A MACHINE DESCRIPTION LANGUAGE FOR COMPILATION

BY

JOHN CHRISTOPHER GYLLENHAAL

B.S, University of Arizona, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. I would also like to thank the IMPACT group, in general, and Rick Hank, Scott Mahlke and Roger Bringmann, in particular, for their helpful discussions. Several people in the Hewlett Packard Laboratories' FAST Group have provided valuable guidance and feedback, especially Bob Rau, Vinod Kathail, and Michael Schlansker. For their three years of financial support through a fellowship, I would like to thank the National Science Foundation. I would like to express my appreciation to my parents for all the encouragement and support throughout my college years. Finally, I wish to thank and express my love for my fiancée Liesl Little who has encouraged me throughout graduate school and who provided invaluable editing help as the deadline for this thesis loomed.

"It's a damn poor mind that can think of only one way to spell a word."
-Andrew Jackson

TABLE OF CONTENTS

1. INTRODUCTION

This document describes the various aspects of the machine description (MDES) language. It was developed jointly by the IMPACT group at the University of Illinois and the FAST group at Hewlett Packard Laboratories with the goal of describing a processor's resources, and how the processor's instruction set uses these resources, in enough detail that a compiler could schedule for that processor using only this information. The realization of this goal makes the MDES language a powerful tool for studying different processor architectures. The MDES language has been used to model the PA-RISC family of processors [1], the AM29000 family of processors [2], the X86 family of processors [3], and the Sparc family of processors [4] [5], as well as several architectual extensions to these processors. The IMPACT compiler [6] has been successfully scheduling based on MDES files since the summer of 1993.

There were several design goals: 1) The language should be organized in a way intuitive for the writer of the MDES with features designed to reduce the tedium of designing and modifying the MDES; 2) The compiler should be able to quickly load and use the MDES information without having to deal with syntax errors and typos; 3) The MDES language should be compiler independent.

In order to meet the conflicting design goals of 1 and 2, the MDES was split into two different forms. To meet the first goal, the high-level machine description (HMDES) is organized in an intuitive way for the writer, allowing comments, text substitution, and flexibility concerning indentation/formatting. To meet the second goal of efficiency, the HMDES file is then translated into a low-level machine description (LMDES). During this translation, powerful preprocessing algorithms are used and the HMDES grammar is extensively checked for errors and typos. After a successful translation, the LMDES file is free of syntax errors and detectable typos. During compilation, the LMDES can be quickly loaded into memory because minimal error checking is needed. The organization of the MDES in memory is designed to facilitate efficient retrieval of information by the compiler.

The third goal is met by designing the MDES compiler interface functions with minimal assumptions about the compiler's underlying structure. The basic MDES interface functions do not use data structures internal to the compiler. These basic functions may then be used to construct more powerful compiler-specific functions.

Figure 1.1 shows a schematic overview of the path followed in translating an HMDES file to the MDES internal representation that the compiler interface functions can efficiently access. The following chapters give a detailed description of the different aspects of the MDES and the translation process. Chapter 2 describes the syntax and purpose

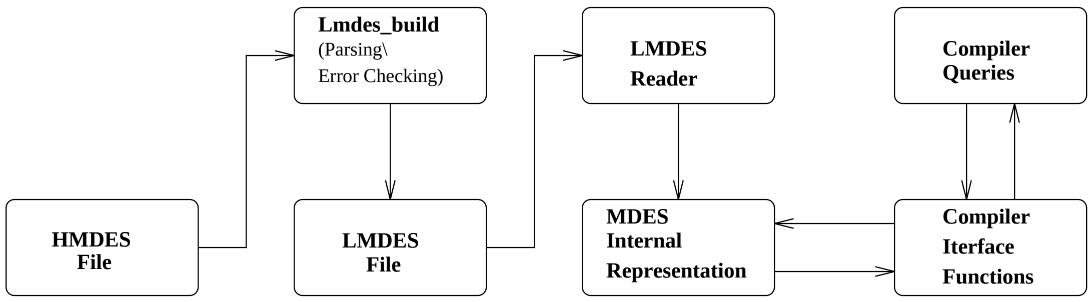

Figure 1.1: Schematic overview of the MDES translation path

of each of the HMDES language features. Chapter 3 describes the HMDES-to-LMDES translator and its command line arguments. Chapter 4 describes the LMDES language, including the format conventions, the representation of the HMDES information, and LMDES reader issues. Chapter 5 outlines the internal data structures used to represent the MDES. Chapter 6 describes the MDES interface functions to the compiler including some more powerful IMPACT-specific interface functions.

The writing of this document has two main goals. The first of these is to provide a detailed description of the MDES language so that a machine description for a processor can be written and used (Chapters 2, 3, and 6). The second goal is to provide documentation of all the innerworkings of MDES so that it may be further enhanced to meet the requirements of more complex processor models (Chapters 4 and 5). The following chapters provide the information necessary to meet these goals.

2. HMDES SYNTAX

2.1 Introduction

The HMDES language was designed to be human readable and modifiable. As a consequence, a machine description may contain syntax errors and it is fairly expensive to parse. To prevent the compiler from having to deal with these problems, the HMDES language is first converted into the LMDES language using a translation program. This allows the HMDES writer to have a user-friendly syntax checker and the compiler to load the machine description quickly using the LMDES file. This chapter first describes some general HMDES features and then explains the syntax of each HMDES section.

2.2 HMDES Preprocessor

The HMDES language includes two features normally associated with a preprocessing step: line comments and text replacement directives. Both features may be used anywhere within the HMDES file and conceptually are handled before any HMDES to LMDES translation takes place.

Line comments start with either a '#' or a '%' and terminate at the end of the line, as shown in the following example.

Text # comment text

Text replacement directives (definitions) have the following form.

$define *name replacement_text*

Subsequent occurrences of $*name*$ will be replaced by the *replacement_text*. The name in the $define has the same form as a C variable name and the replacement text ends at the newline and has leading and trailing whitespace removed. For example, if HEADER_DIR is defined as

$define HEADER_DIR /home/polaris1/gyllen/mthesis # comment

then the line

C_header_file "$HEADER_DIR$/ioform.h"

will be replaced with

C_header_file "/home/polaris1/gyllen/mthesis/ioform.h"

in the HMDES file. The text replacement directive may use previous definitions, which are evaluated when the directive is read in. Comments are removed before the directive is processed, so it is safe to put a comment on the same line as a $define as shown in the example above. A name may be redefined with another text replacement directive.

Currently, a text replacement directive may not be split across more than one line and a $ may only be used in the context of a $define or $name$.

Text replacement directives may also be issued on the command line during the translation from HMDES to LMDES, and in the user's environment. This will be covered in Chapter 3, which deals with translating an HMDES file into an LMDES file.

2.3 Format Considerations

In HMDES, *whitespace* is defined to be one or more spaces, tabs, newlines or commas. Indentation and newline conventions are left to the HMDES writer. Commas are considered whitespace mainly to allow lists of numbers to be separated by commas, but they may be used anywhere.

Keywords in HMDES are not case-sensitive. The HMDES keywords are: the *version1* at the top of the file, all the section names (such as *Define* and *Register_Files*), the word *declaration* that follows each section's name, the *end* that ends a section, the parameter names in the define section (such as *C_header_file* and *processor_model*), and *capacity* and *width* in the Register_Files section. Keywords are not reserved words, so they may be used in other contexts.

All other identifiers (those created by the HMDES writer) are case-sensitive. These identifiers must start with a letter of the alphabet or an underscore, and may contain letters, numbers and underscores. Names must be unique within their HMDES section, but can be defined in more than one section if desired. However, this is not recommended.

```
/*
 * Subset of AMD specific MDES IO_set specifiers (renumbered),
 * taken from machine/m_amd.h
 */
#define MDES_OPERAND_NULL       0
#define MDES_OPERAND_GPR        1
#define MDES_OPERAND_Lit8       2
#define MDES_OPERAND_Lit16      3
#define MDES_OPERAND_REG        4
#define MDES_OPERAND_NIL        5
```

Figure 2.1: #defines from ioform.h which are used by form.hmdes to define IO_Set specifiers

2.4 Compiler Interface Considerations

In C code, it is easier and more efficient to identify items with an integer number than with a string. By creating C header files that contain "#define" statements, names can be used to represent integers in C programs with no loss of efficiency. Similarly, at the MDES compiler interface, it is more efficient to to use integers than it is to use the names defined in the HMDES file. By using the same C header files that were created for the C program (i.e., the compiler), integers can be used at the compiler interface to represent the names in the HMDES file. These integers are referred to as *external* IDs. The use of integer external IDs increases the efficiency of the MDES interface and saves work because the programmer does not need to write a name-to-integer mapping file for the MDES. The C header files used by the sample HMDES file, form.hmdes (see Appendix A), used in this chapter are shown in Figures 2.1, 2.2, and 2.3. The C header files are also shown in their unaltered form in Appendices B, C, and D. It should be

```
/*
 * Subset of standard Lcode MDES Operation flags (renumbered),
 * taken from Lcode/l_flags.h
 */
#define OP_FLAG_CBR    0x00000001
#define OP_FLAG_RTS    0x00000002
#define OP_FLAG_JSR    0x00000004
#define OP_FLAG_LOAD   0x00000008
#define OP_FLAG_STORE  0x00000010
#define OP_FLAG_EXCEPT 0x00000020 /* Can cause exception */
#define OP_FLAG_NI     0x00000040 /* non-interlocking, has delay slot */

/*
 * Subset of standard Lcode MDES Alternative flags (renumbered),
 * taken from Lcode/l_flags.h
 * Not used by form.hmdes, but included to show the name format.
 */
#define ALT_FLAG_SPEC  0x00000001 /* speculative form of instr */
```

Figure 2.2: #defines from flagform.h which are used by form.hmdes to define operation and alternative flags

noted that the HMDES file uses these C header files differently than the C program does. The differences are described below.

The first difference is that in C programs, the names are actually replaced by integers by the C preprocessor. In HMDES to LMDES translation, the names are annotated with the integer that stands for the name. The name used in the HMDES is not lost. This allows HMDES names to be used in error messages by the compiler.

The second difference is that in C programs, the names defined in the “#define” statements are exactly what is used in the C program. In HMDES files, the names defined in the “#define” statements sometimes have their prefix removed when used in

```
/*
 * Subset of standard Lcode opcodes (renumbered),
 * taken from Lcode/l_opc.h
 */
#define Lop_BEQ         1
#define Lop_BNE         2
#define Lop_RTS         3
#define Lop_JSR         4
#define Lop_ADD         5
#define Lop_LD_I        6
#define Lop_ST_I        7

/*
 * Subset of AMD processor specific opcode (renumbered),
 * taken from Lcode/lamd_phase1.h.
 */
#define LAMDop_SUBR     500
```

Figure 2.3: #defines from opcform.h which are used by form.hmdes to define opcodes

the HMDES file. For example, the name "MDES_OPERAND_GPR" that is defined in the C header file shown in Figure 2.1 is referred to as "GPR" in the sample HMDES file form.hmdes. The prefix "MDES_OPERAND_" is required in the C header file to make the name unique but is unwieldy to use in the HMDES file.

The third difference is that in C programs, the C header files can include other C header files. In HMDES header files, the "#include" statements are ignored. This means all of the C header files have to be explicitly specified (in the define section, see Section 2.7).

The last difference is that in C programs, enumeration commands can be used to assign numbers to names. In HMDES files, an error will occur if an enumeration is encountered because the case is not currently handled.

2.5 HMDES Section Delimiters

There are nine sections in an HMDES file. They all begin with

(*section_name* declaration

and end with

end)

where *section_name* is the section name. The section names and the keywords 'declaration' and 'end' are not case sensitive.

All the sections described in this chapter must be present in the HMDES file and they must appear in the order described in this chapter. This ordering is required to allow all the references to other sections to be resolved in one pass of the HMDES file.

2.6 HMDES Version

In the HMDES file, before any section declarations, the HMDES version is indicated. The version, which for this document is version 1, is specified in an HMDES file by a "Version1" before the Defines section declaration.

```
(Define declaration
    C_header_file   "/home/polaris1/gyllen/mthesis/ioform.h"
    C_header_file   "/home/polaris1/gyllen/mthesis/opcform.h"
    C_header_file   "/home/polaris1/gyllen/mthesis/flagform.h"
    predicates      0
    dest_operands   2
    dest_syncs      3   # Used to construct control/sync dependences
    source_operands 4
    src_syncs       3   # Used to construct control/sync dependences
    processor_model     superscalar
end)
```

Figure 2.4: Example Define section from the form.hmdes file

2.7 Define

The Define section specifies the number of predicate, destination, source and synchronization operands that the processor's instruction set supports and the type of processor (superscalar or VLIW) that is being modeled. Additionally, the complete path to one or more C header files are specified. These C header files are used by the HMDES to LMDES translator to map register files, IO sets, and operations to integer numbers. This is an annotation step, not a preprocessing step since the names remain in the file. An example of an HMDES Define section is shown in Figure 2.4.

As can be seen in the example, each entry in the definition consists of a keyword followed by a string or an integer. The keywords are not case sensitive and the order of the entries does not matter. Quotes should be used around any string that contains punctuation characters or spaces. The following must be defined in the Define section:

1. *predicates* is the number of predicate operands,
2. *dest_operands* is the number of destination operands,

3. *source_operands* is the number of source operands,
4. *src_sync* is the number of incoming synchronization operands,
5. *dest_sync* is the number of outgoing synchronization operands,
6. *processor_model* is the processor model.

The current valid processor models are VLIW or superscalar.

Additionally, header files to be used to annotate the LMDES files may be specified using the C_header_file keyword. This keyword may be used as many times as necessary to specify all the desired header files. The complete path to each header file should be specified and the path should be quoted to prevent the punctuation marks such as "/" and "." from being misinterpreted.

2.8 Register_Files

The Register_Files section was originally intended to define the capacity and width of the processor's register files. At the present time, neither the capacity nor width is being used by any IMPACT program. In fact, the capacity and width information is not even stored in the LMDES file. Instead, this section is used to define the operand types allowed in the processor's assembly language.

The entries in this section will be used to enumerate the legal register types and constant sizes. These names will be used later in the HMDES file to specify what the operand types may be for the instructions in the processor's instruction set. These names will also be used, with an "MDES_OPERAND_" prefix, by the compiler to specify an

```
(Register_Files declaration
    NULL ((capacity 0 0)   (width 0))
    GPR  ((capacity 191 0) (width 32))

    # 8 bit field for most arithmetic opcodes
    Lit8 ((capacity 0 0)   (width 8))

    # 16 bit field for const and branch direct
    Lit16 ((capacity 0 0)  (width 16))
end)
```

Figure 2.5: Example Register_Files section from the form.hmdes file

instruction's operands. For more information on how the compiler will use these names and the MDES_OPERAND_prefix, please see Section 2.4.

An example Register_Files section is shown in Figure 2.5. Although only the name is currently used, the following is the official format for each entry.

name ((capacity *static_size rotating_size*) (width *width_in_bits*))

where

1. *name* is the register file's name or operand type's name,
2. *static_size* is the number of static registers in the file,
3. *rotating_size* is the number of rotating registers in the file,
4. *width_in_bits* is the register width in bits.

A required Register_File entry is for the NULL register file. This Register_File entry is implicitly used wherever no operand is expected.

```
(IO_Sets declaration
    REG (GPR)
    ANY (NULL GPR Lit8 Lit16)
    LIT (Lit8 Lit16)
end)
```

Figure 2.6: Example IO_Sets section from the form.hmdes file

2.9 IO_Sets

The entries in the IO_Sets section give a name to a set of register files. This name may be used anywhere a Register_Files entry name can be used in the IO_Items section (see Section 2.10) and indicates that any of the specified register files are valid as that operand.

An example IO_Sets section is shown in Figure 2.6. The example section shows how IO_Sets entries may be used to group register files into more manageable groupings. For example, the IO_Sets entry "ANY" contains all the valid possible operand types and may be used instead of a Register_Files entry name to specify an operand type in an IO_Item. Although it is not illustrated in the example, IO_Sets definitions may include previously defined IO_Sets entry names.

2.10 IO_Items

The entries in this section give names to legal combinations of operands. For each operand in an IO_Items entry, a Register_Files or IO_Sets entry name is used to specify what that operand may be. This information will be used later to bind an opcode with a

```
(IO_Items declaration
    IOI_Store   ([][REG ANY REG])

    IOI_Load    ([REG][REG ANY])

    # Standard operand format
    IOI_Std     ([REG][REG REG])
    IOI_Stdi    ([REG][REG Lit8])

    IOI_ubri    ([][REG])           # uncond indirect

    IOI_cbr     ([][REG Lit8 LIT])  # cond
    IOI_cbri    ([][REG Lit8 REG])  # cond indirect

    IOI_call    ([REG][LIT -])      # call direct
    IOI_calli   ([REG][REG])        # call indirect
end)
```

Figure 2.7: Example IO_Items section from the form.hmdes file

particular set of operands to the resources and latencies associated with that operation. The format of an IO_Items entry follows.

IO_Item_name (<$pred_0$ $pred_1$... > [$dest_0$ $dest_1$...][src_0 src_1...])

where

1. *IO_Item_name* is the entry's name,
2. $pred_0$ $pred_1$... are the predicate operands (the number specified by pred_operands in the define section),
3. $dest_0$ $dest_1$... are the destination operands (the number specified by dest_operands in the define section),
4. src_0 src_1... are the source operands (the number specified by src_operands in the define section).

Trailing operand types may be omitted. If omitted they default to the "NULL" register file, which is required to be defined in the Register_Files section. If no predicate

```
(Resources declaration
    slot[0]
    ialu0
end)
```

Figure 2.8: Example Resources section from the form.hmdes file

operands are specified, the <> are not needed. An example IO_Items section is shown in Figure 2.7.

2.11 Resources

The entries in this section define all the resources that are used in the HMDES to model the processor. An example Resources section is shown in Figure 2.8. For a given resource entry, either a single resource can be defined,

resource_name or equivalently *resource_name[0]*,

or an array of resources.

resource_name[0..3] or equivalently *resource_name[0, 1, 2, 3]*

Arrays of resources are convenient when an operation may use one of a set of resources interchangeably (i.e., register ports). The integers in the index list must be non-negative, may be specified in any order and may skip numbers, if desired.

resource_name[3, 5, 1]

The order of the indexes specifies the default order in which the instruction scheduler will try to allocate the resources for an operation. This order may be overridden in the ResTables section (see Section 2.12).

```
(ResTables declaration
    RL_IBr  (
            (slot[0] 0)
            )
    RL_IAlu (
            (slot[0] 0)
            (ialu0 0)
            )
end)
```

Figure 2.9: Example ResTables section from the form.hmdes file

A required Resources entry is for the "slot" resource or array of resources. This Resources entry defines the number of operations that may be scheduled in a cycle. In the ResTables section, each operation is required to specify which slots it may be scheduled in. This allows decoder restrictions to be modeled. To simplify scheduling algorithms, all slot indexes between 0 and the max index must be defined.

2.12 ResTables

The entries in the ResTables section define the ways operations can use resources over time. An example ResTables section is shown in Figure 2.9. Each ResTable entry is composed of the entry's name and a list of resource usage entries.

$ResTable\_name\ (resource\_usage\_entry_0 \ldots)$

Each resource usage entry consists of a resource name with an optional set of indexes and a resource usage start and stop time. Each index denotes a particular resource usage option, where any one of these usage options may be selected.

```
(Latencies declaration
    Lat1  (1 () (1 0) () ()    ())
    Lat1m (2 () (1 0) () ()    (1)) # for stores
    Lat2  (2 () (1 0) () (0 1) ())  # for control
    Lat2m (2 () (2 0) () (1)   ())  # for loads
end)
```

Figure 2.10: Example Latencies section from the form.hmdes file

(*resource_name[index*$_1$*... index*$_n$*] start..stop*)

The order of the indexes specifies the order the scheduler will try to allocate the resource options. The *[index*$_1$ *... index*$_n$*]* may be omitted if the *resource_name* does not refer to an array of resources or if any of the resources in the array may be used and should be attempted to be scheduled in the default order (see Section 2.11).

The start and stop times are with respect to the cycle during which the instruction leaves the decoder (cycle 0). The *..stop* may be omitted if the stop time is the same as the start time. The start time may not be negative and the stop time must be greater than or equal to the start time.

Each ResTable entry must contain a resource usage entry that specifies to which "slots" the operation may be scheduled. The slot must be used only for cycle 0.

2.13 Latencies

An example Latencies section is shown in Figure 2.10. The entries in the Latencies section specify the cycle during which each register operand is read (predicate, source, or incoming sync operands) or written (destinations or outgoing sync operands). This

```
(Operation_class declaration
    CL_cbr
        ( # conditional branches
            (IOI_cbr RL_IBr Lat2)
        )
    CL_Std_IAlu
        ( # Standard one cycle IALU instructions
            (IOI_Std RL_IAlu Lat1)
            (IOI_Stdi RL_IAlu Lat1)
        )
    CL_Load
        ( # 2 Cycle load instructions
            (IOI_Load RL_IAlu Lat2m)
        )
    CL_Store
        ( # 1 Cycle store instructions
            (IOI_Store RL_IAlu Lat1m)
        )
end)
```

Figure 2.11: Example Operation_Class section from the form.hmdes file

information is used to calculate dependence distances for flow, anti, output, and control dependencies. The entries also specify when exceptions are handled.

$latency\_name\,(exception\_latency\,(pred_0 \ldots)\,(dest_0 \ldots)\,(src_0 \ldots)\,(dest\_sync_0 \ldots)\,(src\_sync_0 \ldots))$

The latencies that are not specified are assumed to be used at cycle 0. Typically, all of the source registers are read at cycle 0 (decode time) and all of the destination registers are written when the instruction leaves the functional unit.

2.14 Operation_Class

The entries in the Operation_Class section are used to describe classes of instructions, where instructions in a class all expect the same operands, use the same resources in the processor and have the same operand latencies. The use of operation classes is optional since all the same information can be also be placed directly in each Operations section entry (see Section 2.15). Each entry contains a list of one or more alternatives, where each alternative represents a different way the instruction can behave.

$Operation\_Class\_name$ $(alternative_1 \ldots)$

Each *alternative* is represented by

$(IO\_Set\_name\ ResTable\_name\ Latency\_name)$

The scheduler, when searching for an alternative to schedule, will test the alternatives in the order that they are listed in the Operation_Class entry. An example Operation_Class section is shown in Figure 2.11.

2.15 Operations

The Operations section entries associate an operation's opcode with opcode flags, assembly name, assembly flags and either an Operation_Class name or direct specification of scheduling alternatives. An example Operations section is shown in Figure 2.12. The two possible forms are:

$opcode\_name <opcode\_flag_0 \ldots> (assembly\_name <assembly\_flag_0 \ldots> operation\_class)$

or

$opcode\_name <opcode\_flag_0 \ldots> (assembly\_name <assembly\_flag_0 \ldots> (alternative_0 \ldots))$

where each *alternative* has the form

$(IO\_Set\_name\ ResTable\_name\ Latency\_name)$

```
(Operations declaration
    Lop_BEQ <CBR NI>        (jmpt   CL_cbr)
    Lop_BNE <CBR NI>        (jmpf   CL_cbr)
    Lop_RTS <RTS NI>        (jmpi   ((IOI_ubri RL_IBr Lat2)))
    Lop_JSR <JSR NI>        (call   ((IOI_call RL_IBr Lat2)))
    Lop_JSR <JSR NI>        (calli  ((IOI_calli RL_IBr Lat2)))
    Lop_ADD                 (add    CL_Std_IAlu)
    LAMDop_SUBR             (subr   CL_Std_IAlu)
    Lop_LD_I <EXCEPT LOAD>  (load   CL_Load)
    Lop_ST_I <EXCEPT STORE> (store  CL_Store)
end)
```

Figure 2.12: Example Operations section from the form.hmdes file

The *opcode_name* specifies the opcode of the operation being described. An integer will be associated with this opcode_name using the C header files as specified in Section 2.4. No prefix is attached to the opcode names during lookup since they are assumed to be unique. Multiple entries with the same opcode_name may be used to specify different assembly names and/or assembly flags for different scheduling alternatives.

The *opcode_flags* allow opcode specific information to be specified in the HMDES. These flags must be defined in one of the C header files (see Section 2.4) with an 'OP_FLAG_' prefix. The values for the opcode flags are treated as bit flags and are ORed together to form a 32-bit wide bit field for that opcode. Possible uses of flags include specifying the type of operation, whether the instruction can generate exceptions, or whether a branch has a delay slot. Entries with the same opcode name must have the same opcode flags specified and if no opcode flags are specified, the <> may be omitted.

The *assembly_name* specifies the assembly name for all the alternatives in that entry. Sometimes the operation's operand types and which function unit the operation was scheduled for are encoded into the operation's assembly name, requiring different assembly names for the operation's alternatives. To handle this case, the HMDES writer must create a separate Operations entry for each assembly name.

The *assembly_flags* allow alternative specific information to be specified in the HMDES. These flags must be defined in one of the C header files (see Section 2.4) with an 'ALT_FLAG_' prefix. Like opcode flags, the values are treated as bit flags and are ORed together to form a 32-bit wide bit field for each alternative specified in the Operations entry. The only current use for these flags is for specifying the alternatives that represent the speculative form of the instruction. To specify different flags for different alternatives, the HMDES writer must create a separate Operations entry for each variation of alternative flags. If no alternative flags are specified, the <> may be omitted.

The *operation_class* is the name of an Operation_Class entry that contains a set of alternatives to be associated with the opcode name, assembly name and assembly flags. Alternatively, the *alternatives* can be specified directly in the operation entry. The method that is more convenient may be used.

3. TRANSLATING HMDES TO LMDES

This chapter describes the use of the HMDES-to-LMDES translation program *Lmdes_build*, which checks the syntax of the HMDES file, provides detailed error messages when necessary, and produces an LMDES file that can be quickly loaded into memory by the compiler. The details of how the HMDES information is represented in LMDES is described in Chapter 4. The command line syntax for *Lmdes_build* is shown below.

```
Lmdes_build HMDES_file_name LMDES_file_name [-Dname=replacement_text]
```

where *HMDES_file_name* is the name of the file containing the HMDES description of the processor, *LMDES_file_name* is the name of the file to write the LMDES into, and the option argument *-Dname=replacement_text* overrides any text replacement directives in the HMDES. More than one *-Dname=replace_text* argument may be specified.

Text replacement directives may also be placed in the user's environment using the unix C shell command

```
setenv name=replacement_text
```

where this directive will override the text replacement directives in the HMDES file. This feature is most commonly used to specify the directory where the compiler's header files are stored. This allows HMDES files to be shared between sites with different installations of the same compiler. These environmental directives may be overridden by the command line directives.

This translation program checks for syntax errors in the HMDES grammar and, when syntax errors occur, indicates how the syntax deviated from the correct syntax. This program also checks whether all the identifiers used in the HMDES have been previously defined, which helps detect typos in identifiers. Lastly, during the step where C header files are used to annotate entries in the HMDES (see Section 2.4), if a critical annotation cannot be found an error message is emitted. This helps detect operations that are undefined in the compiler as well as typos in the operation's names. This extensive checking ensures that the LMDES file produced is free of syntax errors and detectable typos, allowing the compiler to efficiently load the LMDES file with minimal error checking.

4. LMDES FILE FORMAT

4.1 Introduction

This section describes the details of what is contained in the low-level machine description (LMDES) file. The information in this section is not necessary for using the high-level machine description language and the compiler interface. However, it may aid those wishing to modify the LMDES language; this section is aimed at this audience.

An LMDES file is created directly from an HMDES file and its sole purpose is to allow quick building of the internal data structures representing the machine description. To accomplish this, the LMDES file represents the MDES information in a format as close as possible to the final internal representation. An LMDES file is not meant to be user modifiable but some extra information is included in the LMDES file to make it somewhat user readable for debugging purposes.

The LMDES file format was designed to allow the standard C language function "fscanf()" to parse the file. The "fscanf()" function is easy to use if the input file's format is predictable. To make the LMDES file format predictable, a header is included in each section and when required, in each entry (more details below), which contains all the format information necessary to read the file with fscanf(). In addition, the headers also contain enough information to allocate the data structures for that section's internal representation. This allows each section's data structures to be allocated before the section is read and for the section's data to be read directly (if possible) into these data structures.

The rest of this chapter first discusses how three design and implementation issues were solved and then goes on to describe the LMDES file format in detail. To facilitate describing the LMDES file format, it is broken up into seven logical components: a file header and six sections. To aid in the discussion of each of these components, examples are pulled from the LMDES file *form.lmdes*, which is included in Appendix E. This LMDES file was produced from the example HDMES file *form.hmdes* that was used in Chapter 2 (also in Appendix A).

4.2 Total String Size

One limitation of using fscanf() that must be dealt with is that reading a string field into a buffer that is too small will cause undetected memory corruption. A typical solution is to read the string into a very large buffer, determine the string's length, allocate an appropriately sized buffer for it, and copy it to this buffer. This is an acceptable solution but it was not the one taken by the LMDES reader (described in Chapter 5). The

LMDES reader made use of the *total string size* statistic that is present in each section's header. This statistic allows the LMDES reader to allocate a block of memory for each section's string data before the section's data are read in. The LMDES reader then reads the section's strings directly into this block of memory with each string placed just after the previous string's terminator. By using this total string size statistic, the LMDES reader never has to be concerned about buffer overflow, and the reader's performance is enhanced slightly because a string copy is not required.

4.3 Mask Width

The internal representation of MDES uses bit fields in several places for efficiency reasons. Several of these bit fields cannot be reasonably restricted to 32 bits, so an array of 32 bit words is needed. The dimension of this array is referred to as the *mask width* (because these bit fields are often used in masking operations). A bit field is logically considered a continuous block of bits and a bit may be addressed by an ID between zero and n-1 (where n is the maximum number of bits allowed to be addressed). In the array representation of this bit field, the 0th word of the array contains bits 0 through 31, the 1st word contains 32-63, and so on. The lowest numbered bit in each word is the least significant bit.

The LMDES file representation of a bit field is the contents of each word in the array printed in hex and separated by a space. The contents of the leftmost word in the LMDES representation corresponds to the 0th word in the array.

4.4 Internal IDs

After the LMDES file is read in, the MDES internal data structures use pointers to represent the relationship between the various sections' entries. Since the value of a pointer to a section's entry cannot be put into the LMDES file, the pointer's value is converted into the *internal ID* of the desired section's entry. Every entry has an internal ID, which is set to 0 for the first entry in each section and is incremented by 1 for each additional entry in that section. By design, the internal ID is also the entry's index in each section's array of entries. Since internal IDs are only unique within a section, the section referred to must always be implied by context. To make the LMDES file more readable by humans, these internal IDs are printed out as the first field of each entry.

4.5 LMDES File Header

The file header specifies the number of predicate, destination, source and synchronization operands that the processor's instruction set supports. In addition, the type of processor (superscalar or VLIW) is specified. The file starts with an identifier string to prevent parsing of non-LMDES files. An example of an LMDES file header is shown in Figure 4.1. The rest of this section contains a line by line description of the LMDES file header, referencing lines from the example.

The first line of Figure 4.1 is an LMDES file identifier. It prevents parsing of non-LMDES files and LMDES files using out-of-date formats. The number indicates the format version and is incremented with every LMDES format change. If an unexpected

```
Lmdes Version 2

proc_model:  0
sizes:  0  2  4  3  3  0  1
```

Figure 4.1: Example header section from the form.lmdes file

version is encountered, the LMDES reader routine exits and advises the user that either the LMDES file has to be regenerated from the HMDES file or that the program has to be linked to the new version of the LMDES reader. This version number differs from the "Version" found in the HMDES file in that it indicates the format of the LMDES file, not the HMDES file.

The third line (counting the blank line) of Figure 4.1 specifies a superscalar (0) or VLIW (1) processor model. The last line of Figure 4.1 specifies the number of each type of operand and information about scheduling slots. The numbers, from left to right, specify:

1. the number of predicate operands (predicates),
2. the number of destination operands (dest_operands),
3. the number of source operands (source_operands),
4. the number of incoming synchronization operands (src_syncs),
5. the number of outgoing synchronization operands (dest_syncs),
6. the maximum scheduling slot ID, and
7. the number of scheduling slots.

The names in the parenthesis indicate the names used in the HMDES Define section (Section 2.7) to set these values.

4.6 LMDES Section Delimiters

There are six sections in an LMDES file. They all begin with

*section_name*_begin *parameters*

and end with

*section_name*_end

where *section_name* is the section name and *parameters* is one or more integers that specify information about the section. All the six sections described in this chapter must be present in the LMDES file and they must appear in the order described in this chapter. This ordering allows all references to other sections to be resolved in one pass of the LMDES file. The description of the six sections follow.

4.7 IO_Sets

The information in the IO_Sets section is used by the MDES compiler interface to specify what operands a specific instruction has (using the external IDs) and by the MDES data structures to specify what operands each scheduling alternative may have (using the internal IDs which are described in Section 4.4). The external IDs were specified in the HMDES file using C header files as described in Section 2.4. Each ID (external or internal) maps to a set of one or more operand types, which is represented with a bit field. Each bit in this bit field corresponds to an operand type that was defined in the HMDES Register_files section (see Section 4.3 for more details on bit fields).

```
IO_Sets_begin  1     4     7     0     4     32
  0   0 NULL        00000001
  1   1 GPR         00000002
  2   2 Lit8        00000004
  3   3 Lit16       00000008
  4   4 REG         00000002
  5  -1 ANY         0000000f
  6  -1 LIT         0000000c
IO_Sets_end
```

Figure 4.2: Example IO_Sets specification section from the form.lmdes file

The information in the IO_Sets section corresponds to a subset of the information in the HMDES Register_Files section and all the information in the HMDES IO_Sets section. Some of the information in the Register_Files section (namely, register file width and capacity information) is not present in the LMDES file because it contained too little information to do MDES based register allocation.

An example IO_Sets section is shown in Figure 4.2. Using lines from this example, the IO_Sets section header and the IO_Sets section entries are described.

The first line of Figure 4.2 is the IO_Sets section header. The numbers in the header, from left to right, specify:

1. the IO set mask width (see Section 4.3 for details),
2. the number of bits used in IO set mask,
3. the total number of IO sets,
4. the external ID of the "NULL" IO set,
5. the maximum external IO set ID, and
6. the section's total string size (see Section 4.2 for details).

```
IO_Items_begin    9     81
  0 IOI_Store     0  0   4  5  4  0
  1 IOI_Load      4  0   4  5  0  0
  2 IOI_Std       4  0   4  4  0  0
  3 IOI_Stdi      4  0   4  2  0  0
  4 IOI_ubri      0  0   4  0  0  0
  5 IOI_cbr       0  0   4  2  6  0
  6 IOI_cbri      0  0   4  2  4  0
  7 IOI_call      4  0   6  0  0  0
  8 IOI_calli     4  0   4  0  0  0
IO_Items_end
```

Figure 4.3: Example IO_Items specification section from the form.lmdes file

After the header comes the IO_Sets entries. Each entry consists of one line with four fields. The second line of Figure 4.2 is an example of an IO_Sets entry. The entry's fields, from left to right, specify:

1. the internal ID (see Section 4.4 for details),
2. the external ID (-1 indicates none specified),
3. the name of the IO set, and
4. the bit field representing a set of operand types.

4.8 IO_Items

The entries in the IO_Items section are operation operand templates, which specify what operands may be used in each scheduling alternative. The entries use IO_Sets entry internal IDs, one for each possible operand in an operation, to indicate what operand type may be used. An example IO_Items section is shown in Figure 4.3. Using lines from this example, the IO_Items section header and the IO_Items section entries are described.

The first line of Figure 4.3 is the IO_Items section header. The numbers in the header, from left to right, specify:

1. the number of IO items, and
2. the total string size (see Section 4.2 for details).

After the header comes the IO_Items entries, each consisting of one line with a variable number of fields (depending on the number of operands an operation can have). The second line of Figure 4.3 is an example IO_Items entry. The first two fields, from left to right, specify:

1. the internal ID and
2. the IO item's name.

The remaining fields are IO_Sets entry internal IDs, one for each possible operand that an operation can have. These fields, from left to right, specify the internal IDs for each:

1. predicate operand (none in the example),
2. destination operand (two in the example: $dest_0$, $dest_1$), and
3. source operand (four in the example: src_0, src_1, src_2, src_3).

4.9 Resources

The Resources section lists all the resources used to model the scheduling constraints for the processor. Each Resources entry consists of an internal ID and a name. These internal IDs are used as bit indices in the ResList section (see Section 4.10 for details). An example Resources section is shown in Figure 4.4. Using lines from this example, the

```
Resources_begin    2     17
  0 slot[0]
  1 ialu0[0]
Resources_end
```

Figure 4.4: Example Resources section from the form.lmdes file

Resources section header and the Resources section entries are described. The first line of Figure 4.4 is the Resources section header. The numbers in the header, from left to right, specify:

1. the number of resources, and
2. the total string size (see Section 4.2 for details).

After the header come the Resources entries, each consisting of one line with two fields. The second line of Figure 4.4 is an example Resources entry. The fields, from left to right, specify:

1. the internal ID (which is used as a bit index) and
2. the resource's name (which includes the square brackets and number).

4.10 ResList

ResList entries are used by the operations section (Section 4.12) to specify what resources an operation requires as it executes. Each ResList entry consists of a list of resource usage entries, where each resource usage entry specifies a resource usage start and stop time (resources may be used for multiple cycles) and a set of resource usage options, where any one of the resource usage options may be selected (see Section 2.12).

The ability to specify more than one resource option is useful when, for example, there are several scheduling slots that an operation may be scheduled into or there are several functional units that support an operation. Each resource usage option may specify more than one resource (a bit field is used, using the Resources entry internal ID as a bit ID) but the HMDES-to-LMDES translator does not currently take advantage of this.

When scheduling an operation, there are often constraints on which scheduling slot that operation may be placed in (usually due to control, anti, and output dependencies). To allow the resource manager to determine which resource usage entry corresponds to scheduling slot options, the ResList entries are organized so that all the scheduling slot options are in the first resource usage entry of each ResList entry. To facilitate determination of what slot option corresponds to which slot (since they are bit fields), an array of slot IDs (one for each slot option) is included in each ResList entry.

An example ResList section is shown in Figure 4.5. Using lines from this example, the description of the components of the ResList section is given. The first line of Figure 4.5 is the ResList section header. The ResList header from the example is shown above. The numbers in the header, from left to right, specify:

1. the number of ResList entries,
2. the total number of resource usage entries in all the ResList entriess,
3. the total number of scheduling slot options in all the ResList entries,
4. the total number of resource usage options in all the ResList entries,
5. the resource mask width (see Section 4.3 for more details), and
6. the total string size (see Section 4.2 for more details).

```
ResList_begin    2     3     2     3  1     15
 0 RL_IBr          1     1     1
   0
   0  0  1
  00000001   00000000
 1 RL_IAlu         2     1     2
   0
   0  0  1
  00000001   00000000
   0  0  1
  00000002   00000000
ResList_end
```

Figure 4.5: Example ResList section from the form.lmdes file

The first line of each ResList entry is the entry header. The second line of Figure 4.5 is an example of a ResList entry header. The entry header's fields, from left to right, specify:

1. the ResList's internal ID,
2. the ResList's name,
3. the number of resource usage entries,
4. the number of slot options, and
5. the number of resource manager entries required.

The second line of each ResList entry gives the scheduling slots that an operation with this ResList can use (see introduction of this section for more details). The third line of Figure 4.5 is an example of a list of scheduling slots.

The remaining lines in the ResList entry are resource usage entries. Each resource usage entry has a line for its header and then a line for each resource usage option. The fourth line of Figure 4.5 is an example resource usage entry header. The header's fields, from left to right, specify:

1. the resource usage start time,
2. the resource usage end time, and
3. the number of resource usage options.

After the resource usage header there is one line for each resource usage option. The fifth line of Figure 4.5 is the only resource option for the first resource usage entry in the first ResList entry. The resource option fields, from left to right, specify:

1. the unconditional resource usage mask (which is a bit field; see Section 4.3 for more details),
2. the predicated resource usage mask.

The predicated resource usage mask will be all zeros until the HMDES language is extended to handle predicated resource usage.

4.11 Latencies

The Latencies section entries are used by the Operations section (see Section 4.12) to specify when all operation's operands are read (predicate, source, or incoming sync operands) or written (destination or outgoing sync operands). This information can then be used to calculate dependence distances for flow, anti, output, and control dependencies. The Latencies section entries also indicate when exceptions are handled. An example Latencies section is shown in Figure 4.6. Using lines from this example, the Latencies section header and the Latencies section entries are described.

```
Latencies_begin    4      22
  0 Lat1          1    1  0   0  0  0  0   0  0  0   0  0  0
  1 Lat1m         2    1  0   0  0  0  0   1  0  0   0  0  0
  2 Lat2          2    1  0   0  0  0  0   0  0  0   0  1  0
  3 Lat2m         2    2  0   0  0  0  0   0  0  0   1  0  0
Latencies_end
```

Figure 4.6: Example Latencies section from the form.lmdes file

The first line in Figure 4.6 is the Latencies section header. The numbers in the header, from left to right, specify:

1. the number of Latencies section entries and
2. the total string size (see Section 4.2 for details).

After the header comes the Latencies section entries. Each entry consists of one line and has a variable number of fields (depending on the number of operands and syncs that an operation can have). The second line of Figure 4.6 is an example Latencies section entry. The first three fields, from left to right, specify:

1. the entry's internal ID,
2. the entry's name, and
3. the exception latency.

The remaining fields, from left to right, specify the latency for each:

1. predicate operand (none in the example),
2. destination operand (two in the example: $dest_0$, $dest_1$),
3. source operand (four in the example: src_0, src_1, src_2, src_3),
4. incoming sync operand (three in the example: $sync\_in_0$, $sync\_in_1$, $sync\_in_2$), and
5. outgoing sync operand (three in the example: $sync\_out_0$, $sync\_out_1$, $sync\_out_2$).

4.12 Operations

Each Operations section entry consists of an opcode, the opcode's name, a set of opcode flags (used to indicate branches, memory operations, etc.), and a set of alternatives. The set of alternatives specifies the different ways that an instruction with this opcode can use the resources of the processor. Not all alternatives for an opcode are the same. Each alternative can specify what type of operands it expects (using an IO set ID), the time at which it reads and writes each operand (using a latency entry ID), what resources it requires (using a ResList entry ID), what special properties (such as being nontrapping) that it may have (using the alternative's flags), and the assembly language name for it. Using this information, the compiler can model how instructions use the processor's resources.

An example Operations section is shown in Figure 4.7. Using lines from this example, the contents of the Operations section are described. The first line of Figure 4.7 is an Operations section header. The numbers in the header, from left to right, specify:

1. the number of Operations section entries,
2. the total number of alternatives,
3. the maximum opcode defined, and
4. the total string size (see Section 4.2 for more details).

After the section header comes the operation entries, each containing a list of alternative entries. The first line of each operation entry is an operation header. The second line of Figure 4.7 is an example of an operation header. The fields in the operation header, from left to right, specify:

1. The operation's opcode,
2. the opcode's external name,
3. the number of alternatives for this opcode,
4. the opcode's flags in hex.

After the operation header is a list of alternative entries, one per line. The third line of Figure 4.7 is the only alternative entry for the first operation entry. The fields, from left to right, specify:

1. the assembly language name for this alternative,
2. the alternative's flags in hex,
3. the internal ID of an IO item,
4. the internal ID of a ResList entry, and
5. the internal ID of a latency entry.

```
Operations_begin     8    11   500    125
   1 Lop_BEQ       1 00000041
  jmpt        00000000    5   0   2
   2 Lop_BNE       1 00000041
  jmpf        00000000    5   0   2
   3 Lop_RTS       1 00000042
  jmpi        00000000    4   0   2
   4 Lop_JSR       2 00000044
  call        00000000    7   0   2
  calli       00000000    8   0   2
   5 Lop_ADD       2 00000000
  add         00000000    2   1   0
  add         00000000    3   1   0
 500 LAMDop_SUBR   2 00000000
  subr        00000000    2   1   0
  subr        00000000    3   1   0
   6 Lop_LD_I      1 00000028
  load        00000000    1   1   3
   7 Lop_ST_I      1 00000030
  store       00000000    0   1   1
Operations_end
```

Figure 4.7: Example Operations section from the form.lmdes file

5. MDES INTERNAL REPRESENTATION

5.1 Introduction

This section describes the internal representation of the MDES. Ten data structures are used to hold the information contained in the LMDES file described in Chapter 4. These structures are used by the MDES interface functions described in Chapter 6 and by the resource usage (RU) manager (which is not described in this thesis). In addition, the Mdes_Info structures, which are built using the functions described in Section 6.4, are also briefly described.

5.2 The Mdes Data Structure

The Mdes structure is the "root" data structure for the MDES internal representation and is shown in Figure 5.1. In IMPACT, after the MDES is loaded, the pointer "lmdes" points at an initialized structure of this type.

```
typedef struct mdes_st
{
    char           *file_name;
    int             processor_model; /* MDES_SUPERSCALAR, MDES_VLIW */
    int             number[5];       /* Number of each operand/sync type */
    int             offset[5];
/* Offset for each operand/sync type */
    char            *name[5];      /* Name of each operand/sync type */
    int             operand_count;
    int             latency_count;
    int             num_slots;
    int             max_slot;
    int             IOmask_width;
    int             num_reg_files;
    int             max_IO_set_id;
    int             null_external_id;
    Mdes_IO_Set     **IO_set_table;  /* Indexed by external id */
    int             num_IO_sets;
    Mdes_IO_Set     *IO_set;
    int             num_IO_items;
    Mdes_IO_Item    *IO_item;
    int             num_resources;
    Mdes_Resource   *resource;
    int             num_reslists;
    int             Rmask_width;
    Mdes_ResList    *reslist;
    int             num_latencies;
    Mdes_Latency    *latency;
    int             num_operations;
    Mdes_Operation  *operation;
    int             max_opcode;
    Mdes_Operation  **op_table;         /* Indexed by opcode */
    Mdes_IO_Set     **operand_type_buf; /* Used by Build_Mdes_Info */
} Mdes;
```

Figure 5.1: MDES internal representation: Mdes structure

The Mdes structure's main purpose is to provide size information for all the arrays used to hold the MDES information and to provide pointers to those arrays. This information was put into the Mdes structure instead of storing it in global variables to prevent naming conflicts and to allow for the possibility of loading more than one MDES at a time.

The Mdes structure also contains the relevant information for the HMDES Define section (see Section 2.7), some information useful when printing error messages, and some summary fields for values that are used as array dimensions.

The fields that contain the HMDES Define section information are *processor_model* and *number[5]*. The *processor_model* contains either MDES_SUPERSCALAR or MDES_VLIW and is accessed with the interface function *mdes_processor_model()* (see Section 6.8). The *number[5]* array contains the number of each type of operand (pred, dest, src, sync_in, and sync_out) and is indexed by MDES_PRED, MDES_DEST, MDES_SRC, MDES_SYNC_IN, and MDES_SYNC_OUT. The array is accessed by the interface function *mdes_num_operands()* (see Section 6.4).

The fields used for error messages are *file_name* and *name[5]*. The LMDES file name that was read to create the MDES is stored in *file_name*. The *name[5]* array contains the string name of each type of operand (accessed using the same constants as number[5] above). These fields and the fields that store each MDES section entry's name are used extensively to make error messages readable and useful.

The fields that contain summary information used as array dimensions are *operand_count*, *latency_count*, *num_slots*, and the obsolete *max_slot*. The field *operand_count* contains the sum of the number of pred, dest, and src operands. The field *latency_count* contains the sum of the number of pred, dest, src, sync_in, and sync_out operands. These two fields are used internally by the Mdes_IO_Item and Mdes_Latency structures respectively to determine array sizes. They are also used when calling the interface functions *build_mdes_info()* (Section 6.4) and *mdes_calc_min_ready_time()* (Section 6.11) respectively to determine the size of an array argument.

The field *num_slots* specifies the number of scheduling slots the MDES supports and *max_slot* specifies the max scheduling slot ID. The early versions of MDES allowed non-contiguous numbering of slots (i.e., 1, 3, 5) and required both numbers. The MDES described in this document requires slot IDs to be contiguous numbers starting with 0 (i.e., 0, 1, 2, 3). This makes *max_slot* unneeded since it is always *num_slots* - 1. The *num_slots* field is used by the resource usage manager to allocate the space for each cycle's instructions. It is accessed with the interface function *mdes_total_slots*() (see Section 6.8).

The fields *IO_set*, *IO_item*, *resource*, *reslist*, *latency*, and *operation* are pointers to arrays of structures. Each element in these arrays represents section entries in the LMDES and are indexed by the entry's internal ids (see Section 4.4). The length of each array is specified in the fields *num_IO_sets*, *num_IO_items*, *num_resources*, *num_reslists*, *num_latencies*, and *num_operations*. The remaining fields specify more information about IO_sets and operations.

The field *num_reg_files* specifies the number of entries in the HMDES Register_Files section. This number also corresponds to the number of distinct operand types specified in the MDES (see Section 2.8). The IO_set structure contains a set of these operand types which is represented as a bit field and the field *IOmask_width* specifies the length of this bit field, in terms of 32 bit words (see Section 4.3 for more details). Similarly, the *Rmask_width* field specifies the width of the resource bit field.

In addition to accessing the IO_Set and Operation structures with internal ids, these structures also can be accessed using external (user defined) ids (see Section 2.4). The arrays *IO_set_table* and *op_table* are indexed by external ids instead of internal ids and contain pointers to the entries stored in IO_Set and Operation structures. These arrays may have NULL pointers in them if external ids are not contiguous and the lengths of the arrays are *max_IO_set_id* + 1 and *max_opcode* + 1. Negative external ids are not allowed. The field *null_external_id* contains the external id for the NULL register file and is used by the initialization section of the *build_mdes_info()* routine.

5.3 The Mdes_IO_Set Data Structure

The Mdes_IO_Set structure (Figure 5.2) holds the IO_Sets entry information. An array of these structures is allocated, one for each LMDES IO_Sets entry. This array is accessed using the *IO_set* pointer and the *IO_set_table* array of pointers in the Mdes structure. As described in Section 4.7, each LMDES IO Set entry associates an external_id given in the *external_id* field, with a set of operand types given in the bit field pointed

```
typedef struct mdes_IO_set_st
{
    int     id;
    int     external_id;
    char    *name;
    int     *mask;
} Mdes_IO_Set;
```

Figure 5.2: MDES internal representation: Mdes_IO_Set structure

```
typedef struct mdes_IO_item_st
{
    int         id;
    char        *name;
    Mdes_IO_Set **operand_type;
} Mdes_IO_Item;
```

Figure 5.3: MDES internal representation: Mdes_IO_Item structure

to by *mask*. This bit field is actually an integer array whose length is given by the Mdes structure's field *IOmask_width*. The *name* field is used for error messages (mainly in *build_mdes_info()*) and the *id* field contains the internal id (or array index) for this entry which simplifies debugging.

5.4 The Mdes_IO_Item Data Structure

The Mdes_IO_Item structure shown in Figure 5.3 holds the IO_Item entry information described in Section 4.8. An array of these structures is allocated, one for each LMDES IO_Item entry. This array is accessed using the IO_Item pointer in the Mdes structure. To summarize, these entries are operand templates for operations where the template

```
typedef struct mdes_resource_st
{
    int      id;
    char     *name;
} Mdes_Resource;
```

Figure 5.4: MDES internal representation: Mdes_Resource structure

is stored in *operand_type*. This template is an array of Mdes_IO_Set pointers of length (Mdes's) operand_count. Operand indexes, described in Section 6.10, are used to index this array. The internal id and name of the IO_Item entry are preserved in the fields *id* and *name* for debugging purposes.

5.5 The Mdes_Resource Data Structure

The Mdes_Resource structure (Figure 5.4) contains a resource's name and internal id (see Section 4.9). Although not directly needed by any scheduling functions, it is kept to allow resource names to be used in debugging/informational messages.

5.6 The Mdes_ResList Data Structure

The Mdes_ResList structure (Figure 5.5) holds the LMDES ResList entry information described in Section 4.10, with one structure for each ResList entry. This array of structures is accessed using the Mdes structure's *reslist* pointer. The resource usage entries that made up part of each ResList entry are represented by the Mdes_Rused structure (Figure 5.6). The resource usage options that made up part of each resource usage entry are represented by the Mdes_Rmask structure (Figure 5.7).

```
typedef struct mdes_reslist_st
{
    int         id;
    char        *name;
    int         num_used;
    Mdes_Rused  *used;
    int         num_slot_options; /* Number of slot options for this ResList*/
    int         *slot_options;    /* slot numbers, in mdes order (not sorted)*/
    int         num_RU_entries_required;
} Mdes_ResList;
```

Figure 5.5: MDES internal representation: Mdes_ResList structure

```
typedef struct mdes_Rused
{
    int         start_usage;
    int         end_usage;
    int         num_options;
    Mdes_Rmask  *option;
} Mdes_Rused;
```

Figure 5.6: MDES internal representation: Mdes_Rused structure

```
typedef struct mdes_rmask_st
{
    int     *uncond;
    int     *pred;
} Mdes_Rmask;
```

Figure 5.7: MDES internal representation: Mdes_Rmask structure

The root data structure, Mdes_ResList, contains a pointer to an array (*used*) of Mdes_Rused structures, with length *num_used*, that describes the resources used by the operation. For each resource used (such as an ALU), the structure Mdes_Rused specifies the *start_usage* and *end_usage* times and a pointer to an array (*option*) of Mdes_Rmask structures, with length *num_options*, that describes the scheduling options for that resource usage entry. The structure Mdes_Rmask represents each of these scheduling options where *uncond* is a pointer to a bit field representing a set of resources used unconditionally and *pred* is a pointer to a bit field representing a set of resources used if operation is predicated and the predicate is true. Currently, *uncond* will only specify one resource (such as ALU1) and *pred* will specify zero resources. IMPACT's resource usage manager (not described in this document) was written to use both *uncond* and *pred* but the current HMDES specification does not allow predicated resources to be specified.

To facilitate determining the valid scheduling slots, the valid scheduling slots are listed in an array of length *num_slot_options* accessed using *slot_options.* The use of this array is described in detail in Section 6.10. Also to make allocating memory more efficient, the number of resource usage entries required for this resource list is indicated by *num_RU_entries_used.* The *id* and *name* field are kept for use during debugging.

5.7 The Mdes_Latency Data Structure

The Mdes_Latency structure in Figure 5.8 is the internal representation of an LMDES Latency entry described in Section 4.11. The *operand_latency* field points to an array

```
typedef struct mdes_latency_st
{
    int         id;
    char        *name;
    int         exception;
    int         *operand_latency;
} Mdes_Latency;
```

Figure 5.8: MDES internal representation: Mdes_Latency structure

```
typedef struct mdes_operation_st
{
    int         id;
    int         opcode;
    char        *external_name;
    int         num_alts;
    Mdes_Alt    *alt;
    int         op_flags;
    int         heuristic_alt; /* Used for heuristics */
} Mdes_Operation;
```

Figure 5.9: MDES internal representation: Mdes_Operation structure

of length *latency_count* (from the Mdes structure), containing the use time for all of an operation's operands. Operand indexes, described in Section 6.10, are used to index this array. The *exception* field stores the time an exception is reported if it occurs. The *id* and *name* fields are preserved for debugging purposes.

5.8 The Mdes_Operation Data Structure

The Mdes_Operation structure, shown in Figure 5.9, associates an *opcode* with an array of scheduling alternatives (*alt*) of length *num_alts* and to a set of flags (*op_flags*)

```
typedef struct mdes_alt_st
{
    int                           id;
    char                          *asm_name;
    int                           alt_flags;
    struct mdes_operation_st      *operation;
    Mdes_IO_Item                  *IO_item;
    Mdes_ResList                  *reslist;
    Mdes_Latency                  *latency;
} Mdes_Alt;
```

Figure 5.10: MDES internal representation: Mdes_Alt structure

that are used to characterize the operation. The flags are accessed using the *op_flag_set()* function described in Section 6.7. To aid in debugging, the *opcode* field is the numerical opcode associated with this operation by the *external_name* in the HMDES file and *id* is this operation's internal id.

For each scheduling alternative, the structure Mdes_Alt (Figure 5.10) specifies the operand template (*IO_item*), the resources used (*reslist*), the operand use times (*latency*), the assembly name to emit for this alternative (*asm_name*), and the alternative's flag settings (*alt_flags*). The alt_flags are accessed using the routines alt_flags_set() and any_alt_flags_set() described in Section 6.7. The *IO_item*, *reslist*, and *latency* fields are just pointers to the structures indicated in the figure and described in earlier sections. The *id* field is the entry's internal id and the *operation* field is a pointer back to the operation the alternative is associated with.

```
typedef struct mdes_info_st
{
    int                  opcode;
    int                  num_compatible_alts;
    Mdes_Compatible_Alt *compatible_alts; /* Linked list */
} Mdes_Info;
```

Figure 5.11: MDES internal representation: Mdes_Info structure

```
typedef struct mdes_compatible_alt_st
{
    Mdes_Alt                            *alt;
    struct mdes_compatible_alt_st   *next;
} Mdes_Compatible_Alt;
```

Figure 5.12: MDES internal representation: Mdes_Compatible_Alt structure

5.9 The Mdes_Info Data Structure

The Mdes_Info structure shown in Figure 5.11 is returned by the function build_mdes_info() described in Section 6.4. It serves as a header to a linked list of valid scheduling alternatives *compatible_alts* of length *num_compatible_alts* for an operation. The *opcode* field is used for error messages in functions that take an Mdes_Info structure instead of an opcode.

The linked list is made of up of the Mdes_Compatible_Alt structures shown in Figure 5.12. The *alt* field points to an alternative structure in the MDES data structure. The alternative structure is not a copy, so it may not be modified in any way. The *next* field is used to form the linked list.

6. MDES INTERFACE FUNCTIONS

6.1 Introduction

This chapter explains how to use all of the machine description (MDES) compiler interface functions; most of them were designed for use by code schedulers and dependence graph builders. There are relatively few functions because the MDES resource usage information may only be accessed by the resource usage (RU) manager, and the RU manager (which is not described in this document) uses the MDES data structures (which are described in Chapter 5) directly. The RU manager's interface consists mainly of two functions, one to determine if an operation can be scheduled in a certain cycle, and one to schedule it in a cycle.

The MDES information available through these interface functions includes operation classification (load, store, jump, etc.), operation latencies, processor model, and number of scheduling slots. Other functions create and free MDES data structures needed by

```
void L_init_lmdes (char *mdes_file_name);
int lmdes_initialized (void);

/* Use IMPACT interfaces instead of the generic, if possible */
Mdes_Info *build_mdes_info (int opcode,   /* Generic interface */
                            int *io_list);
void L_build_oper_mdes_info (L_Oper *op); /* IMPACT interface */
void L_build_cb_mdes_info (L_Cb *cb);     /* IMPACT interface */

void free_mdes_info (Mdes_Info *info);    /* Generic interface */
L_free_oper_mdes_info (L_Oper *op);       /* IMPACT interface */
L_free_cb_mdes_info (L_Cb *cb);           /* IMPACT interface */

int op_flag_set (int opcode, int mask);
int alt_flag_set (int opcode, int alt_no, int mask);
int any_alt_flag_set (Mdes_Info *mdes_info, int mask);

int mdes_defined_opcode (int opcode);
int mdes_heuristic_alt_id (int opcode);

int mdes_max_opcode (void);
int mdes_operand_count (void);
int mdes_latency_count (void);
int mdes_num_operands (int operand_type);
int mdes_processor_model(void);
int mdes_total_slots(void);

int operand_index (int operand_type, int operand_number);
int max_operand_time (Mdes_Info *mdes_info, int operand_index);
int min_operand_time (Mdes_Info *mdes_info, int operand_index);
int mdes_operand_latency (int opcode, int alt_no, int operand_index);


int mdes_calc_min_ready_time (Mdes_Info *mdes_info, int *ready_times);
int mdes_max_completion_time (int opcode, int alt_no);
```

Figure 6.1: MDES interface function prototypes

the RU manager. The function prototypes for all the interface functions are shown in Figure 6.1.

6.2 Quick Start

The following is a brief summary of the functions used to initialize the MDES so that the MDES interface functions may be called. This summary assumes the use of the IMPACT compiler and references IMPACT specific data structures which are not described in this document.

Use *L_init_lmdes()* to initialize the MDES internal data structures using the LMDES file name passed as an argument. For efficiency reasons, this function should be called only once. After this function is called, the MDES interface functions that do not have an mdes_info argument may be used. These functions include returning the machine model, returning the number of each type of operand, and determining opcode flags.

To use the remaining functions, which are mainly necessary for scheduling, the Mdes_Info structure must be built for each op. In the IMPACT compiler, this can be done using either:

*void L_build_oper_mdes_info (L_Oper *op);* or

*void L_build_cb_mdes_info (L_Cb *cb);* .

The function *L_build_oper_mdes_info* is an IMPACT interface function that uses IMPACT's Mspec to build an io_list (an encoding of the operands that the MDES can understand) for the op. It then uses the io_list to call the *build_mdes_info()* function (described in Section 6.4), which scans the MDES and creates a linked list of all the valid alternatives for the operation based on its opcode and its operands. This information is stored in the Mdes_Info structure (Section 5.9) and is placed in the *mdes_info* field of the op. The *L_build_cb_mdes_info()* function calls *L_build_oper_mdes_info()* for each op in the cb.

6.3 Loading an LMDES

The *L_init_lmdes()* function is used to load an LMDES file into memory. This must be done before any MDES interface functions are called. For efficiency reasons, *L_init_lmdes()* should be called only once unless it is necessary to load a different LMDES file and the function *lmdes_initialized()* can be used to determine if an LMDES file has been loaded (returns 1 if loaded, 0 otherwise). In IMPACT, the LMDES file name specified in the parameter file is stored in the global variable L_lmdes_file_name. A typical IMPACT code sequence is shown below.

```
if (!lmdes_initialized())
    L_init_lmdes (L_lmdes_file_name);
```

6.4 Building Mdes_Info Structures

The Mdes_Info structure (Section 5.9) contains a list of valid scheduling alternatives for a particular operation. Several MDES interface functions use this structure to narrow the scope of the information they provide to only these alternatives and thus provide more accurate information. Therefore, the Mdes_Info structure must be built before these functions can be used.

A *valid scheduling alternative* for an operation is an alternative that is compatible with the operation's operands. The MDES function shown below will return an Mdes_Info structure with a list of valid scheduling alternatives.

*Mdes_Info *build_mdes_info (int opcode, int *io_list);*

The *opcode* is the operation's opcode, and the io_list is an integer array that specifies what each source, destination and predicate operand is. The expected size of the array is determined by

int mdes_operand_count();

and the array index of each operand is determined with the function *operand_index()* (see Section 6.10 for a description of this function). The number of each type of operand can be determined with the following function.

int mdes_num_operands (int operand_type);

The type of each operand is specified in the io_list using an IO set specifier of the form

MDES_OPERAND_*name*

where *name* is the name of an HMDES register file or IO set entry. For example, if all instructions have one destination, two sources and no predicate operands, the io_list for the operation "mov r1, 1" might be:

```
io_list[operand_index(MDES_DEST, 0)] = MDES_OPERAND_REG;
io_list[operand_index(MDES_SRC, 0)] = MDES_OPERAND_Lit8;
io_list[operand_index(MDES_SRC, 1)] = MDES_OPERAND_NULL;
```

where MDES_OPERAND_REG, etc. have been #defined to be integers in a C header file and that C header file was used by the HMDES-to-LMDES translator to map the names to the assigned numbers (see Section 2.4 for more details).

6.5 Building Mdes_Info Structures in IMPACT

In IMPACT, building the Mdes_Info structure is done with the two functions shown below.

*void L_build_oper_mdes_info (L_Oper *op);*

*void L_build_cb_mdes_info (L_Cb *cb);*

These functions build an Mdes_Info structure for each op and places the pointers to these Mdes_Info structures in each op's *mdes_info* field. The above functions call the following Mspec function to obtain a function that takes an Lcode operand and returns the appropriate integer ID for the operand:

IFPTR M_mdes_operand_type();

The function that would be returned for the architecture described by *form.hmdes* is shown in Figure 6.2.

6.6 Freeing Mdes_Info Structures

The function that frees the memory allocated for the Mdes_Info structure is shown below.

*void free_mdes_info (Mdes_Info *info);*

```
int M_operand_type_sample (operand)
L_Operand *operand;
{
    /* If NULL operand pointer, then return MDES_OPERAND_NULL */
    if (operand == NULL)
        return MDES_OPERAND_NULL;

    switch (operand->type)
    {
        case L_OPERAND_INT:
            if (operand->value.i<256)
                return MDES_OPERAND_Lit8;
            else
                return MDES_OPERAND_Lit16;

        case L_OPERAND_MACRO:
        case L_OPERAND_REGISTER:
            return MDES_OPERAND_GPR;

        case L_OPERAND_CB:
        case L_OPERAND_LABEL:
        case L_OPERAND_FLOAT:
        case L_OPERAND_DOUBLE:
        case L_OPERAND_STRING:
            return MDES_OPERAND_Lit16;

        default:
            M_assert(0,"M_operand_type_sample : Unknown type");
    }
}
```

Figure 6.2: Sample Mspec function for determining MDES operand type

The IMPACT interface also provides the following two functions.

*void L_free_oper_mdes_info (L_Oper *op);*

*void L_free_cb_mdes_info (L_Cb *cb);*

6.7 Opcode Specific Information

In the HMDES file, flags may be specified for each opcode. These flags in IMPACT are used to specify types of instructions such as loads, stores, branches and function calls. Other information also may be specified, such as whether the instruction could cause an exception. These flags are #defined to be an integer that has one bit set, and they have the form OP_FLAG_*name*. The op flags used by *form.hmdes* in Section 2.15 are shown again in Figure 6.3.

To determine if a flag is set, use the function

int op_flag_set (int opcode, int mask);

where *mask* is one or more flags ORed together. The function returns 1 if any of the flags in the mask are set, 0 otherwise. For example, the following call will return 1 if the op is a memory operation:

```
op_flag_set (op->proc_opc, OP_FLAG_LOAD | OP_FLAG_STORE)
```

Flags may also be specified for individual scheduling alternatives. In IMPACT, these flags are used mainly to specify speculative versions of instructions. The alt flags used by *form.hmdes* in Section 2.15 are shown in Figure 6.4.

The following two functions are used to determine if alt flags are set

```
#define OP_FLAG_CBR    0x00000001
#define OP_FLAG_RTS    0x00000002
#define OP_FLAG_JSR    0x00000004
#define OP_FLAG_LOAD   0x00000008
#define OP_FLAG_STORE  0x00000010
#define OP_FLAG_EXCEPT 0x00000020 /* Can cause exception */
#define OP_FLAG_NI     0x00000040 /* non-interlocking, has delay slot */
```

Figure 6.3: Sample op flags used in form.hmdes

```
#define ALT_FLAG_SPEC 0x00000001  /* speculative form of instr */
```

Figure 6.4: Sample Alt Flags used in form.hmdes

int alt_flags_set (int opcode, int alt_no, int mask);

*int any_alt_flag_set (Mdes_info *mdes_info, int mask);*

The first function works similarly to the op flags function, except a specific alternative number must be specified. This number is usually returned by the resource manager to identify the alternative scheduled. The second function returns 1 if any of the valid scheduling alternatives in Mdes_info have the flag set. In IMPACT, this function is used to determine if there is a speculative version of the instruction that can be scheduled.

6.8 General Model Information

This section describes the functions for obtaining information and parameters about the processor and machine description. To obtain the maximum opcode defined in the MDES call the following function.

int mdes_max_opcode (void);

To determine if an opcode is defined (not all opcodes between 1 and the maximum need to be defined) call the function shown below.

int mdes_defined_opcode (int opcode);

These last two functions are useful for building data structures that are indexed by opcode. The processor model (MDES_SUPERSCALAR or MDES_VLIW) can be determined with the function shown below.

int mdes_processor_model (void);

The total number of scheduling slots (issue width) (from 0 to number - 1) is returned by the function shown below.

int mdes_total_slots (void);

6.9 Scheduling Alternative IDs

For each opcode, one or more scheduling alternatives may be defined in the MDES. The scheduling alternatives may differ in the number and types of operands they expect (see Section 6.4), the resources they use, the alternative flags they have set (see Section 6.7), and in the operand use times (see Section 6.11). For ease of identification, each of these scheduling alternatives is assigned a number from 0 to the number of alternatives - 1. In functions with an alt_no parameter are expecting a scheduling alternative ID.

The alternative used for each operation is determined when it is scheduled, and the alternative's ID should be used if it is available. However, sometimes an alt_no is needed before scheduling and a heuristically chosen alternative must be used. In IMPACT, this occurs in the first pass of the scheduler when a dependence graph is built to help calculate each operation's initial scheduling priority. The heuristic to pick alternatives in this case should be an optimistic one, so the function below chooses the alternative that produces its results the earliest and, in the case of ties, chooses the lowest numbered alternative.

int mdes_heuristic_alt_id(int opcode);

An opcode/alt_no pair specifies a single scheduling alternative for an instruction which in turn specifies the exact operand use times and alternative flags for the operation. The operands and resources used cannot be determined exactly from this information due to the flexibility introduced by IO_Items (Section 2.10) and scheduling options (Section 2.14). This information can only be determined by calls to the resource usage manager (which is not described in this document).

6.10 Operand Indexes

For ease of use, each possible operand position (such as dest[0] or src[2]) is assigned a unique number called its *operand_index*. These numbers range from 0 to the total number of operand positions - 1. In addition to being passed directly to several MDES functions, they are also used as array indexes when information about all of an operation's values are passed to a function in array form. The array index of an operand position is determined with

int operand_index (int operand_type, int operand_number);

where the possible values for *operand_type* are

1. MDES_DEST for destination operands,

2. MDES_SRC for source operands,
3. MDES_PRED for predicate source operands,
4. MDES_SYNC_IN for incoming synchronization operands (dependencies), or
5. MDES_SYNC_OUT for outgoing synchronization operands,

and the *operand_number* starts at 0, and enumerates the possible operands of each type. For example, the operand index for src[0] would be determined with the function call below.

```
operand_index(MDES_SRC, 0)
```

6.11 Operand Use Times

In the Latencies section of the HMDES, the read time of each src, pred and sync_in operands and the write time of each dest and sync_out operands are specified. These times are used in IMPACT to determine dependence (flow, anti, output, control) distances between instructions. For example, to calculate a flow dependence distance, subtract the source's read time (say 0 cycles after initiation) from the destination's write time (say 1 cycle after initiation) to get a flow dependence distance of 1. To obtain these times from the MDES for each operand use the function

int mdes_operand_latency (int opcode, int alt_no, int operand_index);

where *opcode* is the operation's opcode, *alt_no* is a scheduling alternative ID (see Section 6.9), and *operand_index* specifies the operand of interest (see Section 6.10).

To find the maximum destination latency for a scheduling alternative when the MDES supports multiple destinations, use the following function:

int mdes_max_completion_time (int opcode, int alt_no);

To determine the earliest valid scheduling alternative which may be scheduled, given the times all the operands are available, use

*int mdes_calc_min_ready_time (Mdes_Info *mdes_info, int *ready_times);*

where the *mdes_info* is described in Section 6.4 and *ready_times* is an array containing the ready time for each src, dest, and pred operand and is indexed using *operand_index()* (see Section 6.10). The expected length of the *ready_times* array can be determined with the function *mdes_latency_count()*.

The ready times for dest operands indicate the first cycle each operand is available for writing into the destination. The ready times for src and pred operands specify the first cycle that those operands are available for reading. Please note that this function does not prevent the ready time returned from being earlier than some of the sources are available. This is legal in VLIW models, but can cause problems for superscalar models. Therefore for superscalar models, the ready time returned may have to be increased to the earliest valid cycle it may be scheduled.

7. CONCLUSIONS

This document has described the syntax and features of the HMDES language (Chapter 2), how to use the HMDES-to-LMDES translator (Chapter 3), the format and representation of the MDES information in the LMDES language (Chapter 4), the internal representation of the MDES information in the compiler (Chapter 5), and the use of the MDES interface functions (Chapter 6). Throughout this document, a small MDES *form.hmdes* and supporting C header files have been used to illustrate the concepts (Appendices).

All compilers, in order to produce good code, require a processor description to base code generator and scheduling on. The MDES was designed as a tool to fill this need. It provides a flexible way for a wide variety of processor models to be specified. The use of the MDES tools would eliminate the need for compilers for new processor models to design their own specification language with the added benefit of having more flexibility than traditional, hard coded processor descriptions allow.

The MDES described in this document has been successfully used by the IMPACT compiler since the Summer of 1993. It has been used to model both production RISC and CISC processors, as well as several theoretical processor models. This has allowed the accurate study of many new hardware features without the high implementation time that such studies have taken in the past.

REFERENCES

[1] Hewlett-Packard Co., *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Cupertino, CA: Hewlett-Packard Co., 1990.

[2] M. Johnson, *Am29000 User's Manual.* Sunnyvale, CA: Advanced Micro Devices, Inc., 1990.

[3] Intel Co., *Intel486 Microprocessor Family Programmer's Reference Manual.* Mt. Prospect, IL: Intel Co., 1992.

[4] SPARC International, Inc., *The SPARC Architecture Manual Version 8.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1992.

[5] SPARC International, Inc., *The SPARC Architecture Manual Version 9.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1994.

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

APPENDIX A. FORM.HMDES

The HMDES file used in some of the examples in this document, form.hmdes, is shown below. This file is syntactically correct and was translated to form.lmdes, shown in Appendix E, using Lmdes_build (described in Chapter 3). The processor modeled in this file is a single issue processor with a selected subset of the instructions found in a typical RISC processor.

```
# Sample format for hmdes files
#   by John Gyllenhaal
#
# Sample processor model used based on:
#   AMD-29050 High-Level Machine Description by Roger A. Bringmann
#
Version1

# Text replacement directive
$define HEADER_DIR /home/polaris1/gyllen/mthesis

(Define declaration
    C_header_file "$HEADER_DIR$/ioform.h"
    C_header_file "$HEADER_DIR$/opcform.h"
    C_header_file "$HEADER_DIR$/flagform.h"
    predicates      0
    dest_operands   2
    dest_syncs      3   # Used to construct control/sync dependences
    source_operands 4
    src_syncs       3   # Used to construct control/sync dependences
    processor_model     superscalar
end)

#
# name ((capacity "static" "rotating") (width "size in bits"))
#
(Register_Files declaration
    NULL ((capacity 0 0)   (width 0))
    GPR  ((capacity 191 0) (width 32))

    # 8 bit field for most arithmetic opcodes
    Lit8 ((capacity 0 0)   (width 8))

    # 16 bit field for const and branch direct
    Lit16 ((capacity 0 0)  (width 16))
end)

#
# name (reg_file1 ...)
#
(IO_Sets declaration
    REG (GPR)
    ANY (NULL GPR Lit8 Lit16)
    LIT (Lit8 Lit16)
end)

#
# name (<predicate0 ...>[dest0 ...][src0 ...])
(IO_Items declaration
    IOI_Store   ([][REG ANY REG])

    IOI_Load    ([REG][REG ANY])

    # Standard operand format
    IOI_Std     ([REG][REG REG])
    IOI_Stdi    ([REG][REG Lit8])
    IOI_ubri    ([][REG])           # uncond indirect

    IOI_cbr     ([][REG Lit8 LIT])  # cond
    IOI_cbri    ([][REG Lit8 REG])  # cond indirect

    IOI_call    ([REG][LIT -])      # call direct
    IOI_calli   ([REG][REG])        # call indirect
end)

# Enumerate resources
# If no explicit subscript is declared, it defaults to 0
(Resources declaration
    slot[0]
    ialu0
end)

# If no explicit subscript is specified, it defaults to 0
#
# This model defaults to the keyword slot if none are specified.  Slot is
# the issue slot.

(ResTables declaration
    RL_IBr  (
            (slot[0] 0)
            )
    RL_IAlu (
            (slot[0] 0)
            (ialu0 0)
            )
end)


# name (exception_latency (pred0...) (dest0...) (src0...) (dest_sync0...)
# (src_sync0...)
#
# Defaults to zero if not present
(Latencies declaration
    Lat1  ( 1 () (1 0) () ()    ())
    Lat1m ( 2 () (1 0) () ()    (1))# for stores
    Lat2  ( 2 () (1 0) () (0 1) ()) # for control
    Lat2m ( 2 () (2 0) () (1)   ()) # for loads
end)
```

```
# name ((io_list, resource_list, latency) ...())
(Operation_class declaration
CL_cbr
        ( # conditional branches
            (IOI_cbr RL_IBr Lat2)
        )
CL_Std_IAlu
        ( # Standard one cycle IALU instructions
            (IOI_Std RL_IAlu Lat1)
            (IOI_Stdi RL_IAlu Lat1)
        )
CL_Load
        ( # 2 Cycle load instructions
            (IOI_Load RL_IAlu Lat2m)
        )
CL_Store
        ( # 1 Cycle store instructions
            (IOI_Store RL_IAlu Lat1m)
        )
end)

#
# op_name <opc_flags> (assembly_name <asm_flags> operation_class)
#
(Operations declaration
Lop_BEQ <CBR NI>    (jmpt   CL_cbr)
Lop_BNE <CBR NI>    (jmpf   CL_cbr)
Lop_RTS <RTS NI>    (jmpi   ((IOI_ubri RL_IBr Lat2)))
Lop_JSR <JSR NI>    (call   ((IOI_call RL_IBr Lat2)))
Lop_JSR <JSR NI>    (calli  ((IOI_calli RL_IBr Lat2)))
Lop_ADD             (add    CL_Std_IAlu)
LAMDop_SUBR         (subr   CL_Std_IAlu)
Lop_LD_I <EXCEPT LOAD>  (load   CL_Load)
Lop_ST_I <EXCEPT STORE> (store  CL_Store)
end)
```

APPENDIX B. IOFORM.H

One of the C header files used by form.hmdes (Appendix A), ioform.h, is shown below. This header file specifies the external IDs for all of the Register_Files section entries and some of the IO_Sets section entries found in form.hmdes.

```
/**************************************************************************
 *
 *         File :   ioform.h
 *  Description :   Sample format for processor specific Register_File and
 *                  IO_Set specifiers
 *         Date :   November 1993
 *       Author :   John Gyllenhaal
 *
 **************************************************************************/

/*
 * Subset of AMD specific MDES Register_File and IO_Set
 * specifiers (renumbered), taken from machine/m_amd.h
 */
#define MDES_OPERAND_NULL       0
#define MDES_OPERAND_GPR        1
#define MDES_OPERAND_Lit8       2
#define MDES_OPERAND_Lit16      3
```

```
#define MDES_OPERAND_REG        4
#define MDES_OPERAND_NIL        5
```

APPENDIX C. FLAGFORM.H

One of the C header files used by form.hmdes (Appendix A), flagform.h, is shown below. This header file specifies the operation and alternative flags that may appear in the Operations section entries in form.hmdes. The header file also specifies the bit set for each of these flags.

```
/***************************************************************************
 *
 *         File :   flagform.h
 *  Description :   Sample format for MDES operation flags and
 *                  MDES alternative flags.
 *         Date :   November 1993
 *       Author :   John Gyllenhaal
 *
 ***************************************************************************/

/*
 * Subset of standard Lcode MDES Operation flags (renumbered), taken from
 * Lcode/l_flags.h
 */
#define OP_FLAG_CBR     0x00000001
#define OP_FLAG_RTS     0x00000002
#define OP_FLAG_JSR     0x00000004
```

```
#define OP_FLAG_LOAD    0x00000008
#define OP_FLAG_STORE   0x00000010
#define OP_FLAG_EXCEPT  0x00000020  /* Can cause exception */
#define OP_FLAG_NI      0x00000040  /* non-interlocking, has delay slot */

/*
 * Subset of standard Lcode MDES Alternative flags (renumbered), taken from
 * Lcode/l_flags.h
 * Not used by form.hmdes, but included to show the name format.
 */
#define ALT_FLAG_SPEC   0x00000001  /* speculative form of instr */
```

APPENDIX D. OPCFORM.H

One of the C header files used by form.hmdes (Appendix A), opcform.h, is shown below. This header file specifies the opcode names that may appear in the Operations section entries in form.hmdes. The header file also specifies the integer number that represents each opcode in the compiler.

```
/***************************************************************************
 *
 *         File :   opcform.h
 *  Description :   Sample format for standard Lcode opcode definitions
 *                  and processor specific opcodes
 *         Date :   November 1993
 *        Author:   John Gyllenhaal
 *
 ***************************************************************************/

/* Subset of standard Lcode opcodes (renumbered), taken from Lcode/l_opc.h */
#define Lop_BEQ         1
#define Lop_BNE         2
#define Lop_RTS         3
#define Lop_JSR         4
#define Lop_ADD         5
#define Lop_LD_I        6
```

```
#define Lop_ST_I        7

/*
 * Subset of AMD processor specific opcodes (renumbered), taken from
 * Lcode/lamd_phase1.h.
 */
#define LAMDop_SUBR     500
```

APPENDIX E. FORM.LMDES

The LMDES file used in some of the examples in this document, form.lmdes, is shown below. This file was generated from form.hmdes (Appendix A) using Lmdes_build (described in Chapter 3).

```
Lmdes Version 2

proc_model:  0
sizes:  0  2  4  3  3  0  1

IO_Sets_begin  1     4     7     0     4     32
  0   0 NULL        00000001
  1   1 GPR         00000002
  2   2 Lit8        00000004
  3   3 Lit16       00000008
  4   4 REG         00000002
  5  -1 ANY         0000000f
  6  -1 LIT         0000000c
IO_Sets_end

IO_Items_begin    9     81
  0 IOI_Store     0  0   4  5  4  0
  1 IOI_Load      4  0   4  5  0  0
  2 IOI_Std       4  0   4  4  0  0
  3 IOI_Stdi      4  0   4  2  0  0
  4 IOI_ubri      0  0   4  0  0  0
  5 IOI_cbr       0  0   4  2  6  0
  6 IOI_cbri      0  0   4  2  4  0
  7 IOI_call      4  0   6  0  0  0
  8 IOI_calli     4  0   4  0  0  0
IO_Items_end

Resources_begin    2     17
  0 slot[0]
  1 ialu0[0]
Resources_end

ResList_begin    2     3     2     3  1     15
 0 RL_IBr         1    1    1
   0
   0  0  1
  00000001   00000000
 1 RL_IAlu        2    1    2
   0
   0  0  1
  00000001   00000000
   0  0  1
  00000002   00000000
ResList_end

Latencies_begin    4     22
  0 Lat1          1   1  0   0  0  0  0   0  0  0   0  0  0
  1 Lat1m         2   1  0   0  0  0  0   1  0  0   0  0  0
  2 Lat2          2   1  0   0  0  0  0   0  0  0   0  1  0
  3 Lat2m         2   2  0   0  0  0  0   0  0  0   1  0  0
Latencies_end

Operations_begin     8    11   500    125
   1 Lop_BEQ       1 00000041
  jmpt        00000000    5   0   2
   2 Lop_BNE       1 00000041
  jmpf        00000000    5   0   2
   3 Lop_RTS       1 00000042
  jmpi        00000000    4   0   2
   4 Lop_JSR       2 00000044
  call        00000000    7   0   2
  calli       00000000    8   0   2
   5 Lop_ADD       2 00000000
  add         00000000    2   1   0
  add         00000000    3   1   0
 500 LAMDop_SUBR   2 00000000
  subr        00000000    2   1   0
  subr        00000000    3   1   0
   6 Lop_LD_I      1 00000028
  load        00000000    1   1   3
   7 Lop_ST_I      1 00000030
  store       00000000    0   1   1
Operations_end
```