CSE 30321 – Lecture 10-11 – 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 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**

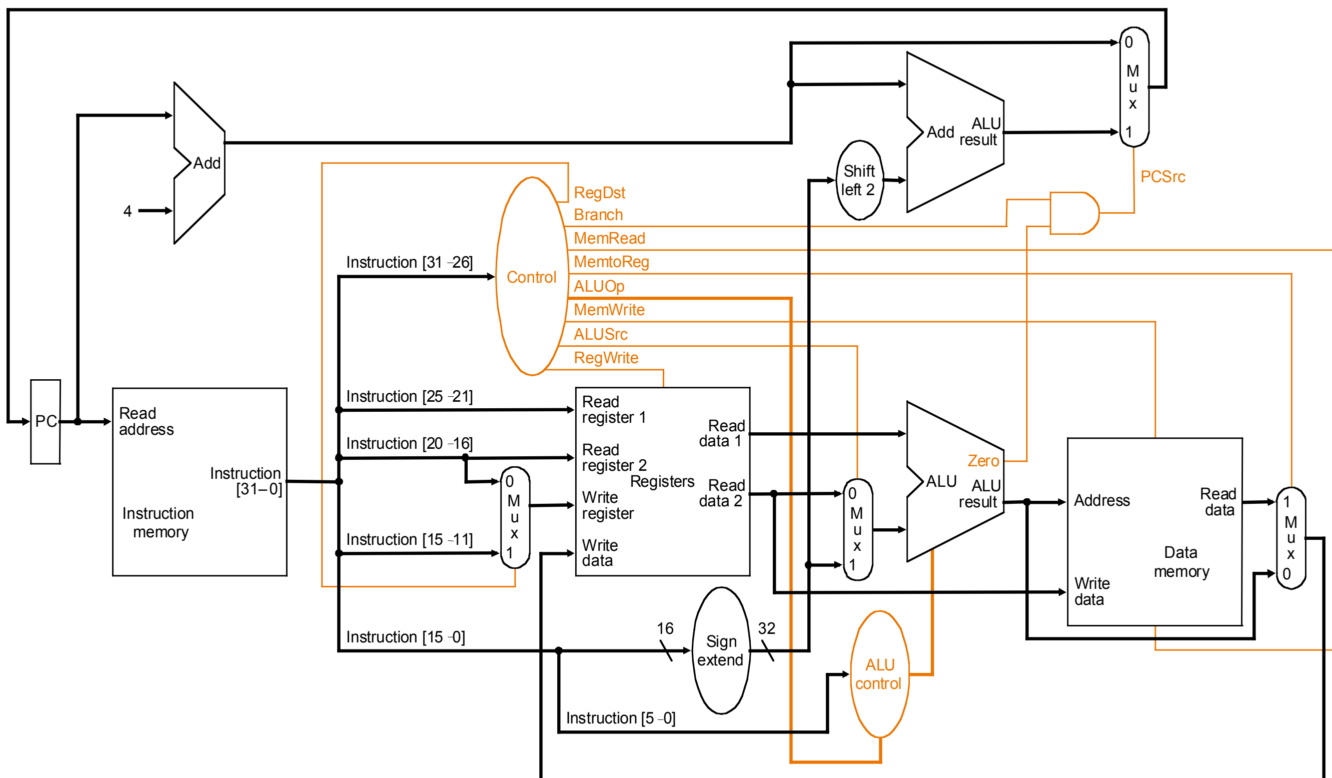
What is the clock rate for that design (in GHz)?

- Clock rate = 1 / CC
- Clock rate = 1 / 140 ps
- Clock rate = 1 / ($140*10^{-12}$)
- Clock rate = 7.14×10^{11} Hz
- Clock rate = 7.14×10^{11} Hz * (1 GHz / 10^9 Hz)
- Clock rate = 7.14 GHz

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

Question 2:

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 / 2 → 45.5 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 3:
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 x 2 = 28 ns

- However, store (sw) instruction only needs to do steps A, B, C, and D
  - Therefore a store only really needs 12 (24) ns
- Similarly, add instruction only needs to do steps A, B, C, and E
  - Therefore could do an add in 10 (20) ns
- *Still, with single CC design, both of the above instructions take 28 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 (20) ns, lw's take 14 (28) ns, and sw's take 12 (24) 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 28 ns	= 28(i)
CPU time (variable CC)	=	i x [(.45 x 20) + (.2 x 28) + (.15 x 24) + (.2 x 20)]	= 22.2(i)
Potential performance gain	=	28(i) / 22.2(i) = 26%	

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 – i.e. 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 delay, multi-cycle) = (i) x (CC / instruction) x (s / CC)

= (i) x [(.45 x 4) + (.2 x 5) + (.15 x 4) + (.20 x 3)] x 6 ns CC

= 24(i)

24(i) is not as good as 22.2(i), but its better than 28! → get speedup of 17% instead of 26%

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.

Question 4: (adding a new instruction)

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:

Item 1: Recap of LW + new functionality.

Remember what the base load does…

- lw <destination register>, offset(<register with value used to calculate address sent to memory>)
- Address sent to memory = offset + data in <register value used to calculated address…>
- Data in destination register = data at address calculated above

Now: ALSO, add 4 to <register with value used to calculate address…>

Item 2: Review RTL, discuss different options for solving problem.

- Fetch:
    - IR ← Mem(PC)
    - PC ← PC +4

  Same for every instruction.
- Decode:
    - A ← RF(IR[25:21])
    - B ← RF(IR[20:16])
    - ALUOut ← PC + (sign-extend (IR[15:0] << 2)

  Same for every instruction.

Item 2a: Digression – have assumed BEQ = 3 CCs (see Lecture 09 slide)

- Why is ALUOut line here…
- For BEQ to be done in 3 CCs, need to calculate address in CC #2
- Otherwise, would do comparison and address calculation in CC #3 and would need antoher ALU
- By doing this in CC #2, we can re-use the ALU … and it does no harm
- Strategy: do something ASAP if HW is available and idle

Item 2b: Back to lw++

- 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 # Just do normal lw first (i.e. write data back to register)

- ***Question: Can we also update $y here? Actually, there are 2 answers…***
  - Option A: Yes… but we need to add more HW…
    - (Namely another path to write data to the register file is needed…)
    - **See datapath on overheard; we'll draw in Part B.**
  - Option B: 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?***
      - Add state coming off of State 4. This would allow 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 datapath or state diagram?

SEE POWERPOINT NOTES.