**Department of Electrical Engineering**

|  |  |
| --- | --- |
| **Faculty Member: Dr. Nasir Mehmood** | **Dated: 18/05/2018** |
|  |  |
| **Course/Section: BSCS-6A**  **GROUP# 02** | **Semester: 4th** |

**Computer Organization and**

**Achitecture (EE321)**

**Lab #12 Pipelining**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Reg. no.** | **Report Marks / 10** | **Viva Marks / 5** | **Total/15** |
| **ABUL Razzaque Jawad** | **191408** |  |  |  |
| **Abdullah Rafaqat** | **146905** |  |  |  |
| **Awais Latif Khatti** | **199189** |  |  |  |
|  |  |  |  |  |

## Lab 12: Pipelining

**Objectives**

At the end of this lab you should be able to:

Demonstrate the difference between pipelined and sequential processing of the CPU instructions

Explain pipeline data dependency and data hazard

Describe a pipeline technique to eliminate data hazards

Demonstrate compiler “loop unrolling” optimization’s benefits for instruction pipelining

Describe compiler re‐arranging instructions to minimize data dependencies

Show the use of jump‐predict table for pipeline optimisation

**Lab Instructions**

* This lab activity comprises two parts, namely Lab tasks, and Post-Lab Viva session.
* Each group to upload completed lab report on LMS for grading.
* The students will start lab task and demonstrate each lab task separately for step-wise evaluation
* There are related questions in this activity give complete answers. Also provide complete code and results.

**Investigating** **Instruction** **Pipelines** **Introduction**

Modern CPUs incorporate instruction pipelines which are able to process different stages of multiple‐stage instructions in parallel thus improving the overall performance of the CPUs. However, most programs include instructions that do not readily lend themselves to smooth pipelining thus causing pipeline hazards and effectively reducing the CPU performance. As a result CPU pipelines are designed with some tricks up their sleeves for dealing with these hazards.

**Lab** **Exercises** **‐** **Investigate** **and** **Explore**

The lab investigations are a series of exercises that are designed to demonstrate the various aspects of CPU instruction pipelining.

**Exercise** **5** **–** **Difference** **between** **the** **sequential** **and** **the** **pipelined** **execution** **of** **CPU** **instructions**

Enter the following source code, compile it and load in simulator’s memory:

**program Ex1**

**for n = 1 to 26**

**p = p + 1**

**next**

**end**

Open the CPU pipeline window by clicking on the **SHOW** **PIPELINE…** button in the CPU simulator’s window. You should now see the **Instruction** **Pipeline** window. This window simulates the behaviour of a CPU pipeline. Here we can observe the different stages of the pipeline as program instructions are processed. This pipeline has five stages. The stages are colour‐coded as shown in the key for the “Pipeline Stages”.

List the names of the stages here:

Instruction Fetch, Instruction decode, Read operands, Execute, Write Result

The instructions that are being pipelined are listed on the left side (in white text boxes). The newest instruction in the pipeline is at the bottom and the oldest at the top. You’ll see this when you run the instructions. The horizontal yellowish boxes display the stages of an instruction as it progresses through the pipeline. At the bottom left corner pipeline statistics are displayed as the instructions are executed.

Check the box titled **Stay** **on** **top** and make sure **No** **instruction** **pipeline** check box is selected. In the CPU simulator window bring the speed slider down to around a reading of 30. Run the program and observe the pipeline. Wait for the program to complete. Now make a note of the following values

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 5.01 |
| SF (Speed‐up Factor) | 1 |

Next, uncheck the **No** **instruction** **pipeline** checkbox, reset and run the above program again and wait for it to complete.

Note down your observation on how the pipeline visually behaved differently

The execution of the instructions are now done in parallel. For non-pipelined, the operations for instruction is performed when its previous makes it whole executed. Now, the instructions are executed in parallel by stages, when particular stage of instruction is being performed then the next instruction’s previous stage is being operated and so on. For this case (pipelined), CPI is decreased and speed factor is increased.

Now once again make a note of the following values

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 1.9 |
| SF (Speed‐up Factor) | 2.63 |

Briefly explain why you think there is a difference in the two sets of values:

For pipelined instructions, the clocks per instructions is ideally decreased to 1/5 as number of stages that are divided are also 5. In pipelined all the resources are utilized all the time and the instructions are being performed by stages in parallel. Since the CPI is decreased, so the speed up factor is increased because of its faster execution.

**Exercise** **6** **–** **CPU** **pipeline** **data** **hazards,** **bubbles** **and** **the** **NOP** **instruction**

CPU pipelines often have to deal with various hazards, i.e. those aspects of the CPU architecture which prevent the pipelines running smoothly and uninterrupted. These are often called “pipeline hazards”. One such hazard is called the “data hazard”. A data hazard is caused by unavailability of an operand value when it is needed. In order to demonstrate this create a program (call it Ex2) and enter the following set of instructions

**MOV** **#1,** **R01**

**MOV** **#5,** **R03**

**MOV** **#3,** **R01**

**MOV** **#7,** **R02**

**MOV #9, R04**

**ADD** **R01,R03**

**ADD R04,R01**

**ADD R03,R02**

**HLT**

Before you carry on, make a note of what value you expect to see in register R03 at the end of the above set of instructions. Make a note of it below:

R03 = **08**

Make sure the **No** **instruction** **pipeline** is NOT checked and **Do** **not** **insert** **bubbles** is checked. Reset the program and run the above instructions. Make a note of the value in register R03 below:

R03 = **08**

Now insert a NOP instruction (use the **Miscellaneous** tab) after the third instruction, i.e. you end up with the following modified set of instructions

**MOV** **#1,** **R01**

**MOV** **#5,** **R03**

**MOV** **#3,** **R01**

**NOP**

**ADD** **R01,** **R03**

**HLT**

Reset the program and run the above set of instructions. Observe the value in register R03 when the program completes. Make a note of this value below

R03 = **08**

You have now recorded three instances of the values of the register R03 the first one being your prediction. Briefly comment on your observations of the three values:

The instructions are executed in proper order. Whenever a particular operand for operation is needed till then the operand values are fully available from the registers and no any such hazard or incorrect value is observed.

In the first case, the instructions were sequential so we predicted right as 8, then similar instructions run in parallel as no hazard is expected so the register value is also 8, at last, a NOP instruction is used that utilize a cycle of execution but performs no operations. So, same value of 8 is observed.

Now delete the NOP instruction from above program and uncheck the option **Do** **not** **insert** **bubbles**. Reset the program and run the instructions. Observe the value in register R03 when the program completes. Make a note of this value below:

R03 = **08**

The value above should be the same as the one when you inserted a NOP instruction in the program. However this value is obtained without the NOP instruction in this case. Briefly explain:

NOP instruction does nothing and just takes instruction cycle. So, by having or not this instruction does not affect the result and it is same as of the above cases.

Have you seen the “bubble”? What colour is it?

Yes. Pink color bubble (Instruction Sequence Synch) is observed at HLT instruction.

Finally, make a note of the following values:

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 2.11 |
| SF (Speed‐up Factor) | 2.37 |
| Data Hazards | 0 |

**Exercise** **7** **–** **A** **pipeline** **technique** **to** **eliminate** **data** **hazards**

One way of dealing with “data hazard” is to get the CPU to “speed up” the availability of operands to pipelined instructions. One such method is called “operand forwarding”, a kind of short‐cut by the hardware. To demonstrate this check the box titled **Enable** **operand** **forwarding** and run the above code again.

The bubble is disappeared because now forwarding is enabled which eliminate the data hazards for this case.

The simulator keeps a count of the pipeline hazards it detects as the instructions go through the pipeline. These can be seen near the bottom of the pipeline window.

Make a note of the following values

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 2.6 |
| SF (Speed‐up Factor) | 1.92 |
| Data Hazards | 0 |

Has there been an improvement?

Yes, because the enabling of forwarding for this case has removed the data hazard.

**Exercise** **8** **–** **Loop** **unrolling** **optimization** **minimizing** **control** **dependencies**

In a previous tutorial on compiler optimizations, we looked at one method of optimization called “loop unrolling”. This method essentially duplicates the inner code of a loop as many times as the number of loops, removing some redundant code as well as the loop’s compare and jump instructions. However, the code size of the program increases. It is shown that “loop unrolling” is well suited to instruction pipelining which takes full advantage of it thus improving CPU performance. Here, we will prove this to be the case.

Enter the following code, select optimization option **Redundant** **Code** and compile it.

**program** **Ex4\_1**

**for** **n** **=** **1** **to** **8**

**t** **=** **t** **+** **1**

**next**

**end**

Make a note of the size of the code generated for Ex4\_1 here:

Code has 7 instructions.

Now, load this code in CPU simulator’s memory.

Next, make sure the optimization option **Loop** **Unrolling** is selected in addition to the option **Redundant** **Code** optimization. Change the program name to Ex4\_2 and compile it again. Load this code in memory too. So, now you should have two versions of the code: Ex4\_1 without “loop unrolling” optimization and Ex4\_2 with “loop unrolling” optimization.

Make a note of the size of the code generated for Ex4\_2 here:

Code has 19 instructions.

Make sure the pipeline window stays on top. Also make sure the **Enable** **operand** **forwarding** and **Enable** **jump** **prediction** boxes are all unchecked. First, select program Ex4\_1 from the **PROGRAM** **LIST** frame in the CPU simulator window then click the **RESET** button. Make sure the speed of simulation is set at maximum. Now click the **RUN** button to run program Ex4\_1. Observe the pipeline and when the program is finished make a note of the following values:

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 1.74 |
| SF (Speed‐up Factor) | 2.87 |
| No of instructions executed | 44 |

Do the same with program Ex4\_2 and make note of the following values:

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 1.84 |
| SF (Speed‐up Factor) | 2.73 |
| No of instructions executed | 19 |

Briefly comment on your observations making references to the code sizes and the number of instructions executed:

The code in the first case has size 7 and 44 instructions executed therefore we see that the code block is executed multiple times.

The code in the second case has same size and instructions executed that are 19, therefore each instruction is executed only once.

**Exercise** **9** **–** **Compiler** **re‐arranging** **instruction** **sequence** **to** **help** **minimize** **data** **dependencies**

The optimization in Exercise 4 is one example of how a modern compiler can provide support for the CPU pipeline. Another example is when the compiler re‐arranges the code without changing the logic of the code. This is done to minimize pipeline hazards such as the “data hazard” we studied in Exercise 3. Here we demonstrate this technique.

Make sure **Show** **dependencies** check box is checked and ONLY the **Redundant** **Code** optimization is selected. Enter the following source code, compile it and load in memory

**program** **Ex5\_1**

**a** **=** **1**

**b** **=** **a**

**c** **=** **2**

**end**

Copy the CPU instruction sequence generated below (do not include the instruction addresses):

MOV #1, R01

MOV R01, R02

MOV #2, R03

HLT

Next, select the optimization option **Code** **Dependencies**. Change the program name to Ex5\_2, compile it and load in memory.

Copy the CPU instruction sequence generated below:

MOV #1, R01

MOV #2, R03

MOV R01, R02

HLT

How do the two sequences differ? Does the change affect the logic of the program? Briefly explain the rationale for the change:

Program remains the same however two instructions are changed their positions. These instructions are independent of each other hence not affecting the purpose of the code. The use of this change is that the third instruction is requiring the data from the first instruction in the second program. So moving an instruction between them remove the data hazard as the instruction is fully executed before the data is needed.

let’s see if we can measure the improvement with this “out of sequence execution” method applied. Modify the above program as below:

**program** **Ex5\_3**

**for** **n** **=** **1** **to** **50**

**a** **=** **1**

**b** **=** **a**

**c** **=** **2**

**next**

**end**

Now, compile and load two version of the above program, one without the **Code** **Dependencies** optimization and one with this optimization (call this one program Ex5\_4).

First run program Ex5\_3 and make note of the values below:

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 1.74 |
| SF (Speed‐up Factor) | 2.87 |

Next, run program Ex5\_4 and make note of the values below:

|  |  |
| --- | --- |
| CPI (Clocks Per Instruction) | 1.6 |
| SF (Speed‐up Factor) | 3.12 |

Do you see any improvement in program Ex5\_4 over program Ex5\_3 (express this in percentage)?

Improvement is observed in both CPI and speed up factor. Overall the execution time is reduced by: 91%.

The CPU pipeline uses a table to keep a record of the predicted jump addresses. So, whenever a conditional jump instruction is being executed this table is consulted in order to see what the jump address is predicted as. If this prediction is wrong then the calculated address is used instead. Often the predicted address will be correct with occasional wrong prediction. However, the overall effect will be an improvement on CPU’s performance.