CSE 30321 – Computer Architecture I – Fall 2011

Lab 03: Datapath Design and Performance
Total Points: 50 points
Assigned: October 4, 2011
Due: October 27, 2011

1. Goals and Description

There are 3 main components to this lab:

- In **Part 1**, you will be asked to augment the *multi-cycle* MIPS datapath to support a new addressing mode.

- **Part 2** will consider the importance of predicting the outcome of a branch instruction in a *pipelined datapath*. If a pipeline is required to stall until the outcome of a branch instruction is known, there will be a significant, and undesirable impact on performance. In part 2, you'll quantify this impact using the MP3 encoding benchmark as a case study. You'll also consider several *branch prediction* schemes that can help to improve the performance of a pipelined datapath.

- Finally, in **Part 3**, we will consider an architectural-level performance metric that is becoming increasingly important – namely *energy per operation*.

2. Addressing Modes – 15 points

Background:
There are 5 different ways that MIPS instructions can address the register file OR memory. (These are summarized below – although for additional information, see p. 132-33 of your textbook.)

1. *Immediate Addressing:* the operand is a constant within the instruction
   - (e.g. addi $2, $3, 10)
2. *Register Addressing:* the operand is in a register
   - (e.g. add $2, $3, $4)
3. *Base (or displacement) Addressing:* the operand is at the memory location whose address is the sum of a register and a constant in the instruction
   - (e.g. a lw or sw instruction)
4. *PC-relative Addressing*: the branch address is the sum of the PC and a constant in the instruction
   - (e.g. a beq instruction)
5. *Psuedo-direct Addressing:* the jump address is 26 bits of the instruction concatenated with the upper bits of the PC
   - (e.g. like a j instruction)

However, these are not the *only* ways that an instruction could address the register file or memory. Notably, the ARM ISA may leverage the *indexed* addressing modes – the syntax of which is shown below in Table 1. Note that this addressing mode might also be useful for array addressing.

Table 1: Displacement vs. Indexed addressing modes.

Addressing Mode	Example Instruction	Meaning	When Used
Displacement	Add R4, 100(R1)	R4 ← R4 + Mem[100+R1]	Accessing local variables
Indexed	Add R3, (R1+R2)	R3 ← R3 + Mem[R1+R2]	Array addressing; R1 = base of array, R2 = index amount

Problem:
Augment the MIPS multi-cycle datapath and finite state diagram to support the *indexed* addressing mode described above. Your answer should be supported with an augmented datapath and finite state machine diagram.

Useful Notes:
- I have posted copies of the datapath diagram and finite state diagram on the course website
  - (the link is next to the Lab 03 handout link)
- I strongly suggest that you follow the procedure/approach discussed in Lecture 12
- If new control signals are needed, you do not have to augment the states associated with other instructions with these signals
- If helpful, you can assume the following:
  - The instruction encoding can be the same as the MIPS R-type (with the shamt and function code fields just ignored).
  - Your mnemonic / RTL might look like:
    - lw-index $x, $y, $z # $x ← Memory[$y + $z]

3. Branch Prediction and Pipelining – 20 points

Background:
As discussed in lecture, predicting the outcome of a branch instruction is of the utmost importance when moving to a pipelined datapath. On average, every 6th instruction will be some kind of a branch. Using the 5-stage MIPS pipeline as context (and assuming that 3 clock cycles are required to fetch the instruction, decode the instruction, and test to see whether or not the branch condition is met), as seen in Lecture 13, branch instructions can degrade the ideal CPI associated with pipelining (i.e. 1). For example, if we always had to stall for a branch, the new baseline CPI of the MIPS pipelined datapath would immediately reflect a 50% performance degradation:

$$CPI_{always-stall} = 1 + \left(\frac{1}{6}\right)(3\text{ CC}) = 1.5\text{ CC}$$

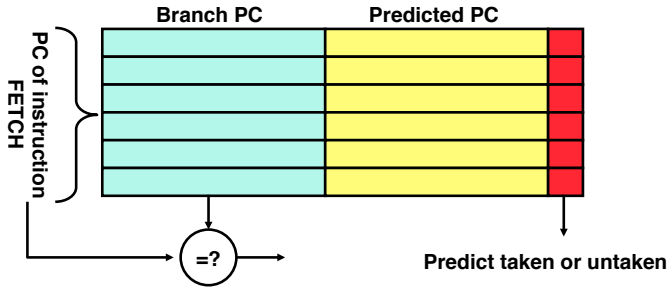
To try to mitigate the negative effects that branch instructions can have on a pipelined datapath, it is actually quite common to try and *predict* the outcome of a branch instruction (i.e. what instruction should be fetched next) and start it down the pipeline. There are a few approaches that one could take (aside from simply stalling the pipeline).

1. We could always predict that the branch will NOT be taken – so the next instruction that will be fetched and started down the pipeline is just at PC + 4. In a sense, this is a better option than just stalling the pipeline. If we assume that 50% of the time a branch is taken, and 50% of the time a branch is not taken, at least one half of the time, we'll be starting useful work – and for the other half of the time, the impact on the performance of the pipelined datapath will be no worse than if we just stalled the pipeline!

$$CPI_{predict-not\ taken} = 1 + \left(\frac{1}{6}\right)(0.5)(3\text{ CC}) = 1.25\text{ CC}$$

2. A second approach involves keeping a "selective history" of all instructions that are executed. This "history" can be referenced during the fetch and decode stages of the pipeline.

    - In the *fetch* stage of the pipeline, the current value of the PC is compared against a small table – the *branch target buffer (BTB)* – that contains the last N PC values that were branch or jump instructions. If there is a match, the PC value is updated with the predicted address rather than PC + 4. (See below)



    - In the next stage of the pipeline, a more careful prediction is made:
        - After a branch instruction is processed for the first time, some information about it is recorded in a *branch history table* (BHT)
        - Each entry in the BHT might consist of several bits of information that are used to make a prediction – i.e. an entry might be the value of the 2-bit counter discussed in class.

- The BHT prediction is generally more accurate – and contains many more entries – than the BTB prediction. If the predictions differ, the BHT prediction would override the BTB prediction.
- Similarly, if we discover in the decode stage that our instruction was a branch, no entry was found in the BTB, and our BHT predicts that the branch will be taken, we can start fetching instructions from the address that was just calculated in the *decode stage.*
- If the prediction suggests that the branch will not be taken, instructions in the pipeline can proceed as normal.
- As an example:
    a. Assume that a branch equal instruction was fetched from address 20000_{10}, and that the BHT has 128 entries
    b. Thus, this particular instruction would map to BHT entry 32.
        - (20000 modulo 128 = 32)
    c. The data in the table entry represents the prediction.

- After the branch outcome is determined in the execute stage of the pipeline, (i) the BTB and BHT would be updated accordingly, and (ii) if necessary, the pipeline would be flushed and the correct instruction started.

Note that a modern desktop or high-performance microprocessor will almost certainly involve a sophisticated "branch predictor," and successful prediction rates of ~95% are common – although it's almost impossible to achieve "perfect" branch prediction. Why? If the instructions at addresses 20000_{10} and 21280_{10} were both branches – they would both map to BHT entry 32. In essence, there would be a conflict, and 1 instruction might make predictions based on the history of a second instruction. The branch predictor also needs time to "warm up".

<u>Simulating how branch prediction impacts performance</u>:
You can change the branch prediction mechanism in a SimpleScalar simulation to see how different branch prediction methods impact performance. (The datapaths that you worked with in Lab 01 are pipelined, so there is a good mapping to Lab 01 and this problem.) We'll look at how 6 different branch prediction schemes can impact performance in the remainder of this lab. They are:

1. *Always stalling*
    - (Thus, no prediction is made, and we just stall the pipeline.)
2. Predicting that the branch is *not taken*
    - (So, in other words, we just fetch the next instruction.)
3. A *bimodal* branch predictor
    - This is essentially the 2-bit predictor discussed in class and above
4. A *2-level* branch predictor
    - This is an example of a more sophisticated branch prediction scheme where a prediction is made based on (i) the current value of the PC and (ii) whether or not recent branch instructions *around* the current instruction were taken or not taken. (Thus, this approach looks at what a given instruction did the last time it was encountered, as well as what surrounding instruction did the last time they were encountered.)
    - The hardware required to implement this predictor is similar to the hardware required to implement the bimodal predictor. The main difference is that we add a global shift register which keeps track of what the last N branch instructions did.

- As an example, assume we have the following 3 instructions:

Address	Instruction	Will be taken / not taken	Shift register with branch history (before)	Shift register with branch history (after)
1000_{10}	BEQ	Taken	0000	0001
1016_{10}	BEQ	Not Taken	0001	0010
1100	BEQ	Taken	0010	0101

If a branch is taken, a 1 is shifted into the global shift register; otherwise, a 0 is shifted in.

This history is concatenated with M bits of the PC – and is used to index the BHT. In this problem, we'll assume a shift register with 8 bits (thus, the last 8 branch instructions are tracked) and a 2048 entry branch history table.

5. A *combination* branch predictor
   - Generally speaking, this approach combines multiple branch prediction schemes, and makes a final decision by majority vote.
6. Assuming *perfect* branch prediction
   - (For this option, every branch is correctly predicted; again, this is practically impossible – but is a useful simulation option to have as it allows you to quantify the impact that mis-predicted branches have on a pipelined datapath.)

FYI: What do *real chips* do?
The Intel Pentium III used a 512 entry BTB and a 2-level branch history algorithm. The AMD K6 processor also uses a 2-level branch history algorithm with a 8192 entry BHT. Finally, the Motorola Power PC leveraged a 512 entry BHT and a 64 entry BTB. Thus, what you are studying is representative of "real hardware."

Simulating how branch prediction impacts performance:
Here, you will need to look at how 6 different "prediction" options impact the MP3 encode benchmark. You only need to consider the small dataset. See Appendix A for a short description of how to change what prediction is used. A script and configuration file are provided at:

/afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/03/

Part 3.A: (4 points)
Using the XScale configuration file, use SimpleScalar to complete Table 2:

Table 2: Execution time of Mad assuming different branch prediction schemes.

	Always Stall	Not Taken	Bimodal	2-level	Combination	Perfect
Mad (small input)						

Part 3.B: (8 points)
Quantitatively, if we always have to stall the pipeline if a branch instruction occurs, what is the performance impact when compared to the ideal case?

Part 3.C: (8 points)
Which branch predictor performs best? Given your results, how well do you think branch predictors do in mitigating performs hits in pipelined datapaths?

4. Energy Efficiency – 15 points

Background:
As we have discussed throughout the semester so far, the best way to compare one computer architecture to another is to consider how long it takes to run a common benchmark (i.e. we compare *execution times*). However, the *energy efficiency* of a given architecture has rapidly become an equally important measure of comparison.

Why is this the case? Let's look at some data from some real computers at Oak Ridge National Laboratories (ORNL). "Scientists and engineers at ORNL conduct basic and applied research and development to create scientific knowledge and technological solutions that strengthen the nation's leadership in key areas of science; increase the availability of clean, abundant energy; restore and protect the environment; and contribute to national security."[1] Table 1 details the power requirements for all of the computers at Oak Ridge National Laboratories. Note that data is broken up into the power requirements to run the computers, and the power requirements to cool the building where the computers reside.

Table 1: Power Requirements for computers at ORNL

Year	Computers (MW)	Cooling (MW)
2005	1.5	1
2006	3.5	2
2007	11	9
2008	20	17
2009	28	22
2010	38	30
2011	47	40

Before continuing, it is worth mentioning that the numbers above are fairly representative for other large computing structures too. For example, the data center that Microsoft recently opened in Chicago needs about 40 MW of power annually to operate and "serves as the guts behind Microsoft's online ambitions, from Bing to Hotmail to Windows Azure [the company's new on-line operating system]."[2]

The magnitude of these power budgets suggests some problems:

1. It becomes increasingly problematic / expensive to supply the power required for operation. As a point of reference, the output of a nuclear reactor is on the order of ~1000 MW!
2. As power is analogous to heat, the practical costs of cooling a data center or computing facility must be taken into account.

Additionally, as more and more information processing moves to mobile devices, battery life has also become an important design driver.

For all of the above reasons, metrics such as execution time per Joule have become important.

---

[1] From: http://www.ornl.gov/ornlhome/about.shtml
[2] From http://news.cnet.com/8301-13860_3-10364746-56.html

Questions:
In the remainder of this lab, we'll consider the energy efficiency of different branch predictors. Based on the data provided in the table below – CPI, instruction count, and average power for the ispell benchmark with a small dataset – answer the questions below.

Predictor	CPI	# instructions	μArch power (W)
Comb 2K entry BHT, 8-bit SR	0.6563	19,509,907	0.9881
Bimodal 128K entry BHT	0.6832	19,509,792	1.3279
Bimodal 64K entry BHT	0.6825	19,509,970	1.1536
Bimodal 2K entry BHT	0.6832	19,509,935	0.9842
2-level 2K entry BHT, 8-bit SR	0.7166	19,510,009	0.9350
2-level 1K entry BHT, 8-bit SR	0.7365	19,509,749	0.9345
Not taken	1.399	19,509,376	0.5797

Part 4.A: (7 points)
Assuming a 233 MHz clock rate, what branch prediction approach is the most energy efficient? Why do you think this is the case? To support your answer, you might determine which prediction scheme leads to the most efficient execution time, which prediction scheme is the most energy efficient, which prediction scheme leads to the higher number of instructions per Joule, etc.

Part 4.B: (8 points)
The average power associated with the *not taken* strategy is significantly lower than all other approaches. Do these power savings translate to energy savings for the benchmark? If you think that the power savings *do* translate to energy savings, explain why. Similarly, if you think they *do not*, explain why.

5. What to Turn In

You should turn in a typed report with answers to the questions listed above. Be sure to include augmented datapath and finite state machine diagrams for the problem discussed in Section 2.

Appendix

To run a benchmark with a different branch predictor, simply edit the following line in the configuration file:

```
# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                  bimod
```

Thus, to run the benchmark assuming that braches are not taken, simply change “bimod” in the second line to “nottaken”. (Similarly, to run a benchmark assuming that all branches are taken or all predictions are perectly made, just change “bimod” to “taken” or “perfect” respectively.)

To mimic the “always stall” case, use the “not taken” case, and edit the following line…

```
# extra branch mis-prediction latency
-fetch:mplat                0
```

…by changing “0” to “3”.

Each benchmark should take approximately 1 minute to run.