1 ADL Language Reference

This document describes the architectural description language ADL. The purpose of an 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

Contents

1.1 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:

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.
```

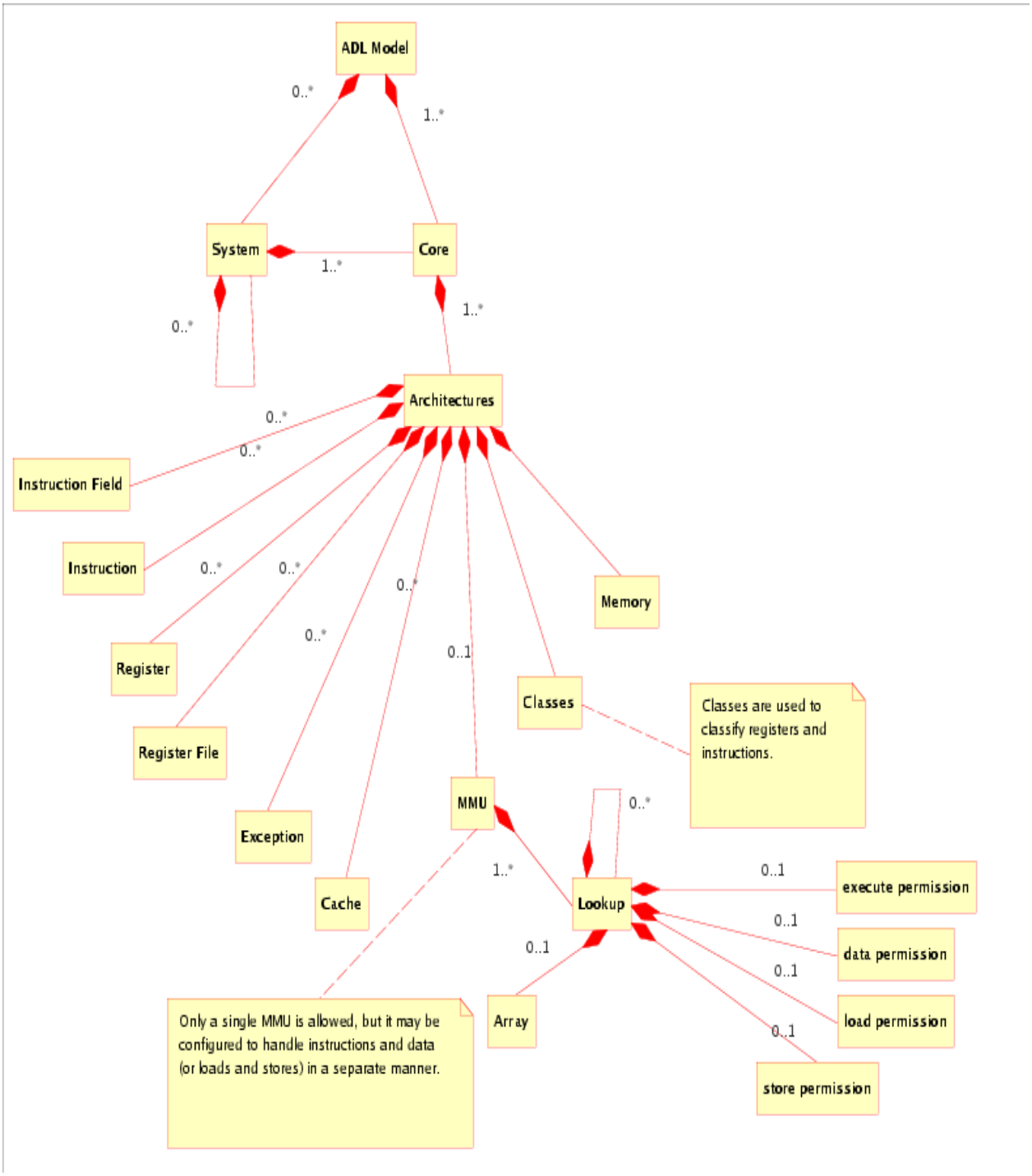

Figure 1: ADL Model Organization

```
  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);
    }
  };
}
```

1.2 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.
- A new block-statement type will be added which will allow a block of code to be named:

```
block (<ident>) {
  ...
}
```

This can be used by a back-end to exclude code, e.g. a documentation tool might exclude all code labeled as *NoDocument*. It will also be useful for aspect oriented programming, as described below.

1.3 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.

1.3.1 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;
  }
  ```

1.3.2 Definition Types And Their Keys

- 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;
    }
    ```

  - *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.
    - *memories* = <*list*>: List shared memories.
    - *eventbuses* = <*list*>: List shared event buses.
  - Definition: *sys* = <*name*>: Systems may be nested. These are treated as local systems, visible only to the containing system block.

- 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 ISSs 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.

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

- *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.
- *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.
- *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_fetch = func()* {}: Called just before an instruction fetch. The default behavior is to do nothing.
- *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.
- *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 reset prefix data, if defined.
- post_packet_asm = func() {}*: Called after all instructions in a parallel execution set*
  are recognized by the assembler, but before instructions arewritten. Currently only assembler parameters and prefix variables are accessible. Default, take no action.
- *parallel_execution = <int>*: If greater then zero, size of parallel execution set, otherwise, serial computation. Default is zero.
- *blk = <ident> — <list>*: List of fuctional blocks in the architetcure. 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. Attribute can have optional parameter. Parameter can be of type: interger, string, list of integers or list of strings. The type of attribute's parameter is determined automatically. It is allowed to use parametrized attributes without parameter. In order to remove some attribute use *remove_attrs = ident — list*.

  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.

- *bit_endianness = <little—big>*: Optional, specifies bit endianness of the core. Default is big endian (the most significant bit is number 0).
- 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".
- 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".
- 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.
- Definition: *aspect*: Define a code aspect. This allows a block of code to be added before or after a specified block of code or instruction action. This can be useful, for example, to specify that certain instructions should signal an unimplemented-instruction exception, rather than actually executing the instruction.
    - *instr = <ident> — <list>*: Specify that the aspect should target instructions. The value is the name of the target instruction, or a list of names of instructions. **Note**: This may not be specified in conjunction with *block*.
    - *blk = <ident> — <list>*: Specify that the aspect should target blocks. The value is the name of the target block, or a list of names of blocks. **Note**: This may not be specified in conjunction with *instr*.
    - *before = <true — false>*: Insert this aspect before the specified code. This is the default location.
    - *after = <true — false>*: Insert this aspect after the specified code.
    - *action = func* { }: The code to insert.
- Definition: *instrfield = <name>*: Define an instruction field.
    - *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.
        - *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, the denotes that a field implements another field.
    - *ref = <ident>*: If the type is one which refers to another resource, such as *regfile* or *memspace*, 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. 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. It's 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.
* *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 the field is not mapped to bits, therfore, may not be used in any actual instruction encoding. It may be used only within syntax strings, by aliases, or by instructions which their encoding is defined either by aliases or by the bit-mapped fields notation. 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 varibable 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 spcecified width, where the specific field is selected based on the instruction's position in the VLES. The width of the field, if not specified, is *paralle_execution* times *indexed*.

* *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.
* *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.
* *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.
  - *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*. It can be together with
* Defintion: *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, represening 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)*.
  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 the another instruction field. Useful then instructions in assembler are distinguished by instruction field syntax. Can be used in syntax similar to how expressions are used – usign functional notation. Examle:: 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*.

– Definition: *subinstr = <name>*: Define a sub-instruction.
  * attrs = <list>*: List any attributes that this sub-instruction is* associated with. Attributes will be inherited by all instruction built from this subinstruction.
  * *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*.
  * *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 subinstruct
Note, that usual syntax::

defmod (subinstr = subinstr1) {
      ...
    }

has different meaning, it modifies parameters of the singe subinstruction.
```

– Definition: *instr* = <*name*>: Define an instruction.
  * *width* = <*int — variable*>: Instruction width, in bits. We currently only support widths of even byte amounts and our maximum is 32 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.
  * *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 subinstrs, or bit-mapped fields. The sub-instruction fields are added to instruction. A subinstrs 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),(bit
  ```

  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*.
  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.
* *blk* = <*ident*>: Optional, specifies the logical block this instruction is associated with. If specified, must be identical to the block of all its fields.
* *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.
* 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.
* *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, sytanx and alias keywords. The alias should refer to the encapsulating instruction. The name of a nested instrcution 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.

* *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 - PC));*
```

where X and Y are instruction fields, X may be a pseudofield but Y must be a real instruction field. Key word "PC" stands for a program counter. Note, expressions are not supported for nested instruction fields.

Fields that do not appear in <*fields ...*> list must be defined by one of expressions. Expression can be simple arithmetic expression consisting of constants or fields name 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 pseudofield 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, A(A+1), P, BS(P>0xFFFF ? 1 : 0), IMM(P>0xFFFF ? P>>16
```

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.
* *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.
* *alias* = <*call*>: The function name 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));
  }
  ```
  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`.

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.

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

- 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.
  - Definition: *field* = <*name*>: Define or modify a register field.
    - *attrs* = <*list*>: Lists any attributes that this instruction 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.

  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.

* *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)` syntax must e used, where *ident* is the name of the register-file and *int* is the index of the element to be aliased. 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*>): Only valid if *alias* is specified. Indicate a slice of the aliased register to use on a read.
  - *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, unless *pseudo* was specified, in which case no underlying register exists.
* 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.
- *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, unless *pseudo* was specified, in which case no underlying register exists.

* *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.
* *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;
```

```
  }
}
```

- 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.
  - 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 must be specified using the `ident(index)` syntax.
    - 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.
      - *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*>: Specifiy a string to be printed in disassembler for instruction field referring to this regfile.
  - *attrs* = <*list*>: Same as for *reg*.
  - *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.
* *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.
* *fields = <list>*: Same as for *reg*. These may only be applied to non-sparse register files. All registers within the file have the same fields.
* *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.

- 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.
  - *attrs* = <*list*>: Lists any attributes that this context is associated with.
  - *regs* = <*list*>: List registers in this context.
  - *regfiles* = <*list*>: List register files in this context.
  - *num_contexts* = <*int*>: Specify the number of contexts.
  - *active* = *func* { ... }: This function specifies which instance is active. It must return an integer representing the active element (0..“num_contexts“).
- 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 by calling the *raiseException* function with the name of an exception. This will cause an exception to be immediately processed. An exception may be merely registered using the function *setException*. This will cause the exception to be handled after the instruction has completed.
  - *attrs* = <*list*>: Lists any attributes that this exception is associated with.
  - *action* = *func()* { }: 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.
  - *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.
- 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.
  - Definition: *lookup* = <*name*>: The MMU contains one or more *lookup* objects which 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.
- *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>*: 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).

- *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.
- *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.
- *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:
  * *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.
- *exec_perm = func(<lookup name> [,<parent-lookup name> ...],addr_t ea,int 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,int 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,int 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,int 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.
- *miss = func (TransType tt,addr_t ea,int 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,int 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.`

  - *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(addr_t)* { }: Code to execute when more than one lookup object finds a mapping and there is no priority order.
  - *data_miss = func(addr_t)* { }: Code to generate when no mapping is found on a data translation. This is the same as specifying a value for *load_miss* and *store_miss*.
  - *instr_miss = func(addr_t)* { }: Code to generate when no mapping is found on an instruction translation.
  - *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.

- 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.
* *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.

* *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)* { }: 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.
* *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)* { }: 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.

- 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, it 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.

- 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. 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.

– Definition: *group*: Create a group.

  * *type = <ident>* : Optional, defines type of the group. If missing it will be deduced from group items.
  * *items = <list>* : Specifies items of the group. Item can be:
    1. An ADL object.
    2. A reference group of ADL objects (denoted '*').
    3. Set intersection (denoted '&') or set difference (denoted '-') 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). E.g.:

```
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
}

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.

– Definition: *assembler*: Configurates 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 to new-line separators, default is ';';
* *queue_size* = <*int*> : Optional, defines the size of packets queue the assemlber maintains, default is 1. The queue allows assembler function code address packet preceding the current one.
* *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.
* *instrtables* = <*list*>: List clases of instructions to be grouped into one decoding tree of dissassembler. The list is orderer, so first disassembler searches instruction in the first instruction table, if not found it trying to look in the next one etc. This is useful for the prefix encodings, where prefix instructions residue in the "prefix" instruction table but no explicit table change is written in assembly.
* exlicit_regs = <bool>*: Optional, specifies if one have to write explicitly*
  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.
* 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;
      }
}
```

1.4 Predefined Global Resources

1.4.1 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.

1.4.2 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;
```

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:

- `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.

1.4.3 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.

1.4.4 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);
```

1.4.5 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 a specific set and way way be 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 specifis 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;
```

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.

1.4.6 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)*: Estables 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) const*: Returns true if the cache is enabled, false if not.
- *void fill(addr_t ra,byte 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 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.
- *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 = ... ;
```

1.4.7 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:

- *num_contexts() const*: Return the number of elements for this context.

- *unsigned active_context() const*: Returns the index of the active context element.

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.

1.4.8 Global Functions

Various predefined helper functions exist for handling common operations.

- *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 raiseException(<exception name>)*: Cause the specified exception to occur immediately.
- *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.
- *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 able to the one specified.
- *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 getCurrnetInstrTable():* Returns the nameof 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 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 dataWriteTanslate(addr_t ea)*: Translates the supplied effective address and returns real address.
- *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, exxecution set results are commited and `post_packet` handler, if defined, is invoked. Defined only for parallel architectures which use prefix.
- *void info(int level, ...)*: Creates an annotation message which is then sent to the test writer. The *level* parameter allows users to assign a priority to each message for filtering purposes. All subsequent arguments 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);*. 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`.

1.4.9 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.
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.
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.

1.5 Usage Notes

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

1.5.1 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.

1.5.2 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;
      }
    };
  }
}
```

1.5.3 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.

1.5.4 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) {
  raiseException(ITlbMiss);
```

```
  };

  data_miss = func (addr_t) {
    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.

1.5.5 Delay Slots

Delay slots may be modeled within ADL by using the *post_fetch* handler to modify how the next-instruction-address is calculated. For example, a model might define the following registers:

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

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

define (reg=TIA) {
  """
  Address of the delay slot instruction.
  """;
}
```

It would also define a parameter to act as a flag in order to identify if it was executing a delay slot:

```
define (parm=Fetch) {
  options = (Normal,DelaySlot);
  value = Normal;
}
```

A branch would then update the TIA register with the target address of the branch:

```
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);
      TIA = CIA + ea;
      Fetch = DelaySlot;
    }
  };
}
```

Finally, a post-fetch handler would query the flag and update the next-instruction-address register appropriately:

```
post_fetch = func(unsigned size) {
  NIA = (Fetch == DelaySlot) ? TIA : (CIA + size);
  Fetch = Normal;
};
```

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
```

1.5.6 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.

1.5.7 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.

1.5.8 Assembler Instructions

In some architectures, like StarCore, assembler tasks are more complex than just recoginizing a single instruction and correctly encoding it. Assembler should handle packets of instructions, add required prefix information, and allow an instruction to effect encoding of a future instruction/packet.

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

packet_grouping defines charaters or characters 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 notaion, 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.

parm block in the *assembler* resource can define additional parameters of *integer* or *label* type. Integer parameters must have default value. Label parameters can't have defualt.

The blow noted code is a simplified version of StarCore's skipls instruction. The instruction is a conditional jump whose target always succeeds the 'loopend' assembler directive. If 'loopend' occurs two packet, 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=LoopEnd) {
      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;
  };
};
```

1.6 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.

1.6.1 Assembler/Disassembler Generation

The ADL program *adl2asm* generates assembler and disassembler files. These C files can then be placed into a special GNU binutils distribution and compiled to create a the *gas* assembler and *objdump* disassembler. This special distribution is stripped down to just handle a PowerPC target, at present, and adds some auxiliary files which are generic to all ADL generated assemblers. To handle another type of target, the user must add necessary auxiliary files and modify the build system to include them in the necessary libraries.

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

1. Untar the *binutils* distribution into the current directory, creating the directory *binutils-adl*:

   ```
   tar zxvf *<adl distribution dir>*/share/adl/binutils-adl.tar.gz
   ```

   The location of this directory will be referred to as *binutils-dir*.

2. Create the assembler. This assumes that the ADL source file is named *model.adl* and that all entry-point functions will include a `ppc` prefix.

./adl2asm model.adl –bprefix=<binutils-dir> –arch-pfx=ppc

3. Configure and make the binutils package:

```
cd <binutils-dir>
./configure --target=ppc-elf
make
```

At present, the generated assembler does not inherently handle relocations; these must be handled by user supplied code. In the future, we hope to expand ADL to generate relocation information.

Multiple Instruction Tables The genearted 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
```

1.6.2 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.

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.

Usage **Usage**:

```
make-iss|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.

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

–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,

–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.

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

–syscall-enabled*:*
Enable system-call 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.

–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).

–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.

–verbose*:*
Show the output of all internally executed commands.

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.
    - *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.
    - *latency* = <*int*>: The latency assigned to these instructions. The default is 1.
    - *delay* = <*int*>: The delay before a member of this group may be issued again. This models a non-pipelined resource, but may be combined with *latency* to model complex units which are pipelined but cannot accept a new instruction every cycle. The default is 0 (fully-pipelined).
    - *count* = <*int*>: Specifies the number of resources able to handle instructions within this group. For example, all load-store instructions might be grouped together with a count of 1. A multi-issue machine would then be able to issue only one load/store instruction during a given cycle. The default is 1 if a delay is specified, otherwise there are no limits.
  - Definition: *config*: Configure the core.

    - * *issue_width* = <*int*>: Specify the number of instructions that may be issued during a single cycle.
    - * *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*.
    - * *tracing_on* = *func()* { ... }: This hook is called when tracing is turned on.
    - * *tracing_off* = *func()* { ... }: This hook is called when tracing is turned off.
  - 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.
  - *trace_mode* = <*bool*>: Turn on or off trace-mode code generation in the model.
  - *debug_mode* = <*bool*>: Turn on or off debug-mode code generation in the model.
  - *verbose_mode* = <*bool*>: Turn on or off verbose-mode during model generation.
  - *line_directives* = <*bool*>: Turn on or off the generation of line directives in the model.
  - *time_tagged* = <*bool*>: Turn on or off time-tagged code generation in the model.
  - *extern_mem* = <*bool*>: Specify whether *Mem* is external or internal.
  - *rnumber* = <*bool*>: Enable or disable generation of RNumber (dynamic arbitrary-width integer) support.
  - *jit* = <*bool*>: Enable or disable generation of JIT (just-in-time compilation) support.
  - *parallel-execution* = <*bool*>: Enable or disable parallel execution of instructions.
  - *exec-set-size* = <*int*>: Fixed size of parallel execution set.

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.
- Registers with an attribute of `unarchitected` are suppressed throughout the UVP, both for initial and final state and for intermediate results.
- Registers with an attribute of `indeterminate` will have their intermediate results suppressed.
- Exceptions with an attribute of `unarchitected` are suppressed within intermediate results.
- 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.

1.6.3 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, and generate a document using these strings, as well as other information from the resource. Its output will be reStructured Text that can then be converted into HTML or PDF via Latex.

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.

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.

The documentation generation tool is named *adl2doc* and 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-line-directives* turns off the insertion of line directives.

Usage **Usage**:

```
adl2doc [options] <model file>
```

Options:

–help, –h*:*
Display help

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

–title=str, -t=str
Specify a title for the generated documentation.

–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.

–prest=path
Specify an alternate location for the reStructured Text parsing program.

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

1.6.4 Compiler Generation Notes

This will need to be fleshed out at a later date. The lack of a generic means for configuring a compiler will make this task somewhat difficult. It seems to me that there are two aspects to be concerned with: Mapping C code to assembly and adding support for special intrinsics. For example, gcc already supports PowerPC, but it would be nice to be able to configure it to support a new APU by adding the instructions as intrinsics that could be used from within a C file.

This second problem seems fairly tractable: A back-end could extract the instructions of interest, i.e. by specifying an architecture or class, and produce

a *.md* file with these instructions specified as intrinsics. The *spe.md* file in gcc would be an example of this kind of output.

For the other problem, actually mapping an ADL's instructions to C, the problem is slightly more difficult, since there is no standard way to specify this mapping. On possibility is to basically use gcc's RTL syntax. The task of configuring gcc would then be trivial and we could hopefully use that same information to configure other compilers. This aspect, of course, needs more research and input from the DevTech

1.7 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=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=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* instructio
  """;
  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.set(4);
      raiseException(Program);
    }
    SPR(SPRN) = GPR(RS);
  };
}
```

```
define (instr=mfspr) {
  fields=(OPCD(31),RT,SPRN,XO(339));
  action = {
    if (!SPR.validIndex(SPRN)) {
      ESR.set(4);
      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=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=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.set(4);
    raiseException(Program);
  };

  //
  // Post-Fetch handler.
  //
  post_fetch = func (unsigned size) {
    NIA = NIA + size;
  };

  //
  // Program interrupt.
  //
  define (exception=Program) {
    action = {
      SRR0 = CIA;
      SRR1 = MSR;
      MSR = 0;
      NIA = concat(IVPR.get<16>(0,15),IVOR6.get<16>(16,31));
    };
  }

}

define (core = P) {
  archs = minippc;
}
```