Chapter 1

Implementation of a Vectorized DLX Simulator

1.1 Introduction

This project involved extending the **DLXsim**, the simulator written in the University of California at Berkeley, to incorporate the simulation of a vector machine. The vector architecture simulated is the DLXV architecture described in *Computer Architecture, A Quantitative Approach* by Hennessy and Patterson ([1]). The **DLXVsim** (the DLXV architecture simulator) uses the same user interface as **DLXsim** described in [2], augmented by few more commands to examine vector machine behavior. When extending the **DLXsim** there were two major tasks to consider: creating and operating on the data structures for vector architecture simulation and extending the instruction set already implemented in **DLXsim**. These two issues will be addressed in this report along with the description of the machine simulated and the design decisions and assumptions made. This will be followed by some examples of how to use the simulator. We expect that a major use of this tool will be in Computer Architecture classes to help in understanding of vector architecture. Few sample exercises that can be used in these classes, may be found at the end of this report.

1.2 The DLXV Architecture

The DLXV architecture simulated is basically the one described in Chapter 7 of [1]. The major characteristics of the DLXV architecture are as follows.

- There are 8 **vector registers**. Each register is a fixed-length bank holding a single vector. A vector consists of 64 double precision floating point numbers. However, these are just default values. The user has the opportunity of altering these hardware parameters (as well as the others described later). The number of vector registers can be anything from 0 to 16, and the maximum vector length supported by the simulator is 1024 doublewords. This covers the most general existing vector architectures like CRAY-1, CRAY-2, X-MP, Y-MP, Convex C-1 and others.

- There are special **vector functional units** to carry out vector operations. These units are: vector add/subtract unit, vector multiply unit, vector divide unit and the logical unit. All the units are fully pipelined and can start a new operation on every clock cycle. Each unit has a *startup cost (startup penalty)* associated with it. The startup cost is the number of clock cycles elapsed from the time the first element of the vector enters the unit until the time the first result is ready. It equals to the number of pipeline stages in it. Since the number of functional units is limited, structural hazards may arise during the execution and vector stalls need to be inserted to prevent a hazard. By default there is only one copy of each functional unit and the startup penalties are 6 clock cycles for vector add, 7 – for vector multiply, and 20 – for vector divide. Both, the number of functional units and the start-up cost for each of them can be specified by the user.

- **Vector load/store unit** is also needed. It is fully pipelined, i. e. a new word can be written to or from the memory on each clock cycle. The default start-up cost for this unit is 12 clock cycles but can be changed by the user.

The **DLXVsim** supports all the instructions described on p. 356 of [1]. All the instructions work on double-precision floating point numbers.

1.3 Extending The Instruction Set Of DLXsim

The **DLXsim** simulator gives you the opportunity to extend the standard instruction set. Obviously, this had to be done in order to incorporate vector machine simulator. There are few details one should consider when extending the instruction set. Here is the pattern which may be followed when doing that.

1.3.1 Modifying Instruction Tables

File *sim.c* contains instruction tables with information about DLX instruction set. The first is opTable. It contains entries corresponding to possible opCodes. Each entry consists of the opCode itself and the type of the instruction: I-type, R-type or J-type. (See p. 166 of [1] for the description of these types). Several entries in the table have the opcode OP_RES and are reserved for new instructions. So, if you want to have a new instruction in the instruction set, you put its opcode instead of any OP_RES. Be careful not to change the existing order of instructions if you do not want to encounter unnecessary complications with assembling the DLX code.

All the register-register instruction have one of the following two opcodes in this table: SPECIAL or FPARITH corresponding to integer and floating point operations. These opcodes indicate that another table should be used to identify the instruction. The tables are specialTable and FParithTable correspondingly. All the vector instructions have VECTARITH opcode in the opTable. This was done because all the vector instructions in DLXV are register-register ones and we decided to follow the idea of separating each class of the R-type instruction in a different table. This allows identifying all the vector instructions just by one opcode and also reserves space for up to 64 vector operations.

There is one more table (operationNames) which is used to print out the names of the instructions when the dynamic instruction count is requested. It contains the alphabetic list of all integer instructions, followed by floating point instructions, followed by vector instructions. The instruction index in this table should be equal to the number corresponding to each opCode as defined in the #define statements in the *dlx.h* file. So when modifying one (the table or the definitions) do not forget to modify the other.

1.3.2 Assembling And Disassembling New Instructions

New instructions need to be assembled and disassembled when they appear in the source file. The procedures for doing that are contained in the *asm.c* file. This file also contains the information about all opcodes in the opcodes table. New entries describing new instructions should be added to this table. The following comments might be useful when you want to do that.

- For each opcode there is a certain *class* it belongs to. Few more classes were introduced solely for vector operations. Instructions belonging to one class have the same format, meaning the same number of operands, parenthesis, delimiters, etc. This division makes the work of lexical analyzer easier.
- There is a *bit pattern* corresponding to each operation. First six bits of the pattern constitute the opcode, and the value represented by these bits should be the index of this instruction in the opTable in *sim.c* file. If the auxiliary table is needed to identify the instruction, the last six bits of the pattern contain the index in that table. The bits in the middle should remain zeroes (these are the bits for the operands information). The more detailed description of the bit patterns for the operations can be found in Appendix E of [1].
- Various *flags* are used to give more information about the instruction, such as type of operands (general purpose register, floating point register, immediate) or range checking information. For vector operations new flags were needed to show that vector register or some special register like *vector-mask register* or *vector-length register* is required.

Some changes in the lexical analizer are also required to parse new types of instructions and delimiters.

1.3.3 Implementing New Instructions

The last and, probably, the most obvious change needed, is to add new cases to the very large switch statement in Simulate function. These cases should contain the implementation of the instruction itself. Here the semantics of the instruction is considered and the corresponding actions implied from it, are simulated. All the vector "cases" call the corresponding functions from the *vect.c* file. The description of these functions follows in the next section.

1.4 Vector Execution Control

Data structures and execution control for vectors are similar to those for floating point data structures and control implemented in **DLXsim** .

1.4.1 Vector Data Structures

The vector variables and data structures are all declared in the file *dlx.h*. The variables are the fields of the DLX structure describing the state of the machine.

Vector Registers

Each vector register is simulated by an array of 1024 double-precision floating point numbers which is defined as type VR. To simulate vector registers of the smaller length, field mvl (maximum vector length) in DLX structure is used. The default value of this parameter is 64. Pointers to the content of all the register are in the array variable VRegs.

Vector Units

Five units operate on vectors in DLXV: add/subtract, divide, multiply, load/store and compare. For each type of the functional units there is:

- a variable, specifying the number of the units of this type (e.g. num_vect_add)
- a variable, specifying the start-up cost for this unit (e.g. vect_add_startup)
- an array, containing the status of each vector unit of this type (e.g. vect_add_units). A non-zero value means that the corresponding unit is busy. This value specifies at which clock cycle the unit is able to start getting new data. It is **not** the clock cycle when the unit finishes the operation it is doing: all the functional units are fully pipelined and can get a new value on every clock cycle. So, when one vector is not fully processed yet, the unit can start working on the next one.

The arrays described in the last item are accessed via the array of pointers vect_units.

Special Vector Registers

For efficient operation of the vector machine two special registers are required: *vector-mask register* and *vector-length register*. (See sections 7.3 and 7.6 of [1] for detailed description of their use.) The value of the *vector length register* is stored in the VLR variable. The *vector-mask register* is an array of bytes pointed by VM where the number of bits is equal to the maximum vector length. Therefore, each bit corresponds to each element of the vector. The value of variable VM_Loaded indicates if this vector has been loaded and should be taken into consideration or not.

Data structures for hazard detection

There are two arrays used for vector data hazard detection: waiting_VectRs and being_read_VectRs. A non-zero value in the former one means that the corresponding vector is waiting for the result and the value is the clock cycle when this result is being written. The second array is needed because of the following feature of the vector operations: when the instruction starts its execution, during the first 64 cycles (or whatever the vector length is) the values are fed from the register to the vector unit. Although the instruction is not completed at this point, the source vector register values are not needed any more and the register can become available for writing. So, if any instruction is stalled to prevent WAR hazard, it can be issued now. The non-zero value in the array being_read_VectRs is the clock cycle when the register becomes available for writing. There is one extra element in each of the arrays to keep track of the status of the *vector-mask register*.

Pending Vector Operations List

The list VectOpsList is very similar to the list FPopsList used for floating point operations (see [2]). It is a linked list of structures of type Vectop, where each element contains all the information needed about pending vector operation. The first element of the list is the instruction which will finish first, followed by the one that will finish second, and so on.

1.4.2 Issuing Vector Operations

Before each vector operation can start its execution, it needs to be issued. So, from each switch case for vector instructions in the *vect.c* file, VectIssue is called. This function detects if there are any data or structural hazards. It checks for the availability of appropriate registers for reading or writing and availability of the corresponding functional units. If any hazard occurs and the instruction needs to be stalled, VectIssue returns the number of clock cycles it needs to be stalled for. Otherwise, the return value is zero and the appropriate entry in the VectopsList is created. The results are computed at this stage and the values in the appropriate controlling data structures are modified.

Parameters of VectIssue

Since all the vector operations are different in terms of type of registers used (vector, floating point or general purpose registers), functional units needed, types of results returned, the number of different flags are passed to this procedure to ensure that all hazards are detected correctly and at the same time no extra stalls are inserted.

type Indicates what are the types of the operands of the instruction and what is the type of the result. This flag consists of two parts divided by '_'. First one can be:

- VECTOR, if the result is a vector;
- MASK, if the instruction sets the new value for the *vector-mask register*;
- UNIMP is used for vector store operations because they do not produce any result.

The second part of the flag are the two letters specifying type of both source operands and can be:

- VV indicating that both register are vector ones;
- FV indicating that the first operand is a floating point register and the second is a vector one. If the instruction takes the floating point register as the second one (e.g. SUBSV), this register is still passed as the first one to the VectIssue procedure.
- IV stands for general purpose and vector register. This situation can occur for vector loads and stores.

unitType is the type of vector functional unit which is used for this instruction.

source1, source2, dest are source and destination registers. If a scalar register is present, it is always passed as the first operand. If any of the values is NONE, it means that there is only one source operand (e.g. for LV) or no destination register correspondingly.

penalty can be 0 or 4. There is a penalty of 4 clock cycles for the vector instruction dependence. So, if the instruction has been stalled before and is trying to issue more than once, this parameter is equal to 4.

Data Hazard Detection

The first thing to check here is the *vector-mask register*. It is used for all the instructions so it is better to start with it. If it is in the process of being loaded, no other instruction can be issued.

If the destination register of the new instruction is at the same time a source operand of the instruction already in execution, the new one has to stall before that register will be read and, therefore, will become available for writing. If it is being written, more stalls are needed to wait when the writing is finished. The only thing to care about for the source operands is that they are not being written. If yes, stall also.

Issuing One-Cycle Instructions

There is a number of vector instructions which take only one clock cycle to execute. These instructions do not use any of the vector functional units and do not need to be recorded in VectOpsList. These are moves from fp registers to *vector-mask register* and vice-versa (MOVF2S and MOVS2I). The choice has been made to make CVM (set *vector-mask register*) and POP (count the number of 1's in the *vector-mask register*) execute in one clock cycle as well. For the first one resetting the whole vector-mask is not necessary: it is sufficient just to change the value of the flag which indicates whether the register is loaded or not. There is more "action" necessary to execute POP, but if it takes more than 1 cycle to complete, then we will have such thing as "pending operation with the integer result" and this one will be the only one of this kind. Firstly, it will only complicate the things: separate structure for this kind of instructions will be needed and checking for hazards before issuing *any* instruction (integer, fp, vector) becomes necessary. Secondly, if we want to make it possible to execute POP in one cycle, the extra hardware needed for that is not that difficult. So, the choice has been made towards one-cycle execution.

Structural Hazard Detection

The structural hazard detection is very similar for that of floating-point arithmetic. It forces the instruction to stall if the functional unit is not available. Since all the vector functional units are fully pipelined, the condition to check is: is the first stage of the unit pipeline empty or not?

Computing the result

If the instruction has been successfully issued, its result is computed, and the new entry is inserted in the VectopsList. When the value is being computed, the *vector-mask register* and *vector-length register* values are taken into consideration. For the store operations the values are written in the DLX memory at this stage. For all others, the destination registers are not being modified here:

this will be done later in the VectWriteBack when the number of clock cycles needed to complete the instruction elapses.

1.4.3 Writing Back the Results

Each time when the counter of the clock cycles is incremented, the VectWriteBack function is called. It scans through the list of the pending vector operations, and if some of the instructions have finished by this time, it writes the new values in the destination registers and resets the values in the hazard control structures. The instructions which have finished are at the beginning of the list, so there is no necessity to go through the whole list to find them. However, some instructions can just finish reading the source operands by this time and this also needs to be reflected in the control structures: the appropriate values in the being_read_VectRs array should be reset and the functional units should become available for incoming instructions. The list is not ordered with respect to the value of the read field, but it is easy to note that the maximum difference between ready and read values is the maximum possible startup cost (which is obviously the one for the vector division). Therefore, once the read values become more than $cycle\ count + division\ startup$, we can stop scanning through the list. After this VectWriteBack returns.

1.4.4 The Synchronizing Instruction – sync

When writing the vector machine simulator, the necessity of one more instruction came up. This instruction is not included in the vector instruction set of DLXV, however, it appears to be useful. It is a synchronizing instruction sync which is equivalent to as many nops as necessary to complete all the operations in execution at that moment. The reason for doing this is as follows.

When the control is passed to the operating system by trap, there still could be pending floating point and vector operations. This situation is much more frequent for vector case because of the long latency of all vector instructions. So, to have the results ready by the time the control is passed to the operating system, certain number of stalls should be inserted. The sync instruction does it. It goes through the lists of pending floating point and vector operations, finds the one which takes the longest time to complete and sets the cycle count to that cycle when it will be ready.

Hardware implementation of this instruction does not seem to be very costly. The only thing needed is some circuitry to check if any of the functional units is busy, and if yes, insert stalls.

As far as the simulator is concerned, the process of finding the last instruction is easy because the lists of instructions in execution are ordered with respect to the cycle when they will be ready.

1.5 Additions To the Manual of DLXsim to Examine Vector Machine Behavior

Few more commands were added to the user interface of **DLXsim** (see manual entry of [2]). These are the commands to specify vector hardware features, examine content of vectors, get the information on the vector instructions in execution.

1.5.1 Calling DLXVsim

To make **DLXsim** to be a vector machine simulator (**DLXVsim**) one should call it with -VECTOR option:

```
% dlxsim -VECTOR
```

1.5.2 Specifying Vector Hardware Features

DLXVsim can be called with the number of parameters such as number of vector functional units, their startup costs, maximum vector length and the number of vector registers. The vector options of the **dlxsim** command are the following:

-vas# Select the startup cost for a vector add.

-vau# Select the number of vector add units.

-vds# Select the startup cost for a vector divide.

-vdu# Select the number of vector divide units.

-vms# Select the startup cost for a vector multiply.

-vmu# Select the number of vector multiply units.

-vcs# Select the startup cost for a vector compare.

-vcu# Select the number of vector compare units.

-vls# Select the startup cost for a vector load/store.

-vlu# Select the number of vector load/store units.

-mvl# Select the maximum vector length.

-vrs# Select the number of vector registers.

1.5.3 Accessing Vector Elements

Vectors located in the DLX memory can be accessed by the **fget** and **fput** commands provided by **DLXsim**. However, this commands do not give the access to vector registers. To do that and also to address vectors in memory in the vector-oriented style (with index), one can use the following **DLXVsim** commands:

vget *address* [*[index* [*..index*] *]*]

Return the values of one or more elements of vector located in memory or in a vector register. *Address* identifies a memory location or register, and *index*, if present, indicates values of which elements of the array to print. The values are printed as double precision floating point numbers.

vput *address [index] number*

Store *number* in the *index* element of vector register or memory location given by *address*. The number is stored as a double precision floating point number (in two words).

1.5.4 More Statistics Options

Some of the options of the **stats** (dump statistics) command of the simulator were revised to incorporate the information of the vector machine behavior. One new option has been added to examine the vector hardware configuration. Here is the list of the **stats** options which were changed as compared to the **DLXsim**:

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

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

vhw Show the current vector hardware setup for the simulated machine.

The next section gives an example of an interactive session with **DLXVsim**.

1.6 An Example of an Interactive Session with DLXVsim

Here is an example of the interactive session with the **DLXVsim**. We will consider the following simple program written in the DLXV assembly language:

```
        .data 0
        .global A
A:      .double 1, 2, 3, 4, 5, 6, 7, 8
        .global B
B:      .double 1, 0, 1, 0, 1, 0, 1, 0


_main:
        cvm                     ; clear vector mask register
        addi    r1, r0, A       ; store the address of A in r1
        addi    r2, r0, B       ; store the address of B in r2
        lv      v1, r1          ; load vector A in v1
        lv      v2, r2          ; load vector B in v2
        addv    v1, v1, v2      ; A = A + B
        sv      r1, v1          ; store v1 in A
        sync                    ; synchronize
        trap    #0              ; pass the control to operating system
```

As one can see, this program takes the two vectors, adds them and stores the result in the first one.

The observation that can be made is that the vector length is only 8 numbers, so the parameter specifying the vector length can be used when calling **DLXVsim** :

```
% dlxsim -VECTOR -mvl8
(dlxsim)
```

Now the code should be loaded. It is contained in the file *example.s*

```
(dlxsim) load example.s
```

Here is the vector hardware configuration for the machine which is being simulated:

```
(dlxsim) stats vhw

Memory size: 65536 bytes.

Vector Hardware Configuration
 8 vector registers
 8 is the maximum vector length
 1 add/subtract unit(s),  startup cost =  6 cycles
 1 multiply unit(s),      startup cost =  7 cycles
 1 divide unit(s),        startup cost = 20 cycles
 1 comparison unit(s),    startup cost =  6 cycles
 1 load/store unit(s),    startup cost = 12 cycles
```

That is, all the parameters except the maximum vector length are the default values of the machine.

Now we can start step-by-step execution to display the abilities of the simulator.

```
(dlxsim) step _main
stopped after single step, pc = _main+0x4: addi r1,r0,0x0
(dlxsim) step
stopped after single step, pc = _main+0x8: addi r2,r0,0x40
(dlxsim) step
stopped after single step, pc = _main+0xc: lv v1,r1
```

So, r1 now contains the address of A, r2 – the address of B and vector load can be started now. There are no pending vector operations so far:

```
(dlxsim) stats pending

Pending Floating Point Operations:
none.

Pending Vector Operations:
none.
```

Nothing prevents the load from issuing: the load/store unit is unused and the source general-purpose register (r1) is available.

```
(dlxsim) step
stopped after single step, pc = _main+0x10: lv v2,r2
(dlxsim) stats stalls pending
Load Stalls = 0
Floating Point Stalls = 0
Vector Stalls = 0

Pending Floating Point Operations:
none.

Pending Vector Operations:
loader       #1 :  will complete in 19 more cycle(s)  ==> v1
```

The last line shows that the loader number 1 (in fact, the only one in our machine) will complete its work in 19 clock cycles:

- 12 clock cycles for the first element to go through the whole load/store unit pipeline
- plus 8 cycles to get all the eight results
- minus 1 clock cycle which has been finished already.
- $12 + 8 - 1 = 19$

The result from the loader goes to the vector register v1.

There were no stalls so far. But now we need to issue one more vector load. However, only one load/store unit is available, so stalls occur before it is issued.

```
(dlxsim) step
stopped after single step, pc = _main+0x14: addv v1,v1,v2
(dlxsim) stats stalls
Load Stalls = 0
Floating Point Stalls = 0
Vector Stalls = 7
```

There were only 7 stalls because by this time all the elements of vector A are already fed in the load unit and the unit can take new values on the next clock cycle. Now the loader is working on the two instructions simultaneously:

```
(dlxsim) stats pending

Pending Floating Point Operations:
none.

Pending Vector Operations:
loader       #1 :  will complete in 11 more cycle(s)  ==> v1
loader       #1 :  will complete in 23 more cycle(s)  ==> v2
```

The second load takes more time than the first one (24 vs. 20) because of the 4 clock-cycle instruction dependence penalty. And more stalls are needed now: `addv` cannot issue before both loads complete, because addition requires both vectors.

```
(dlxsim) step
stopped after single step, pc = _main+0x18: sv r1,v1
Load Stalls = 0
Floating Point Stalls = 0
Vector Stalls = 30

Pending Floating Point Operations:
none.

Pending Vector Operations:
adder        #1 :  will complete in 17 more cycle(s)  ==> v1
 ( register(s) v1  v2  will be read in 11 more cycles )
```

Since the addition is not finished yet, the register v1 should contain vector A. It can be examined now and we will do one more step after that.

```
(dlxsim) vget v1[0..7]
 v1[0] : 1.000000
 v1[1] : 2.000000
```

```
 v1[2] : 3.000000
 v1[3] : 4.000000
 v1[4] : 5.000000
 v1[5] : 6.000000
 v1[6] : 7.000000
 v1[7] : 8.000000
(dlxsim) step
stopped after single step, pc = _main+0x1c: sync
(dlxsim) stats pending

Pending Floating Point Operations:
none.

Pending Vector Operations:
loader       #1 :  will complete in 23 more cycle(s)
 ( register(s) v1  will be read in 11 more cycles )
```

The instruction which will execute next (sync) will insert the number of vector stalls necessary to complete the pending store operation.

```
(dlxsim) step
stopped after single step, pc = _main+0x20: trap 0x0
(dlxsim) stats stalls pending
Load Stalls = 0
Floating Point Stalls = 0
Vector Stalls = 70

Pending Floating Point Operations:
none.

Pending Vector Operations:
none.
```

And to finish:

```
(dlxsim) step
TRAP #0 received
(dlxsim) vget A[0..2]
 A[0] : 2.000000
 A[1] : 2.000000
 A[2] : 4.000000
```

The values of the elements of vector A have been modified.

One could notice the big number of stalls in this example. The vector machine does not look very efficient here. This is the reason why various refinements are used to improve the performance: vector-mask capability, load/store with stride or with index. The influence of these improvements can be explored by using this simulator. All the instructions described on the p. 356 of [1] can be used in the assembly codes to be run on the simulator. For better understanding of the details of vector architecture and DLXV instruction set, the exercises after this section can be used.

Exercises

Exercise 1.

Consider the following code.

```
for (i = 0; i < 64; i++) {
       A[i] = A[i] / B[i] + x * C[i]
}
```

(a) Write the DLX and DLXV codes for this loop.

(b) Try to optimize both codes.

(c) Compare the performance. Use such parameters as number of operations, number of clock cycles, number of stalls. How would you explain the results?

Exercise 2.

In this exercise you will see how different hardware parameters can effect performance of the machine.

The DLXV simulator has default hardware parameters as described on pp. 353-354 of [1]. That is by examining configuration of the machine you will see the following:

```
dlxsim % dlxsim
(dlxsim) stats vhw

Memory size: 65536 bytes.

Vector Hardware Configuration
 8 vector registers
64 is the maximum vector length
 1 add/subtract unit(s),  startup cost =  6 cycles
 1 multiply unit(s),      startup cost =  7 cycles
 1 divide unit(s),        startup cost = 20 cycles
 1 comparison unit(s),    startup cost =  6 cycles
 1 load/store unit(s),    startup cost = 12 cycles
```

However, you may change this configuration. In this case, you should be aware that by increasing the number of functional units, you may increase the corresponding startup cost. Make your assumptions of how much this cost is increased. For example, reasonable assumptions may be as follows: *with the addition of each new load/store unit, the startup cost is increased by 30%, and the same number for the vector add unit maybe 5%*

Now consider the following DLXV code (you will find it in the file ex2.data):

```
.text   0x800
addi    r1, r0, A       ; store address of vector A in r1
addi    r2, r0, B       ; store address of vector B in r2
addi    r3, r0, C       ; store address of vector C in r3
lv      v1, r1          ; load A
lv      v2, r2          ; load B
lv      v3, r3          ; load C
addv    v1, v1, v2      ; A = A + B
sv      r1, v1          ; store A
addsv   v3, f0, v3      ; C = 10 + C
addv    v3, v1, v3      ; C = A + C
sv      r3, v3          ; store C
sync                    ; complete all the pending operations
trap    #0
```

Note: consider that each vector has 16 elements

Find the hardware configuration which results in the least number of clock cycles for this code. Read the DLXV manual to see which parameters you can alter. Explain, why adding more functional units even with the small increase of startup costs may not be profitable. Do you think it is a typical situation? Why? Try to optimize this code considering the original DLXV configuration.

Exercise 3.

Consider the following code.

```
for (i = 0; i < 64; i++) {
    if (B[i] != 0)
       B[i] = A[i] + B[i]
}
```

(a) Write the DLXV code for this loop using the vector-mask capability.

(b) Write the DLXV code for this loop using scatter/gather.

(c) Run the DLXV simulator for different number of 0's in vector B (1, 32, 63). Compare the performance. Use the parameters like number of clock cycles, number of vector stalls, etc. Considering hardware costs, which would you build if each of the above loops was typical?

Chapter 2

Implementation of a Pipelined DLX Simulator

2.1 Introduction

This project involved extending the **DLXsim**, the simulator written in the University of California at Berkeley, to be a DLX pipeline simulator. The DLX pipeline and the DLX instruction set architecture are described in *Computer Architecture, A Quantitative Approach* by Hennessy and Patterson ([1]). The instructions execution in the extended **DLXsim** is pipelined. Each instruction takes at least five clock cycles to complete (there are five pipe stages), and the user can trace what is happening at each stage at any moment of time. The simulator gives the opportunity to use integer and floating point instructions and supports the whole DLX instruction set introduced in [1]. It also gives additional statistics values as compared to the ones described in [2]. This report contains a brief overview of the DLX pipeline architecture followed by the detailed discussion on the simulator implementation issues. It also includes an example of an interactive session with **DLXsim** with the focus on examining pipeline features. The expectation is that the major use of this tool will be in Computer Architecture classes to help the students in understanding what pipeline is, what problems it may cause and how to deal with them.

2.2 DLX Pipeline Architecture

The pipeline implemented in this simulator is the standard DLX pipeline described in Chapter 6 of [1]. It is the five stages pipeline where new instruction is fetched on each clock cycle and one instruction completes its execution at each cycle (provided, of course, that no hazards occur).

2.2.1 Stages of the DLX Integer Pipeline

DLX pipeline consists of five stages:

IF – Instruction Fetch;

ID – Instruction Decode;

EX – Execute;

MEM – Memory Access;

WB – Write Back.

The first two stages are the same for all types of instructions:

- a new instruction is fetched from DLX memory on IF stage and the program counter is modified;
- the instruction is being decoded at ID stage and the values of the operands are stored at latches. Since the decoding is done only by the end of the clock cycle, the operand values are fetched in both ways: assuming that the operands are register numbers and assuming that the second operand is immediate data. After the instruction is decoded, the unnecessary value is discarded. The program counter of the following instruction is stored at the latch for further use in preventing big penalties from control hazards.

Now, when it is known what kind of instruction it is, different actions can be done according to that. Obviously, EX, MEM and WB stages are different for different groups of instructions. Three major groups are: arithmetic instructions, load/stores and control instructions (branches and jumps). For each of the groups the content of the last three pipe stages is as follows:

1. Arithmetic operations:
   - ALU computes the result at the EX stage;
   - no memory access is needed, so nothing is done at MEM stage;
   - the result is written to destination register during WB.
2. Load/Store instructions:
   - if it is load, then address which should be accessed is computed at EX stage and is written to the Memory Address Register; if it is store, the value to be stored is written to the Memory Data Register
   - everything is ready for memory access now, and it is done at MEM stage; value is read or written – depending on the instruction.
   - WB is not active for stores: there is nothing to write back; the result of load is written to the destination register.
3. Control Instructions: to prevent big penalties from the control hazards, the new program counter is computed at the first two stages of the pipeline. It is possible to do that because of the following reasons:
   - by the end of ID instruction is decoded, therefore, it is known whether it is branch/jump or not;
   - immediate data is at the corresponding latch, therefore, the target address is known;
   - additional hardware detects whether the value of the first operand is zero or not, therefore, the branch decision is made.

The software implementation of the simulator of this kind of pipeline is discussed in further sections.

2.2.2 Floating Point Pipeline

Floating point instructions create more complications for the pipeline. The reason is that the EX stage takes more than one clock cycle for most of the floating point instructions. Being an intricacy by itself it causes another one: overlapping of integer and floating point instructions in MEM or in WB stages. To simplify pipeline control in this case, the integer and floating point registers are separated in two different register files, each with its own read/write port. This implies that one integer and one fp instruction can be in the same pipe stage (MEM or WB) at the same clock cycle, however, two instructions of the same kind cannot. This situation will not arise for integer instructions: they issue one-by-one and each of them takes the same time, whereas two floating point instructions can finish EX stage at the same clock cycle. In this case one of the instructions has to stall. The instruction which is stalled is the one with the smaller latency.

This was just a brief overview of DLX pipeline and we assumed that the user is familiar with the general concept of pipelining. For detailed discussion on this topic refer to Chapter 6 of [1].

2.3 The Main Simulation Loop

The core function of the pipeline Simulator is the function Pipe_Simulate contained in the file *pipe.c*. Here we discuss the main points of this function.

2.3.1 Data structures

The information about each pipeline stage is contained in the array machPtr->stages. This array has an entry corresponding to each stage and is indexed according to that (e.g. machPtr->stages[IF], machPtr->stages[ID], etc.). Elements of the array are the structures of the type InstrInExec. Here is the meaning of all the fields of this structure:

valid — Indicates if there is any instruction in this pipeline stage. If its value is YES, the corresponding stage is active at this clock cycle.

InstrPtr — The pointer to the structure of type MemWord representing the instruction itself.

rs1, rs2 — The values of the first and second operands of the instruction. These fields are being written at the ID stage. One can think about this as latches between register file and functional units.

stalled — Shows if the instruction should be stalled at this pipeline stage. If this value is YES, the corresponding pipeline stage is not active at the next clock cycle.

res_exists — If the value of this field is YES, the corresponding instruction produces some result, otherwise – not. Instructions which do not produce any results are stores, branches, jumps. This information is needed to indicate if anything should be written back at the WB stage or if there is any value to forward from here.

pc — The program counter of the instruction.

result — The result of the instruction (if any). This field is valid only for integer instructions.

FPPtr — If the instruction is a floating point one, all additional information needed about it, is contained here. It is the information about the floating point functional unit being used, the number of the clock cycle when the instruction completes, the result it produces.

nextPtr — Since the same structure is used for the linked list of pending floating point operations, this is the pointer to the next element of this list.

The list of the floating point instructions in the EX stage at any point of time is called Pipe_FPopsList and is a linked list of the structures described above.

Another data structure used for the pipeline management is res_for_bypass. This structure contains the information about the values to be bypassed and has the following fields:

rd — The register where the value will be eventually written.

resultType — The type of result to be bypassed (integer, single-precision floating point, double-precision floating point)

result The value to bypass

available Indicates if the value is available at the particular clock cycle. The value can be not available after the EX stage of any load operation.

stage Indicates from which pipeline stage the value comes.

The array to manage bypassing is machPtr–>bypass with the elements being pointers to structures of the type described above. The details on how bypassing is implemented can be found in later sections.

2.3.2 Integer Pipeline Simulation

Let us consider the Pipe_Simulation function. One execution of the while loop in this function is equivalent to one clock cycle of the simulated machine. At any clock cycle there can be up to five integer different instructions in the pipeline: one for each pipeline stage. For each of the stages the corresponding element of the array machPtr–>stages describes its state. When the new execution of the loop starts (i.e. simulation of a new clock cycle), each element contains the instruction that is **entering** this pipeline stage.

The stages are analized in the loop in the following order: Write Back, Memory Access, Execute, Instruction Decode and the last one is Instruction Fetch. This order was chosen because of the following reason: the instruction information needs to be passed from earlier stages in the pipeline to the later ones. There are two places in the loop where it can be done: at the end of the processing of each stage, or at the end of the whole loop for all of the stages. The first option is more convenient because it does not require storing of any additional information in the course of loop execution. So, when whatever needs to be done at the stage i is done, all the instruction information is copied in the variable describing the next stage. For better understanding of the following discussion about data flow in the main simulation loop, refer to **Figure 1.**

Everything needed to process each particular stage is contained in the corresponding structure machPtr–>stages[i]. There are common steps in the processing of each stage of the pipeline:

- If the valid field is NO (no instruction is entering this stage) do not do anything, otherwise, proceed.
- The actions which need to be done to simulate a certain stage itself are done (see the details later).
- If there can be values to forward from this stage, the corresponding structures are modified.
- The information is passed to the next pipeline stage.
- The entry for this stage is invalidated. It will become valid again if anything is passed from the earlier stages.

Events that happen on each pipe stage are close to the ones described on p. 256 of [1]. Here is a detailed description of what the simulator does at each pipe stage:

WB The value is getting written to the destination register (if any). If the instruction in this stage is TRAP, all the preparations are done to return the control to the operating system: flag trapCaught is set to true and the pipeline is cleared.

MEM In fact, nothing. All the values are being computed at the EX stage, so no DLX memory access is simulated here. This was done just to simplify the structure of the simulator. And it does not affect any results.

EX This is a very large switch statement where the results of all instructions are computed.

ID If the memory word representing the instruction is not compiled yet (this happens when the instruction is fetched for the first time), it is being compiled. After that, it is determined whether there are any values that need to be forwarded from the instructions in other pipe stages. If yes, the next thing to decide is whether the values are already available or not. Depending on that, either the new value is passed along with the instruction to the next stage or the pipeline is stalled.

Another problem which is dealt with during the ID stage, is execution of branches and jumps. Branch decisions are made at this stage and the program counter is modified here. Here arises one more case when the pipeline has to stall: if the value of the register to be checked as branch condition is being computed in the EX stage at the same clock cycle. Therefore, it is not available at the beginning of the cycle and the branch decision can not be made now.

One more assumption had to be made here to allow the new program counter to be written at this stage. For JALR (jump and link register) and JR (jump register) the pc is getting the value of one of the registers. So, the value has to be read and written at one clock cycle. We assume that it is possible to do so: read on the first half of the cycle and write on the second one.

IF If ID stage was stalled, this stage is stalled also; otherwise a new instruction is fetched from the DLX memory.

2.3.3 Bypassing

In order to make pipeline more efficient and to reduce the number of stalls *bypassing*, or *forwarding* is used. It works as follows: Before the result is written back to the register file, it can be fed back to the ALU if a later instruction needs it.

The structure used in the simulator to implement bypassing is an array machPtr->bypass (see **Figure 1**). It contains the information of all the values available for bypassing. At each execution of WB, MEM or EX stage the corresponding entries are modified. The indicator that a certain element of the array is a valid one is a non-zero value of the stage field. At the end of each cycle these values are set to zero. When a new cycle starts, for each of the stages WB, MEM, EX, if the stage is valid the corresponding entry of machPtr->bypass is validated and the fields are filled with the necessary values. Later, at the ID stage, after the instruction gets decoded (compiled), the rd (destination register) of all the elements of the bypass array, are compared to the source operands of the instruction. If they match, the next thing to check is whether the appropriate value is already available (to be precise, what is getting checked is whether the value **will be** available at the beginning of the next cycle). For all instructions except loads, the values are available right away. If the instruction following the load requires its result, this instruction has to stall.

2.3.4 Floating Point Pipeline

The execution of floating point instructions differs from that of integer ones. The major difference, from the point of view of pipeline implementation, is that the Execute pipe stage takes more than one clock cycle. This requires changes and additions to the general structure of the pipeline simulator. The modifications made to the simulator to manage floating point pipeline are as follows.

Function Pipe_FPIssue

When an instruction with floating point operands or result enters the ID stage of pipeline, the function Pipe_FPIssue is called. This function:

- checks for RAW and WAW hazards with the instructions already in execution;
- checks for the necessity of bypassing values and if the one exists – for the possibility of doing it at that clock cycle.;
- checks for structural hazards;

If everything needed for the instruction to start execution is available, this function returns 0 and writes the appropriate values in the FPPtr field of the information about instruction and the instruction is issued. If any hazard occurs, Pipe_FPIssue returns the number of stalls needed before the first hazard condition is ended.

Execution of Floating Point Instructions

The Execute pipe stage for floating point instruction takes more than one clock cycle. When the instruction enters this stage, its result is computed and it is inserted in the list of pending floating point operations Pipe_FPopsList. Besides, this list is checked to find out if any of the operations has completed by this time. If yes, it can be passed to the MEM stage. However, a problem can arise here. Only one instruction with floating point result can enter MEM stage at each clock cycle. So, if there is a one-cycle floating point instruction (e.g. move to floating point register) already in the EX stage, no instruction from the list can proceed. If there are more than one instruction in the list which finish at the same time, only one can proceed, the other ones have to stall. The instruction with the longest latency is given the priority here. At this point one of the hardware assumptions needs to be made also. The prpoblem is as follows: suppose, the result of, say, floating point add is stalled because floating point multiply has finished at the same time and has the priority to go further. Suppose also, that another floating point add is about to start its EX stage. The floating point adder has already finished its job for the first add: the result has been produced. However, it is still at the result latch and may stay there for some time (although more than one stall is very unlikely here, it is still possible under a certain execution pattern). The question is whether to allow the functional unit to start working on the second add or to make it wait until it gets rid of the first result. The decision has been made in favor of the second option. Although it may produce few extra stalls, the situation is very rear and it is not worth it to complicate the control logic in order to avoid these stalls.

MEM and WB for Floating Point Instructions

An assumption was made that there are separate register files for floating point and integer registers. This assumption allows to have two instructions in MEM or WB stage at the clock cycle, as long as these instructions reference different register files. So, we can say that there are two MEM stages — MEM and MEM_FP — and two WB stages — WB and WB_FP. There are entries in the array machPtr−>stages corresponding to the newly introduced stages: machPtr−>stages[MEM_FP] and machPTr−>stages[WB_FP]. These stages are analyzed in the fashion similar to that of the appropriate integer pipeline stages. The only difference is that a special function is called to write the results back: Pipe_WriteBack. This function writes the result back to a floating point register with the appropriate type casting.

Floating Point Loads

Another design decision which was to be made is about the status of floating point load: whether to consider it as integer or floating point instruction. If we follow the terminology of [1] and consider all loads, no matter integer or fp, to be integer instructions, fp load can overlap with some other floating point instruction in MEM and WB stage. This creates the problem when both instructions reach WB stage and try to write to floating point register file: only one instruction at a time can access the file. So, in fact, load of fp data being considered as integer instruction does not allow any overlapping with fp instructions in the later pipe stages. Another solution is to treat load of fp data as an fp instruction. The way pipeline is implemented now, it does not save anything versus the first solution: other fp instruction still have to stall if they finish the EX stage at the time the load does. However, if the simulator is changed so that MEM stage may take more than one clock cycle (e.g. cache miss), than the second solution allows overlapping of fp load with some integer instruction in case the fp load is stalled. So, our choice was to consider fp load as an fp instruction.

Bypassing Floating Point Values

The floating point results can also be forwarded to the earlier pipe stages if needed. The same machPtr−>bypass array is used for this. See **Figure 1** for the information on which element of bypass corresponds to the values forwarded from which stage. The necessity of floating point bypassing is checked in the FPIssue function and if the values are needed they are written to the latches. The same problem with the availability of load results and registers for branching arises here: sometime forwarding still can not prevent the necessity of stalls. This is one more case when Pipe_FPIssue may return non-zero value.

An "Extra" Pipeline Stage: DONE

There is one more "pipeline stage" which as can be seen at **Figure 1** and in the program code: DONE (DONE_FP). This is needed only for bookkeeping: to save the information about the instruction which has just finished execution and print out this information if requested by the user with **stats pipeline** command. The "stage" DONE_FP serves for the same purpose for the floating point instructions, since two instructions — one integer and one floating point — can finish execution at the same time.

Data Flow For Main Simulation Loop

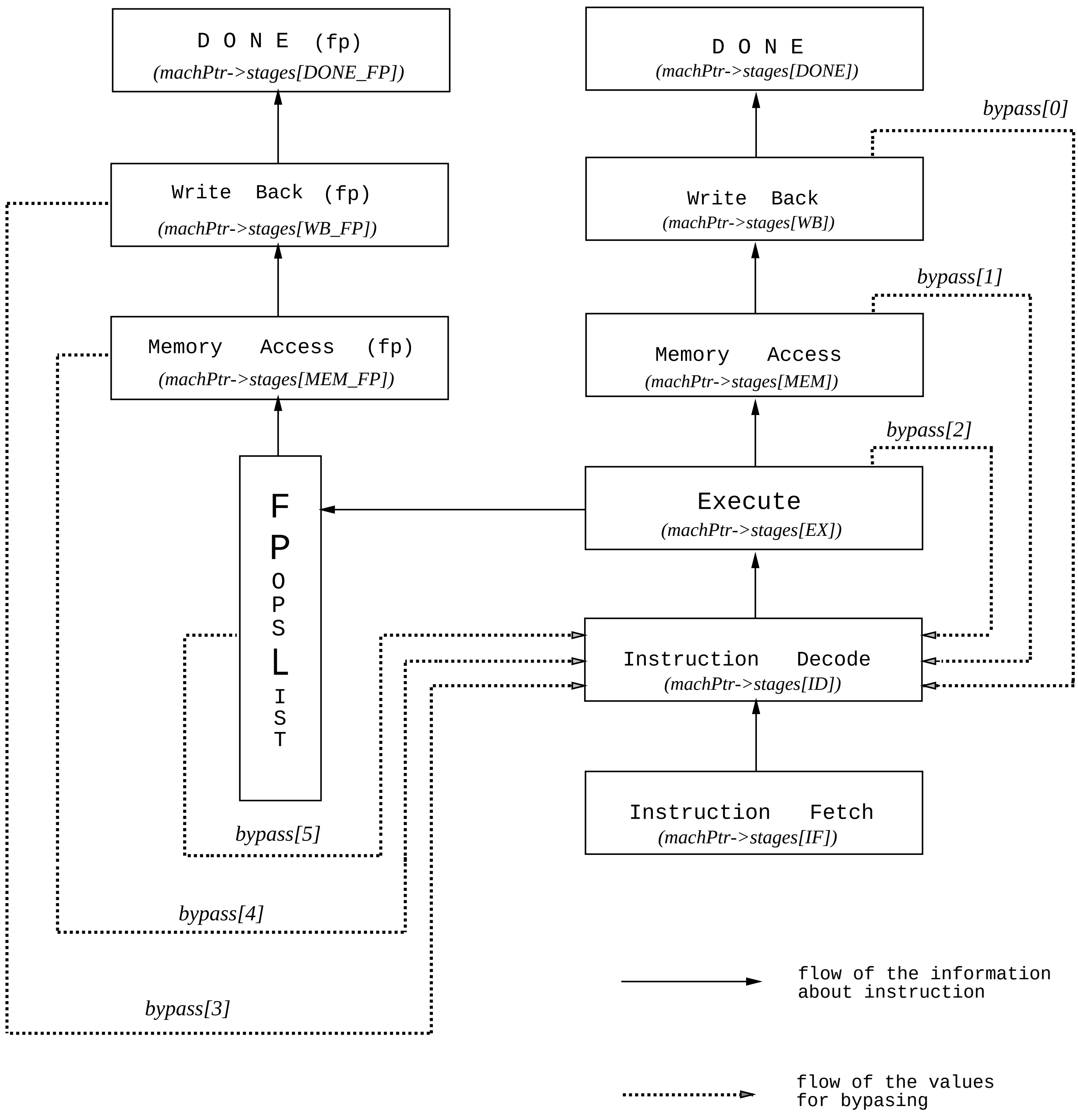

Figure 1

2.4 Additions to The Manual of DLXsim to Examine Pipeline Behavior

To make **DLXsim** execute its pipelined version one should call it with -PIPE option:

```
% dlxsim -PIPE
```

There is a number of changes and additions made to the user interface of **DLXsim** session introduced in the manual entry of [2]. This is their description:

step The meaning of this command is changed: it executes single clock cycle, not single instruction as in original **DLXsim** ;

stats New option is added to this command: **pipeline**. When this option is chosen, the complete information on all the pipeline stages is dumped by the simulator: which instruction is entering or is done with which stage; which stage is stalled, etc. For better understanding on how to interpret this data, read the next section of the report which gives an example of interactive session with **DLXsim**.

When **stats** is called with the **stalls** option, number of bypassed values is also reported for the pipelined version of the simulator.

clear This is a new instruction which exists only in pipelined version of **DLXsim**. The usage is:

```
(dlxsim) clear pipeline
```

It deactivates all pipeline stages and the pipeline becomes empty as in the beginning of the simulation. This might be useful if you want, for example, to start execution of a piece of code as if nothing was executed before it.

2.5 An Example of Interactive Session with DLXsim

Here is an example of interactive session with the pipelined version of **DLXsim** with the focus on examining pipeline behavior. The program to run is contained in the file *example.s* and its text follows here:

```
        .global A
A:      .double 2
        .global B
B:      .double 1
        .global C
C:      .double 17

        .align  4
        .global _main
_main:
        addi    r1, r0, A       ; store address of vector A in r1
        addi    r2, r0, B       ; store address of vector B in r2
        addi    r3, r0, C       ; store address of vector C in r3
        addi    r4, r0, #10     ; store the value of x in f0
        movi2fp f0, r4
        cvti2d  f0, f0
        ld      f2, 0(r1)       ;load A
        ld      f4, 0(r2)       ;load B
        addd    f2, f2, f4      ;A + B
        ld      f6, 0(r3)       ;load C
        multd   f6, f0, f6      ;x * C
        subd    f2, f2, f6      ;A + B - x * C
        sd      0(r1), f2       ;store into A
        trap    #0
```

This program loads three numbers, does some calculations on them and then stores the result. We will show the execution of this code on the pipelined machine. To do that, **DLXsim** should be called with **-PIPE** extension:

```
% dlxsim -PIPE
(dlxsim)
```

Now we can load the program in the DLX memory:

```
(dlxsim) load example.s
```

To get the information about the state of the pipeline at any point, one should type **stats pipeline** (statistics on the pipeline). At the beginning the pipeline is empty:

```
(dlxsim) stats pipeline
```

```
                              PIPELINE STATE

      no instruction                                  starting      WB
      no instruction                                  starting      MEM
      no instruction                                  starting      EX
      no instruction                                  starting      ID
      no instruction                                  starting      IF
```

The abbreviations are the standard abbreviations adopted in [1]:

IF Instruction Fetch

ID Instruction Decode

EX Execute

MEM Memory Access

WB Write Back

Here is the pipeline state after the first step (the step here is always one clock cycle).

```
(dlxsim) step _main
stopped after single step, pc = _main+0x4: addi r2,r0,0x108
(dlxsim) stats pipeline

                              PIPELINE STATE

      no instruction                                  starting      WB
      no instruction                                  starting      MEM
      no instruction                                  starting      EX
    addi r1,r0,0x100    done with      IF      and starting      ID
    addi r2,r0,0x108                                  starting      IF
```

So, during this clock cycle the first instruction went through its IF stage and will be in ID at the next cycle. The last line shows the instruction which is about to be fetched.

Now we can proceed for few more steps while the instructions proceed through the pipeline.

```
(dlxsim) step
stopped after single step, pc = _main+0x8: addi r3,r0,0x110
(dlxsim) step
stopped after single step, pc = _main+0xc: addi r4,r0,0xa
(dlxsim) step
stopped after single step, pc = _main+0x10: movi2fp f0,r4
(dlxsim) step
stopped after single step, pc = _main+0x14: cvti2d f0,f0
(dlxsim) stats pipeline
```

```
                              PIPELINE STATE

    addi r1,r0,0x100    done with      WB
    addi r2,r0,0x108    done with      MEM     and starting        WB
    addi r3,r0,0x110    done with      EX      and starting        MEM
      addi r4,r0,0xa    done with      ID      and starting        EX
       movi2fp f0,r4    done with      IF      and starting        ID
        cvti2d f0,f0                               starting        IF
(dlxsim) stats stalls
Integer Pipeline Stalls = 0
Floating Point Stalls = 0
Number of bypassed values = 1
```

All the pipeline stages were busy at the last clock cycle. The first instruction of the program has just completed its execution. Everything was "smooth" so far: no stalls; new instruction was fetch at each step. However, there was a necessity to forward one value: the **move** instruction needs the result of the instruction which is right before it. The result of **add** is available at the end of its EX stage, so **move** does not have to stall.

```
(dlxsim) step
stopped after single step, pc = 0x0: ld f2,0x0(r1)
(dlxsim) step
stopped after single step, pc = 0x0: ld f4,0x0(r2)
(dlxsim) step
stopped after single step, pc = _main+0x20: addd f2,f2,f4
(dlxsim) step
stopped after single step, pc = 0x0: ld f6,0x0(r3)
(dlxsim) stats pipeline

                              PIPELINE STATE

       movi2fp f0,r4    done with      WB (fp)
      no instruction                               starting        WB
        cvti2d f0,f0    done with      MEM(fp) and starting        WB (fp)
      no instruction                               starting        MEM
       ld f2,0x0(r1)    done with      EX (fp) and starting        MEM(fp)
       ld f4,0x0(r2)    done with      ID      and starting        EX
       addd f2,f2,f4    done with      IF      and starting        ID
       ld f6,0x0(r3)                               starting        IF
```

Here we had few floating point instructions. All of them were able to get there source operands without stalling, however, bypassing was necessary (for **load** and **convert** instructions). One can see here separate pipeline stages for floating point instructions. Information is always given on all integer pipeline stages (even if there is no instruction executing that stage), but for floating point stages only the active ones are mentioned.

One can notice that a pipeline stall will occur at the next step: **add** cant start its execution because the result from the second **load** is not ready yet: it will be ready only after MEM stage.

```
(dlxsim) step
stopped after single step, pc = 0x0: ld f6,0x0(r3)
(dlxsim) stats pipeline

                                PIPELINE STATE

       cvti2d f0,f0    done with      WB (fp)
      no instruction                              starting        WB
       ld f2,0x0(r1)    done with      MEM(fp) and starting        WB (fp)
      no instruction                              starting        MEM
       ld f4,0x0(r2)    done with      EX (fp) and starting        MEM(fp)
      no instruction                              starting        EX
       addd f2,f2,f4    done with      ID      and stalled before  EX
       ld f6,0x0(r3)    done with      IF      and stalled before  ID
```

As it was expected the pipeline is stalled. The **add** can proceed only now.

```
(dlxsim) step
stopped after single step, pc = _main+0x28: multd f6,f0,f6
(dlxsim) step
stopped after single step, pc = _main+0x2c: subd f2,f2,f6
(dlxsim) stats pipeline

                                PIPELINE STATE

       ld f4,0x0(r2)    done with      WB (fp)
      no instruction                              starting        WB
      no instruction                              starting        MEM
       addd f2,f2,f4    will complete  EX  in 1 cycle(s)
       ld f6,0x0(r3)    done with      ID      and starting        EX
      multd f6,f0,f6    done with      IF      and starting        ID
       subd f2,f2,f6                              starting        IF
```

Here is the situation when there are more than one instruction in the EX stage of the pipeline: there are few floating point functional units, so few instruction can execute at the same time. For floating point **add** it takes more than one cycle to execute, so it is still in the EX stage.

After this cycle two instruction will be done with EX. Both are floating point. So, only one of them can enter MEM stage. The other one has to stall:

```
(dlxsim) step
stopped after single step, pc = _main+0x2c: subd f2,f2,f6
(dlxsim) stats pipeline
```

```
                         PIPELINE STATE

   no instruction                                starting       WB
   no instruction                                starting       MEM
    ld f6,0x0(r3)    done with    EX (fp) and starting       MEM(fp)
    addd f2,f2,f4    done with    EX (fp) and stalled before MEM(fp)
   no instruction                                starting       EX
   multd f6,f0,f6    done with    ID      and stalled before EX
    subd f2,f2,f6    done with    IF      and stalled before ID
```

The execution will proceed in the same fashion more or less until a trap is received:

```
(dlxsim) go
TRAP #0 received
(dlxsim) stats pipeline

                         PIPELINE STATE

   no instruction                                starting       WB
   no instruction                                starting       MEM
   no instruction                                starting       EX
   no instruction                                starting       ID
   no instruction                                starting       IF


(dlxsim) fget A d
A:      19.000000
```

So, the pipeline is empty now and the value of A has been modified.

Exercises

Exercise 1.

This exercise will help you to understand the reasons of pipeline stalls. Consider the following code (it is contained in the file *ex1.data*):

```
        .global A
A:      .double 10
        .global B
B:      .double 5
        .global C
C:      .double 2
        .global D
D:      .double 17



        .align  4
        .global _main
_main:
        addi    r1, r0, A
        addi    r2, r0, B
        addi    r3, r0, C
        addi    r4, r0, D
foo:
        ld      f0, 0(r1)
        ld      f2, 0(r2)
        multd   f2, f2, f0
        ld      f4, 0(r3)
        ld      f6, 0(r4)
        addd    f4, f4, f0
        subd    f6, f6, f0
        multd   f2, f2, f6
        multd   f4, f4, f2
        sd      0(r3), f4
        addi    r5, r0, #0
        bnez    r5, foo
        nop
        trap    #0
```

This program does simple calculations. Which? What are the pipeline stalls occurring here? Trace the execution of the program using the simulator and explain the reasons for all the stalls in it.

Exercise 2.

Do the exercise 6.16 from [1]. You may use the **DLXsim** to collect the data and to try various ordering of instructions to maximize performance.

Bibliography

[1] John L. hennessy & David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc. San Mateo, CA

[2] Larry B. Hostetler & Brian Mirtich. *DLXsim — A Simulator for DLX*