CSE 30321 – Lecture 11 – In Class Example Handout

Question 1:

First, we briefly review the notion of a clock cycle (CC). Generally speaking a CC is the amount of time required for (i) a set of inputs to propagate through some combinational logic and (ii) for the output of that combinational logic to be latched in registers. (The inputs to the combinational logic would usually come from registers too.)

Thus, for the MIPS, *single cycle* datapath that was derived in class last week, the "combinational logic" referred to above would really be the instruction memory, register file, ALU, data memory, and register file. The inputs come from instruction memory and the output might be the main register file (in the case of an ALU instruction) or the PC (in the case of a branch instruction).

Putting the MIPS datapath aside, its generally a good idea to minimize the logic on the critical path between two registers – as this will help shorted the required clock cycle time, will result in an increase in clock rate, which generally means better "performance".

To be more quantitative, hypothetically, let's say that there are two gate mappings that can implement the logic function that we need to evaluate. The inputs to these gates would come from 1 set of registers, and the outputs from these gates would be stored in another set of registers. The composition of each is specified in the table below:

	AND gates	NOR gates	XOR gates
Design 1	7	17	13
Design 2	24	4	7

Furthermore, the delay associated with each gate is also listed below:

AND gate	NOR gate	XOR gate
2 picoseconds*	1 picosecond	3 picoseconds

Which design will lead to the shorted CC time?

Design 1: = (7 ANDs * 2 ps/AND) + (17 NORs * 1 ps/NOR) + (13 XORs * 3 ps/XOR)
= 14 ps + 17 ps + 39 ps = **70 ps**

Design 2: = (24 ANDs * 2 ps/AND) + (4 NORs * 1 ps/NOR) + (7 XORs * 3 ps/XOR)
= 48 ps + 4 ps + 21 ps **= 73 ps**

What is the clock rate for that design (in GHz)?

- Clock rate = 1 / CC
- Clock rate = 1 / 70ps
- Clock rate = $1 / (70*10^{-12})$
- Clock rate = 1.428×10^{11} Hz
- Clock rate = 1.428×10^{11} Hz * (1 GHz / 10^{9} Hz)
- Clock rate = 14.3 GHz

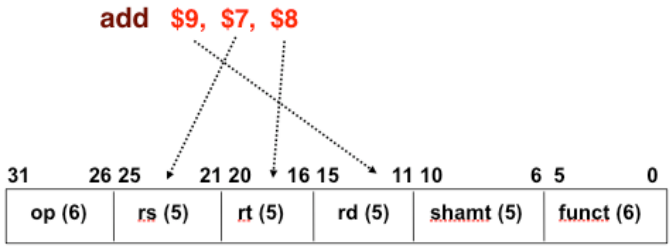
Take Away: The longest critical path determines the clock cycle time

Question 2:

Without looking back in your notes, write out the register transfer language (RTL) for a generic MIPS R-type, I-type, and J-type instruction. (Don't be overly concerned with getting all of the sign extensions, etc. exactly right.)

For your reference, the instruction encodings for the R, I , and J instruction types are shown below.

R-Type



I-Type

J-Type

Answers:

	Your Solution	My Solution
R-Type		- get ← Mem(PC) - Reg(Rd) ← Reg(Rs) op Reg(Rt) - PC ← PC + 4
I-Type (lw/sw)		- get Mem(PC) - Addr ← Reg(Rs) + <0000_h> <imm(15:0)> **- if lw** - Reg(Rt) ← Mem(Addr) **- if sw** - Mem(addr) ← Reg(Rs) - PC ← PC + 4 (?) – lw or sw = longer latency? (?) – # of register/memory transfers in each?

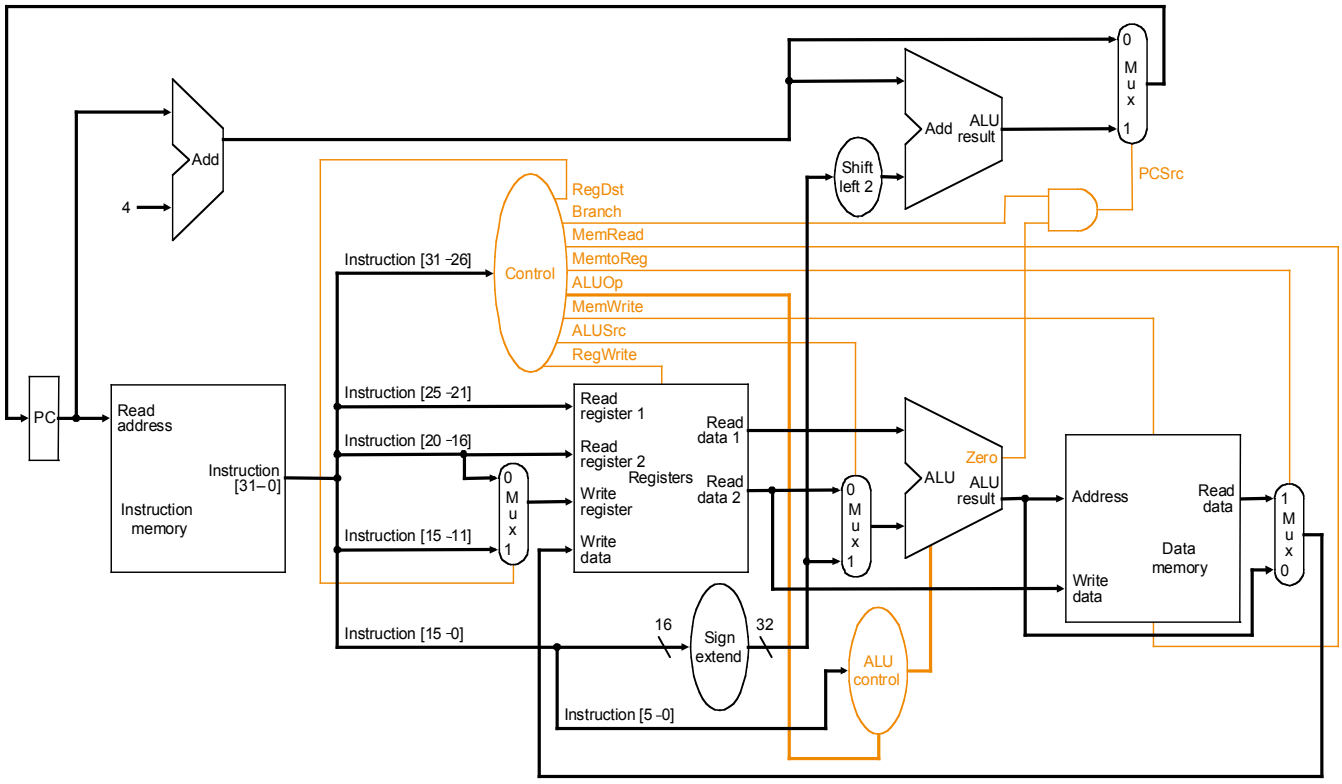
I-Type (beq)		- get ← Mem(PC) - Cond ← Reg(Rs) – Reg(Rt) **- if cond met** - PC ← new value - PC ← PC + 4 + signext(imm(15:0))x4 **- if cond *not* met** - PC ← PC + 4
J-Type		- get ← Mem(PC) - PC ← <PC extension> <IR(25:0)>

Take Away: There's a lot of stuff common between instructions! For example:

- For R-Type:
  - Everything the same except what the ALU does
- Looking at R-type vs. I-type
  - Still, lots of stuff the same – register reads, ALU operation, register write back

Question 3:

In class last week, you derived the single cycle datapath for the MIPS ISA. A block diagram is shown below:



Assume that the following latencies would be associated with the datapath above:

Operation	Time
Get instruction encoding – Mem(PC)	2 ns
Get data from register file	2 ns
Perform operation on data in registers	3 ns
Increment PC by 4	1 ns
Change PC depending on conditional instruction	1 ns
Access data memory	2 ns
Write data back to the register file (from the ALU or from memory)	2 ns

Questions:

(a) What type/class of instruction in the MIPS ISA will set CC time (and hence clock rate)? (e.g. ALU type, conditional branch, load, store, etc.)

(b) Given the latencies in the above table, what would the clock rate be for our single cycle datapath?

Note: for both (a) and (b), pay particularly close attention to the diagram shown above and try to *define the critical path.*

Answers:

First, the load instruction sets the critical path. For the load, we must:

- Get the instruction encoding from memory
- Read data from the register file
- Perform an ALU operation
- Access data memory
- Write data back to the register file

Note that we can ignore the overhead associated with PC ← PC + 4 b/c this would happen in parallel with operations

Second, the clock rate would just be the inverse of the some of the delays:

- (2 ns + 2 ns + 3 ns + 2 ns + 2 ns)
- 11 ns
- 1 / 11 ns = 91 MHz

Take Away:

- In a single cycle implementation, the instruction with the longest critical path sets the clock rate for EVERY instruction.
- Think about Amdahl's Law – this is good if we can improve something that we use often…
    - …but here, we're sort of making a lot of things worse!
    - (How worse? We'll see next.)

Question 4:
Let's look at the impact of the load instruction's domination of the CC time a bit more:

Recall, that a load (lw) needs to do the following:

a) Get Memory(PC)
b) Read data from the register file and "sign extend" an immediate value
c) Calculate an address
d) Get data from Memory(Address)
e) Write data back to the register file

With this design, let's assume that each step takes the amount of time listed in the table

(a)	(b)	(c)	(d)	(e)
4 ns	2 ns	2 ns	4 ns	2 ns

(for simpler math)

Part 1:
With these numbers, what is the CC time?

A: 14 ns

- However, store (sw) instruction only needs to do steps A, B, C, and D
  - Therefore a store only really needs 12 ns
- Similarly, add instruction only needs to do steps A, B, C, and E
  - Therefore could do an add in 10 ns
- *Still, with single CC design, both of the above instructions take 14 ns.*

Take away: Our CC time is limited by the longest instruction.

Part 2:
Let's quantify performance hit that we're taking – and assume that we could somehow execute each instruction in the amount of time that it actually takes:

Assume the following:

ALU	lw	sw	Branch / jump
45 %	20 %	15 %	20 %

Per the previous discussion, we can assume that ALU instructions take 10 ns, lw's take 14 ns, and sw's take 12 ns. We can also estimate how long it takes to do a branch/jump:

- Assume that steps A, B, C, and E are needed
  - (A) Fetch, (B) Read data to compare, (C) Subtract to do comparison, (D) Update PC (like a register write back)

CPU time = (instruction / program) x (seconds / instruction)
CPU time (single CC) = i x 14 ns = 14(i)
CPU time (variable CC) = i x [(.45 x 10) + (.2 x 14) + (.15 x 12) + (.2 x 10)] = 11.1(i)
Potential performance gain = 14(i) / 11.1(i) = 27%

Take away: By using a single, long CC, we're losing performance.

Part 3:
Unfortunately, such a "variable CC" is not practical to implement… however, there is a way to get some of the above performance back.

- To make this solution work, want to balance the amount of work done per step. Why?
    - Because if every step has to take the same amount of time, if we have 2, 2, 2, 2, and 4 ns, we're still at 4!

As an example, let's see what happens if we can make each step take 3 ns:

CPU time (3 ns, multi-cycle) = (i) x (CC / instruction) x (s / CC)
= (i) x [(.45 x 4) + (.2 x 5) + (.15 x 4) + (.15 x 4)] x 3
= 11.8(i)

11.8(i) is not as good as 11(i), but its better than 14! → 19% vs. 27%

Now, we have a shorter clock rate and each instruction takes an integer number of CCs.

Take away: There is a "catch" to this approach. We have to store the intermediate results. Remember, a CC is defined as the time some logic was evaluated and the result was latched…. and that inputs often come from another, previous latch.

Realistically, the latching time of the intermediate registers will add a nominal overhead to each stage.

Part 4:
Let's see how the overhead might affect performance. We'll assume its 0.1 ns.

CPU time (3 ns, overhead) = (i) x (CC / instruction) x (s / CC)
= (i) x [(.45 x 4) + (.2 x 5) + (.15 x 4) + (.15 x 4)] x (3+0.1)
= (i) x [1.8 + 1 + 0.6 + 0.6] x 3.1
= (i) x [4] x 3.1
= 12.4(i)

Take away:
- Again, we lose a little performance, but we're still better than the single cycle method
    - 14(i) / 12.4(i) = 13%
- Every approach is better than the single cycle approach:
    - 11.1(i) < 11.8(i) < 13(i) < 14(i)
- The 11.8(i) and 13(i) approach are feasible

Question 5:
If we take the above approach, how should we break up the instructions?

Think about each step … what needs to be done to ensure R, I, and J type instructions all work?
- *Hint: Think about each instruction and think about what can be done in parallel.*

(1) IF	**(2)** ID	**(3)** EX	**(4)** M / R-Type	**(5)** WB
(every) Fetch instruction at address in PC	**(every)** Decode instruction type (i.e. send opcode to control logic)	**(lw/sw)** Compute memory address	**(lw)** Read memory, store result in MDR (memory-data-reg)	**(lw)** Write MDR to register file.
(every) Store result in instruction register (IR)	**(every)** Store results read from the register file	**(R-type)** Perform correct logical operation	**(R-type)** Write ALUout to reference	
(every) increment PC	**(jump)** Compute new PC address****	**(Branch)** Update PC post comparison	**(sw)** Write to memory	
		(All) Store intermediate result in ALU **(Note)** All use ALU		

Let's talk about **** more…
- What if we have an R-type instruction: I opcode I rs I rt I rd I shamt I function code I ?
  - Would send rs, and rt encodings to RF as indices
  - Data would be stored in intermediate registers A and B
- What if we have a J-type instruction: I opcode I offset I ?
  - Answer:
    - Do the same as above anyway!
    - There's no harm and no control signals based on opcode available to end of CC anyhow…
  - Similarly, we can calculate a new PC address – i.e. for a jump or branch – even if we don't know if we have a jump or a branch
    - Again, it does no harm + it subscribes to ASAP philosophy
      - We can do this now, and save a CC later

Question 6: 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.