HMDES Version 2.0 Specification

John C. Gyllenhaal Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois, Urbana-Champaign, IL 61801
gyllen, hwu@crhc.uiuc.edu
http://www.crhc.uiuc.edu/Impact/

B. Ramakrishna Rau

Hewlett Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304
rau@hpl.hp.com

Technical Report IMPACT-96-3

Contents

1 Introduction

This document is an initial specification of how a compiler-independent machine description should be written in the MD language (one that both HP labs and IMPACT can share) and how an compiler-specific machine description could be built on top of this machine description (I.e, for IMPACT or Elcor). Since the machine descriptions for both IMPACT and Elcor continue to evolve, their current state may not be perfectly reflected in this specification (accurate as of October 1995). In addition, the compiler-independent/compiler-specific division has not yet been set in stone. Both are described in this document to provide some context for the compiler-independent machine description and to allow easy moving of the division between the compiler-independent and compiler-specific information.

This specification document starts with an overview of the MD language in Section 2. The MD language will be extensively used in this specification to define the structure and contents of a machine description. Section 3 then describes the syntax and features of the MD preprocessor. MD preprocessor's directives will be used in many of the machine description specification examples to illustrate how they can be used to write concise machine descriptions.

The remaining sections of this specification describe how a compiler-independent and a compiler-specific machine description can be written using the MD language. The structural overview of the MD sections used to represent a machine description is shown in Figure 1. The interconnection between MD sections are represented by arrows between the section names. The compiler-independent machine description sections and interconnections are shown with bold type and arrows. One possible way to build a compiler-specific machine description for IMPACT, on top of the compiler-independent machine description, is shown with highlighted type and arrows.

This machine description specification breaks down the machine description into six types of information, denoted by the labeled dashed boxes in Figure 1. Format information, described in Section 4, specifies what operands are allowed by each type of operation. Resource usage information, described in Section 5, specifies how operations use the processor's resources as they execute. Latency information, described in Section 6, specifies how to calculate dependence distances between operations. Operation information, described in Section 7, specifies the operations supported by the architecture and describes them in terms of format, resources usage, and latency information. Register information, described in Section 8, specifies how registers overlap and, in IMPACT's description, the information necessary to do register allocation. Compiler-specific information, described in Section 9, specifies any other information that the compiler needs in the machine

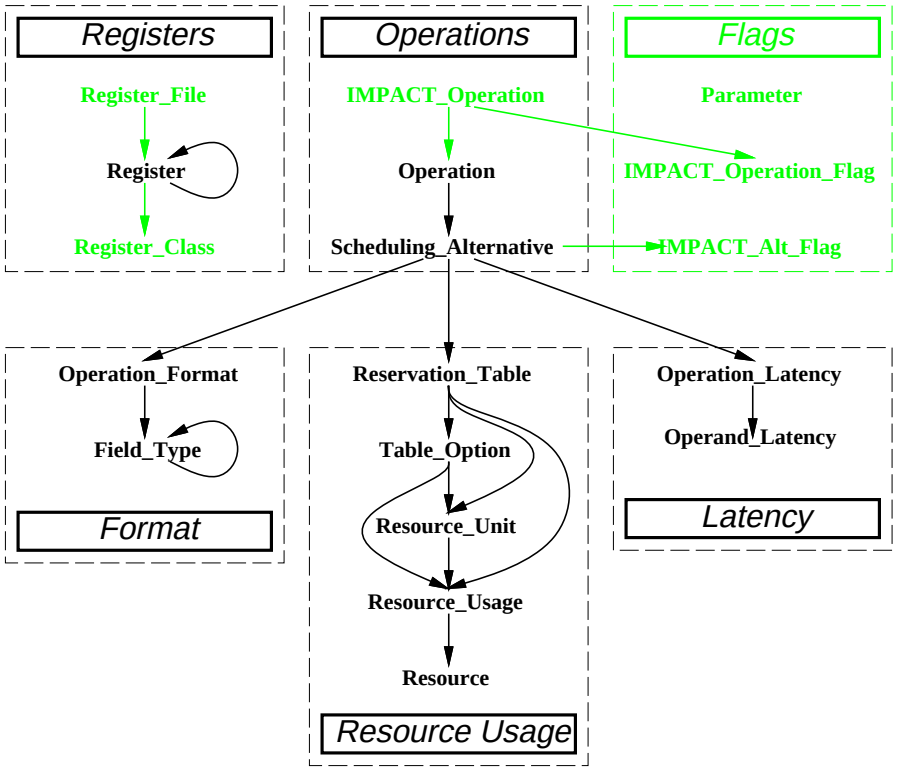

Figure 1: Structural overview of the machine description. Bold section names and arrows indicates compiler-independent features. Highlighted section names and arrows indicates IMPACT-dependent features.

description.

The examples in this document are taken from a prototype machine description of the PA-7100 based on the this document's specifications. The compiler-independent section definitions, which should be included at the top of all compiler-independent machine descriptions, can be found in "structure_pristine.hmdes2" shown in Appendix A. The compiler-independent description of the PA-7100 based on these definitions can be found in "PA_7100_pristine.hmdes2" shown in Appendix B.

The IMPACT-dependent section definitions, which should be included in all IMPACT machine descriptions files after including the pristine machine description file, can be found in "structure_IMPACT.hmdes" shown in Appendix C. Appendix D, "PA_7100_IMPACT.hmdes2", contains the IMPACT-dependent description of the PA-7100, which includes and expands upon the compiler-independent description.

2 The MD Language

The MD language, used in conjunction with the the MD libraries and tools, represents structured information in a way that is easy to read, modify, and write, for both a compiler and the compiler's users. It is a powerful language, designed so that it can be used to write complex machine descriptions, as described in Sections 4-9. It also has features to facilitate a hierarchical specification of structured information, where each level in the hierarchy can be placed in a separate MD file. These features are useful when writing machine descriptions because they allow compiler-specific machine descriptions to be built on top of compiler-independent machine descriptions, and they allow variations of an architecture to be built on top of the base architecture's machine description. These features also allow MD declarations defining the machine description's overall structure to be placed in a separate file that then can be included in all the machine descriptions. Lastly, the MD language supports powerful preprocessing directives, described in Section 3, that facilitates concise representation of structured information.

Structured information that is being represented in the MD language is first broken down into sections, based on a high-level classification of the information being represented. For example, a machine description's information might be broken down so that one section contains the description of the processor's resources, another the description of the processor's instruction set, a third the description of the processor's registers, and so on. Then, the information represented in each section is broken down into individual entries. For example, in a machine description, there might be an entry for each resource in the processor, for each assembly instruction in the processor, and for each of the processor's registers. Lastly, information about each entry is broken down into MD fields, based on a low-level classification of the information. For example, entries describing processor registers might use one field to specify the register's width and another field to specify the name of the registers that overlap this register. The MD fields are declared in each section to allow type checking to be done.

The rest of this section describes the details of representing structured information in the MD language. Section 2.1 describes how MD sections are created and referenced. Section 2.2 describes how to declare each section's MD fields. Section 2.3 describes how the add new MD entries and how to add additional information to existing MD entries. Section 2.4 describes how to modify the contents of fields that already have been specified.

2.1 Creating New Sections And Referencing Existing Sections

The command to create a new section with the name 'section_name' has the following form:

```
CREATE SECTION section_name
{
}
```

Each section may only be created once. To reference an existing section, remove the 'CREATE' keyword. Field declarations (described in Section 2.2) are placed between the section name and the opening curly brace ('{'). The section's entries (described in Section 2.3) are placed between the opening ('{') and closing ('}') curly braces for the section. The section_name must be a legal C identifier, where a legal C identifier starts with a letter or an underscore (_) and may only contain letters, numbers and underscores.

Example 2.1.1: In Figure 2, the section 'usage' is first created, then referenced again later by removing the 'CREATE' keyword. Field declarations, entry creations, and entry modifications may be done when the section is created and/or referenced, as illustrated by the comments in the figure.

```
// Create the section 'usage'
CREATE SECTION usage
   // Field declarations may go here
{
   // Entries may be created and/or modified here
}

// Reference the existing section 'usage'
SECTION usage
   // Additional field declarations may go here
{
   // Entries may be created and/or modified here
}
```

Figure 2: An example of creating and referencing a section, described in Example 2.1.1.

2.2 Declaring Fields For New And Existing Sections

Field declarations are placed between the section_name and the opening '{' when creating new or referencing existing sections, as was just illustrated in Example 2.1.1. Note that field declarations may be added to a section even after entries for that section have been made.

Each field declaration consists of a field type (REQUIRED, OPTIONAL, or IGNORED), the field's name, and a declaration of the expected contents of the field (a list of zero or more element types (INT, FLOAT, STRING, or LINK(section1|section2|...|sectionN))). The form of a field declaration is as follows:

```
FIELD_TYPE field_name (ELEMENT_TYPE1 ELEMENT_TYPE2 ... ELEMENT_TYPEn);
```

The last element type may be Kleene starred (an '*' after the element type), if 0 or more element of that type are allowed. This is useful when a field may contain a variable length list of elements.

The field types are used indicate how the fields should be handled by MD compiler. The supported three field types are:

1. REQUIRED: Indicates that the contents of this field is required to be specified for every entry in this section, after all the MD commands in the file have been processed. Note that required fields to not have to be specified when an entry is created, as long as they are specified before the end of processing.
2. OPTIONAL: Indicates that the contents of this field is not required to be specified for every entry in this section.
3. IGNORED: Indicates that, if the contents for this field is specified, that it will be ignored. This last field type is useful for "commenting out" fields out of an existing MD file. This feature is not currently implemented but will be soon.

The element types are used to specify the type of information may be placed in the field. The four element types supported are:

1. INT: An integer is expected. Decimal, hex and octal bases are supported. For hex, prefix with '0x'. For octal, prefix with '0'.
2. FLOAT: An floating point number is expected. C-style scientific notation is supported.
3. STRING: A 'single quoted string', a "double quoted string", or a whitespace delimited string, that does not contain an unbackslashed MD delimiter (( ) [ ] { } < > ; ' " * ! |), is expected. A string may not contain newlines, or be specified across multiple lines. Quotes, backslashes, and other special characters can be included in a string by proceeding the special character by a backslash. In the rare cases where the information type is variable, the STRING type can be used to get a string version of the information to the compiler which it can convert to the desired type.
4. LINK(section1|section2|...|sectionN): The name of an entry, from one of the sections specified in the declaration, is expected. The name must also uniquely identify an entry in one of the sections specified, ambiguous links are not allowed. In addition, an entry must be created in the MD file before it can be referenced by a link (no forward references are allowed). After the entire MD file is processed, the information is scanned for ambiguous links caused by entries created after the link was made.

Example 2.2.1: In Figure 3, a section 'example' is created that has three fields declared. The first field, 'number', is a required field, and expects exactly one integer to be specified for it. The second field, 'list', is also a required field, and expects zero or more integers to be specified (the INT is Kleene starred). The

third field, 'list2', is an optional field, and expects one or more integers to be specified. Note that specifying a field's contents to be nothing (an empty list), is not equivalent to leaving the field's contents unspecified. Thus the 'list' field is required to be specified, even if contents of this field is set to be nothing. Also, the 'list2' field is not required to be specified for any of the entries, but if it is, the field's contents must be specified to be a list of one or more integers.

```
CREATE SECTION example
  REQUIRED number(INT);
  REQUIRED list(INT*);
  OPTIONAL list2(INT INT*);
{
}
```

Figure 3: An set of example field declaration, described in Example 2.2.1.

Example 2.2.2: In Figure 4, two sections, 'example1' and 'example2', are created and several fields are declared using element types other than INT. The only field declared in the example1 section, 'subproblems', is an optional field, and allows zero or more entry names from the example1 section to be specified. This type of field is useful when an entry can be related to other entries in a single section.

The first field declared in the example2 section, 'examples', is a required field, and allows zero or more entry names from either the example1 section or the example2 section to be specified. This type of field is useful when an entry can be related to more than one section, in a flexible way. The second field declared in the example2 section, 'misc', is an optional field, and expects three elements of information, an int, a float, and a string, to be specified.

```
CREATE SECTION example1
  OPTIONAL subproblems(LINK(example1)*);
{
}

CREATE SECTION example2
  REQUIRED examples(LINK(example1|example2)*);
  OPTIONAL misc(INT FLOAT STRING);
{
}
```

Figure 4: An more complex set of field declaration, described in Example 2.2.2.

2.3 Creating New Entries And Modifying Existing Entries

Directives to create or modify existing entries for a section are placed between the section's open '{' and close '}'. An entry is created or modified using directives with the following form:

```
entry_name (field1_name(field1_contents) ...
            fieldN_name(fieldN_contents));
```

The entry's name must be either a valid C identifier or a quoted string (the quotes are not considered part of the entry's name). The first time an entry's name is encountered in a section, an entry with that name is created. The contents of some, or all, of that entry's fields can then be specified between that entry's opening '(' and closing ');'. The contents of a field is specified by using directives of the form `field_name(field_contents)`, as shown above.

An entry may be modified later by specifying the entry's name again in the section. The contents of previously unspecified fields can be then specified, or the contents of previously specified fields can be modified, using the directives described in Section 2.4.

Example 2.3.1: In Figure 5, two entries, 'usage1' and 'usage2', are described. The first entry, 'usage1', has the contents for fields 'time1' and 'time2' specified when it is created. To show how entry information can be specified in a piecemeal fashion, the second entry, 'usage2', is first created with nothing about its fields specified. Then, on the next line, the contents of 'time2' is specified. Later, when the section 'usage' is referenced again, the contents of 'time1' is finally specified. As mentioned in Section 2.2, even though 'time1' is a required field, it does not need to be specified right away.

```
CREATE SECTION usage
  REQUIRED time1(INT*);
  OPTIONAL time2(INT*);
{
     usage1 (time1(1) time2(3 4));
     usage2 ();
     usage2 (time2(8 9));
}

SECTION usage
{
     usage2 (time1(7))
}
```

Figure 5: An example of entry creation and modification, described in Example 2.3.1.

2.4 Modifying Field Values

The contents of an already specified field can be modified using two different field modifiers. The first field modifier, the concat modifier '||', has the following form:

```
field_name||(element1 element2 ... elementN)
```

The concat modifier causes element1 thru elementN to be appended to the elements already specified for that named field. This modifier may only be used with fields declared with a Kleene star. The concat modifier may be used on an unspecified field, if that field allows zero or more elements to be specified.

Example 2.4.1: In Figure 6, the contents of the field 'times' is first specified as a list of one number, '1'. Then, using the concat modifier, the contents of 'times' field becomes a list of three numbers, '1 2 3'.

```
CREATE SECTION usage
  REQUIRED times(INT*);
{
     usage1 (times(1))
     usage1 (times||(2 3));
}
```

Figure 6: An example of concatenating to a field's contents, described in Example 2.4.1.

The second modifier is the replace field modifier '!' and it has the following form:

```
field_name!(element1 element2 ... elementN)
```

This modifier replaces the contents of the field with the new contents specified. This modifier may be used even if the field was previously unspecified.

Example 2.4.2: In Figure 7, the contents of 'times' is first specified as a list of one number, '1'. Then, using the replace modifier, the contents of 'times' is then specified as a list of two numbers, '2 3'.

```
CREATE SECTION usage
  REQUIRED times(INT*);
{
     usage1 (times(1));
     usage1 (times!(2 3));
}
```

Figure 7: An example of replacing a field's contents, described in Example 2.4.2.

3 The MD Preprocessor

The MD preprocessor was designed to facilitate the writing of machine descriptions in the MD language. It is based on the C preprocessor and the Unix C shell language. It is a fully recursive implementation, allowing the nesting of preprocessor directives.

3.1 Comment Handling

The MD preprocessor strips out both nested C-style comments (/* */) and C++-style line comments (//). Comments are removed before any other processing is done, so it is safe to use a C++-style comment in a text replacement directive (unlike in the C++ preprocessor).

3.2 File Inclusion

To include the text of another file into the file being preprocessed, the include directive shown below is used:

```
$include "name_of_file_to_include"
```

The $include directive must be placed at column 0, and nothing else may be placed on that line.

3.3 Text Replacement

The $def text replacement directive has the following two forms:

```
$def def_name def_value
$def def_name {bounded_def_value}
```

where the def_name must be a valid C identifier. To make redefinitions of def_name easier, text replacement is suppressed for def_name in the $def directive. If it is desired that def_name be replaced before the $def directive is processed, the 'force replacement' directive (${def_name}, described later) can be used.

In the first form, def_value is taken to be everything between def_name and the end of the line, with leading and trailing whitespace removed. This form is analogous to the C #define directive.

In the second form, bounded_def_value is taken to be everything between the '{' and '}'. The bounded_def_value may include newlines and matched pairs of curly brackets {}. Single curly brackets should be backslashed.

Just like in the C preprocessor, text replacement occurs on any clearly delimited name that has been defined to a value. Also, just like in Unix C shell scripts, text replacement can occur in double quoted text but text replacement is prevented between single quoted text. The MD language (Section 2) treat both types

of quoted strings identically, so the only diference is in how the contained text is preprocessed. Newlines may not occur in quoted text.

As with the C preprocessor, text replacement directives may be issued on the MD preprocessor command line with an -Dname=value. Also, as with Unix C shell scripts, Unix environment variables may be accessed, with the force replacement directive ${name} described a little later. Command line directives override directives in the text, and both command line and text directives override the unix environment variables.

Example 3.3.1: The two different forms of the $def directive, and the effect of quotes, are shown in Figure 8.

```
$def NAME1 name1 text
$def NAME2 { name2 text }
"NAME1"                                  "name1 text"
"NAME2"                                  " name2 text "
'NAME1'                                  'NAME1'

(a) Before Preprocessing                  (b) After Preprocessing
```

Figure 8: A example using the two forms of the $def text replacement directive. The effects of single and double quotes on text replacement is also shown.

Text replacement can be forced, or unix environment variables can be accessed, using the 'force replacement' directive, which has the following form:

```
${name}
```

MD preprocessor command line text replacement directives have highest precedence, so 'name' is defined on the command line, that replacement text will be used. If 'name' is not defined on the command line, but in the text, that replacement text will be used. Otherwise, if 'name' is an Unix environment variable, the variable's value will be used. If 'name' is not defined anywhere, then an error will occur.

Example 3.3.2: The 'force replacement' directive is useful for concatenating text, as shown in Figure 9.

```
$def OP_NAME   sub
OP_${OP_NAME}                            OP_sub

(a) Before Preprocessing                  (b) After Preprocessing
```

Figure 9: A example using the force replacement directive ${name}.

Text replacement can be prevented for a name using the 'prevent replacement' directive which has the following form:

```
$\name
```

Example 3.3.3: The 'prevent replacement' directive is useful determining what a name is defined to, as shown in Figure 10.

```
$def I      1
$\I = I                                 I = 1
$def I    2
$\I = I                                 I = 2

(a) Before Preprocessing                (b) After Preprocessing
```

Figure 10: A example using the prevent replacement directive $\name.

A text replacement directive can be nullified using the $undef directive, which has the following form:

```
$undef name
```

Example 3.3.4: The $undef directive can be used to prevent replacement for the rest of the text, as shown in Figure 11.

```
$def I      3
$\I = I                                 I = 3
$undef I
$\I = I                                 I = I

(a) Before Preprocessing                (b) After Preprocessing
```

Figure 11: A example using the $undef directive.

It is often useful in conditional constructs (Section 3.5) to be able to determine if a name has been defined. The 'is defined?' directive serves this purpose and has the following form:

```
$?{name}
```

Example 3.3.5: The text if Figure 12 shows what $?{name} is replaced with when name is not defined, and when it has been defined to a value.

```
Before $\I defined = $?{I}          Before I defined = 0
$def I      10
After $\I defined = $?{I}           After I defined = 1

(a) Before Preprocessing            (b) After Preprocessing
```

Figure 12: A example showing the two possible values substituted in for the $?{name} directive.

3.4 Integer Expression Evaluation

The form of the directive to replace an integer expression with its integer value is:

```
$={expr}
```

where expr is any integer C-style expression. The operators allowed in expressions and their relative precedence level is shown in Figure 13.

```
( )                         <- Highest precedence
! ~  +(unary) -(unary)
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||                          <- Lowest precedence
```

Figure 13: The relative precedence of $={expr} operators.

Example 3.4.1: A example calculation of two values using $={expr} is shown in Figure 14

```
$={5 + -1}                          4
$={((10 * 5) + 2)/5}                10

(a) Before Preprocessing            (b) After Preprocessing
```

Figure 14: A example using $={expr} directive.

3.5 Conditional Inclusion

Text may be conditionally included, and preprocessor directives may be conditionally executed, using the following "conditional inclusion" directives:

```
$if(cond){body}
$elif(cond){body}
$else{body}
```

Zero or more $elif directives may be used after an $if directive, and the $else directive is optional.

Example 3.5.1: In Figure 15, conditional inclusion directives are used to determine if 'NUM' has been defined to '1'. In Figure 16, conditional inclusion directives are used first to set 'NUM' to '5', if it has not already been defined to a value, and then to determine which of three ranges the value of 'NUM' falls into.

```
$def NUM 5
$if(NUM == 1) {one}                    not one
$else {not one}
```

(a) Before Preprocessing (b) After Preprocessing

Figure 15: A simple example using conditional inclusion directives, described in Example 3.5.1.

```
Before $\NUM = NUM;                    Before NUM = NUM;
$if(!$?{NUM})                          (Defining NUM)
   {(Defining NUM) $def NUM {5}}       After NUM = 5;
After $\NUM = NUM;                     NUM > 4
$if(NUM == 1) {$\NUM = 1}
$elif(NUM < 4) {$\NUM < 4}
$else{$\NUM > 4}
```

(a) Before Preprocessing (b) After Preprocessing

Figure 16: An more complex example using conditional inclusion directives, described in Example 3.5.1.

3.6 Range Expansion

A range of signed integers can be expanded into a list of numbers using the "range expansion" directive, which has the following form:

```
$x..y
```

The placeholders x and y may be replaced with either a signed integer, or a parathesized expression '`(expr)`'.

Example 3.6.1: An example using range expansion directive is shown in Figure 17.

`$2..-2`	`2 1 0 -1 -2`
`$(5+1)..(5+5)`	`6 7 8 9 10`
`$1..(2-1)`	`1`
`(a) Before Preprocessing`	`(b) After Preprocessing`

Figure 17: Example uses of the range expansion directive.

3.7 Looping Over Text

The format of the $for directive is the following:

```
$for (def_name in value1 value2 ... valueN) {for_body}
```

where for_body will be replicated for each value given and def_name will be defined to this value.

The values may be specified in one of the following ways:

- A string containing letters, numbers, '_', '.', '+', and '-'. No whitespace allowed.
- A quoted string.
- or a string bounded by {}.

The for_body will be replicated exactly as specified between the bounding {}. All whitespace, including newlines, will be replicated. A $for loop that iterates over the values '1', '3', and '2' is shown in Figure 18. A $for loop that illustrates how the list of values is parsed is shown in Figure 19.

```
$for (I in 1 3 2) {$\I=I }          I=1 I=3 I=2

(a) Before Preprocessing            (b) After Preprocessing
```

Figure 18: An example using the $for directive.

```
$for (I in "text 1" {text 2} text 3)     "text 1"
{I                                       text 2
}                                        text
                                         3

 (a) Before Preprocessing                (b) After Preprocessing
```

Figure 19: An example using the $for directive that illustrates how the value list is parsed.

3.8 Preprocessor Output

Like the C preprocessor, the preprocessor by default, outputs $line directives that tell the MD compiler where all the preprocessed text came from. These $line directives may be turned off with a command line option.

3.9 Preprocessor Directive Precedence

The order that the various preprocessor tasks are performed on each piece of text is shown below:

1) comment removal

2) text replacement

3) expression calculation

4) range expansion

5) other directives, in the order encountered.

This order allows the preprocessor directives to be fully utilized while preventing some unintuitive combinations. For example, this order allows a $for directive to contain a range expansion that uses expressions containing defined text and comments, but prevents '`NAME$={1+1}`' from being replaced with the defined value of '`NAME2`'.

This concludes the description of the MD language and preprocessor. The rest of this document describes how a compiler-independent machine description and a compiler-specific machine description, with

the structure shown in Figure 1, can be written in the MD language.

4 Format Information

The machine description's format information describes what operands are supported by each of the architecture's operations. Two MD sections, Field_Type and Operation_Format, are used to represent this information. The fields declared in each of these MD sections, and how these sections are linked together, is shown in Figure 20.

Field_Type entries are used to describe the types of operands that may be placed in the operand fields of an assembly operation. Operation_Format entries are used to to describe the operation formats supported by the processor, in terms of field types supported by each the operation's operands. The rest of this section describes each of these MD sections in detail and provides examples of how these MD sections are used.

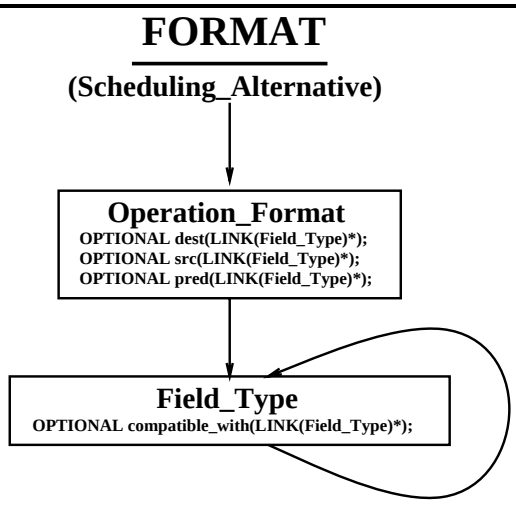

Figure 20: Structural overview of the machine description's format information.

4.1 Field_Type

The various types of operands (a register, a five bit literal, part of an label, etc.) that may be placed in an assembly operation are enumerated in the 'Field_Type' section. The MD language definition for this section

is shown below:

```
CREATE SECTION Field_Type
  OPTIONAL compatible_with(LINK(Field_Type)*);
{
}
```

The 'compatible_with' field is used specify other operand types that may be placed in a field of that type (I.e., a five bit literal can be placed in a field big enough for an 11 bit literal). By convention, the customizer and/or compiler is expected to use the transitive property of the 'compatible_with' relationship to 'fill out' this field, since the mdes version 1 language used this transitive property. In general, the compiler-independent machine description should provide as much detail as possible about field types and operation formats (Section 4.2), and restrict abstractions to the compiler-specific part.

Example 4.1.1: The PA-7100 supports several literal widths, including several which contain the high or low bits of a label. One way to model the field types supported by the PA-7100 are shown in Figure 21. By convention, the 'NULL' field type specifies that no operand is allowed in that field. Figure 22 shows this section after a mdes customizer uses the transitive property to fill out 'compatible_with', so that 'compatible_with' field explicitly lists all the field types each entry is compatible with.

```
SECTION Field_Type
{
   NULL   ();
   REG    ();
   Lit5   ();
   Lit11  (compatible_with(Lit5));
   Label  ();
   Lit12  (compatible_with(Lit11 Label));
   Lit14  (compatible_with(Lit12));
   Lit17  (compatible_with(Lit14));
   Lit21  (compatible_with(Lit17));
   any    (compatible_with(Lit21 REG NULL));
}
```

Figure 21: An example Field_Type section for the PA-7100, described in Example 4.1.1.

4.2 Operation_Format

The formats of the assembly-level operations supported by the architecture are enumerated in the 'Operation_Format' section (I.e, one operation might require that the dest be a register, that src0 be a register,

```
SECTION Field_Type
{
   NULL   ();
   REG    ();
   Lit5   ();
   Lit11  (compatible_with(Lit5));
   Label  ();
   Lit12  (compatible_with(Lit11 Label Lit5));
   Lit14  (compatible_with(Lit12 Lit11 Lit5 Label));
   Lit17  (compatible_with(Lit14 Lit12 Lit11 Lit5 Label));
   Lit21  (compatible_with(Lit17 Lit14 Lit12 Lit11 Lit5 Label));
   any    (compatible_with(Lit21 REG NULL Lit17 Lit14 Lit12 Lit11 Lit5 Label));
}
```

Figure 22: The example Field_Type section after a customizer expands 'compatible_with' as described in Example 4.1.1.

and that src1 be a 11 bit literal). The MD language definition for this section is shown below:

```
CREATE SECTION Operation_Format
  OPTIONAL dest(LINK(Field_Type)*);
  OPTIONAL src(LINK(Field_Type)*);
  OPTIONAL pred(LINK(Field_Type)*);
{
}
```

The fields 'dest', 'src', and 'pred' list the field type expected for each destination operand, source operand, and predicate operand, respectively. By convention, if a field type is not specified for an operand (or the entire field is not specified), then nothing may be specified for those operands, which is equivalent to the fields being padded with the 'NULL' field type. This convention requires special treatment of these fields by the customizer and/or compiler, perhaps by padding unspecified fields with the 'NULL' field type.

Example 4.2.1: The PA-7100 has three standard formats for arithmetic operations, which can be specified as shown in Figure 23. As shown in this figure, these arithmetic operations may not be predicated, and may have only two sources and one destination. The entries 'OF_Std2' and 'OF_Std3' model the cases where a 11 bit literal is required in src0 or src1, respectively.

5 Resource Usage Information

The machine description's resources usage information describes the processor's resources and the ways that operations can use these resources as they execute. Five MD language sections (Resource, Resource_Usage,

```
SECTION Operation_Format
{
  OF_Std1  (dest(REG) src(REG REG));
  OF_Std2  (dest(REG) src(Lit11 REG));
  OF_Std3  (dest(REG) src(REG Lit11));
}
```

Figure 23: An example Operation_Format section enumerating the three standard arithmetic operation formats in the PA-7100, described in Example 4.2.1.

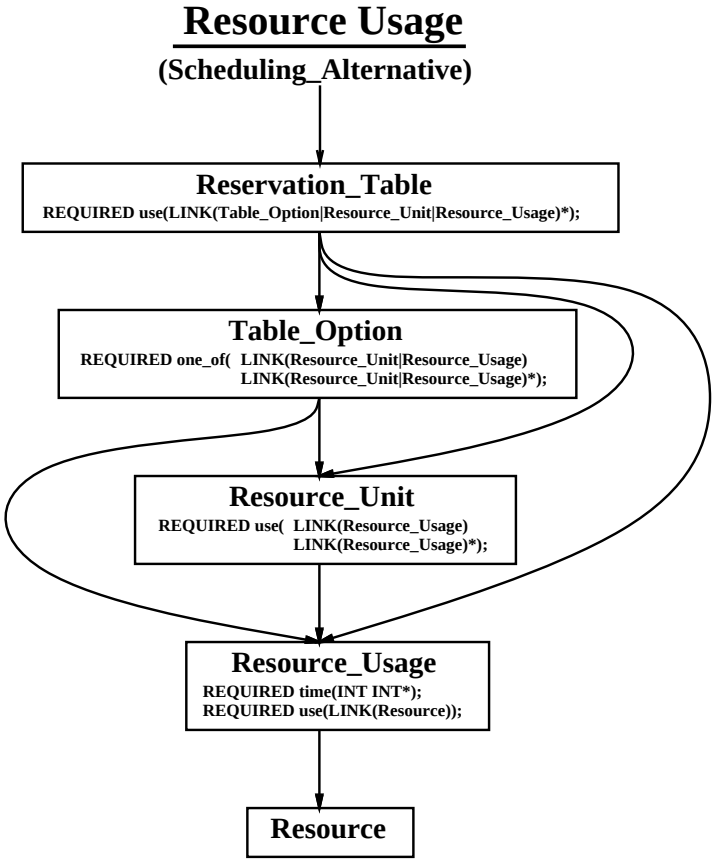

Figure 24: Structural overview of the machine description's resource usage information.

Resource_Unit, Table_Option, and Reservation_Table) are used to represent this information. The fields declared in each of these MD language sections, and how these sections are linked together, is shown in Figure 24.

The Resource section's entries are used to enumerate the processor's resources. The Resource_Usage

section's entries describe the times that these resources can be used. The Resource_Unit section's entries group these resource usages into more intuitive units, such as function units. The Table_Option section's entries describe the cases where one of a set of resources may be used. The Reservation_Table section's entries describe the way an operation can use the processor's resources as it executes. The rest of this section describes each of these MD sections in detail, and provides examples of how these MD sections are used.

5.1 Resource

The resources used as operations execute are enumerated in the 'Resource' section (such as issue slots, stages in the ialu and falu, etc.). The MD language specification of this section is shown below:

```
CREATE SECTION Resource
{
}
```

There are no compiler-independent fields declared for this section since the main purpose of this section is to enumerate the processor's resources. By convention, the resource names do not have any implicit meaning, they just identify the resource for later use in the machine description. If an compiler wishes to associate some meaning with these resources, such as which resources correspond to ialu pipeline stages, extra compiler-specific fields should be added to this section for this purpose.

```
SECTION Resource
{
   slot0    ();    // Instruction issue slots
   slot1    ();
   fissue   ();    // Only one floating point operation/cycle
   ialu     ();
   falu_s0  ();
   falu_s1  ();
   fmul_s0  ();
   fmul_s1  ();
   fmulfake ();    // Used to model interlock with divide
   fdiv     ();
   mem      ();
}
```

Figure 25: An example Resource section enumerating the resources that can be used to model the PA-7100's resources, described in Example 5.1.1.

Example 5.1.1: A set of resources that can be used to model the PA-7100 is shown in Figure 25. The PA-7100 can issue up to two operations per cycle (two 'issue slots', modeled by slot0 and slot1) but is limited to one floating point operation and one integer operation per cycle (loads and stores are considered integer operations). The constraint of one integer operation per cycle is modeled by the presence of only one integer alu (ialu). There are however, several floating point function units (falu_sX, fmul_sX, fdiv) in the PA-7100, so the resource 'fissue' is used to model the limit of one float operation per cycle.

The floating point alu and multiply units are fully pipelined (depth two), so both stages are modeled as separate resources (falu_s0, falu_s1, fmul_s0, and fmul_s1). The floating point divide is not pipelined, so only one resource is used to model it (fdiv). The floating point divide also shares an unspecified resource with the floating point multiply unit that can cause the multiply to interlock for a cycle when the multiply issues several cycles after a divide. This unspecified shared resource is modeled by 'fmulfake'.

The PA-7100's memory unit is modeled by 'mem' and loads use both this unit and the integer alu for 1 cycle. Stores use also the memory unit, but for two cycles, and the integer unit for one cycle. Because loads and stores use the memory unit differently, it was necessary to model this memory resource.

5.2 Resource_Usage

Resources are used at distinct times as operations execute. For example, the floating point add operation uses the first stage of the floating point alu one cycle before it uses the second stage. Because all these resource usage times are relative, fixing time 0 at any point in the execution of the operation is valid. Since scheduling is usually centered around modeling resources in the execution unit, time 0 is fixed, by convention, to be the first cycle an operation spends in the execution stage (i.e., the time the operation enters a function unit). Thus, the floating point add operation is described as using the first stage of the floating point alu (falu_s0) at time 0 and the second stage (falu_s1) at time 1.

The times that resources maybe used as operations execute are enumerated in the 'Resource_Usage' section. The MD definition of this section is shown below.

```
CREATE SECTION Resource_Usage
  REQUIRED use(LINK(Resource));
  REQUIRED time(INT INT*);
{
}
```

Each Resource_Usage entry associates a resource 'use' with a list of one or more usage times. As mentioned

above, the first cycle in the execution unit is considered time 0. By convention, resources used to constrain the number and type of operations issued (such as slot0, slot1, and fissue) are also used at time 0.

Example 5.2.1: The Resource_Usage entries that can be used to model the PA-7100 shown in Figure 26. The naming convention used prefixes all Resource_Usage entry names with "RU_", and if the resource is used at more than one time, postfixes the entry names with the time used (such as "_t0") or the range of times used (such as "_t0_14").

```
SECTION Resource_Usage
{
   RU_slot0            (use(slot0) time(0));
   RU_slot1            (use(slot1) time(0));
   RU_fissue           (use(fissue) time(0));

   RU_ialu             (use(ialu) time(0));

   RU_falu_s0          (use(falu_s0) time(0));
   RU_falu_s1          (use(falu_s1) time(1));

   RU_fmul_s0_t0       (use(fmul_s0) time(0));
   RU_fmul_s0_t6_7     (use(fmul_s0) time(6 7));
   RU_fmul_s0_t13_14   (use(fmul_s0) time(13 14));
   RU_fmul_s1          (use(fmul_s1) time(1));

   /* fmulfake used to model the interlocks between fmul and fdiv */
   RU_fmulfake_t0      (use(fmulfake) time(0));
   RU_fmulfake_t5      (use(fmulfake) time(5));
   RU_fmulfake_t12     (use(fmulfake) time(12));

   RU_fdiv_t0_7        (use(fdiv) time($0..7));
   RU_fdiv_t0_14       (use(fdiv) time($0..14));

   RU_mem_t0           (use(mem) time(0));
   RU_mem_t0_1         (use(mem) time(0 1));
}
```

Figure 26: An example Resource_Usage section, described in Example 5.2.1.

5.3 Resource_Unit

It is often desirable to group a set of resource usages together to form more intuitive units, such as function units or issue units. This grouping is accomplished in the machine description using the 'Resource_Unit'

section. The MD definition of the this section is shown below:

```
CREATE SECTION Resource_Unit
  REQUIRED use(LINK (Resource_Usage) LINK(Resource_Usage)*);
{
}
```

Each Resource_Unit entry groups together one or more Resource_Usage entries. A recommended usage is to describe the units available in each major stage in the processor's pipeline. In example 5.3.1, units are described for the issue stage (namely floating point issue), and the for the execution stage (all the function units, which may overlap in resource usage).

Example 5.3.1: One way to group Resource_Usage entries for the PA-7100 is shown in Figure 27. The entries 'fissue0_unit' and 'fissue1_unit' are used to model the two possibilities for issuing floating point operations (in slot0 or slot1). Both Resource_Unit entries use the fissue resource to model the constraint that only one floating point operation may be issued per cycle.

```
SECTION Resource_Unit
{
   fissue0_unit  (use(RU_slot0 RU_fissue));
   fissue1_unit  (use(RU_slot1 RU_fissue));
   ialu_unit     (use(RU_ialu));
   ibr_unit      (use(RU_ialu));
   falu_unit     (use(RU_falu_s0 RU_falu_s1));
   fmul_unit     (use(RU_falu_s0 RU_falu_s1 RU_fmul_s0_t0 RU_fmul_s1
                      RU_fmulfake_t0));
   fmulI_unit    (use(RU_falu_s0 RU_falu_s1 RU_fmul_s0_t0 RU_fmul_s1));
   fdivS_unit    (use(RU_fdiv_t0_7 RU_fmulfake_t5 RU_fmul_s0_t6_7));
   fdivD_unit    (use(RU_fdiv_t0_14 RU_fmulfake_t12 RU_fmul_s0_t13_14));
   load_unit     (use(RU_ialu RU_mem_t0));
   store_unit    (use(RU_ialu RU_mem_t0_1));
}
```

Figure 27: An example Resource_Unit section, described in Example 5.3.1.

The fmulfake resource is used by several Resource_Unit entries to model the one cycle interlock that occurs if a floating point multiply (fmul) is issued 6 cycles after a single-precision fdiv or 13 cycles after a double-precision fdiv. This interlock causes the fmul to take three cycles instead of the normal two cycles. The Resource_Unit entry fmul_unit models the two cycle fmul, and its use of fmulfake resource conflicts with the fdivS_unit and fdivD_unit, when the fmul should interlock. The fmulI_unit entry models the interlocked, three cycle fmul, and it doesn't use the fmulfake resource. In the Scheduling_Alternative entry for the fmul

(Example 7.1.1), the use of fmulI_unit will imply a three cycle latency and the use of fmul_unit will imply a two cycle latency.

5.4 Table_Option

It is often the case that operations have some flexibility in terms of the resources they use during execution. For example, there may be several function units capable of executing the operation. This flexible use of resources is modeled using Table_Option entries, where one of a set of Resource_Usage entries (described in Section 5.2) and/or Resource_Unit entries (described in Section 5.3) may be selected from. The 'Table_Option' section is described by the following MD declaration:

```
CREATE SECTION Table_Option
  REQUIRED one_of(LINK(Resource_Unit|Resource_Usage)
                  LINK(Resource_Unit|Resource_Usage)*);
{
}
```

Each table option entry consists of a list of Resource_Unit or Resource_Usage entry names in the 'one_of' field. By convention, the first entry in the list is the most desirable option, the second entry is the second most desirable option, and so on. A list consisting of one entry name in the 'one_of' field is allowed to facilitate parameterized machine descriptions. The allowing of a Resource_Unit or a Resource_Usage entry to be specified in the 'one_of' field (verses requiring only Resource_Unit entries) is to minimize the number of resource units that must be declared.

Example 5.4.1: The PA-7100 allows any ordering of the integer and floating point instructions. This flexibility can be modeled by the table options 'any_iissue' and 'any_fissue' shown in Figure 28. Other types of flexibility that can be modeled in this section includes available register ports, function units, and result buses.

```
SECTION Table_Option
{
   any_iissue    (one_of(RU_slot0 RU_slot1));
   any_fissue    (one_of(fissue0_unit fissue1_unit));
}
```

Figure 28: An example Table_Option section, described in Example 5.4.1.

5.5 Reservation_Table

The Resource_Usage entries, Resource_Unit entries, and Table_Option entries described in the last three sections are used in the 'Reservation_Table' section to build reservation tables. Each Reservation_Table entry describes how an operation can use resources over its entire execution. For example, a Reservation_Table entry for an integer alu operation might specify that one of the integer issue units and the only ialu must be used. The MD declaration for the 'Reservation_Table' section is shown below:

```
CREATE SECTION Reservation_Table
  REQUIRED use(LINK(Table_Option|Resource_Unit|Resource_Usage)*);
{
}
```

For each Reservation_Table entry, the Table_Option entries, Resource_Unit entries, and/or Resource_Usage entries that are used as the operation executes are listed in the 'use' field. In order to schedule the operation, the requirements of everything specified in the 'use' field must be met in order to schedule the operation. The 'use' field is allowed to be empty for pseudo operations like compiler directives, which can be specified in the compiler-specific part of the machine description and don't use any resources.

Example 5.5.1: The PA-7100's issue and execution stages may be modeled using the Reservation_Table entries shown in Figure 29. For the PA-7100, each reservation table entry needs to specify which resources are necessary to issue the operation (any_iissue or any_fissue), and what function unit resources are needed to execute the operation (ialu_unit, ibr_unit, etc.). Other architectures may require different resources to be modeled. For example, Sparc-based machines, also require modeling of the register read and write ports.

6 Latency Information

The machine description's latency information describes how to calculate dependence distances for dependencies between operations. Two MD language sections, Operand_Latency and Operation_Latency, are used to represent this information. The fields declared in each of these MD language sections, and how these sections are linked together, is shown in Figure 30.

The Operand_Latency section's entries are used to specify the times when operands can be used. The Operation_Latency section's entries are used to specify when an operation uses its operands. The rest of this section describes each of these MD sections in detail, and provides examples of how these MD sections are used.

```
SECTION Reservation_Table
{
   RT_INOP       (use(any_iissue ialu_unit));
   RT_IAlu       (use(any_iissue ialu_unit));
   RT_IBr        (use(any_iissue ibr_unit));
   RT_FPAlu      (use(any_fissue falu_unit));
   RT_FPMul2     (use(any_fissue fmul_unit));
   RT_FPMul3     (use(any_fissue fmulI_unit));
   RT_FPAluMul   (use(any_fissue fmul_unit));
   RT_FPDivS     (use(any_fissue fdivS_unit));
   RT_FPDivD     (use(any_fissue fdivD_unit));
   RT_Load       (use(any_iissue load_unit));
   RT_Store      (use(any_iissue store_unit));
}
```

Figure 29: An example Reservation_Table section, described in Example 5.5.1.

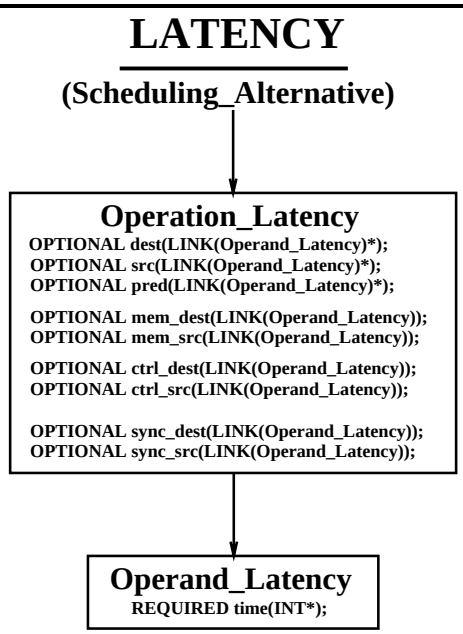

Figure 30: Structural overview of the machine description's latency information.

6.1 Operand_Latency

Operations use their operands at various times during execution. Source register operands are typically read just before entering the execution stage, and destination register operands are effectively written just as the operation exits the execution stage (assuming full bypass logic). By describing when operations use

their operands in the machine description, dependence distances can be calculated for the various flow, anti, and output dependences between operations. An example of how this information can be used to calculate dependence distances is presented in Section 6.2. The times that operands may be used are enumerated in the machine description's 'Operand_Latency' section, which has the following MD definition:

```
CREATE SECTION Operand_Latency
  REQUIRED time(INT*);
{
}
```

For each Operand_Latency entry, the times that the operand may be used is listed in the 'time' field. By convention, time 0 is fixed to be the moment just before an operation enters the execution stage (at the very end of the cycle before the execution stage is entered). Thus, time 1 is at the very end of the first cycle in the execution stage.

Typically, a single usage time is specified for each Operand_Latency entry, corresponding to when that operand is read or written. Multiple times may be specified in order to model the operand being read from or written to multiple times or at varying times (this is something not previously handled in IMPACT). The ability to specify an empty list in the 'time' field is provided to allow marking of operands that should not be read or written to at all.

Example 6.1.1: On way to enumerate the operand use times for the PA-7100 is shown in Figure 31. The PA-7100's integer alu operations write their result at time 1 (d1), loads and most floating point operations write their result at time 2 (d2), an interlocked floating point multiply writes its result at time 3 (d3), and the floating point divides write their result at time 8 (d8) or 15 (d15) depending on their precision. The PA-7100's operations typically read their source operands at time 0 (s0), but the store operation reads the data to be stored at time 2 (s2). The sync operands (ss0, sd0, and sd1) will be used and explained in Section 6.2.

6.2 Operation_Latency

In Section 6.1, it was described how operand use (read/write) times are enumerated in the 'Operand_Latency' section. Those operand use times are used by the 'Operation_Latency' entries to describe when an operation

```
SECTION Operand_Latency
{
   $for (N in 1 2 3 8 15)
   {
      d${N} (time(N));                  // Destination operands
   }

   $for (N in 0 2) {s${N} (time(N)); }  // Source operands

   $for (N in 0) {ss${N} (time(N)); }   // Sync Source operands

   $for (N in 0 1) {sd${N} (time(N)); } // Sync Dest operands

   x  (time());                         // For when operand should not be a register
}
```

Figure 31: An example Operand_Latency section, described in Example 6.1.1.

uses each of its operands. The MD definition for the 'Operation_Latency' section is shown below:

```
CREATE SECTION Operation_Latency
  OPTIONAL dest(LINK(Operand_Latency)*);
  OPTIONAL src(LINK(Operand_Latency)*);
  OPTIONAL pred(LINK(Operand_Latency)*);

  OPTIONAL mem_dest(LINK(Operand_Latency));
  OPTIONAL ctrl_dest(LINK(Operand_Latency));
  OPTIONAL sync_dest(LINK(Operand_Latency));

  OPTIONAL mem_src(LINK(Operand_Latency));
  OPTIONAL ctrl_src(LINK(Operand_Latency));
  OPTIONAL sync_src(LINK(Operand_Latency));
{
}
```

The fields 'dest', 'src', and 'pred' associate an Operand_Latency entry with each destination, source, and predicate operand of the operation, respectively. The Operand_Latency entries specify the time(s) each of those operands are used by the operation, and can be used to calculate dependence distances for all register flow, anti, and output dependences.

For example, to calculate the dependence distance for a register flow dependence between an operation that writes r1 at time 2 and an operation that reads r1 at time 0, the the read time 0 is subtracted from the write time 2 to get a two cycle dependence distance. If the flow dependent operation read r1 at time 2 instead of time 0, then a zero cycle dependence (read time 2 minus write time 2) would be calculated.

The remaining fields in the Operation_Latency section are used to calculate dependence distances for memory, control, and synchronization dependencies. The 'mem_dest' and 'mem_src' fields are used to calculate distances for memory dependences (dependences that preventing ambiguous or dependent stores from reordering with other loads and stores). The 'ctrl_dest' and 'ctrl_src' fields are used to calculate control dependence distances (dependencies preventing operations from moving above or below a branch). The 'sync_dest' and 'sync_src' fields are used to calculate synchronization dependence distances (dependences from operations that no other operations should move past).

By convention, it is an error if a dependence distance calculation attempts to use information that was left unspecified in a Operation_Latency entry.

Example 6.2.1: The dependence distances modeling the PA-7100 can be calculated from the Operation_Latency entries shown in Figure 32. Most of the PA-7100's operations have one destination. The two exceptions are memory operations (which support pre and post increments of an address register) and a set of special floating point operations that do two independent calculations with one assembly instruction.

All PA-7100 operations, except stores, read their source register just before entering the execution stage (time 0). Store operations read the register containing the value being stored at time 2. This design allows PA-7100 store operations to be issued in parallel with the operation that is producing the value being stored, if that operation has a latency of 1 or 2.

The PA-7100 does not allow a load to be issued until 2 cycles after the store is issued (presumably to allow memory-flow-dependent loads to get the correct data). This issue constraint is modeled with resources (in Figure 27), so using a 0 cycle mem flow dependence would be sufficient for correctness. But the distance is set to 2 in this example, so that the control flow graph more accurately models the situation, which helps scheduling heuristics. Note that only operations that write memory need to have the field 'mem_dest' specified, and only operations that read memory need to have the field 'mem_src' specified.

The PA-7100's issue logic prevents operations control dependent on the branch from issuing in the same cycle as the branch. A 1 cycle control flow dependence distance (specified by Operation_Latency entry OL_Branch) is used to model this constraint.

Although synchronization dependences are not used often, their function is to prevent any operation from moving past a synchronization operation. Typically, synchronization operations are those that change the processor or register state, such as the operation to rotate the register window in the Sparc architecture. (Side note: One common operation in SPEC'92 benchmarks that needs to be treated as a synchronization

```
SECTION Operation_Latency
{
   // Non-memory operations don't use mem_src or mem_dest
   // All operations may have ctrl or sync dependences.
   $for(N in 1 2 3 8 15)
   {
     OL_Lat${N}  (dest(d${N} x)    src(s0 s0 s0 s0)
                  ctrl_dest(sd0)   ctrl_src(ss0)
                  sync_dest(sd0)   sync_src(ss0));
   }
   // Some floating point operands with latency 2 have two destinations
   OL_Lat2        (dest!(d2 d2));

   OL_Branch      (dest(d1 x)      src(s0 s0 s0 s0)
                   ctrl_dest(sd1)  ctrl_src(ss0)    // 1 cycle ctrl flow dep
                   sync_dest(sd0)  sync_src(ss0));

   // Stores read the value to be stored two cycles after issue
   // They also may have two destinations for post-increment stores
   // Loads flow dependent on stores must follow store by at least 2 cycles.
   // Only mem_dest specified since stores only write memory
   OL_Store       (dest(d1 d1)     src(s0 s0 s2 s0)
                   mem_dest(sd2)                    // 2 cycle mem flow dep
                   ctrl_dest(sd0)  ctrl_src(ss0)
                   sync_dest(sd0)  sync_src(ss0));

   // Label stores store 4th operand instead of 3rd
   // Only mem_dest specified since stores only write memory
   OL_Label_Store (dest(d1 d1)     src(s0 s0 s0 s2)
                   mem_dest(sd2)                    // 2 cycle mem flow dep
                   ctrl_dest(sd0)  ctrl_src(ss0)
                   sync_dest(sd0)  sync_src(ss0));

   // Loads may have two destination.  Second dest for addr post-increment.
   // Only mem_src specified since loads only read memory
   OL_Load        (dest(d2 d1)     src(s0 s0 s0 s0)
                                   mem_src(ss0)
                   ctrl_dest(sd0)  ctrl_src(ss0)
                   sync_dest(sd0)  sync_src(ss0));
}
```

Figure 32: An example Operation_Latency section, described in Example 6.2.1.

operation is a jsr to 'setjmp' (or 'sigsetjmp'). The spec for setjmp states that all registers are saved but values in memory, such as, spilled registers are not. So in reality, any register value may be destroyed across a setjmp. The benchmarks are written so very few, if any, registers should be live across a setjmp, but

scheduling can cause a serious bug if code motion is allowed across a setjmp.)

7 Operation Information

The machine description's operation information describes the operations supported by the architecture and how they may be scheduled. Two compiler-independent MD language sections, Scheduling_Alternative and Operation, are used to represent this information. Additionally, other compiler-specific MD language sections and fields may needed to interface this information with the compiler. The IMPACT-specific MD language section IMPACT_Operation is used in this section to illustrate this. The fields declared in each of these MD language sections, and how these sections are linked together, is shown in Figure 33.

The Scheduling_Alternative section's entries are used to describe the ways operations can be scheduled in terms of their operands, the resources they use, and their dependence distance information. The Operation section's entries enumerate the assembly-level operations supported by the architecture and associate scheduling alternatives with each of these operation. The IMPACT-specific IMPACT_Operation section's entries associate the compiler's operations to the architecture's operations. The rest of this section describes each of these MD sections in detail, and provides examples of how these MD sections are used.

7.1 Scheduling_Alternative

A scheduling alternative is a set of three requirements that must be met in order to schedule an operation using that alternative (one or more scheduling alternatives are associated with an operation in Section 7.2). These three requirements that must be met in order to schedule an operation are: 1) The resources the alternative requires (specified by a Reservation_Table entry name) must be available, 2) the register, control, memory, and synchronization dependence distances for the alternative (specified by a Operation_Latency entry name) must be satisfied, 3) and the operation's operands must be compatible with the alternative's allowable operands (specified by one or more Operation_Format entry names).

These Scheduling_Alternative entries will be used in Section 7.2 to specify how each assembly-level oper-

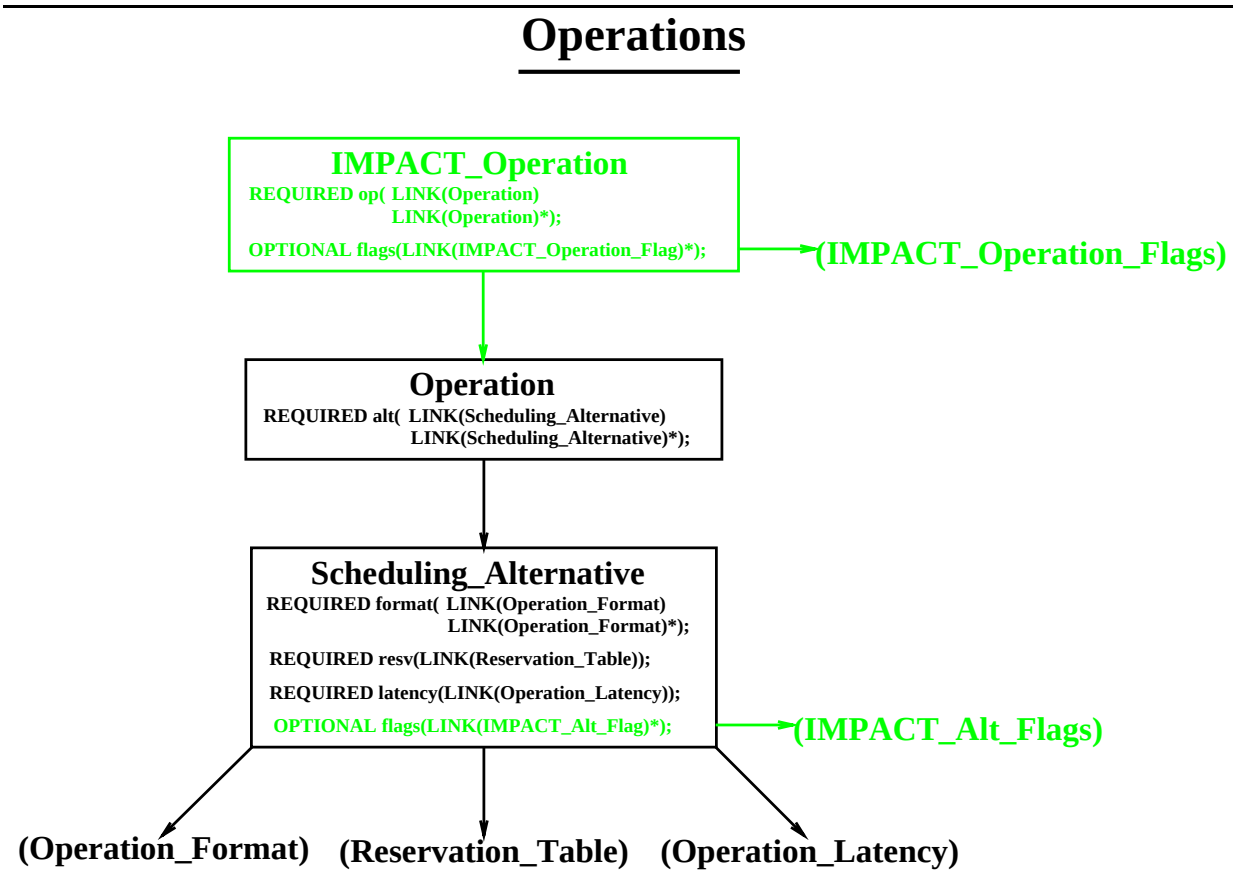

Figure 33: Structural overview of the machine description's operation information. Highlighted section names, field names, and arrows indicate IMPACT-specific features.

ation can be scheduled. The MD declaration for the 'Scheduling_Alternative' section is shown below:

```
CREATE SECTION Scheduling_Alternative
  REQUIRED format (LINK(Operation_Format) LINK(Operation_Format)*);
  REQUIRED resv (LINK(Reservation_Table));
  REQUIRED latency (LINK(Operation_Latency));
{
}
```

Often, additional compiler-specific information is needed about a Scheduling_Alternative entry. This compiler-specific information can be added in a separate compiler-specific file using MD language features. In IMPACT, a 'flags' field is added to the Scheduling_Alternative section for this reason (see Section 9.2 for details). This additional IMPACT-specific field has the flowing MD language declaration:

```
OPTIONAL flags (LINK(IMPACT_Alt_Flag)*);
```

Example 7.1.1: One way to model the scheduling alternatives for PA-7100's floating point multiply

and the PA-7100's load operations is shown in Figure 34. As mentioned in Example 5.3.1, the PA-7100's floating-point multiply normally takes 2 cycles, but can interlock and take 3 cycles when issued 6 cycles after a single-precision fdiv or 13 cycles after a double-precision fdiv. The 2 cycle, non-interlocking, floating-point multiply case is modeled by the Scheduling_Alternative entry 'ALT_FMul2' and the 3 cycles, interlocking, case is modeled by 'ALT_FMul3'. The two cycle and three cycle latencies are set by the Operation_Latency entry names 'OL_Lat2' and 'OL_Lat3', respectively. In Example 7.2.1, 'ALT_FMul2' will be given higher scheduling priority, so that 'ALT_FMul3' will be used only when an interlock occurs.

```
SECTION Scheduling_Alternative
{
   ALT_FMul2        (format(OF_Std1) resv(RT_FPMul2) latency (OL_Lat2));
   ALT_FMul3        (format(OF_Std1) resv(RT_FPMul3) latency (OL_Lat3));

   // Load operations
   ALT_LD_Short     (format(OF_Load1) resv(RT_Load) latency(OL_Load));
   ALT_LD_Long      (format(OF_Load2 OF_Load2_Reg OF_Load2_Label)
                     resv(RT_Load) latency(OL_Load));
   ALT_LD_Index     (format(OF_Load3) resv(RT_Load) latency(OL_Load));
   ALT_LD_Inc_Short (format(OF_Load4) resv(RT_Load) latency(OL_Load));
   ALT_LD_Inc_Long  (format(OF_Load5) resv(RT_Load) latency(OL_Load));
   ALT_LD_Inc_Index (format(OF_Load6) resv(RT_Load) latency(OL_Load));
}
```

Figure 34: An example Scheduling_Alternative section, described in Example 7.1.1.

The various Scheduling_Alternative entries for loads explicitly model the operand combinations allowed in the PA-7100's load operations. The exact operand combinations allowed depends on the type of data the operation loads. It will be shown in Example 7.2.1 that floating point loads support the fewest combinations, and that integer loads support the most. Note that all the load operation alternatives have the same resource usage and latency info, and could have been specified with just one alternative with a flexible instruction format. These variations were spelled out in order to more closely model the architecture and to provide a consistency check for the operation's operands.

7.2 Operation

Every assembly-level operation in the architecture is enumerated in the 'Operation' section, which has the following MD definition:

```
CREATE SECTION Operation
  REQUIRED alt(LINK(Scheduling_Alternative) LINK(Scheduling_Alternative)*);
{
}
```

An entry in created in the Operation section for each assembly-level operation. By convention, the entry's name should be the actual assembly mnemonic for each operation, with necessary adaptations when the mnemonic by itself is not specific enough (namely when part of the opcode is encoded into the operands or the operands' value).

As the above MD definition shows, each Operation entry requires the enumeration of one or more scheduling alternatives for the operation, in the 'alt' field. Through these scheduling alternatives, the operation formats supported, the resources the operation used during execution, and the operation's latency information are associated with the operation (see Section 7.1).

By convention, the first alternative in the 'alt' field should be given the highest scheduling priority, the second alternative should be given the second highest scheduling priority, etc. This priority convention is used extensively to model unusual interlock and bypass conditions in real architectures, such as the fmul and fdiv interlock in the PA-7100 and the ability of the SuperSparc to sometimes issue flow dependent instructions in the same cycle. Thus, it is important to take these priorities into account when scheduling (it is currently unclear how these priorities interact with some software pipelining and operation scheduling algorithms).

Example 7.2.1: One way to enumerate the PA-7100's floating-point multiply and load operations, in an Operation section, is shown in Figure 35. In this example, a variation of IMPACT's opcode names are used in this example instead of the PA-7100's assembly names for clarity purposes (the assembly names for loads are a little cryptic).

The Operation entries modeling the PA-7100's floating point multiplies, OP_MUL_F and OP_MUL_F2, list the two cycle alternative ALT_FMul2 first, giving that Scheduling_Alternative entry the highest scheduling priority. Due to ALT_FMul3's lower priority, support for the the same operation formats as ALT_FMul2, and the same read times for its source operands (thus dependence-wise, ready at the same time) as ALT_FMul2, ALT_FMul3 will only be scheduled when the fmulfake resource is not available (which indicates the fmul is

```
SECTION Operation
{
   // FP Mul takes 2 cycles, unless interlocks due to conflict with divide
   OP_MUL_F     (alt(ALT_FMul2 ALT_FMul3));
   OP_MUL_F2    (alt(ALT_FMul2 ALT_FMul3));

   /*
    * The rather extensive collection of HP-PA load operations
     */
   $for(TYPE in C C2 I F F2)
   {
      OP_LD_SHORT_${TYPE}       (alt(ALT_LD_Short));
      OP_LD_INDEX_${TYPE}       (alt(ALT_LD_Index));
      OP_LD_SHORT_PRE_${TYPE}   (alt(ALT_LD_Inc_Short));
      OP_LD_SHORT_POST_${TYPE}  (alt(ALT_LD_Inc_Short));
      OP_LD_INDEX_POST_${TYPE}  (alt(ALT_LD_Inc_Index));
   }

   // The C C2 and I loads have long versions
   OP_LD_LONG_C         (alt(ALT_LD_Long));
   OP_LD_LONG_C2        (alt(ALT_LD_Long));
   OP_LD_LONG_I         (alt(ALT_LD_Long));

   // Int loads also has long increment versions
   OP_LD_LONG_PRE_I             (alt(ALT_LD_Inc_Long));
   OP_LD_LONG_POST_I            (alt(ALT_LD_Inc_Long));
}
```

Figure 35: An example Operation section, described in Example 7.2.1.

interlocking due to a fdiv).

The Operation entries modeling all the loads supported by the PA-7100 are also shown in this example. Each load operation's entry specifies only one Scheduling_Alternative entry, because the PA-7100 has a different assembly name for each operand combination. In will be shown in the compiler-specific IMPACT_Operation section (Section 7.3) how all these variations are grouped into about a third as many IMPACT operations. For example, the PA-7100 Operation entries OP_LD_SHORT_I, OP_LD_INDEX_I, and OP_LD_LONG_I are all grouped into the IMPACT_Operation entry Lop_LD_I (the PA-7100 operations differ only in operand formats supported).

7.3 IMPACT_Operation

The compiler-independent Operation entries described in Section 7.2 often cannot be used "as is" by a compiler. When this is the case, some compiler-specific information needs to be added in order to interface the machine description with the compiler. One way to add this information is to define compiler-specific operation entries in terms of the compiler-independent Operation entries. The IMPACT compiler does this through the 'IMPACT_Operation' section, which has the following MD declaration:

```
CREATE SECTION IMPACT_Operation
  REQUIRED op(LINK(Operation) LINK(Operation)*);
  OPTIONAL flags(LINK(IMPACT_Operation_Flag)*);
{
}
```

An entry is created in the IMPACT_Operation section for each operation supported the architecture's code generator. By IMPACT's convention, the operation names defined in IMPACT's header files are used to name the entries (for example, Lop_LD_I and LHPPAop_LDIL). Each IMPACT_Operation entry's 'op' field lists the Operation entries, in priority order, that this entry can be mapped to. This allows a generic IMPACT operation to be mapped to multiple assembly-level operation. In addition, compiler-specific information about the IMPACT_Operation entry can be specified in the 'flags' field (see Section 9.3 for more details).

Example 7.3.1: The way PA-7100's floating-point multiply operations and load operations are described in the IMPACT_Operation section is shown in Figure 36. The IMPACT_Operation entries for the floating-point multiplies (Lop_MUL_F and Lop_MUL_F2) map directly to assembly-level operations (OP_MUL_F and OP_MUL_F2). Their flags field specifies that these floating-point multiplications can cause exceptions.

The IMPACT_Operation entries for the PA-7100's load operations map to multiple assembly-level operations, due to the multiple addressing modes supported by the PA-7100. For this reason, the generic integer load operation entry Lop_LD_I lists in its 'op' field the Operation entries OP_LD_SHORT_I, OP_LD_INDEX_I, and OP_LD_LONG_I. The Lop_LD_I entry is also flagged as a load and as a possibly excepting operation.

8 Register Information

The machine description's compiler-independent register information describes the way the processor's registers overlap. In addition, compiler-specific machine description extensions can be used to describe how to register allocate for the architecture. One MD language section, Register, is used to represent the compiler-independent information. Two more MD language sections, Register_Class and Register_File, and some

```
SECTION IMPACT_Operation
{
    Lop_MUL_F    (flags(EXCEPT) op(OP_MUL_F));
    Lop_MUL_F2   (flags(EXCEPT) op(OP_MUL_F2));

   // The rather extensive collection of HP-PA load operations
   $for(TYPE in C C2 I F F2)
   {
      Lop_LD_${TYPE}      (flags(LOAD EXCEPT)
                           op (OP_LD_SHORT_${TYPE} OP_LD_INDEX_${TYPE}));
      Lop_LD_PRE_${TYPE}  (flags(LOAD EXCEPT) op(OP_LD_SHORT_PRE_${TYPE}));
      Lop_LD_POST_${TYPE} (flags(LOAD EXCEPT)
                           op(OP_LD_SHORT_POST_${TYPE} OP_LD_INDEX_POST_${TYPE}));
   }

   // The C C2 and I loads have long versions
   Lop_LD_C      (op||(OP_LD_LONG_C));
   Lop_LD_C2     (op||(OP_LD_LONG_C2));
   Lop_LD_I      (op||(OP_LD_LONG_I));

   // Int loads also has long increment versions
   Lop_LD_PRE_I  (op||(OP_LD_LONG_PRE_I));
   Lop_LD_POST_I (op||(OP_LD_LONG_POST_I));
}
```

Figure 36: An example IMPACT_Operation section, described in Example 7.3.1.

additional Register section fields, can be added to represent the compiler-specific register allocation information. The fields declared in each of these MD language sections, and how these sections are linked together, is shown in Figure 37.

The way that the processor's registers overlap is described by the Register section's entries. The information necessary to do machine description based register allocation can be added, by linking each Register section entry to a Register_Class entry in order to specify its properties to the register allocator, and by using Register_File entries to group registers together into register banks. The rest of this section describes how each of these MD sections is used to convey these two types of register information and provides examples of their usage.

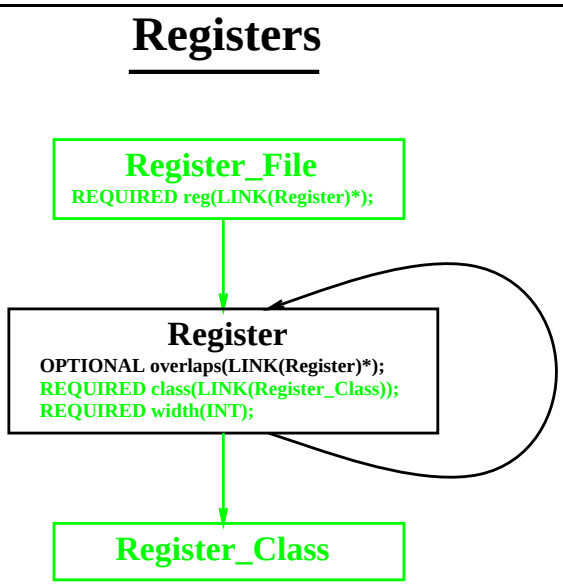

Figure 37: Structural overview of the machine description's register information. Highlighted entry names, fields, and arrows are IMPACT-specific.

8.1 Register Overlap

When determining which register dependencies are needed, it is necessary to know how know how physical (and virtual) registers overlap so that proper flow, anti, and output dependencies may be created after (before) register allocation. The architecture's implementation determines how physical registers overlap, so this information is considered compiler-independent and should be specified in the compiler-independent machine description. Because the way virtual registers are handled and overlap is compiler-specific, virtual register overlap information should be placed in the compiler-specific machine description. The 'Register' section used to describe this overlap is defined in the MD language as follows:

```
CREATE SECTION Register
  OPTIONAL overlaps (LINK(Register)*);
{
}
```

By convention, if the 'overlap' field is not specified, the register overlaps no other register (same as if an empty list is specified for overlap). Since the 'overlaps' relation is not transitive, all overlaps must be explicitly specified. Also by convention, if register x overlaps register y, this overlap must be specified for both registers x and y. The customizer and/or the compiler should punt if this condition is not met (IMPACT's customizer detects this case) instead of "filling in" the missing overlap.

Example 8.1.1: The PA-7100's double register %fr1 overlaps with the two float registers %fr1R and %fr1L. This overlap can be specified in the Register section as shown in Figure 38.

```
SECTION Register
{
   '%fr1'  ();                              // Declare double's names
   '%fr1R' (overlaps ('%fr1'));
   '%fr1L' (overlaps ('%fr1'));
   '%fr1'  (overlaps ('%fr1R' '%fr1L')); // Now define double's overlap
}
```

Figure 38: An example Register section specifying the overlap of PA-7100's %fr1, %fr1R, and %fr1L registers, as described in Example 8.1.1.

Example 8.1.2: The PA-7100 has 32 double registers (0 - 31) which overlap with its 64 float registers in the manor described in Example 8.1.1. It also has 32 integer registers which do not overlap with any other registers. This can be specified in the Register section as shown in Figure 39.

```
SECTION Register
{
   $for (REG_NUM in $0..31)
   {
      /* Integer registers */
      "%r${REG_NUM}" ();

      /* Floating point registers */
      "%fr${REG_NUM}" ();                // Declare double register's name
      '%fr${REG_NUM}R' (overlaps ('%fr${REG_NUM}'));
      '%fr${REG_NUM}L' (overlaps ('%fr${REG_NUM}'));
      '%fr${REG_NUM}'  (overlaps ('%fr${REG_NUM}R' '%fr${REG_NUM}L'));
   }
}
```

Figure 39: An example Register section specifying the overlap for all of the PA-7100's registers as described in Example 8.1.2.

Note that the 'Register' section is also used in IMPACT's compiler-specific machine description to describe the registers to the register allocator (see Section 8.2).

8.2 MDES-Driven Register Allocation

The register allocator needs to know which physical registers a that virtual register can be allocated to, and what the properties of that physical register is (such as is callee/caller saved and may the register be pre-allocated using one of IMPACT's so-called macro registers). The capabilities of the register allocator, and the treatment of virtual registers, is compiler-specific, so all the information in this section is considered compiler-specific.

(In the current IMPACT, arrays with the information described in this section is created by the code generators and passed to the generic register allocator. This hard-coded implementation sometimes limits flexibility when evaluating virtual architectures. It is envisioned that Limpact and Lplaydoh will use the information described in this section to build the arrays passed to the register allocator instead of using the hard coded method. The code generators for existing architectures, such as Lhppa, Lsparc, and Lx86, will probably continue to use their hard coded, non-mdes driven, versions.)

There are three mdes sections used to describe the processor's registers to the register allocator. The first, the 'Register_Class' section, has the following definition in the MD language:

```
CREATE SECTION Register_Class
{
}
```

This section's entries simply enumerates the types of registers the register allocator supports. This section, and its contents, appears only in the "structure_IMPACT.hmdes2" header file and is used to coerce the MD language into doing some error checking on register class names.

Example 8.2.1: Figure 40 shows the Register_Class section as it appears in "structure_IMPACT.hmdes2".

```
CREATE SECTION Register_Class
{
   callee();
   caller();
   macro_callee();
   macro_caller();
   reserved();
}
```

Figure 40: The Register_Class entries supported by IMPACT as described in Example 8.2.1.

The second section used to describe the processor's registers to the register allocator, is the 'Register'

section. Its MD definition is expanded (from Section 8.1's definition) to allow specification of each register's class, and the width, in bits, of each register, as shown bellow:

```
CREATE SECTION Register
  REQUIRED class (LINK(Register_Class));
  REQUIRED width (INT);
  OPTIONAL overlaps (LINK(Register)*);
{
}
```

The 'class' field is defined as a LINK instead of a STRING to provide some error checking. These classes specify whether it is a general register with a particular saving convention (callee and caller), a pre-allocated register that the register allocator may use (macro_callee and macro_caller), or a register that may not be used by the register allocator (reserved). The width information is used as a sanity check.

Example 8.2.2: Figure 41 shows the definition of the PA-7100's floats that use the caller save convention.

```
SECTION Register
{
   /* Caller floats */
   $for (REG_NUM in $8..11 $22..31)
   {
      "%fr${REG_NUM}R" (class(caller) width(32));
      "%fr${REG_NUM}L" (class(caller) width(32));
      "%fr${REG_NUM}"  (class(caller) width(64));
   }

   /* Macro caller floats */
   $for (REG_NUM in $4..7)
   {
      "%fr${REG_NUM}R" (class(macro_caller) width(32));
      "%fr${REG_NUM}L" (class(macro_caller) width(32));
      "%fr${REG_NUM}"  (class(macro_caller) width(64));
   }
}
```

Figure 41: An example of adding IMPACT-specific information to the Register section as described in Example 8.2.2.

The third section used to describe the procesor's registers to the register allocator, is the 'Register_File'

section. The MD definition for this section is as follows:

```
CREATE SECTION Register_File
  REQUIRED reg(LINK(Register)*);
{
}
```

Each entry represents a type of register (int, float, double, or predicate), and the field 'reg' enumerates the registers of that type that the register allocator may use to allocate virtual register to physical registers. By IMPACT's convention, registers listed first in the 'reg' field will be used first by the register allocator.

Example 8.2.3: The PA-7100 register files may be modeled as shown in Figure 42.

```
SECTION Register_File
{
    i  (reg ($for (N in $3..18 $19..22 $23..26 28 29 31) {"%r${N}" }));

    f  (reg ($for (N in $12..21 $8..11 $22..31 $4..7)
            {"%fr${N}R" "%fr${N}L" }));

    f2 (reg ($for (N in $12..21 $8..11 $22..31 $4..7) {"%fr${N}" }));
}
```

Figure 42: An example Register_File section, described in Example 8.2.3.

9 Compiler-Specific Information

The machine description's compiler-specific information can be used to provide any addition information the compiler needs about the architecture. As an example of such a usage, three IMPACT-specific MD language sections (Parameter, IMPACT_Operation_Flag, and IMPACT_Alt_Flag) are described in this section. The fields declared in each of these MD language sections, and how these sections are linked to other machine description sections, is shown in Figure 43.

The Parameter section's entries are used to pass architecture-specific parameters to the compiler. The IMPACT_Operation_Flag and IMPACT_Alt_Flag sections' entries are used to enumerate the flags used by an IMPACT-specific extension of the machine description. The rest of this section describes each of these MD sections in detail, and provides examples of how these MD sections are used.

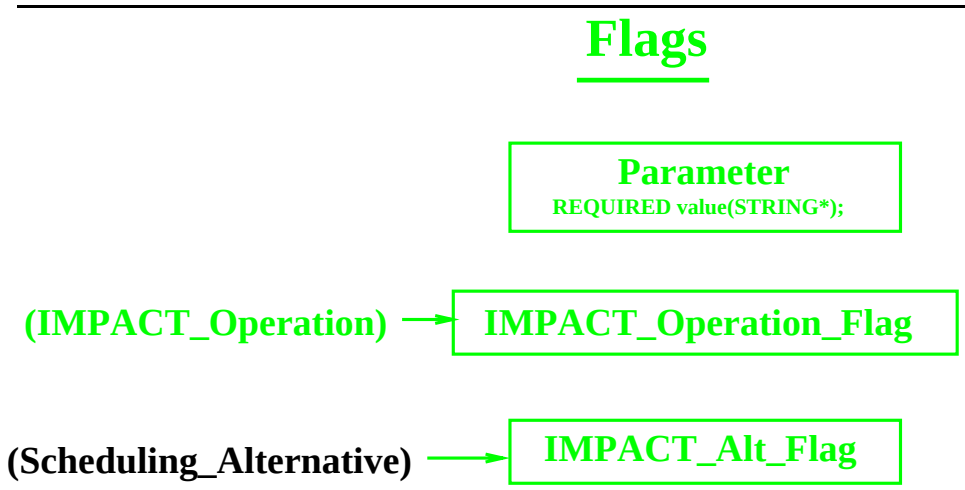

Figure 43: Structural overview of the machine description's compiler-specific information for IMPACT.

9.1 Parameter

Often new compiler-specific sections are needed to pass information of some sort to the compiler. The "Parameter", used by IMPACT, is an example of such a compiler-specific section. This section is used to pass machine description related parameters to the compiler/scheduler/optimizer/etc. The section is defined in the MD language as follows:

```
CREATE SECTION Parameter
  REQUIRED value (STRING*);
{
}
```

All Parameter entries are required to have the 'value' field specified but an empty value field may be valid for some parameters. This makes this section's usage similar to IMPACT's parameter file, but unlike IMPACT's parameter file, strict type checking is not done by the MD language (they are all treated as strings), so it is up to the customizer and/or compiler to check the validity of the values specified.

Example 9.1.1: A sample IMPACT Parameter section is shown in Figure 44. The 'processor_model' parameter effects the layout of the code generated by the code scheduler. The 'customization_headers' parameter is used to specify the IMPACT header files that should be used by the IMPACT lmdes customizer.

```
SECTION Parameter
{
   processor_model  (value(superscalar));

   customization_headers      (value(
      ${IMPACT_REL_PATH}/src/include/Lcode/l_opc.h
      ${IMPACT_REL_PATH}/src/include/Lcode/l_flags.h
      ${IMPACT_REL_PATH}/src/include/machine/m_spec.h
      ${IMPACT_REL_PATH}/src/include/machine/m_hppa.h
      ${IMPACT_REL_PATH}/src/include/Lcode/lhppa_phase1.h));
}
```

Figure 44: An example Parameter section, described in Example 9.1.1.

9.2 IMPACT_Alt_Flag

Often additional compiler-specific information needs to be added to the compiler-independent entries in the machine description. Using the MD language features, it is easy to add new fields to existing sections, and to fill in these new fields on existing entries. These MD language features are used by IMPACT to add a 'flags' field to the 'Scheduling_Alternative' section described in Section 7.1. This 'flags' field, in the Scheduling_Alternative section, has the following declaration:

```
OPTIONAL flags (LINK(IMPACT_Alt_Flag)*);
```

The "IMPACT_Alt_Flag" section, that is linked to in the above declaration, has the following MD declaration:

```
CREATE SECTION IMPACT_Alt_Flag
{
}
```

Example 9.2.1: An example of the scheduling flags IMPACT uses is shown in Figure 45. The two flags 'NT' and 'NN' are used to specify the squashing convention used by each branch's Scheduling_Alternative entry. The flag 'SILENT' is used when an Scheduling_Alternative entry represents a silent operation, where exceptions are suppressed.

```
CREATE SECTION IMPACT_Alt_Flag
{
   NT     ();     // Nullify Taken (squashes delay slot when cbr taken)
   NN     ();     // Nullify Not-taken (squashes when cbr not taken)
   SILENT ();     // Alternative is a silent version of the operation.
                  // Indicates that this form is valid when operation is
                  // being speculated above a branch.
}
```

Figure 45: An example IMPACT_Alt_Flag section, described in Example 9.2.1.

9.3 IMPACT_Operation_Flag

The compiler-specific "IMPACT_Operation_Flag" section is used in IMPACT's machine description to enumerate the IMPACT_Operation entry flags. The MD declaration of this section is shown below:

```
CREATE SECTION IMPACT_Operation_Flag
{
}
```

Example 9.3.1: An example set of IMPACT_Operation flags are shown in Figure 46. The comments in the figure explain what each flag does.

```
CREATE SECTION IMPACT_Operation_Flag
{
   IGNORE  ();     // Marks that ignored by scheduler
   SYNC    ();     // Nothing may move past this operation
   NI      ();     // Non-interlocking (has delay slot)
   EXPANDS ();     // Expands into multiple operations after scheduling.
                   // May not be put in delay slot of branch.
   EXCEPT  ();     // May cause exceptions, may not be moved above a
                   // branch unless has SILENT flag also set.

   JSR     ();     // Operation types. Used by the dependence graph builder
   RTS     ();     // and the  register allocator.
   JMP     ();
   CBR     ();
   LOAD    ();
   STORE   ();
}
```

Figure 46: An example IMPACT_Operation_Flag section, described in Example 9.3.1.

Acknowledgments

The authors would like to thank Rick Hank, Roger Bringmann, Scott Mahlke, Sabrina Hwu, Dan Lavery, Dave Gallagher, Andrew Hsieh, and all the members of the IMPACT research group, whose comments and suggestions over the years have helped improve the quality of the MDES facility significantly. The authors would also like to thank Mike Schlansker, Vinod Kathail, Shail Aditya, and the rest of the CAR group at HP Labs for valuable insight and discussion concerning countless machine description issues.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devices, Hewlett-Packard, SUN Microsystems, NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

A Initial specification of structure_pristine.hmdes2

```
/*
 * Preliminary pristine Hmdes2 structure.
 *
 * Created by John C. Gyllenhaal 2/10/95
 */

/*
 * Specifies the registers in the architectures.
 * Also specifies how these registers overlap each other.  The overlapping
 * conditions are assumed to be completely specified (up to the compiler
 * to check for errors in this case).
 */
CREATE SECTION Register
  OPTIONAL overlaps (LINK(Register)*);
{
}


/*
 * Specifies the field types supported by the architecture's
 * instruction set (assembly language).
 *
 * Currently uses LINKs to specify the hierarchy of field types.
 *
 * When the CHILD datum type gets implemented, the LINK will get changed
 * to CHILD.  For example, lit11 is compatible with lit5.
 */
CREATE SECTION Field_Type
  OPTIONAL compatible_with(LINK(Field_Type)*);
{
}

/*
 * Specifies the architecture's assembly language operation formats
 */
CREATE SECTION Operation_Format
  OPTIONAL pred(LINK(Field_Type)*);
  OPTIONAL src(LINK(Field_Type)*);
  OPTIONAL dest(LINK(Field_Type)*);
{
}

/*
 * Specifies the architecture's resources (real or for only
 * interlock purposes)
 */
CREATE SECTION Resource
{
}
```

```
/*
 * Specifies the characteristic's of the architecture's resource usage.
 * Namely, when a resource is used by an operation relative to
 * issue time.
 *
 * Possible extension:  Specify if resource used if operation predicate
 *  squashed.
 */
CREATE SECTION Resource_Usage
  REQUIRED use(LINK(Resource));
  REQUIRED time(INT INT*);
{
}

/*
 * Specifies resource units (such as function units) that should
 * be treated like a unit by the scheduler/simulator
 */
CREATE SECTION Resource_Unit
  REQUIRED use(LINK(Resource_Usage) LINK(Resource_Usage)*);
{
}

/*
 * Specifies option for the reservation tables.  For example,
 * that any one of the ialus may be used.
 */
CREATE SECTION Table_Option
  REQUIRED one_of(LINK(Resource_Unit|Resource_Usage)
  LINK(Resource_Unit|Resource_Usage)*);
{
}


/*
 * Specifies the reservation tables used to model the architecture.
 * (How resources are used as an operation executes.)
 *
 */
CREATE SECTION Reservation_Table
  REQUIRED use(LINK(Table_Option|Resource_Unit|Resource_Usage)*);
{
}

/*
 * Used to specify how register and sync operands interact so that
 * a dependence distance can be calculated.
 *
 * For now, a list of possible use times are specified for each operand.
 *
```

```
 * Other information that it may be desirable to specify is whether
 * the dependence holds for predicated operations on disjoint predicates.
 */
CREATE SECTION Operand_Latency
  REQUIRED time(INT*);
{
}


/*
 * The section is used to specify the dependence distances between
 * operations (if dependence exists).  All dependences (register,
 * memory, control, sync, etc) are modeled as flow,
 * output, and anti dependencies between operands.
 *
 * Uses Operand_Latency above to specify how operands interact.
 *
 *
 */
CREATE SECTION Operation_Latency
  OPTIONAL dest(LINK(Operand_Latency)*);
  OPTIONAL src(LINK(Operand_Latency)*);
  OPTIONAL pred(LINK(Operand_Latency)*);

  OPTIONAL mem_dest(LINK(Operand_Latency));
  OPTIONAL ctrl_dest(LINK(Operand_Latency));
  OPTIONAL sync_dest(LINK(Operand_Latency));

  OPTIONAL mem_src(LINK(Operand_Latency));
  OPTIONAL ctrl_src(LINK(Operand_Latency));
  OPTIONAL sync_src(LINK(Operand_Latency));
{
}

/*
 * Describes a scheduling alternative for an operation.
 *
 * Allow multiple operation formats to be specified
 * (for operations that have multiple operation formats but behave the same).
 */
CREATE SECTION Scheduling_Alternative
  REQUIRED format (LINK(Operation_Format) LINK(Operation_Format)*);
  REQUIRED resv (LINK(Reservation_Table));
  REQUIRED latency (LINK(Operation_Latency));
{
}

/*
 * Describes the operations in the architecture.
 *
 * The scheduling alternatives for each operations may be specified.
```

```
 */
CREATE SECTION Operation
  REQUIRED alt(LINK(Scheduling_Alternative) LINK(Scheduling_Alternative)*);
{
}
```

B Initial specification of PA_7100_pristine.hmdes2

```
/*
 * Preliminary conversion of PA_7100.hmdes to pristine hmdes2 format.
 *
 * Written by John C. Gyllenhaal 2/10/95
 */

/* Include the structure specification for this pristine hmdes. */
$include "structure_pristine.hmdes2"

/*
 * Specifies the registers in the architectures.
 * Also specifies how these registers overlap each other.  The overlapping
 * conditions are assumed to be completely specified (up to the compiler
 * to check for errors in this case).
 *
 * I am using the PA_7100 assembly format for register names as
 * the entry names.  Since these names have a '%' in them, the names
 * must be quoted.
 */
SECTION Register
{
/*
   * Specify overlap conditions.
         *
         * Ints do not overlap.
         * Floats overlap with double with same number.
         */
$for (REG_NUM in $0..31)
{
/* Integer registers */
"%r${REG_NUM}" ();

/* Floating point registers */
"%fr${REG_NUM}" (); /* Declare double register's name */
"%fr${REG_NUM}R" (overlaps("%fr${REG_NUM}"));
"%fr${REG_NUM}L" (overlaps("%fr${REG_NUM}"));
"%fr${REG_NUM}" (overlaps("%fr${REG_NUM}R" "%fr${REG_NUM}L"));
}
}

/*
 * Specifies the field types supported by the architectures
 * instruction set (assembly language).
 */
SECTION Field_Type
{
NULL ();  // NULL and any should be explicit
REG ();
```

```
Lit5 ();
Lit11 (compatible_with(Lit5));
Label ();
Lit12 (compatible_with(Lit11 Label));
Lit14 (compatible_with(Lit12));
Lit17 (compatible_with(Lit14));
Lit21 (compatible_with(Lit17));
any (compatible_with(Lit21 REG NULL));
}

/*
 * Specifies the architecture's assembly language operation formats
 */
SECTION Operation_Format
{
    // Standard operation formats
    OF_Std1  (dest(REG) src(REG REG));
    OF_Std2 (dest(REG) src(Lit11 REG));
    OF_Std3 (dest(REG) src(REG Lit11));

    OF_ZVDEPI (dest(REG) src(Lit5 Lit11 REG));
    OF_ZVDEP (dest(REG) src(REG Lit11 REG));
    OF_VSHD (dest(REG) src(REG REG REG));
    OF_EXTRU (dest(REG) src(REG Lit11 Lit11));
    OF_DEPI (dest(REG) src(Lit11 Lit11 Lit11 REG));

    OF_Lit5 (dest(REG) src(Lit5 REG));
    OF_Lit11 (dest(REG) src(Lit11 REG));
    OF_Lit14 (dest(REG) src(Lit14 REG));
    OF_Lit21_Mov (dest(REG) src(Lit21));
    OF_Lit21_Add (dest(REG) src(Lit21 REG));
    OF_Label_Add (dest(REG) src(Label Lit21 REG));
    OF_Label_Mov        (dest(REG) src(Label Lit21));

    // Operation format for "move"
    OF_Mov  (dest(REG) src(REG));

    // Operation format for fp 5op
    OF_5op (dest(REG REG) src(REG REG REG REG));

    // Operation format for unconditional branch
    OF_Rts              (src(any));
    OF_Jsr1             (dest(any any) src(Lit17));
    OF_Jsr2             (dest(any any) src(REG));
    OF_Ubr1   (src(Lit17));
    OF_Ubr2 (src(REG));
    OF_Ubr3 (src(REG REG));

    // Operation format for conditional branch
    OF_Cbr1 (src(REG REG Lit12));
    OF_Cbr2 (dest(any) src(Lit5 REG Lit12));
```

```
    OF_Cbr_BB (src(REG Lit11 Lit12));

    // Operation format for Stores
    OF_Store1 (src(Lit21 REG REG));
    OF_Store2 (src(Lit21 REG REG));
    OF_Store2_Label     (src(Label Lit21 REG REG));
    OF_Store3 (src(REG REG REG));
     // pre/post inc short store
    OF_Store4 (dest(REG) src(Lit5 REG REG));
     // pre/post inc long store
    OF_Store5 (dest(REG) src(Lit14 REG REG));
     // post inc index store
    OF_Store6 (dest(REG) src(REG REG REG));

    // Operation format for loads
    OF_Load1 (dest(REG) src(Lit21 REG));
    OF_Load2 (dest(REG) src(Lit21 REG));
    OF_Load2_Reg  (dest(REG) src(Lit21 REG REG));
    OF_Load2_Label      (dest(REG) src(Label Lit21 REG));
    OF_Load3 (dest(REG) src(REG REG));
     // pre/post inc short load
    OF_Load4 (dest(REG REG) src(Lit5 REG));
     // pre/post inc long load
    OF_Load5 (dest(REG REG) src(Lit14 REG));
    // post inc index load
    OF_Load6 (dest(REG REG) src(REG REG));

    OF_Nil  (); // Nop
}

/*
 * Specifies the architecture's resources (real or for only
 * interlock purposes)
 *
 * Naming convention for resources:
 * name[id_#][_s{stage_#}]
 *
 * where 'name' is the name of the resource.
 *       'id_#' is used when there is replicated resources.
 *       'stage_#' is used when resource is a pipeline stage.
 */
SECTION Resource
{
    slot0 ();
    slot1 ();
    ialu ();
    idummy ();
    fissue ();
    falu_s0 ();
    falu_s1 ();
    fmul_s0 ();
```

```
    fmul_s1 ();
    fmulfake ();
    fdiv ();
    mem ();
}

/*
 * Specifies the characteristic's of the architecture's resource usage.
 * Namely, when a resource is used by an operation relative to
 * issue time.
 *
 * Naming convention for resource usages:
 * RU_{name}[id_#][_s{stage_#}][_t{times}]
 *
 * where 'name' is the name of the resource.
 *       'id_#' is used when there is replicated resources.
 *       'stage_#' is used when resource is a pipeline stage.
 *       'times' is the times used if it is used a more than 1 time.
 *               If a range is used, use {start}_{stop}.
 */

SECTION Resource_Usage
{
    RU_slot0  (use(slot0) time(0));
    RU_slot1 (use(slot1) time(0));

    RU_ialu_t0 (use(ialu) time(0));
    RU_ialu_t0_1 (use(ialu) time(0 1));

    /* idummy used to prevent placement of operations in delay slot */
    RU_idummy_t0 (use(idummy) time(0));
    RU_idummy_t1 (use(idummy) time(1));

    RU_fissue (use(fissue) time(0));

    RU_falu_s0 (use(falu_s0) time(0));
    RU_falu_s1 (use(falu_s1) time(1));

    RU_fmul_s0_t0 (use(fmul_s0) time(0));
    RU_fmul_s0_t6_7 (use(fmul_s0) time(6 7));
    RU_fmul_s0_t13_14 (use(fmul_s0) time(13 14));
    RU_fmul_s1   (use(fmul_s1) time(1));

    /* fmulfake used to model the interlocks between fmul and fdiv */
    RU_fmulfake_t0 (use(fmulfake) time(0));
    RU_fmulfake_t5 (use(fmulfake) time(5));
    RU_fmulfake_t12 (use(fmulfake) time(12));

    RU_fdiv_t0_7 (use(fdiv) time($0..7));
    RU_fdiv_t0_14 (use(fdiv) time($0..14));
```

```
    RU_mem_t0 (use(mem) time(0));
    RU_mem_t0_1 (use(mem) time(0 1));
}

/*
 * Specifies resource units (such as function units) that should
 * be treated like a unit by the scheduler/simulator
 *
 * Naming convention for resource units:
 * {name}_unit
 */
SECTION Resource_Unit
{
    /* Resources required for issuing floating point operations */
    fissue0_unit  (use(RU_slot0 RU_fissue));
    fissue1_unit  (use(RU_slot1 RU_fissue));

    /* Integer units */
    ialu_unit (use(RU_ialu_t0));
    icmp_unit (use(RU_ialu_t0_1 RU_idummy_t0));
    ibr_unit (use(RU_ialu_t0 RU_idummy_t1));
    ibr2_unit (use(RU_ialu_t0_1)); // For dynamic jsrs, no delay slot

    /* Floating point units */
    falu_unit   (use(RU_falu_s0 RU_falu_s1));

    fmul_unit   (use(RU_falu_s0 RU_falu_s1 RU_fmul_s0_t0 RU_fmul_s1
                     RU_fmulfake_t0));
    /* Interlock case, where fmul interlocks with fdiv */
    fmulI_unit  (use(RU_falu_s0 RU_falu_s1 RU_fmul_s0_t0 RU_fmul_s1));

    fdivS_unit (use(RU_fdiv_t0_7 RU_fmulfake_t5 RU_fmul_s0_t6_7));
    fdivD_unit (use(RU_fdiv_t0_14 RU_fmulfake_t12 RU_fmul_s0_t13_14));

    /* Memory units */
    load_unit (use(RU_ialu_t0 RU_mem_t0));
    store_unit (use(RU_ialu_t0 RU_mem_t0_1));
}

/*
 * Specifies option for the reservation tables.  For example,
 * that any one of the ialus may be used.
 *
 * Naming convention for table options:
 * {desc}_{name}
 *
 * where desc is any, some, etc.
 */
SECTION Table_Option
{
    /* Integer issue requirements */
```

```
    any_iissue (one_of(RU_slot0 RU_slot1));

    /* Float issue requirements */
    any_fissue (one_of(fissue0_unit fissue1_unit));
}

/*
 * Specifies how resources are used as operations execute.
 */
SECTION Reservation_Table
{
    RT_INOP  (use(any_iissue));

    /* Integer operations */
    RT_IAlu (use(any_iissue ialu_unit));
    RT_ICmp (use(any_iissue icmp_unit));

    RT_IBr (use(any_iissue ibr_unit));
    //dynamic subroutine, no delay slot
    RT_IBr2 (use(any_iissue ibr2_unit));


    /* Floating point operations */
    RT_FPAlu  (use(any_fissue falu_unit));

    RT_FPMul2 (use(any_fissue fmul_unit));
    /* interlocking fmul case, takes extra cycle */
    RT_FPMul3 (use(any_fissue fmulI_unit));

    RT_FPAluMul (use(any_fissue fmul_unit));

    RT_FPDivS (use(any_fissue fdivS_unit));
    RT_FPDivD (use(any_fissue fdivD_unit));


    /* Memory operations */
    RT_Load (use(any_iissue load_unit));
    RT_Store (use(any_iissue store_unit));
}

/*
 * Used to specify how register and sync operands interact so that
 * a dependence distance can be calculated.
 *
 * For now, a list of possible use times are specified for each operand.
 */
SECTION Operand_Latency
{
   /* Source operands */
   $for (N in 0 2)
   {
```

```
     s${N} (time(N));
   }

   /* Destination operands */
   $for (N in 1 2 3 8 15)
   {
     d${N} (time(N));
   }

   /* Sync Source operands */
   $for (N in 0)
   {
     ss${N} (time(N));
   }

   /* Sync Dest operands */
   $for (N in 0 1 2)
   {
     sd${N} (time(N));
   }

   /* For operands we want to be warned about if they are being used
    * For debugging purposes.
    */
   x (time());
}

/*
 * The section is used to specify the dependence distances between
 * operations (if dependence exists).  All dependences (register,
 * memory, control, sync, etc) are modeled as flow,
 * output, and anti dependencies between operands.
 *
 * Uses Operand_Latency above to specify how operands interact.
 *
 * May need to replace more operands with 'x' for debugging purposes.
 */
SECTION Operation_Latency
{
   // Non-memory operations don't use mem_src or mem_dest
   // All operations may have ctrl or sync dependences.
   $for(N in 1 2 3 8 15)
   {
     OL_Lat${N} (dest(d${N} x)
 src(s0 s0 s0 s0)
 ctrl_dest(sd0)
 sync_dest(sd0)
 ctrl_src(ss0)
 sync_src(ss0));
   }
```

```
   // Some floating point operands with latency 2 have two destinations
   OL_Lat2 (dest!(d2 d2));


   OL_Branch    (dest(d1 x)
 src(s0 s0 s0 s0)
 ctrl_dest(sd1) // 1 cycle ctrl flow dep
 sync_dest(sd0)
 ctrl_src(ss0)
 sync_src(ss0));

   // Stores read value to be stored two cycles after issue
   // They also may have two destinations for post-increment stores
   // Loads flow dependent on stores must follow store by at least 2 cycles.
   // Only mem_dest specified since stores only write memory
   OL_Store     (dest(d1 d1)
 src(s0 s0 s2 s0)
 mem_dest(sd2) // 2 cycle mem flow dep
 ctrl_dest(sd0)
 sync_dest(sd0)
 ctrl_src(ss0)
 sync_src(ss0));

   // Label stores store 4th operand instead of 3rd
   // Only mem_dest specified since stores only write memory
   OL_Label_Store (dest(d1 d1)
 src(s0 s0 s0 s2)
 mem_dest(sd2) // 2 cycle mem flow dep
 ctrl_dest(sd0)
 sync_dest(sd0)
 ctrl_src(ss0)
 sync_src(ss0));

   // Loads may have two destination.  Second dest for addr post-increment.
   // Only mem_src specified since loads only read memory
   OL_Load     (dest(d2 d1)
 src(s0 s0 s0 s0)
 ctrl_dest(sd0)
 sync_dest(sd0)
 mem_src(ss0)
 ctrl_src(ss0)
 sync_src(ss0));
}

/*
 * Describe the scheduling alternatives for the operations this
 * processor supports.
 *
 * Allow multiple operation formats to be specified
 * (for operations that have multiple operation formats but behave the same).
 */
```

```
SECTION Scheduling_Alternative
{
   // 1 cycle NO-OP
   ALT_NOP (format(OF_Nil) resv(RT_INOP) latency(OL_Lat1));

   // Integer conditional branches
   ALT_Cbr_COMB_FWD (format(OF_Cbr1) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_COMB_BWD (format(OF_Cbr1) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_COMIB_FWD (format(OF_Cbr2) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_COMIB_BWD (format(OF_Cbr2) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_ADDIB_FWD (format(OF_Cbr2) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_ADDIB_BWD (format(OF_Cbr2) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_BB_FWD (format(OF_Cbr_BB) resv(RT_IBr) latency(OL_Branch));
   ALT_Cbr_BB_BWD (format(OF_Cbr_BB) resv(RT_IBr) latency(OL_Branch));


   // Unconditional branches
   ALT_Jsr (format(OF_Jsr1) resv(RT_IBr) latency(OL_Branch));
   ALT_Jsr_DynCall (format(OF_Jsr2) resv(RT_IBr2) latency(OL_Branch));
   ALT_Rts (format(OF_Rts) resv(RT_IBr) latency(OL_Branch));
   ALT_Jump (format(OF_Ubr1) resv(RT_IBr) latency(OL_Branch));
   ALT_Jump_RG (format(OF_Ubr3) resv(RT_IBr) latency(OL_Branch));

   // Integer ALU operations
   ALT_Ialu (format(OF_Std1) resv(RT_IAlu) latency(OL_Lat1));
   ALT_Shift (format(OF_Std1 OF_Std3) resv(RT_IAlu)
 latency(OL_Lat1));
   ALT_Copy (format(OF_Mov) resv(RT_IAlu) latency(OL_Lat1));
   ALT_LDO_Mov (format(OF_Lit14) resv(RT_IAlu) latency(OL_Lat1));
   ALT_LDO_Add (format(OF_Lit21_Add OF_Label_Add) resv(RT_IAlu)
 latency(OL_Lat1));
   ALT_LDIL (format(OF_Lit21_Mov OF_Label_Mov) resv(RT_IAlu)
       latency(OL_Lat1));
   ALT_ADDIL (format(OF_Lit21_Add OF_Label_Add) resv(RT_IAlu)
 latency(OL_Lat1));
   ALT_ADD (format(OF_Std1) resv(RT_IAlu) latency(OL_Lat1));
   ALT_ADDI (format(OF_Lit21_Add) resv(RT_IAlu) latency(OL_Lat1));
   ALT_SUB (format(OF_Std1) resv(RT_IAlu) latency(OL_Lat1));
   ALT_SUBI (format(OF_Lit11) resv(RT_IAlu) latency(OL_Lat1));
   ALT_ZVDEPI (format(OF_ZVDEPI) resv(RT_IAlu) latency(OL_Lat1));
   ALT_ZVDEP (format(OF_ZVDEP) resv(RT_IAlu) latency(OL_Lat1));
   ALT_VEXTRS (format(OF_ZVDEP) resv(RT_IAlu) latency(OL_Lat1));
   ALT_VSHD (format(OF_VSHD) resv(RT_IAlu) latency(OL_Lat1));

   // Integer comparison operations
   ALT_ICmp (format(OF_Std1 OF_Std2) resv(RT_ICmp)
 latency(OL_Lat2));

   // Floating point operations
   ALT_FMov (format(OF_Mov) resv(RT_FPAlu) latency (OL_Lat2));
   ALT_FAlu (format(OF_Std1) resv(RT_FPAlu) latency (OL_Lat2));
```

```
   ALT_FMul2 (format(OF_Std1) resv(RT_FPMul2) latency (OL_Lat2));
   ALT_FMul3 (format(OF_Std1) resv(RT_FPMul3) latency (OL_Lat3));
   ALT_FDivS (format(OF_Std1) resv(RT_FPDivS) latency (OL_Lat8));
   ALT_FDivD (format(OF_Std1) resv(RT_FPDivD) latency (OL_Lat15));
   ALT_FSqrtS (format(OF_Mov) resv(RT_FPDivS) latency (OL_Lat8));
   ALT_FSqrtD (format(OF_Mov) resv(RT_FPDivD) latency (OL_Lat15));
   ALT_FAluMul (format(OF_5op) resv(RT_FPAluMul) latency (OL_Lat2));

   // Load operations
   ALT_LD_Short (format(OF_Load1) resv(RT_Load) latency(OL_Load));
   ALT_LD_Long (format(OF_Load2 OF_Load2_Reg OF_Load2_Label)
         resv(RT_Load) latency(OL_Load));
   ALT_LD_Index (format(OF_Load3) resv(RT_Load) latency(OL_Load));
   ALT_LD_Inc_Short (format(OF_Load4) resv(RT_Load) latency(OL_Load));
   ALT_LD_Inc_Long (format(OF_Load5) resv(RT_Load) latency(OL_Load));
   ALT_LD_Inc_Index (format(OF_Load6) resv(RT_Load) latency(OL_Load));

   // Short operations
   ALT_ST_Short (format(OF_Store1) resv(RT_Store) latency(OL_Store));
   ALT_ST_Long   (format(OF_Store2) resv(RT_Store) latency(OL_Store));
   ALT_ST_Index (format(OF_Store3) resv(RT_Store) latency(OL_Store));
   ALT_ST_Inc_Short (format(OF_Store4) resv(RT_Store) latency(OL_Store));
   ALT_ST_Inc_Long (format(OF_Store5) resv(RT_Store) latency(OL_Store));
   ALT_ST_Inc_Index (format(OF_Store6) resv(RT_Store) latency(OL_Store));
}

/*
 * Describe the operations for this architecture
 *
 * The scheduling alternatives for each operation is specified.
 *
 * I currently used OP_ instead of Lop_ in front of operation names.
 * I plan to convert to quoted PA_7100 assembly names as soon as I get
 * the preprocessor array directives implemented so that mapping
 * GE_U to '<<=', etc is much easier.
 */
SECTION Operation
{
   // NOP
   OP_NOP (alt(ALT_NOP));

   /*
    *  Integer conditional branch operations
    */
   $for (CASE in EQ NE GT GE LT LE GT_U GE_U LT_U LE_U)
   {
      OP_COMB_${CASE}_FWD (alt(ALT_Cbr_COMB_FWD));
      OP_COMIB_${CASE}_FWD (alt(ALT_Cbr_COMIB_FWD));
      OP_COMB_${CASE}_BWD (alt(ALT_Cbr_COMB_BWD));
      OP_COMIB_${CASE}_BWD (alt(ALT_Cbr_COMIB_BWD));
      OP_ADDIB_${CASE}_FWD (alt(ALT_Cbr_ADDIB_FWD));
```

```
   OP_ADDIB_${CASE}_BWD (alt(ALT_Cbr_ADDIB_BWD));
}

// Branch on bit
$for (CASE in BB_0 BB_1)
{
   OP_${CASE}_FWD (alt(ALT_Cbr_BB_FWD));
   OP_${CASE}_BWD (alt(ALT_Cbr_BB_BWD));
}

// Unconditional branches
OP_JSR (alt(ALT_Jsr));
// Dynamic subroutine call doesn't have slot to fill
OP_JSR_DYNCALL (alt(ALT_Jsr_DynCall));
OP_RTS (alt(ALT_Rts));
OP_JUMP (alt(ALT_Jump));
OP_JUMP_RG (alt(ALT_Jump_RG));

/*
 * Integer ALU operations
 */
OP_COPY (alt(ALT_Copy));
OP_LDO_MOV (alt(ALT_LDO_Mov));
OP_LDO_ADD (alt(ALT_LDO_Add));
OP_LDIL (alt(ALT_LDIL));
OP_ADDIL (alt(ALT_ADDIL));
OP_ADD  (alt(ALT_ADD));
OP_ADDI  (alt(ALT_ADDI));
OP_SUB (alt(ALT_SUB));
OP_SUBI (alt(ALT_SUBI));

$for (OPCODE in ZVDEPI ZVDEP VSHD VEXTRS)
{
  OP_${OPCODE} (alt(ALT_${OPCODE}));
}
// logical operations
$for (OPCODE in OR AND XOR AND_COMPL)
{
  OP_${OPCODE} (alt(ALT_Ialu));
}

// shift operations
$for (OPCODE in LSL ASR LSR)
{
  OP_${OPCODE} (alt(ALT_Shift));
}


// Integer comparison operations
$for(CMP in EQ NE GT GE LT LE GT_U GE_U LT_U LE_U)
{
```

```
    OP_${CMP} (alt(ALT_ICmp));
  }

  /*
   * Floating point operations
   */
  $for(OPCODE in MOV_F MOV_F2 ABS_F ABS_F2 I_F I_F2 F2_I F_I F_F2 F2_F)
  {
    OP_${OPCODE} (alt(ALT_FMov));
  }

  $for(OPCODE in ADD SUB EQ GT GE NE LT LE)
  {
     $for(TYPE in F F2)
     {
OP_${OPCODE}_${TYPE} (alt(ALT_FAlu));
     }
  }

  // FP Multiply takes 2 cycles, unless interlocks due to conflict with divide
  OP_MUL_F (alt(ALT_FMul2 ALT_FMul3));
  OP_MUL_F2 (alt(ALT_FMul2 ALT_FMul3));
  OP_DIV_F (alt(ALT_FDivS));
  OP_DIV_F2 (alt(ALT_FDivD));
  OP_SQRT_F (alt(ALT_FSqrtS));
  OP_SQRT_F2 (alt(ALT_FSqrtD));

  // Floating-point 5ops operations
  OP_FMPYADD (alt(ALT_FAluMul));
  OP_FMPYSUB (alt(ALT_FAluMul));

  /*
   * The rather extensive collection of HP-PA load operations
   */
  $for(TYPE in C C2 I F F2)
  {
     OP_LD_SHORT_${TYPE} (alt(ALT_LD_Short));
     OP_LD_INDEX_${TYPE} (alt(ALT_LD_Index));
     OP_LD_SHORT_PRE_${TYPE}   (alt(ALT_LD_Inc_Short));
     OP_LD_SHORT_POST_${TYPE}  (alt(ALT_LD_Inc_Short));
     OP_LD_INDEX_POST_${TYPE}  (alt(ALT_LD_Inc_Index));
  }

  // The C C2 and I loads have long versions
  OP_LD_LONG_C (alt(ALT_LD_Long));
  OP_LD_LONG_C2 (alt(ALT_LD_Long));
  OP_LD_LONG_I         (alt(ALT_LD_Long));

  // Int loads also has long increment versions
  OP_LD_LONG_PRE_I   (alt(ALT_LD_Inc_Long));
  OP_LD_LONG_POST_I   (alt(ALT_LD_Inc_Long));
```

```
  /*
   * The almost as extensive collection of HP-PA store operations
   */
   $for(TYPE in C C2 I)
   {
      OP_ST_SHORT_${TYPE} (alt(ALT_ST_Short));
      OP_ST_LONG_${TYPE} (alt(ALT_ST_Long));
      OP_ST_SHORT_PRE_${TYPE}   (alt(ALT_ST_Inc_Short));
      OP_ST_SHORT_POST_${TYPE}  (alt(ALT_ST_Inc_Short));
   }

   // Int stores also has long increment versions
   OP_ST_LONG_PRE_I   (alt(ALT_ST_Inc_Long));
   OP_ST_LONG_POST_I   (alt(ALT_ST_Inc_Long));

   $for(TYPE in F F2)
   {
      OP_ST_SHORT_${TYPE} (alt(ALT_ST_Short));
      OP_ST_INDEX_${TYPE} (alt(ALT_ST_Index));
      OP_ST_SHORT_PRE_${TYPE}   (alt(ALT_ST_Inc_Short));
      OP_ST_SHORT_POST_${TYPE}  (alt(ALT_ST_Inc_Short));
      OP_ST_INDEX_POST_${TYPE}  (alt(ALT_ST_Inc_Index));
   }
}
```

C Initial specification of structure_IMPACT.hmdes2

```
/*
 * Preliminary IMPACT specific Hmdes2 structure
 * (built upon the pristine Hmdes2 structure).
 *
 * Created by John C. Gyllenhaal 2/10/95
 */

/* Section for passing parameters to the compiler */
CREATE SECTION Parameter
  REQUIRED value (STRING*);
{
}



/*
 * Specifies the valid register classes that may be specified.
 */
CREATE SECTION Register_Class
{
callee();
caller();
macro_callee();
macro_caller();
reserved();
}

/*
 * Add specification of class and width of the the registers in the
 * architectures, in order to do register allocation.
 *
 * (The pristine mdes specifies how registers overlap)
 *
 *
 */
SECTION Register
  REQUIRED class (LINK(Register_Class));
  REQUIRED width (INT);
{
}

/*
 * Specifies the Register files for the architecture
 *
 * The registers that are contained by each register file is specified.
 */
CREATE SECTION Register_File
  REQUIRED reg(LINK(Register)*);
{
```

```
}


/*
 * IMPACT scheduling alternative flag section
 *
 * Flags used for IMPACT scheduling alternatives.
 * Flags show are those that historically have been used as alternative flags.
 *
 * Currently just defining the flag names.
 */
CREATE SECTION IMPACT_Alt_Flag
{
   NT (); // Nullify Taken (squashes delay slot when cbr taken)
   NN (); // Nullify Not-taken (squashes when cbr not taken)
   SILENT (); // Alternative is a silent version of the operation.
               // Indicates that this form is valid when operation is
  // being speculated above a branch.
}

/*
 * Add a flag field to scheduling alternatives to allow flags to be specified.
 */
SECTION Scheduling_Alternative
  OPTIONAL flags(LINK(IMPACT_Alt_Flag)*);
{
}


/*
 * IMPACT operation flag section.
 *
 * Flags for IMPACT operations.
 * Used to specify operation types, scheduling information, etc.
 *
 * Currently just defining the flag names.
 */
CREATE SECTION IMPACT_Operation_Flag
{
   IGNORE (); // Marks that ignored by scheduler
   SYNC (); // Nothing may move past this operation
   NI (); // Non-interlocking (has delay slot)


   EXPANDS (); // Expands into multiple operations after scheduling.
// May not be put in delay slot of branch.

   EXCEPT (); // May cause exceptions, may not be moved above a
// branch unless has SILENT flag also set.
```

```
   /*
    * Operation types.
    *
    * Used by dependence graph builder and register allocator.
    */
   JSR ();
   RTS ();
   JMP ();
   CBR ();
   LOAD ();
   STORE ();
}


/*
 * IMPACT operation section.
 *
 * Maps impact names to on or more operations in the pristine hmdes2.
 * One IMPACT operation may map to many pristine operations and
 * many IMPACT operations may map to one pristine operation.
 *
 * Also specifies flags about the operation that the compiler needs,
 * see IMPACT_Operation_Flag section.
 */
CREATE SECTION IMPACT_Operation
  REQUIRED op(LINK(Operation) LINK(Operation)*);
  OPTIONAL flags(LINK(IMPACT_Operation_Flag)*);
{
}
```

D Initial specification of PA_7100_IMPACT.hmdes2

```
/*
 * Preliminary conversion of PA_7100.hmdes to IMPACT specific hmdes2 format.
 *
 * Written by John C. Gyllenhaal 2/10/95
 */

/*
 * Start with the pristine PA_7100.hmdes2
 */
$include "PA_7100_pristine.hmdes2"

/*
 * Add in the IMPACT specific syntax.
 */
$include "structure_IMPACT.hmdes2"

/* For now, specify release path here */
$def IMPACT_REL_PATH /home/altair1/impact

/*
 * Specify processor model and the header files to use in the
 * customization in the customization_headers entry.
 */
SECTION Parameter
{
// Indicate this is a superscalar processor
processor_model (value(superscalar));

customization_headers  (value(
    ${IMPACT_REL_PATH}/src/include/Lcode/l_opc.h
            ${IMPACT_REL_PATH}/src/include/Lcode/l_flags.h
            ${IMPACT_REL_PATH}/src/include/machine/m_spec.h
            ${IMPACT_REL_PATH}/src/include/machine/m_hppa.h
            ${IMPACT_REL_PATH}/src/include/Lcode/lhppa_phase1.h));

}



/*
 * Specifies the registers in the architectures.
 * Also specifies how these registers overlap each other.  The overlapping
 * conditions are assumed to be completely specified (up to the compiler
 * to check for errors in this case).
 *
 * I am using the PA_7100 assembly format for register names as
 * the entry names.  Since these names have a '%' in them, the names
 * must be quoted.
 *
```

```
 * Specifying the class and width information for the int and float registers.
 * Adding predicate registers for supporting emulation of predicated code.
 */
SECTION Register
{
   /*
    * Integer registers
    */
   /* Integer callee */
   $for (REG_NUM in $3..18)
   {
      "%r${REG_NUM}" (class(callee) width(32));
   }

   /* Integer caller */
   $for (REG_NUM in $19..22)
   {
      "%r${REG_NUM}" (class(caller) width(32));
   }

   /* Integer macro caller */
   $for (REG_NUM in 1 2 $23..26 28 29 31)
   {
      "%r${REG_NUM}" (class(macro_caller) width(32));
   }

   /* Integer reserved */
   $for (REG_NUM in 0 27 30)
   {
      "%r${REG_NUM}" (class(reserved) width(32));
   }


   /*
    * Float registers
    */
   /* Callee floats */
   $for (REG_NUM in $12..21)
   {
      "%fr${REG_NUM}R" (class(callee) width(32));
      "%fr${REG_NUM}L" (class(callee) width(32));
      "%fr${REG_NUM}" (class(callee) width(64));
   }

   /* Caller floats */
   $for (REG_NUM in $8..11 $22..31)
   {
      "%fr${REG_NUM}R" (class(caller) width(32));
      "%fr${REG_NUM}L" (class(caller) width(32));
      "%fr${REG_NUM}" (class(caller) width(64));
   }
```

```
   /* Macro caller floats */
   $for (REG_NUM in $4..7)
   {
      "%fr${REG_NUM}R" (class(macro_caller) width(32));
      "%fr${REG_NUM}L" (class(macro_caller) width(32));
      "%fr${REG_NUM}" (class(macro_caller) width(64));
   }

   /* Special registers */
   $for (REG_NUM in $0..3)
   {
      "%fr${REG_NUM}R" (class(reserved) width(32));
      "%fr${REG_NUM}L" (class(reserved) width(32));
      "%fr${REG_NUM}" (class(reserved) width(64));
   }

    /* Predicate block register (overlaps with all predicate registers) */
    "%pr_all" (class(callee) width (64));

    /* Predicate registers (all callee) */
    $for (REG_NUM in $0..63)
    {
       "%pr${REG_NUM}" (class(callee) width(1));
    }


    /* Set the overlap conditions between %pr_all and the predicate registers*/
    $for (REG_NUM in $0..63)
    {
       "%pr${REG_NUM}" (overlaps("%pr_all"));
       "%pr_all" (overlaps||("%pr${REG_NUM}"));
    }
}


/*
 * Specifies the Register files for the architecture
 *
 * The registers that are contained by each register file is specified.
 */
SECTION Register_File
{
    i (reg ($for (N in $3..18 $19..22 $23..26 28 29 31) {"%r${N}" }));
    f (reg ($for (N in $12..21 $8..11 $22..31 $4..7)
       {"%fr${N}R" "%fr${N}L" }));
    f2 (reg ($for (N in $12..21 $8..11 $22..31 $4..7)
   {"%fr${N}" }));
    // Add a predicate register file for emulation by Lhppa
    p (reg ($for (N in $0..63) {"%pr${N}" }));
}
```

```
/*
 * Specifies the field types supported by the architecture's
 * instruction set (assembly language).
 *
 * Add predicate field type
 */
SECTION Field_Type
{
/* p -> predicate register */
p ();
any (compatible_with||(p));
pany (compatible_with(p NULL));
}

/*
 * Specifies the architecture's assembly language operation formats
 *
 * Declare operation formats for compiler directives and
 * emulated operations
 *
 * Add two optional predicate fields for each operation format.
 */
SECTION Operation_Format
{

    // Standard operation formats
    OF_Std1             (pred(pany pany));
    OF_Std2             (pred(pany pany));
    OF_Std3             (pred(pany pany));

    OF_ZVDEPI           (pred(pany pany));
    OF_ZVDEP            (pred(pany pany));
    OF_VSHD             (pred(pany pany));
    OF_EXTRU            (pred(pany pany));
    OF_DEPI             (pred(pany pany));

    OF_Lit5             (pred(pany pany));
    OF_Lit11            (pred(pany pany));
    OF_Lit14            (pred(pany pany));
    OF_Lit21_Mov        (pred(pany pany));
    OF_Lit21_Add        (pred(pany pany));
    OF_Label_Add        (pred(pany pany));
    OF_Label_Mov        (pred(pany pany));

    // Operation format for "move"
    OF_Mov              (pred(pany pany));

    // Operation format for fp 5op
    OF_5op              (pred(pany pany));
```

```
    // Operation format for unconditional branch
    OF_Rts              (pred(pany pany));
    OF_Jsr1             (pred(pany pany));
    OF_Jsr2             (pred(pany pany));
    OF_Ubr1             (pred(pany pany));
    OF_Ubr2             (pred(pany pany));
    OF_Ubr3             (pred(pany pany));

    // Operation format for conditional branch
    OF_Cbr1             (pred(pany pany));
    OF_Cbr2             (pred(pany pany));
    OF_Cbr_BB           (pred(pany pany));

    // Operation format for Stores
    OF_Store1           (pred(pany pany));
    OF_Store2           (pred(pany pany));
    OF_Store2_Label     (pred(pany pany));
    OF_Store3           (pred(pany pany));
        // pre/post inc short store
    OF_Store4           (pred(pany pany));
        // pre/post inc long store
    OF_Store5           (pred(pany pany));
        // post inc index store
    OF_Store6           (pred(pany pany));

    // Operation format for loads
    OF_Load1            (pred(pany pany));
    OF_Load2            (pred(pany pany));
    OF_Load2_Reg        (pred(pany pany));
    OF_Load2_Label      (pred(pany pany));
    OF_Load3            (pred(pany pany));
        // pre/post inc short load
    OF_Load4            (pred(pany pany));
        // pre/post inc long load
    OF_Load5            (pred(pany pany));
        // post inc index load
    OF_Load6            (pred(pany pany));

       // Nop
    OF_Nil      (pred(pany pany));


    // For compiler directives
    OF_Directive   (pred(pany pany) dest(any any) src(any any any any));

    // For operations emulated by Lhppa
    OF_Emulation    (pred(pany pany) dest(any any) src(any any any any));
}
```

```
/*
 * Specifies how resources are used as operations execute.
 *
 * Create resources usages for compiler directives and emulated operations
 */
SECTION Reservation_Table
{
    // For compiler directives
    RT_Directive (use(any_iissue));

    // For operations emulated by Lhppa
    RT_Emulation (use(any_iissue));
}

/*
 * Used to specify how register and sync operands interact so that
 * a dependence distance can be calculated.
 *
 * For now, a list of possible use times are specified for each operand.
 */
SECTION Operand_Latency
{
    p0 (time(0));
}

/*
 * The section is used to specify the dependence distances between
 * operations (if dependence exists).  All dependences (register,
 * memory, control, sync, etc) are modeled as flow,
 * output, and anti dependencies between operands.
 *
 * Uses Operand_Latency above to specify how operands interact.
 *
 * Specify latency 1 for emulated operations and compiler
 * directives (compiler directives will not be scheduled so
 * it really doesn't matter).
 *
 * Add Operand_Latency specifiers for predicate operands to
 * can emulate predicated code with Lhppa.  May have two predicate
 * operands (the second operand is used for debugging predicate
 * promotion, and does not effect execution of operation)
 *
 */
SECTION Operation_Latency
{
   // For compiler directives, dependence graph should not be drawing arcs
   // to or from these
   OL_Directive (pred(x x)
 dest(x x)
        src(x x x x)
 mem_dest(x)
```

```
 ctrl_dest(x)
 sync_dest(x)
 mem_src(x)
 ctrl_src(x)
 sync_src(x));

   // For operations emulated by Lhppa, model as latency 1
   OL_Emulation (pred(p0 p0)
                 dest(d1 d1)
         src(s0 s0 s0 s0)
 mem_dest(sd0)
 ctrl_dest(sd0)
 sync_dest(sd0)
 mem_src(ss0)
 ctrl_src(ss0)
 sync_src(ss0));

   /*
    * Add predicated operands for all Operation_Latency types
    * specified in the pristine hmdes2
    */
   $for(N in 1 2 3 8 15)
   {
     OL_Lat${N} (pred(p0 p0));
   }

   OL_Branch    (pred(p0 p0));
   OL_Store     (pred(p0 p0));
   OL_Label_Store (pred(p0 p0));
   OL_Load     (pred(p0 p0));
}


/*
 * Describe the scheduling alternatives for the operations this
 * processor supports.
 *
 * Allow multiple operation formats to be specified
 * (for operations that have multiple operation formats but behave the same).
 *
 * Specify scheduling alternatives for compiler directives and emulation
 * operations.
 */
SECTION Scheduling_Alternative
{
   // Add squashing flags for branch alternatives
   ALT_Cbr_COMB_FWD     (flags(NT));
   ALT_Cbr_COMB_BWD     (flags(NN));
   ALT_Cbr_COMIB_FWD    (flags(NT));
   ALT_Cbr_COMIB_BWD    (flags(NN));
   ALT_Cbr_ADDIB_FWD    (flags(NT));
```

```
   ALT_Cbr_ADDIB_BWD    (flags(NN));
   ALT_Cbr_BB_FWD       (flags(NT));
   ALT_Cbr_BB_BWD       (flags(NN));

   ALT_Rts (flags(NT));
   ALT_Jump (flags(NT));
   ALT_Jump_RG (flags(NT));


   // For compiler directives
   ALT_Directive (format(OF_Directive) resv(RT_Directive)
          latency(OL_Directive));

   // For operations emulated by Lhppa
   ALT_Emulation (format(OF_Emulation) resv(RT_Emulation)
          latency(OL_Emulation));
}


/*
 * Describe the operations for this architecture
 *
 * The scheduling alternatives for each operation is specified.
 */
SECTION Operation
{
   // For compiler directives
   OP_DIRECTIVE (alt(ALT_Directive));

   // For operations emulated by Lhppa
   OP_EMULATION (alt(ALT_Emulation));
}


/*
 * IMPACT operation section.
 *
 * Maps impact names to operations in the pristine hmdes2.
 * Specifies flags about the operation that the compiler needs.
 */
SECTION IMPACT_Operation
{
   // NOP
   Lop_NO_OP (op(OP_NOP));

   /*
    *  Integer conditional branch operations
    */
   $for (CASE in EQ NE GT GE LT LE GT_U GE_U LT_U LE_U)
   {
      LHPPAop_COMB_${CASE}_FWD (flags(CBR NI) op(OP_COMB_${CASE}_FWD));
```

```
   LHPPAop_COMIB_${CASE}_FWD (flags(CBR NI) op(OP_COMIB_${CASE}_FWD));
   LHPPAop_COMB_${CASE}_BWD (flags(CBR NI) op(OP_COMB_${CASE}_BWD));
   LHPPAop_COMIB_${CASE}_BWD (flags(CBR NI) op(OP_COMIB_${CASE}_BWD));
}

// Lhppa only uses the LT version
LHPPAop_ADDIB_LT_FWD (flags(CBR NI) op(OP_ADDIB_LT_FWD));
LHPPAop_ADDIB_LT_BWD (flags(CBR NI) op(OP_ADDIB_LT_BWD));

// Branch on bit
$for (CASE in BB_0 BB_1)
{
   LHPPAop_${CASE}_FWD (flags(CBR NI) op(OP_${CASE}_FWD));
   LHPPAop_${CASE}_BWD (flags(CBR NI) op(OP_${CASE}_BWD));
}

// Unconditional branches
Lop_JSR (flags(JSR NI) op(OP_JSR));
Lop_JSR_FS (flags(JSR NI) op(OP_JSR));
Lop_JSR_ND (flags(JSR NI) op(OP_JSR));

// Dynamic subroutine call doesn't have slot to fill
LHPPAop_JSR_DYNCALL (flags(JSR) op(OP_JSR_DYNCALL));

Lop_RTS (flags(RTS NI) op(OP_RTS));
Lop_RTS_FS (flags(RTS NI) op(OP_RTS));
Lop_JUMP (flags(JMP NI) op(OP_JUMP));
Lop_JUMP_FS (flags(JMP NI) op(OP_JUMP));
Lop_JUMP_RG (flags(JMP NI) op(OP_JUMP_RG));
Lop_JUMP_RG_FS (flags(JMP NI) op(OP_JUMP_RG));


/*
 * Integer ALU operations
 */
Lop_MOV (op(OP_COPY OP_LDO_MOV OP_LDIL));
LHPPAop_LDIL (op(OP_LDIL));

LHPPAop_LDO (op(OP_LDO_ADD));
LHPPAop_ADDIL (op(OP_ADDIL));

Lop_ADD  (op(OP_ADD OP_ADDI));
Lop_SUB (op(OP_SUB OP_SUBI));

$for (OPCODE in ZVDEPI ZVDEP VSHD VEXTRS)
{
  LHPPAop_${OPCODE} (op(OP_${OPCODE}));
}

// logical operations
$for (OPCODE in OR AND XOR AND_COMPL)
```

```
  {
    Lop_${OPCODE} (op(OP_${OPCODE}));
  }

  // shift operations
  $for (OPCODE in LSL ASR LSR)
  {
    Lop_${OPCODE} (op(OP_${OPCODE}));
  }



  // Integer comparison operations
  $for(CMP in EQ NE GT GE LT LE GT_U GE_U LT_U LE_U)
  {
    Lop_${CMP} (flags(EXPANDS) op(OP_${CMP}));
  }

  /*
   * Floating point operations
   */
  $for(OPCODE in MOV_F MOV_F2 ABS_F ABS_F2)
  {
    Lop_${OPCODE} (flags() op(OP_${OPCODE}));
  }

  $for(OPCODE in I_F I_F2 F2_I F_I F_F2 F2_F)
  {
    Lop_${OPCODE} (flags(EXCEPT) op(OP_${OPCODE}));
  }

  $for(OPCODE in ADD SUB EQ GT GE NE LT LE)
  {
     $for(TYPE in F F2)
     {
Lop_${OPCODE}_${TYPE} (flags(EXCEPT) op(OP_${OPCODE}));
     }
  }

  Lop_MUL_F (flags(EXCEPT) op(OP_MUL_F));
  Lop_MUL_F2 (flags(EXCEPT) op(OP_MUL_F2));
  Lop_DIV_F (flags(EXCEPT) op(OP_DIV_F));
  Lop_DIV_F2 (flags(EXCEPT) op(OP_DIV_F2));
  Lop_SQRT_F (flags(EXCEPT) op(OP_SQRT_F));
  Lop_SQRT_F2 (flags(EXCEPT) op(OP_SQRT_F2));

  // Floating-point 5ops operations
  LHPPAop_FMPYADD (flags(EXCEPT) op(OP_FMPYADD));
  LHPPAop_FMPYSUB (flags(EXCEPT) op(OP_FMPYSUB));
```

```
  /*
   * The rather extensive collection of HP-PA load operations
   */
  $for(TYPE in C C2 I F F2)
  {
     Lop_LD_${TYPE} (flags(LOAD EXCEPT)
        op (OP_LD_SHORT_${TYPE}
    OP_LD_INDEX_${TYPE}));
     Lop_LD_PRE_${TYPE} (flags(LOAD EXCEPT)
op(OP_LD_SHORT_PRE_${TYPE}));
     Lop_LD_POST_${TYPE}   (flags(LOAD EXCEPT)
     op(OP_LD_SHORT_POST_${TYPE}
           OP_LD_INDEX_POST_${TYPE}));
  }

  // The C C2 and I loads have long versions
  Lop_LD_C (op||(OP_LD_LONG_C));
  Lop_LD_C2 (op||(OP_LD_LONG_C2));
  Lop_LD_I (op||(OP_LD_LONG_I));

  // Int loads also has long increment versions
  Lop_LD_PRE_I   (op||(OP_LD_LONG_PRE_I));
  Lop_LD_POST_I (op||(OP_LD_LONG_POST_I));


 /*
  * The almost as extensive collection of HP-PA store operations
  */
  $for(TYPE in C C2 I)
  {
     Lop_ST_${TYPE} (flags(STORE EXCEPT)
op(OP_ST_SHORT_${TYPE}
   OP_ST_LONG_${TYPE}));
     Lop_ST_PRE_${TYPE}   (flags(STORE EXCEPT)
op(OP_ST_SHORT_PRE_${TYPE}));
     Lop_ST_POST_${TYPE}  (flags(STORE EXCEPT)
op(OP_ST_SHORT_POST_${TYPE}));
  }

  // Int stores also has long increment versions
  Lop_ST_PRE_I   (op||(OP_ST_LONG_PRE_I));
  Lop_ST_POST_I   (op||(OP_ST_LONG_POST_I));

  $for(TYPE in F F2)
  {
     Lop_ST_${TYPE} (flags(STORE EXCEPT)
op(OP_ST_SHORT_${TYPE}
   OP_ST_INDEX_${TYPE}));
     Lop_ST_PRE_${TYPE}   (flags(STORE EXCEPT)
op(OP_ST_SHORT_PRE_${TYPE}));
     Lop_ST_POST_${TYPE}  (flags(STORE EXCEPT)
```

```
 op(OP_ST_SHORT_POST_${TYPE}
    OP_ST_INDEX_POST_${TYPE}));
   }


   /*
    * Compiler directives.  Ignored by the scheduler
    */
   $for(DIRECTIVE in DEFINE ALLOC EPILOGUE PROLOGUE)
   {
      Lop_${DIRECTIVE} (flags(IGNORE) op(OP_DIRECTIVE));
   }

   /*
    * Operation emulated by Lhppa
    */
   LHPPAop_PRED_MOV (op(OP_EMULATION));

   $for(OPCODE in PRED_CLEAR PRED_LD PRED_ST PRED_ST_BLK PRED_LD_BLK
  CMOV CMOV_COM CMOV_F CMOV_COM_F CMOV_F2 CMOV_COM_F2
  AND_NOT OR_NOT SELECT SELECT_F SELECT_F2 PREF_LD)
   {
     Lop_${OPCODE} (op(OP_EMULATION));
   }

   // Predicate setting operations
   $for(CASE in EQ NE GT GT_U GE GE_U LT LT_U LE LE_U)
   {
     Lop_PRED_${CASE} (op(OP_EMULATION));
   }

   $for(CASE in EQ NE GT GE LT LE)
   {
     Lop_PRED_${CASE}_F (op(OP_EMULATION));
     Lop_PRED_${CASE}_F2 (op(OP_EMULATION));
   }

}
```