Understanding Instruction Pipelines

Introduction

Objectives

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

- Configure the simulator for pipeline simulations
- Use the simulator to observe and identify pipeline hazards
- Use the simulator to apply methods of eliminating hazards
- Demonstrate “loop unrolling” optimization’s benefits for instruction pipelining

Processor (CPU) Simulators

The computer architecture tutorials are supported by simulators, which are created to underpin theoretical concepts normally covered during the lectures. The simulators provide visual and animated representation of mechanisms involved and enable the students to observe the hidden inner workings of systems, which would be difficult or impossible to do otherwise. The added advantage of using simulators is that they allow the students to experiment and explore different technological aspects of systems without having to install and configure the real systems.

Basic Theory

Modern CPUs incorporate instruction pipelines which are able to process different stages of multi-stage instructions in parallel thus improving the overall performance of the CPUs. However, most programs have instructions which 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.

Simulator Details

This section includes some basic information on the simulator, which should enable the students to use the simulator. The tutor(s) will be available to help anyone experiencing difficulty in using the simulator.

The simulator for this lab is an application running on a PC and is composed of multiple windows.

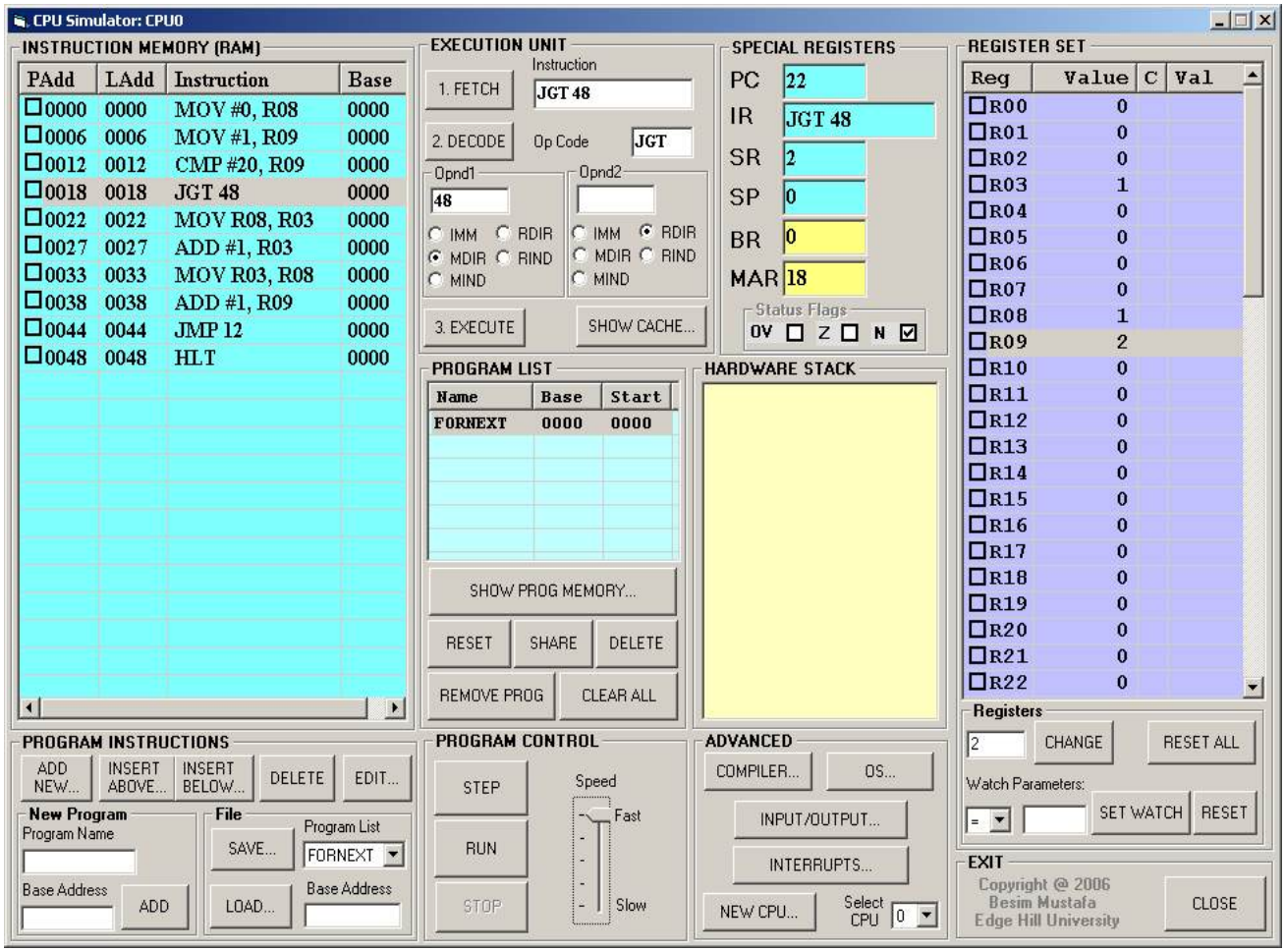

Image 1 - Main simulator window

The main window shown in Image 1 is composed of several sub-views, which represent different functional parts of the simulated processor. For this lab session we are interested only in the compiler part of the simulator.

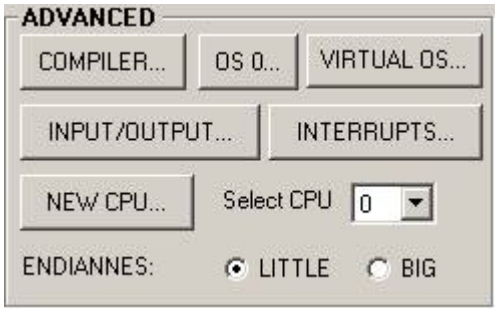

Image 2 - Advanced functions

In order to access the compiler, click on the **COMPILER…** button as shown in Image 2 on the right. The compiler window shown in Image 3 below will show.

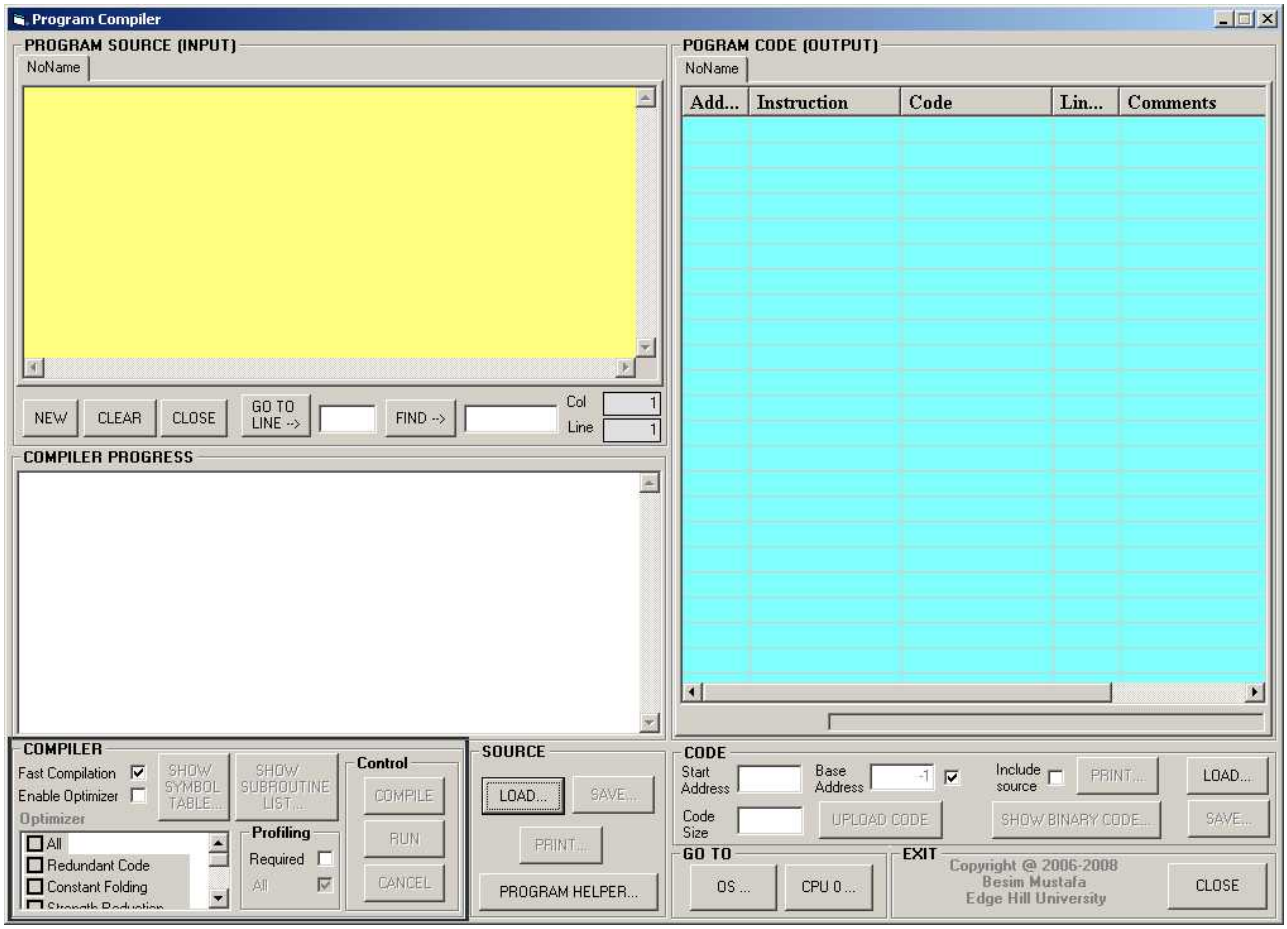

Image 3 - The main compiler window

In the compiler window there are three main sub-windows

- **Program Source** - all high-level source statements appear here.
- **Compiler Progress** - information on the progress of a compilation appear here.
- **Program Code** - assembly code generated by the compiler appear here.

Lab Exercises - Investigate and Explore

The lab exercises are a series of experiments, which are attempted by the students under guidelines. The students are expected to carry out further investigations on their own in order to form a better understanding of the technology.

Now, have a go at the following activities:

1. First, enter the following source code

```
program Test1
    for n = 1 to 20
        p = p + 1
    next
end
```

Compile the above code by clicking on the **COMPILE…** button in the **SOURCE** frame and load the compiled into memory using the **LOAD IN MEMORY** button.

Now, open the CPU pipeline window by clicking on the **SHOW PIPELINE…** button in the CPU simulator 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.

Carefully study the contents of this window and try to answer the following questions. Do not proceed until you answer all questions.

- How many stages does this CPU pipeline have?
- List the names of the stages here.

The pipelined instructions are listed on the left side (in white boxes). The instructions are listed from top to bottom. The latest instruction 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 goes through the pipeline. The stages are colour-coded to aid visualisation. At the bottom left corner, some data on the pipeline are displayed as the instructions are executed. Now we are ready to use the pipeline.

Check the boxes titled **Stay on top** and **No instruction pipeline**. In the CPU simulator window bring the speed slider down to around a reading of 30. Now 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 and run the above program again and wait for it to complete.

- How does the pipeline behave differently (visually)?

Make a note of the following values

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

- Can you explain the differences in the two sets of values?

2. Let's look at the pipeline a bit closer. Enter the following set of instructions directly in the CPU instruction memory.

```
MOV #3, R01
MOV R00, R02
MOV #8, R04
HLT
```

Make sure the pipeline window stays at top. Check the **No instruction pipeline** checkbox. Click the **STEP** button and observe the pipeline.

- Look at the result and briefly explain what is happening

Also observe how the used instructions are displayed in this window. Now, uncheck the **No instruction pipeline** checkbox and check the **Allow out of sequence** checkbox. Use the **STEP** button to step through all above the instructions one by one.

- Look at the result and briefly explain what is happening
- How does it differ from the previous result?

3. CPU pipelines often have to deal with various hazards, i.e. those aspects of CPU architecture which prevent the pipelines running uninterrupted. These are often called "pipeline bubbles" for the reasons you'll see. 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 enter the following set of instructions

```
MOV #2, R01
ADD #1, R01
HLT
```

Make sure the **No instruction pipeline** is not checked and the **Allow out of sequence** checkbox is checked. Now run the above set of instructions.

- Have you observed a bubble? What colour is it?
- Look at the code above and try to explain the bubble

One way of dealing with this type of 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. Check the box titled **Enable operand forwarding** and run the above code again.

- Has the bubble 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.

4. Another pipeline hazard is caused by the control instructions which change the control flow of a series of instructions. This is known as the "control hazard". One such control instruction is the jump instruction when it changes the direction of flow as in loops. A jump instruction cannot decide which way to go until the previous compare instruction is executed thus creating a bubble in the pipeline. To demonstrate this, enter the following set of instructions. The numbers on the left are addresses.

```
0000 MOV #1, R03
0006 CMP #1, R03
0012 JNE 20
0016 JMP 0
0020 HLT
```

Make the pipeline window stay on top. Also make sure the **Allow out of sequence** checkbox is unchecked. Now, click the **STEP** button 3 times one after the other and stop. You should see a bubble on the instruction "JNE 20".

- Have you observed a bubble? What colour is it?
- Look at the code above and try to explain the bubble

One way of dealing with this type of hazard is to get the CPU to "predict" the direction in which the flow of execution should proceed. One such method is called "jump prediction". Check the box titled **Enable jump prediction** and run the above code again.

- Has the bubble disappeared?

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.

5. 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, including the loop's jump instruction, as a result. However, the code size of the program increases. It is shown that "loop unrolling" is well suited to instruction pipelining and takes full advantage of it thus improving CPU performance. Here, we will prove this to be the case.

Enter the following code and compile it making sure the optimization **redundant code** is selected.

```
program Test2
        for n = 1 to 8
              t = t + 1
        next
end
```

Load this code in memory.

Next, make sure the optimization option **loop unrolling** is selected in addition to the **redundant code** optimization option. Change the program name to Test3 and compile it again. Load this code in memory too. So, now you should have two versions of the code, one with (i.e. Test2) and one without (i.e. Test3) “loop unrolling” optimization.

Make sure the pipeline window stays on top. Also make sure the **operand forwarding**, **jump prediction** and **out of sequence** boxes are all unchecked. First, select program Test2 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 Test2. Observe the pipeline and when the program is finished make a note of the following values

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

Do the same with program Test3 and make note of the following values

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

And finally, check the **operand forwarding** and **jump prediction** boxes and re-run Test3. Make note of the following values

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

Now, think about what has been done and look at the three sets of values recorded.

- Is there a jump instruction in Test3 code? Think about the impact of this on the second set of instructions
- What is the % increase in the SF between the 1st set and the 3rd set of values recorded?
- Briefly explain the significance of tutorial exercise 5