CSE 30321 – Computer Architecture I – Fall 2010

Lab 03: Datapath Design and Pipelining
Total Points: 100 points
Assigned: October 5, 2010
Due: October 28, 2010

1. Goals and Description

The questions associated with this lab touch on 3 of the 6 course goals (to be discussed below).

There are 3 main components to this lab:

- The **first component** will consider the correlations between instruction encodings and datapath hardware. You have already seen (for MIPS and the 6-instruction ISA) what an encoding tells you about the hardware available to execute the instruction. In Part 1 of this lab, you'll be given encodings for sample ARM instructions, and will be asked to describe the datapath based on these encodings.
  - This is directly related to course goal #1 – i.e. "Describe the fundamental parts of a microprocessor, how they interact with each other, etc."
- The **second component** is directly related to course goals #2 and #5 – i.e. (#2) "applying knowledge about a processor's datapath and performance metrics to design a machine so it meets a target set of performance goals", and (#5) "using knowledge about underlying hardware to write more efficient software". More specifically, you will be asked to augment the multi-cycle MIPS datapath to support a new *addressing mode* (to be discussed further below), and will also consider how new addressing modes could be used to speedup common software routines.
- Finally, the **third component** 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 negative) impact on performance.
  - This is directly related to course goal #2.

2. Datapath Design and Instruction Encodings

During Lectures 2 and 3 (for the 6-instruciton ISA) and Lecture 10 (for the MIPS ISA), we discussed how, by looking at an instruction encoding, you could learn a lot about the capabilities of the datapath used to execute a given instruction. For example, in Figure 1, I've included snapshots of the Lecture 10 slides that illustrate the MIPS R-type and I-type arithmetic instructions. Looking at the I-type slide, you can see that the 6-bit opcode allows you to specify at least 2^6 (or 64) unique instructions.

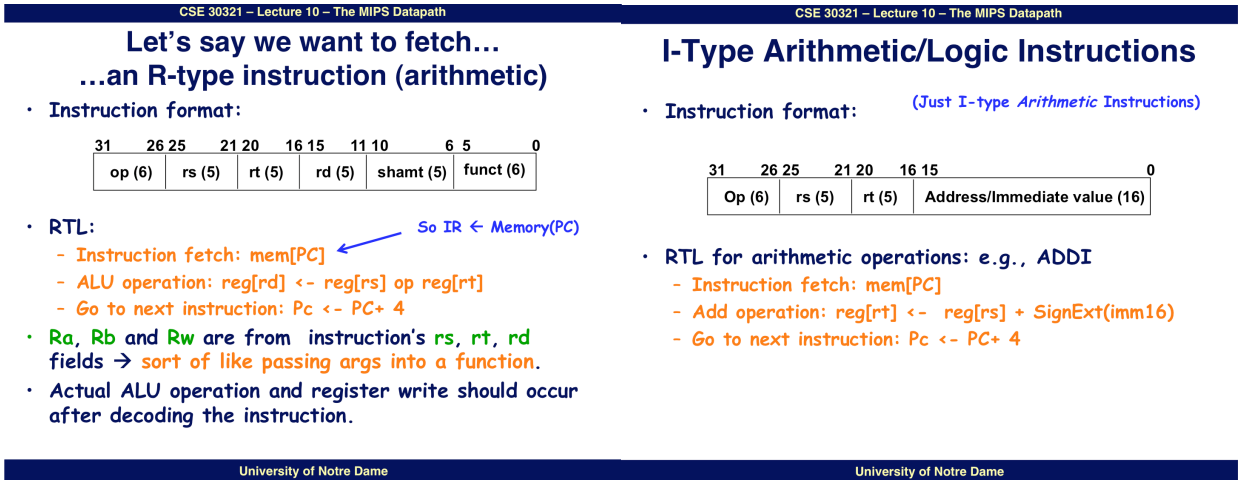

Figure 1: Example MIPS R-type and I-type (arithmetic) encodings.

In Figure 2 below, I have included the instruction encodings for 2 types of ARM instructions:

- The first is an encoding for a 32-bit "register-immediate" instruction (i.e. 1 value is an immediate value like the I-Type MIPS).
- The second is an encoding for a 16-bit instruction for ARM's "Thumb" ISA – where, quite simply, instruction encodings may be 16 bits instead of 32 bits. The 16-bit encoding shown in Figure 2 is also for a "register-immediate" instruction.

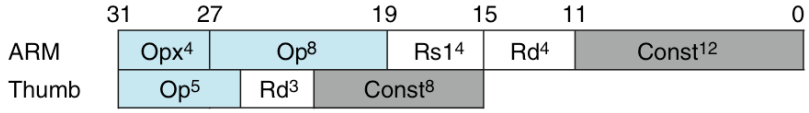

Figure 2: Encodings for 32- and 16- bit ARM "register-immediate" instructions. Note that "Opx" stands for opcode extension, "Op" stands for opcode, and "Const" stands for constant/immediate value. "Rs" and "Rd" stand for register source and register destination respectively. The superscripts denote the number of bits.

Part 2.A:
Describe the datapath, register file, instruction mnemoic, etc. associated with the ARM 32-bit and 16-bit register-immediate instruction encodings. (I am not looking for a "user manual quality" description, but am interested in what hardware you think is available, the source and destination of the data used in the instruction, etc.)

Part 2.B:
Why do you think ARM released a 16-bit ISA? What application spaces do you think 16-bit instructions map well to? Why?

3. Addressing Modes

There are 5 different ways that MIPS instructions can address the register file OR memory. As summarized on pages 132-133 of your textbook, the MIPS addressing modes are enumerated below, and illustrated graphically in Figure 3. (The text and graphics here are from Hennessey and Patterson.)

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

However, these are not the *only* ways that an instruction could address the register file or memory. Other common addressing modes used in other ISAs (as well as when they may be useful from a software perspective) are summarized in Table 1. Notably, the ARM ISA may leverage the *indexed, scaled, auto-increment, and auto-decrement* addressing modes (among others).

Part 3.A:
Augment the MIPS multi-cycle datapath and finite state diagram to support the *indexed* addressing mode described above. Note that:

- 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

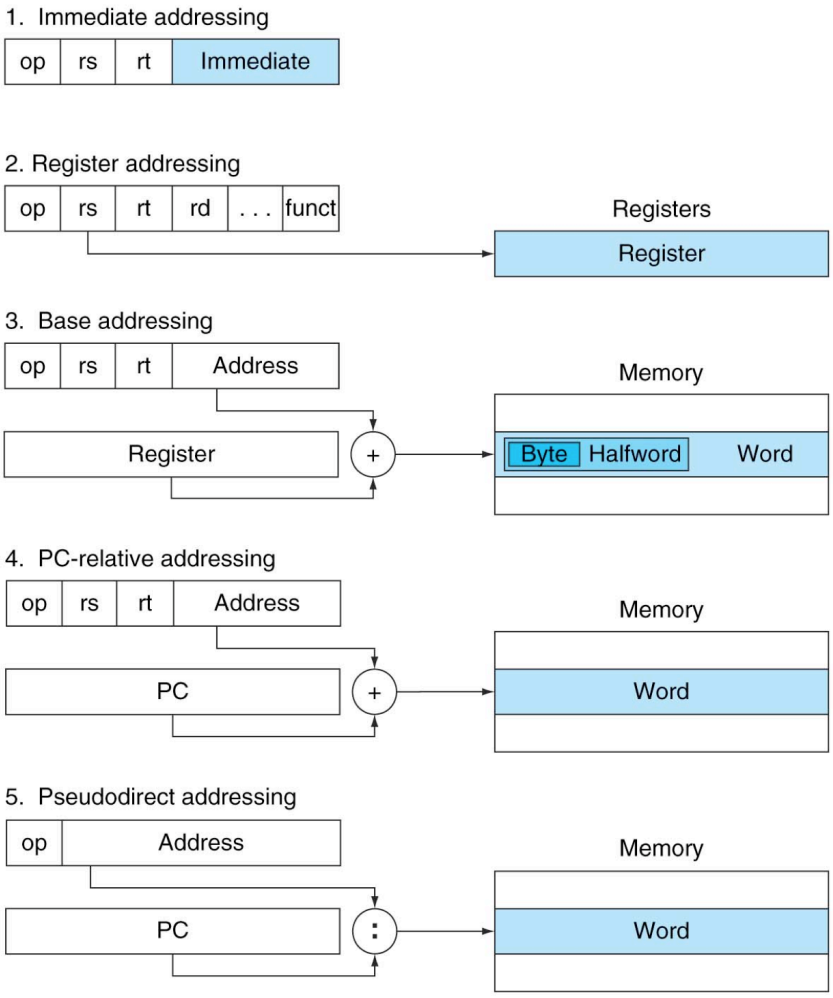

FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC. Copyright © 2009 Elsevier, Inc. All rights reserved.

- **Figure 3:** Summary of MIPS addressing modes (from HP textbook).

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]

Table 1: Commonly used addressing modes. (Shaded entries are used in MIPS)

Addressing Mode	Example Instruction	Meaning	When Used
Register	Add R4, R3	R4 ← R4 + R3	When a value is in a register
Immediate	Add R4, #3	R4 ← R4 + 3	For constants
Displacement	Add R4, 100(R1)	R4 ← R4 + Mem[100+R1]	Accessing local variables (for lw/sw)
Register deferred *or* Indirect	Add R4, (R1)	R4 ← R4 + Mem[R1]	Accessing/using pointer or computed address
Indexed	Add R3, (R1+R2)	R3 ← R3 + Mem[R1+R2]	Array addressing; R1 = base of array, R2 = index amount
Direct or Absolute	Add R1, (1001)	R1 ← R1 + Mem[1001]	Accessing static data; Address constant may need to be big
Memory indirect *or* Memory deferred	Add R1, @(R3)	R1 ← R1 + Mem[Mem[R3]]	If R3 is the address of a pointer *p*, then this mode yields **p*
Autoincrement	Add R1, (R2)+	R1 ← R1+Mem[R2]; R2 ← R2 + *d*	Useful for stepping through arrays within a loop; R2 points to start of array; each reference increments R2 by *d*
Autodecrement	Add R1, (R2)-	R1 ← R1-Mem[R2]; R2 ← R2 + *d*	Same as autoincrement; can be used for push/pop on stack
Scaled	Add R1, 100(R2), [R3]	R1 ← R1 + Mem[100+R2+R3**d*]	Used to index arrays

Part 3.B:
In Lab 01, I described several different *benchmark suites*. One benchmark suite that was mentioned (but not used) was LINPACK (described again in Table 2):

Table 2: Description of the LINPACK benchmark suite

LINPACK
The ***LINPACK*** benchmark suite might be used if you were designing an architecture for a high-performance computing system (i.e. something that might be used for weather/climate modeling). "LINPACK is a software library for performing numerical linear algebra on digital computers. The LINPACK Benchmarks are a measure of a system's floating point computing power. Introduced by Jack Dongarra, they measure how fast a computer solves a dense *N* by *N* system of linear equations *Ax* = *b*, which is a common task in engineering."[1]

Thus, at the heart of the LINPACK benchmark suite is code that implements the operation: y = aX + y.

As a computer architect, you have been tasked with improving the performance of this code. Presently, you are running this code on a multi-cycle datapath where the array accesses and index counter updates are very MIPS-like. In other words, an add instruction would be used to update the array index/loop counter, and separate load and store instructions are needed to access array elements.

[1] text from: http://en.wikipedia.org/wiki/LINPACK

Thus, the assembly language might look something like:

```
Loop:   load   R3, 0(R1)     # load x[j]
        mult   R4, R3, R7    # multiply a*x[j]; assumes a maps to R7
        load   R5, 0(R2)     # load y[i]
        add    R5, R4, R5    # add a*X[j] + y[i]
        store  R5, 0(R2)     # store y[i]
        addi   R1, R1, 4     # increment X index
        addi   R2, R2, 4     # increment Y index
        bneq   z, R2, Loop   # start loop again if condition not met (z is # of iterations)
```

The processor datapath that you are designing is specifically targeted to run code similar to that of the LINPACK benchmark suite. As such, reducing the execution time of code like that shown above is important. One option that you are considering is adding new instructions – store++ and load++ – that will automatically update the base register by 4.

Part 3.B.i:
If store++ and load++ are implemented, what instructions in the above loop could be changed, and what instructions could be eliminated?

Part 3.B.ii:
Assume that you are working with a multi-cycle datapath, and the number of clock cycles for each instruction is as shown in Table 3 below. For a 100,000 iteration loop, how many clock cycles are saved?

Table 3: Assumed Cycles Per Instruction for Problem 3.B (Part 2)

Instruction	Clock Cycles Required
load	7
mult	8
add	5
store	6
addi	5
bneq	4
load++	8
store++	7

4. Branch Prediction and Pipelining

As you have seen 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 significantly degrade the ideal CPI associated with pipelining (i.e. 1).

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

1. We could always predict that 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!

2. Alternatively, we could predict that the branch will be taken. However, this is a harder decision to implement – as the new target address will not be calculated until the second clock cycle of the branch instruction at the earliest. So, as a branch instruction moves from the fetch stage to the decode stage of the pipeline, we will not yet have calculated a new address to "fetch" from.

3. The third approach involves keeping a "selective history" of all instructions that are executed. This "history" can be referenced during the fetch stage of the pipeline. Essentially, every time there is a branch instruction, the address of the PC (and the outcome of the branch / new PC value) is recorded in a table. If the table suggests that the current instruction being fetched is a branch, there are 2 possible outcomes:
    a. The table predicts that the branch should not be taken – and the next instruction fetched will just be from PC + 4.
    b. The table predicts that the branch will be taken – and the next instruction fetched will be from the PC value listed in the table.

    Graphically, this process is illustrated in Figure 4. Note that a modern desktop or high-performance processor will almost certainly involve a sophisticated "branch predictor" and successful prediction rates of ~95% are common. To learn more about a simple (yet reasonably accurate) predictor, see pages 380-381 of your textbook.

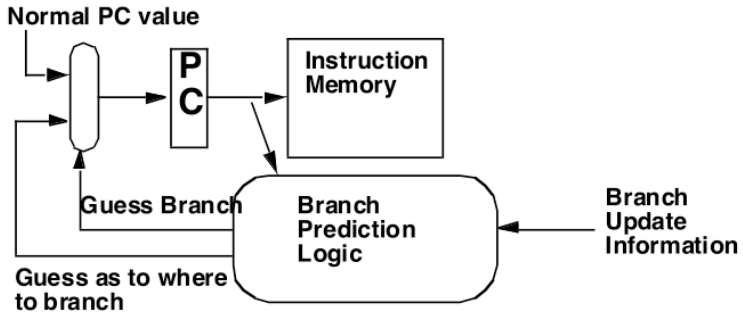

Figure 4: Graphical view of branch prediction.

Part 4.A:
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 the simulations you did for Lab 01 and this problem.) Here, you will need to look at how four different "prediction" options impact the MP3 encode/decode benchmarks and the ispell benchmark. You should consider larger and small datasets. The options that we will consider include:

1. Predicting that the branch is Not Taken
    - (So, in other words, we just fetch the next instruction.)
2. Predicting that the branch is Not Taken and there is a branch mis-prediction latency of 3 CCs.

- (This is the easiest way to make SimpleScalar mimic the "always stall" case.)

3. 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.)
4. Assuming a bimodal branch predictor
   - (While slightly more sophisticated, this predictor is similar to the 2-bit predictor described on pages 380-381 of your textbook. You do not need to know the exact details in order to complete this part of the lab.)

Using the XScale configuration file, use SimpleScalar to complete Table 4[2]:

Table 4: Execution time of Lame and Mad assuming different branch prediction schemes.

	Execution Time (Predict Not Taken)	Execution Time ("Always Stall")	Execution Time (Perfect)	Execution Time (Bimodal)
Lame (small input)				
Lame (large input)				
Mad (small input)				
Mad (large input)				
ispell (small input)				
ispell (large input)				

(to calculate execution time, you can assume that the clock rate of the processor is 233 MHz)

Part 4.B:
Quantitatively, how does branch prediction affect speedup? Do you think the bimodal predictor does a good job?

5. What to Turn In

You should turn in a *typed* report with answers to:

- Parts 2.A and 2.B
- Parts 3.A, 3.B.i, and 3.B.ii
- Parts 4.A and 4.B

I expect that datapath modifications and state machine diagrams should be neatly drawn (or done with powerpoint) so that changes are easy to follow.

[2] See Appendix for explanation of how to run benchmarks.

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}
-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, simply edit the following line…

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

…by changing “0” to “3”.