Investigating Instruction Pipelines

Introduction

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

Basic Theory

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 1 – 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 20
        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:

| |
| |

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)	
SF (Speed-up Factor)	

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

| |
| |

Now once again make a note of the following values

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	

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

Exercise 2 – CPU pipeline data hazard and bubble

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
ADD R01, R03
HLT
```

Reset the program and run the above instructions.

Have you seen the "bubble"? What colour is it?

Make a note of the following values

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	
Data Hazards	

Exercise 3 – 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.

Has the bubble seen in Exercise 2 disappeared (or burst!)?

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)	
SF (Speed-up Factor)	
Data Hazards	

Has there been an improvement?

Exercise 4 – 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:

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:

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)	
SF (Speed-up Factor)	
No of instructions executed	

Do the same with program Ex4_2 and make note of the following values:

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	
No of instructions executed	

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

Exercise 5 – 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):

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:

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

Let's see if we can measure any improvement introduced by this "out of sequence execution" method.

First reset and run program Ex5_1 and make note of the values below:

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	

Next, reset and run program Ex5_2 and make note of the values below:

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	

Do you see any improvement in program Ex5_2 over program Ex5_1 (express this in percentage)?

Exercise 6 – Jump predict table

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.

Enter the following program and compile it with <u>ONLY</u> the **Enable optimizer** and **Remove redundant code** check boxes selected. Load the compiled program in the CPU.

```
program Ex6
    i = 0
    for p = 1 to 40
        i = i + 1
        if i = 10 then
            i = 0
            r = i
        end if
    next
end
```

Run the program and make a note of the following pipeline stats:

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	

Now, in the pipeline window select the **Enable jump prediction** check box. Reset the program and run it again. Make a note of the following pipeline stats:

CPI (Clocks Per Instruction)	
SF (Speed-up Factor)	

Do you see a difference? Is it an improvement?

Click on the **SHOW JUMP TABLE…** button. You should see the **Jump Predict Table** window showing. This table keeps an entry relevant to each conditional jump instruction. The information contained has the following fields. Can you suggest what each field stands for? Enter your suggestions in the table below:

V	
JInstAddr	
JTarget	
PStat	
Count	