

UNIVERSITY OF ENERGY AND NATURAL RESOURCES

**COMPUTER ARCHITECTURE LAB REPORT 3**

**AGYEMAN JOSEPH YAW**

**UE20023617**

DEPARTMENT OF COMPUTER AND ELECTRICAL ENGINEERING

PROGRAMME: BSC. COMPUTER ENGINEERING

INSTRUCTOR: Mr. Andrews Quansah

DATE OF SUBMISSION:5th April, 2021

# 

Table of contents

[Table of contents 2](#_Toc528881465)

[Abstract 3](#_Toc528881466)

[Introduction 3](#_Toc528881466)

[Study Methodology 4](#_Toc528881467)

[Analysis to Project 4](#_Toc528881468)

[Approach to Project 5](#_Toc528881469)

[Results 6](#_Toc528881470)

Test [Results 6](#_Toc528881470)

[References 7](#_Toc528881471)

[Conclusion 27](#_Toc528881487)

**Abstract**

Using an interactive, visual pipeline simulator, called dlxview, for the DLX instruction set and pipeline described in Computer Architecture A Quantitative Approach by Hennessy and Patterson [1]. This software provides animated versions of key figures and tables from the text and allows the user to readily follow details of pipeline activity as a code is simulated, to vary pipeline

implementation, and to compare performance across different pipeline designs.

The software package requires a system running Unix and X11, with Tcl/Tk installed, and using the GNU gcc compiler is recommended.

# 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](https://www.cs.bu.edu/~best/crs/cs550/fridman/pdlx/report-pipe.html#book)]). The instructions execution in the extended DLXsimis 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](https://www.cs.bu.edu/~best/crs/cs550/fridman/pdlx/report-pipe.html#book)]. It also gives additional statistics values as compared to the ones described in [[2](https://www.cs.bu.edu/~best/crs/cs550/fridman/pdlx/report-pipe.html#report)]. 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 interactive session with DLXsimwith 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 in understanding what pipeline is, what problems it may cause and how to deal with them.

**Study Methodology**

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.

## 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 analysed 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 in [[1](https://www.cs.bu.edu/~best/crs/cs550/fridman/pdlx/report-pipe.html#book)]. 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.

## 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.

## Floating Point Pipeline

The execution of floating point instructions differ 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 problem 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 favour 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.

### 3.4.3 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.

### 3.4.4 Floating Point Loads

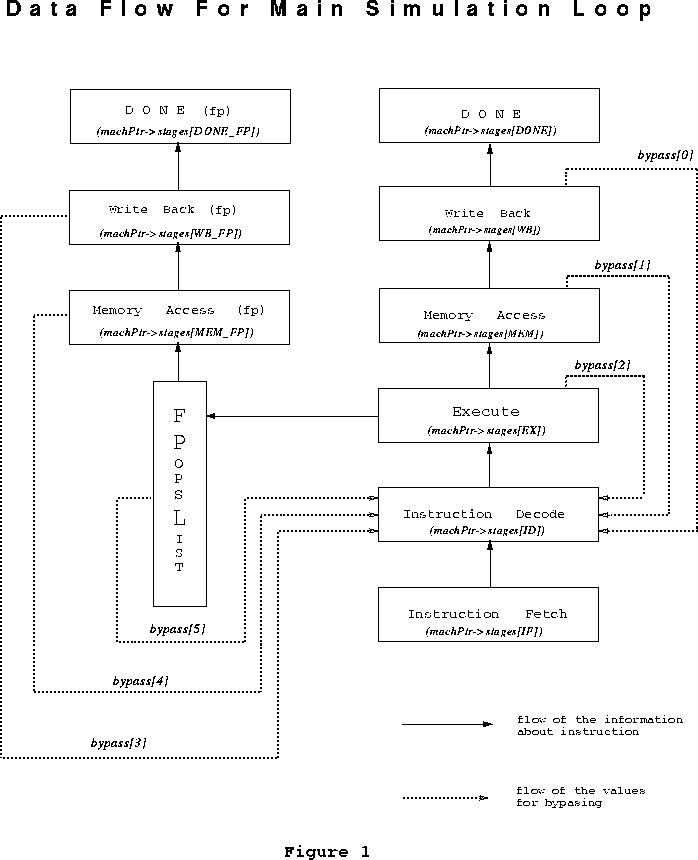
Another design decision which was to be made is about the status of floating point load: whether to consider it as integer or floating point instruction. If we follow the terminology of [[1](https://www.cs.bu.edu/~best/crs/cs550/fridman/pdlx/report-pipe.html#book)] and consider all loads, no matter integer 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.



# rESULTS

Here is the interactive session with the pipelined version of DLXsimwith the focus on examining pipeline behaviour.

.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, DLXsimshould be called with **-**PIPE extension:

% dlxsim -PIPE

(dlxsim)

Now we can load the program in the DLX memory:

(dlxsim) load “file”

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.

**Test Results**

Test Results are placed in a same directory with name tests.

**Conclusion**

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 interactive session with DLXsimwith the focus on examining pipeline features.

**References**

[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