DLXsim – A Simulator for DLX

Larry B. Hostetler Brian Mirtich

February 3, 1992

1 Introduction

Our project involved writing a simulator (**DLXsim**) for the DLX instruction set (as described in *Computer Architecture, A Quantitative Approach* by Hennessy and Patterson). **DLXsim** is an interactive program that loads DLX assembly programs and simulates the operation of a DLX computer on those programs, allowing both single-stepping and continuous execution through the DLX code. **DLXsim** also provides the user with commands to set breakpoints, view and modify memory and registers, and print statistics on the execution of the program allowing the user to collect various information on the run-time properties of a program. We expect that a major use for this tool will be in association with future CS 252 classes to aid in the understanding of this instruction set.

A complete overview of the interface provided by the simulator can be found in the user manual for **DLXsim**, which has been included after this section. Later in this paper, a few sample runs of the simulator will also be given.

We decided that since the MIPS instruction set has many similarities with DLX, and a good MIPS simulator (available from Ousterhaut) already exists, it would be a better use of our time to modify that simulator to handle the DLX description. This simulator was built on top of the Tcl interface, providing a programming type environment for the user as well.

The main problem we encountered when rewriting the simulator was that there are a couple of fundamental differences between the DLX and MIPS architectures. Following is a list of the main differences we identified between the two architectures.

- In MIPS, branch and jump offsets are stored as the number of words, where DLX stores the number of bytes. This has the effect of allowing jumps on MIPS to go four times as far.
- MIPS jumps have a non-obvious approach to determining the destination address: the bits in the offset part of the instruction simply replace the lower bits in the program counter. DLX chooses a more conventional approach in that the offset is sign extended, and then added to the program counter.
- In the MIPS architecture, conditional branches are based on the result of a comparison between any two registers. DLX has only two main conditional branch operations which branch on whether a register is zero or non-zero.
- DLX provides load interlocks, while the MIPS 2000 does not.
- MIPS 2000 provides instructions for unaligned accesses to memory, while DLX does not.

- The result of a MIPS multiply or divide ends up in two special registers (HI and LO) allowing 64 bit results; the result of a DLX multiply is placed in the chosen general purpose register, and must therefore fit into 32 bits.

Because of the large number of similarities between DLX and MIPS, we based our opcodes on those used by the MIPS machine (where MIPS had equivalent instructions). Where DLX had instructions with no MIPS equivalent, we grouped such similar DLX instructions and assigned to them blocks of unused opcodes. Below, you will find the opcode numbers used for the DLX instructions. Register-register instructions have the **special** opcode, and the instruction is specified in the lower six bits of the instruction word. Similarly, floating point instructions have the **fparith** opcode, and the actual instruction is again found in the lower six bits of the word.

Main opcodes

	\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07
\$00	SPECIAL	FPARITH	J	JAL	BEQZ	BNEZ	BFPT	BFPF
\$08	ADDI	ADDUI	SUBI	SUBUI	ANDI	ORI	XORI	LHI
\$10	RFE	TRAP	JR	JALR				
\$18	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI		
\$20	LB	LH		LW	LBU	LHU	LF	LD
\$28	SB	SH		SW			SF	SD
\$30	SEQUI	SNEUI	SLTUI	SGTUI	SLEUI	SGEUI		

Special opcodes (Main opcode = \$00)

	\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07
\$00	SLLI		SRLI	SRAI	SLL		SRL	SRA
\$08					TRAP			
\$10	SEQU	SNEU	SLTU	SGTU	SLEU	SGEU		
\$18	MULT	MULTU	DIV	DIVU				
\$20	ADD	ADDU	SUB	SUBU	AND	OR	XOR	
\$28	SEQ	SNE	SLT	SGT	SLE	SGE		
\$30	MOVI2S	MOVS2I	MOVF	MOVD	MOVFP2I	MOVI2FP		

Floating Point opcodes (Main opcode = \$01)

	\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07
\$00	ADDF	SUBF	MULTF	DIVF	ADDD	SUBD	MULTD	DIVD
\$08	CVTF2D	CVTF2I	CVTD2F	CVTD2I	CVTI2F	CVTI2D		
\$10	EQF	NEF	LTF	GTF	LEF	GEF		
\$18	EQD	NED	LTD	GTD	LED	GED		

The manual entry for **DLXsim** follows.

NAME

DLXsim - Simulator and debugger for DLX assembly programs

SYNOPSIS

dlxsim

OPTIONS

[-al#] [-au#] [-dl#] [-du#] [-ml#] [-mu#]

-al# Select the latency for a floating point add (in clocks).

-au# Select the number of floating point add units.

-dl# Select the latency for a floating point divide.

-du# Select the number of floating point divide units.

-ml# Select the latency for a floating point multiply.

-mu# Select the number of floating point multiply units.

DESCRIPTION

DLXsim is an interactive program that loads DLX assembly programs and simulates the operation of a DLX computer on those programs. When **DLXsim** starts up, it looks for a file named **.dlxsim** in the user's home directory. If such a file exists, **DLXsim** reads it and processes it as a command file. **DLXsim** also checks for a **.dlxsim** file in the current directory, and executes the commands in it if the file exists. Finally, **DLXsim** loops forever reading commands from standard input and printing results on standard output.

NUMBERS

Whenever **DLXsim** reads a number, it will accept the number in either decimal notation, hexadecimal notation if the first two characters of the number are **0x** (e.g. 0x3acf), or octal notation if the first character is **0** (e.g. 0342). Two **DLXsim** commands accept only floating pointer numbers from the user; these are **fget** and **fput** and will be described later.

ADDRESS EXPRESSIONS

Many of **DLXsim**'s commands take as input an expression identifying a register or memory location. Such values are indicated with the term *address* in the command descriptions below. Where register names are acceptable, any of the names **r0** through **r31** and **f0** through **f31** may be used. The names **$0** through **$31** may also be used (instead of **r0** through **r31**), but the dollar signs are likely to cause confusion with Tcl variables, so it is safer to use **r** instead of **$**. The name **pc** may be used to refer to the program counter.

Symbolic expressions may be used to specify memory addresses. The simplest form of such an expression is a number, which is interpreted as a memory address. More generally, address expressions may consist of numbers, symbols (which must be defined in the assembly files currently loaded), the operators $*$, /, %, +, $-$, <<, >>, &, |, and $\uparrow$ (which have the same meanings and precedences as in C), and parentheses for grouping.

COMMANDS

In addition to all of the built-in Tcl commands, **DLXsim** provides the following application-specific commands:

asm *instruction* [*address*]

Treats *instruction* as an assembly instruction and returns a hexadecimal value equivalent to *instruction*. Some instructions, such as relative branches, will be assembled differently depending on where in memory the instruction will be stored. The *address* argument may be used to indicate where the instruction would be stored; if omitted, it defaults to 0.

fget *address* [*flags*]

Return the values of one or more memory locations or registers. *Address* identifies a memory location or register, and *flags*, if present, consists of a number and/or set of letters, all concatenated together. If the number is present, it indicates how many consecutive values to print (the default is 1). If flag characters are present, they have the following interpretation:

d Print values as double precision floating point numbers.

f Print values as single precision floating point numbers (default).

fput *address number* [*precision*]

Store *number* in the register or memory location given by *address*. If *precision* is **d**, the number is stored as a double precision floating point number (in two words). If *precision* is **f** or no *precision* is given, the number is stored as a single precision floating point number.

get *address* [*flags*]

Similar to **fget** above, this command is for all types except floating point. If flag characters are present, they have the following interpretation:

B Print values in binary.

b When printing memory locations, treat each byte as a separate value.

c Print values as ASCII characters.

d Print values in decimal.

h When printing memory locations, treat each halfword as a separate value.

i Print values as instructions in the DLX assembly language.

s Print values as null-terminated ASCII strings.

v Instead of printing the value of the memory location referred to by *address*, print the address itself as the value.

w When printing memory locations, treat each word as a separate value.

x Print values in hexadecimal (default).

To interpret numbers as single or double precision floating point, use the **fget** command.

go [*address*]
Start simulating the DLX machine. If *address* is given, execution starts at that memory address. Otherwise, it continues from wherever it left off previously. This command does not complete until simulated execution stops. The return value is an information string about why execution stopped and the current state of the machine.

load *file file file* ...
Read each of the given *files*. Treat them as DLX assembly language files and load memory as indicated in the files. Code (text) is normally loaded starting at address 0x100, but the **codeStart** variable may be used to set a different starting address. Data is normally loaded starting at address 0x1000, but a different starting address may be specified in the **dataStart** variable. The return value is either an empty string or an error message describing problems in reading the files. A list of directives that the loader understands is in a later section of this manual.

put *address number*
Store *number* in the register or memory location given by *address*. The return value is an empty string. To store floating point numbers (single or double precision), use the **fput** command.

quit Exit the simulator.

stats [**reset**] [**stalls**] [**opcount**] [**pending**] [**branch**] [**hw**] [**all**]
This command will dump various statistics collected by the simulator on the DLX code that has been run so far. Any combination of options may be selected. The options and their results are as follows:

reset Reset all of the statistics.

stalls Show the number of load stalls and stalls while waiting for a floating point unit to become available or for the result of a previous operation to become available.

opcount Show the number of each operation that has been executed.

pending Show all floating point operations currently being handled by the floating point units as well as what their results will be and where they will be stored.

branch Show the percentage of branches taken and not-taken.

hw Show the current hardware setup for the simulated machine.

all Equivalent to choosing all options except **reset**. This is the default.

step [*address*]
If no *address* is given, the **step** command executes a single instruction, continuing

from wherever execution previously stopped. If *address* is given, then the program counter is changed to point to *address*, and a single instruction is executed from there. In either case, the return value is an information string about the state of the machine after the single instruction has been executed.

stop [*option args*]

This command may take any of the forms described below:

stop Arrange for execution of DLX code to stop as soon as possible. If a simulation isn't in progress then this command has no effect. This command is most often used in the *command* argument for the **stop at** command. Returns an empty string.

stop at *address* [*command*]

Arrange for *command* (a **DLXsim** command string) to be executed whenever the memory address identified by *address* is read, written, or executed. If *command* is not given, it defaults to **stop**, so that execution stops whenever *address* is accessed. A stop applies to the entire word containing *address*: the stop will be triggered whenever any byte of the word is accessed. Stops are not processed during the **step** commands or the first instruction executed in a **go** command. Returns an empty string.

stop info

Return information about all stops currently set.

stop delete *number number number ...*

Delete each of the stops identified by the *number* arguments. Each *number* should be an identifying number for a stop, as printed by **stop info**. Returns an empty string.

ASSEMBLY FILE FORMAT

The assembler built into **DLXsim**, invoked using the **load** command, accepts standard format DLX assembly language programs. The file is expected to contain lines of the following form:

- Labels are defined by a group of non-blank characters starting with either a letter, an underscore, or a dollar sign, and followed immediately by a colon. They are associated with the next address to which code in the file will be stored. Labels can be accessed anywhere else within that file, and in files loaded after that if the label is declared as **.global** (see below).
- Comments are started with a semicolon, and continue to the end of the line.
- Constants can be entered either with or without a preceding number sign.
- The format of instructions and their operands are as shown in the Computer Architecture book.

While the assembler is processing an assembly file, the data and instructions it assembles are placed in memory based on either a text (code) or data pointer. Which pointer is used is selected not by the type of information, but by whether the most recent directive was **.data** or **.text**. The program initially loads into the text segment.

The assembler supports several directives which affect how it loads the DLX's memory. These should be entered in the place where you would normally place the instruction and its arguments. The directives currently supported by **DLXsim** are:

.align *n* Cause the next data/code loaded to be at the next higher address with the lower *n* bits zeroed (the next closest address greater than or equal to the current address that is a multiple of 2^{n-1}).

.ascii "*string1*", "*string2*", ...
: Store the *strings* listed on the line in memory as a list of characters. The strings are not terminated by a 0 byte.

.asciiz "*string1*", "*string2*", ...
: Similar to **.ascii**, except each string is followed by a 0 byte (like C strings).

.byte "*byte1*", "*byte2*", ...
: Store the *bytes* listed on the line sequentially in memory.

.data [*address*]
: Cause the following code and data to be stored in the data area. If an *address* was supplied, the data will be loaded starting at that address, otherwise, the last value for the data pointer will be used. If we were just reading code based on the text (code) pointer, store that address so that we can continue from there later (on a **.text** directive).

.double *number1*, *number2*, ...
: Store the *numbers* listed on the line sequentially in memory as double precision floating point numbers.

.float *number1*, *number2*, ...
: Store the *numbers* listed on the line sequentially in memory as single precision floating point numbers.

.global *label*
: Make the *label* available for reference by code found in files loaded after this file.

.space *size*
: Move the current storage pointer forward *size* bytes (to leave some empty space in memory).

.text [*address*]
: Cause the following code and data to be stored in the text (code) area. If an *address* was supplied, the data will be loaded starting at that address, otherwise, the last value for the text pointer will be used. If we were just reading data based on the data pointer, store that address so that we can continue from there later (on a **.data** directive).

.word *word1*, *word2*, ...
  Store the *words* listed on the line sequentially in memory.

VARIABLES

DLXsim uses or sets the following Tcl variables:

codeStart
  If this variable exists, it indicates where to start loading code in **load** commands.

dataStart
  If this variable exists, it indicates where to start loading data in **load** commands.

insCount
  DLXsim uses this variable to keep a running count of the total number of instructions that have been simulated so far.

prompt If this variable exists, it should contain a **DLXsim** command string. **DLXsim** will execute the command in this string before printing each prompt, and use the result as the prompt string to print. If this variable doesn't exist, or if an error occurs in executing its contents, then the prompt "(dlxsim)" is used.

SEE ALSO

Computer Architecture, A Quantitative Approach, by John L. Hennessy and David A. Patterson.

KEYWORDS

DLX, debug, simulate

2 Interactive Sessions with DLXsim

To illustrate some of the features of **DLXsim**, this section describes two interactive sessions using examples taken from Chapter 6 of *Computer Architecture, A Quantitative Approach* by Hennessy and Patterson. The programs used are on page 315 and 317. The ADDD instructions have been replaced with MULTD instructions, however, to show the effects of a slightly longer latency. Also, TRAP instructions have been added to terminate execution of the programs when simulating.

2.1 Sample Datafile

The examples which follow operate on arrays of numbers. A common datafile is used for input to the programs. This datafile is named **fdata.s** and is shown below:

```
        .data     0
        .global   a
a:      .double   3.14159265358979
        .global   x
x:      .double   1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
        .double   17,18,19,20,21,22,23,24,25,26,27
        .global   xtop
xtop:   .double   28
```

The **.data** directive specifies that the data should be loaded in at location 0. The **.global** directive add the specified labels to a global symbol table so that other assembly files can access them. The **.double** directive stores double precision data to memory.

2.2 First Example

The first example uses the program at the bottom of page 315 (with the ADDD replaced by MULTD). The program is shown below.

```
        ld      f2,a
        add     r1,r0,xtop
loop:   ld      f0,0(r1)
                            ; load stall occurs here
        multd   f4,f0,f2
                            ; 4 FP stalls
        sd      0(r1),f4
        sub     r1,r1,#8
        bnez    r1,loop
        nop                 ; branch delay slot
        trap    #0          ; terminate simulation
```

The simulator is invoked by typing **dlxsim** at the system prompt.

```
% dlxsim
```

First the datafile is loaded, using the load command:

(dlxsim) `load fdata.s`

Next, the program may be loaded. The program above was created with an editor and saved in the file **f1.s**. It is loaded in the same way as the datafile.

(dlxsim) `load f1.s`

To verify that the program has been loaded, the **get** command can be used to examine memory. The program is loaded at location 256 by default. The second parameter to **get** indicates how many words to dump. The **i** suffix tells **get** to dump the contents in instruction format (i.e. produce a disassembly).

(dlxsim) `get 256 9i`

```
start: ld f2,a(r0)
start+0x4: addi r1,r0,0xe0
loop: ld f0,a(r1)
loop+0x4: multd f4,f0,f2
loop+0x8: sd a(r1),f4
loop+0xc: subi r1,r1,0x8
loop+0x10: bnez r1,loop
loop+0x14: nop
loop+0x18: trap 0x0
```

To make sure that the statistics are all cleared (as they should be when **DLXsim** is first invoked), use the **stats** command with the relevant parameters:

(dlxsim) `stats stalls branch pending hw`

```
Memory size: 65536 bytes.

Floating Point Hardware Configuration
 1 add/subtract units, latency =  2 cycles
 1 divide units,       latency = 19 cycles
 1 multiply units,     latency =  5 cycles
Load Stalls = 0
Floating Point Stalls = 0

No branch instructions executed.

Pending Floating Point Operations:
none.
```

The **hw** specifier causes the memory size and floating point hardware information to be dumped. The **stalls** specifier causes the total load stalls and floating point stalls to be displayed. The **branch** specifier causes the branch information (taken vs. not taken) to be displayed; in this case no branches have been executed yet. Finally, the **pending** specifier causes the pending operations in the floating point units to be displayed (none in this case). Below, the first four instructions are executed using the **step** command:

(dlxsim) `step 256`

```
stopped after single step, pc = start+0x4: addi r1,r0,0xe0
```

(dlxsim) `step`

```
stopped after single step, pc = loop: ld f0,a(r1)
```

(dlxsim) `step`

```
stopped after single step, pc = loop+0x4: multd f4,f0,f2
```

(dlxsim) `step`

```
stopped after single step, pc = loop+0x8: sd a(r1),f4
```

The **stats** command can produce some more interesting results at this point.

(dlxsim) `stats stalls pending`

```
Load Stalls = 1
Floating Point Stalls = 0

Pending Floating Point Operations:
multiplier   #1 :  will complete in  4 more cycle(s)  87.964594 ==> F4:F5
```

A load stall occurred between the third and fourth instructions because of the F0 dependency. The multiply instruction has issued, and is being processed in multiplier unit #1. It will complete and store the double precision value 87.96 into F4 and F5 in four more clock cycles.

The double precision value in F4 can be displayed using the **fget** command with a **d** specifier (for double precision).

(dlxsim) `fget f4 d`

```
f4: 0.000000
```

As expected, F4 hasn't received its value yet. Executing one more instruction will change the statistics:

(dlxsim) `step`

```
stopped after single step, pc = loop+0xc: subi r1,r1,0x8
```

(dlxsim) `stats stalls pending`

```
Load Stalls = 1
Floating Point Stalls = 4

Pending Floating Point Operations:
none.
```

Since the SD instruction used the result from the multiply instruction, the multiply was completed before the SD was executed. The four floating point stalls required for the multiply to complete were recorded as well. If F4 is examined now, its value after the writeback is displayed.

(dlxsim) `fget f4 d`

```
f4: 87.964594
```

To execute the program to completion, the **go** command can be used. When the TRAP instruction is detected, the simulation will stop.

(dlxsim) `go`

```
TRAP #0 received
```

To view the cumulative stall and branch information, the **stats** command can be used.

(dlxsim) `stats stalls branch`

```
Load Stalls = 28
Floating Point Stalls = 112

Branches:  total 28, taken 27 (96.43%), untaken 1 (3.57%)
```

The loop executed 28 times. There was a single load stall per iteration, for a total of 28 load stalls. There were 4 floating point stalls per iteration, for a total of 112 floating point stalls. Finally, the conditional branch at the bottom of the loop was taken 27 times, and fell through on the final time. All these statistics are reflected above.

To verify the program operated properly, the memory locations containing the original data can be examined with the **fget** command. The original data was stored in the 28 double words beginning at location 8.

(dlxsim) `fget 8 28d`

```
x: 3.141593
x+0x8: 6.283185
x+0x10: 9.424778

... etc. ...

x+0xc8: 81.681409
x+0xd0: 84.823002
xtop: 87.964594
```

As expected, the initial integer values have all been multiplied by π.

2.3 Second Example

The second example is from page 317 of the aforementioned text. It demonstrates the effects of unrolling loops when multiple execution units are available. The program, which is shown below, performs the same operations on the list of numbers as the previous example program.

```
start:  ld      f2,a
        add     r1,r0,xtop
loop:   ld      f0,0(r1)
        ld      f6,-8(r1)
        ld      f10,-16(r1)
        ld      f14,-24(r1)
        multd   f4,f0,f2
        multd   f8,f6,f2
        multd   f12,f10,f2
        multd   f16,f14,f2
                              ; FP stall here
        sd      0(r1),f4
        sd      -8(r1),f8
        sd      -16(r1),f12
        sub     r1,r1,#32
        bnez    r1,loop
        sd      8(r1),f16     ; branch delay slot
        trap    #0
```

To take full advantage of this unwound loop, **DLXsim** can be invoked with a command line argument specifying 4 floating point multiply units should be included in the hardware configuration.

```
% dlxsim -mu4

(dlxsim) stats hw

Memory size: 65536 bytes.

Floating Point Hardware Configuration
 1 add/subtract units, latency =  2 cycles
 1 divide units,       latency = 19 cycles
 4 multiply units,     latency =  5 cycles
```

After loading the data and program files, the **step** instruction can be used to execute the first 10 instructions. At this point, the last MULTD instruction has just issued. The **stats** command can display the stalls and pending operations.

```
(dlxsim) stats stalls pending

Load Stalls = 0
Floating Point Stalls = 0
```

```
Pending Floating Point Operations:
multiplier   #0 :  will complete in  1 more cycle(s)  87.964594 ==> F4:F5
multiplier   #1 :  will complete in  2 more cycle(s)  84.823002 ==> F8:F9
multiplier   #2 :  will complete in  3 more cycle(s)  81.681409 ==> F12:F13
multiplier   #3 :  will complete in  4 more cycle(s)  78.539816 ==> F16:F17
```

It is intersting to see what happens after the next instruction is executed.

(dlxsim) `step`

```
stopped after single step, pc = loop+0x24: sd 0xfff8(r1),f8
```

(dlxsim) `stats stalls pending`

```
Load Stalls = 0
Floating Point Stalls = 1

Pending Floating Point Operations:
multiplier   #2 :  will complete in  1 more cycle(s)  81.681409 ==> F12:F13
multiplier   #3 :  will complete in  2 more cycle(s)  78.539816 ==> F16:F17
```

Since the SD instruction was dependent on the first MULTD instruction, a floating point stall occurred so the MULTD could complete. This added stall cycle also caused the second MULTD to also complete. The MULTDs have "caught up" with the SDs, and no more stalls will occur on this iteration. This is the reason loop unrolling works. To run the program to completion, the **go** command can be used.

(dlxsim) `go`

```
TRAP #0 received
```

To dump all the statistics gathered, the **stats** command is used without any parameters.
(dlxsim) `stats`

```
Memory size: 65536 bytes.

Floating Point Hardware Configuration
 1 add/subtract units, latency =  2 cycles
 1 divide units,       latency = 19 cycles
 4 multiply units,     latency =  5 cycles
Load Stalls = 0
Floating Point Stalls = 7

Branches:  total 7, taken 6 (85.71%), untaken 1 (14.29%)

Pending Floating Point Operations:
none.
INTEGER OPERATIONS
==================
```

```
     ADD        0      ADDI         1      ADDU         0     ADDUI         0
     AND        0      ANDI         0      BEQZ         0      BFPF         0
    BFPT        0      BNEZ         7       DIV         0      DIVU         0
       J        0       JAL         0      JALR         0        JR         0
      LB        0       LBU         0        LD        29        LF         0
      LH        0       LHI         0       LHU         0        LW         0
    MOVD        0      MOVF         0   MOVFP2I         0   MOVI2FP         0
  MOVI2S        0   MOVS2I          0      MULT         0     MULTU         0
      OR        0       ORI         0       RFE         1        SB         0
      SD       28       SEQ         0      SEQI         0        SF         0
     SGE        0      SGEI         0       SGT         0      SGTI         0
      SH        0       SLE         0      SLEI         0       SLL         0
SLLI/NOP        0       SLT         0      SLTI         0       SNE         0
    SNEI        0       SRA         0      SRAI         0       SRL         0
    SRLI        0       SUB         0      SUBI         7      SUBU         0
   SUBUI        0        SW         0      TRAP         1       XOR         0
    XORI        0
Total integer operations = 74

FLOATING POINT OPERATIONS
=========================

    ADDD        0      ADDF         0    CVTD2F         0    CVTD2I         0
  CVTF2D        0    CVTF2I         0    CVTI2D         0    CVTI2F         0
    DIVD        0      DIVF         0       EQD         0       EQF         0
     GED        0       GEF         0       GTD         0       GTF         0
     LED        0       LEF         0       LTD         0       LTF         0
   MULTD       28     MULTF         0       NED         0       NEF         0
    SUBD        0      SUBF         0
Total floating point operations = 28
Total operations = 102
Total cycles = 109
```

The dynamic counts for all instructions are shown, as well as the statistics previously discussed. The number of load stalls is seven in this case, compared to 28 in the first example. This is the result of unrolling the loop four times and providing four multiply units in hardware. An estimate of the clocks per instruction (CPI) can be obtained by dividing the total cycles (109) by the total operations (102).

The two examples above give only a flavor for the types of operations which may be done in **DLXsim**. The possibilities are endless.

3 Internal Operation

Some information concerning how **DLXsim** operates internally may be useful to some users, particularly those who wish to modify or enhance the simulator. This section provides an overview of the simulator and a discussion of the underlying data structures used. *This information is not necessary to use* **DLXsim**. All of the code discussed below is contained in the file sim.c.

3.1 Instruction Tables

DLXsim contains four tables which contain information about the DLX instruction set. The first is opTable. This table contains 64 entries corresponding to the 64 possible values of the opcode field. Each entry consists of an instruction-format pair. For example, the value of opTable[5] is {OP_BNEZ, IFMT} indicating that opcode 5 is a branch not equal to zero instruction, which uses the I-type format. Several entries in this table have OP_RES as the instruction. These entries are reserved for future extensions to the DLX instruction set.

The zero opcode indicates a different table should be used to identify the instruction. A second table called specialTable handles this case. In this table are all the register-register operations. The format is not specified explicitly for these instructions (as it was in opTable) because they are all R-type format. These instructions all contain a zero in the opcode field and a function encoding in the lower six bits of the instruction word. There is also room in this table for expansion by using entries currently containing OP_RES.

An opcode of one indicates a floating point arithmetic operation. A third table, FParithTable handles these instructions. As with specialTable, all instructions in this table have R-type format. The exact operation is again specified by the lower six bits of the instruction word, which are used to index into this table. Currently 32 entries contain OP_RES and are available for future expansion to the floating point instruction set.

The final table is operationNames. This table contains a list of all the integer instruction names followed by the floating point instruction names. Each group is arranged alphabetically. These tables are used to print out the names of the instructions when a dynamic instruction count is requested.

3.2 Simulator Support Functions

This subsection describes the various routines which handle simulator commands and provide support for the main simulator code. The function Sim_Create initializes a DLX processor structure and is invoked when **DLXsim** is first started. The memory size of the machine along with the floating point hardware specification (i.e. unit quantities and latencies) are specified as parameters.

Two functions, statsReset and Sim_DumpStats, process the `stats` command in **DLXsim**. The former resets all the statistics to zero, and the latter processes requests for various statistics. The statistics currently taken during simulation are: load stalls, floating point stalls, dynamic instruction counts, and conditional branch behavior. In addition, the floating point hardware and pending floating point operations can also be examined. See the description of the `stats` command for more information on how to request and reset the various statistics.

The functions Sim_GoCmd and Sim_StepCmd process the simulator's `go` and `step` commands, respectively. See the description of these commands for more information on using them.

The functions ReadMem and WriteMem provide the interface between the simulator and the DLX memory structure. They insure that the address accessed is valid, which means in must be within the memory's range and it must be on a word boundary. Otherwise, appropriate error handling occurs.

3.3 Compilation of Instructions

To improve efficiency, **DLXsim** "compiles" the instructions as it first encounters them. To understand how this works, it is necessary to examine the structure of a single word of the DLX memory. A single memory word contains several fields: value, opCode, rs1, rs2, rd, and extra. A DLX program to be simulated is written in DLX assembly language. Such a program is automatically assembled into machine code as it is loaded. The actual machine codes are stored in the value fields of the memory words. The value field represents the number actually stored at a particular memory word. The opCode field of each memory word is initially set to the special value OP_NOT_COMPILED.

When the simulator executes an instruction, it first examines the opCode field of the memory word pointed to by the program counter. If this field is a valid opcode (specified in the tables discussed above), the appropriate action for that instruction occurs. If the opCode field contains the value OP_NOT_COMPILED, the function Compile is invoked. This function looks at the actual word stored in the value field. The bits corresponding to the opcode and function fields are examined to determine what the instruction is. Depending on the instruction type, the two source register specifiers and destination register specifier may be extracted and stored in the fields rs1, rs2, and rd. If a 16-bit immediate value is present (for I-type instructions) or a 26-bit offset is present (for J-type instructions), this value is extracted and stored in the extra field of the memory word. The special code for the instruction is stored in the opCode field of the word, which previously contained the value OP_NOT_COMPILED. These special codes are not the real DLX opcodes, but rather the pseudo-opcodes defined in the file dlx.h.

When a compiled instruction is subsequently encountered, no shifting or masking operations are required to access the register specifiers or immediate values; the required information is already present in the appropriate fields of the memory words (rs1, rs2, rd, and extra). This allows the simulator to execute much faster. The actual machine code for the instruction can still be examined through the value field, and this is the value printed when the word is examined with the get command.

3.4 Main Simulation Loop

Simulate is the main function of the simulator. The heart of this function is basically a very large switch statement, based on the opCode field of the memory word pointed to by the program counter. There is a case for each integer and floating point instruction. Simulate loops through the basic fetch-decode-execute cycle until a stop command is received or some other exceptional condition occurs.

3.4.1 Load Stalls

DLX has a latency of one cycle on load instructions. In other words, the result is not yet present in the destination register on the cycle immediately following the load instruction. To address this problem, DLX has load interlocks which cause the pipeline to stall if an instruction immediately following a load instruction reads the value in the load's destination register. **DLXsim** records the occurance of these load stall cycles for statistical purposes. Several variables are set during the processing of the following load instructions: LB, LBU, LH, LHU LW, LF, and LD. LHI is not included since the value to be loaded is contained in the instruction and there is no extra latency. For the other load instructions, the destination register (or registers in the case of load double) are stored in loadReg1 and loadReg2 (if this is a load double). The corresponding values to be stored in these registers (on the next cycle) are stored in loadValue1 and loadValue2.

When an instruction that reads registers (such as an ADD instruction) is encountered during simulation, the contents of loadReg1 and loadReg2 are examined before any other action occurs. If either of the registers specified by these variables were loaded in the previous instruction, a load stall is detected and tallied. Different register fields must be checked for different instructions. All the load stall detection logic is contained in the macros at the top of the Simulate function definition.

Of interest is the fact that while load stalls would slow down the execution speed of a real DLX machine, they do not affect the performance of the simulator. This is because load stall cycles are not actually simulated. Instead, it is simply noted that a load stall occurred at a particular point, and execution proceeds normally.

3.4.2 Dynamic Instruction Counts

Statistics on the number of each type of instruction executed are also recorded during simulation. This is a simple operation of incrementing the appropriate element of the array operationCount, which is indexed by the pseudo-opcodes discussed above. The information in the array can be accessed by the `stats` command.

3.4.3 Conditional Branch Behavior

DLXsim also keeps statistics on the conditional branch behavior during program execution. There are four instructions in this category: BEQZ, BNEZ, BFPT, and BFPF. The latter two instructions are branches based on the status of the floating point condition register. Two fields of the DLX machine structure, branchYes and branchNo record how many conditional branches where taken and not taken, respectively. These values are accessible via the `stats` command.

3.5 Floating Point Execution Control

A large portion of the **DLXsim** code is devoted to the floating point side of the machine. The floating point scheme currently implemented requires instructions to issue in order, but they may complete out of order. In addition to managing the allocation of the floating point units, **DLXsim** must also handle all the hazard checking associated with out of order completion of instructions. By requiring instructions to issue in order, the write-after-read (WAR) hazard is avoided. The three hazards which may occur are read-after-write (RAW) hazards, write-after-write (WAW) hazards, and structural hazards.

3.5.1 Floating Point Data Structures

The variables and data structures which manage the floating point execution are all declared in the file dlx.h as part of the basic DLX structure. The variables num_add_units, num_div_units, and num_mul_units specify how many of each type of floating point execution unit are available on the machine. The variables fp_add_latency, fp_div_latency, and fp_mul_latency specify the corresponding latencies (in clock cycles) of each of the execution units. All six of these variables have default values which may be overridden via command line parameters when **DLXsim** is invoked.

The variable FPstatusReg is the status register which is examined on a BFPT or BFPF instruction. The various floating point set instructions (EQF, NED, etc.) write to this register.

The array fp_add_units contains the status of all the floating point adders during execution. If fp_add_units[i] is zero, adder i is available. A non-zero value means that the unit is currently performing an operation – the value specifies the clock cycle when the operation will complete. The array fp_div_units and fp_mul_units contain analogous information for the floating point dividers and multipliers. All three structures can be accessed through the array fp_units which is an array of pointers to the three execution unit status arrays.

The array waiting_FPRs contains 32 elements, corresponding to the 32 floating point registers in DLX. A zero in waiting_FPRs[i] means floating point register Fi can be read from; it contains its most current value. A non-zero value means register Fi is the destination register of a pending floating point operation (one which has issued but not yet completed). Attempting to read or write to such a register means a hazard condition exists. The non-zero value indicates the cycle at which the writeback to the register will occur.

The variable FPopsList points to the chain of pending floating point operations. Each item in this chain is of type FPop, a structure with the following fields:

type — Indicates the type of operation. Normally this is implied by what type of floating point unit is executing the operation, however adders can perform both additions and subtractions.

unit — The unit number of the execution unit which is executing the operation.

dest — The destination register for the operation. For a double precision operation, this is the lower-numbered destination register.

isDouble — Indicates if the operation is single or double precision.

result — An array of two floats used to store the result of the operation (only the first element is used for single precision operations). The result is actually computed at the time of issue.

ready — The cycle when the operation will complete and writeback will occur.

nextPtr — Points to the next FPop in the chain of pending operations.

To maximize performance, the list of pending floating point operations is sorted based on when the operations will complete. The operation which will complete soonest is at the head of the list.

The variable checkFP is a copy of the ready field of the first floating point operation on the pending operation list. If its value is zero, no floating point operations are pending. Otherwise checkFP indicates when the next (soonest) floating point operation will complete. This provides for very quick checking in the fast-path of the simulator. Only one value needs to be checked in a cycle when no writebacks should occur.

Many of the previously discussed structures refer to a clock cycle count when a particular operation will complete. The current clock cycle is kept in the variable cycleCount. This variable is incremented each time the simulator executes its main loop. It is also incremented an extra time when a load stall is detected since the floating point units are still executing during a load stall. When the cycle count reaches a large value specified by the constant CYC_CNT_RESET, cycleCount is "reset" back to a small number (5), and all references to clock cycles in the floating point data structures are adjusted accordingly. This operation is necessary to prevent cycleCount from overflowing, becoming negative, and thereby wreaking havoc on the sorted list of pending operations. Making cycleCount an unsigned integer does not work, since there are still problems with sorting the pending operations when cycle counts "wrap around" to zero.

3.5.2 Issuing Floating Point Operations

The function FPissue initiates a floating point operation. It is called from eight of the switch cases in the main loop: ADDF, DIVF, MULF, SUBF, ADDD, DIVD, MULD, and SUBD. When a floating point instruction issues, three hazard conditions must be checked. A structural hazard occurs if a floating point unit of the required type is not available. A RAW hazard occurs if one of the source operands is the destination of a pending floating point operation. Finally, a WAW hazard occurs if the destination register is the destination register of a pending floating point operation. All three conditions can be checked by examing the floating point data structures discussed above. If any of these hazards are present (and there may be more than one), the current instruction is not issued. Instead a non-zero value is returned which indicates the soonest cycle when one of the hazard conditions will be over. This may be a cycle when one of the floating point units will complete its current operation (eliminating a structural hazard), or when some register will be written back (eliminating a RAW or WAW hazard). When the caller receives a non-zero value from FPissue, the appropriate number of floating point stalls are simulated by adjusting the variables cycleCount and FPstalls. The function FPwriteBack (see below) is called to perform any writebacks which may now occur. Then FPissue is re-invoked. If another hazard condition exists, the whole process may be repeated, but eventually all of the hazard conditions will terminate.

If no hazards are present, the instruction is issued. That is, an new FPop structure is placed in the appropriate spot in the pending operations list. The appropriate elements of waiting_FPRs are also set to indicate that the destination registers are waiting for values to be written back. FPissue returns a zero value to indicate a successful issue, and the simulation continues.

3.5.3 Writing Back Floating Point Results

The function FPwriteBack is the second function involved in floating point execution. It is called whenever cycleCount reaches checkFP, indicating that a result is ready to be written back on the current cycle. FPwriteBack does exactly that. It removes the first FPop from the list of pending operations, and stores the result (computed at time of issue) in the appropriate register(s). It also zeroes the appropriate element(s) in waiting_FPRs. Since more than one operation may complete on the same cycle, FPwriteBack repeats this process until the value in the ready field of the operation at the head of the list exceeds the current value in cycleCount.

3.5.4 Handling RAW and WAW Hazards

The function FPissue (discussed above) handles the RAW and WAW hazards when a new floating point operation is issued. However, several other instructions can generate such hazards. Any instruction which reads from or writes to a floating point register must check that the register is not the destination of a pending operation. The following instructions fall into this class:

Loads LF and LD.

Stores SF and SD.

Moves MOVFP2I, MOVI2FP, MOVF, MOVD.

Converts CVTD2FP, CVTD2I, CVTFP2D, CVTFP2I, CVTI2D, CVTI2FP.

Sets SEQF, SNEF, SLTF, SLEF, SGTF, SGEF, SEQD, SNED, SLTD, SLED, SGTD, SGED.

When any of these instruction are executed, a call to FPwait is made. This is the third and final function for handling floating point execution. It checks that all writebacks into the appropriate registers have occurred. The number of registers which need to be checked varies. For a LF instruction, only a single register needs to be checked, while four registers must be checked on a MOVD. If any of the registers are the destinations of pending operations, FPwait will adjust cycleCount and FPstalls appropriately, and call FPwriteBack to write the results back to the registers. When FPwait returns, all RAW and WAW hazard conditions will have passed.