a register file is a simple, direct alias of another register file, then the *alias* key may be used. This is equivalent to defining a read and write hook which both have an *alias* key.
- Definition: *read*: Same as for *reg*, except that an extra index as the first argument is required for the action function, if specified. The following items have been added:

  - *regs* = <*list(pair(int))*>: If an alias is defined, this specifies a subset of the registers of the alias to be used for this register file. For example, a value of *(0,15)* would mean that only the first sixteen registers of the alias are to be used. A list is allowed to describe a split register file, e.g. *((0,7),(24,31))* means to map the first and last eight registers of the alias to the register file being defined.
- Definition: *write*: Same as for *reg*, except that an extra index is required for the action function, if specified. The following items have been added:

  - *regs* = <*list(pair(int))*>: If an alias is defined, this specifies a subset of the registers of the alias to be used for this register file. For example, a value of *(0,15)* would mean that only the first sixteen registers of the alias are to be used. A list is allowed to describe a split register file, e.g. *((0,7),(24,31))* means to map the first and last eight registers of the alias to the register file being defined.
- Definition: *field* = <*name*>: Same as for *reg*. These may only be applied to non-sparse register files. All registers within the

file have the same fields. A field is just a subset of bits within a register-file which is assigned a name and may be directly read or written. Refer to Registers for information about how to use fields within action code.
    - *serial* = <*bool*>: In parallel execution mode this regfile is updated sequentially, i.e., immediately. Default is false.
    - *enumerated* = <*list(ident)*>: Same as for *reg*. It may only be applied
      to non-sparse register files.

3.3.3.6 Context

- Definition: *context* = <*name*>: Define a context. This is a group of resources arranged into an array of instances. The active instance is set via a predicate function. This allows for the straight-forward modeling of blocked-multithreading architectures, as well as other constructs which switch between several instances of a set of items.
    - *active* = *func { ... }*: This function specifies which instance is active. It must return an integer representing the active element (0..``num_contexts``).
    - *attrs* = <*list*>: Lists any attributes that this context is associated with.
    - *mem_layout* = <*list*>: List the order in which register and register-file resources should be written or read from memory using the built-in `read` and `write` methods. All elements of a register-file will be contiguous in memory. This list, if specified, must cover all context resources, except those explicitly omitted via the **mem_omit** list.
    - *mem_omit* = <*list*>: Specify resources to omit from the built-in `read` and `write` operations.
    - *num_contexts* = <*int*>: Specify the number of contexts.
    - *regs* = <*list*>: List registers in this context.
    - *regfiles* = <*list*>: List register files in this context.

3.3.3.7 Parameter

- Definition: *parm* = <*name*>: Define an architectural parameter. These may be used to modify the behavior of instructions, registers, etc., by querying their value (and modifying them if they are not constant) in action code. Parameters, by default, are modifiable.

    - *options* = <*list*>: Each list element is an identifier listing a possible value for the parameter. The options "true" and "false" are only allowed if those are the only options, i.e. if it is boolean.

- *value* = <*ident*>: A default value for the parameter. If not listed, or set to *undefined*, then a value must be specified by a subsequent architectural block or else the instantiation of the core will fail.
- *constant* = <*true* | *false*>: If true, the parameter may not be modified by action code. Otherwise, it may be assigned one of the identifiers listed in *options*.
- *watch* = *func() {}*: The user may optionally set a monitor function which may be used to update the value of the parameter. The function is called if any of the registers mentioned within the expression are modified.

Parameters are frequently used to communicate between different subsystems. For example, an MMU translation may set a parameter which is then checked by a memory hook. When using an internal direct-memory-interface cache, the simulator will look at the memory hooks to see what parameters must be saved within a cache entry. Any extra parameters that might be needed may be given an attribute of `dmi_store`, which will cause them to also be saved, and set when a hit occurs in the DMI cache.

3.3.3.8 Exception

- Definition: *exception* = <*name*>: Define an exception. Exception names must be valid C++ identifiers and may be used within the model as an `enum` of type `Exception`. An exception may be raised by action code using several methods:

  - By calling the *raiseException* function with the name of an exception. This will cause an exception to be immediately processed. If the exception contains any fields, then the data for this exception will be supplied with their reset values.
  - By calling *setException*. This will cause the exception to be handled after the instruction has completed. If the exception contains any fields, then the data for this exception will be supplied with their reset values.
  - By calling *raiseException(const <name>_t &)*. This will cause the exception to be immediately processed. Field data for the exception will be take from the supplied argument.
  - By calling *setException(const <name>_t &)*. This will cause the exception to be handled after the instruction has completed. Field data for the exception will be take from the supplied argument.

For level sensitive exceptions, a pending exception may be canceled by calling the cancelException* function.

Generally, an exception's processing consists of modifying architectural state to store two types of information: Information describing why an exception was taken, e.g. that a Power program-interrupt was taken due to an illegal instruction, and information describing how the exception should be handled, e.g. the saving of state information and a change in control.

For the first type of information, it may seem natural to simply modify architectural state before calling *raiseException*. For example, a Power *mtspr* instruction might modify the ESR (Exception Syndrome Register) to set the appropriate bit indicating that an attempt was made to access an illegal SPR register, then call *raiseException*. This approach has several problems. First, it divides an exception's modifications into multiple locations, which makes it hard to understand the overall effect. Second, for a performance model which is modeling a pipeline, it may be necessary to ignore exceptions, such as those caused by speculative instructions. If architectural changes are made immediately, then these effects cannot easily be ignored.

For this reason, it is highly recommended that exception fields be used to convey information to the exception handler. If fields are defined for an exception, then a data structure named *<name>_t* will be created, containing members named for each field. The user should store all relevant information into this data structure, then raise an exception using the appropriate, exception-specific raise function. The exception handler will then receive a copy of this data structure, from which architectural modifications may be made.

The keys and defines for the *exception* definition are:

- Definition: *field* = <*name*>: Define a field for storing exception information.
  - *bits* = <*int*>: Specifies that the field is an integer of *bits* size. If set to 0 (the default), then the field's type is a 32-bit integer.
  - *attrs* = <*list*>: Lists any attributes that this field is associated with.
  - *reset* = <*int* | *ident*>: Specify the default value for this field.
- *attrs* = <*list*>: Lists any attributes that this exception is associated with.
- *action = func([<name>_t]) { }*: This code is executed when the exception is raised. It takes care of modifying the environment as needed, such as by modifying the program counter to point to the new instruction location. If fields are defined for this exception, then the exception's action code will take a single argument: A

copy of the data structure set by the function which raised the exception.
  - *enable = func() {}*: A boolean predicate which specifies whether the exception is or is not enabled. If it returns false, then the side-effects of the exception, as specified by `action` are not taken. If the exception is `level` sensitive and is raised from an external source, then if the exception is currently disabled, it will remain pending until the exception is either enabled, at which point the exception side-effects will occur, or it is canceled.
  - *priority* = <*int*>: Specifies the priority class for the exception. The highest priority value is 0. Higher priority exceptions are processed after lower priority exceptions so that their side effects will be seen last.
  - *sensitivity = level* | *edge*: Describes the behavior of the exception when generated externally. The default is `edge` sensitive, which means that if the exception is currently disabled, all effects will be ignored. If the exception is `level` sensitive, then if it is generated externally and it is currently disabled, it will remain pending until the exception is either enabled, at which point the exception side-effects will occur, or it is canceled.

3.3.3.9 MMU

- Definition: *mmu*: The general idea is that instruction descriptions will communicate with the memory via the use of the Mem object, using effective addresses. Based upon whether an MMU is defined or not, translation of this address into a physical address may take place.

  An ADL-generated simulator may implement an MMU such that translations are cached in order to reduce the overhead of looking up a real address for each instruction fetch, data access, etc. In such a situation, callback functions such as `hit`, `miss`, etc. may not be called for each and every translation. However, they will be called for each new translation, so that the system may check or modify its state as necessary.

  In order to detect when such a cache becomes invalid, ADL examines the registers used in these various predicates and callbacks and flushes the cache when these registers change. However, sometimes this can be overly conservative. There may be a situation where a register change should not cause the cache to be invalidated because the register is not actually a control register, but rather simply a conveyor of information. In that case, a register may be given an attribute of `no_mmu_control`. This will cause the register to be excluded from the set of registers which, upon a change, will cause a translation cache to be flushed.

- Definition: *lookup* = <*name*>: The MMU contains one or more *lookup* objects which represent TLBs and map an effective address to a real address. If multiple *lookup* objects are present then they must either be assigned priorities in order to handle the situation where more than one object returns a mapping, or else an event handler must be defined. This would normally generate an exception. The *lookup* objects may define an array for storing information or else they may define an action function for performing the mapping operation.

  Note that in the functions which require as arguments of <*lookup name*> or <*parent-lookup name*>, the parent names are the names of the parent lookup objects. The lookup name argument may be either the name of the current lookup object or the special type *TransType*. Use of the latter may be preferred so that a generic MMU lookup object may be used for multiple cases through the use of the *inherit* and *interface* keys.

  - Definition: *array*: Define an array for this lookup object. This defines an associative data structure which stores field information.

    - *entries*: Total number of entries in the array.
    - *set_assoc*: Set-associativity of the array. The number of sets in the array is thus `entries / set_assoc`. A value of 0 is interpreted to be fully-associative.
  - Definition: *lookup* = <*name*>: *lookup* blocks may be nested. This means that if a hit is found, the child *lookup* blocks will be searched. Expressions within the child blocks will have access to the parent's matching entry by using the name of the parent block. So, for example, a child whose parent is named *Seg* may access the parent's matching entry by specifying *Seg.X*, where X is the name of the field.
  - Definition: *setfield* = <*name*>: Define a field for each set in an MMU lookup.

    - *bits* = <*int*>: Specify the number of bits in this field. If set to 0 (the default), then the field's type is a 32-bit integer.
    - *attrs* = <*list*>: Lists any attributes that this field is associated with.
    - *reset* = <*int*>: Specify the reset value for this field item.

- Definition: *wayfield* = <*name*>: Define a field for each way in an MMU lookup.

    - *bits* = <*int*>: Specify the number of bits in this field. If set to 0 (the default), then the field's type is a 32-bit integer.
    - *attrs* = <*list*>: Lists any attributes that this field is associated with.
    - *reset* = <*int*>: Specify the reset value for this field item.
- Definition: *test* = <*name*>: Address-selection criteria can be grouped together into *test-sets*, in order to allow an MMU to apply different matching criteria, based upon the state of the system. The order of application of these different test-sets is specified via the `test_order` key in the lookup.

    - *enable = func(TransType) {}*: This predicate is used to tell whether the test-set should be used for matching purposes. It should return a boolean value of `true` to indicate that it is enabled. The `TransType` parameter is used to specify the type of translation: `LoadTrans`, `StoreTrans`, or `InstrTrans`.
    - *test* = <*test function*>: Specify a matching function. These entries may be repeated and all match functions must be satisfied to register as a successful lookup. The valid matching functions are:

        - *Bounded(<expr>,<expr>[,<expr>])*: The two expressions denote a lower and upper bound (inclusive), against which to check the address. Note that this is similar to using a *Check* test, except that this tells the system how to describe the boundary and thus makes it possible to cache the translation for faster model performance.

          The optional third expression allows the user to specify a mask to be used to mask the address before it is compared against the bounds. The result of the expression should be a mask value to be applied against the effective address. Keep in mind that the address is always a 64-bit `unsigned long long` and thus an offset may need to be applied when dealing with 32-bit address.
        - *Check(<expr>)*: Execute the expression and return true if the expression returns true. For example, *Check((MSR(PR) == 0) ? VS : VP)*, where *MSR* is

    a register, *PR* is a register field, and *VS* and *VP* are fields of the lookup object.
  - *Compare(<field-name>,<expr> [,<expr>...])*: The first argument specifies a field name. Subsequent arguments define match expressions. The field must match the value of one of the specified expressions. For example, *Compare(TID,PID0,PID1,PID2)*, where *PIDn* are registers and *TID* is a lookup field. If this lookup is a child of another lookup, parent fields may be accessed by specifying the name of the parent, e.g. *Seg.VSID*.
  - *AddrIndex(<start-bit>,<stop-bit>)*: Specify address bits which select an entry in the array. This may only be applied to arrays with a set-associativity of 1 (direct-mapped). The start and stop bits are calculated relative to the ra_mask value. Currently, the ea_mask must be a constant value in order to support this test.
  - *AddrComp(<field-name>)*: Match the address to the specified field name. The offset is calculated using the *pagesize*.

  The expressions may use the *Instr*, *Data*, *Load*, or *Store* predicates to test for the type of translation being performed and may use *ea* to check the effective-address.
- *attrs* = <*list*>: Lists any attributes that this lookup is associated with.
- *type* = <*Instr* | *Data* | *Both*>: The type of translation for which to use this lookup. The default is *Both*.
- *priority* = <*int* | *'ignore'*>: Priority of this lookup function versus others at its same level. If priorities are not used, then a multi-hit handler must be present in order to handle the case where multiple hits occur. The default priority is 0 (highest). If the value is set to **ignore** then the lookup is not used for the general translation process. It is still generated, however, so that it may be used explicitly by action code, by directly referring to it.
- *inherit* = <*ident*>: Specify a lookup object from which to inherit. This essentially makes a copy of the specified lookup object.
- *interface* = <*bool*>: Specify that this item is only an interface and should not be used for translations. Interface objects are useful for specifying requirements; another lookup object can use an interface object by specifying the *inherit* key.

- *pageshift* = <*int*>: Specify a value to shift the effective and real page numbers by before combining with the page offset. Use this for when there is an implicit shift for these fields.
- *sizetype* = <*ident*>: Specify how size is encoded. The type may be *BitSize* (default), which means that the page size should be interpreted as (2^(sizescale*size+sizeoffset) << sizeshift) bytes, or *LeftMask*, which means that the field name is interpreted as a mask with 1's specifying the bits of the address which are ignored, shifted by sizeshift.
- *size* = <*int* | *field-name*>: Specify a size for this translation. If an integer, the size is taken to be that value in bytes.
- *sizescale* = <*int*>: Specify a scaling factor for the size field. For example, Power uses 4 as the base for the size calculation. Thus, a scaling factor of 2 would be used.
- *sizeoffset* = <*int*>: Offset value added to the size before it is scaled. Use this for when the size value has an implicit offset, e.g. "a size 0 page is 16KB".
- *sizeshift* = <*int*>: Amount by which to shift the size value.
- *test* = <*test function*>: Tests may be specified directly within a lookup if there is only a single set of selection criteria. This means that only a single test-set exists, which is always used.
- *test_order* = *list(ident)*: Specifies the order for applying test-sets when performing an address lookup. Each element of the list should identify a `test` define. All test-sets other than the last item must have an enable predicate; the last item's enable predicate will be ignored, as it represents the *else* clause.
- *valid* = *func() { }*: A predicate function which returns true if the translation is considered valid. The function may only refer to fields within the translation and not to any design resources such as registers.
- *realpage* = <*field-name*>: Specify the field used to construct the real address. The offset is derived from the *size* field. This is then concatenated to the *realpage* value to form the final address. If not specified, then no translation is performed- the real address is simply the effective address.
- *exec_perm* = *func(<lookup name> [,<parent-lookup name> ...],addr_t ea,unsigned seq) {}*: Function called on an instruction hit. This should check to see if the access is allowed and take any appropriate action if it is not. The `seq` parameter specifies which translation this represents, if a sequence of translations is required, such as for a misaligned access.
- *data_perm* = *func(<lookup name> [,<parent-lookup name> ...],addr_t ea,unsigned seq) {}*: Function called on

a load or store. This should check to see if the access is allowed and take any appropriate action if it is not. The `seq` parameter specifies which translation this represents, if a sequence of translations is required, such as for a misaligned access.

- *load_perm = func(<lookup name> [,<parent-lookup name> ...],addr_t ea,unsigned seq) {}*: Function called on a load. This should check to see if the access is allowed and take any appropriate action if it is not. The `seq` parameter specifies which translation this represents, if a sequence of translations is required, such as for a misaligned access.
- *store_perm = func(<lookup name> [,<parent-lookup name> ...],addr_t ea,unsigned seq) {}*: Function called on a store. This should check to see if the access is allowed and take any appropriate action if it is not. The `seq` parameter specifies which translation this represents, if a sequence of translations is required, such as for a misaligned access.
- *multi_hit = func(TransType tt,addr_t,unsigned seq) { }*: If this hook is defined, then a search will continue after an initial translation is found. If another translation in the same way is also found, then this hook will be invoked.
- *final_hit = func(TransType tt,addr_t,unsigned seq,unsigned found_count) { }*: If this hook is defined then a search will continue after an initial translation is found. this function is called after a search is performed and one or more matches are found. The `found_count` parameter contains the number of matching entries.
- *miss = func (TransType tt,addr_t ea,unsigned seq) { }*: Miss-handler. If the item is not found in the specified array, this function is executed. If no array is defined, then this function is always executed. It may be used, for example, to implement a hardware table-walk routine. The `seq` parameter specifies which translation this represents, if a sequence of translations is required, such as for a misaligned access.

  It must return an object whose type is the name of the lookup object. The object is basically a structure whose constructor arguments are the list of fields and whose members are the fields found in this lookup object.
- *hit = func(TransType tt,<lookup name> [,<parent-lookup name> ...],addr_t ea,unsigned seq) {}*: Hit-handler. Called if there is a hit in the array. There is no return value; the purpose of this function is to cause any side-effects that are required. The `seq` parameter specifies which translation this

represents, if a sequence of translations is required, such as for a misaligned access.

- *reset = func { }*: Reset handler function. This function will execute during reset and may be used to initialize the lookup to a reset state. The lookup can be accessed using the function call API. For example, a lookup named `TlbCam` for a Power part might be reset in the following manner:

```
reset = {
  // Reset state: One 4k translation at top of memory which does a 1-1
  // translation.
  TlbCam_t t;

  t.EPN  = 0xfffff;
  t.TID  = 0;
  t.V    = 1;
  t.TS   = 0;

  t.RPN  = 0xfffff;
  t.SIZE = 1;

  t.SX   = 1;
  t.SW   = 1;
  t.SR   = 1;

  t.UX   = 0;
  t.UW   = 0;
  t.UR   = 0;
  t.WIMG = 0;

  TlbCam(0,0) = t;
}
```

  The reset function is called after all fields have been set to their default values.
- *mem_read = func(unsigned set,unsigned way,addr_t addr) {}*: Read the contents of a specific set and way of the lookup from memory, starting at address `addr`. This routine is required if the lookup is part of a context. The function should return an updated address pointing to the next location after the entry that was read.
- *mem_write = func(unsigned set,unsigned way,addr_t addr) {}*: Write the contents of a specific set and way of the lookup to memory, starting at address `addr`. This routine is required if the lookup is part of a context. The function should return an updated address pointing to the next location after the entry that was written.
- *mem_size* = <*int*>: Size, in bytes, of a lookup element when written to memory. This must be defined if the `mem_read` and `mem_write` hooks are defined.

o *attrs* = <*list*>: Lists any attributes that this MMU is associated with.

- *both_enable = func() { }*: If present, this expression must evaluate to true in order for the MMU to be enabled.
- *instr_enable = func() { }*: If present, this expression must evaluate to true in order for the MMU to be enabled for instruction translations.
- *data_enable = func() { }*: If present, this expression must evaluate to true in order for the MMU to be enabled for data translations.
- *multi_hit = func(TransType tt,addr_t ea,unsigned seq) { }*: Code to execute when more than one lookup object finds a mapping and there is no priority order.
- *final_hit = func(TransType tt,addr_t,unsigned seq,unsigned found_count) { }*: If this hook is defined, then this function is called after a search is performed and one or more matches are found. The `found_count` parameter contains the number of matching lookups. Note that if this hit exists, then we do obey priorities in terms of search order and we also call the **multi_hit** hook.
- *data_miss = func(addr_t ea,unsigned seq) { }*: Code to generate when no mapping is found on a data translation. The function may raise an exception or return a boolean. If true is returned, the translation process is re-tried. If false, then the effective address is used as the translated address.
- *store_miss = func(addr_t ea,unsigned seq) { }*: Same as data-miss, except only called on stores (writes). If both a **data_miss** and a **store_miss** hook exist, then **store_miss** is called first.
- *load_miss = func(addr_t ea,unsigned seq) { }*: Same as data-miss, except only called on loads (reads). If both a **data_miss** and a **load_miss** hook exist, then **load_miss** is called first.
- *instr_miss = func(addr_t ea,unsigned seq) { }*: Code to generate when no mapping is found on an instruction translation. The function may raise an exception or return a boolean. If true is returned, the translation process is re-tried. If false, then the effective address is used as the translated address.
- *misaligned_read = func(addr_t ea,addr_t ra)*: If a misaligned read occurs, then this hook is called between memory accesses which might cross a page.
- *aligned_write = func(addr_t ea,addr_t ra)*: If a write occurs, then this hook is called with the initial effective and real address. If this is a misaligned write, the address is the initial, misaligned address of the access.
- *misaligned_write = func(addr_t ea,addr_t ra)*: If a misaligned write occurs, then this hook is called between memory accesses which might cross a page. It can be used to implement non-atomic writes. For example, suppose a misaligned store operation crosses a page, such that the second page is not accessible. If no portion of the operation should be performed, then the model

can simply raise an exception in one of the permission checking functions. It may also be used for broadcasting information about writes, such as for implementing coherency protocols.

If, however, the first portion of the store should be performed, but no the second part, then the permission function might simply set a flag, such as by using a non-architected register. Then, the `page_crossing_write` hook would actually raise the exception.

- *pre_read = func(addr_t ea,addr_t ra)*: This hook allows actions to be performed just prior to a read being performed, after all translations have been performed. This is only called once, even for misaligned accesses. The addresses are the initial addresses of the access. This hook may only be used if reads are non-interleaved (see interleaved_reads).
- *post_read = func(addr_t ea,addr_t ra)*: This hook allows actions to be performed once a read has been performed, but before the value is returned to an instruction's action code. This is only called once, even for misaligned accesses. The addresses are the initial addresses of the access.
- *pre_write = func(addr_t ea,addr_t ra)*: This hook allows actions to be performed just prior to the execution of a write, after all translations have been performed. This is only called once, even for misaligned accesses. The addresses are the initial addresses of the access. This hook may only be used if writes are non-interleaved (see interleaved_writes).
- *post_write = func(addr_t ea,addr_t ra)*: This hook allows actions to be performed once a write has been performed, but before control returns to an instruction's action code. This is only called once, even for misaligned accesses. The addresses are the initial addresses of the access.

3.3.3.10 Cache

- Definition: *cache = <name>*: Specify a cache.
  - Definition: *setfield = <name>*: Specify a field for each set of the cache. The value of the field is then accessible via a unary function-call, e.g. `ThisCache(set).field`.
    - *bits = <int>*: Specify the width of the field in bits. If set to 0 (the default), then the field's type is a 32-bit integer.
    - *attrs = <list>*: Lists any attributes that this field is associated with.
    - *reset = <int>*: Specify the reset value for this field. Default is 0.

- Definition: *wayfield* = <*name*>: Specify a field for each way of each set of the cache. The value of the field is then accessible via a binary function-call, e.g. `ThisCache(set,way).field`.

  - *bits* = <*int*>: Specify the width of the field in bits. If set to 0 (the default), then the field's type is a 32-bit integer.
  - *attrs* = <*list*>: Lists any attributes that this field is associated with.
  - *reset* = <*int*>: Specify the reset value for this field. Default is 0.

  Note: All caches contain the way-fields of `valid`, `dirty`, `tag`, and `locked`. The names of way-fields and set-fields must all be unique.
- *type* = <*instr*|*data*|*unified*>: Specify the type of the cache. A harvard architecture is described by creating separate data and instruction caches. Once a level in the memory hierarchy has become unified, all subsequent caches must be unified.
- *level* = <*int*>: Level of cache hierarchy. A value of 1 is the smallest allowed value and corresponds to an L1 cache.
- *attrs* = <*list*>: Lists any attributes that this cache is associated with.
- *sets* = <*int* | *list(int)*>: Explicitly specify the number of sets in the cache. This may be a list in order to allow the user to model a cache with a dynamic structure, in which case a `sets_func` must be specified in order to select between different numbers of sets. If not specified, then the number of sets will be calculated from the size of the cache.

  If a model contains both model-based cache configuration code, i.e. a `sets_func` and supports dynamic parameters, then the dynamic parameters will take precedence: If a dynamic parameter is set on a cache, then the cache will ignore any changes generated by the model.
- *size* = <*int*>: Size of the cache, in bytes. May be omitted if the number of sets is explicitly specified. If both the size and the number of sets are specified, then the size must be smaller than the largest size possible from the number of sets and ways specified. However, a smaller size is permitted for when a design does not allow for this maximum size.

  For example, a cache for a design might have two configurations: 8-way 128 sets or 4-way 256 sets, each with a 128-byte cache line size. By default, the size would be based upon the largest possible value, e.g. 8 ways and 256 sets, but in this case, the actual size is 128 * 256 * 4 = 128k.
- *linesize* = <*int*>: The size of a cache line, in bytes.

- *set_assoc = <int | list(int)>*: Set associativity of the cache. A value of 0 implies a fully-associative cache.. This may be a list in order to allow the user to model a cache with a dynamic structure, in which case a `assoc_func` must be specified in order to select between different associativities.

  If a model contains both model-based cache configuration code, i.e. a `sets_func` and supports dynamic parameters, then the dynamic parameters will take precedence: If a dynamic parameter is set on a cache, then the cache will ignore any changes generated by the model.
- *sets_func = func () { }*: A predicate to specify the number of sets currently active in the cache. The returned value must be an unsigned integer whose value corresponds to one of the values listed in the `sets` key.
- *assoc_func = func () { }*: A predicate to specify the number of ways currently active in the cache. The returned value must be an unsigned integer whose value corresponds to one of the values listed in the `set_assoc` key.
- *enable = func (CacheAccess ca [,addr_t addr]) { }*: If present, this expression must evaluate to true in order for the cache to be enabled. Otherwise, it is disabled and the access passes directly to the next level in the memory hierarchy. This function may take a single argument which indicates the type of access and an optional second argument which specifies the address of the access.
- *hit = func (CacheAccess,addr_t) { }*: Function called when an address hits in the cache.
- *hit_pred = func(CacheAccess,unsigned set,unsigned way) {}*: Function called when a set is being searched for a valid way. This predicate, if it exists, is only called if the given way is valid and has a matching tag. It may be used to skip a way due to other reasons, such as to model a way-disabling feature. It should return true if the way is valid, or false to skip this way and proceed with the search.
- *miss = func (CacheAccess,addr_t) { }*: Function called when an address misses in the cache.
- *line_access = func(CacheAccess ca,unsigned set,unsigned way,addr_t addr) { }*: Function called when a line is hit or loaded into the cache.
- *replace = func(CacheAccess ca,unsigned set) { }*: Defines a custom replacement algorithm. The function should return the way to be replaced.
- *invalidate_line = func(CacheAccess ca,unsigned set,unsigned way) {}*: Called when a line is invalidated. Refer to the Cache API

  for methods used to manipulate the cache. Default behavior is to do nothing. This routine can be useful for modifying state, such as locks, when a line is invalidated.
- *read_line = func(CacheAccess ca, unsigned set,unsigned way,addr_t addr) { }*: Function called when a new line (specified by *addr*) is needed. This function can be used to implement, for example, victim cache behavior by reading from memory rather than the next level of the cache hierarchy. Refer to the Cache API for methods used to manipulate the cache. The default behavior is to query the next level of the memory hierarchy.
- *miss_enable = func(CacheAccess ca,addr_t addr,unsigned set) {}*: Should return true if, on a miss, a line may be allocated (or replaced). If false, then the access proceeds to the next level in the memory hierarchy.
- *write_through = func(CacheAccess ca, addr_t addr) {}*: Function called on a write. Return true to indicate that the behavior should be write-through and false to indicate write-back behavior. Default behavior is to always return false.

3.3.3.11 Memory

- Definition: *mem = <name>*: Specify a local memory.
  - *instr_mem = <true | false>*: If true, will be the source for instructions. Only one is allowed per-core. Optional, defaults to false.
  - *size = <int>*: Size of the memory, in bytes.
  - *addr_unit = <int>*: Addressable unit, in bytes. Must be a power of 2.
  - *parent = (<mem name>, <offset>)*: Optional, allows a memory to be implemented as a subset of another memory.
  - *read = func(unsigned size,bits<s> data)*: This allows the user to define a read hook to manipulate data read from memory. The value in `data` is what has just been read and is the size of the access just performed. In other words, if a 32-bit read is executed, then `data` will be 32-bits wide. The function should perform any necessary manipulations and then return the new value.
  - *instr_read = func(unsigned size,bits<s> data)*: Same as the `read` hook, except that this applies only to instructions. If not specified, then the `read` hook is used for instructions and data. If it is specified, then this hook is used for instruction fetches and the `read` hook is only used for data.
  - *write = func(unsigned size,bits<s> data)*: This allows the user to define a write hook to manipulate data written to memory. The value in `data` is what will be written to memory and is the size of the access being performed. In other words, if a 32-bit write

is executed, then `data` will be 32-bits wide. The function should perform any necessary manipulations and then return the new value. This hook is called just after the address is translated, so that any processor state modified by a MMU hit function will be visible to this hook.

- *prefetch_mem* = <*true* | *false*>: If true, this memory serves as a prefetch buffer for instructions. Optional, defaults to false. Buffer is updated from the instruction memory at the pre-fetch phase, before the pre_fetch hook is invoked. Buffer is filled starting at the current-instruction-address. Only one is allowd per-core. This memory can not have read/write hooks, parent memory or addr_unit defined. Accesses to this memory are not logged and modifications are not written back.

3.3.3.12 Event Bus

- Definition: *eventbus*: Create a broadcast bus for interprocessor communications. A bus allows one core to transmit a packet to all other cores which have a bus of the same name. It is an error to have buses of the same name but with different data-packet formats.

  To send data on a bus, the syntax is:

  ```
  <bus name>.send(<bus data>,bool signal_self = false);
  ```

  Where `bus data` is an object of type <`bus name`>`_t`. The default constructor sets all fields to either the specified reset value, or else 0 (or the first enumerated value if the field is enumerated). Each field may then be assigned to directly. For example, given a bus named `Foo` with two fields, `field1` and `field2`:

  ```
  Foo_t x;

  x.field1 = 10;
  x.field2 = 20;

  Foo.send(x);
  ```

  The `send` function may be called anywhere within a core. If the `signal_self` parameter is set to true, then all registered handlers for the event bus, including that of the sender, are invoked. Otherwise, the sender's handler is not invoked.

  - *action* = *func(<bus name>_t x) { ... }*: Handler function called when data is transmitted on the bus. All attached handlers, except the handler of the transmitting core, are called by default. As with all other hook functions, any core resources may be modified.
  - Definition: *field* = <*name*>: Define a field for the bus's data packet.

- *bits* = <*int*>: Specifies that the field is an integer of *bits* size. If set to 0 (the default), then the field's type is a 32-bit integer.
- *attrs* = <*list*>: Lists any attributes that this field is associated with.
- *reset* = <*int* | *ident*>: Specify the default value for this field.

3.3.3.13 Group

- Definition: *group*: Create a group.

  - *attrs* = <*ident* | *list(ident)*>: Lists any attributes that this group is associated with. This is either a single identifier or a list of identifiers.
  - *check_parent* = <*bool*>: If true, then when applying a filter, such as `has_attrs`, this will also check a parent instruction. Otherwise, the parent instruction is skipped and not included in a group.
  - *type* = <*ident*>: Optionally defines type of the group. If missing it will be deduced from group items.
  - *has_attrs* = <*ident*|*ident(str*|*int[,...])*> | <*list(ident*|*ident(str*|*int[,...]))*>: Specify that group items must contain one or more of the listed attributes. If this filter is used then the `type` must be explicitly specified. For example:

```
define (instr = i1) { attrs = (a1); fields = (OPCD(1),A,B); action  = {};}
define (instr = i2) { attrs = (a2,a4(1)); fields = (OPCD(1),A,B); action  = {};}
define (instr = i3) { attrs = (a3,a1,a4(2)); fields = (OPCD(1),A,B); action  = {};}

define (group=g1) {
  has_attrs = a1;
  type = instr;
}

define (group = g2) {
  has_attrs = a4(1);
  type = instr;
}
```

    The group **g1** would select instructions **i1** and **i3** and **g2** will have instruction i2.
  - *has_fields* = <*ident* | *list(ident)*>: Specify that group items must contain the listed fields. This is only applicable for instructions, registers, and register-files. If this filter is used then the `type` must be explicitly specified. For example:

```
define (instr = i1) { fields = (OPCD(1),A,B); action  = {};}
define (instr = i2) { fields = (OPCD(1),A,C); action  = {};}
define (instr = i3) { fields = (OPCD(1),A,D); action  = {};}

define (group=g1) {
  has_fields = (A,C);
  type = instr;
}
```

The group **g1** would select instruction **i2**.

- *items* = <*list*> : Specifies items of the group. Item can be:

1. An ADL object.
2. A reference to all elements of a given type (denoted '*'). For example:

   ```
   define (group = all_insts) {
     type = instr;
     items = "*";
   }
   ```

   This creates a group of all instructions in the system.
3. Set intersection (denoted '&'), set difference (denoted '-'), or set union (denoted by '|' or ',') of two groups. All items should be of the same type except for instructions and subinstructions that are considered to be of the same type (subinstruction refers to all instructions built from it). Sub-expressions are supported. For example:

   ```
   define (group = big_group) {
     items = (r15,                 // simple item
              *reg_g1,             // reference to some group
              *reg_g2 - *reg_g4,   // set difference
              *reg_g2 & *reg_g4 ); // set intersection
   }

   define (group = reg_g5) {
     items = ( (*reg_g1 | *reg_g2) - (reg1,reg2,reg3));
   }

   defmod (reg = (r23,*reg_big)) {
     attrs = small;
   }
   ```

One more example: Suppose, we want to indicate in ADL the time that is required for instruction to be completed. Here subinstruction *multOp* refers to all instructions built from it, e.g. all instructions that implement multiplication. Now we are able to define:

```
define (group = fast) {
  items = (add,sub,subi,subi,div);
}

define (group = slow) {
  items = (multOp,div,divi);
}

defmod (instr = *fast) {
  attrs = time(1);
}

defmod (instr = *slow) {
  attrs = time(5);
}
```

```
defmod (instr = (div,divi)) {
  attrs = time(10);
}

define (group = debug_illegal) {
  items = (*fast & *slow);
}
```

Group references are resolved when defmod blocks are parsed, so defmod groups can be specified in any order, although cycle references are forbidden.

3.3.3.14 Assembler

- Definition: *assembler*: Configures assembler generator.
  - *comments = <list(string)>* : Optional, specifies strings that start a comment. Default is "#".
  - *line_comments = <list(string)>* : Optional, specifies strings that start a comment only at the beginning of the line. Default is "#". Note, that usual comments cannot be used at the beginning of the line.
  - *line_separators = <list(char)>* : Optional, defines additional characters used to denote an "end of line". For parallel architectures, this means the end of a packet.
  - *instr_separators = <list(char>**: Optional, defines characters used to mark an explicit separation between instructions. This is only valid for architectures with packetized encodings and instructions separated by these characters are considered to be part of the same packet.
  - *queue_size = <int>* : Optional, defines the size of the packet queue the assembler maintains. The default is 1. The queue allows assembler function code to address packets preceding the current one.
  - *queue_offset = <int>*: Optional, defines which element of the queue should be processed by the *post_packet_asm* hook. By default, this is 0, meaning that the current packet is always processed. By setting this to 1, for example, the second-most-current packet is processed, allowing the *post_packet_asm* hook to potentially combine two packets together. Other elements in the queue may be accessed via the **getPriorPacket** function.
  - *packet_grouping = <char|(<char,char)> >* : Optional, defines packet grouping separators. Can be defined only for parallel architectures. Default is "\n". Note, that if "\n" is used as packet separator then every character defined in line_separator key will be considered a packet separator.

- *disasm_instrtables* = <*list*>: List classes of instructions to be grouped into one decoding tree of the dissassembler. The list is ordered, so first the disassembler searches instructions in the first instruction table, if not found it tries to look in the next one etc. This is useful for the prefix encodings, where prefix instructions reside in the "prefix" instruction table but no explicit table change is written in assembly.
- *asm_instrtables* = <*list*>: List classes of instructions to be grouped together into a single table for the assembler. This allows multiple sets of instructions to be assembled directly, without an explicit change of encoding tables, as long as their names do not conflict.
- *exlicit_regs* = <*bool*>: Optional, specifies if one has to write an explicit register file prefix in assembly programs. Default is false (PowerPC). This feature is useful when instruction can be recognized only by its operands, example: StarCore instructions 'move.l #$1,r0' and 'move.l #$1,d0".
- *symbol_chars* = <*list(characters)*> : Optional, specifies the list of characters that may appear in operand in addition to the alphanumeric characters. This affects the way how the assembler remove whitespaces before such characters.
- *operand_symbols_excluded* = <*list(characters)*> : Optional, specifies the list of characters that may not appear in operand. This may help reject invalid syntax earlier, at parsing and not at encoding, as required by some assembler application.
- Definition: *rule* = <*name*>: Define an assembler programming rule. It is invoked by the assembler after the *post_asm* hook for instruction rules, and after the *post_packet_asm* hook for packets.

  - *action = func(<InstrBundle|InstrInfo>) {}*: Specifies an assembler rule. The function takes an *InstrInfo* argument for instruction rules, and an *InstrBundle* argument for packet rules. See the *Assembler Programming Rules* section for more details.
- Definition: *parm* = <*name*>: Define an assembler parameter. Used to modify assembler behaviour by setting and querying their value by *assembler*.

  - *constant* = <*bool*>: If true, parameter can't be modified by assembler action code. Defualt, modifiable parameters.
  - *type* = <*ident*> : Specifies assembler parameter type. Valid options are:
    - *integer*: specifies an integer type parameters implemented as a 64-bit unsigned integer.

  - *label*: specifies a label parameter that can be be assigned the current instruction/packet label, denoted by "'*'", or checked for equality with the current label.
- *value* = <*int*>: specifies parameter default value for *integer* type parameters, not be specified for *label* paramaters their default, by definition, is *0*.

Example

```
define (assembler) {
      comments = ("//","#");
      line_comments = ("#rem");
      line_separators = (";");
      packet_grouping = (("[","]","\n"));
      define(parm=L1) {
         type = label;
      }
      define(parm=P1) {
         type = integer;
         value = 0;
      }
}
```

3.3.3.15 Real-Address Mask

- Definition: *ra_mask*: Specify a real-address mask. This will be applied to all addresses after translation, but before the request to memory. Since the native memory model for ADL is 64-bit, this allows the user to model a processor which has fewer address bits.
  - *value* = <*addr_t*>: Specify an initial value for the address mask.
  - *watch* = *func () { }*: Specify a watch function to dynamically modify the mask based upon resource changes. To modify the mask within the function, write to the variable "RaMask".

3.3.3.16 Effective-Address Mask

- Definition: *ea_mask*: Specify an effective-address mask. This will be applied to all addresses immediately before translation.
  - *value* = <*addr_t*>: Specify an initial value for the address mask.
  - *watch* = *func () { }*: Specify a watch function to dynamically modify the mask based upon resource changes. To modify the mask within the function, write to the variable "EaMask".

3.3.3.17 External Resource

- Definition: *ext_resource* = <*name*>: Defines a legal, global resource, defined externally, for the core. The object may then be used in action code in the design; no checking is performed other than to allow the name to be used. This is useful, for example, if an external class is to be used to implement additional functionality in the design, similar to how external functions maybe called by action code.

Note that the resource's declaration will need to be included via a header file.

In the case of a resource declared as **external**, a class method is created, named `void set_<ext-resource>(objtype *)`, so that a pointer to the object may be specified and a method called `objtype *get_<ext-resource>()` is created to access the pointer. The standard interface function `IssNode::setExtResource` may also be used to set a pointer to the resource object by name.

For a resource not declared as **external**, an accessor called `objtype &get_<ext-resource>()` is created.

- *objtype* = <*ident*>: The object's type. For model generation, this will be instantiated as a member variable in the core's class.
- *constr_args* = <*list*>: A list of constructor arguments for a non-external resource. The arguments are supplied as-is to the object's constructor.
- *external* = <*bool*>: If true, only a pointer to the object is created and initialized to 0. An external application must assign the pointer before use. The default is false: The object is instantiated as a normal member variable.
- *reset* = <*bool*>: If true, a call to the object's `reset(bool init,bool por)` method is made when the core is reset. The **init** parameter will be true upon initial object creation and **por** will be true upon subsequent reset operations.

3.3.3.18 Prefix Counter

- Definition: *prefixcounter* = <*name*>: Define a counter for indexing into instruction fields or for other uses within a parallel architecture.

  The basic idea is that, for some architectures such as StarCore, optional prefix instructions are used to increase the size of an instruction's encoding space. When present, a field within the prefix instruction supplies a fixed number of bits to certain instructions within the VLIW packet, e.g. three bits per arithmetic instruction, where those three bits are used to contribute one extra bit to each of the operand fields.

  However, the method by which each slice is retrieved from the field may be complex, e.g. only arithmetic instructions use this field, etc. For this reason, the user may employ a counter to index into such a field. A counter is bound to a specific field within an instruction definition, using the `prefix_counters` parameter, and incremented or decremented by the relevant instructions using the `prefix_counter_incr` or `prefix_counter_decr` parameter.

- *reset* = <*int*>: Specify the reset value for the counter. The reset value is applied whenever a new packet is encountered.

3.3.3.19 Relocation

- Definition: *reloc* = <*name*>: Define a linker relocation type. A relocation is the method by which an assembler communicates with a linker, when symbol addresses cannot be determined at assembly time. For example, a function's address might be in a different translation unit. The assembler will see that a symbol is not local and will insert a relocation so that the linker can insert the symbol's address later, once it is known.
    - *abbrev* = <*str*>: Optional abbreviation used within the assembly file. If not specified, then the relocation's name is used instead.
    - *value* = <*int*>: Integer value of the relocation.
    - *pcrel* = <*bool*>: Optional, whether or not this is a pc-relative relocation. If not specified and the relocation is associated with a field, then the flag will be set to true if the instruction field is marked as pc-relative. If specified in the relocation and used by an instruction field, then the flag values must match.
    - *field_width* = <*int*>: Width of field used with this relocation, in bits. If not specified and the relocation is associated with a field, then the width of the relocation will be taken from the field width. If a width is specified and it is also used by an instruction field, then the widths must match.
    - *width* = <*int*>: Optional, width of instructions the relocation was associated with. If specified, and the relocation is used in an instruction with a different width, then the assembler will correct the offset to compensate for the difference. This is applicable if relocation is aligned to the end of the instruction.
    - *offset* = <*int*>: Optional, the relocation offset relative to the instruction width. in bytes. This may be useful if the relocation is not aligned to the end of the instruction.
    - *is_signed* = <*bool*>: Optional (default is false). The relocation is considered a signed value. This is only used for overflow checking. If the relocation is associated with an instruction field, then the instruction field's signed value will be used or must match what is set in the instruction field.

  - *instrfield = <ident> | (<ident>,<ident>)*: Specify an optional instruction field to be associated with this relocation. This allows the linker to know how to set a value in a field, especially for fields with non-contiguous bits. This is also used to associate a default relocation with an instruction field. In the second form, two instruction fields are specified. This allows a relocation to be specialized to a particular micro-operation field and a particular field used to store the micro-operation (the instruction field of the macro-operation).
  - *check_overflow = <bool>*: Optional (default is true). Check for overflow when performing the relocation.
  - *right_shift = <int>*: Optional, used to specify the number of bits the relocation value is right-shifted before it is encoded.
  - *bitpos = <int>*: Optional, used to specify a position of the relocation within the instruction. This acts as a left-shift operation on the value.
  - *action = func(uint64_t,int)*: Specify a special handling function. This function receives the relocation address and should perform any necessary actions, returning the modified value. This is done before any shifting operations are performed, as specified by `right_shift`, etc.

    The second parameter is a flag which is 1 if the function is being invoked by the linker or by the assembler for a non-relocation element, e.g. a local label which will not be emitted as a relocation (the flag is an integer only due to the need to be compatible with C).
  - *src_mask = <int>*: Optional, mask to use to extract out a portion of the source to combine with the relocation value. This is 0 by default.
  - *dst_mask = <int>*: Optional, mask to use to specify the portion of the instruction in which to place the value. This defaults to a value of all ones for the specified field width.

3.3.3.20 Other Definitions

- *active = func() {}*: Called to determine whether the core should be active or not. This is a watch which is only called when the resources mentioned in the function are modified.
- *addr_check = func(addr_t ea,addr_t ra,TransType tt [, int id, int set, int way]) {}*: Called whenever an address is generated for a memory operation. This occurs even for hits within the same translation page. Since this hook is called extremely frequently, it should contain a minimal amount of code in order to maintain performance. It can be used, for example, to implement various kinds of debug exceptions.

The `tt` parameter specifies the type of transaction and can have the values `LoadTrans`, `StoreTrans` or `InstrTrans`.

The `id`, `set`, and `way` parameters specify the TLB and location in the TLB used for this translation. These parameters are optional and may be omitted for performance reasons. They may be -1 to indicate an invalid value, e.g. a real-mode translation which doesn't use a TLB specifies an id of -1. The `id` values may be compared against an enumeration of valid TLBs in the system. Refer to MMU Lookup Identifiers for more information.

- *branch_taken* = *func() { }*: Optional. Called when a write occurs to the next-instruction-address (nia) register other than implicit nia updates performed before each instruction is executed. The nia register has already been updated when this is called, so the new address may be queried by reading the nia register.
- *decode_retry* = *func(addr_t,uint32) { }*: Optional. Called when an instruction does not decode. First argument is the address of the instruction, second is the instruction read. Returns a *boolean*. If it returns *true* decoding is retried else *decode_miss* is invoked.
- *decode_miss* = *func(addr_t,uint32) { }*: Called when an instruction does not decode. First argument is the address of the instruction, second is the instruction read.
- *exception_is_branch* = *<bool>*: If true, an exception is considered a branch, from the point of view of logging and the branch_taken hook. The default is **true**.
- *immediate_writehooks* = *<bool>*: This acts as a global flag, within the core, for specifying whether write-hooks, for a parallel architecture, execute immediately or during the commit phase, at the end of a packet. By default, write-hooks execute during the commit phase, but setting this flag to true means that the write-hook executes immediately. This flag may be overridden by the `immediate` key within the `write` definition of a register or register-file.

- *interleaved_reads* = *<bool>*: If true, then misaligned reads are performed such that translations are interleaved with the data accesses, i.e. **translate**, **read**, **translate**, **read**. If non-interleaved, then all translations are performed first, before any memory-system accesses, i.e. **translate**, **translate**, **read**, **read**. The default is `true`: Reads are interleaved.

- *interleaved_writes* = *<bool>*: If true, then misaligned writes are performed such that translations are interleaved with the data accesses, i.e. **translate**, **write**, **translate**, **write**. If non-interleaved, then all translations are performed first, before any memory-system accesses, i.e. **translate**, **translate**, **write**, **write**. The default is `true`: Writes are interleaved.
- *invalid_entry_read = func(unsigned id,unsigned index) {}*: For sparse register files only. If a read occurs and no entry exists for the specified index, then the default behavior is to abort. This hook allows the user to specify an alternate behavior. It should return a value that is the result of the read operation, raise an exception, etc. This is a global default which will be applied, if it exists, to all sparse register files unless a per-register-file override exists.
- *invalid_entry_write = func(unsigned id,unsigned index,bits<width> value) {}*: For sparse register files only. If a write occurs and no entry exists for the specified index, then the default behavior is to abort. This hook allows the user to specify an alternate behavior. This is a global default which will be applied, if it exists, to all sparse register files unless a per-register-file override exists.
- *itable_watch* = *func() {}*: Specify a watch function for switching instruction tables. The function should return an enum representing the current instruction table in use. This function should be used, versus the direct calling of `setCurrentInstrTable()`, when the instruction table depends upon register values, since this will ensure that the system is consistent after state has been loaded from a source such as a testcase, since write-hook functions are not called during testcase loading.
- *pre_cycle* = *func() {}*: For a standard ISS, this is called immediately before the `pre_fetch` hook, if one exists, and is modified by the clock_cycle_ratio configuration parameter, if one was specified. In a cycle-based mode, this is called just before each clock cycle. The default action is to do nothing.
- *pre_exception* = *func(Exception e) {}*: Called when an exception occurs, before the exception's handler code. The function's argument specifies what type of exception occurred.
- *pre_fetch* = *func() {}*: Called just before an instruction fetch. The default behavior is to do nothing.
- *post_cycle* = *func() {}*: For a standard ISS, this is called immediately after the `post_exec` hook, if one exists, and is modified by the clock_cycle_ratio configuration parameter, if one was specified. In a cycle-based mode, this is called just after the completion of each clock cycle. The default action is to do nothing.
- *post_exception* = *func(Exception e) {}*: Called when an exception occurs, after the exception's handler code. The function's argument specifies what type of exception occurred.

- *post_fetch = func(unsigned instr_size) {}*: Called after an instruction fetch. The argument is the size of the instruction just fetched. The default behavior is:

  NIA = CIA + instr_size;

  where NIA is the next-instruction-address register identified by the *nia* class and CIA is the current-instruction-address register identified by the *cia* class.
- *post_exec = func() {}*: Called just after an instruction is executed. The default behavior is to do nothing.
- *pre_asm = func() {}*: Called just before an instruction is assembled.
- *post_asm = func(unsigned instr_width) {}*: Called just after an instruction is recognized in the assembler. Currently can access only prefix instruction fields and assembler parameters, the argument denotes the current instruction width in bits. Default behavior is to do nothing.
- *post_packet = func() {}*: Called just after an parallel execution set of instructions terminates. Default is to reset prefix data, if defined.
- *post_packet_commit = func() {}*: Called just after an parallel execution set of instructions terminates and all delayed writes have been made. This hook will be able to see architectural changes made by the instructions in the last packet.
- *post_packet_asm = func(InstrBundle b) {}*: Called after all instructions in a parallel execution set are recognized by the assembler, but before instructions are written. Currently only assembler parameters and prefix variables are accessible. Default: Do nothing.

  Note: The default action is to set prefix fields, if the architecture contains prefix fields. If a *post_packet_asm* hook is written, this action is **not** taken by default. Instead, the user must do this explicitly by calling `InstrBundle::set_prefix_fields()`. Otherwise, the prefix values will not be set. This is done so that an architecture may override the default prefix-field setting logic, if necessary.
- *parallel_execution* = <*int*>: If greater then zero, size of parallel execution set, otherwise, serial computation. Default is zero.
- *packet_pos_incr* = <*int*>: Increment amount per instruction within a parallel architecture. If 0, then the packet position is advanced by 1 per instruction. Otherwise, the size of the instruction (in bits) is divided by `packet_pos_incr` and the resulting value is used to increment the position counter, e.g. a value of 16 increments the position by 1 for every 16-bits, so that a 32-bit instruction would increment the position counter by 2.

- *blk* = *<ident>* | *<list>*: List of functional blocks in the architecture. An instruction or instruction field can be associated with a block, which must be defined using this keyword.
- *attrs* = *<ident[(list)]* | *list(ident[(list)])>*: Define attribute flags for various resources. Attributes are simply arbitrary, user-defined symbols, with optional values, which may be used by various generators to alter behavior, or by clients, e.g. users of the ADL database, to convey information. Refer to the Attributes section for more information.
- *reserved_bits_value* = *<0|1>*: Specify the value to use for instruction reserved bits. The default is 0, but this flag allows this to be changed to a value of 1. Setting the flag at the core/architecture level means that the value applies to all instructions, unless overridden on a per-instruction basis.
- *bit_endianness* = *<little|big>*: Optional, specifies bit endianness of the core. Default is big endian (the most significant bit is number 0).
- *instr_endianness* = *<little|big>*: Specifies byte ordering for instructions. The default is **big**.
- *data_endianness* = *<little|big>*: Specifies byte ordering for data. The default is **big**.

4 Predefined Global Resources

4.1 Global Variables

Each instruction action has access to its opcode fields and all registers will appear as globals and may be accessed within a function's action sequence using the register's name:

```
MSR(20) = 1;
```

This is also true for register hook functions. In addition, within hooks, the special name `ThisReg` may be used to refer to the register with which the hook is associated. For example:

```
defmod (reg=(A,B,C)) {
  define (write) {
    action = func(bits<64> value) {
      if (Mode == LowPower) {
        ThisReg = concat( ThisReg.get<32>(0,31) , value.get<32>(32,63) );
      } else {
        ThisReg = value;
      }
    };
  }
}
```

This applies a write hook to registers `A`, `B`, and `C`. The hook is valid for all three registers because of the use of `ThisReg` to modify the register's value.

Register files may be accessed using function-call notation:

```
uint32 r = RS;
GPR(r) = 10;
```

The method *valid(<index>)* may be called on a register file to see if an index is valid:

```
SPR.valid(20);
```

A register's fields will be accessed using the method variable access syntax:

```
CR.CR0 = 0x4;
```

or:

```
bits<4> tmp = CR.CR1;
```

Exception names are also present as global symbols and may be used by the *raiseException* routine to raise an exception.

4.2 Registers

4.2.1 Reading and Writing Registers

As mentioned above, registers are accessed via their name. Registers may be read from and written to, but mutable side-effect operations are not allowed. For example, one cannot do the following:

```
A.set(0,20,3);
```

However, writes to slices of a register are allowed:

```
A(0,15) = 0xdead;
```

Writes to fields are also supported:

```
A.a = 0xdead;
```

For indexed fields:

```
VPR.halfword(i) = ...;
```

Elements of a register file are specified using the function-call notation:

```
GPR(3) = GPR(4) + GPR(5);
```

Slices of register file elements may also be read or written using the same notation as for registers:

```
GPR(x)(0,15) = X;
```

Register files define these additional methods:

- `bool hasAttr(unsigned i,unsigned attr)`: Returns true if the specified index of the register file contains the specified attribute. It is really only useful for sparse register files, as non-sparse register-files are homogeneous and thus all elements contain the same registers. The first parameter is the index to query and the second parameter contains the attribute to check.
- `bool hasAttr(bits<Nb> i,unsigned attr)`: Same as above, except that the index may be a bits object.
- `unsigned size()`: Returns the number of elements in the register-file. For sparse register files, this is the total number of possible elements.
- `bool validIndex(unsigned i)`: Returns true if `i` is a valid index of the register file. For non-sparse register files, this will always return true. For sparse register files, this returns true only if the entry exists.
- `bool validIndex(bits<Nb> i)`: Same as above, except that the index may be a `bits` object.

If fields are defined, for either registers or register-files, then *getter* and *setter* helper functions are defined which allow the user to manipulate bits in an integer which correspond to those of a register field. These functions have the name `getfield_<name>_<field>` and `setfield_<name>_<field>`.

For example, given a register definition such as:

```
define (reg=FOO) {
  define (field=A) { bits = (0,15); };
  define (field=B) { bits = (16,32); };
}
```

user action code may manipulate bits in another integer value using the *getter* and *setter* functions:

```
action = {
  ...
  bits<32> x = ... ;
  ...
  var y = getfield_FOO_A(x);

  y = setfield_FOO_B(x,0x1234);
  ...
};
```

The prototypes for non-indexed register and register-file fields are:

```
// Extract bits corresponding to <field> from x and return the value.
bits<field-width> getfield_<name>_<field>( const bits<reg-width> &x );

// Set the bits corresponding to <field> in x and return this new value.
bits<reg-width> setfield_<name>_<field>( bits<reg-width> x, const bits<field-width> &y);
```

The prototypes for indexed register and register-file fields are:

```
// Extract bits corresponding to <field> from x and return the value.
bits<field-width> getfield_<name>_<field>( const bits<reg-width> &x, unsigned index );

// Set the bits corresponding to <field> in x and return this new value.
bits<reg-width> setfield_<name>_<field>( bits<reg-width> x, unsigned index,const bits<field-width &y);
```

4.2.2 Delayed Register Writes

ADL supports delayed writes to registers. These writes are queued up and applied a certain number of cycles later. The syntax is:

```
delayed_write(R,D) = V;
```

where:

R:
: The register write expression. This may be a register name, a register-file element, a field access, or a slice.

D:
: The delay, in cycles. For example, a value of 1 means that the update should occur on the next instruction, whereas a value of 0 means that the update should occur at the end of the current instruction's execution.

V:
: The value to be written.

For example:

```
delayed_write(NIA,1) = CIA + BD;
```

This feature may be used to implement delay-slots in architectures which have delayed branches. Refer to Delay Slots for more information.

The maximum number of outstanding delayed writes and the maximum number of delay slots are static values determined by model-build configuration options. Refer to Configuration File Syntax for more information.

4.2.3 Direct Register Accesses

A register read or write may be surrounded by the function `direct_access`, implying that the access should not be renamed or tracked if used in a model with micro-architectural knowledge. For example:

```
direct_access(FOO) = direct_access(BAR(x));
```

Registers FOO and BAR should be read and written directly. If the model is a transactional mode, then FOO and BAR will not be part of the instruction transaction. In general, their usage will not be tracked.

4.3 Memory

Memory is a global named *Mem*. Memory is accessed using a function call interface where the first argument is the address and the second is the access size in bytes which must be an integer literal.:

```
Mem(ea,4) = GPR(RS);
```

or:

```
GPR(RT) = Mem(ea,4);
```

The memory object itself is a facade and may hide MMU activity and such.

An optional third parameter may be specified which can be used to specify an access type. This type is interpreted as a `CacheAccess` type, although the parameter is specified as an integer so that arbitrary values may be specified by the user. This is currently only useful for transactional and hybrid ISSs, where this value is propagated to the external driver via the `MemAccess::_type` member.

For example, to perform a memory read which should be interpreted as an instruction read:

```
var foo = Mem(ea,4,CacheIFetch);
```

For custom values, simply add on to the value of `MaxCacheAccess`, e.g. `(MaxCacheAccess+1)` will be one higher than any defined `CacheAccess` enumeration.

An additional five parameter form of memory access also exists for specifying a critical address for logging purposes. This form adds on a flag to specify that the critical word is important and an address parameter which is used for logging. This form can be used to signify a critical address for when the actual address is automatically aligned as part of the instruction action code. For example:

```
var foo = Mem(ea,4,CacheRead,true,orig_ea);
```

In this example, `orig_ea` signifies the original effective address for logging purposes, even though the actual address will be to the address specified by `ea`.

4.3.1 Delayed Writes

Delayed writes to memory are supported using the same syntax as for delayed register writes:

```
delayed_write(M,D) = V;
```

where:

M:
: The memory write expression.

D:
: The delay, in cycles. For example, a value of 1 means that the update should occur on the next instruction, whereas a value of 0 means that the update should occur at the end of the current instruction's execution.

V:
: The value to be written.

For example:

```
delayed_write(Mem(ea,4),1) = GPR(RS);
```

4.4 Local Memory

A named local memory is user defined. Local memory is accessed using a function call interface where the first argument is the address and the second is the access size in bytes which must be an integer literal. Accessing IMem is done using the following syntax:

```
IMem(ea,4) = GPR(RS);
```

or:

```
GPR(RT) = IMem(ea,4);
```

4.4.1 External Register Functions

When a *library-mode* ISS is generated, the resulting class is a template which takes as an argument the name of a base class. This base class must implement external access routines for reading and writing all registers. These functions take the form of:

```
bits<reg-size> <reg-name>_external_read();

void <reg-name>_external_write(const bits<reg-size> &);
```

For register files:

```
bits<reg-size> <reg-name>_external_read(unsigned index);

void <reg-name>_external_write(unsigned index, const bits<reg-size> &);
```

For example, given an ISS which implements a 32-bit register **FOO**, when generated using library-mode, the base class must implement the following functions:

```
bits < 32 > FOO_external_read (  ) const { ... }
void FOO_external_write ( const bits < 32 > &x ) { ... }
```

All accesses to **FOO** by the instruction handlers in the ISS will use these handlers to read and write **FOO's** data.

4.5 MMU API

The datatype `TransType` is used to describe various types of MMU operations. Its possible values are:

InstrTrans
: Specifies that this is a translation being performed for an instruction read operation.

LoadTrans
: Specifies that this is a translation being performed for a data-read operation.

StoreTrans
: Specifies that this is a translation being performed for a data-write operation.

WriteTrans
: Specifies that this is a write to a TLB. This value is generally only used for internal logging operations and is not a value that is generally seen by hook functions.

The MMU may be modified by reading or writing to the appropriate lookup object. For example, if a lookup named *TlbCam* is defined, then you can specify a specific set and way it is written to using the following syntax:

```
TlbCam(set,way) = ... ;
```

and read from using the following syntax:

```
MAS0.EPN = TlbCam(set,way).EPN;
```

The first argument specifies the set and the second argument specifies the way. The return value of the read, or the right-hand-side of an assignment must be the appropriate data type for that lookup, which is assigned the name *<lookup>_t*.

When accessing set fields, the same syntax is used, with only a single argument:

```
MAS1.NV = Tlb4k(set).NV;
```

Please note that currently, writes to set fields do not generate intermediate results. Therefore, implementing an instruction which updates a way as well as a set, update all set fields first, then write to the way, so that a single intermediate result will be generated containing all of the new information.

Additional MMU API functions:

- *bool search(unsigned &set,unsigned &way,addr_t ea,TransType tt)*: Search the specified lookup for a match against the specified address. `TransType` is an enumerated variable with the values `InstrTrans`, `LoadTrans`, and `StoreTrans`. This indicates the type of search to be performed. If the search method returns true, then `set` and `way` are updated with the set and way information of the matching entry.

For example, to see if an address of 0x1000 is found within the `TlbCam` lookup, from the point of view of an instruction read:

```
unsigned set,way;
if (TlbCam.search(set,way,0x1000,InstrTrans)) {
  ...
}
```

- *unsigned num_sets()*: Return the number of sets found within this lookup.
- *unsigned num_ways()*: Return the number of ways per set for this lookup.

In some cases, it may be necessary to explicitly log a translation. For example, a hardware tablewalk might use intermediate TLBs in order to find a page-table entry. Since those entries are not directly used, they will not be logged, but the logging might be required by an external application. For such situations, the function `logMmuTranslation` may be called explicitly:

- *void logMmuTranslation(const MmuBase &t,TransType tt,int seq,addr_t ea,addr_t ra)*: Explicitly log a translation. A reference to the entry to be logged, the translation type, sequence number, effective address, and real address must be supplied. The entry need not exist in a TLB but may be a local variable within a hook function. For example:

```
miss = func(TransType tt,addr_t ea,unsigned seq) {
  addr_t ra = ea | 0xf0000000
  TlbCam_t t;
  t.RPN = ra >> 10;
  t.SIZE = 4;
  logMmuTranslation(t,tt,seq,ea,ra);
  return t;
};
```

4.5.1 MMU Lookup Identifiers

All MMU lookup types in a core are assigned a unique integer for identification purposes. This id can be accessed via `MmuBase::id()` and is an argument to the `addr_check` function. An enumerated type, **TlbIds**, is automatically created which contains enumerations for each of the lookups' identifiers. The enumeration values have the name **<lookup-name>_id**. So, for example, a lookup called **TlbCam** will define the enumeration **TlbCam_id**.

4.6 Cache API

The datatype `CacheAccess` is used to specify various kinds of cache operations. The possible values are:

CacheIFetch
    An instruction fetch operation.
CacheRead
    A data read operation.
CacheWrite
    A data write operation.
CacheFlush
    A flush operation.
CacheTouch
    A touch operation.
CacheAlloc
    A line-allocation operation.
CacheInvalidate
    A line-invalidate operation.
CacheLock
    A line-locking operation.
CacheUnlock
    A line-unlocking operation.
CacheLogRead
    This indicates a read for logging purposes.

Caches may be accessed or modified by calling the appropriate method of the cache object, which has the same name as the cache. So, for example, a cache named *L2* can have a line touched using the following syntax:

```
L2.touch(addr);
```

Within a cache hook, such as the *read_line* or *invalidate_line* functions, the hook may refer to its own cache using the special name *ThisCache*, e.g.:

```
ThisCache.read_from_next(ca,set,way,addr);
```

The cache API is:

- *void allocate(addr_t ra)*: Establishes the line specified by the supplied real address in the cache. This does not load any data into the cache.
- *unsigned get_set(addr_t ra)*: Returns the set associated with the specified address.
- *bool enabled(CacheAccess ca,addr_t addr = 0) const*: Returns true if the cache is enabled, false if not.
- *void fill(addr_t ra,byte_t c)*: All bytes in the line specified by the supplied real address are set to *c*.
- *bool find(int &set,int &way,addr_t addr)*: Looks up *addr* in the cache. If the line is in the cache and is valid, sets *set* and *way* and returns true. Otherwise, does not modify *way* (*set* is modified) and returns false.

- *void flush(addr_t ra)*: Flushes the line specified by the supplied real address. If the line is modified, it is written to the next level in the memory hierarchy. The valid and dirty state of the line is not changed.
- *void flush(int set,int way)*: Flushes the specified line. If the line is modified, it is written to the next level in the memory hierarchy. The valid and dirty state of the line is not changed.
- *bool fully_locked(int set)*: Returns true if the specified set is fully locked, i.e. all ways are locked.
- *bool fully_unlocked(int set)*: Returns true if the specified set is fully unlocked, i.e. all ways are unlocked.
- *bool fully_unlocked()*: Returns true if the cache is currently completely unlocked.
- *bool invalidate(addr_t ra)*: The line specified by the supplied real address is invalidated.
- *bool invalidate(int set,int way)*: The specified line is invalidated.
- *bool is_dirty(int set,int way)*: Returns true if the specified way is dirty.
- *bool is_locked(int set,int way)*: Returns the locked/unlocked status of a particular way.
- *bool is_write_through(CacheAccess ca,addr_t addr)*: Returns true if the cache is in write-through mode. The parameters `ca` and `addr` are passed directly to the relevant write-through predicate function.
- *bool is_valid(int set,int way)*: Returns true if the specified way is valid.
- *bool lock(int set,int way)*: Lock a specific way. The lock persists until *unlock* is called. A locked way will not be selected as a victim unless all ways are locked. Returns the prior value of the lock-state.
- *bool lock(addr_t ra)*: Same as above, but performs a touch first, in order to bring in the line.
- *int num_sets() const*: Returns the number of sets in the cache.
- *int num_ways() const*: Returns the number of ways in a set in the cache.
- *bool read_from_next(CacheAccess ca,int set,int way,addr_t addr,bool do_load)*: Load the cache line specified by *addr* into the specified set and way from the next cache in the hierarchy. Generally called by the cache's *read_line* hook. If *do_load* is true, then if the line is not present in the next-level cache, it will be loaded and the function will return true. If *do_load* is false, then this function will perform no action and will return false.
- *void read_from_mem(CacheAccess ca,int set,int way,addr_t addr)*: Load the cache line specified by *addr* into the specified set and way directly from memory. Generally called by the cache's *read_line* hook.
- *void store(addr_t ra)*: Flushes the line specified by the supplied real address. If the line is modified, it is written to the next level in the memory hierarchy. The cache line remains valid but is no longer considered dirty.
- *void store(int set,int way)*: Flushes the specified line. If the line is modified, it is written to the next level in the memory hierarchy. The cache line remains valid and is no longer considered dirty.

- *void touch(addr_t ra)*: Touches the line specified by the supplied real address. This loads the line from the next level of the memory hierarchy if it is not currently in the cache.
- *bool unlock(int set,int way)*: Unlock a specific way. Returns the prior value of the lock state.

In addition, cache fields defined by the user may be accessed using a function call notation:

```
ThisCache(set).field = ... ;
```

or:

```
ThisCache(set,way).field = ... ;
```

4.7 Context API

Limited access to elements of a context and information about a context is accessible through the context object. The name of this object is the name of the context, e.g. a context named `thread` may be accessed through the object named `thread`.

The following methods are available:

- *clear(unsigned index)*: Clear all items in the specified context element to 0.
- *num_contexts() const*: Return the number of elements for this context.
- *unsigned active_context() const*: Returns the index of the active context element.
- *void write(unsigned index,addr_t addr)*: Write the specified context element to memory.
- *void read(unsigned index,addr_t addr)*: Read the specified context element from memory.

In addition, specific elements of a context may be accessed using the function-call operator. For example:

```
thread(0).FOO
```

This will access the register named `FOO` in element 0 of the context named `thread`. Note that accessing elements in this manner bypasses the usual read and write hooks. Thus, these accesses are only allowed for simple, non-pseudo registers or register-files. Reads and writes to register fields and slices of registers are allowed, using the same syntax as for normal registers.

4.8 Global Functions

Various predefined helper functions exist for handling common operations.

- *unsigned getChildId()*: Returns the integer ID associated with each child. This is the same value as used for child-relative indexing.
- *unsigned getCoreId()*: Returns the integer ID associated with each core. By default, this is a monotonically increasing value, incremented for each core that is allocated in the simulation, though within a platform, this may be controlled by the integration code through `createTopLevelNode`. This may be used, for example, to initialize a processor ID register within a core.
- *uint64_t getInstrCount()*: Returns the number of instructions executed by the core.
- *uint64_t getCycleCount()*: If the model is using the temporal decoupling API, then this returns the temporal decoupling counter. Otherwise, this returns the current instruction count.
- *unsigned getMicroOpOffset()*: For architectures with packetized encodings (micro operations) and which use the micro-operation offset feature, this returns the current micro-operation offset, which specifies which instruction in a packet is executing. This may be used to store the offset so that a branch can set up a return value to return to the next micro operation.
- *bool instrHasAttr(<attribute name>)*: Returns true if the currently executing or assembled instruction has the specified attribute. This function requires that a literal be supplied as an argument. The literal may be either a quoted string or an identifier.
- *bool instrHasAttrVal(<attribute name>,<int|string>)*: Returns true if the currently executing or assembled instruction has the specified attribute and the attribute has the specified value. This function requires that a literal be supplied as an argument. The literal may be either a quoted string or an identifier. The value may be either an integer or a string.
- *bits<size2> signExtend(bits<size1> x,int size2)*: Returns the value of *x* within a bits object *size2* bits in width and replicates the sign bit.
- *bits<size2> zeroExtend(bits<size1> x,int size2)*: Returns the value of *x* within a bits object *size2* bits in width. Any needed leading bits are set to 0.
- *bits<size1+size2> signedMultiply(bits<size1> x,bits<size2> y)*: Returns the product of *x* and *y* using signed multiplication.
- *bits<max(size1,size2)> signedDivide(bits<size1> x,bits<size2> y)*: Returns the quotient of *x* and *y* using signed division.
- *bits<size> zero(int size)*: Returns a bits object *size* bits in width containing the value 0.
- *bits<max(size1,size2)> Carry(bits<size1> x,bits<size2> y, bits<1> cin)*: Returns the carry-out of each bit from the operation (x+y+cin).
- *bits<size1+size2...> concat(bits<size1> x,bits<size2> y, ...)*: Returns a bits object whose width is the sum of the widths of all arguments, containing the result of concatenating all arguments together. Note: This operation is currently limited to a maximum of 5 arguments.
- *void cancelException(<exception-name>)*: Cancel a pending level-sensitive exception.

- *void raiseException(<exception-name>)*: Cause the specified exception to occur immediately.
- *void raiseException(const <exception>_t &)*: This will cause the exception to be immediately processed. Field data for the exception will be take from the supplied argument.
- *void setException(<exception-name>)*: Register the fact that the specified exception has occurred. Execution of the current instruction continues; the exception will be handled once the instruction has completed.
- *setException(const <exception-name>_t &)*. This will cause the exception to be handled after the instruction has completed. Field data for the exception will be take from the supplied argument.
- *void resetCore()*: Reset the core to its power-on-reset state. The new initial state is dumped as intermediate results, if tracing is enabled. Note that this type of reset operation, as opposed to the start of a simulation, which is considered a power-on-reset operation, will leave any registers or register files without a specified reset value unchanged.
- *void setCurrentInstrTable(<instr table name>)*: Modify the current instruction table to the one specified.
- *void setMicroOpOffset(<unsigned>)*: Set the micro-operation offset to control micro operation execution for a subsequent instruction. By default, all micro ops in an instruction execute, but this allows one instruction to control such execution, allowing a branch, for example, to selectively control which micro instructions execute in the branch target. This only works if the subsequent instruction has an implicitly generated action function.
- *void syscall_add_arg([bits<n>|uint64_t] arg)*: Add an argument to the list of system call arguments. This list is cleared by a call to *syscall_trigger*.
- *bool syscall_enabled()*: Returns true if system calls are enabled, false otherwise.
- *uint32_t syscall_errno()*: Returns the *errno* value from the last system call that was triggered.
- *int64_t syscall_return_code()*: Returns the return code from the last system call that was triggered.
- *void syscall_trigger(<syscode>)*: Trigger a system call. The reason code is supplied as an argument.
- *int getCurrentInstrTable():* Returns the name of the current instruction table. It allows to define table dependent handlers.
- *void retryDecode()*: Retry decoding the last fetched instruction. Enables to switch tables on decode miss.
- *void halt()*: Cause the core which calls this function to cease to be active.
- *void set_active_count(unsigned)*: Set the core's activity. Calling this with 0 is the same as calling `halt()`.
- *unsigned get_active_count()*: Return the core's current active status. For a core, this will always be 1 if the core is active.
- *bool is_active()*: Returns true if the core is currently active, false otherwise.

- *void skip_instruction()*: Skips execution of the next instruction. This call may only be placed within the *pre_fetch* hook.
- *addr_t instrReadTranslate(addr_t ea)*: Translates the supplied effective address and returns real address.
- *addr_t dataReadTranslate(addr_t ea)*: Translates the supplied effective address and returns real address.
- *addr_t dataWriteTranslate(addr_t ea)*: Translates the supplied effective address and returns real address.
- *void checkDataPerms()*: This forces checking of any data-related permissions, e.g. the load_perm and store_perm permission check specified in the MMU configuration.
- *void checkInstrPerms()*: This forces checking of any instruction-related permissions, e.g. the exec_perm permission check specified in the MMU configuration. By default, permissions are not always checked if they would normally pass. For example, a branch to an address in the same page as the branch is assumed to be executable. However, in some cases, this assumption is incorrect. For example, the branch might be to an incorrectly aligned address and the `exec_perm` hook might be used to check for proper alignment.
- *addr_t lastRealAddr()*: Returns the result of the last translation done by the MMU.
- *uint32_t *getCurrentInstr(int &num_bytes)*: Returns a pointer to the last instruction fetched and its lenght in bytes.
- *unsigned getPacketPosition() const*: Get the word position of the current instruction in

    the parallel execution set. Defined only for parallel architectures which use prefix.
- *void setPacketPosition(unsigned pos)*: Set the word packet position counter to specified value. If `pos` is zero, execution set results are commited and `post_packet` handler, if defined, is invoked. Defined only for parallel architectures which use prefix.
- *bool instrBlk(<blk name>)*: Returns true if the current assembled instruction is associated with the specfied block. This function requires that a literal be supplied as an argument.
- *void get_dest(RegWrites &rw)*: Populates it argument with a list of registers and write masks of the currently assembled instruction and sets the *size* data member to the number of such pairs.
- *void intersect_dest(const RegWrites &rw1,const RegWrites &rw2, RegWrites &rw3)*: This function populates its last argument with the registers common to its first two arguments and the corresponding write masks.
- *run(instrfield &field, ...)*: This function executes the instructions within the fields as specified in the arguments. Note that all the fields must be instruction-type.
- *run_commit(instrfield &field, ...)*: This function executes the instructions within the fields as specified in the arguments and then commits after the last instruction. Note that all the fields must be instruction-type.

- *void info(int level, ... [AdlDataStart , (key,value) ...])*: Creates an annotation message which is then sent to the logging system, e.g. any test writers which are present. The *level* parameter allows users to assign a priority to each message for filtering purposes. All subsequent arguments, up to the special symbol `AdlDataStart` (if it exists) are converted to a string and sent to the writer. You may use standard C++ *iomanip* functions for modifying printing, e.g. *info(1,"Address is: 0x",hex,x);*.

  Any arguments after the `AdlDataStart` symbol are assumed to be additional data arguments of the form `("key",value)`, where the key is a string and the value is an integer or string (constant character pointer) expression.

  This message is only sent if tracing is enabled.
- *void warning(int level, ...)*: Same as `info`, except that this message will always be sent, even if tracing is off.
- *void error(int level, ...)*: Same as `info`, except that this message will always be sent, even if tracing is off. It will also abort the simulation by throwing a C++ `runtime_error`.

4.8.1 External Pipeline Model API

These functions are designed for enabling communication between the ISS and an external application, such as a performance model. An unsigned integer is available as a flag for storing arbitrary information. This may be retrieved or set via these functions within the ISS.

- *unsigned getExtInstrFlag()*: Returns the external-application instruction flag value. Returns 0 for a default, standalone ISS.
- *void setExtInstrFlag(unsigned)*: Set the external-application instruction flag. No action when executed within a default, standalone ISS.

4.8.2 Rollback API

- *void enable_rollback(bool e)*: Toggle rollback on or off. This is only valid if the model has been built with rollback (via the **--rollback-mode** option).
- *commit_rollback()*: Apply the items in the stack. This undoes the changes made since the last *enable_rollback* call.
- *flush_rollback()*: Discard all items from the stack.

4.8.3 Ignore-Mode API

- *void set_ignore_mode(bool i)*: Toggle ignore-mode on or off. When on, writes to registers and memory are ignored and memory reads return 0 immediately, with no read performed. For a hybrid ISS, usage of the respective register resource is still logged. This is only valid if the model has been built with ignore-mode support (via the **--ignore-mode** option).

- *bool ignore_enabled()*: Returns true if ignore-mode is enabled.

4.9 System-Call Support

ADL contains a simple interface for handling system calls. This is simply the ability for a program running on an ADL simulator to interact with the host operating system in order to perform such actions as file I/O, time queries, etc. By default, the following system calls are implemented:

Supported System Calls

System Call	Code	Description
exit	1	Currently, no action is taken.
read	3	OS file read.
write	4	OS file write.
open	5	OS opening of a file.
close	6	OS closing of a file.
annotate	7	Add a message to the trace.
core-id	10	Returns the core's unique id.
brk	17	Adjust system memory. Currently, no action is taken and this always succeeds.
access	33	OS file permission check.
dup	41	OS duplication of a file descriptor.
gettimeofday	116	Returns current cycle/instruction count in the simulator.
sched_yield	158	Yield current thread to scheduler. Mainly only useful for multi-threaded simulations.
getcwd	183	Return the current working directory.
lseek	199	OS file seek.

Arguments for the system call must first be set up by calling *syscall_add_arg(<arg>)*. The system call is then triggered by calling the function *syscall_trigger(<syscode>)*. The return code and errno value may then be retrieved by calling *syscall_return_code()* and *syscall_errno()*.

Generally, system calls will be triggered by some sort of a special instruction. For example, the PowerPC **sc** instruction might be set up to directly trigger system calls. However, this scheme would not be suitable for a simulator which is intended to run operating system code. In such a situation, an artificial instruction might be created to trigger the system call.

For example, the following code shows how a PowerPC **sc** instruction might be implemented:

```
define (instr="sc") {
  fields=(OPCD(17),XO_DS(2));
  action = {
    if (syscall_enabled()) {
      syscall_add_arg(GPR(1));        // stack pointer (brk needs it)
      syscall_add_arg(GPR(3));        // arg0
      syscall_add_arg(GPR(4));        // arg1
      syscall_add_arg(GPR(5));        // arg2
      syscall_trigger(GPR(0));        // syscode - 32 bit mode
      GPR(3) = syscall_return_code(); // the return value
    } else {
      raiseException(sc);
    }
  };
}
```

The calling convention demonstrated in this example matches the system-call code utilized by GNU GCC and *newlib*.

New system calls may be added to the simulator using the plugin interface. Refer to The ADL Plugin Interface document for more information.

5 Usage Notes

This section contains snippets of ADL code for the purpose of illustrating how various architectural elements might be implemented within ADL.

5.1 32/64 Mode Behavior

In order to implement Power within ADL and then model a 64-bit implementation, it is first necessary for the Power specification to describe what behavior is undefined in 32-bit mode and then for an implementation to specify its behavior. First, a mode parameter is needed:

```
define (parm=Mode) {
  options = (Mode32,Mode64);
  constant = false;
}
```

This is non-constant so that it may be changed by a modification of processor state. For example, the write-hook of a register might change it:

```
define (reg=HID0) {
  """
  A hardware control register.
  """;
  define (write) {
    // If HID0[0]==1, the implementation operates in 64-bit mode,
    // otherwise it operates in 32-bit mode.
    action = func(bits<32> value) {
      HID0 = value;
      if (!value(0)) {
        Mode = Mode32;
      } else {
        Mode = Mode64;
      }
    };
  }
}

// This adds HID0 to the SPR register file at index 50.
defmod (regfile=SPR) {
  define (entry=50) { reg = HID0; }
}
```

An *addc.* instruction's action code would use the tristate class to propagate the unknown value:

```
define (instr="addco") {
  fields=(OPCD(31),RT,RA,RB,XO_X(522),RC(0));
  action = {
    var m = (Mode == Mode64) ? 0 : 32;
    var carry = Carry(GPR(RA),GPR(RB),0);
    XER.OV = carry(m) ^ carry(m+1);
    XER.SO = XER.SO | XER.OV ;
    var sum = GPR(RA) + GPR(RB);
    sum.setUndefined(0,m);
    GPR(RT) = sum;
    XER.CA = carry(m);
  };
}
```

The *sum* variable is a tri-state which will have its undefined mask set for the first 32-bits. When this is assigned to the GPR, its write-hook will be called:

```
define (regfile=GPR) {
  define (write) {
    action = func(unsigned index,bits<64> value) {
      if (value.isUndefined(0,31) && (CompatMode == G4)) {
        GPR(index) = concat(GPR(index)(0,31),value(32,63));
        return;
      }
      GPR(index) = value;
    };
  }
}
```

In the above code, if the upper half of the supplied value is undefined, then if we are in G4 compatibility mode (determined by a parameter named *CompatMode*), the upper half of the register will retain its prior value. Otherwise, we will store the complete value into the GPR.

5.2 64-Bit Only Instructions

Some instructions in Power, such as *rldic*, are undefined for 32-bit operation. If the desired behavior of a particular implementation is to simply leave a target register unchanged, for example, then this can be handled using a write-hook on the register file, as shown above. However, if the behavior is to take an exception, then an *aspect* can be used. For example:

```
define (aspect) {
  instr = rldic;
  before = true;
  action = {
    if (Mode == Mode32) {
      raiseException(Unimpl);
      return;
    }
  }
}
```

This will insert the action code before the existing code for the *rldic* instruction. If the core is in 32-bit mode, an unimplemented-instruction exception will be taken and execution of the instruction will be skipped.

Alternatively, the register write-hook could be modified so that if the entire register result is undefined, an exception is taken:

```
define (regfile=GPR) {
  define (write) {
    action = func(unsigned index,bits<64> value) {
      if (value.isUndefined(0,63)) {
        raiseException(Unimpl);
      } else if (value.isUndefined(0,31)) {
        if (CompatMode == G4) {
          GPR(index) = concat(GPR(index)(0,31),value(32,63));
        } else {
          GPR(index) = value;
        }
      } else {
        GPR(index) = value;
      }
    };
  }
}
```

5.3 Simd Registers and Instructions

Indexed register fields provide a convenient mechanism for manipulating SIMD registers. For example, the Altivec vector registers might be described with the following code:

```
define (regfile=VPR) {
  """
  Altivec vector registers.
  """;
  width=128;
  size = 32;

  define (field=B) {
    indexed = 8;
  }
  define (field=H) {
    indexed = 16;
  }
  define (field=W) {
    indexed = 32;
  }
}
```

This describes a register file that has 32 registers, each 128 bits in width. Three indexed fields are also described: Each **B** field is 8 bits in width for a total of 16 fields, each **H** field is 16 bits in width for a total of 8 fields, and each **W** field is 32 bits in width, for a total of 4 fields.

Within a function, an indexed field is selected using the method-call syntax. For example:

```
define (instr=vaddubm) {
  fields=(OPCD(4),RT,RA,RB,XO(0));
  action = {
    for (unsigned i = 0; i != VPR(RT).size()/8; ++i) {
      VPR(RT).B(i) = VPR(RA).B(i) + VPR(RB).B(i);
    }
  };
}

define (instr=vadduhm) {
  fields=(OPCD(4),RT,RA,RB,XO(32));
  action = {
    for (unsigned i = 0; i != VPR(RT).size()/16; ++i) {
      VPR(RT).H(i) = VPR(RA).H(i) + VPR(RB).H(i);
    }
  };
}

define (instr=vadduwm) {
  fields=(OPCD(4),RT,RA,RB,XO(64));
  action = {
    for (unsigned i = 0; i != VPR(RT).size()/32; ++i) {
      VPR(RT).W(i) = VPR(RA).W(i) + VPR(RB).W(i);
    }
  };
}
```

The above code shows three vector add instructions, each of which use a different data-element size. Rather than forcing the bit-indexing arithmetic to be in the instruction, the indexed fields allow this to be abstracted out so that the description remains concise.

5.4 MMU Modeling

This section discusses how ADL may be used to model memory management units (MMUs). The examples in this section are meant to be representative of how an MMU might be modeled and are not meant to be 100% accurate with respect to the actual device they purport to model.

A Power implementation's MMU might be modeled in the following manner:

```
// Power MMU
define (mmu) {

  both_enable = HDBCR0(MMU_ENABLE);

  define (lookup=TlbCam) {

    define (field=V)    { bits = 1; };
    define (field=TID)  { bits = 32; };
    define (field=SIZE) { bits = 4; };
    define (field=RPN)  { bits = 22; };
    define (field=EPN)  { bits = 22; };
    define (field=UX)   { bits = 1; };
    define (field=SX)   { bits = 1; };
    define (field=UW)   { bits = 1; };
    define (field=SW)   { bits = 1; };
    define (field=UR)   { bits = 1; };
    define (field=SR)   { bits = 1; };
    define (field=WIMGE){ bits = 5; };
    define (field=UA)   { bits = 4; };

    define (array) {
      entries = 16;
    }
    type = Both;
    pagesize = SIZE;
    test = Compare(V,1);
    test = Compare(TS, ((Instr) ? MSR(IR) : MSR(DR)) );
    test = Compare(TID,0,PID0,PID1,PID2);
    test = AddrComp(EPN);
    realpage = RPN;
    define (execute) {
      require = (MSR(PR)) ? SX : UX;
      fail = {
        raiseException(ProtectionFault);
      };
    }
    define (load) {
      require = (MSR(PR)) ? SR : UR;
      fail = {
        raiseException(ProtectionFault);
      };
    }
    define (store) {
      require = (MSR(PR)) ? SW : UW;
      fail = {
        raiseException(ProtectionFault);
      };
    }
  }

  define (lookup=Tlb4k) {

    define (field=V)    { bits = 1; };
    define (field=TID)  { bits = 32; };
    define (field=RPN)  { bits = 22; };
    define (field=EPN)  { bits = 22; };
    define (field=UX)   { bits = 1; };
    define (field=SX)   { bits = 1; };
    define (field=UW)   { bits = 1; };
    define (field=SW)   { bits = 1; };
    define (field=UR)   { bits = 1; };
    define (field=SR)   { bits = 1; };
    define (field=WIMGE){ bits = 5; };
```

```
      define (field=UA)   { bits = 4; };

      define (array) {
        entries = 128;
        setassoc = 2;
      }

      pagesize = 12; // Size in bits.
      match = Compare(V,1);
      match = Compare(TS, ((Instr) ? MSR(IR) : MSR(DR)) );
      match = Compare(TID,0,PID0,PID1,PID2);
      match = AddrComp(EPN);
      realpage = RPN;
      define (execute) {
        require = func(TransType t) {
          return (MSR.PR()) ? t.SX : t.UX;
        };
        fail = func(TransType) {
          raiseException(ProtectionFault);
        };
      }
      define (load) {
        require = func (TransType t) {
          return (MSR.PR()) ? t.SR : t.UR;
        };
        fail = func (TransType) {
          raiseException(ProtectionFault);
        };
      }
      define (store) {
        require = func (TransType t) {
          return (MSR.PR()) ? t.SW : t.UW;
        };
        fail = func (TransType t) {
          raiseException(ProtectionFault);
        };
      }
    }

    instr_miss = {
      raiseException(ITlbMiss);
    };

    data_miss = {
      raiseException(DTlbMiss);
    };

    multihit = {
      raiseException(MachineCheck);
    };

  }
```

This model defines two *lookup* objects: TlbCam and Tlb4k. The TlbCam lookup object is fully associative and consists of 16 entries. In order for an address to match an entry, four matching criteria must be met:

- *match = Compare(V,1)*: The *V* field of the entry must be equal to 1.

- *match = Compare(TS, ((Instr) ? MSR(IR) : MSR(DR)))*: The *TS* field of the entry must be equal to *MSR(IR)* if this is an instruction translation, or *MSR(DR)* if it is a data translation.
- *match = Compare(TID,0,PID0,PID1,PID2)*: The TID field must be equal to 0 or the value of one of the PID registers.
- *match = AddrComp(EPN)*: The address must match the *EPN* field, with the offset masked out.

If a match is found, the various permission checks are made. These examine whether the system is in privileged or user mode, then test the appropriate bit. The other lookup object, *Tlb4k*, operates in an analogous manner. Since neither has a priority assigned, if both lookup objects find a match, the *multihit* hook executes, which generates a machine check exception.

A PowerPC classic MMU model might look like the following:

```
define(mmu) {

  instr_enable = { return MSR.IR(); };
  data_enable = { return MSR.DR(); };

  define (lookup=BatBase) {
    interface = true;
    priority = 0;

    define (field=BEPI) {
      bits = 15;
    }
    define (field=BL) {
      bits = 11;
    }
    define (field=Vs) {
      bits = 1;
    }
    define (field=Vp) {
      bits = 1;
    }
    define (field=BRPN) {
      bits = 15;
    }
    define (field=WIMG) {
      bits = 4;
    }
    define (field=PP) {
      bits = 2;
    }

    define (array) {
      entries = 8;
    }
    pagesize = (BL,LeftMask);
    realpage = BRPN;
    test = Check( (MSR.PR() == 0) ? Vs : Vp );
    test = AddrComp(BEPI);
    define (execute) {
      require = func(TransType t) {
        return (t.PP(1) == 1);
      };
      fail = func(TransType) {
        raiseException(ProtectionFault);
      };
    }
    define (load) {
      require = func(TransType t)  {
        return (t.PP(1) == 1);
      };
      fail = func(TransType) {
        raiseException(ProtectionFault);
      };
    }
    define (store) {
      require = func(TransType t) {
        return (t.PP == 2);
      };
      fail = func(TransType) {
        raiseException(ProtectionFault);
      };
    }
  }

  define (lookup=IBat) {
    inherit = BatBase;
    type = Instr;
  }

  define (lookup=DBat) {
    inherit = BatBase;
    type = Data;
  }

  define (lookup=Seg) {
    type = Both;
    priority = 1;

    define (field=T) {
      bits = 1;
    }
    define (field=Ks) {
      bits = 1;
    }
    define (field=Kp) {
      bits = 1;
    }
    define (field=N) {
      bits = 1;
    }
    define (field=VSID) {
      bits = 24;
    }

    define (array) {
      entries = 16;
      set_assoc = 1;
    }
    test = AddrIndex(0,3);

    define (lookup=PTE) {
      priority = 0;

      define (field=V) {
        bits = 1;
      }
      define (field=VSID) {
        bits = 24;
      }
      define (field=H) {
        bits = 1;
      }
      define (field=API) {
        bits = 16;
      }
      define (field=RPN) {
        bits = 20;
      }
      define (field=R) {
        bits = 1;
      }
      define (field=C) {
        bits = 1;
      }
      define (field=WIMG) {
        bits = 4;
      }
      define (field=PP) {
        bits = 2;
      }

      define (array) {
        entries = 128;
        set_assoc = 2;
      }
      pagesize = 12;
      test = Compare(VSID,Seg.VSID);
      test = AddrComp(API);
      realpage = RPN;
      define (execute) {
        require = func(PTE p,Seg s) {
          return (s.N == 1);
        };
        fail = func(PTE p,Seg s) {
          raiseException(ProtectionFault);
        };
      }
      define (load) {
        require = func(PTE p,Seg s) {
          return !((MSR.PR() ? s.KS : s.KP) == 1 && (p.PP == 0));
        };
        fail = func(PTE p,Seg s) {
          raiseException(ProtectionFault);
        };
      }
      define (store) {
        require = func(PTE p,Seg s) {
          return !(((MSR.PR() ? s.KS : s.KP) == 1 && (p.PP == 1)) || (p.PP==11));
        };
        fail = func(PTE p,Seg s) {
          raiseException(ProtectionFault);
        };
      }
    }
  }

  instr_miss = func (addr_t,int) {
    raiseException(ITlbMiss);
  };
```

```
  data_miss = func (addr_t,int) {
    raiseException(DTlbMiss);
  };

}
```

The *IBat* and *DBat* lookup objects are used for instruction and data translations, respectively. They inherit from *BatBase* in order to reduce code duplication. Note that the BAT registers, in this case, are stored as a fully-associative MMU array. The BAT registers defined within the design, rather than being real registers, would in fact be pseudo registers with read and write hooks which reference this array.

The other lookup object is named *Seg* and does a segment translation. If a hit is found, a child lookup object, *PTE*, is then called. If this fails, then the *miss* handler is called, which performs a hardware table-walk operation in memory. The classic MMU architecture allows for overlapping Bat and PTE translations with BAT taking precedence, so the *priority* keyword is used to specify the priority of the two lookups.

5.5 Delay Slots

Delay slots may be modeled within ADL by using the *delayed_write* feature:

```
define (reg=CIA) {
  """
  Current instruction address.
  """;
  attrs = cia;
}

define (reg=NIA) {
  """
  Next instruction address.
  """;
  attrs = nia;
}
```

A branch would then update the NIA with the target address, where the write should take place one instruction later:

```
define (instr=bc) {
  fields=(OPCD(16),BO,BI,BD,AA(0),LK(0));
  syntax = ("%x,%x,%-2a",BO,BI,BD);
  action = func() {
    if ( (BO(2) ) == 0) {
      CTR = CTR - 1;
    }
    var ctr_ok = BO(2) || ( (CTR!=0) ^ BO(3));
    var cond_ok = BO(0) || ( CR(BI) == BO(1));
    if ( ctr_ok && cond_ok ) {
      var ea = signExtend(concat(BD,zero(2)),32);
      delayed_write(NIA,1) = CIA + ea;
    }
  };
}
```

The following code demonstrates this in action:

```
 addi r1,r1,1
 bc 20,0,L1
 addi r2,r2,1
 addi r3,r3,1
L1:
 addi r4,r4,1
```

This results in the following trace:

```
INSTR 0x0 0x0 0x38210001 #1   addi 1,1,1
R GPR 1 0x00000001

INSTR 0x4 0x4 0x4280000c #2   bc 20,0,3
R TIA 0 0x00000010

INSTR 0x8 0x8 0x38420001 #3   addi 2,2,1
R GPR 2 0x00000001

INSTR 0x10 0x10 0x38840001 #4   addi 4,4,1
R GPR 4 0x00000001
```

Notice that the instruction executed after the branch is the instruction which occurs immediately after the branch in memory, followed by the target of the branch.

The following code demonstrates the behavior of a branch within a delay slot:

```
 addi r1,r1,1
 bc 20,0,L1
 bc 20,0,L2
 addi r1,r1,1
 addi r1,r1,1
 addi r1,r1,1
L1:
 addi r2,r2,1
 addi r3,r3,1
L2:
 addi r4,r4,1
 addi r5,r5,1
```

This results in the following trace:

```
INSTR 0x0 0x0 0x38210001 #1   addi 1,1,1
R GPR 1 0x00000001

INSTR 0x4 0x4 0x42800014 #2   bc 20,0,5
R TIA 0 0x00000018

INSTR 0x8 0x8 0x42800018 #3   bc 20,0,6
R TIA 0 0x00000020

INSTR 0x18 0x18 0x38420001 #4   addi 2,2,1
R GPR 2 0x00000001

INSTR 0x20 0x20 0x38840001 #5   addi 4,4,1
R GPR 4 0x00000001

INSTR 0x24 0x24 0x38a50001 #6   addi 5,5,1
R GPR 5 0x00000001
```

5.6 Split Branches

ADL naturally supports the modeling of split branches as simply computed branches. A split branch is a branch in which the target of the branch is computed in one instruction e.g. prepare-to-branch, then the actual change of control occurs in a subsequent instruction. However, simulating computed branches via a decode-cache or JIT model is slower than a branch with a fixed target.

To support higher simulation speeds with split branches, the user must identify relevant instructions in an ISS configuration file (.ttc file). The setup instructions are listed with the **split_branch_begin** key in a core's *config* block and the actual branch is listed with the **split_branch_end** key. An instruction which should cancel the normal split branch behavior, e.g. an instruction which moves a different value to the branch target register, should be listed with the key **split_branch_cancel**.

For example, the ADL description might look like this:

```
define (core=Foo) {
  define (instr=ptb) {
    fields=(OPCD(20),LI,AA(0),LK(0));
    attrs = split_branch_1;
    action =  {
      CTR = LI;
    };
  }

  define (instr=bctr) {
    fields=(OPCD(19),BO(0),BI(0),LK(0),XO(528));
    attrs = split_branch_2;
    action =  {
      NIA = CTR;
    };
  }
}
```

The corresponding configuration file would look like this:

```
define (core=Foo) {
  define (config) {
    split_branch_begin = ptb;
    split_branch_end = bctr;
  }
}
```

In this example, the **ptb** instruction moves a pc-relative value to the CTR register and is marked as the first half of a split branch via the **split_branch_begin** key in the configuration file. The **bctr** instruction changes program flow. Since it is marked as the second half of a split branch, a decode-cache model will stay within the simulation kernel, since the target of the branch can be computed at translation time.

5.7 VLIW instructions

In VLIW, architecture instructions are executed in packets (sets). The assembler (compiler) determines the packets as well as their instruction order. Thus, in VLIW architecture, there isn't any need for the hardware to verify dependencies between instructions in the same execution packet.

Packet instructions read the data, execute in parallel, and queue their updates until all packet instructions are executed. Updates are then committed as per a packet's instruction order. The assembler guarantees data isn't overwritten, except in the case of status registers, and that a packet's instruction order results in correct values.

A VLIW packet is a single pipeline entity whose timing is determined by the packet's slowest instruction. Consequently, an ADL execution step in single-step mode consists of a full VLES execution. If defined, an ADL key *parallel_execution* specifies the maximal packet size allowed in an architecture. The key doesn't have a default value and by not setting one it implies that the architecture isn't VLIW. NOTE: Due to differences in executing updates, *parallel_execution = 1* isn't equivalent to not setting a key.

5.8 Prefix Instructions

In parallel architectures like StarCore the prefix instructions encode, in the execution set, information common to all instructions. A prefix is an instruction field with a key named 'prefix' set to a value of 'true'.

Prefixes encode two types of information:

1. Information that enhances instruction encoding, e.g., an extra bit for instruction encoding.
2. General execution set information, e.g., size. This information, encoded by a single instruction field, is stored by the core and is accessible using the field's name. See the example below:

   ```
   define(instrfield=VlesSize) {  // Prefix field definition
     width = 3;
     prefix = true;
     pseudo = true;
   };
   ```

This field can be used in action code:

```
post_exec = func() {
   if (getPacketPosition() == VlesSize) {
     setPrefixPosition(0);
   }
};
```

At the start of an execution set prefix fields reset to their default value. This handles sets without a prefix instruction or to fields that do not occur in a set's prefix.

The relation between instruction-specific information and regular instructions is described using pseudo-instructions. Consider three instruction fields named *HighRegister*, *Esg*, and *Enc*, where the first two are prefix fields and the latter is regular. The pseudo-instruction is as follows:

```
define(instr=OneWord) {
  fields = ((bits(0,15),Enc),
           (bits(16,18),HighRegister),
           (bits(19,15),Esg));
  pseudo = true;
};
```

Instructions with a *OneWord* key take four encoding bits from *HighRegister* and *Esg* prefix fields. These types of prefix fields are indexed- each subfield slice may correspond to an execution set instruction. Assigning bits to instructions is specified in a field's action code. Consider the following prefix fields:

```
 define(instrfield=HighRegister) {
   indexed = 3;
   prefix = true;
   pseudo = true;
   blk = (agu,dalu);
 };

 define(instrfield=HighDalu) {
   prefix=true;
   pseudo = true;
   blk = dalu;
   indexed = 3;
   type = HighRegister;
   action = {
     unsigned i = getPacketPosition();
     ThisField = bits((i*3),(i*3)+2);
  };
};
```

The above action code specifies that a dalu instruction at position *i* gets the *ith* triplet in a this field. This field is a realization of the *HighRegister* prefix field, i.e., if a dalu instruction uses *HighRegister* information in its encoding and the current prefix had a *HighDalu* field then the information is dispatched to the instruction.

Note: *HighRegister* is associated with blocks agu and dalu while *HighDalu* corresponds only to a dalu block. This notation indicates that a *HighRegister* field may store agu or dalu information or both. Field information is distributed to instructions according to their block (*blk* key value); this is done using rules specified in the action code of the fields that occur in the current prefix instruction.

5.9 Assembler Instructions

In some architectures, such as StarCore, assembler tasks are more complex than just recognizing a single instruction and correctly encoding it. The assembler should handle packets of instructions, add required prefix information, and allow an instruction to affect encoding of a future instruction or packet.

The *assembler* resource is used to define assembler configurations and resources.

packet_grouping defines characters or character-pairs used as packet separators.

queue_size defines an instruction queue or instructions packets (for parallel-execution architectures). The queue allows delayed encoding of packets; the last packet in the queue is encoded with each step. This allows assembler action code to address preceding instructions or instruction packets.

For pseudo prefix instruction fields ADL allocates an integer variable accessible using the field name, by an instruction's assembler code or by the `post_asm` and `post_packet_asm` hooks. If *queue_size* is larger than *1*, then the field is accessed using a C array notation, where index *0* denotes the current packet. ADL adds an prefix instruction to the encoding if any prefix variable value differs from its default value.

The *parm* block in the *assembler* resource can define additional parameters of *integer* or *label* type. Integer parameters must have a default value. Label parameters are not allowed to have a default.

The code below is a simplified version of StarCore's **skipls** instruction. The instruction is a conditional jump whose target always follows the 'loopend' assembler directive. If 'loopend' occurs two packets, or more, after the skipls target, then the Lpamrk prefix field in the packet two-packets ago is set to *1*, else it is set for the preceding packet. Note that the VlesSize prefix field counts the number of packet instructions.

For Example:

```
define(instrfield=VlesSize)
{
  size   = 3 ;
  prefix = true;
  pseudo = true;
}

define(instrfield=Lpmark)
{
  size   = 1;
  prefix = true;
  pseudo = true;
}
```

```
define(assembler) {
   queue_size = 3;
   define(parm=L1) {
     type = label;
   }
   define(parm=PktCnt) {
     type = integer;
     default = 0;
   }
}

define(instr=skipls)
{
   // skipls is a COF instruction with special behavior
   fields = (opcd(0x1346),BO);   // Its assembler syntax is
   syntax = ("%i %f",BO);         // skipls    Label1
   action = {
    .....
   };
   assembler = {
     L1 = BO;   // store the instruction's target label in a global variable
   };
}

post_packet_asm = {
     ...
  if (L1 != 0 && L1 == '*') { // Is the current packet is the skipls target.
    L1 = 0;        // reset L1 parameter
    PktCnt = 1;    // start the packet counter
  }
  else if (PktCnt > 0) {
    ++PktCnt;
  }
};

post_asm = {
  VlesSize[0] += 1;
};

define(instr=Loopend) {  // assembler directive
  syntax = ("loopend");

  assembler = {
    if (PktCnt == 1) { // skipls target was current or previous packet
      L1 = 0;
      Lpmark[1] = 1;
    } else {
      Lpmark[2] = 1;
    }
  }
  PktCnt = 0;
};
```

5.10 Assembler Programming Rules and Post-Packet Assembler Hook

The validity of an assembler instruction can be specified in several contexts. The legal values that an instruction field can assume may be restricted using the the *valid_ranges* and *valid_masks* ADL keys. Field values combinations, in an instruction, can be restricted using instruction assembler rules. Restrictions on combination of instructions in a VLES, e.g., to restrict access to shared resources, can be enforced using the packet assembler rules.

Programming rules are defined in the assembler section of the ADL in the *rule* define block. Instruction programming rules have an *InstrInfo* arguments, while packet rules have a single *InstrBundle* argument. The instruction key *asm_rules* specifies the rules to be applied to an instruction.

Instruction rule action code can access the instruction's fields by their name, and use the methods *size*, *instrHasAttr*, *instrBlk* and helper functions *info* and *error*.

InstrInfo API:

- *const char * instrName()*: Returns an instruction's ADL name.
- *bool instrBlk(<InstrBlocks>)*: Returns true if the instruction has the specified instruction-block type. The argument is an enum of the valid block types known by the architecture.
- *bool numInstrBlks()*: Return the number of blocks that this instruction has.
- *bool instrHasAttr(<InstrAttr>)*: Returns true if the instruction has the specified attribute. The argument is an enum of attributes known by the architecture.
- *bool instrHasField(<field-name>)*: Returns true if the instruction has the specified field. The argument must be a constant, e.g. `b[i].instrHasField(RT)`.
- *int instrOrder()*: Return the instruction's order. This is initially set to the index within the bundle, and can be used to preserve information about the original ordering, even after instructions are re-ordered.
- *int setInstrOrder(int)*: Set an instruction's order. The *InstrInfo* copy-constructor and *replaceInstr* propagate the order, but it is necessary to set the order explicitly if *createInstr* is used.
- *unsigned pos()*: Returns the relative position of this instruction in the packet.
- *unsigned size()*: Returns the size of the instruction (in bytes).
- *unsigned width()*: Returns the width of the instruction (in bits).
- *bool has_prefix()*: Returns true if the bundle has a prefix instruction.
- *string getReloc(unsigned index)*: Return a relocation abbreviation for the specified operand, if one exists, otherwise returns an empty string.
- *bool replaceReloc(unsigned index,const string &new_reloc)*: Replace a relocation abbreviation in the specified operand with the new relocation. Returns true if the replacement occurred, false otherwise.

Additional API functions (not methods of InstrInfo):

- *InstrBundle &getPriorPacket(unsigned index)*: Retrieve another packet. This is only useful for when a queue size of more than one is being used, and a non-zero queue-offset value is being used. The index parameter specifies the element in front of the current packet being processed by the post-packet handler. For example, for a queue size of 2 and a queue-offset of 1, the post-packet handler operates on the second-most-recent packet, so **getPriorPacket(1)** returns the most recent packet.

- *bool hasLabel()*: Returns true if the post-packet handler is running with a packet that is associated with a label.
- *void adjustCurLabel(int offset)*: This allows the post-packet handler to modify the value of the current label. This type of modification is necessary if a non-zero queue-offset value is being used, since this affects allocation of the underlying instructions.
- *InstrInfo createInstr(const std::string &new_name, ...)*: Creates a new instruction specified by `name`. Issues an error if the specified instruction does not exist. All arguments after the name are used to fill in operand values. For example: `createInstr("add",1,2,3)`.
- *InstrInfo replaceInstr(const InstrInfo &instr,const std::string &new_name)*: Given an instruction, this replaces the instruction with the instruction specified by `new_name`, keeping all operand values from the prior instruction. This may be used to replace one instruction with another, such as to choose an alternative encoding.

For example:

```
define(instrfield=OPCD) {
  bits = (0,12);
}
define(instrfield=RA) {
  bits = (13,17);
}
define(instrfield=RB) {
  bits = (18,23);
}
define(instrfield=RT) {
  bits = (24,29);
}

define(instr=Foo) {
  fields = (OPCD(1),RA,RB,RT);
  asm_rules = (R1,R2);
}

define (assembler) {
   define (rule=R1) {
     action = func (InstrInfo ii) {
         if (ii.instrHasAttr(orderd) && RA > RB) {
           error(1,"R1 violation: unordered fields.");
       }
     };
   }

   define (rule=R2) {
      action = func(InstrInfo ii) {
         if (ii.instrBlk(dalu) && RT == RA) {
             error(1,"R2 violation: equal source/target in dalu instruction.");
         }
      };
   }
};
```

- *void savePrefix(InstrBundle &)*: This function will determine whether a prefix instruction is required for the bundle, in order to store all of the instructions' operand bits. This is normally called by default, for parallel architectures, after `post_packet_asm`, but may be called by the user if further actions need to be taken after the prefix is generated. In that case, the default version will not be called.

  Note that after the prefix is generated, `InstrBundle::has_prefix()` will return true if a prefix exists.
- *InstrInfo combineInstr(const string &name, ...)*: Create an instruction with the `name` by filling its instruction-type fields with the encodings of other instructions. It is required that the fields of the instruction created be all instruction-type. The total number of such fields should be equal to that of arguments excluding `name`. The arguments after `name` correspond to the fields of the instruction created. The value of the argument represents the position of the instruction in the packet whose encoding is used to fill the corresponding field. You can, however, "skip" some fields by using "-1" as the argument. The "skipped" fields will be filled with zeros.

  For example:

```
define(instrfield=opA) {
  display = hex;
  pseudo = true;
  width = 15;
  type = instr;
  ref = instr_opA;
}

define(instrfield=opB) {
  display = hex;
  pseudo = true;
  width = 15;
  type = instr;
  ref = instr_opB;
}

define (instr = format){
  width = 64;
  fields = (
            (bits(55,52),b1010),
            (bits(49,35),opA), (bits(34,20),opB)
           );
  syntax = ("format %f, %f", opA, opB);
  attrs = (instr_macro);
  action={
          run(opA, opB);
         };
}

parallel_execution = 1;
define (assembler) {
  comments = ("//", "#");
  line_comments = ("//", "#");
  packet_grouping = ("\n");
  line_separators = ("\n");
  instrtables = (other, instr_opA, instr_opB);
}

post_packet_asm = func(InstrBundle b) {

  InstrBundle newb;

  if (b.size() == 1) {
      if (b[0].instrHasAttr(instr_opA)) {
          newb.push_back(combineInstr("format", 0, -1));
      }
      else if (b[0].instrHasAttr(instr_opB)) {
          newb.push_back(combineInstr("format", -1, 0));
      }
    }

  if (b.size() == 2) {
      if (b[0].instrHasAttr(instr_opA) && b[1].instrHasAttr(instr_opB)) {
          newb.push_back(combineInstr("format", 0, 1));
      }
  }

  b = newb;
};
```

- *InstrInfo patchInstr(InstrInfo& instr, InstrInfo& patch, int position)*: Patches an instruction at one of its instruction-type fields with the encodings of another instruction. The first two arguments correspond to the patched and patch instructions. The third argument represents the position of the LSB of the instruction-type field being patched. The patched bits are limited to the 64 least significant ones of the patched instruction. It is also required that (1) the patched instruction-type field be continuous; (2) the width of the patched instruction be greater than that of patch instruction; (3) the width of the patch instruction be less than or equal to 32 bits; (4) the patch instruction should not be patched by patchInstr() before.

  For example:

```
define (reg = a) {
  width = 10;
}

define(instrfield=Imme10){
  display = dec;
  pseudo = true;
  width = 10;
}

define(instrfield=opZ) {
  pseudo = true;
  width = 1;
  type = instr;
  ref = instr_opZ;
}

define (instr = foo){
  width = 64;
  fields = (
            (bits(55,48),b11001100),
            (bits(40,31),Imme10),
            (bits(20),opZ)
           );
  syntax = ("foo %f", Imme10);
  attrs = (instr_macro);
  action={
          a = Imme10;
          run(opZ);
         };
}

define (instr = Z1){
  width = 1;
  fields = (
           (bits(0),1)
           );
  syntax = ("z1");
  attrs = (instr_opZ);
  action={
          a = a + 1;
         };
}

define (instr = Z0){
  width = 1;
  fields = (
           (bits(0),0)
           );
  syntax = ("z0");
  attrs = (instr_opZ);
  action={
          a = a - 1;
         };
}

parallel_execution = 1;
define (assembler) {
  comments = ("//", "#");
  line_comments = ("//", "#");
  packet_grouping = ("\n");
  line_separators = ("\n");
  instrtables = (other, instr_opZ);
}

post_packet_asm = func(InstrBundle b) {

  InstrBundle newb;

  if (b.size() == 2) {
      if (b[0].instrHasAttr(instr_macro) && b[1].instrHasAttr(instr_opZ)) {
          newb.push_back(patchInstr(b[0], b[1], 20));
      }
  }

  b = newb;
}
```

Instruction programming rules are invoked after the post_asm hook of relevant instructions.

Packet programming rules use two predefined data structures: *InstrBundle* and *RegWrites*.

InstrBundle is the packet rule and post-packet handler argument. Its elements are the instructions assembled in the current packet (*IntrInfo* objects). The *InstrBundle* has a "vector"-like interface, i.e., it has a *size* method, and its elements are accessed using the standard subscripting '[]'. Rule methods are invoked for bundle elements using dot notation. You may create a new *InstrBundle* and then assign to the function argument in order to re-order and replace the packet. Since the packet also contains prefix information, it is recommended to use a copy-constructor to create a new bundle from the original, then use element assignments to update the new bundle. For example:

```
post_packet_asm = func(InstrBundle b) {
  InstrBundle newb = b;
  newb[0] = b[1];
  newb[1] = b[0];
  b = newb;
};
```

InstrBundle API:

- *unsigned size() const*: Size of the bundle (number of instructions), not including the prefix.
- *unsigned group_size()*: Size of the bundle, including the prefix if one exists.
- *bool empty() const*: True if the bundle is empty.
- *InstrInfo &get_prefix()*: Return the instruction info object for the prefix.
- *bool has_prefix() const*: True if the bundle has a prefix.
- *unsigned prefix_index() const*: Returns the prefix placement index. This is 0 if the prefix should be placed at the start of the packet (default).
- *void set_prefix_index(unsigned)*: Set the prefix placement index.
- *set_prefix_fields()*: Set prefix fields with relevant bits from each instruction's operands.

This class publicly inherits from `std::vector`, so the usual vector interface is accessible, such as assignment, copy-constructor, `empty()`, `size()`, etc.

You may create a new *InstrBundle* and then assign to the function argument in order to re-order and replace the packet. Since the packet also contains prefix information, it is recommended to use a copy-constructor to create a new bundle from the original, then use element assignments to update the new bundle.

The *RegWrites* data structure stores information on instructions' target registers and their write masks. It has a "vector"-like interface and its elements implement the functions *reg()* - a target register/registerfile, and *mask(0)* and *mask(1)* two write masks associated with the registers. *RegWrites* elements are populated using the helper *get_dest()*, which sets *mask(0)* to the write mask and *mask(1)* to zero.

intersect_dest() method populates its last argument with registers common to its two first arguments, i.e., those written by both instructions. A register-file will be added to the result only if both instructions write the same file elements. For this type of object *mask(0)* and *mask(1)* correspond to the write masks of the first and second instruction, respectively. Mask value can be tested against a register field using the standard dot notation. For example:

```
define (rule=R3) {
  action = func(InstrBundle b) {
          int i;
          int foos = 0;
          int bars = 0;
    for(i=0; i < b.size(); ++i) {
             if (b[i].hasInstrAttr(foo)) {
         ++foos;
       }
             if (b[i].instrBlk(bar)) {
               ++bars;
       }
    }
    if (foos > 2 || bars > 3) {
            error(1,"Too many foos/bars.");
    }
  };
}

define (rule=R4) {
  action = func(InstrBundle b) {
          int i;
          RegWrite rw1;
          RegWrite rw2;
    RegWrite rw3;

          get_dest(rw1);
    det_dest(rw2);
    intersect_dest(rw3);

          for (i=0; i < rw3.size(); ++i) {
            if (rw1[i].reg() != SR || rw1[i].mask() != SR.T) {
       error(1,"Mutual exclusion error.");
      }
    }
  };
}
```

Packet rule are not associated with any instruction and are called after the post_packet_asm hook.

5.11 Variable Width Instructions

ADL fully supports variable-width encodings. Depending upon the architecture, however, different ways of encodings instructions may be more convenient than others. In the simplest case, e.g. a RISC architecture with variable-width instructions, the procedure is quite straight-forward: Simply define fields for the appropriate bits and include them within an instruction. The instruction's width is then deduced from its fields.

For example:

```
define (instrfield=OPCD) {
  bits = (0,7);
}

define (instrfield=X) {
  bits = (8,11);
}

define (instrfield=Y) {
  bits = (11,15);
}

define (instrfield=D) {
  bits = (16,32);
}

// A 16-bit width instruction.
define (instr=Foo) {
  fields = (OPCD(1),X,Y);
}

// A 32-bit width instruction.
define (instr=Foo) {
  fields = (OPCD(1),X,Y,D);
}
```

In other cases, an architecture may have a series of addressing modes for a series of different operations, where the addressing modes may be of different widths. In addition, little-endian bit-numbering may be used, in which case assigning fixed-bit positions to fields becomes tedious, since it depends upon the width of the instruction. In this situation, pseudo fields and sub-instructions may be very useful.

The following example contains a memory-to-memory move operation in which two addressing modes are supported, a 16-bit address and a 24-bit address, for a total of four individual instructions:

```
define (instrfield=F) {
  pseudo = true;
  width = 4;
}

define (instrfield=X1) {
  pseudo = true;
  width = 16;
}

define (instrfield=X2) {
  pseudo = true;
  width = 24;
}

define (subinstr=a1) {
  fields = (F(0),X1);
  action = func (bits<32> &addr) {
    addr = X1;
  };
}
```

```
define (subinstr=a2) {
  fields = (F(1),X2);
  action = func (bits<32> &addr) {
    addr = X2;
  };
}

define (subinstr=b1) {
  fields = (F(0),X1);
  action = func (bits<32> &addr) {
    addr = X1;
  };
}

define (subinstr=b2) {
  fields = (F(1),X2);
  action = func (bits<32> &addr) {
    addr = X2;
  };
}

define (instr=mov) {
  fields = (OPCD(0x1e),A,B);

  define (subinstrs=A) { subs = (a1,a2); }
  define (subinstrs=B) { subs = (b1,b2); }

  action = {
    bits<32> addr1, addr2;
    A(addr1);
    B(addr2);

    Mem(addr1,4) = Mem(addr2,4);
  };
}
```

In this situation, all instruction-fields are marked as `pseudo`, indicating that ADL should assign bits to each instruction field, based upon the field order in the instruction definition. Note that the same fields are used in both sub-instructions; this is allowed, and each field will be assigned its own bit range. The result is that four instructions are created, named `mov_a1_b1`, `mov_a1_b2`, `mov_a2_b1`, `mov_a2_b2`. If desired, the `names` key may be used in the instruction in order to assign user-specified names to each instruction.

6 Generators

The basic design for an ADL application will consist of a front-end library and a back-end which will generate a tool or model. The front-end will do all of the parsing and most of the semantic analysis, such as checking that needed fields exist, that an instruction's action code references valid resources, etc. It will then produce a data structure of all of the various resources and helper C++ code and the back-end will then have access to this data structure in order to perform its activities. A trivial back-end will exist that will dump this data structure to XML.

The front-end will be implemented in garbage-collected C++. A back-end may be implemented in any language: If it is C++, then it may directly use the front-end. If it is implemented in another language, it can use the XML dump feature. A future version of the ADL front-end might provide a SWIG interface so that other languages can directly use the library.

6.1 Assembler/Disassembler Generation

The ADL program *adl2asm* generates assembler and disassembler files, compiles them, and links them against a special distribution of *GNU binutils*. This special distribution is stripped down to just handle an ELF-based target, at present, and adds some auxiliary files which are generic to all ADL generated assemblers.

For example, to create a custom *gas* and *objdump*, the user would do the following:

```
./adl2asm model.adl
```

This would generate the assembler, named `as-model`, and the disassembler, named `objdump-model`.

The program will also generate an addr2line program and a basic linker. The linker is fairly simple, but is able to handle relocations defined within an ADL description. It is currently based around a stripped-down PowerPC linker.

The command-line option `--dbg` instructs the generator to also generate data useful by a debugger, such as a disassembler which disassembles to a structure, and lists out registers associated with the attribute `debug`. This data is generated in the disassembler file and a header file is created called `instr-info-<model>.h`. The `debug` attribute is also used by the ISS generator in order to create a default register map for use by the debugger. This attribute may be given an integer value, which is then used to order the registers and register-files in the register map.

6.1.1 Multiple Instruction Tables

The generated assembler and disassembler can handle an architecture which contains multiple multiple instruction tables. When a change in table occurs, the assembler inserts mapping symbols; the disassembler looks for these symbols in order to select the correct disassembly table.

For the assembler, a different instruction table can be specified using either the command-line option of -m<name>, where <name> is the name of the instruction table, or a pseudo-op of the form:

```
.switch -m<name>
```

The default instruction table, designated using the attribute `other` can be selected using the name `default`.

For example, given an architecture containing two instruction tables, the default and one named "vle", the following code fragment is an example of a mixed-instruction file:

```
loop:
    add     r2,r2,r1
    subic.  r3,r3,1
    bne     loop
    mtspr   20,r4

   # Switch to using the vle instruction table.
   .switch -mvle
    se_add  rr9,rr8

loop2:
    se_add  rr9,rr8
    subf.   r26,r27,r26
    se_bne  loop2
        se_or   0,0

   # Switch back to using the default instruction table.
       .switch -mdefault

       add            r3,r2,r1
       add            r6,r5,r4
```

6.1.2 Debug Prints

Assembler can be run with the option "-d" which will turn on debug warnings about undefined symbols.

6.2 ISS Generation

The ADL project can currently generate an untimed ISS or a time-tagged ISS. The time-tagged ISS is a cycle-approximate model, where each instruction may have a fixed latency, but where we model delays due to operand availability. Refer to Time Tagged ISSs for more information.

The ISS generator is named *make-iss*. It should generally be invoked using the wrapper script *adl2iss* which will generate the ISS source files and optionally compile this into an executable model. Both programs take the following arguments. Flags may be negated by preceding the option with *no*, e.g. *--no-line-directives* turns off the insertion of line directives.

6.2.1 Usage

Usage:

```
adl2iss [options] <model file>
```

Options:

-help, --h:
Display help

--man, -m:
Display the complete help as a man page.

--target=[exe|*so*|*base-so]*:
Specify the target type. The default is `exe` which means that a standalone executable will be produced. If the `so` option is selected, a shared object will be generated which includes the standalone-ISS infrastructure. The `base-so` option produces a library containing only the basic model without the extra infrastructure.

--output=file, -o=file:
Specify the output file name. If not specified, the base name of the input file will be used, with the appropriate extension of the desired output.

--top-level=s, -tl=s:
Specify the top-level element for which to generate the ISS.

--rnumber:
Generate code with RNumber support. RNumber is a dynamic arbitrary-sized integer class and is used to provide support for setting and getting register values of arbitrarily-large size. Without this, only values of 64-bits in width or smaller may be accessed. This is a negatable option. The default is TRUE.

--trace-cache:

Generate a model with a trace-cache for simulation speed improvements. This is a negatable option. The default is FALSE.

Note: By default, a trace-cache model does not support the --max-instr-count option. To enable this, use the --td-api option, which is on by default for trace-cache models.

--graphite:
Generate a simulator which allows a performance model to be installed. The performance model is based upon the MIT Graphite core performance models.

--jit:
Generate JIT (just-in-time) compilation support code. This is a negatable option. The default is FALSE.

--disassembler:
Generate a disassembler function for the model. Default is FALSE.

--lib-mode:
Generate a library-mode ISS. In library mode, the ISS is generated as a template, where the template parameter is used as a base class. The base class must implement the storage of register data and must implement read and write routines for all registers. In addition, the normal simulation logic, such

as the decode tree, is not generated. Default is FALSE. This option may be negated.

--dmi-cache:
Toggle the use of an internal software cache for improving performance. This is on by default if either the JIT or the decode-cache are being generated. This option may be negated.

--event-bus-send:
If set, then event buses within models will broadcast their data to all other registered cores in the system. If negated, then the event bus data is instead broadcast to a registered functor object. Generally, this option will only be turned off when the user wishes to integrate a model into a platform model, or co-simulation environment, where the sending and receiving of event bus data must be handled by this external environment. The default is TRUE.

--extern-event-bus:
If set, then event buses within the model broadcast via an installed handler. This is similar to **--no-event-bus-send**, but differs slightly: In this case, the event bus data is all opaque and can only be copied or transferred and not examined. However, this system is completely core model agnostic: There is no need to include the model header and derive from it in order to understand the underlying data structures. Use this method, for example, when all that is necessary is to broadcast event bus information in an alternative manner, such as via an IPC mechanism and no examination of the data is required.

--extern-mem, -em:
Generate a model with the global memory defined externally. This is a negatable option. The default is FALSE. This information may also be specified within the configuration file, which is the only way to generate externally-defined local memories,

--extern-dmi:
Toggle support for an external direct-memory-interface cache. This is only valid if *--extern-mem* is also set.

--extern-dmi-entry-size:
Specify the entry size of the external dmi cache in bits, e.g. 7 = 128 byte entries.

--extern-dmi-size:
Specify the number of entries in the external dmi cache.

--extern-syscalls:
Enable the use of external system-call emulation. The user of a model must then install an object of type `SysCallHandler` into each core by calling `IssNode::setSysCallHandler`.

--incl-unused-helpers:
Specify whether unused helper functions should be included in the generated model. The default is **false**: Unused helpers are not included. This option may be negated.

--tlm2-endianness:
Specify that TLM 2.0 compatible address swizzling should be used for handling endianness conversions, rather than byte-swapping. The default is **false**.

--trace-mode, -t:
: Generate code for tracing (producing intermediate results). This is a negatable option. The default is TRUE.

--time-tagged, -tm:
: Generate a time-tagged ISS. This is a negatable option. The default is FALSE. Refer to Time Tagged ISSs for more information.

--debug-mode, -dm:
: Generate a model with debug support. This is a negatable option. The default is TRUE.

--print-data, -pd:
: Display ADL data to standard-out, do not generate an ISS.

--dep-file=file, -df=file:
: Instruct the preprocessor to generate a dependency file suit- able for inclusion by a Makefile. This is done as a side- effect and does not affect the compilation process.

--gen-only, -go:
: Only generate the ISS source code, producing a C++ source file. Do not compile or link.

--compile-only, -co:
: Generate the ISS source code and compile it, producing an object file.

--static:
: Link all dependent libraries statically, including the compiler run-time. Only standard system libraries are dynamically linked. This creates a model which is as portable as possible.

--optimize=[level]:
: Compile the model with optimization. The default optimization level is 3, corresponding to compiling with -O3. Another level may be specified. A value of 0 turns off optimization (equiva- lent to using the --no-optimize option).

--pgo-1:
: Enable the first pass of profile-guided optimization. This generates a model and compiles it such that profile-guided optimization files (.gcda files for gcc) will be created.

--pgo-1:

Enable the second pass of profile-guided optimization. This does not generate a model, but instead re-uses an existing model. It recompiles it using branch-prediction results obtained from profile-guided optimization (.gcda files) generated during the first pass.

Note: This does not re-generate the model, so do not edit the source files between the first and second pass. Be sure to not change any other compilation options, such as optimization level, since the PGO data is only relevant for the same level of optimization.

--config-file=file, -cf=file:
Specify a configuration file for model generation.

--preamble=str:
Add a preamble string to the model. This data is displayed by the model at startup time and is also included in the comment block at the top of the generated C++ code.

--iss-namespace=str:

Specify an alternate namespace name which will wrap the ISS. The default is `adliss`. This allows you to generate multiple simulators and then include them into a single application.

--mflags=str:
Specify flags to be given to the model generator.

--cflags=str:
Specify flags to be given to the compiler. This option may be repeated.

--ldflags=str:
Specify flags to be given to the linker. This option may be repeated.

--no-optimize:
Turn off compiler optimization.

--low-mem:
Enable code generation that requires less memory to compile.

--jobs=n, -j=n:
Specify the number of jobs into which to break the ISS compilation. A value of 1 is valid- this means that a single implementation file will be created, in addition to the class declaration file. A value of 0 (the default), means that only a single file, containing the class and all implementation, will be created.

--mt-support:
Enable multi-threaded simulation support. This turns on thread-safe memory if using the default, internal global memory, and adds necessary support to ensure thread safety with event buses, etc.

--mt-rw-locked-mem:
If compiling a model with **--mt-support**, this causes all memory accesses to invoke a per-core lock. This prevents contention between external accesses from another thread, e.g. via **setMem**, and an internal access via a load or store instruction. The default is true.

--mt-locked-mem:
If compiling a model with **--mt-support**, this causes all memory accesses to invoke a lock. Normally, locking is only done for atomic instructions.

--parallel-build, -p:
Run the compile jobs in parallel. This is the default. Negate this feature to compile all items sequentially.

--transactional-iss:
Create a transactional ISS.

--hybrid-iss:

Create a hybrid-mode ISS.

--safe-mode-trans-iss:

Create a safe-mode transactional ISS.

--track-reads:

For hybrid or transactional ISSs, track operand reads. If on, the ISS will track partial reads by setting bits in a read mask. The default is off.

--log-reg-reads:

Log register read operations. The default is off.

--log-reg-masks:

When partial register read/write operations occur, log the mask describing what part of the register was accessed. The default is off.

--log-unarch:

If true, then registers, exceptions, and other resources marked with the *unarchitected* attribute are logged. If false, then they are not logged. Note that even if this is true, the UVP writer will not log them. Toggling this flag off is mainly useful for ADL models which will be integrated into another model or application, so that the integrator does not need to filter logging themselves. The default is true.

--log-td-commit:

Log instruction completion times using the temporal decoupling counter value.

--log-usage[=prog:ver]:

Turn usage logging on or off. The user may supply an optional program-name and version string to be logged. If omitted, then the input-file root will be used as the program-name and the version will default to <year>.<month>.<day>.

To disable the option, use --log-usage=false or --log-usage=no. You may also use the negated form of --no-log-usage.

--dep-tracking:

Enable dependency tracking. This means that the ISS will log memory and register reads and use information from an instruction's `dependencies` block to correlate register and memory writes with these reads.

The **--log-reg-reads** option must also be enabled for this feature to work. This option may be negated. The default is false.

--instrs_info:

Add to transactional-iss code two types of information.Instrcution attributes information, and data that maps fields to instruction packet slots.

--cleanup:

Remove temporary, intermediate files. This is on by default, but may be negated to keep these files around.

--verbose:
  Show the output of all internally executed commands.

--rollback-mode:
  Toggle support for restoring state after an instruction is executed. Only relevant for normal or hybrid-ISS modes. Default is FALSE.

--ignore-mode:
  Toggle support for ignore-mode, which allows the model to toggle whether to ignore register updates and stores. Default is FALSE.

--strict:

  Turns on strict mode for parser. The parser will warn about possible encoding conflicts and will report objects that were modified using *defmod*.

--syscall-enabled:
  Enable system-call support. This is a negatable option. The default is TRUE. When using a standalone model, use the **--syscall-enabled** command-line option to enable system call support at simulation time.

--td-api:

  Add support for temporal decoupling, which allows a model to execute extra instructions within a platform model as long as they are performed within a certain time quantum. If enabled, the user may set an increment and threshold value. An internal counter is incremented by the increment value for each instruction executed. When this exceeds the threshold, then simulation stops and control returns to the user. This feature is also used to implement the --max-instr-count option when running using --trace-cache or --jit.

  This option may be negated. The default is TRUE for decode-cache or JIT models, FALSE otherwise.

--per-instr-td-increment:
  Enable per-instruction temporal-decoupling counter increments for decode-cache and JIT models. False means that increments are grouped at the end of a basic block. If true, then fidelity is higher at a cost of some performance. The default is true.

--tags-only-caches:
  If the ADL description contains caches, the generated model will only track cache tag information and not any actual cache data. This makes it simple to model a multi-core system, since no coherency protocol is required, but all caches will act in a write-through manner, thus not allowing for the modeling of incoherent data.

--define=str, -D=str
  Specify a preprocessor define.

--include=path, -I=path
  Specify a preprocessor include directory.

--cpp-cmd=str

Specify an additional include or define directive to add to the generated source code. Normally, the preprocessor is run before the ADL parser, but there may be reasons, such as including a standard library header, where only the compiler should process the directive and not the ADL parser.

The format is type:option, where type can be 'include' for an #include directive or 'define' for a #define directive. For example: *--cpp-cmd=include:"foo.h"* will result in **#include "foo.h"** appearing in the resulting C++ code. Note that include directives require the quotes, since includes can either use angle brackets or double quotes.

6.2.2 Configuration File Syntax

The configuration file uses a format similar to the ADL model format. It is used to specify information related to the generation of the ISS, such as instruction latency information for time-tagged ISSs.

The configuration file uses *define* blocks to specify a hierarchy of cores and systems. This hierarchy must match the hierarchy of the model. However, nodes may be missing. In such a case, the default behavior is assumed. For example, suppose a model contains the following hierarchy:

```
define (sys=FooSystem) {
  define (sys=FoChip) {
    define (core = Foo) {
      archs = power;
    }
    Foo core0;
    Foo core1;
  }
  FooChip chip0;
  FooChip chip1;
}
```

The configuration file could configure the *FooSystem::FooChip::Foo* core by using the same hierarchy:

```
define (sys=FooSystem) {
  define (sys=FooChip) {
    define (core = Foo) {
      define (group=foo) {
        items = mullw;
        latency = 8;
      }
    }
  }
}
```

Note that only core declarations may be configured, not definitions. In other words, the user may not specify different timing information for *core0* and *core1*. This is due to the fact that the timing information is hard-coded into the code which describes each core.

Configuration file *define* types and valid keys:

- Definition: *sys* = <*name*>: Specify a system in the hierarchy. This groups may contain other *sys* or *core* defines.
    - Definition: *mem*=<*name*>: Specify a local memory in the system. * *extern_mem* = <*bool*>: Specify whether the memory is internal or external.

- Definition: *core* = <*name*>: Specify a core in the hierarchy.

    - Definition: *group*: Assigns timing behavior to a group of instructions or instruction classes, for use by the time-tagged ISS generator.

        - *allow_issue = func(unsigned index) {}*: Add logic for deciding, in a multi-issue machine, whether an instruction may be issued. The index parameter specifies the issue index, e.g. 0 for the first instruction to be issued in a cycle, 1 for the second instruction, etc. Typically, such a function, in a dual-issue machine, will allow any instruction with an index of 0 to issue, but then limit the second instruction, e.g. not allowing two multiplies to issue in the same cycle.

          This instruction may call a special version of the **instrHasAttr** to query instruction attributes:

          ```
          bool instrHasAttr(unsigned index,unsigned attr);
          ```

          This allows the function to query if an instruction which has previously issued issued had one or more of the specified attributes.

          The *allow_issue* function should return true if an instruction may issue, false if it may not. Refer to Time Tagged ISSs for more information about how to use this function predicate.
        - *items* = <*ident*> | <*list*>: Specifies the members of the group. These items may be either instructions or instruction classes. If *defmod* is specified, the *items* list is concatenated with the existing list. The special instruction **default_instruction** may be used to specify that any instruction not handled by other groups should be included in this group. Note that all instructions must be handled by a group, or else the generator will report an error.
        - *latency* = <*int*>: The latency assigned to these instructions. The default is 1.

- *stages = <list(ident|ident=ident)>*: List pipeline stages associated with this group of instructions. This consists of a list of identifiers (the stages). A pipeline may have a tag associated with it. The valid tags are:

    - *compl*: The stage is considered the completion stage, for instruction logging purposes. If this tag does not exist, then the last stage listed will be assumed to be the completion stage.
    - *issue*: The stage is considered the issue stage, for instruction logging purposes. If this tag does not exist, then the first stage listed will be assumed to be the issue stage.
    - *exec*: The stage is considered the execute stage, for the purposes of computing instruction latency. Any extra latency, such as that caused by a cache miss, will be added in this stage. If this tag does not exist, then any extra latency will be ignored.

  For example, a simple five stage pipeline might be described as:

  ```
  stages = (Fetch,Decode = issue,Exec,Mem = exec,Writeback = compl);
  ```

o Definition: *config*: Configure the core.

- *compl_time = func(uint64_t) {}*: This hook, if present, is called to compute a completion time for a function, when the simulator is built with **log_td_commit**. Otherwise, the current value of the temporal decoupling counter is used as the completion time.
- *trans_const_parms = <ident|list(ident)>*: List architectural parameters which can be considered to be constant, at translation time, by the JIT (if a JIT model is being created). For example, a 32-bit/64-bit mode parameter can be considered to be a constant, since any given sequence of code will either be 32-bit or 64-bit, but not either. On the other hand, a parameter which is set based upon MMU translation attributes cannot be considered to be a constant, since its value will depend upon the translation used by a load or store instruction, and thus depend upon the address, which will depend upon current register values.

  By specifying parameters as constant, the JIT is able to fold out conditional code which depends upon the parameter, thus improving performance of a JIT model.
- *dmi_cache = <bool>*: Specify whether a direct-memory-cache (DMI cache) should be generated for this core. This is an internal cache which maps effective addresses (pre-translation) to host memory blocks, in order to increase performance. It is turned on, by default, if a core has trace-cache or JIT support and has a cache or MMU.
- *external_dmi_write_update = <bool>*: Disable updates from the external DMI to the internal DMI on writes. It is necessary to disable

this when a model has two levels of cache and the L1 is write-through only, since this means that data exists in more than one location, which is not compatible with the concept of the DMI cache. The default is true (updates are performed).

- *bb_end_instrs* = <*list(ident)*>: Specify instructions other than branches which are considered to end a basic block. This is necessary for the decode cache, since the decode cache assumes that only branches end a basic block, and thus cause any kind of a change in control. Other types of instructions, e.g. a move to a special register which can change the program counter, must be specified explicitly in order for the decode-cache model to work correctly.
- *opt_bb_end_instrs* = <*list(ident)*>: Specify instructions which can optionally end a basic block via the expiration of the temporal decoupling counter. This allows an instruction other than a basic block to exit the simulation loop of the decode cache.
- *issue_width* = <*int*>: Specify the number of instructions that may be issued during a single cycle, for a time-tagged ISSs.
- *generate_mmu* = <*bool*>: Specify whether the MMU should be generated, assuming one is defined for the core. The default is *true*.
- *generate_caches* = <*bool*>: Specify whether the caches should be generated, assuming there are caches for this model. The default is *true*.
- *enable_syscall* = <*bool*>: Specify whether system-call support is enabled, The default is *true*.
- *extern_syscalls* = <*bool*>: Enable/disable external system call support.
- *log_original_mem_access* = <*bool*>: If true, then store information about the original, top-level memory access. For example, a misaligned, 128-bit access will store the misaligned address and a size of 16 (the value is stored in bytes). These values are then accessible using the core's methods `addr_t get_orig_access_addr()` and `unsigned get_orig_access_size()`.
- *tracing_on* = *func() { ... }*: This hook is called when tracing is turned on.
- *tracing_off* = *func() { ... }*: This hook is called when tracing is turned off.
- *post_sim* = *func() { ... }*: This hook is called at the end of a simulation run, either due to program exit or the expiration of a time quantum.
- *clock_cycle_ratio* = <*float*>: Specify a clock cycle ratio for triggering the `pre_cycle` and `post_cycle` hooks. This can be a decimal number in order to represent a non-integer ratio. A decimal ratio works in the following manner: A tally is kept, which is incremented by the ratio value for each simulation loop. When the tally exceeds 1, the hooks are executed n times, where n is the integer portion of the tally. The integer portion of the tally is then cleared, keeping the fractional portion, and the process is repeated.

If 0 (the default), then the hooks are executed once per simulation loop.

- *reg_callbacks* = <*ident*|*list(ident)*>: List registers which should have callback support. The user may then call `adl::IssNode::setRegCallback` on these registers in order to install a callback. A callback is a functor derived from `adl::RegCallback`. When the register's value is updated, the new value is reported to a registered callback. Only one callback may be registered at a time. The default call-back action is to do nothing.
- *reg_read_callbacks* = <*ident*|*list(ident)*>: List registers which should have read callback support. The user may then call `adl::IssNode::setRegReadCallback` on these registers in order to install a read callback. A read callback is a functor derived from `adl::RegReadCallback`. When the register is read, the callback is called first, in order to obtain a new value for the register.
- *exception_callbacks* = <*ident*|*list(ident)*>: List exceptions which should have call-back support. The user may then call `adl::IssNode::setExceptCallback` on these exceptions in order to install a callback. A callback is a functor derived from `adl::ExecptCallback`. When the exception is raised, the callback is called after the exception's action code is executed. Only one callback may be registered at a time. The default call-back action is to do nothing.
- *dyn_caches* = <*ident*|*list(ident)*>: List caches whose parameters may be set dynamically at run-time. This may be done either through the scripting interface (for standalone executables) or via the C++ API for platform models. Currently, the number of sets, number of ways, and line-size may be set when a cache is listed here.
- *dyn_tlbs* = <*ident*|*list(ident)*>: List MMU lookups (TLBs) whose parameters may be set dynamically at run-time. This may be done either through the scripting interface (for standalone executables) or via the C++ API for platform models. Currently, the number of sets and number of ways may be set when a TLB is listed here. However, if a TLB is fully-associative, then only the number of ways may be set; it cannot be transformed into a fixed-size TLB.
- cache_miss_latency = <list(ident=(int|func))>*: List miss-penalty latencies for caches in the core. This takes the form of a list of one or more elements of the form **cache-name** = **latency**. For example:

  ```
  cache_miss_latency = (L1d=10,L2=50);
  ```

  This would associate a miss penalty of 10 cycles for the L1d cache and 50 cycles for the L2 cache.

The latency may also be an arity-1 function which returns an integer, so that the result can be dynamically evaluated at run-time. For example:

```
cache_miss_latency = (L1d=10,L2=func(addr_t ra) { return
external_resource->getL2MissPenalty(); });
```

- *cache_line_size* = <*int*>: Specifies the size of a cache line, should we need to know this, but do not have caches defined in the model. For example, if the **--icache-touch** flag is explicitly set for a decode-cache model without caches, e.i. the caches are modeled externally, then we use this line size value for calculating when to insert touch operations.

  - Definition: *mem*=<*name*>: Specify a local memory in the core. * *extern_mem* = <*bool*>: Specify whether the memory is internal or external.

- Definition: *config*: Configure the global model generation process.

  - *debug_mode* = <*bool*>: Turn on or off debug-mode code generation in the model.
  - *dmi_cache* = <*bool*>: The global flag for specifying whether a direct-memory-cache (DMI cache) should be generated. If false, then support is disabled for all cores. Otherwise, the per-core flag is used.
  - *dissasembler* = <*bool*>: If true, generate disassembler functions for all cores.
  - *exec-set-size* = <*int*>: Fixed size of parallel execution set.
  - *extern_mem* = <*bool*>: Specify whether *Mem* is external or internal.
  - *extern_dmi* = <*bool*>: Create an external-memory DMI interface. Only applicable if *extern_mem* is set.
  - *extern_dmi_entry_size* = <*int*>: Specify the size of each entry in the external DMI cache in bits, e.g. 7 means an entry size of 128 bytes.
  - *extern_dmi_size* = <*int*>: Specify the number of entries in the external DMI cache.
  - *gen_full_regs* = <*list(str)*>: For each specified register or register file, generate all relevant access code in the model for each core which contains this register or register file. By default, only the code needed by the model is generated. So, for example, if a register is never written in the model, the standard write code is not generated.
  - *gen_all_reg_code* = <*bool*>: Generate all code for all registers and register-files in all cores. The default is false.
  - *iss_type* = <*normal*|*hybrid*|*transactional*>: Specify the type of ISS to be generated:

    normal
    : No tracking of operands, execution is immediate.

    hybrid
    : Operands are tracked in an instruction packet, execution is immediate.

transactional
Operands are tracked in an instruction packet, execution occurs within the packet, memory accesses are handled by the external model.

- *log_td_commit* = <*bool*>: Log instruction completion times using the temporal decoupling counter value.
- *mem_type* = <*blocking*|*nonblocking*|*logging*>: Specify the memory interface to be used. Only relevant for non-normal mode models.

  mNonBlocking
  Only relevant for transactional ISSs. All memory is handled by the external model via a coroutine interface.
  mBlocking
  Generally only used for hybrid models, though it could be used for a transactional ISS. A memory access is handled via a virtual function, no coroutines are used.
  mLogging
  Only relevant for hybrid models, e.g. used by uADL for iss-mem-mode. In this mode, memory is handled directly by the ISS, but the accesses are also logged so that the external model can track it.
- *icache_touch* = <*bool*>: Toggle insertion of instruction cache touch operations inserted for the decode cache and JIT simulators. The default is true if caches are present. If set to true and no caches are present, then loads are inserted and the size of the cache line is specified via the **cache_line_size** parameter in the appropriate core configuration block.
- *jit* = <*bool*>: Enable or disable generation of JIT (just-in-time compilation) support.
- *jit_ea_hash_size* = <*int*>: Specify the size of the effective-address fast-hash for the JIT. The size is specified in bits, i.e. the number of entries in the hash is 2^n. The default is 10 (1024 entries).
- *lib_mode* = <*bool*>: Turn on or off the generation of a library-mode ISS. In library mode, the ISS is generated as a template, where the template parameter is used as a base class. The base class must implement the storage of register data and must implement read and write routines for all registers. In addition, the normal simulation logic, such as the decode tree, is not generated. Default is false.
- *line_directives* = <*bool*>: Turn on or off the generation of line directives in the model.
- *max_delay_slots* = <*int*>: Maximum number of delay slots supported. The default is 3 (the current cycle plus 2 cycles ahead).
- *mt_support* = <*bool*>: Enable multi-threaded simulation support. This turns on thread-safe memory if using the default, internal global memory, and adds necessary support to ensure thread safety with event buses, etc.
- *mt_rw_locked_mem* = <*bool*>: If compiling a model with **--mt-support**, this causes all memory accesses to invoke a per-core lock. This prevents

contention between external accesses from another thread, e.g. via **setMem**, and an internal access via a load or store instruction. The default is true.

- *mt-locked-mem* = <*bool*>: If compiling a model with **--mt-support**, this causes all memory accesses to invoke a lock. Normally, locking is only done for atomic instructions.
- *packet_pool_size* = <*int*>: Maximum number of elements in the delayed-register-write pool objects used for delayed writes and parallel architectures. The default is 1000.
- *parallel-execution* = <*bool*>: Enable or disable parallel execution of instructions.
- *rnumber* = <*bool*>: Enable or disable generation of RNumber (dynamic arbitrary-width integer) support.
- *syscall_enabled* = <*bool*>: Enable or disable system call support for the cores in the system.
- *time_tagged* = <*bool*>: Turn on or off time-tagged code generation in the model.
- *trace_cache* = <*bool*>: Enable or disable generation of a trace-cache in the model, in order to improve simulation speed.
- *trace_mode* = <*bool*>: Turn on or off trace-mode code generation in the model.
- *track_reads* = <*bool*>: For transactional or hybrid ISSs, turn on or off the tracking of reads. If on, we track partial read masks. Default is off.
- *verbose_mode* = <*bool*>: Turn on or off verbose-mode during model generation.

6.2.3 Time Tagged ISSs

The standard ISS generator, **adl2iss**, may be used to generate a simulator with limited cycle-approximate knowledge. This is called a *time-tagged ISS*, as each register resource, and the memory subsystem, contains a tag, used to track when a resource becomes available. The model differs from typical cycle-approximate models in that it is still fundamentally an ISS: Simulator execution occurs at the level of instructions, rather than cycle-by-cycle with an actual pipeline model. Instead, each instruction updates the resources it touches (registers, pipeline stages, etc.) to specify when the resource will next be available. This allows for much higher simulation speed, at the cost of cycle-accuracy. For example, a divide which takes 8 cycles, executing at time 0, would update its target register to specify that it will next be available at time 8. If the next instruction uses that target, then it will execute at time 8, when the resource becomes available.

To create a time-tagged ISS, the user must specify a configuration file (.ttc file). This must contain a **core** configuration which then contains configuration group defines for each instruction specifying pipeline stages and any extra latency associated with various classes of instructions. All instructions must be covered by the groups (via the items key), but a group may use the special **default_instruction** identifier in its instruction list to specify that any instruction not already covered by a group should be included in this group. The model assumes appropriate forwarding paths exist; extra latency should be added if this is not the case.

The issue-width of the processor may be specified using the issue_width key in the core configuration group. Each instruction group may then specify instruction-issue restrictions via the allow_issue function. For example, the function may specify that loads and stores may only be issued individually.

The following is a simple example configuration file:

```
define (core=P) {

  define(group=loadstores) {
    items = (load,store);
    stages = (Fetch,Decode = issue,Exec,Mem = exec,Writeback = compl);
    // Allow this instruction to issue if we're in the first slot or if the
    // instruction in the first slot wasn't a load or store.
    allow_issue = func(unsigned index) {
      return (index == 0) || !((instrHasAttr(0,load) || instrHasAttr(0,store))
                               && (instrHasAttr(index,load) || instrHasAttr(index,store)));
    };
  }

  define(group=divides) {
    items = (divd,divdu);
    latency = 8;
    stages = (Fetch,Decode = issue,Exec,Mem = exec,Writeback = compl);
  }

  define(group=allinsts) {
    items = (default_instruction);
    stages = (Fetch,Decode = issue,Exec,Mem,Writeback = compl);
  }

  define (config) {
    cache_miss_latency = (L1d = 10,L1i = 10);
    issue_width = 2;
  }
}
```

In this example, we model a simple dual issue machine with a five-stage pipeline. We have three groups: One for loads and stores, one for divides, and one for everything else. For the *loadstores* group, the 10 cycle miss penalty of the data cache, *L1d*, will be calculated in the *Mem* stage, as that is identified as the execution stage, via the *exec* tag. The group also contains an *allow_issue* function to specify issue restrictions: A load or store may only issue if it is the first instruction to issue in the current cycle or if another load or store has not already issued. For the *divide* group, the instruction adds 8 cycles of latency. All other instructions fall into the *allinsts* group, in which no additional latency is present.

6.2.4 Graphite Integration

ADL currently supports an integration with the core performance models based upon MIT Graphite. To use this integration, simply run **adl2iss** or **uadl2model** with the **--graphite** option. This will then allow a plugin or external framework to install a performance model derived from the `adl::CoreModel` class declared in **graphite/GraphiteCore.h**. The method `CoreModel::handleInstruction` is called with information about each instruction executed. In addition, branch information is specified via `CoreModel::handleBranch` and memory information (currently only cache miss latencies) are specified via calls to `CoreModel::handleMem`.

Instruction classes can be defined in the configuration file (.ttc file) within **graphite_group** define blocks. Instructions or instruction attributes are listed in the **items** keyword to specify the contents of an instruction class. For example:

```
define (graphite_group=load_instrs) {
  items = load;
}
```

The special keyword `default_instruction` can be used within the item list to specify a default instruction class.

This information is used to generate a special header file, named **<pfx>-graphite.h**, where **<pfx>** is the base-name of the input file, which declares an enum listing all of the instruction classes. A performance model should include this header in order to be able to identify instruction types when `CoreModel::handleInstruction` is called.

An example integration is included in the ADL source tree, under **tests/graphite** demonstrating a simple performance model integration.

6.2.5 Graphite Usage With LSF

Graphite has been enhanced to work with Platform Computing's LSF compute cluster software. This allows Graphite to work across a series of hosts in a compute farm where LSF is the sole means of launching remote jobs and where ssh is not allowed. To use LSF, first run the script `launch-lsf-graphite`. By default, this launches four LSF jobs to act as Graphite servers and modifies the Graphite configuration file (default is *carbon_sim.cfg* in the current directory) to update the `process_map` section with the hosts running these processes. Run the script with **--help** or **--man** for more information about options.

Once this step is complete, simply run the **spawn_master.py** program as you normally would. The script will spawn jobs based upon the configuration file. These Graphite servers will continue to run until they are shutdown by running `launch-lsf-graphite --kill`.

Note that if you are running a multi-host simulation, then no element of the process map may be set to **localhost**; all entries must be set to absolute hosts. Otherwise, the various Graphite processes will not be able to communicate with each other. If the `launch-lsf-graphite` script is used, then absolute host names will always be present.

6.2.6 Graphite Peripheral Models

The Graphite integration has support for simple memory-mapped peripheral models. These are supplied to the simulator via a shared-object plug-in system. These models are able to respond to reads and writes via a callback mechanism and generate exceptions for cores which support the proper interface.

To create a Graphite plug-in, you must declare a class which derives from `adl::GraphitePlugin`, declared in `iss/GraphitePlugin.h`:

```
struct AccumPlugin : public adl::GraphitePlugin { };
```

You must also create the interface function, used by the simulator, to retrieve the plug-in object. Each plug-in should have only a single such object, but this object may then install an arbitrary number of peripherals. The interface function must be named according to the `GraphitePluginEntry` macro:

```
// Main entry point- returns a pointer to our service-provider object.
extern "C" adl::GraphitePlugin *GraphitePluginEntry()
{
  return new AccumPlugin;
}
```

The entry-point function should return a pointer to a heap-allocated instance of the plug-in object. This object will then be deleted by the simulation framework at the end of simulation.

The plug-in object currently only needs to overload a single function for installing callbacks:

```
void install_io_callbacks(bool main_process,IOServerBase *io_server);
```

The argument `main_process` will be true only for the Graphite's main process (process 0). For all other processes, `main_process` is false. Only the main process's peripherals are actually ever used; the other processes only need to have callbacks installed in order for each process to understand what addresses represent I/O accesses. Therefore, the code within `install_io_callbacks` should install all callbacks regardless of the value of `main_process`, but should limit any other operations to only when `main_process` is true, e.g. opening files, constructing large objects, etc. The `io_server` parameter represents a pointer to the object into which callbacks should be installed, and is declared in `io_server.h`.

A pointer to this object may be stored if `main_process` is true, and may then be used for generating exceptions.

Installing callbacks is just a matter of calling the helper macros `IOMapRegisterRead` and `IOMapRegisterWrite` with the appropriate arguments:

```
IOMapRegisterWrite(*io_server,addr,&AccumPeriph::write_accum);
IOMapRegisterRead(*io_server, addr,&AccumPeriph::read_accum);
```

In the example above, methods of the class `Accumperiph` are installed to handle reads and writes to the address specified by `addr`. Note that these helper macros assume that they are being called within a method of the same class as the callback methods.

The read callbacks are expected to have a method signature of:

```
uint32_t read_callback(core_id_t c,IntPtr addr,UInt64 &latency);
```

The `core_id_t c` parameter stores the address of the core performing the read, `addr` is the address being read, and `latency` may be updated with a latency value in order to track the time consumed by the read. The return value of the method is the result value of the read operation.

The write callbacks have the following method signature:

```
void write_callback(core_id_t c,IntPtr addr,uint32_t v,UInt64 &latency);
```

The value for the write is stored in `v`; the function does not return a value.

To generate exceptions, call the `send_exception` method of the `IOServer` object:

```
virtual void send_exception(core_id_t core_id,unsigned flags);
```

Where the `core_id` parameter is the address of the core which will receive the exception message and `flags` represents the exception flags to be sent.

Each Graphite plug-in is instantiated after the main Graphite simulator has been initialized. Therefore, the standard Graphite configuration database may be queried for setup information. This object is declared in `simulator.h` and may be referenced by calling the Graphite simulator singleton. For example:

```
_base_addr = Sim()->getCfg()->getInt("accum/base_addr",0);
```

In the above example, the peripheral expects that the configuration file has a section named **accum** with an entry called **base_addr**. For example:

```
[accum]
base_addr = 0x1000
```

An example of a simple accumulator peripheral is as follows:

```
tr.dynamic: no tr.dynamic policy could be found for the command 'states'.
```

To compile this object, use the `adl-config` utility to supply compiler and linker flags:

```
g++ -o accum-periph.so -shared -fPIC AccumPeriph.C $(adl-config --graphite --cflags)
```

Then specify the plug-in on the command-line via the **--plugin** option:

```
./ppc-graphite --config=./carbon_sim.cfg --general/total_cores=1 ./iotest1.elf --fep --no-output  --plug
```

6.2.6.1 The UVP Testwriter

The UVP format is a testcase file format produced by such tools as Raptor and used by testbenches such as UnitSim and the Vera Base Classes. It consists of an initial state, an instruction trace with intermediate results, and a final state. The ADL ISS models currently support the creation of a UVP testcase by either setting the output format (--output-format or --of command-line option) to `uvp` or by settin the output file name (--output or --o command-line option) to a filename with an extension of ".uvp".

In order to be compatible with Raptor, the UVP writer examines the resources of a model and modifies its behavior based upon the attributes assigned to those resources. The following lists the actions taken:

- MMU lookup fields with an attribute of `translation_attr_`<n> are considered to be translation-attribute fields and are printed within a UVP's `ta` field. The ordering is determined by the integer value <n>, with lower values placed to the left of higher values. So, for example, a field with the attribute of `translation_attr_0` would be the left-most component of the `ta` field.
- The *EPN* and *RPN* fields of each UVP T card will be shifted by the `pageshift` amount (right-shifted upon reading, left-shifted upon writing), in order to be compatible with Raptor-generated UVPs.
- Registers, exceptions, and TLB fields with an attribute of `unarchitected` are suppressed throughout the UVP, both for initial and final state and for intermediate results.
- Registers, exceptions, and TLB fields with an attribute of `indeterminate` will have their intermediate results suppressed.
- If an exception has an attribute of `squash_instr`, it will suppress the printing of the instruction it is associated with. This is useful for when an exception is designed to completely cancel an instruction, such as a thread-switch event. For example, this feature may be used if an instruction generating a thread switch event should not be printed when the thread switch happens, but rather when the thread switches back to the current thread.
- If an exception has an attribute of `asynchronous`, the exception will be displayed as an `E :` A card instead of as an intermediate result.

6.3 Documentation Generation

ADL makes use of reStructured Text for literate programming within an ADL specification. This is a light-weight plain-text mark-up language. The example code at the end of this document utilizes this format.

By convention, if the first statement of any *define* is a string, it will be taken to be the description for that block. An overriding *defmod* will replace this string. The documentation back-end will extract resources that it cares about, such as instructions and registers, along with certain attributes specific to documentation (see below).

For example, a register entry might consist of the register's name, a table graphically showing all of the fields of the register, followed by the description string. An instruction might have a graphical depiction of the opcodes and fields, followed by the code that describes the semantics of the instruction, followed by the documentation, and then a list of all registers that it reads or modifies.

6.3.1 Usage

The documentation generation tool is named *adl2doc*. It will generate the reStructured Text output file, and then optionally convert this to HTML. The program takes the following arguments. Flags may be negated by preceding the option with *no*, e.g. *--no-full-path* turns off the display of full paths.

Usage:

```
adl2doc [options] <model file>
```

Options:

--help, --h:
 Display help

--man, -m:
 Display the complete help as a man page.

--config=path, -cf=path
 Specify a configuration file.

--title=str, -t=str
 Specify a title for the generated documentation. Only one title is allowed; multiple title options will overwrite the prior value.

--subtitle=str, -st=str
 Specify a subtitle for the document. Multiple options are allowed; each will produce a separate subtitle line.

--html
 Generate HTML. This option may be negated, in which case only the intermediate reStructured Text file will be generated.

--output=file, -o=file
: Specify the output file name.

--print-source-location, --psl
: Print source location information (filename and line number) for action code. Default is off. This option may be negated.

--full-path, --fp
: Display the full path for all location information. If off, only the base-name is displayed. The default is true. This option may be negated.

--orig-action
: Display all original action/hook code without any aspect expansion or other processing. The default is false. This option may be negated.

--instr-prefix-bits, --ipb
: Include prefix bits in instruction encodings. The default is true. This option may be negated.

--ops-in-affect-tables, --oat
: Include explicit operands in affected/affected-by tables in instruction documentation. If set to false, only implicit operands will be shown in these tables, with the operands specified only in the B<Operands> table. Default is true. This option may be negated.

--proportional-subfields, --psf
: Force sub-fields to be displayed with widths proportional to their bit sizes. Default is false. This option may be negated.

--syntax-prefix-fields, --spf
: Display prefix fields in the syntax string. These are fields which precede the instruction mnemonic. The default is true. This option may be negated.

--registers, --reg
: Include a section on registers. The default is true if any are defined. This option may be negated.

--register-files, --rf
: Include a section on register files. The default is true if any are defined. This option may be negated.

--instruction-fields, --if
: Include a section on instruction fields. The default is true if any are defined. This option may be negated.

--instructions, --instr
: Include a section on instructions. The default is true if any are defined. This option may be negated.

--instr-by-attr, --iba
: Include a cross-section on instructions by attribute. The default is true if any are defined. This option may be negated.

--instr-by-block, --ibb
: Include a cross-section on instructions by block. The default is true if any are defined. This option may be negated.

--exceptions, --exc
: Include a section on exceptions. The default is true if any are defined. This option may be negated.

--caches
: Include a section on caches. The default is true if any are defined. This option may be negated.

--memories, --mem
: Include a section on memories. The default is true if any are defined. This option may be negated.

--event-buses, --eb
: Include a section on event buses. The default is true if any are defined. This option may be negated.

--contexts
: Include a section on contexts. The default is true if any are defined. This option may be negated.

--mmu
: Include a section on the MMU. The default is true if any lookups are defined. This option may be negated.

--core-level-hooks, --clh
: Include a section on core-level hooks. The default is true if any are defined. This option may be negated.

--helper-functions
: Include a section on helper functions. The default is true if any are defined. This option may be negated.

--prest=path
: Specify a path to the Prest reStructured Text processing program.

--prest-args=str
: Specify additional arguments to Prest. This option may be repeated.

--define=str, -D=str
: Specify a preprocessor define.

--include=path, -I=path
: Specify a preprocessor include directory.

--hl-level=int

0:
: Code is displayed using a verbatim block.

1:
: Code is highlighted using basic C++ highlighting.

2:
: Additional highlighting is performed, including transformation of C++ operators to mathematical logic symbols, where appropriate.

6.3.2 Influential ADL Keys and Attributes

By convention, keys in defines that are prefixed with "doc_" are specific to documentation. See Definition Types And Their Keys, and "doc_title".

Attributes are also sometimes used to affect documentation, and they follow a similar "doc_" prefix convention. The following attributes are currently used:

doc_hidden:
: Currently allowed in registers, register files, and instruction fields; suppresses all of the documentation for the object. See also "doc_no_code".

doc_no_code:
: Currently allowed in registers and register files; prevents the source code for read/write from appearing in documentation.

doc_no_expand_exprs:
: Currently supported for instructions. If the instruction has this attribute, then its extracted documentation will not be expanded (only relevant if the database is generated with the **--expand-exprs** option).

In addition, the user can designate a portion of code to be designated as the *operation* for a given instruction using the special label `doc_op`. For example:

```
define (instr=add_family) {
  fields=(Src1,Src2,Trg);
  action={
  doc_op : GPR(Trg) = GPR(Src1) + GPR(Src2);
    setCrField(0,GPR(Trg),0);
  };
```

Only the code `GPR(Trg) = GPR(Src1) + GPR(Src2)` would be extracted as the operation because it has the special label. This code is then presented separately in the documentation.

6.3.3 Configuration File

The operation code may then undergo further transformations by specifying them in the `op_transforms` block within the configuration file supplied to **adl2doc**. The format is:

- Definition: *op_transforms*: Specify transformations for operation code.
  - *item = (<src> , <result>)*: Specify a source and result for a transformation. This key may be repeated.

Example input file:

```
define (op_transforms) {
  item = ( fp_fusedmpy(0x80000000,$y,$z,true), ($x + $y) );
  item = ( fp_fusedmpy($x,$y,$z,true), ($x + $y * $z) );
  item = ( GPR(Src1) , Ra );
  item = ( GPR(Src2) , Rb );
  item = ( GPR(Trg) , Rt );
}
```

In the above example, `GPR(Src1)` will be transformed to `Ra` in the operation description.

Any token in the **src** expression which starts with $ is considered a capturing token. It will match against any expression and store the result. During the substitution phase, the token of the same name will be replaced by the captured value.

Note that matching is performed by scanning through the transformations in order, starting with the first item listed. Therefore, place the most restrictive transformation first, such as in the above example, so that it will be compared before a less restrictive transformation.

6.3.3.1 Special Transform Functions

Special functions are defined for use in the transformation expression. They are listed here.

- *_eval_(<expr>)*: Evaluates simple numerical expressions. Substitution is first performed on the argument expression, then the expression is evaluated. If it cannot be evaluated, the original expression (after substitutions) is returned. Only basic binary and unary mathematical operations and numeric constants (after substitution) are handled.
- *_null_()*: Return a null (empty) expression. Useful for creating a transform designed to simply remove the input pattern entirely.
- *_str_(<format> [,args])*: Create a new token using the first argument as a format string. Subsequent arguments are used according to the format tags within the format string. The format string supports the following tags:

    - **%p**: Insert the argument directly.
    - **%s**: Insert the argument, converting it to a string first.
    - **%%**: Insert the character "%".

  This function is useful for when the resulting transform uses tokens or a format which cannot be expressed as a standard C++ expression.

7 Example Code

The following is an example model which implements a small portion of a PowerPC:

```
// Various helper routines.

define (arch = minippc) {
```

```
void setCrField(bits<32> field,bits<32> x,bits<32> y)
  {
    bits<4> r =
      ( (x.signedLT(y)) ? 0x8 : 0) |
      ( (x.signedGT(y)) ? 0x4 : 0) |
      ( (x == y)        ? 0x2 : 0) ;

    CR.set(4*field,4*field+3,r);
  }

//
// Registers.
//

define (reg=CIA) {
  """
  Current instruction address.
  """;
  attrs = cia;
}

define (reg=NIA) {
  """
  Next instruction address.
  """;
  attrs = nia;
}

define (reg=CR) {
  """
  The condition register.
  """;
}

define (reg=CTR) {
  """
  The counter register.
  """;
}

define (reg=ESR) {
  """
  Exception syndrome register.
  """;
}

define (reg=MSR) {
  """
  Machine state register.
  """;
  define (field=EE) {
    bits = 9;
  }
  define (field=PR) {
    bits = 10;
  }
}

define (reg=SRR0) {
  """
  Save-restore register 0.
  """;
}
```

```
define (reg=SRR1) {
  """
  Save-restore register 1.
  """;
}

define (reg=IVPR) {
  """
  Interrupt-vector prefix register.
  """;
}

define (reg=IVOR5) {
  """
  Interrupt-vector offset register 5.
  """;
}

define (reg=IVOR6) {
  """
  Interrupt-vector offset register 6.
  """;
}

define (reg=SPRG0) { }

define (regfile=GPR) {
  """
  General purpose registers.
  """;
  size = 32;
}

define (regfile=SPR) {
  size=1024;
  define (entry=9) { reg=CTR; }
  define (entry=62) { reg=ESR; }
  define (entry=26) { reg=SRR0; }
  define (entry=27) { reg=SRR1; }
  // Supervisor-mode register: Throw a program exception if not in supervisor
  // mode.
  define (entry=50) {
    reg = SPRG0;
    define (read) {
      action = {
        if (MSR.PR == 1) {
          raiseException(Program);
        }
        return SPRG0;
      };
    }
    define (write) {
      action = func (bits<32> x) {
        if (MSR.PR == 1) {
          raiseException(Program);
        }
        SPRG0 = x;
      };
    }
  }
  // User-mode register:  Read-only.
  define (entry = 51) {
    reg = SPRG0;
    define (write) { ignore = true; }
  }
}

//
// Instruction fields.
//

define (instrfield=OPCD) {
  """
  Primary opcode.
  """;
  bits = (0,5);
}

define (instrfield=XO) {
  """
  Extended opcode.
  """;
  bits = (21,30);
}

define (instrfield=BO) {
  """
  Field used to specify options for the Branch Conditional instructions.
  """;
  bits = (6,10);
}

define (instrfield=BI) {
  """
  Field used to specify a bit in the Condition Register to be used
  as the condition of a Branch Conditional instruction.
  """;
  bits = (11,15);
}

define (instrfield=BD) {
  """
  Immediate field specifying a 14-bit signed two's complement branch displacement
  which is concatenated on the right with 0b00 and sign-extended.
  """;
  bits = (16,29);
}

define (instrfield=BF) {
  """
  Field used to specify one of the Condition Register fields or one of the
  Floating-Point Status and Control Register fields to be used as a target.
  """;
  bits = (6,8);
}

define (instrfield=AA) {
  """
  Absolute address bit.
  """;
  bits = 30;
}

define (instrfield=LK) {
  """
  LINK bit.
  """;
  bits = 31;
```

```
}

define (instrfield=SPRN) {
  """
  Field used to specify a Special Purpose Register for the *mtspr* and *mfspr* instructions.
  """;
  bits = ((16,20),(11,15));
}
define (instrfield=RA) {
  """
  Field used to specify a General Purpose Register to be used as a source.
  """;
  bits = (11,15);
}
define (instrfield=RB) {
  """
  Field used to specify a General Purpose Register to be used as a source.
  """;
  bits = (16,20);
}
define (instrfield=RT) {
  """
  Field used to specify a General Purpose Register to be used as a target.
  """;
  bits = (6,10);
}
define (instrfield=RS) {
  """
  Field used to specify a General Purpose Register as a target.
  """;
  bits = (6,10);
}
define (instrfield=D) {
  """
  Immediate field used to specify a 16-bit signed two's complement integer
  which is sign-extended to 64-bits.
  """;
  bits = (16,31);
}
define (instrfield=SI) {
  """
  Signed immediate field for arithmetic operations.
  """;
  bits = (16,31);
}
define (instrfield=UI) {
  """
  Unsigned immediate field for arithmetic operations.
  """;
  bits = (16,31);
}

define (instrfield=SH) {
  bits = (16,20);
}
define (instrfield=MB) {
  bits = (21,25);
}
define (instrfield=ME) {
  bits = (26,30);
}

//
// Instructions.
```

```
//

define (instr=add) {
  fields=(OPCD(31),RT,RA,RB,XO(266));
  action = {
    GPR(RT) = GPR(RA) + GPR(RB);
  };
}

define (instr=addi) {
  fields=(OPCD(14),RT,RA,SI);
  action = {
    var si = signExtend(SI,32);
    if (RA == 0) {
      GPR(RT) = si;
    } else {
      GPR(RT) = GPR(RA) + si;
    }
  };
}

define (instr="addic.") {
  fields=(OPCD(13),RT,RA,SI);
  action = {
    var si = signExtend(SI,32);
    GPR(RT) = GPR(RA) + si;
    setCrField(0,GPR(RT),0);
  };
}

define (instr=addis) {
  fields=(OPCD(15),RT,RA,SI);
  action = {
    bits<32> si = SI;
    if (RA == 0) {
      GPR(RT) = si << 16;
    } else {
      GPR(RT) = GPR(RA) + (si << 16);
    }
  };
}

define (instr=bc) {
  fields=(OPCD(16),BO,BI,BD,AA(0),LK(0));
  action = func() {
    if ( (BO(2) ) == 0) {
      CTR = CTR - 1;
    }
    var ctr_ok = BO(2) || ( (CTR!=0) ^ BO(3));
    var cond_ok = BO(0) || ( CR(BI) == BO(1));
    if ( ctr_ok && cond_ok ) {
      var ea = signExtend(concat(BD,zero(2)),32);
      NIA = CIA + ea;
    }
  };
}

define (instr=cmpi) {
  fields=(OPCD(11),BF,RA,SI);
  action = func () {
    var si = signExtend(SI,32);
    setCrField(BF,GPR(RA),si);
  };
}
```

```
define (instr=cmp) {
  fields=(OPCD(31),BF,RA,RB,XO(0));
  action = {
    setCrField(BF,GPR(RA),GPR(RB));
  };
}

define (instr=lwz) {
  fields=(OPCD(32),RT,RA,D);
  action = {
    var d = signExtend(D,32);
    var b = (RA == 0) ? 0 : GPR(RA);
    var addr = b + d;
    GPR(RT) = Mem(addr,4);
  };
}

define (instr=lwzx) {
  fields=(OPCD(31),RT,RA,RB,XO(23));
  action = {
    var b = (RA == 0) ? 0 : GPR(RA);
    var addr = b + GPR(RB);
    GPR(RT) = Mem(addr,4);
  };
}

define (instr=mtspr) {
  fields=(OPCD(31),RS,SPRN,XO(467));
  action = {
    if (!SPR.validIndex(SPRN)) {
      ESR(4) = 1;
      raiseException(Program);
    }
    SPR(SPRN) = GPR(RS);
  };
}

define (instr=mfspr) {
  fields=(OPCD(31),RT,SPRN,XO(339));
  action = {
    if (!SPR.validIndex(SPRN)) {
      ESR(4) = 1;
      raiseException(Program);
    }
    GPR(RT) = SPR(SPRN);
  };
}

define (instr=mullw) {
  fields=(OPCD(31),RT,RA,RB,XO(235));
  action = {
    GPR(RT) = GPR(RA) * GPR(RB);
  };
}

define (instr=or) {
  fields=(OPCD(31),RS,RA,RB,XO(444));
  action = {
    GPR(RA) = GPR(RS) | GPR(RB);
  };
}

define (instr=ori) {
  fields=(OPCD(24),RS,RA,UI);
  action = {
    GPR(RA) = GPR(RS) | UI;
  };
}

define(instr=rlwinm) {
  fields=(OPCD(21),RS,RA,SH,MB,ME);
  action = {
    var r = GPR(RS).left_rotate(SH);
    bits<32> m;
    m.mask(MB,ME);
    GPR(RA) = r & m;
  };
}

define (instr=rfi) {
  fields=(OPCD(19),RS,RA,RB,XO(50));
  action = {
    MSR = SRR1;
    NIA = SRR0;
  };
}

define (instr=stw) {
  fields=(OPCD(36),RS,RA,D);
  action = {
    var b = (RA == 0) ? 0 : GPR(RA);
    var d = signExtend(D,32);
    var addr = b + d;
    Mem(addr,4) = GPR(RS);
  };
}

define (instr=stwu) {
  fields=(OPCD(37),RS,RA,D);
  action = {
    var d = signExtend(D,32);
    var addr = GPR(RA) + d;
    Mem(addr,4) = GPR(RS);
    GPR(RA) = addr;
  };
}

define (instr=stwx) {
  fields=(OPCD(31),RS,RA,RB,XO(151));
  action = {
    var b = (RA == 0) ? 0 : GPR(RA);
    var addr = b + GPR(RB);
    Mem(addr,4) = GPR(RS);
  };
}

// Special instruction:  This is used for simulation purposes and is
// not a PPC instruction.
define (instr=halt) {
  fields=(OPCD(0));
  action = {
    halt();
  };
}

//
// Decode miss handler.
```

```
  //
  decode_miss = func (addr_t ea,unsigned) {
    ESR(4) = 1;
    raiseException(Program);
  };

  //
  // Post-Fetch handler.
  //
  post_fetch = func (unsigned size) {
    NIA = NIA + size;
  };

  //
  // Program interrupt.
  //
  define (exception=Program) {
    priority = 1;
    action = {
      SRR0 = CIA;
      SRR1 = MSR;
      MSR = 0;
      NIA = concat(IVPR.get<16>(0,15),IVOR6.get<16>(16,31));
    };
  }

  //
  // External interrupt.
  //
  define (exception=External) {
    sensitivity = level;
    enable = { return MSR.EE == 1; };
    action = {
      SRR0 = CIA;
      SRR1 = MSR;
      MSR = 0;
      NIA = concat(IVPR.get<16>(0,15),IVOR5.get<16>(16,31));
    };
  }

}

define (core = P) {
  archs = minippc;
}
```

Generated on: 2018/02/28 15:17:16 MST.