# CSE 30321 – Computer Architecture I – Fall 2009
## Lecture 17-18– In Class Examples
October 27 & 29, 2009

## Question 1: Multi-cycle performance

You have determined that the critical path that sets the clock cycle length of the MIPS multi-cycle datapath is memory accesses for loads and stores (not for fetching instructions). This has caused a new implementation of this processor to run at a clock rate of 4.8 GHz instead of the target 5.6 GHz.

You're contemplating a "solution" that involves taking all of the cycles that access memory and breaking them down into 2 cycles. This way, the machine can run at its target clock rate.

Using the SPEC CPUint 2000 mix shown below, determine how much faster the machine with the two-cycle memory access is compared with the 4.8 GHz machine with single-cycle memory accesses.

| Instruction Type | CCs | Frequency |
|---|---|---|
| ALU | 4 | 48% |
| Store | 4 | 10% |
| Load | 5 | 27% |
| Branch / Jump | 3 | 15% |

***Fundamentally, how will you answer this question?***

- Calculate 2 execution times and compare
- Use the CPU time equation where instruction count will be the same – only clock rate and CPI will change

***Quantitative Answer:***

$\text{CPU time}_{4.8}$ = IC * [(0.48)(4) + (0.1)(4) + (0.27)(5) + (0.15)(3)] * $(4.8 \times 10^9)^{-1}$ = (IC) * $8.58 \times 10^{-10}$s

$\text{CPU time}_{5.6}$ = IC * [(0.48)(4) + (0.1)(**5**) + (0.27)(**6**) + (0.15)(3)] * $(5.6 \times 10^9)^{-1}$ = (IC) * $8.02 \times 10^{-10}$s

8.58 / 8.02 = 1.07. Therefore, the new version is about 7% faster than the old version.

***Question: Qualitatively, why is making this change a good idea?***

The instructions that now take more clock cycles to execute only account for 37% of the total instruction mix. Therefore, we're now making "the common case" (67% of instructions) execute (5.6 / 4.8 = 17% faster…)

**Would you consider the further step of splitting instruction fetch into two cycles if it would raise the clock rate up to 6.4 GHz?**

$\text{CPU time}_{6.4}$ = IC * [(0.48)(**5**) + (0.1)(**6**) + (0.27)(**7**) + (0.15)(**4**)] * $(6.4 \times 10^9)^{-1}$ = (IC) * $8.58 \times 10^{-10}$s

No. This is not a good idea. The execution time is the same as that with the 4.8 GHz clock! Put another way, we've now made *every instruction* one clock cycle slower. Look at just the ALU class. With a 5.6 GHz clock rate, each instruction takes 4 CCs. This implies that every ALU instruction takes 714 ps to execute. With the faster clock rate, each instruction takes 781 ps to execute. The CC may take less time, but there are more of them.

## Question 2: Multi-cycle control

Referring to the extra handouts, modify the multi-cycle datapath shown below to support a new instruction: load++ $x n($y).

The RTL for load ++ is as follows:

$x ← Mem(n + RF($y))
$y ← $y + 4

Show any necessary changes to the FSM as well.

Part A:

Describe – cycle-by-cycle (starting with Fetch) – what this instruction needs to do:

- Fetch:
  - IR ← Mem(PC)
  - PC ← PC +4
- Decode:
  - A ← RF(IR[25:21])
  - B ← RF(IR[20:16])
  - ALUOut ← PC + (sign-extend (IR[1:0] << 2)
- Execute:
  - ALUOut = A + sign-extend(IR[15:0])
- Memory:
  - MDR ← Mem[ALUOut]
  - Question: Can we update the $y register here?
    - Yes!
    - The ALU is idle for all intent and purposes
    - Therefore, can also do: $y ← $y + 4
- Write Back:
  - RF(IR[20:16]) ← MDR
  - Can we also update $y here? Actually, there's 2 answers…
    - Answer 1: Yes… but we need to add more HW…
      - (Namely another path to write data to the register file is needed…)
    - Answer 2: No… we need to add another state…
      - Look at mux … you can only select the MDR or ALUOut … not both!
      - Begs another question… how do you modify the FSM?
        - My answer … add State 12 coming off of State 4. This would allows us to handle the load++ case. There would then be a path back to fetch

Part B:

Given your answer to Part A, how would you modify the state machine?

If Answer A, we would need to modify both the state machine and the datapath. One solution would be:

- Change "Write Register" on the RF to Write Register 1 – and add Write Register 2
- Similarly, change "Write Data" on the RF to "Write Data 1" – and add "Write Data 2"
- Then, duplicate the multiplexors that feed the initial input
- This will require the addition on 3 control signals – MemToReg(2), RegDst(2), and RegWrite(2)
- A new state would still need to be added however… State 11 would branch off of State 3 to ensure that 2 registers are written, not just 1.

If Answer B, we would need to add another state as specified above…however, no new control signals would need to be added.

- Let's assume we would write data into the MDR first…
- Then in State 12, we would want to write data in ALUOut to the register file.
- Therefore, we would need to do the following:
    - In State 3, we would need to add the following:
        - ALU control = ADD
        - ALUSrcB = 01 (select #4)
        - ALUSrcA = 1 (select A register)
    - For State 12 (added above) we would need to:
        - Add another control signal. We need a path from IR[25:21] to the multiplexor that selects which register is written. (note that every other state that asserts this control signal would need to be modified)
        - MemToReg = 0 (select ALUOut)
        - RegWrite = 1 (enable register to be written)

Some general comments:

- These are the kind of tradeoffs that you might consider. Do you add more HW or do you pay an extra CC?
- Note: for you final project, you may be asked to modify the Vahid Processor Datapath to support the execution of several pseudocode benchmarks. Extra credit will be given for the fastest, simplest, etc. design. Here, Answer 1 would give better performance while Answer 2 would be simpler.

## Question 3: Microprogramming

Looking at control signals, we can really break them down into different categories (i.e. signals are associated with different parts of the datapath…)

| Field Name | Function of Field |
|---|---|
| ALU Control | Specify the operation being done by the ALU during this clock; the result is always written to ALU Out |
| SRC 1 | Specify the source for the first ALU operand |
| SRC 2 | Specify the source for the second ALU operand |
| | Specity read or write for the register file, and the source of the value for a write |
| Memory | Specify read or write, and the source for the memory. For a read, specify the destination register |
| PCWrite Control | Specfy the writing of the PC |
| Sequencing | (More Later) Specify how to chose the next instruction to be executed. |

Let's look at the slides for a bit more detail…

- See "Microinstruction Format (2), Microinstruction Format (3), Sample Microinstruction (1), Sample Microinstruction (2)"

**Now it's your turn:**

Question A:

- Look at the state machine on the extra handout.
- If you are executing a branch instruction, after decode you would:
    - Check to see if A= B
    - Set PC to ALU out
        - (Remember that an address is calculated by default in decode as just seen)

Fill in the table below for the BEQ microinstruction:

| ALU Control | SRC 1 | SRC 2 | Reg Control | Memory | PC Write | Sequence |
|---|---|---|---|---|---|---|
| Sub | A | B | | | ALUout-cond | Fetch |

Question B:

- Look at the state machine on the extra handout.
- Complete the tables for a load instruction.

| ALU Control | SRC 1 | SRC 2 | Reg Control | Memory | PC Write | Sequence |
|---|---|---|---|---|---|---|
| Add | A | Extend | | | ALUout-cond | Dispatch2 |

Why Dispatch2? Because the above state is the same for both a LW and a SW. If it's a LW, we need to go to a different microinstruction than if it's a SW.

| ALU Control | SRC 1 | SRC 2 | Reg Control | Memory | PC Write | Sequence |
|---|---|---|---|---|---|---|
| LW2 | | | | Read ALU | | Seq |

Why Seq? Look at the state machine… once we are in state 3, we *always* go to state 4.

| ALU Control | SRC 1 | SRC 2 | Reg Control | Memory | PC Write | Sequence |
|---|---|---|---|---|---|---|
| | | | Write MDR | | | Fetch |

Why Fetch? We're done with this instruction so we need to get ready to fetch the next one.

**Now, let's talk about a bit more about *how* we implement a microprogram and *how* we select the next microinstruction (this gets into the whole "Sequence" field). I'll conclude this discussion on the board…**