CprE 381 Homework 5

[Note: This homework gives you some more practice with MIPS programming and thinking about testing your processor. As a happy coincidence, it will help provide a class-wide test program that I will share with you. Then the homework begins to cover processor design choices.]

1. Processor Implementation Details

   P&H(4.2) <§4.1>. The basic single-cycle MIPS implementation in Figure 4.2 can only implement some instructions. New instructions can be added to an existing Instruction Set Architecture (ISA), but the decision whether or not to do that depends, among other things, on the cost and complexity the proposed addition introduces into the processor datapath and control. The first three problems in this exercise refer to the new instruction:

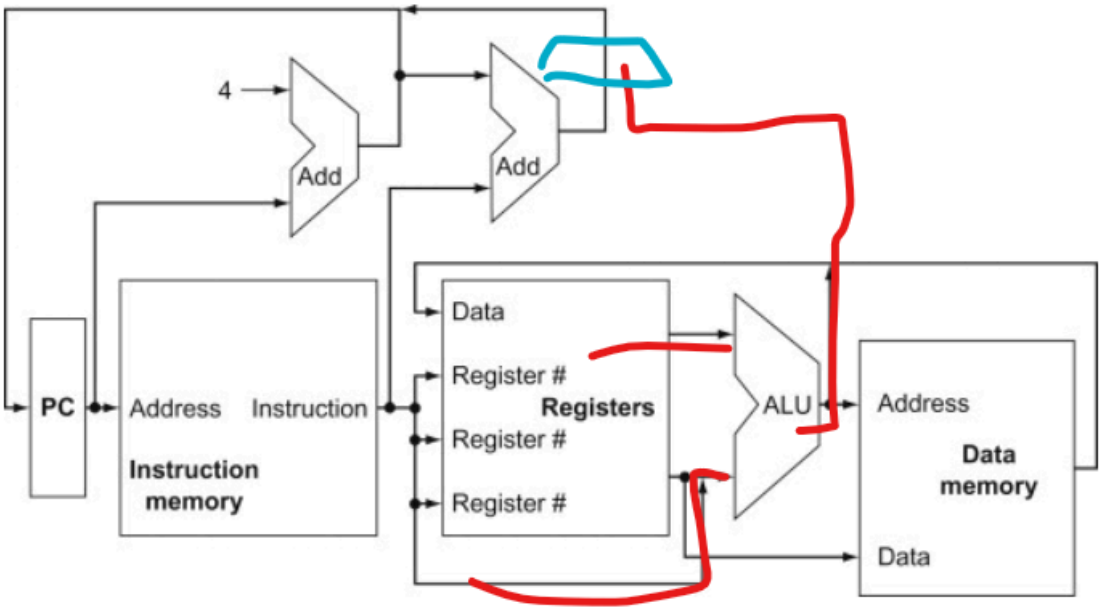
   Instruction: **`JIC`** rt, imm
   Interpretation: PC = Reg[rt]+sign_extend(imm)

   a. Which existing blocks (if any) can be used for this instruction? Briefly describe them.

   The ALU and the MIPS register file can be used together. If we used them we can use a preexisting I type instruction, even if it's wasting 5 bits for other address space, and go through a normal addiu to get the new value of PC.

   b. Which new functional blocks (if any) do we need for this instruction? Briefly describe their functioning.

   Right now the only way to increment the pc is with jump type instructions. You would probably use a immediate type instruction to implement this. The only problem is your wasting a register space value (rs). You could make a new type instruction but you would have to add additional logic in your processor to handle that. If you were using an I type instruction you can call an addi with rt and the immed value you wanted to increment. The change you would have to make though is to have a mux to choose between the pc adder value and the output of the ALU. This would make it so that you could gap the MIPS regfile and ALU output to the PC arithmetic.



c. What new signals do we need (if any) from the control unit to support this instruction? Include your control signal names (or widening of existing signals) and their specific assignment (including updated meaning of all values for widened control signals).

You would need one additional signal from the output ALU. The wire that has Add on it would need to become a signal as it would turn into the output of a mux between it and the ALU output. This mux then requires a new control line to choose between all of our preexisting pc operations and the JIC instruction.

2. Processor Cycle Time Determination

Assume the following latencies for the logic blocks in Figure 4.17 from the textbook.

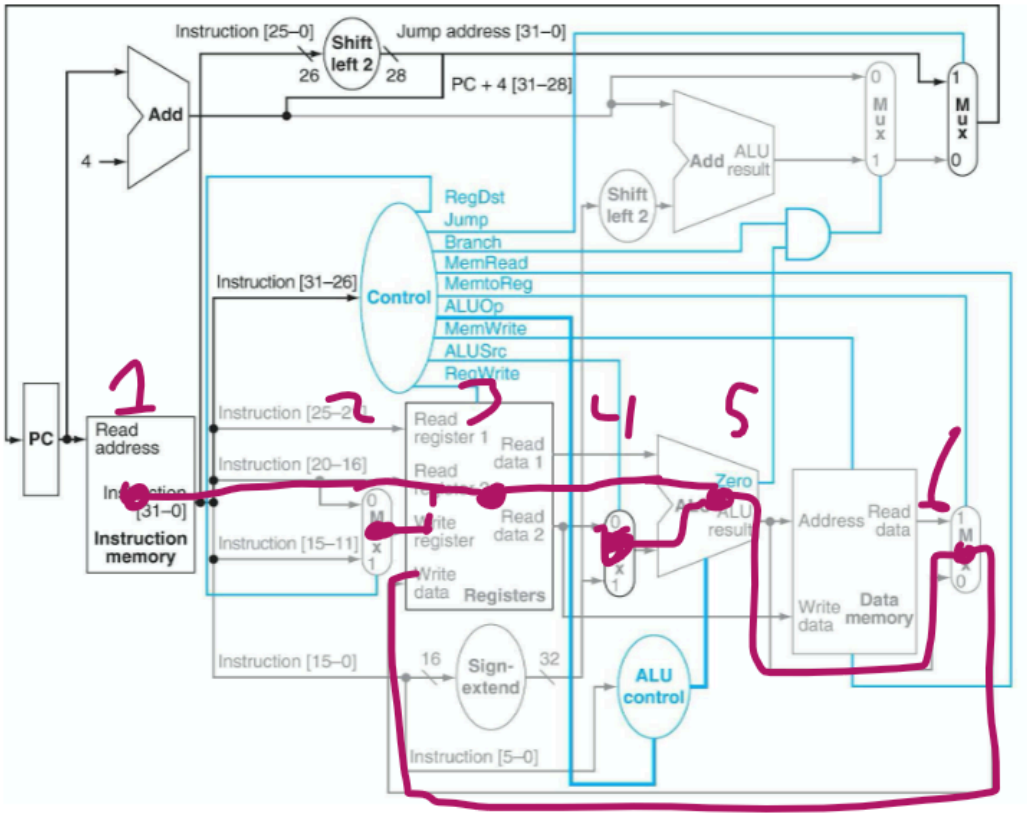
I-Mem	Adder	MUX	ALU	Reg Read	D-Mem	Sign Extend	Shift Left-2	Control	ALU Control	AND gate
225ps	85ps	15ps	100ps	110ps	340ps	15ps	10ps	70ps	15ps	10ps

a. Identify and quantify (i.e., give the path through the blocks and the time for that path) the worst-case path for each of the following: an arithmetic R-format instruction, a **`lw`** instruction, and a conditional branch instruction.

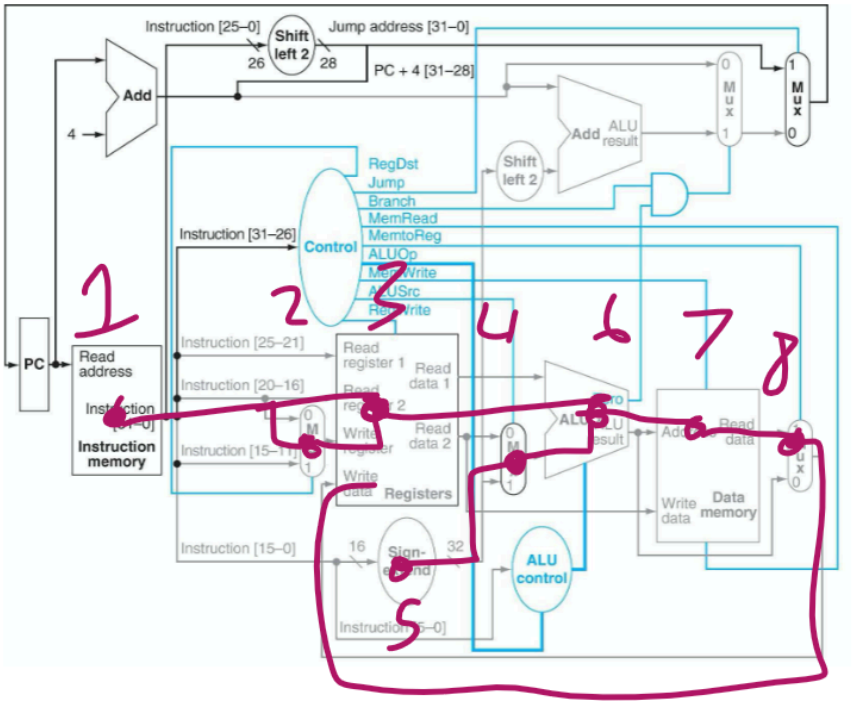
[Note that in your project you will actually synthesize your processor and be asked to identifyits critical path—you'll find out the specific delays for the components you've designed when they are mapped to an FPGA. This problem should help you develop an intuitive sense of how to reason about critical Paths.]

R Type: 225(instruction) + 15(mux) +110(reg read) + 15(mux) + 100(ALU) + 15(mux) + reg

write = 480 ps (assuming control is not part of the critical path because it is in all paths, if so they would all have +70 at the beginning) Also, PC +4 is separate so it doesn't accumulate onto the time and it takes less time than datapath from Imem back to the MIPSregfile so it isn't the critical path for this instruction.



lw instruction:
225(instruction) + 15(mux) + 110(mips regfile) + 15(mux) + 100(alu) + 340(D-Mem) + 15(mux) = 820 ps
*sign extension not included because the mips regfile takes longer than it so the sign extension will wait at the mux until the mips regfile outputs. PC datapath is takes less than the datapath in pic so is not critical path.

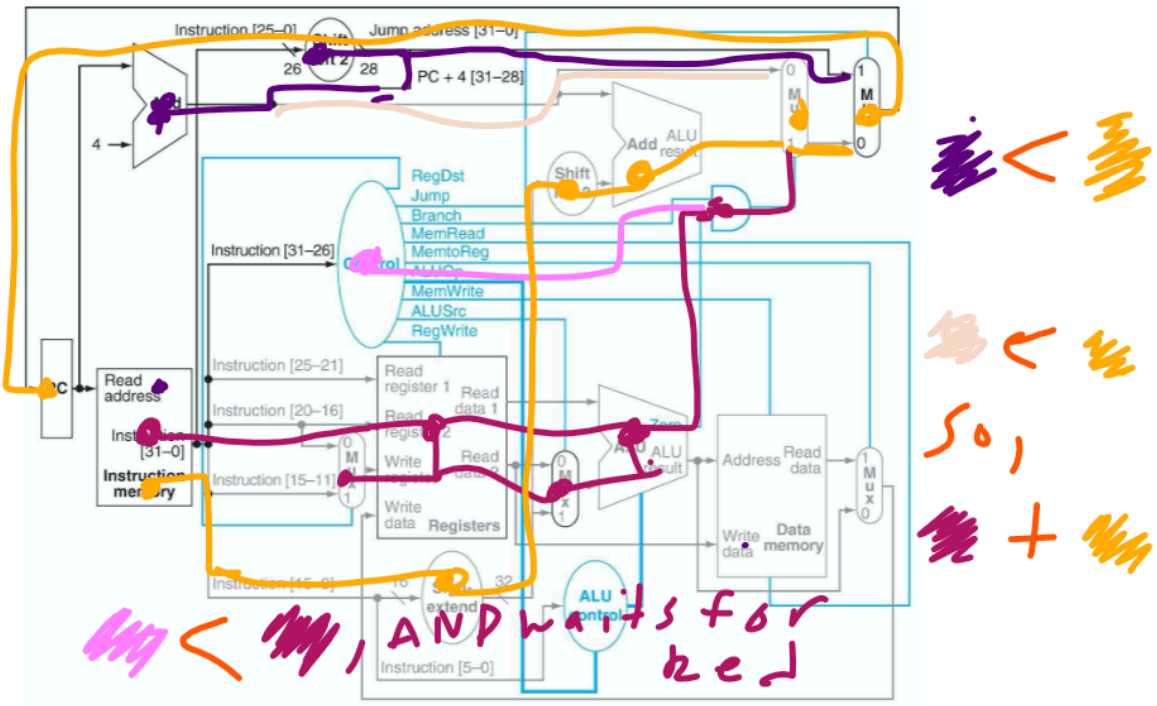


Conditional branch:
225(instruct) + 15(mux) + 110(mips regfile) + 15(mux) + 100(alu) + 10(AND) Red
110(pc read) + 15(sign extension) + 10(shift left) + 100(alu) Orange

Orange < Red so take red time, then add on rest off path getting to PC

475(Red) + 15(mux) + 15(mux) + write pc time = 505 ps

*worst case is you do beq with an immediate and the regs are equal so it jumps



b. Rank the following design approaches in terms of which improve the cycle time the most. You **must** justify your ranking.

i. Creating word addressable IMEM to eliminate branch / jump address shifting HW

This would be less relevant. The shift operation is only 10 ps. On top of that it isn't for all operations, only the branch/jump instructions. I think the other improvements, assuming it isn't a trivial amount of time reduced, would be better than this change.

ii. Implementing quicker memory to have quicker DMEM access times

This would be a good change as the D-MEM takes 340 picoseconds which is the longest operation that takes place. The only problem is that this is only for lw and sw instructions. So improving this would only improve those instructions. It depends on how much the program relies on dealing with memory. If the memory gets accessed a lot this would definitely be a major deal to help reduce times.

iii. Designing a lower-latency control unit

The plus about this is that all instructions have to use the control unit so it is a constant time added onto all instructions. The only thing is that it's a medium time operation around 70 ps where some of the other operations can be 225 to 340 which is around 3-4 times more. It is hard to say without knowing the actual numbers of how much the reduction will be. But, from the assumption that both changes will take a meaningful percent off of the execution time I think this would be the better option.

3. Performance Analysis

You are in charge of selecting processors for computing simple Natural Language Processing (NLP) tasks. You are considering two processors, A and B (the only ones you and your roommates can afford) that have the following CPIs:

Instruction Type	**Cycles per Instruction**	
	Processor A	**Processor B**
Arithmetic, Logical, Shifts, Stores	3	3
Jumps	3	2
Conditional Branch	3	2
Loads	3	5

The following applications are the primary ones that you will need to run on your system:

```
# Application 1:
# This is an implementation of a single stop word index
identification # in a string.
# $a0 contains &string (string is an asciiz array of size string_size)
# $a1 contains &stop_word (stop_word is an asciiz array of size
```

```
sw_size) # $a2 contains string_size variable
# $a3 contains sw_size variable
 addiu $t0, $0, 0           A
 j outer_cond               J
outer_loop:
 addiu $t1, $0, 0           A
 jal inner_cond             J
outer_loop_cont:
 subu $t2, $t1, $a3         A
 ori $at, $0, 1             A
 sltu $t2, $t2, $at         A
 addiu $t0, $t0, 1          A
 beq $t2, $0, outer_cond    C
 lui $at, 0x1001            2A, pseudo instruction 2 A instructions
 ori $a0, $at, 0x19         A
 addiu $v0, $0, 4           A
 syscall
 lui $at, 0                 2A
 ori $at, $at, 1            A
 subu $t0, $t0, $at         A
 addu $a0, $0, $t0          A
 addiu $v0, $0, 1           A
 syscall                    ?
 j exit                     J
outer_cond:
 subu $t2, $a2, $a3         A
 slt $10, $8, $10           A
 beq $t2, 1, outer_loop     C
 j exit                     J
inner_loop:
 addu $t2, $t0, $t1         A
 addu $t2, $a0, $t2         A
 lb $t3, 0($t2)             L
 addu $t2, $a1, $t1         A
 lb $t4, 0($t2)             L
 subu $t2, $t3, $t4         A
 sltu $t2, $0, $t2          A
 addiu $t1, $t1, 1          A
 beq $t2, 1, outer_loop_cont C
inner_cond:
 slt $t2, $t1, $a3          A
 beq $t2, 1, inner_loop     C
 jr $ra                     J
exit:

Total A: 25
Total J: 5
Total C: 4
Total L: 2

Processor A: 25 * 3 + 5 * 3 + 4 * 3 + 2 * 3 = 108 / 36 = 3
Processor B: 25 * 3 + 5 * 2 + 4 * 2 + 2 * 5 = 103 / 36 = 2.8611111

# Application 2:
# This is an implementation of simple string tokenizer.
# $a1 contains string_size variable
# $a2 contains &string (string is an asciiz array of size
string_size) # $a3 contains &delimiter (delimiter is a byte array of
size 1)
outer_cond:
 addiu $t0, $t0, 1          A
 slt $t4, $t0, $a1          A
 beq $t4, 1, outer_loop     C
 j exit                     J
outer_loop:
 addu $t4, $t0, $a2         A
```

```
 lb $t4, 0($t4)             L
 lb $t6, 0($a3)             L
 subu $t5, $t4, $t6         A
 ori $at, $0, 1             A
 sltu $t5, $t5, $at         A
 beq $t5, 1, set_ending_index C
 addi $at, $0, 0            A
 subu $t5, $t4, $at         A
 ori $at, $0, 1             A
 sltu $t5, $t5, $at         A
 beq $t5, 1, set_ending_index C
 j outer_cond               J
set_ending_index:
 addiu $t3, $t0, 1          A
 addu $t1, $0, $t2          A
 j inner_cond               J
inner_cond:
 slt $t4, $t1, $t3          A
 beq $t4, 1, inner_loop_print C
 addiu $a0, $0, '\n'        A
 addiu $v0, $0, 0xB         A
 syscall
 j set_starting_index       J
set_starting_index:
 addu $t2, $0, $t3          A
 j outer_cond               J
inner_loop_print:
 addu $t4, $t1, $a2         A
 lb $t4, 0($t4)             L
 addu $a0, $0, $t4          A
 addiu $v0, $0, 0xB         A
 syscall
 addiu $t1, $t1, 1          A
 j inner_cond               J
exit:

Total A: 20
Total J: 6
Total C: 4
Total L: 3

Processor A: 20 * 3 + 6 * 3 + 4 * 3 + 3 * 3 = 99 / 33 = 3
Processor B: 20 * 3 + 6 * 2 + 4 * 2 + 3 * 5 = 95 / 33 = 2.87
```

a. Consider the two applications above. Calculate the average CPI for each application on each processor (two applications cross two processors means you should have 4 different CPI values). Assume **`string_size`** is always 20 in both applications and **`sw_size`** is always 3 (i.e., the stop word is “the”) for application 1.

Application 1:

Processor A: 25 * 3 + 5 * 3 + 4 * 3 + 2 * 3 = 108 / 36 = 3

Processor B: 25 * 3 + 5 * 2 + 4 * 2 + 2 * 5 = 103 / 36 = 2.8611111

Application 2:

Processor A: 20 * 3 + 6 * 3 + 4 * 3 + 3 * 3 = 99 / 33 = 3

Processor B: 20 * 3 + 6 * 2 + 4 * 2 + 3 * 5 = 95 / 33 = 2.87

* You might have to actually take apart the code, there is loops so the loops might run more than once

b. Which processor has better performance? *[Careful answering this part…it is a tricksy professor question.]* Provide the quantitative evidence of "better performance" and include a description of how you evaluated the two applications together. What would the relative frequencies have to be between the two processors in order for their performance to be identical (i.e., calculate the "breakeven frequency")?

It depends. If the processor is a multi cycle pipeline processor it will then benefit from having a higher cpi as it will have a higher throughput. If it is a non pipelined multi cycle processor then it would depend upon the time per cycle as the one processor with a lower CPI might have a longer clock speed and end up having a worse performance than the processor with the higher cpi.

CPI1 * (s/cyc) * IC = CPI2 *(s/cyc) * IC
=> CPI1 * f1 = CPI2 * f2
=> 3 * f1 = 2.87 * f2
=> 1.045f1 = f2

I evaluated the two applications by assuming that every instruction would run once and counted the total of each type of instruction and calculated accordingly.