An Introduction to the FAST System and the ADL Language

Aaron Dittrich and Mike MacGregor

Technical Report TR03-18
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

{dittrich|macg}@cs.ualberta.ca

Contents

List of Figures

Abstract

The following document provides an overview of how to use the Flexible Architecture Simulation Tool, and an in-depth look at the Architecture Description Language. The examples presented are centered around the MIPS RISC architecture. At first a single-stage MIPS architecture is presented, and then the remainder of the examples are centered around a simplified MIPS R2000/R3000 architecture.

Chapter 1

Overview

1.1 Purpose

The purpose of the Flexible Architecture Simulation Tool is to enable rapid prototyping of new processor architectures. Rather than coding a simulation environment manually in a general purpose programming language such as C, FAST provides a much more abstract and higher-level *Architecture Description Language* (ADL) that is used to describe both the processor and instruction set architecture (ISA). A description written in ADL is then used by FAST to automatically generate a microarchitecture simulator, assembler, and disassembler. This reduces the development time from the 18 to 24 man months required to write a simulator in C, to around 3 months to describe the same architecture in ADL. In addition to the time savings gained from using a domain-specific language such as ADL, architectural changes are much more easily made at the architecture specification level, allowing the simulator to be automatically regenerated.

1.2 The FAST toolchain

The FAST toolchain consists of a number of stages that transform a machine description and benchmark program into a usable architecture simulator. In the first stage, the ADL compiler is used to transform an architecture description written in ADL into C++ source code. The generated Makefile is then used to compile this source code into the FAST simulation suite. A 6000 to 8000 line ADL architecture description is transformed into about 35000 lines of C++ code.

Once the three FAST programs are available, benchmark programs can be written, compiled, and executed on the simulated target architecture.

The first step is to take an HLL program written in C and cross-compile it with the MIPS-Ultrix cross compiler provided with the FAST distribution. The result of this is an assembly language program using MIPS-Ultrix system call conventions. Although the FAST system can be targeted to any calling convention, the examples presented here are all centred around MIPS-Ultrix. The FAST assembler is then used to transform the assembly file into a binary file executable on the target architecture via the FAST simulator. The simulator is then responsible for producing all of the performance results necessary to gauge the viability of the target architecture.

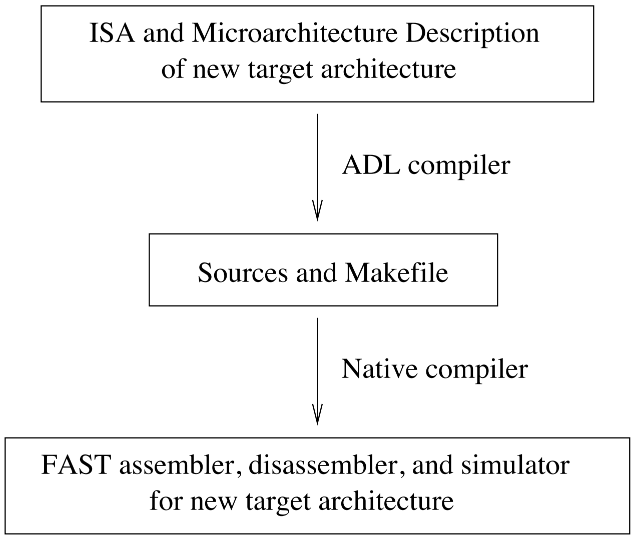
ISA and Microarchitecture Description of new target architecture
ADL compiler
Sources and Makefile
Native compiler
FAST assembler, disassembler, and simulator for new target architecture

Figure 1.1: Toolset generation

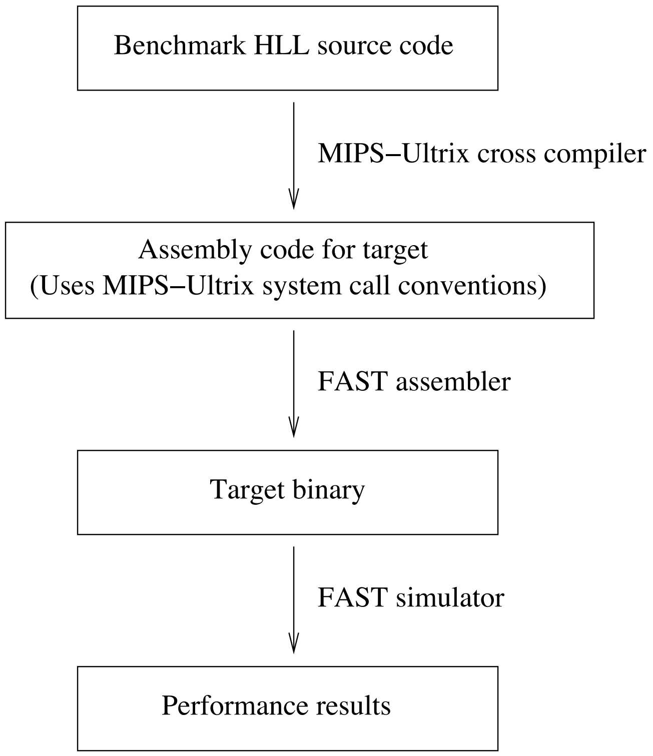
Benchmark HLL source code
MIPS−Ultrix cross compiler
Assembly code for target (Uses MIPS−Ultrix system call conventions)
FAST assembler
Target binary
FAST simulator
Performance results

Figure 1.2: Generating and executing an object file

Chapter 2

Installation

The FAST distribution is available for download at `http://www.cs.ualberta.ca/~dittrich/fast/`. The tarball will extract to a directory named `FAST`. In this directory you will find an `INSTALL` script. Run this script to create the following directories in your home directory:

```
~/object
~/trace
```

The next step requires you to edit your login scripts. If you are using bash, for example, add the following lines to your `.bash_profile` file:

```
FAST_LOCATION=$HOME/FAST/
export FAST_LOCATION
```

so that the `FAST_LOCATION` environment variable points to the location of the `FAST` directory. If you extracted the distribution to a location different than your home directory, edit the above. Setting this variable allows the special MIPS-Ultrix cross-compiler to find the libraries required to properly compile high level code.

The next step is to build the ADL compiler. Go to `$FAST_LOCATION/adl-compiler` and execute the following two commands:

```
make
make dep
```

This takes very little time and places the ADL compiler, called `adl`, in your `~/object` directory.

The next step is to build the MIPS-Ultrix cross-compiler. Go to `$FAST_LOCATION/mips-cross-compiler` directory and execute:

```
make
```

This will create an `xgcc` executable in `$FAST_LOCATION/adl-compiler`. You are free to copy this file to wherever is most convenient for you, provided that you create a link to the `cc1` executable in `$FAST_LOCATION/adl-compiler`. Failure to do so will prevent `xgcc` from running.

Chapter 3

Compiling and running an example

Having successfully built both the ADL compiler and the MIPS-Ultrix cross-compiler, you are now ready to build some examples and run FAST. In `$FAST_LOCATION/Descriptions` you will find two example architectures, one a simple MIPS architecture, and the other a standard MIPS pipelined architecture. Let's assume that we'll choose the `simple` example.

The first step is to build the simulator from the architecture description given in the ADL program files. Go to the `simple` directory and execute the following command:

```
~/object/adl simple.adl
```

This will run the ADL compiler on the ADL description, creating a `~/machine-architecture` directory if it doesn't exist, and placing various source files and a Makefile in `~/machine-architecture/simple`. Compile the source files by executing `make` in the `~/machine-architecture/simple` directory. The result of this will be three new executable files in `~/object`. These will be:

```
simple.asm
simple.dasm
simple.sim
```

The first program, `simple.asm`, is the assembler for the target architecture. This program takes as input a DEC Ultrix assembly file (produced by the cross-compiler `xgcc` and produces an object file executable by the target (i.e. simulated) architecture.

The second program, `simple.dasm`, is a disassembler used for debugging. It is called automatically by the FAST debugger, although it can be called separately as well.

The third program, `simple.sim`, is the simulator for the target architecture. This program simulates the target architecture executing the object file created by the cross assembler.

A more in-depth description of how each of these programs work will be provided later. First we need to write a program to run on the target architecture. Using your favorite editor, write a small C program. Try something short like the following:

```
int main() {
   int i = 0, j = 0;
   for (i; i < 10; i++)
      j = j + i;
}
```

Suppose you called this program `prog.c`. The first step is to translate this program into Ultrix assembly code. Here is where you make use of the xgcc cross-compiler. Execute the following command:

```
xgcc -S -o prog.s prog.c
```

Here the `-S` option tells the compiler that you do not wish to execute the program, but rather want it left in assembly language. Go ahead and take a look at the resulting Ultrix assembler `prog.s` file. The top of it should look something like this:

```
.file 1 "prog.c"

 # GNU C 2.7.2 [AL 1.1, MM 40] DECstation running ultrix compiled by GNU C

 # Cc1 defaults:

 # Cc1 arguments (-G value = 8, Cpu = 3000, ISA = 1):
 # -quiet -dumpbase -o
```

If the top of your file looks like this, then `xgcc` should be working correctly.

Once you have the Ultrix assembly file for your test program, you can proceed to assemble it using the assembler for the target architecture. In this case, the assembler is `simple.asm`. Since FAST implements system services using the MIPS-Ultrix system call format, the standard C libraries cannot be used when assembling a FAST program. Instead, the equivalent MIPS-Ultrix libraries must be used. These libraries are provided in the distribution. Unfortunately, running the assembler while providing the locations of these libraries as options produces a very lengthy command! You may wish to place this in a script:

```
~/object/simple.asm --new new/prog.s                          \
             --lib ${FAST_LOCATION}/libraries/libc/libgcc2    \
             --lib ${FAST_LOCATION}/libraries/libc/time       \
             --lib ${FAST_LOCATION}/libraries/libc/posix      \
             --lib ${FAST_LOCATION}/libraries/libc/termios    \
             --lib ${FAST_LOCATION}/libraries/libc/stdio      \
             --lib ${FAST_LOCATION}/libraries/libc/stdlib     \
             --lib ${FAST_LOCATION}/libraries/libc/ctype      \
             --lib ${FAST_LOCATION}/libraries/libc/string     \
             --lib ${FAST_LOCATION}/libraries/libc/locale     \
             --lib ${FAST_LOCATION}/libraries/libc/signal     \
             --lib ${FAST_LOCATION}/libraries/libc/io         \
             --lib ${FAST_LOCATION}/libraries/libc/misc       \
             --lib ${FAST_LOCATION}/libraries/libc/math       \
             --lib ${FAST_LOCATION}/libraries/libc/global     \
             --lib ${FAST_LOCATION}/libraries/libc/malloc     \
             --lib ${FAST_LOCATION}/libraries/libc/dirent     \
             --lib ${FAST_LOCATION}/libraries/libc/resource   \
             --lib ${FAST_LOCATION}/libraries/libc/pwd        \
             --lib ${FAST_LOCATION}/libraries/libc/socket     \
             --object ~/object/prog prog.s
```

Here the `--new` option produces an assembly file in the `new` directory that can be used to single-step through the code or for debugging the target in the visual debugger. The –lib arguments provide the paths to the various libraries that are required when linking the program. The `--object` argument tells the assembler where you wish to place the object file executable by the target architecture.

Once the object file has been successfully created, it can be executed on the target architecture. To run your program on the simulator, issue the following command:

```
~/object/simple.sim --bench prog new/prog.s
```

The `--bench` option specifies the name of our benchmark file, in this case `prog`. The second argument is the name of the assembly file that was generated by the assembler for single-stepping and debugging. The `--bench` option assumes that the object file is located in the `~/object` directory. If you wish to store your object files elsewhere, substitute the `--bench` option with the `--object` option.

The first time that you execute a particular simulator, a file will appear in your home directory. The name of this file will be `benchmark.summary.<machine_name>`, where `<machine_name>` is the name of the host system, not the simulator itself. Each time you execute a FAST simulator, a line will get added to this file summarizing the execution of a particular simulation. This line is divided into a number of fields. An example file is shown below.

```
simple.sim:testfile.s CPI=1.000000 IPC=1.000000 00:00:00 Retired=4181 Cycles=4181
simple.sim:fsimple.s CPI=1.000000 IPC=1.000000 00:00:00 Retired=771733 Cycles=771733
mips_1.sim:testfile.s CPI=1.626644 IPC=0.614763 00:00:00 Retired=4181 Cycles=6801\
float_cc=0 ld_d_dep=2019 fl_d_dep=0 mem_dc=0 mem_ic=0 fpmulfull=0 fpaddfull=0\
latency_m=0 latency_d=0 latency_f=0 ext_ref=0
```

In this example, three separate programs were run on the simulator `simple.sim`. The first six fields are common to all three cases. The first field gives the name of the target architecture followed by the name of the benchmark. In the case of the simulation we just executed, it should look like `simple.sim:prog.s`. The next field specifies the average number of cycles per instruction (CPI) needed for the target to execute the benchmark. The third field specifies the average number of instructions the target was able to execute per simulator cycle. The fourth field gives the execution time, the fifth field the number of instruction contexts retired, and the sixth the total number of cycles that occurred in the simulation. Depending on the simulation, other fields may appear after these. Some extra data is printed from the third test program. This is program dependent and is up to the programmer to decide.

You should notice that in some cases the number of instruction contexts retired is the same as the number of cycles, while in other cases they differ. This is architecture dependent and will be discussed in more detail later. If the processor is capable of retiring one instruction per cycle, these numbers should be the same. For more complex architectures, however, the number of cycles is typically greater.

The other output file of interest at this point can be found in the `~/trace/statistics` directory. Having run the `simple` simulator on the `prog` benchmark, the statistics file will be called `simple.sim.prog`. If you view the file you will see that there is much more information to be found than in the benchmark summary file.

For example, the output in this file provides data on each instruction in the simulated architecture. The first two columns of data are of the most interest. The first column specifies how many times the given instruction was executed, while the second column gives the percentage of the total instruction count used by that particular instruction. Since the instructions are divided into categories, FAST is able to output the number of instructions executed from each category and the corresponding percentage.

3.1 Other options for the assembler, disassembler, and simulator

The assembler, disassembler and simulator have options in addition to those mentioned above. A brief description of each of these is given below.

Assembler options

`--debug`: Used for debugging the assembler itself.

`--new <filename>`: Specify the name of a new assembly file to create to be used for single-stepping and debugging.

`--object`: Specify the name of the object file to create.

`--version`: Display the version of FAST.

Disassembler options

`--debug`: Used for debugging the assembler itself.

`--version`: Display the version of FAST.

Simulator options

`--asm <filename>`: Specify the name of the assembly source file. This option is useful when debugging and you should therefore specify the path to the file created with the `--new` option on the assembler. Failure to specify the assembly source file when running the simulator will cause the simulator to issue a warning.

`--bench <filename>`: Specify the name of the benchmark. This differs slightly from `--object` in that the argument cannot be a path, but only a single filename. If this argument is used without `--object` then the simulator will look in `~/object` for an object file of name `filename`. The corresponding output will then be named `<simulator>.sim.<filename>`. If the name of the object file is specified explicitly with `--object`, then the name of the benchmark can be arbitrary.

`--continuous`: Causes the simulator to ignore any `pause` statements in the ADL code.

`--debug`: Used for debugging the FAST system itself.

`--monitor`: Run the simulation with the visual debugger.

`--object <filename>`: Specify the name of the object file. The argument to this option can be a path to any object file created by the assembler.

`--params <filename>`: Specify input data for the benchmark program itself. The filename can be a path to any file.

`--stop`: Used in conjunction with `--until`. When this option is specified the debugging window is opened after the number of cycles specified.

`--trace <directory>`: Specify a different directory in which to place the trace data output from the simulator.

`--until <num_cycles>`: Execute the simulation until `num_cycles` cycles.

`--version`: Display the version of FAST.

Chapter 4

A simple ADL program

An ADL program consists of a number of different components, each of which must be in place for the program to produce something meaningful. Unlike other languages such as C, it isn't possible to write a 2 line program in ADL that compiles and runs. In fact, while it is possible to write a relatively "empty" ADL program that is acceptable to the ADL compiler, the resulting C++ code that is generated will still be lacking in some elements. As a result, a minimum useful ADL program will have a somewhat higher complexity.

This chapter introduces a simple ADL program based on the MIPS RISC architecture and describes the different elements of the program and how they fit together to form a complete description of a micro-architecture. In the following chapters, each of these elements will be discussed in much greater detail.

4.1 Components of an ADL program

An ADL architecture description consists of three main components. These are the artifact declarations, the pipeline description, and the ISA description. In addition to these elements, you will normally require some other pieces in your program. The artifact declarations describe the memory visible to the processor. This can be external memory as well as registers and cache memory. The pipeline description is where you describe the architecture itself and how the architecture handles various instructions. The purpose of the ISA description is to specify the formats of the processor instruction set and what data each instruction expects.

The example presented in this chapter is a simplified version of the `simple.adl` program you have worked with already.

4.2 Overview of an ADL program

A typical ADL program has a certain general structure. Often times the program is divided into more than one file, however the functionality is identical. In general, the structure of the program is:

Initial declarations
Artifact declarations
Instruction set declaration
Calling convention specification
Pipeline description
Simulator and assembler supplements

Since the sum of all this code amounts to a fairly long program, it is often good practice to separate it into several different files. Typically the ISA and calling convention are included from different files. In particular, the ISA, normally being the lengthiest component of an ADL program, is best placed in its own file.

4.3 Initial declarations

The first few lines of an ADL program are reserved for some required initial declarations. Among these are the processor name, the endianness of the processor, and any external functions that may be used. Typically, they look like the following.

```
####################################
# Initial stuff                    #
####################################
processor processor_0 highbit 31
begin

lilliput big_endian;

Machineid "simple_mips";

external
begin
   int sprintf(char *,const char *,...);
   int printf(const char *,...);
end external;
```

Comments in ADL begin with a # character. Everything following this character up to the end of the line is ignored by the compiler.

The `processor` keyword signals the beginning of the processor description. The `processor_0` identifier is arbitrary and can be replaced with whatever you wish. The `highbit` keyword tells the compiler how large the instructions are. In this case the program uses 32 bit instructions. The length of all fields that make up each instruction must sum to 32 bits. Following this declaration is the `begin` statement, which signals the beginning of the processor description block.

The `lilliput` keyword decides the endianness of the processor. The two options are `little_endian` and `big_endian`.

The `Machineid` keyword is used to specify the name of the target machine. The argument to it is simply your choice of name in quotes. If you choose `''my_sim''` as the name, then the compiler will produce `my_sim.sim`, `my_sim.asm`, and `my_sim.dasm`.

The remaining items are external declarations. These are optional and serve to tell the ADL compiler of any C library functions you intend to call from within the ADL code. For example, you may wish to insert `printf` statements at points within your pipeline in order to assist in debugging. For each function you intend to use, you must provide its prototype in the `external` section. Prototypes are the only things accepted here. You can not define an entire function body, nor can you specify the prototypes of functions that you have written elsewhere.

4.4 Memory, register, and cache declarations: artifacts

Typically, the next part of an ADL program is where you describe the memory visible to the processor. This consists of external memory, registers, and caches, as well as special registers such as the program counter and instruction register. This section could look something like the following:

```
####################################################
# Memory, register, and cache declarations         #
####################################################

shadow register
   code_start 32, ex_trace   32, _hi        32,
   _lo        33, linebreak  32, dest_r     32,
   check_ex    1, check_mem   1, check_wb    1,
   ex_has_it   1, mem_has_it  1, wb_has_it   1,
   target     32, data_tmp   32, ptemp      32,
   dummy      32, which       2, equal       1,
   less        1, debug_me    1, unordered   1;

latch
   exception       1, new_pc         32,
   branch_input    1, branch_target  32;

shadow register file dtemp[2,32];
shadow register file scratch[2,32];

register file gpr [34,32]  # [34 regs,32 bits each].
   $0       0, $1       1, $2       2,
   $3       3, $4       4, $5       5,
   $6       6, $7       7, $8       8,
   $9       9, $10     10, $11     11,
   $12     12, $13     13, $14     14,
   $15     15, $16     16, $17     17,
   $18     18, $19     19, $20     20,
   $21     21, $22     22, $23     23,
   $24     24, $25     25, $26     26,
   $27     27, $28     28, $29     29,
   $30     30, $31     31, $zero    0,
   $at      1, $v0      2, $v1      3,
   $a0      4, $a1      5, $a2      6,
   $a3      7, $t0      8, $t1      9,
   $t2     10, $t3     11, $t4     12,
   $t5     13, $t6     14, $t7     15,
   $s0     16, $s1     17, $s2     18,
   $s3     19, $s4     20, $s5     21,
   $s6     22, $s7     23, $t8     24,
   $t9     25, $k0     26, $k1     27,
   $gp     28, $sp     29, $fp     30,
   $ra     31;

shadow register
       hi_val                 32,
       lo_val                 32;
```

```
instruction register ir 32;
instruction pointer  pc 32;

memory mem_0  latency 0 width 32;
memory ncache latency 0 width 32;

controldata register
   my_pc       32;

shadow register
   ls_bypass    1, mem_stat     1, access_type 32,
   byte         2, lop_r        6, rop_r        6,
   simm        32, zimm        32, smdr        32,
   store_v     32, lmar        32, smar        32,
   dest        32, dest2       32, lop         32,
   lop2        32, rop         32, rop2        32;

bitconstant
   _BYTE        0 0, _HALFWORD    0 1,
   _TRIPLEBYTE  1 0, _WORD        1 1;

constant generate_trace         0;
constant machine_drained        1;
constant cpc_register_number   32;
constant lo_hi_register_number 32;
```

For the sake of brevity the floating point register file has been omitted. In this section you see a number of different constructs. Each of these will be discussed in greater detail in Chapter 6. Of particular importance are `instruction register` and `instruction pointer`, as well as the `controldata register` declaration. Every ADL program must contain these three declarations. The remainder are left up to the programmer.

4.5 The pipeline description

The pipeline, being the major component of any processor, also makes up a key component of an ADL program. ADL allows the programmer to specify more than one pipeline, such as an integer pipe, floating point pipe, etc. In addition, pipelines can contain an arbitrary number of stages. In order to introduce the pipeline declaration, a single stage pipeline is presented. In Chapter 7 we will discuss more complex pipelines.

The single stage pipeline from the example that you've seen so far is presented below. First the pipeline has to be declared, and then each stage is implemented as a set of procedures.

```
pipeline IPIPE       (s_ID);
source s_ID;
```

The `pipeline` keyword is used to declare the pipeline, which can be given a name of your choosing. Following the name is a comma-separated list of stage names in parenthesis. In this case our single stage is named `s_ID`. The source keyword allows the programmer to specify which stages of the pipeline can be source stages. Since this example only has a single stage, that stage must act as the source. Failure to specify at least one source is a compile-time error.

Once the pipeline has been declared, its stages can then be implemented. Processor major cycles are divided into minor cycles, a prologue, intermissions, and an epilogue. Each stage much implement at least a prologue and epilogue. This will be discussed in more detail in Chapter 7.

Let's first look at the prologue.

```
procedure s_ID prologue
begin
   my_pc = pc;                     #- Required by jal.
   ir = mem_0[pc];
   if debug_me then
      printf("PC is %08x\n",my_pc);

   if (branch_input) then
      begin
         branch_input=0;
         pc=branch_target;
      end
   else
      pc=pc + 4;

   ### Fetch input registers. Sign extend the immediate portion ###
   decode;
```

The first couple of lines determine the next instruction based on the program counter. The program counter indexes into the main memory and the result is stored in the instruction register. Following this is an illustration of the use of external functions. The C `printf` function is called if the `debug_me` variable has a non-zero value. Unlike in C/C++ where blocks are contained within braces, in ADL the beginning of a block is denoted by the `begin` keyword while the end of the block is denoted with `end`. If a block contains only one statement, the `begin` and `end` keywords are not necessary. The next `if` statement checks whether the previous instruction was a branch instruction. If that turns out to be the case, the `branch_input` flag is reset and the program counter is changed to the branch target. Otherwise the program counter is incremented by one word.

The `decode` keyword decodes the attributes of the current instruction, which causes them to become read-only variables accessible by the pipeline stages.

```
   dest_r = ordinal(dest_reg);

   ### Read operands ###
   case lop_type of
   begin
      cpc_register:
         lop_r     = cpc_register_number;
         lop       = fpr[lop_r];

      integer_register:
         lop_r=rs;
         lop=gpr[lop_r];

      float_register  :
         lop_r=fs;
```

```
        lop =fpr[lop_r];

      double_register :
        lop_r=fs;
        lop =fpr[lop_r];
        lop2=fpr[lop_r+1];

      special_input   :
        lop_r     = 2;
        lop       = gpr[2];

      lo_hi_register:
        lop_r     = lo_hi_register_number;
        lop=gpr[lop_r];
        lop2=gpr[lop_r+1];
   end;

   case rop_type of
   begin
      cpc_register:
        rop       = tf;

      integer_register:
        rop_r=rt;
        rop=gpr[rop_r];

      float_register :
        rop_r=ft;
        rop =fpr[rop_r];

      double_register:
        rop_r=ft;
        rop =fpr[rop_r];
        rop2=fpr[rop_r+1];
   end;

   if (i_class == branch_class) then
      condition_code(lop,rop);
end s_ID;
```

The next series of statements determine the types of left and right operands. The case statement is analogous to a C switch statement. `lop_type` and `rop_type` are *attributes*. These can be thought of as enumerated data types. Their declaration will be discussed in Chapter 8. In the case of `rop_type`, for example, it can take on the values `float_register,integer_register,double_register, special_input, cpc_register, lo_hi_register`. Depending on what types the operands are, different actions are taken.

```
procedure s_ID epilogue
begin
   case dest_type of
   begin
```

```
      lo_hi_register:
              gpr[dest_r]=dest;
              gpr[dest_r+1]=dest2;

      integer_register :
              gpr[dest_r]=dest;

      cpc_register     :
      float_register   :
              fpr[dest_r]=dest;

      double_register  :
              fpr[dest_r]=dest;
              fpr[dest_r+1]=dest2;

      else : if ordinal(dest_type) then
               printf("skip skip %d\n");
   end;

   ### Output trace data ###
   sprintf(pointer(scratch),"%6d\n",my_pc - code_start);
   if generate_trace then
   if builtin fast_write_file(ex_trace, 7, scratch) ^= 7 then
     begin
        builtin perror("Store ex trace");
        builtin simulation_exit(-1);
     end;

   retire stat;
   newcontext;
end s_ID;
```

Since the operand types were already determined in the prologue of the stage, what's left to be determined now is the type of the destination register. This occurs in the epilogue above. The `retire` keyword is responsible for deallocating the instruction context and is always found in the last pipeline stage. The `newcontext` keyword creates a new instruction context. Instruction contexts will be discussed in more detail later.

4.6 Simulator and assembler supplements

Each ADL micro-architecture description requires both a simulator and an assembler supplement. These two sections of code are not written in ADL but rather in C++. The ADL compiler leaves these two sections alone and includes them in the C++ code that it generates.

The purpose of the simulator supplement is to set up the initial program image for the target-specific machine code you will be executing. Essentially this amounts to passing parameters to the simulated 'process'. In reality there is no separate process since execution of the code is simulated. In MIPS parameters are passed in \$4, \$5 etc. For a different architecture/calling convention, you need to map the remainder of the simulator parameters to the appropriate registers/stack locations so that the simulated program can find them in the proper place. Similarly, the return address register is mapped to a call to the O/S routine *exit* which is implemented by the simulator so that the simulated program will perform a 'return from sub' and the *exit* function will perform *syscall-exit* which in turn will call *simulation-exit.*

Let's look at the simulator supplement for the MIPS architecture.

```
simulator begin
%%

   int pseudo_procedure_call;
   int pseudo_pipeline_flush;
   int nop_line[4];
   int nop_flush[4];

   void initialize_machine(code_file_header &H, int arg_count, char **args, int stack_top) {

      while (stack_top & 0x7) stack_top--;
      $sp = (stack_top - 8);

      pseudo_procedure_call = (int)&nop_line;
      nop_line[0]=0;
      nop_line[1]=0;
      nop_line[2]=0;
      nop_line[3]=0;

      pseudo_pipeline_flush = (int)&nop_flush;
      nop_flush[0]=0;
      nop_flush[1]=0;
      nop_flush[2]=0;
      nop_flush[3]=0;

      for (int i=0; i < fast_ext_count; i++) {
         if (strcmp(externals[i].name,"exit") == 0) {
            $31 = e_ref_table_start + ((int)&externals[i] - (int)&externals);
            break;
         }
      }

      $4  = arg_count;
      $5  = (int)args;
      $28 = H.sbss_segment_start;
   }

%%
end simulator;
```

The simulator supplement begins after the `simulator begin %%` keywords and ends before the `%% end simulator` keywords. The first two `int` variables and the two `int` arrays must always be declared with the same names. The simulator will expect these to be in place. The `initialize_machine` function is the function the simulator calls whenever it loads a new codefile to execute. This function takes four arguments: a code file header, the number of arguments to the program, the array of arguments, and a pointer to the top of the simulated machine's stack. The purpose of the `for` loop is to find the location of the external call `exit`. The appropriate registers are then loaded with the program's parameters.

The purpose of the assembler supplement is to implement the addressing modes for the simulated ISA. In MIPS there is only one addressing mode implemented in the hardware but the assembler needs to synthesize different addressing modes. The current version of the ADL language lacks the capability to describe the addressing modes of the architecture. Hence these routines need to be written manually.

4.7 Instruction set declaration

Not unlike an ADL program in general, the instruction set specification is made up of a number of different segments. As this is typically the longest part of an ADL program, it is good practice to place it in a separate file. We will be looking at the MIPS ISA, which is found in the `mips-instruction-set-simple.adl` file.

In general, the instruction set specification consists of the following components.

Field type declarations
Attribute declarations
Assertions
Opcode constants
Instruction declarations

The field type declarations tell the compiler which fields can make up a given instruction. Fields can have a number of different types as you'll see in Chapter 8. A very simplified version of this section is presented below.

```
type
   opcode       constant      field         31   6,
   rs           register      fixedfield    25   5,
   rt           register      fixedfield    20   5,
   rd           register      field         15   5,
   shamt        integer       field         10   5,
   funct        constant      field          5   6,
   code         integer       field         25  20;
```

These fields correspond to the standard fields for the MIPS instruction set. For example, the `shamt` field, or *shift amount*, is an integer field and starts at bit 10 and is 5 bits wide.

Next comes the attribute declarations. An example attribute section is shown below.

```
attributes
     i_class       : float_class,
                     integer_class,
                     branch_class,
                     long_integer_class;

     i_cycles      : single_cycle,
                     multiple_cycles;
end;
```

An attribute declaration consists of an attribute type, a colon, and then a comma separated list of possible values. The `end` keyword always follows the last attribute.

Assertions perform the same function as assertions in a C program.

```
assertion

   1 : i_cycles  == single_cycle      : (exu == integer_unit)     |
                                        (exu == call_unit)        ;

   2 : (lop_type == integer_register) |
       (rop_type == integer_register) : (exu == integer_unit)      |
                                        (exu == call_unit)         |
                                        (exu == divide_unit)       |
                                        (exu == load_unit)         |
                                        (exu == store_unit);
end;
```

In this example, the first assertion checks to make sure that all single cycle operations are executed by the integer unit or call unit. The program will check whether the condition `i_cycles == single_cycle` is true. If this is the case, it will then evaluate the second expression. If the second expression is found to be false, then an exception is raised.

Preceding the actual instruction set declarations you must declare the opcode bit constants for your instruction set. In the case of the two instructions you'll see below, the only opcodes required are the following.

```
bitconstant
   #
   #          31..29  28..26
   #          ------  ------
   _special    0 0 0   0 0 0 ,
   _sll        0 0 0   0 0 0 ,
   _syscall    0 0 1   1 0 0 ;
```

Declaring bit constants is just a matter of specifying a comma-separated list of names followed by bit strings.

The instruction declarations begin with the keyword `instruction`. Following this keyword is a comma-separated list of instructions. The two instructions presented here are the `syscall` instruction and `sll` instruction.

```
instruction

   syscall code
         emit opcode=_special code funct=_syscall
         attributes
             (
              i_class   : integer_class,
              i_cycles  : multiple_cycles,
              exu       : call_unit,
              c_what    : none,
              dest_type : none,
              lop_type  : special_input,
              rop_type  : none,
              i_type    : alu_type,
              dest_reg  : none
             )
```

```
        begin
           exact s_ID
                 $2=builtin do_mips_syscall(lop,$4,$5,$6,$7);
                 dest=0;
                 $7=0;
           end;
        end,

  syscall macro
        begin
           syscall : code=0;
        end,
```

The first line gives the instruction mnemonic and its associated fields. The second line tells the assembler what to output when it parses the instruction. The attributes component sets all the respective attributes for the given instruction. For example, a system call requires multiple cycles to execute and has no destination register. Within the RTL block exist any statements that produce the result of the instruction when executed. In the case of `syscall`, a built-in MIPS system call is executed. The `syscall macro` declaration is enables the use of pseudo ops and will be discussed in Chapter 8.

The `sll` instruction obeys the same syntax as the `syscall` instruction, while demonstrating the use of the register fields.

```
  sll rd rt shamt
        emit opcode=_special rs=0 rt rd shamt funct=_sll
        attributes
            (
             i_class   : integer_class,
             i_cycles  : single_cycle,
             exu       : integer_unit,
             c_what    : none,
             dest_type : integer_register,
             lop_type  : none,
             rop_type  : integer_register,
             i_type    : alu_type,
             dest_reg  : rd
            )
        begin
           exact s_ID
                 dest=(+rop) << (+shamt);
           end;
        end;
```

The `dest` and `rop` registers were declared as artifacts in the main file. The `<<` operator performs the same function as in C. Therefore, `dest` gets the value stored in `rop` left-shifted by `shamt` bits.

Chapter 5

ADL Operators, Control Structures, and Procedures

Like any programming language, ADL has a variety of operators, control structures and a means of specifying procedures. Procedures come in handy when you must frequently use the same LRTL statements.

5.1 Operators

The following is a list of operators and a brief explanation of their meaning. Where necessary, an example is also presented.

`=`
Assignment operator. Provides a means of assigning a value to a variable, register, or field. The right hand side may be another variable, constant, or attribute.

`==`
Equality operator. Tests whether the left hand side is equal to the right hand side. Returns true if equal, false otherwise.

`<`
Less than operator. Tests whether the left hand side is less than the right hand side. Returns true if less than, false otherwise.

`>`
Greater than operator. Tests whether the right hand side is greater than the left hand side. Returns true if greater than, false otherwise.

`^=`
Not equal operator. Tests whether the left hand side is not equal to the right hand side. Returns true if not equal, false if equal.

`<<`
Left shift operator. Shifts the bits in the left hand side left by the number of spaces specified on the right.

`>>`
Right shift operator. Shifts the bits in the left hand side right by the number of spaces specified on the right

hand side.

`+`
Addition operator. Adds the value of the left hand side and the right hand side and returns the sum.

`-`
Subtraction operator. Subtracts the value of the right hand side from that of the left hand side and returns the difference.

`*`
Multiplication operator. Multiplies the value of the left hand side with that of the right hand side and returns the result.

`/`
Division operator. Divides the value of the left hand side by that of the right hand side and returns the result.

`%`
Modulus operator. Returns the modulus of the left hand side by the right hand side.

`&`
Logical AND/bitwise AND operator. When the operands are boolean values, this operator performs a logical AND. When the operands are bit values, it performs a bitwise AND.

`|`
Logical OR/bitwise OR operator. When the operands are boolean values, this operator performs a logical OR. When the operands are bit values, it performs a bitwise OR.

`|<`
Bit extension operator. Extends a bit on the left hand side the number of times specified on the right hand side and returns the result. For example, suppose you wish to extend the 1 bit 16 times. You could use: `1 |< 16`. This would return 16 1 bits.

`||`
Concatenation operator. Concatenates the bits on the left hand side with those on the right hand side and returns the result. For example, suppose you wish to extend a 1 bit 16 times and concatenate that to another value. You could use `1 |< 16 || x`.

`[ ]`
Index operator. Indexes into an array, register file, or pipeline stage. If you are indexing into a register file, an example would be `gpr[25]`, which will return the contents of the 26th register file location (indexing begins at 0). If you are indexing into a pipeline stage, the syntax is the same, however the value of the index is replaced with the stage name. For example, to get the value of `dest` in `s_MEM`, the following syntax is used: `dest[s_MEM]`. More on this later.

`[ : ]`
Bit group operator. Used in conjunction with the dot operator. This extracts a sequence of bits from a register or variable. For example, if you have a 32 bit integer register and wish to extract bits 16 through 23, the following syntax is used: `$7.[16:8]`. The first argument is the register name, the second the start bit, and the third the width.

.
Dot operator. Used to pass a parameter to an artifact when it represents either an *lvalue* or *rvalue*.
:
Field width operator. This is used to specify the width of the argument to a procedure. For example: `x:8` means that `x` is 8 bits wide.

5.2 Control structures

ADL provides two conditional statements that can be used to test the values of variables, registers, and attributes. The first is an `if-else` construct, while the second is a `case` statement that is analogous to a C `switch` statement.

The `if-else` statement works as follows.

```
if (condition ) then
    block
else
    block
```

The `else` clause is optional. A block consists of a `begin` keyword, followed by statements, followed by an `end` keyword. Note that if the block only contains a single statement, the `begin` and `end` keywords are not necessary.

The `case` statement can only be used with attributes. Attributes will be discussed in a chapter 8. Essentially it operates in much the same way as a `switch` statement. The format looks like

```
case attribute of
begin
    value1 : LRTL-statements
   .
   .
   .
    valueN : LRTL-statements
end
```

ADL also provides two looping control structures. The first is the `forall` statement that is used to initialize all the elements of a register file or integer array. Its use is demonstrated with the following example.

```
integer array my_array[1024,32];
forall my_array=0;
```

An array of 1024 32-bit integers was declared, and then each integer was initialized to 0.

The second loop structure more closely resembles a traditional `for` loop. This example performs the same function as the previous, but using the `for` construct.

```
integer array my_array[1024,32];
for i = 0 step 1 until 1023 do
   my_array[i] = 0;
end;
```

The flexibility is somewhat more limited than a C/C++ `for` loop, since a boolean condition is not checked at the end of each iteration, but rather only a counter is incremented.

5.3 Procedures

ADL procedures can be used when the same code must be executed multiple times or to break up large chunks of code into more manageable segments. The method of returning values from ADL procedures, however, is somewhat different than in some other languages. Below is a simple procedure that we can consider.

```
procedure sign_extend_24 (x:8)
begin
   sign_extend_24=(x.[7:1] |< 24) || x;
end sign_extend_24;
```

A procedure begins with the `procedure` keyword and is followed by the name of the procedure and then the parameter list. As mentioned in the previous section, you can specify the size of each parameter with the `:` operator. In this case, the `sign_extend_24` procedure takes an argument that is 8 bits wide. Between `begin` and `end` keywords lies the body of the procedure. In the case of this procedure, bit 7 of `x` is extended 24 spaces and concatenated with the original 8 bits of `x`, resulting in a new 32 bit value. The trick to returning values from procedures is to assign the return value to a variable of the same name as the procedure itself. This variable does not have to be explicitly declared beforehand.

Procedures can take any number of arguments. If a procedure takes no arguments, there is no need to have the parenthesis after the procedure name, and when calling the procedure, you do not need to use parenthesis either (such as with a C function without arguments).

Procedures without return values are said to be `untyped`. This keyword is specified after the list of formal parameters of the procedure. For example, suppose you wish to write a procedure called `do_forwarding` that does not need to return a value and does not take any arguments. It would look like the following:

```
procedure do_forwarding untyped
begin
   .
   .
   .
end do_forwarding;
```

To call the procedure from elsewhere in your code, just use the `do_forwarding` procedure name among your LRTL statements.

Variables, registers, and latches declared globally are also visible from procedures and can be modified by the procedures.

Chapter 6

Artifacts

Artifacts in ADL correspond to hardware objects with well-established operational semantics. ADL supports these objects as built-in types. When declaring artifacts, attributes are supplied that determine the behavior of a specific implementation of the artifact.

An artifact declaration can take on one of the following six forms:

```
artifact-declaration => register-declaration
                      | register-file-declaration
                      | memory-port-declaration
                      | cache-declaration
                      | buffer-declaration
                      | latch-declaration
```

6.1 Register declarations

A register declaration declares a simple register artifact. The only attribute associated with a register is its size in bits. The syntax of a register declaration is the following:

`(shadow) register` *identifier integer* `;`

For example, if you wanted to declare a register named `my_reg` of size 32 bits, you would use the following syntax:

```
register my_reg 32;
```

The optional `shadow` attribute makes the given register invisible to the instruction set.

6.2 Register file declarations

A register file declaration declares an array of registers. Not surprisingly, it requires at least two attributes, those being the number of registers in the array and the size of each register. Register files can also be given the attribute `shadow`. An example shadow register file declaration is presented below.

```
shadow register file gpr[64,32];
```

This declares a shadow register file named gpr that has 64 entries of 32 bits each.

Register files have the quality that each entry in the register file can be assigned one or more aliases. This is taken care of in the same declaration along with the register file. For example, if you wanted to declare a register file with 8 32-bit entries while assigning aliases for each of them, you could use the following syntax:

```
register file mrf[8,32]
             $r0 0,
             $zero 0,
             $r1 1,
             $r2 2,
             $r3 3,
             $r4 4,
             $r5 5
             $r6 6
             $r7 7;
```

Following the initial declaration is a comma-separated list of aliases and indices. Aliases always begin with the `$` character. Notice that register 0 has two aliases. You are free to specify as many aliases as you wish for a given entry, or none at all.

If you want to store some value into register 4 of this register file, you now have two options. Both of the following RTL statements are valid:

```
mrf[4] = data_value;
$r4 = data_value;
```

6.3 Memory port declarations

Memory declarations define memory ports with given access latencies in units of machine cycles and and data path widths in units of bits. There are no optional attributes for memory declarations, and as such all such declarations have the following format:

`memory` *identifier* `latency` *num_cycles* `width` *num_bits* `;`

The memory declaration declares a memory port, and not the memory itself. As such, the size of the memory cannot and need not be specified. It is assumed to be arbitrarily large to suit the needs of the program being executed on the target architecture. Multiple ports can be specified to the same memory. An example memory declaration where the latency is 20 machine cycles and the data path width is 32 bits would look like the following:

```
memory main_mem latency 20 width 32;
```

6.4 Cache declarations

The `cache` artifact is used to declare either instruction or data caches. Caches are *stackable*, meaning they can be used to construct memory hierarchies. Because of this, they need to be declared in relation to the memory object directly below them in the hierarchy. This is accomplished with the `of` clause. The general syntax of this looks like:

(`data` | `instruction`) `cache` *identifier* `of` *identifier size words_per_line* `;`

For example, suppose you wish to declare a 64 kilobyte direct-mapped instruction cache of the main memory declared in the previous section. The following would be an acceptable declaration:

```
instruction cache icache of main_mem 64 4;
```

Now suppose you wanted a data cache hierarchy from the same memory port. The following would be acceptable:

```
data cache l2 of main_mem 64 4;
data cache l1 of l2 8 4;
```

This declares a 64 kilobyte level 2 cache with 4 words per line. Following this, an 8 kilobyte level 1 cache of the level 2 cache is declared.

Instead of always storing or extracting complete words from cache or memory locations, it is possible to access or store bytes and halfwords as well. Suppose you wanted to store a single byte to a level 1 cache location and later extract a half word from the same location. This could be accomplished in the following manner:

```
bitconstant
   _BYTE      0 0,
   _HALFWORD  0 1;

l1.(_BYTE)[addr] = data_value;
data_value = l1.(_HALFWORD)[addr];
```

This illustrates the use of the dot operator.

6.5 The `access_complete` statement

The `access_complete` statement allows you to query the status of the result from a cache or memory access. The statement returns true if the operation has been completed successfully, and false otherwise. Slow artifacts and structural hazards are possible causes of `access_complete` returning false.

An example of how this statement can be used is seen below.

```
my_pc = pc;
ir = icache[pc];
if access_complete then
   begin
      new_pc = pc+4;
      unfreeze;
   end
else
   begin
      freeze;
      stall mem_ic;
```

```
    end;
```

The second statement attempts to fetch the next instruction from the instruction cache. If this operation completes successfully, the program counter is incremented and the pipeline is unfrozen. Should there be a reason for the access not to complete successfully in that cycle, the pipeline is frozen and the current stage stalled. `freeze`, `unfreeze`, and `stall` statements will be discussed in the Chapter 7.

6.6 The `instruction register` and `instruction pointer`

Recall in Chapter 4 when you saw the `instruction register` and `instruction pointer` declarations. These two declarations are critical in every ADL description. Failure to provide them in the program results in a compile-time error.

The `instruction register` is where the next instruction gets loaded at the beginning of your pipeline. This instruction is determined by means of the `instruction pointer`, also known as the *program counter*. The instruction pointer stores the memory address of the next instruction.

Declaration of these two artifacts looks like this:

```
instruction register ir;
instruction pointer  pc;
```

As you might expect, `pc` denotes *program counter*. Once these two are declared, they can be used in your pipeline.

```
my_pc = pc;
ir = icache[pc];
```

The above is an excerpt from the *instruction fetch* stage of the MIPS architecture. The `my_pc` variable is a controldata register, and is used to keep track of the given instruction's address in memory. These registers are discussed in Chapter 7. The instruction pointer is used to index into the instruction cache to find the next instruction word. Once this is available, the *instruction decode* stage can then decode the instruction and its operands.

6.7 Other artifacts

In addition to memory, registers, and cache, there are other items that can be declared. One such artifact is the `latch`. Latches can be used in place of registers when only a single bit needs to be stored, and can be accessed from anywhere in the description. An example latch declaration is presented below:

```
latch
    exception   1,
    is_branch   1;
```

ADL also provides a means of declaring constants and integer variables. While not actual architectural artifacts, they provide some means of abstraction when writing your code.

Constants and bit constants are declared in a similar manner. The syntax for declaring constants is:

`constant` *constant-name constant-value*

To declare bit constants, use the following syntax:

`bitconstant` *constant-name bit-string*

where a bit string is simply a string of 0s and 1s separated by spaces.

The remaining item that ADL provides is the `integer` variable. Integers are general-purpose variables that can be used to store whatever integer values are necessary in your ADL description. To declare an integer, use the `integer` keyword followed by a name:

`integer` *variable-name*

6.8 User-defined artifacts

When the selection of builtin artifacts isn't enough, ADL allows you to define your own artifacts with custom properties and behaviours. Artifacts are specified with the `artifact` keyword followed by a number of attributes. An artifact definition consists of a number of procedures that describe the behaviour of the artifact.

Consider the following example. This artifact defines a simple 2-bit branch predictor.

```
artifact predictor attributes (size)
begin

   # Artifact data
   integer array predict_bits[size,2];
   integer hash_val;

   # Initialization stuff
   initialization
   begin
      forall predict_bits=0;
   end initialization;

   # What to do if it is an rvalue
   rvalue (addr)
   begin
      hash_val = addr.[10:8];
      rvalue = predict_bits[hash_val];
   end rvalue;

   # What to do if it is an lvalue
   lvalue (addr, rval)
   begin

      hash_val = addr.[10:8];
      if (rval == 0) then # Not taken
         begin
            if (predict_bits[hash_val] == ZERO) then
            begin
               predict_bits[hash_val] = ZERO;
            end
            else if (predict_bits[hash_val] == ONE) then
            begin
               predict_bits[hash_val] = ZERO;
            end
            else if (predict_bits[hash_val] == TWO) then
            begin
               predict_bits[hash_val] = ZERO;
            end
            else if (predict_bits[hash_val] == THREE) then
            begin
               predict_bits[hash_val] = TWO;
            end;
         end;

      if (rval == 1) then # Taken
         begin
            if (predict_bits[hash_val] == ZERO) then
            begin
               predict_bits[hash_val] = ONE;
            end
            else if (predict_bits[hash_val] == ONE) then
            begin
               predict_bits[hash_val] = THREE;
            end
            else if (predict_bits[hash_val] == TWO) then
            begin
               predict_bits[hash_val] = THREE;
            end
            else if (predict_bits[hash_val] == THREE) then
            begin
               predict_bits[hash_val] = THREE;
            end;
         end;

   end lvalue;

   # Print statistics
   statistics
   begin
      builtin printf(``Branch predictor %s\n'', myself.name);
   end statistics;

end predictor;
```

Typically the first part of the artifact definition consists of data declarations. In the above example, only one array is required, that which holds the two bits that determine the branch prediction. The size of the array is set at the value passed via the size attribute.

When an instance of an artifact is declared in an ADL program, the `initialization` procedure of the artifact is called. The purpose of this procedure is to perform any initializations on the artifact data. The statistics procedure is called at the end of the simulation. Any statements that you insert into this procedure will be performed when the <machine_name>.sim file is being written. Clearly, it makes the most sense to insert printf statements here. This is accomplished, as discussed earlier, with the builtin keyword. The myself keyword refers to the instance of the artifact being operated upon (similar to the this keyword in C++). The name you assigned the artifact can be printed this way.

The key to defining the behaviour of the artifact is via the rvalue and lvalue procedures. These procedures characterize how the artifact behaves when it is an r-value and l-value, respectively. In the case of the branch predictor artifact above, when it is an r-value we simply wish to return the prediction bits for the given address. The rvalue procedure takes one argument, which corresponds to the index passed to the artifact instance elsewhere in your ADL code. For example:

```
predictor my_predictor size=256;

if (my_predictor[pc] == 0) then
   begin
      ...
   end;
```

Values for the attributes of the artifact are passed during declaration of the artifact instance by giving the name of the attribute, followed by an = sign, followed by the value. Artifacts are accessed as r-values by passing an index value to the artifact. The return value can then be used as necessary. Recall that values are returned from procedures by assigning a value to the procedure name. In the above case, a value is assigned to rvalue.

Accessing an artifact as an l-value works in a similar manner, except that the procedure takes two arguments. The first argument corresponds to an index, the second to the r-value in the expression. Consider the following statements:

```
bitconstant TWO 1 0;
my_predictor[pc] = TWO;
```

The constant value TWO is assigned to the location specified by pc.

Aside from the basic procedures mentioned above, an artifact definition can contain other procedures, separate pipelines, and other artifacts.

Chapter 7

Pipelines

Specifying the pipeline(s) of the micro-architecture involves declaring the stage names and ordering of the stages, followed by specifying procedures to implement each of these stages. In addition, you must specify which data make up the *instruction context*.

7.1 The instruction context

As an instruction progresses through the pipeline, a set of data values is carried along through each pipeline stage. These data values are grouped together in a structure called `controldata`. Since there is only one `controldata` declaration in an ADL program, each pipeline stage has the same type of context. The instruction context is the union of all the data required by all the stages in the pipeline.

Declaring the instruction context follows a pattern similar to other register declarations. This declaration is given by the `controldata register` keyword followed by a list of register names and sizes. Failure to declare the `controldata register` in an ADL program is a compile-time error.

In a pipelined MIPS architecture, the following data might make up the instruction context.

```
controldata register
   my_pc       32,
   ls_bypass    1,
   mem_stat     1,
   access_type 32,
   byte         2,
   lop_r        6,     # lop_r indicates the register number for the lop.
   rop_r        6,     # rop_r indicates the register number for the rop.
   dest_r       6,     # dest_r holds the register number to write.
   simm        32,     # Sign extended immediate.
   zimm        32,     # Zero extended immediate.
   smdr        32,     # Store Memory data register.
   store_v     32,
   lmar        32,     # load memory address register.
   smar        32,     # store memory address register.
   dest        32,     # dest holds the value to be written.
   dest2       32,     # dest holds the value to be written.
   lop         32,     # lop holds the left operand value.
   lop2        32,     #
   rop          32,      # rop holds the right operand value.
   rop2         32;      #
```

Elements of the `controldata` structure are accessible from pipeline stages and in the semantic parts of instruction declarations. Both qualified and unqualified access modes are possible. In the case of unqualified access, the data accessed is that of the respective pipeline stage performing the access, or the stage associated with the label for the LRTL segment in the semantic portion of the instruction declaration. For example, in the following `add` instruction, the LRTL statement under the `exact s_EX` label will access the `dest`, `lop`, and `rop` instruction context data from the `s_EX` pipeline stage.

```
add rd rs rt
   emit opcode=_special rs rt rd shamt=0 funct=_add
      attributes
         (
         ...
         )
      begin
         exact s_EX
            dest=lop + rop;
         end;
      end,
```

The syntax for qualified access of data involves specifying the name of the `controldata` element along with its respective pipeline stage. The general syntax looks like *controldata-element*[*stage-name*]. For example, in our pipelined MIPS processor, we may wish to check for data hazards and take appropriate action.

```
if has_context s_EX then
   if dest_type[s_EX] == lo_hi_register then
      stall;
if has_context s_MEM then
   if dest_type[s_MEM] == lo_hi_register then
      stall;
if has_context s_WB then
   if dest_type[s_WB] == lo_hi_register then
      stall;
```

In this example, the instruction contexts of three different pipeline stages are accessed from the `s_ID` stage.

7.2 Implementation of pipeline stages

Pipelines are implemented in ADL in a distributed fashion in the form of procedures. As mentioned previously, machine cycles in FAST are divided into several minor cycles. These minor cycles include a *prologue*, a series of *intermissions* and an *epilogue*. In general, the prologue is responsible for receiving an instruction context from the previous pipeline stage, the intermissions operate upon the instruction context, and the epilogue sends the instruction context to the next pipeline stage.

For each pipeline stage, you may specify a procedure for each minor cycle. A procedure for both the prologue and epilogue must be declared for each stage, while procedures for the intermissions are optional.

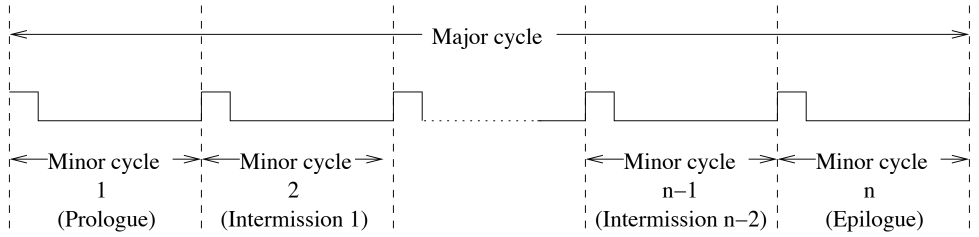
Major cycle
Minor cycle 1 (Prologue)
Minor cycle 2 (Intermission 1)
Minor cycle n−1 (Intermission n−2)
Minor cycle n (Epilogue)

Figure 7.1: FAST processor cycles

The general format for defining procedures is as follows.

```
procedure stage-name [prologue | intermission | epilogue]
begin
    RTL statements
    .
    .
    .
end stage-name ;
```

The RTL statements within the procedure define the behavior of the stage. Depending on the stage and on the micro-architecture, you may need to check for data or structural hazards, vary behavior based on the type of instruction in the stage, etc. In order to accomplish many of these functions, ADL provides several keywords that are used to affect the behavior of the pipeline.

7.3 Instruction flow through the pipeline

Instruction flow through pipelines is facilitated through a number of keywords that affect the behavior of the pipeline and instructions. The `send` statement is responsible for forwarding a particular instruction context from one stage to the next. `send` statements can be successful or unsuccessful. A successful send is achieved when the next stage is idle or performing a send in the same cycle. If a stage does not execute a send in a particular cycle, send operations of preceding stages will fail in that cycle, and those stages will repeat the send operation in the next cycle. All pipeline stages, with the exception of the final stage, must execute a `send` somewhere in their epilogue minor cycle.

The `send` keyword takes a single argument corresponding to the pipeline stage to which the current instruction context should be sent. This is important since instruction contexts can be sent to more than one pipeline stage. For example, a floating point add instruction may need to be forwarded to a different stage than an integer add instruction. This becomes especially apparent when the micro-architecture has more than one pipeline.

An example of how the `send` statement is used is given below.

```
procedure s_ID epilogue
begin
   if ((exu == integer_unit ) | (exu == load_unit) |(exu == store_unit)) & (send_enabled(s_EX) == 0) th
      stall;
   if (exu == f_add_unit) & (send_enabled(f_add1) == 0) then
      stall fpaddfull;

   if (exu == f_mul_unit) & (send_enabled(f_mul1) == 0) then
      stall fpmulfull;

   if dest_type == float_register then
      fpr_tag[dest_r]=my_pc
   else
   if dest_type == double_register then
      begin
         fpr_tag[dest_r]=my_pc;
         fpr_tag[dest_r+1]=my_pc;
      end;

   if exu == f_add_unit then
      send f_add1
   else
   if exu == f_mul_unit then
      send f_mul1
   else
      send s_EX;
end s_ID;
```

This procedure defines the epilogue of the instruction decode stage for the MIPS processor. You can see how the instruction context is forwarded to different stages depending on some of its attributes. In this example, a check is made to determine which execution unit the instruction will require. If it requires the floating point add unit, then the instruction context is forwarded to the first stage of the floating point add pipeline. If it requires the floating point multiply unit, it is forwarded to the floating point multiply pipeline. In all other cases, the instruction is sent to the execute stage of the main instruction pipeline.

Before they can be used, instruction contexts must first be allocated using the `newcontext` statement. This normally appears in the first pipeline stage. When an instruction is loaded into the instruction register, this context is filled in with its respective data.

Use of the `newcontext` statement is illustrated in the following example.

```
procedure s_IF epilogue
begin
   if send_enabled(s_ID) then
      begin
         send s_ID;

         if (branch_input) then
            begin
               branch_input=0;
               pc=branch_target;
            end
         else
            pc=new_pc;
```

```
        newcontext;
     end;
end s_IF;
```

As you can see, the `newcontext` statement appears in the epilogue minor cycle of the Instruction Fetch stage. Once the new program counter is calculated, a new instruction context is allocated, and then the Instruction Decode stage fills in the instruction context.

With the `newcontext` statement comes a corresponding `retire` statement. As you might guess, the `retire` statement is responsible for deallocating the current instruction context.

```
procedure s_WB epilogue
begin
   retire stat;
end s_WB;
```

Above is an example of how to use the `retire` statement. The Write back stage is the final stage in the MIPS pipeline, and not surprisingly the `retire` statement appears in the stage's epilogue minor cycle. The `retire` statement takes one of two arguments: `stat` or `nostat`. If the instruction needs to commit, you should use `stat`. If the instruction is to be discarded, then you can use `nostat`.

Decoding of instructions is accomplished with the `decode` statement. The following segment of ADL code is the prologue minor cycle of the Instruction Decode stage in the MIPS pipelined processor.

```
procedure s_ID prologue
begin
   decode;
end s_ID;
```

The decode statement establishes a mapping from the instruction name to the current instruction context. The mapping is computed from the binary section of the instruction declaration. The purpose of the decode instruction is to load the attributes of the instruction into the `controldata` register. Once this has been accomplished, the `controldata` variables can be accessed by the pipeline stage.

Recall the variables present in the `controldata register` for the MIPS pipelined architecture. Let's say the next instruction fetched by the Instruction Fetch stage is `SLL`, or Shift Left Logical. According to the ISA specification, the `SLL` instruction has the following attributes:

```
attributes
(
i_class   : integer_class,
        i_cycles  : single_cycle,
        exu       : integer_unit,
        c_what    : none,
        dest_type : integer_register,
        lop_type  : none,
        rop_type  : integer_register,
        i_type    : alu_type,
        dest_reg  : rd
        )
```

Once the instruction gets decoded using the `decode` statement, the attributes can be accessed as read only `controldata` variables. In the following segment of code, the left operand type is checked in order to decide on a course of action.

```
case lop_type of
   begin
      integer_register:
         lop_r=rs;
         lop=gpr[lop_r];

      lo_hi_register:
         lop_r=lo_hi_register_number;
         lop=gpr[lop_r];
         lop2=gpr[lop_r+1];
   .
   .
   .
   end;
```

This also illustrates the use of the `case` statement. Depending on the instruction's attributes, different values are loaded into the `controldata register`'s fields. The decision on how best to accomplish this is programmer dependent.

For the purposes of handling data flow, structural, control, and data hazards, ADL provides a number of other keywords that affect the behavior of the pipeline. Stages may stall themselves through the use of the `stall` statement. The `stall` statement terminates processing of that stage for the remainder of the machine cycle. As a result, no `send` operation will be executed by that pipeline stage during that machine cycle. The `stall` statement has no effect on other stages. Consider the following example:

```
stall category mem_ic, ld_d_dep, pool_full;
instruction register ir;

procedure s_ID epilogue
begin
   if i_type[EX] == load_type & (dest_r[EX] == lop_r | dest_r[EX] == rop_r) then
      stall ld_d_dep;
end s_ID;
```

In this example, the Instruction Decode stage checks whether the instruction that precedes it is fetching data that it needs. Since loads take more than one machine cycle, this amounts to a data hazard. The only way around this problem is to insert a *bubble* into the pipeline. Since the `stall` statement only stalls the pipeline stage in which it is executed, `send` operations of successive stages will continue and the `s_EX` stage will enter the next cycle in the idle state.

The `stall` statement takes one argument that corresponds to the *stall category*. Stall categories are declared in advance in the manner shown in the example. In this example, since the stall is due to a load data dependency, this category is used. Stall categories are left to programmer preference.

In addition to stalling single pipeline stages, it is also possible to `freeze` and `unfreeze` the entire pipeline. The `freeze` statement functions in a manner somewhat opposite to that of the `stall` statement. While the `stall` statement will stall only the stage in which it was executed, the `freeze` statement will stall all stages except that in which it was executed. The only stage that may execute the `unfreeze` statement is that which executed the corresponding `freeze` statement. Use of the `freeze` statement is illustrated below.

```
procedure s_IF prologue
begin
   ir = icache[pc];
   if access_complete then
   begin
      unfreeze;
      pc = pc + 4;
   end
   else
   begin
      freeze;
      stall mem_icl;
   end;
end s_IF;
```

This illustrates how the pipeline gets frozen in the event that a cache miss occurs in the Instruction Fetch stage. A `stall` is necessary after the `freeze` so that the epilogue will not attempt to execute a `send`. If `access_complete` evaluates to true in the next machine cycle, then the pipeline will be unfrozen. You will notice that `unfreeze` is always executed whenever there is a cache hit. An `unfreeze` operation on a pipeline that is not frozen is a null operation. This saves coding, as you are not required to keep track of whether or not the pipeline is frozen.

7.4 Other pipelines

ADL is capable of describing multiple pipelines. In addition to the standard MIPS pipeline, for example, you can have a pipeline for floating point addition and another for floating point multiplication. Here is the floating point add pipeline from the MIPS architecture:

```
procedure f_add1 prologue
begin
end f_add1;

procedure f_add1 epilogue
begin
   send f_add2;
end f_add1;

procedure f_add2 prologue
begin
end f_add2;

procedure f_add2 epilogue
begin
   send f_add3;
end f_add2;

procedure f_add3 prologue
begin
end f_add3;

procedure f_add3 epilogue
begin
   send s_wb;
end f_add3;
```

As you can see, the floating point add pipeline has three stages, each of which is implemented with a prologue and an epilogue. None of these functions, however, contain any LRTL statements, since they are only simulating an instruction traveling through a floating point pipeline, and the end result is the same in each case. If a floating point addition instruction is detected in the *instruction decode* pipeline stage, its instruction context is sent to `f_add1`. Once the instruction context has traveled through all the floating point pipeline's stages, it is sent to the *writeback* stage, bypassing both the *execute* and *memory* stages.

Chapter 8

The Instruction Set Architecture

8.1 Instruction formats

Any given architecture places specific requirements on the format of its machine instructions. These instructions normally consist of a number of different fields that contain opcodes, operands, and other data. In the MIPS architecture, for example, instructions consist of a single 32-bit word aligned on a word boundary. MIPS has three different instruction formats, the R, J, and I formats, which correspond to arithmetic, jump, and branch/immediate instructions, respectively.

Each of these instruction formats consist of a unique set of instruction fields. These fields vary in width, but the sum of the widths always equals 32 bits.

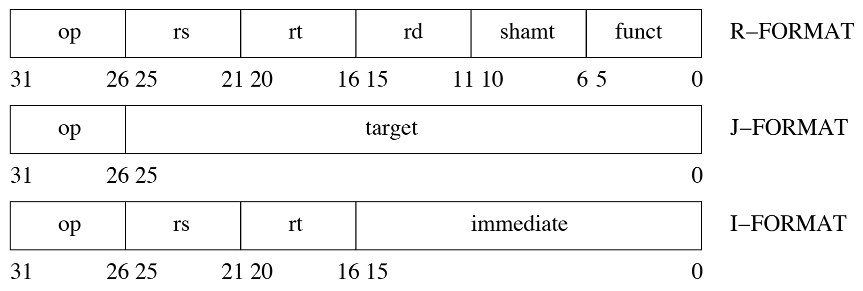

Figure 8.1: MIPS Instruction Formats

Instruction fields are defined in ADL by associating a *start bit* and *field width* pair with the name of the field. Instruction formats are not defined explicitly, but rather are defined implicitly as part of each instruction's binary part. The binary part of instructions are represented as a sequence of field expressions. This is simply an assignment of a value to a particular field of the instruction. This will be discussed in more detail later.

If we consider the instruction formats in the above diagram, the fields could be defined with the following start bits and widths:

```
op       31  6
rs       25  5
rt       20  5
rd       15  5
shamt    10  5
```

```
funct      5  6
target    25 26
immediate 15 16
```

Syntax for specifying instruction fields

Declaration of the instruction fields is accomplished via the `type` keyword, followed by a list of field declarations. Each field declaration follows the following syntax.

field-name `[constant | integer | register | signed] [field | fixedfield]` *start-bit field-width*

Constant fields are those that have a constant value for all instructions in the instruction set. This is not to say that the field is the same for all instructions, but rather that it is not determined by the assembly instruction itself. Fields that are not constant are considered to be variable fields. The values of these fields are typically determined by the assembly instruction itself.

The difference between a `field` and a `fixedfield` has to do with when the value of the field is determined. Any instruction fields which are accessed before the instruction is decoded must be declared `fixedfield`. For example, the values of the rs and rt fields from the instruction register are available immediately after the instruction register is loaded with a new instruction. You can then access the `rt` field of the instruction in the instruction register before it is decoded.

If we wish to declare the above instruction fields, the resulting code would look like this:

```
type
   op        constant field 31  6,
   rs        register field 25  5,
   rt        register field 20  5,
   rd        register field 15  5,
   shamt     integer  field 10  5,
   funct     integer  field  5  6,
   target    integer  field 25 26,
   immediate signed   field 15 16;
```

The `op` field is declared to be `constant` since for every instruction in the instruction set, it carries an opcode value that is determined ahead of time. The `rs`, `rt`, and `rd` fields are all declared to be of type `register` since they always refer to registers, whether these registers contain operand values, or are destination registers. The remaining fields are declared as `integer` fields and `signed` fields since they contain integer values. The `shamt` field contains a shift amount, while the `funct` field contains a value that represents a function type for those instructions whose opcode is 0 or 1. The `target` field contains an address for jump instructions, while the immediate field contains addresses for branch instructions or constant operand values. Since it has to contain these operand values for MIPS immediate addressing modes, it must be declared signed.

In addition to the `field` definition, you can also declare fields of type `variable` and `temporary`. These two types do not take a start bit or field width value, as they have nothing to do with the instruction formats themselves. For example, suppose you required three variables and a a temporary.

```
type
   rdest     register variable,
   rsrc1     register variable,
   rsrc2     register variable,
   tx        integer  temporary;
```

The three variables are of type register, since they will be used in macro instructions to refer to registers, while the temporary is of type integer, since it will be used to store the contents of integer registers.

8.2 Attributes

Attributes in ADL can be thought of as enumerated data types such as in C++. Any given attribute can take on one of a specific set of values declared along with the attribute name. These attributes can then be used later to keep track of instruction types, operand types, and any other necessary conditions that must be checked to properly control instruction flow.

To declare an attribute is simply a matter of specifying an attribute name followed by a comma-separated list of possible values. The list is terminated with a semicolon. Multiple attributes can be declared with the same `attribute` keyword.

An example of declaring two attributes looks like this:

```
attributes
   i_class        : float_class,
                    integer_class,
                    branch_class,
                    long_integer_class;

   i_cycles       : single_cycle,
                    multiple_cycles;
end;
```

8.3 Assertions

Assertions provide a means of checking whether certain conditions are met for each instruction loaded into the pipeline. While not necessary in an ADL program, they are useful in debugging and ensuring your code works as you had planned. An assertion consists of two boolean expressions. If the first is evaluated to true, the second expression is evaluated. If the second expression evaluates to true, the assertion is successful. If it evaluates to false, an exception is raised. If the first expression is false, then the assertion is skipped.

Let's say you wanted to make sure that single cycle instructions were always executed by the call or integer unit, and if either operand type is an integer register, it is executed with the integer, call, divide, load, or store unit. To accomplish this via assertions, you could write the following block of code:

```
assertion

   1 : i_cycles  == single_cycle      : (exu == integer_unit)     |
                                        (exu == call_unit)        ;

   2 : (lop_type == integer_register) |
       (rop_type == integer_register) : (exu == integer_unit)      |
                                        (exu == call_unit)         |
                                        (exu == divide_unit)       |
                                        (exu == load_unit)         |
                                        (exu == store_unit);
end;
```

You have already seen this example in a previous chapter. An assertion first begins with a number. You should number your assertions starting with 1. While it is possible to have two assertions with the same number, this would complicate matters during debugging if either of those assertions raised an exception.

Following the assertion number is a colon character, after which comes the first boolean condition. This is the condition the program checks to decide whether to evaluate the second expression. If the condition is true, the second expression is evaluated. The second expression comes after another colon character and is terminated with a semicolon.

Whenever an assertion fails, a line like the following is printed:

```
Assertion 1 failed : bc1t__
Some assertions(s) failed. Exiting
```

The simulator then terminates. In the case of this failure, assertion 1 failed, which means that a single cycle instruction appeared whose execution unit was neither the integer nor call unit. The simulator also tells us which instruction this was, in this case the `bc1t__` instruction.

8.4 Opcode constants

Opcode constants tell the assembler what binary form of an instruction to emit during assembly. These constants are used in the binary parts of the instruction definitions. An opcode constant consists of a name and a simple string of bits. In general the length of these opcodes should be consistent over your entire instruction set.

Declaring opcode constants begins with the `bitconstant` keyword, which is followed by a list of names and bit strings. You've already seen an example of how this is done in a previous chapter.

8.5 Instruction declarations

The instruction declarations are the heart of the ISA description and tell the assembler how to parse instructions and the simulator how to execute them. An instruction definition consists of a syntax part, binary part, and a semantic part. The syntax part tells the assembler how to parse the instruction. The binary part tells the assembler what machine code to emit, and indirectly describes the instruction formats of the architecture. The semantic part provides an implementation of the instruction with LRTL statements.

The first part of the instruction definition is the syntax part. This consists of the instruction mnemonic followed by any arguments the assembler should expect. For example, for the MIPS *shift left logical* (`sll`) instruction, this piece of code looks like `sll rd st shamt`, since these are the three fields that are used in the instruction.

The `emit` keyword signifies the beginning of the binary part. What follows tells the assembler what to emit (hence the name) during the assembly stage. As part of the bit constants, we defined the `_special` opcode. This is the opcode that is used for the `sll` instruction. Since the `rs` field is not used for `sll`, it can be filled with 0s. What follows are the `rt` and `rd` fields and then `shamt` and finally the `funct` field, which is assigned the function value for `sll`. The resulting binary part looks like: `emit opcode=_special rs=0 rt rd shamt funct=_sll`.

The binary parts of the instruction definitions are what define the instruction formats for the architecture in question.

The third part of the instruction declaration is semantic part. Here is where the instruction's attributes are set, and then the implementation of the instruction is provided. The attributes component looks like:

```
attributes
   (
    attribute1 : value,
    attribute2 : value,
    .
    .
    .
    attributeN : value
   )
```

Following the attributes come the actual LRTL statements that implement the instruction. Here you specify the actions to be taken in each pipeline stage the instruction passes through. You may specify LRTL statements for any pipeline stage you wish. The format looks like this:

```
begin
   exact STAGE_1
      LRTL statements...
   end;

   exact STAGE_2
      LRTL statements...
   end;

   .
   .
   .
end;
```

The `begin` and `end` keywords simply signify the beginning and end of the semantic block. The `exact` keyword takes one argument corresponding to the pipeline stage that you are specifying operations for. Each `exact` keyword also has a corresponding `end` keyword.

Let us now look at an example of all these pieces put together. The following piece of code is the instruction declaration for the MIPS `div` (divide word) instruction.

```
div rs rt
   emit opcode=_special rs rt rd=0 shamt=0 funct=_div
   attributes
   (
    i_class   : long_integer_class,
    i_cycles  : multiple_cycles,
    exu       : integer_unit,
    c_what    : none,
    dest_type : lo_hi_register,
    lop_type  : integer_register,
    rop_type  : integer_register,
    i_type    : alu_type,
    dest_reg  : lo_hi_register_number
   )
   begin
      exact s_ID
         latency 7;
      end;
      exact s_EX
         if rop == 0 then
            exception = 1
         else
            begin
               dest2 = lop / rop;
               dest  = lop % rop;
            end;
      end;
   end,
```

The `div` instruction takes two integer register arguments. The contents of the first register are divided by the contents of the second and the result is stored in the first register. In the binary part of the instruction declaration, the `_special` opcode is used with the `_div` function. The `rd` and `shamt` instruction fields both get filled with 0s. The attributes section sets the necessary attributes for the instruction. Since division is a multiple-cycle operation, that must be specified. Since there are two destination registers in division, the `dest_type` attribute is `lo_hi_register`. In addition, the `div` instruction only takes integer registers as arguments. For floating point division, a separate instruction exists.

For many MIPS instructions, it suffices to specify LRTL statements for only one pipeline stage. The `div` instruction, among a few others, however, requires a little more. The `latency` keyword takes an argument corresponding to the number of cycles that the instruction requires to return its result. This is specified in the `s_ID` (instruction decode) stage, since this is where the machine determines the type of instruction it is dealing with.

The `s_EX` (instruction execute) stage is where the real work gets done, and this is where the majority of the LRTL statements are found. At this point you may be wondering how the `rs` and `rd` registers translate into `lop` and `rop`, respectively. This is the job of the Instruction Decode stage. If you look at the following excerpt from `s_ID`, you can see how `lop is calculated`:

```
case lop_type of
   begin
      integer_register :
         lop_r=rs;
         lop=gpr[lop_r];
   .
   .
   .
```

Since the left operand type for the divide instruction is of type integer register, the machine looks to the integer register file for the value of `lop`.

The same can be seen for `rop`:

```
case rop_type of
   begin
      integer_register :
         rop_r=rt;
         rop=gpr[rop_r];
   .
   .
   .
```

Likewise, the type of the left operand register determines where the machine goes to look for the value of the left operand. Once the values of the operands are known, they can be used to perform the required computations. The instruction now moves on to the execution stage.

The first thing that must be checked in `s_EX` is whether the right operand has a value of zero. Since this is a division operation, dividing by zero will raise an exception. If this is not the case, then the actual calculation proceeds. `dest2` receives the value of the quotient, while `dest` gets the value of the remainder. The instruction now moves on through the `s_MEM` stage into the `s_WB` stage, where the values of `dest` and `dest2` get written back to the integer register file. Note that there are no explicit LRTL statements needed in the instruction declaration to accomplish this. This is handled automatically by the Writeback stage for all instructions.

```
case dest_type of
   begin
      lo_hi_register   :
         gpr[dest_r]=dest;
         gpr[dest_r+1]=dest2;
   .
   .
   .
```

Recall that the destination register type for the divide word instruction is `lo_hi_register`. As a result, two registers must be written to.

I want an exact description of how the destination address is computed here!!!!!!!

Let us now look at a different instruction, the `lw` or load word instruction. This instruction demonstrates some additional options available for the binary part of instruction declarations.

```
lw rt address
   emit opcode=_lw rs=<address.base> rt immediate=<address.offset>
   attributes
   (
    i_class   : integer_class,
    i_cycles  : multiple_cycles,
    exu       : load_unit,
    c_what    : none,
    dest_type : integer_register,
    lop_type  : integer_register,
    rop_type  : none,
    i_type    : load_type,
    dest_reg  : rt
   )
   begin
      exact s_ID
         simm=sign_extend_16(immediate);
      end;

      exact s_EX
         lmar=lop + simm;
      end;

      exact s_MEM
```

```
        if ls_bypass then
           dest=dest2
        else
           begin
              dest=dcache [lmar];
              mem_stat=access_complete;
           end;
     end;

  end,
```

The first thing you should notice is that the binary part looks a little bit different from the other examples you've seen so far. Recall that the binary part takes the form of a sequence of field expressions. These field expressions can be made up of a single field, a constant assigned to a field, or a field transformed by a built-in function or *pure function*. Pure functions are functions that take a single argument and return a transformation of this argument.

Thus, the sequence of *field expressions* that make up the binary part of an instruction definition have the following format:

field | *field* = *constant* | *field* = <*fog-list*>

where a *fog-list* is a sequence of predeclared or pure function calls on the field. The predeclared functions that are available are: `base`, `offset`, `absolute`, `delta`, and `segoffset`.

In the example above, once the assembler parses the `lw` instruction in the assembly file, it will emit the opcode, followed by the base register number corresponding to the address, then the `rt` register number, followed by the offset representing the address.

ADL does not limit you to performing a single transformation on a label variable, and as such you can combine functions. For example, suppose you wish to use the `delta` built in function and transform its output with your own `jump_address` function, assigning the result to the `b_offset` label variable. You can use the following code:

```
b_offset=<address.delta.jump_address>;
```

8.6 Macro instructions

ADL provides a means of declaring macro instructions, often called pseudo instructions. Such instructions are common in many compilers today such as `gcc`. Macro instructions normally break down into several machine instructions. This is the task of the assembler, and thus allows a slightly higher level of abstraction for the programmer and simplifies the assembly code.

The syntax part of macro instructions looks the same as that for machine instructions. The difference is that all the arguments to the instruction must be variables. No instruction fields are allowed. Additionally, since macro instructions do not lead directly to binary code, there is no binary part in the macro definitions.

Let's look at a very simple macro instruction.

```
sra rdest rsrc1 rsrc2 macro
  begin
     srav : rd=rdest rt=rsrc1 rs=rsrc2;
  end,
```

The syntax part of this macro instruction tells the assembler that whenever it sees the `sra` mnemonic that it should look for three values after it and load those into the variables `rdest`, `rsrc1` and `rsrc2`. These variables have been defined in advance and are of the type you saw when instruction fields were discussed.

The remainder of the definition describes what instructions to generate. Instructions to be generated are specified using an *instruction call* statement. The syntax for this looks like:

instruction-call = instruction-mnemonic : field-assignment-list
field-assignment-list = field=constant | variable

In the above example, only a single instruction is generated, that being the `srav` instruction. This instruction takes three arguments, and those are assigned the values in `rdest`, `rsrc1` and `rsrc2`, respectively.

Of course the point of macro instructions is to reduce many machine instructions to a single pseudo instruction. In addition, a different set of instructions can be generated based on the values of the arguments. Let's take a look at a more complex example.

```
li rdest src2 macro
   begin
      if (src2 > 0) | (src2 == 0) then
         begin
            tx=src2;
            if tx < 65536 then
               ori : rt=rdest rs=0 immediate=src2
            else
               begin
                  ty = src2.[31:16];
                  tx = src2.[15:16];
                  lui : rt=rdest immediate=ty;
                  if tx then
                  ori : rt=rdest rs=rdest immediate=tx;
               end;
         end
      else
         if (src2 == 0x80000000) then
            begin
               ty = src2.[31:16];
               lui : rt=rdest immediate=ty;
            end
         else
            begin
               tx=-src2;
               if (tx < 32768) then
                  addiu : rt=rdest rs=0 immediate=src2
               else
                  begin
                     ty = src2.[31:16];
                     tx = src2.[15:16];
                     lui : rt=rdest immediate=ty;
                     if tx then
                        ori : rt=rdest rs=rdest immediate=tx;
                  end;
```

```
          end;
end,
```

In this example, the value of the variable `src2` is the deciding factor in which instructions are generated.

Chapter 9

Statistics Collection and Debugging

Collection of statistics reported in the statistics files is facilitated through a number of ADL constructs for this purpose. Instructions can be categorized and this categorization is reflected in the statistics files.

Suppose you wanted to categorize some instructions in an `integer_arithmetic` category. Here's how you could do this:

```
instruction category integer_arithmetic
   add,
   addi,
   addiu,
   addu;
```

Any use of these instructions will now be reported under an `integer_arithmetic` heading in the statistics files.

As mentioned before, stalls can also be classified into categories to aid in performance evaluation and debugging. The `stall` keyword takes an optional category argument, and then stalls are classified into these categories in the statistics files. Declaring stall categories involves the `stall category` keywords followed by a list of category names. For example:

```
stall category
     ext_ref,
     latency_f,       # Floating point latency.
     latency_d,       # Divide latency.
     latency_m,       # Multiply latency.
     fpaddfull,       # Floating add pipeline is full with long latency op.
     fpmulfull,       # Floating multiply pipeline is full with lo la op.
     mem_ic,
     mem_dc,
     fl_d_dep,
     ld_d_dep,
     float_cc;
```

When you need to stall the pipeline, you can choose to categorize it or leave it uncategorized. For example, if you need to stall due to floating point latency, issue the following command:

```
stall latency_f;
```

ADL provides two statements that are used specifically for the debugging purposes. The `pause` statement can be inserted anywhere among the LRTL segments and causes the simulator to start the debugger at the point where that statement is reached. The registers to be displayed in the debugger window are specified with the `monitor` statement.

Suppose you wanted to display the the first six integer and floating point registers along with the program counter and instruction register in the debugging window. This can be accomplished by placing the following code in your ADL program:

```
monitor
   $0, $1, $2, $3, $4, $5,
   linebreak,
   linebreak,
   $f0, $f1, $f2, $f3, $f4, $f5,
   linebreak,
   linebreak,
   pc,
   ir;
```

All that is needed is to specify a comma separated list of registers to monitor. The `linebreak` keyword simply inserts a blank line in the debugging window to aid in readability.

In addition to the `pause` statement, the debugger can be entered automatically using a the `--monitor` option on the command line. The simulator will also automatically invoke the debugger in the event of a *deadlock*. A deadlock occurs when an instruction is not retired for a large number of cycles. The debugger consists of two windows. The first window displays a memory image consisting of the line number of the program, followed by the memory location, binary encoding of the instruction, machine instructions, and the corresponding line from the assembly language program. The second window displays those registers that were specified with the `monitor` keyword, along with the contents of each pipeline stage. Finally, some other data, such as the current machine cycle and number of stall cycles are displayed.

To single step through the program execution, you can use the arrow keys and space bar with the cursor positioned in the second window. The arrow keys cause the program to step in major cycles (ie: from prologue to prologue), while the space bar will step in minor cycles (ie: prologue to intermission to epilogue to prologue).

Appendix A

Compilation and run-time issues

A.1 Compilation issues

The xgcc compiler

C programs that require use of `stdarg.h` or `varargs.h` will experience compilation errors with `xgcc`. The type `__builtin_va_list` is built in to `gcc` version 2.96 and higher, and as such, is undefined to `xgcc`.

The solution is to use the following argument to xgcc:

```
-D__builtin_va_list=void*
```

The ADL compiler

The ADL compiler appears to have a bug when it comes to user-defined artifacts. The generated C++ code in `<machine_name>.simulator.c` fails to call the `Cause_initialization()` function, which is responsible for calling the `initialization` functions of any user-defined artifacts that you declare. The result is that any user-defined artifacts that require initialization may exhibit unpredictable behaviour. The solution is to manually edit the `<machine_name>.simulator.c` file, inserting the following line of code:

```
Cause_initialization();
```

This statement should be inserted in the `boot_up` function after all calls to the `You_are` function. For example, given user-defined artifact declarations `tbuf` and `bp`, you should change the `boot_up` function to look like:

```
inline void procedure boot_up(integer Fast_stage_id)
begin
   pipestage pipeline_register=Fast_stage_indirect[Fast_stage_id];
   tbuf.You_are("tbuf", 0, 256);
   bp.You_are("bp", 0, 256);

   Cause_initialization();  /* Add this line */

   set_register_file ( gpr, 0, 34 );
   set_register_file ( fpr_tag, 0, 33 );
end // boot_up //
```

Running the simulator

Executing the simulator will issue the following errors. They can safely be ignored.

```
Not Found etext:#global.ulimit.etext text @ (nil) offset =0 label_type =5 segment_type =0
start_bit =15 nbits =16 patch_address=0x2acaa9fc which =3 fog =4
Not Found etext:#global.ulimit.etext text @ (nil) offset =0 label_type =5 segment_type =0
start_bit =15 nbits =16 patch_address=0x2acaaa00 which =3 fog =5
```

Bibliography

[1] Gerry Kane. MIPS RISC architecture. Prentice-Hall, Inc, Englewood Cliffs, NJ 07632, 1989.

[2] Soner Onder. Architecture description language user's manual. Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931-1295, 2001.

[3] Soner Onder and Rajiv Gupta. Automatic generation of microarchitecture simulators. IEEE International Conference on Computer Languages (ICCL98), Chicago, May, 1998.

[4] John Waldron. Introduction to RISC assembly language programming. Addison-Wesley Longman Ltd, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 1999.