Final Solutions

CSEE W3827 - Fundamentals of Computer Systems — May 10, 2018
Spring 2018 — Prof. Rubenstein

This final contains 5 questions (not counting question 0), totaling 120 points. Question 0 gives an additional 5 points. **BOOKS, NOTES, ELECTRONIC DEVICES ARE NOT PERMITTED!** The time allowed is 3 hours.

Please answer all questions **in the blue book**, using a **separate** page for each question. **Show all work!** We are not just looking for the right answer, but also how you reached the right answer.

QUESTION 0 (5 points off if you don't do this): write your name **CLEARLY**: LAST NAME, FIRST NAME, and UNI on the cover of the blue book and start each of the remaining questions on a new page. If, when sorting the exams according to last name, yours is sorted incorrectly because of a lack of clarity, you lose the 5 points.

1. (30 pts) A lighting system has 3 modes: OFF, DIM, and BRIGHT. In each clock cycle, the system is in one of these 3 modes, and takes a 2-bit input $I = I_1 I_0$, to determine its mode in the next clock cycle as follows:

   - If in OFF or BRIGHT mode during clock cycle $t-1$ and input $I(t) = 00$, the system stays in the same mode for clock cycle t.
   - If in OFF mode in clock cycle $t-1$ and input $I(t) = 10$ is received, the system switches to BRIGHT mode for clock cycle t. Similarly, if in BRIGHT mode for clock cycle $t-1$ and input $I(t) = 10$ is received, the system switches to OFF mode for clock cycle t.
   - If in OFF mode in clock cycle $t-1$ and input $I(t) = 11$ is received, the system switches to DIM mode for clock cycle t, and BRIGHT mode for clock cycle $t+1$. Similarly, if in BRIGHT mode in clock cycle $t-1$ and input $I(t) = 11$ is received, the system switches to DIM mode for clock cycle t and OFF mode for clock cycle $t+1$.

   Some observations:

   - the system cannot be in DIM mode for two consecutive clock cycles.
   - The input has no effect for the clock cycle when the system is in DIM mode (the next mode was determined by the input of the previous clock cycle).
   - It is appropriate to assume input 01 never occurs.

   You are to design a sequential circuit using JK flip-flops that generates a 2-bit output $O(t) = O_1(t)O_0(t)$ each clock cycle that specifies the mode of the lighting system for that clock cycle t as follows:

$O_1(t)$	$O_0(t)$	mode
0	0	Off
1	0	Dim
1	1	Bright

   An example follows, assuming the initial state is OFF.

clock cycle(t)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$I(t)$	00	00	10	00	10	00	00	11	00	00	11	11	11	00	10
$O(t)$	00	00	11	11	00	00	00	10	11	11	10	00	10	11	00

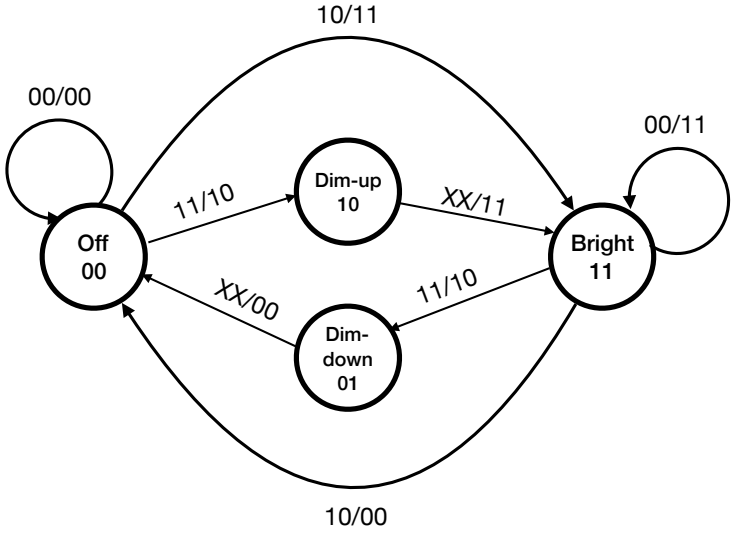
   Produce algebraic expressions that feed into the JK inputs of the flip-flops you use, as well as algebraic expressions for $O_1(t)$ and $O_0(t)$. For full credit, your solutions need to be simplified (and produce the correct behavior).

   Any partial work (e.g., state machine, excitation table, K-map, etc.) will be considered and can be used for partial credit.

   For your reference, the following table excitation table describes the behavior of a JK flip-flop:

$J(t)$	$K(t)$	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\overline{Q(t)}$

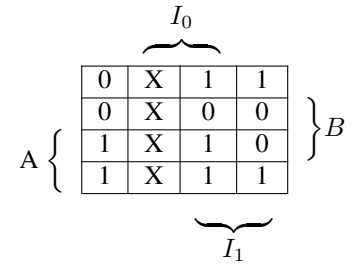
Answer: There are several ways to design this circuit. However, the circuit requires a minimum of 4 states: a state to represent OFF, a state to represent BRIGHT, and two states to represent DIM, one when the next state will be OFF, the other for when the next state will be BRIGHT. The State Machine is pictured below.



The excitation table of this state machine using two JK flip flops with A representing the high order bit of the state and B representing the low-order bit is as follows:

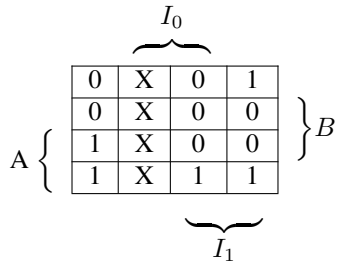
$A(t)$	$B(t)$	$I_1(t)$	$I_0(t)$	$O_1(t)$	$O_0(t)$	$A(t+1)$	J_A	K_A	$B(t+1)$	J_B	K_B
0	0	0	0	0	0	0	0	X	0	0	X
0	0	0	1	X	X	X	X	X	X	X	X
0	0	1	0	1	1	1	1	X	1	1	X
0	0	1	1	1	0	1	1	X	0	0	X
0	1	0	0	0	0	0	0	X	0	X	1
0	1	0	1	X	X	X	X	X	X	X	X
0	1	1	0	0	0	0	0	X	0	X	1
0	1	1	1	0	0	0	0	X	0	X	1
1	0	0	0	1	1	1	X	0	1	1	X
1	0	0	1	X	X	X	X	X	X	X	X
1	0	1	0	1	1	1	X	0	1	1	X
1	0	1	1	1	1	1	X	0	1	1	X
1	1	0	0	1	1	1	X	0	1	X	0
1	1	0	1	X	X	X	X	X	X	X	X
1	1	1	0	0	0	0	X	1	0	X	1
1	1	1	1	1	0	0	X	1	1	X	0

O_1:



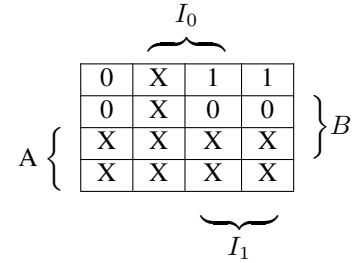
$O_1 = A\overline{I_1} + AI_0 + \bar{B}I_1$

O_0:



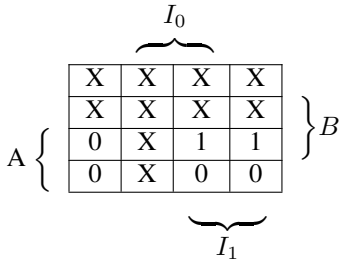
$O_0 = A\overline{I_1} + A\bar{B} + \bar{B}\overline{I_0}I_1$

J_A:



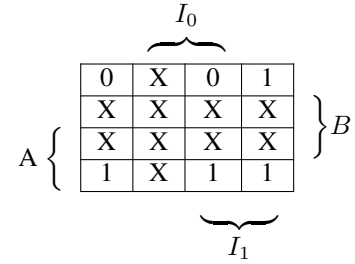
$J_A = \bar{B}I_1$

K_A:



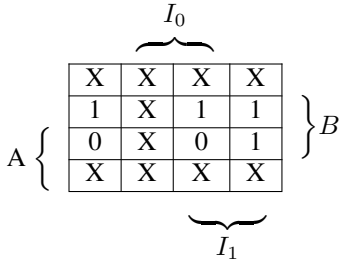
$K_A = BI_1$

J_B:

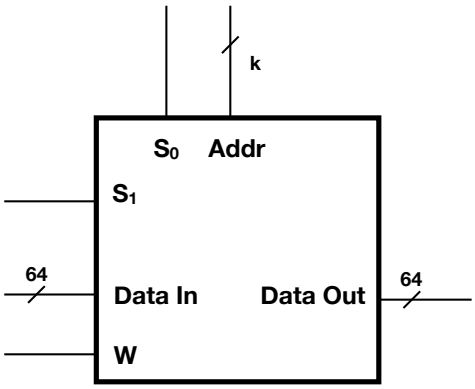


$J_B = A + \overline{I_0}I_1$

K_B:



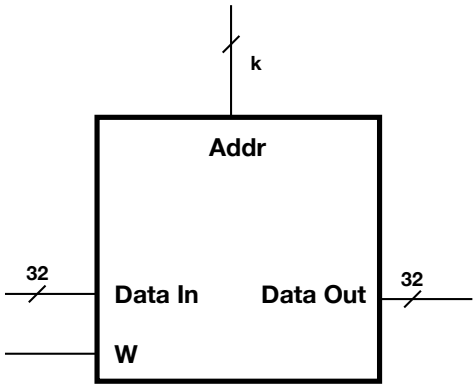
$K_B = \bar{A} + \overline{I_0}I_1$



2. (30 pts) The memory chip pictured above can be used to store 2^k 64-bit words, or alternatively can be used to store 2^{k+1} 32-bit words. It behaves as follows:

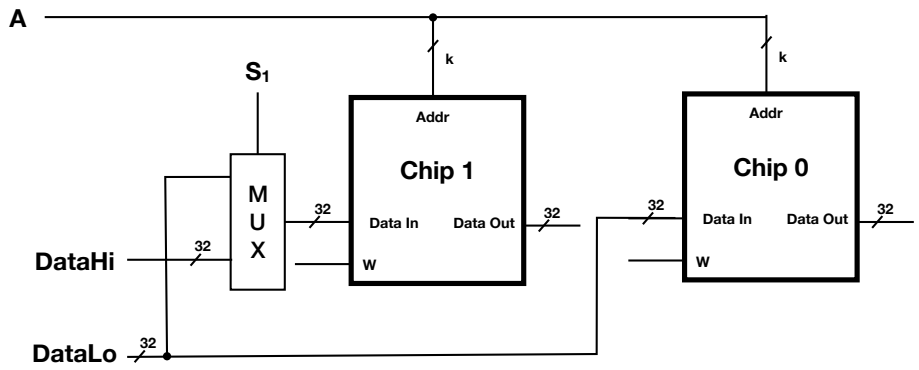
   - It takes a 64-bit DATA input, I, a k-bit address input $A = A_{k-1}A_{k-2}\cdots A_1A_0$, a 1-bit WRITE input W, and 2-bit selector S_1S_0, and produces a 64-bit DATA output O,
   - When $W = 1$, the memory is written to.
   - When $S_1 = 1$, the chip reads or writes 64 bits of data from/to address A.
   - When $S_1 = 0$, the chip reads or writes 32 bits of data with the following behavior:
     - the address is specified by $S_0A_{k-1}A_{k-2}\cdots A_1A_0$ (i.e., a $k+1$-bit address).
     - When writing to memory, data is written from the 32 least significant bits of the input, i.e., the most significant 32 bits are unused for write.
     - When reading from memory, the data is written to the 32 least significant bits of the output, and the 32 most significant bits are all set to 0.

   In this problem, you will build this chip from two 32-bit word 2^k address chips (one is pictured below) and basic circuitry (e.g., MUXes, Decoders, AND, OR, NOT gates).



   (a) (10 pts) Let's enumerate the two chips you will use as Chip 0 and Chip 1. Draw circuitry showing what feeds into the DataIn and Addr inputs of these two chips (i.e., in terms of S_0, S_1, A, and I).

   Answer: A feeds in as the address on both chips. We split the 64-bit input into two 32-bit chunks, DataHi and DataLo. When $S_1 = 1$, we need DataHi feeding into one chip (we'll use Chip1) and DataLo feeding into the other (Chip0). When $S_1 = 0$, we want to use one chip to store addresses where $S_0 = 1$ and the other chip when $S_0 = 0$. We'll use Chip i for when $S_0 = i$. So, we always feed DataLo into Chip 0, and into Chip 1 we feed DataHi when $S_1 = 0$, and DataLo when $S_0 = 0$.

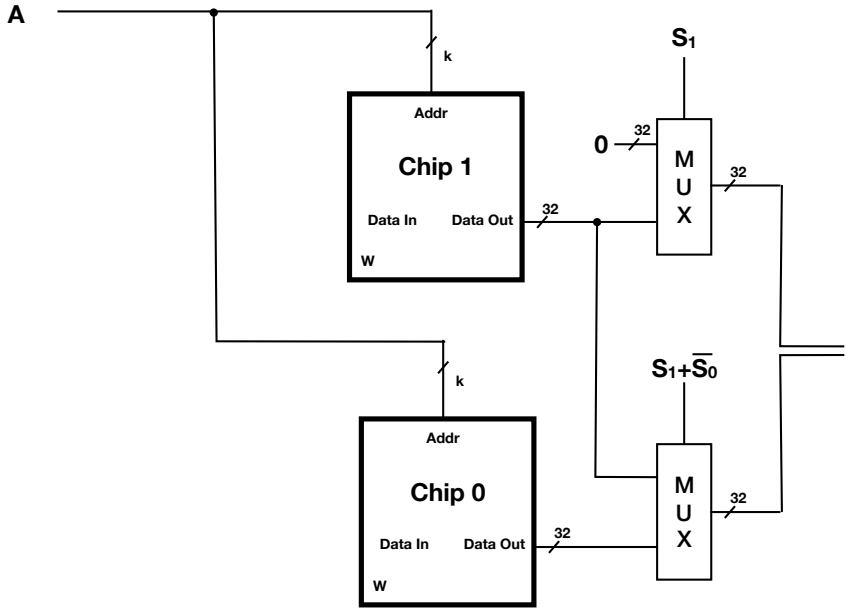


(b) (10 pts) Give algebraic expressions for W_i, which will feed into Chip i's W input.

Answer: We want to write to both chips when $S_1 = 1$, and when $S_1 = 0$, we only want to write to chip i when $S_0 = i$. Hence, $W_1 = W(S_1 + S_0)$ and $W_0 = W(S_1 + \overline{S_0})$.

(c) (10 pts) Draw (in a separate diagram from part a) the circuitry to produce the 64-bit output of the chip being built.

Answer: Let OutHi and OutLo describe 32-bit chunks of the output we wish to generate, with OutHi being the 32 high order-bits and OutLo being the 32 low-order bits. We again feed A into the address input. When $S_1 = 1$, we want OutHi to come from Chip 1 and OutLo to come from chip 0. When $S_1 = 0$, then we want OutHi to be all 0's, and OutLo to come from Chip i when $S_0 = i$. Hence S_1 feeds into a MUX for OutHi that chooses between Chip 1 output and 0. $S_1 + \overline{S_0}$ feeds into a MUX that chooses between Chip 1 and Chip 0 outputs.



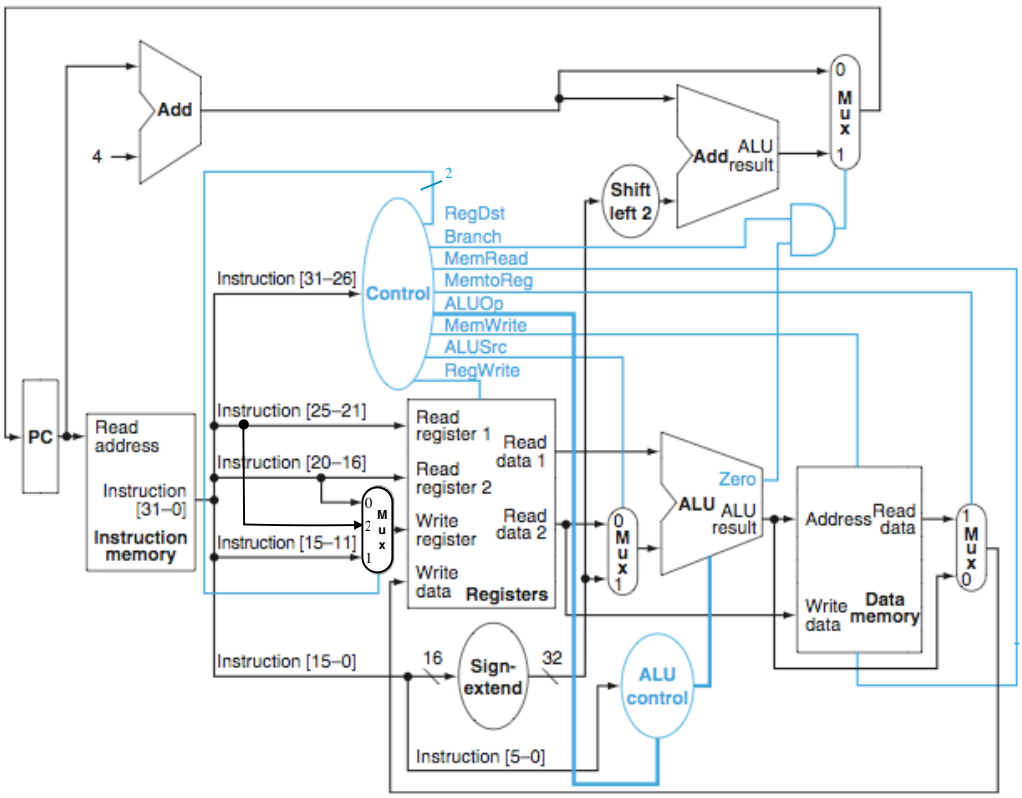
3. (30 pts) Consider the following MIPS code:

```
LOOPSTART:
  addi $s1, $s1, 4
  lw $t0, 0($s1)
  beq $t0, $s2, END
  j LOOPSTART
END:
```

An address A is initially stored in \$s1, and a data value D is stored in \$s2. The code iterates through memory starting at the address $A + 4$ until it determines the smallest $i \geq 1$ for which $M[A + 4i] = D$.

You are to design a new MIPS instruction, rpe, for the **single-cycle architecture** that performs this operation. Namely:

- rpe \$reg1, \$reg2 finds the first address beyond the value in \$reg1 for which the value stored in \$reg2 occurs.
- The instruction differs from other MIPS instructions in the single cycle architecture in that it can (and often will) run for multiple clock cycles until the match is found, adding 4 to the value in \$reg1 each cycle.
- when the instruction completes (perhaps after multiple clock cycles), the value the address it found will be stored in \$reg1.
- Instead of the code snippet above, a user could simply use the instruction rpe \$s1, \$s2 to obtain a similar result, and in fewer clock cycles.



Use the schematic above to help answer the following questions (note that jump functionality is not included in this diagram, only branch). Also note the minor addition to the MUX feeding into the write register (an obvious hint).

(continued on next page)

(This is a continuation for question 3)

(a) (10 pts) Indicate how the "Control" circuit that parses the instruction into various selectors should behave for rpe. In particular, what are RegDst, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite each set to? Note RegDst is now a 2-bit value. For ALUOp, the following table indicates how ALUOp affects the computation of the ALU:

ALUOp	Operation
00	add
01	subtract
10	(see 6-bit instruction code fed into ALU control)
11	unused

Answer: Generally, we want to read in the two registers $reg1 and $reg2 specified, use the ALU to ADD a CONSTANT 4 to the first register $reg1, feed this in as an address, and also write the value of $reg1$+4$ back to $reg1. So:

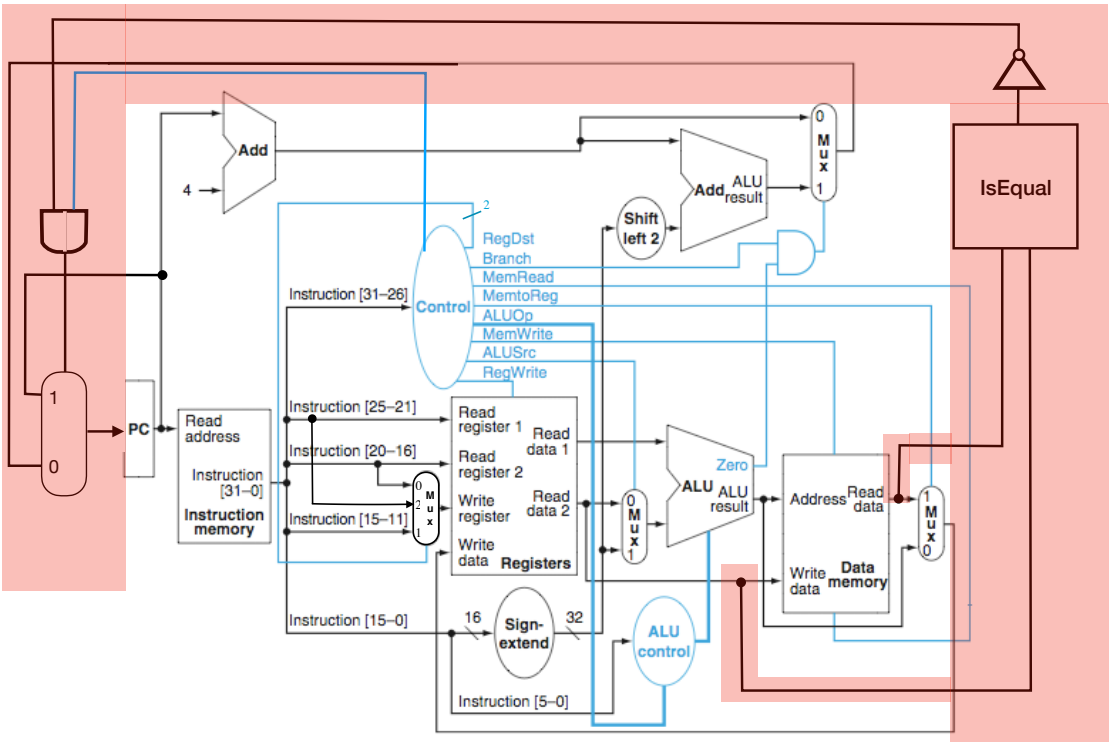
- RegDst = 10 (i.e., 2), choose the same bits used as read register 1 to write to
- MemtoReg = 0, write the result of the ALU back to the register file
- ALUOp = 00, we want to ADD $reg1 and 4 together
- MemWrite = 0, there is no writing to memory
- ALUSrc = 1, we want the CONSTANT to add to $reg1
- RegWrite = 1, we want to write $reg1 $+4$ back to $reg1

(b) (10 pts) Indicate the bit layout of the instruction. For instance, the six highest order bits (31-26) will be the op-code. How do you use the remaining bits to specify $reg1, $reg2, and any other information that might be relevant?

Answer: Bits 25-21 correspond to $reg1, bits 20-16 correspond to $reg2, and bits 15-0 represent the constant 4, i.e., $0000\ 0000\ 0000\ 0100$.

(c) (10 pts) Assume the "Control" circuit has one additional output, not pictured above: IsRPE, which is set to 1 only when the instruction is rpe. Using this additional control output and an IsEqual circuit that takes two 32-bit inputs and returns 1 when the inputs are equal, show how to use these to appropriately modify the program counter.

Answer: When the instruction is a rpe instruction and IsEqual is false, the instruction is to be repeated again. So there must also be an option to provide a "stall" to the PC, i.e., feed it's own value back into itself. This can be done by adding a MUX with the same PC value being selected when IsRPE is true and IsEqual is false. The additional circuitry is shown in the shaded region.



4. (15 pts) The following 5 code snippets consist of an add instruction followed by a branch instruction. Normally, when a branch instruction is pulled in during the first (IF) stage, the information needed to determine the outcome of the branch is not available until the branch instruction reaches a later stage, so either stalling or *branch prediction* is applied. However, in some instances, the information that would be needed by the instruction already exists elsewhere in the pipeline, and could conceivably be forwarded to the IF stage and evaluated there. When this is possible, it would allow a determination to be made within the IF stage so that branch prediction is not needed.

   Suppose, for instance, you are permitted to use an IsEqual circuit within the IF stage that takes two 32-bit inputs and outputs 1 when the two inputs are equal. This, along with data forwarding and some basic (AND,OR,NOT,MUX) circuitry, could sometimes be used to determine the outcome of the branch before it exited the IF stage.

   Which, if any, of the code snippets below, could the branch outcome be determined within the IF stage? Be sure to explain your answer.

   (a)
   ```
   add $s0,$s1,$s2
   beq $s1, $s3, LABEL
   ```

   (b)
   ```
   add $s0,$s1,$s2
   beq $s2, $zero, LABEL
   ```

   (c)
   ```
   add $s0,$s1,$s2
   beq $s0, $zero, LABEL
   ```

   (d)
   ```
   add $s0,$s2,$s0
   beq $s0, $s2, LABEL
   ```

   (e)
   ```
   add $s0,$s1,$s2
   beq $s1, $s2, LABEL
   ```

   Answer: Normally, all information is available to the branch immediately except the two registers that must be compared to one another, which must be pulled from the register file in the ID stage. However, in some of these examples, the register values needed by the branch are brought into the pipeline by the preceding add instruction. For each case:

   - a: No: $s3 is not in the pipeline yet
   - b: Yes: $s2 is in the pipeline and $zero is known to have the value 0.
   - c: No: depends on the value of $s0 computed by the add, which does not happen until the EXE stage (add will only be in the ID stage when the beq is in the IF stage).
   - d: No: same reason as c. [One student has pointed out that if you were permitted to check if $s0 is 0, then you could in fact make the determination within IF because the add is effectively a NOP]
   - e: Yes: both $s1 and $s2 will be in the pipeline.

5. (15 pts) Suppose a second EXE stage is added after the MEM stage, such that the order of stages is IF, ID, EXE, MEM, EXE, WB, and that each EXE stage has its own separate ALU. A MIPS instruction would normally utilize only the first EXE stage and do nothing with the ALU in the second EXE stage, but, in the event that a stall would be required to use the first EXE stage, the instruction could perhaps instead move the instruction through the pipeline, not use the ALU in the first EXE stage and complete its execution with the ALU in the second EXE stage.

   For instance, consider the following code snippet:

```
lw $s0, 8($s1)
add $s2, $s0, $s3
```

   This code would normally require a stall between the lw and add instructions. However, in the modified pipeline described above, the add could proceed behind the lw, skip the ALU in the first EXE stage and utilize the ALU in the second EXE stage (when the lw instruction is in the WB stage, to perform its necessary computation.

   (a) (5 pts) Suppose the ALU in the second EXE stage is never utilized, such that the stage does not perform any function other than passing the information it receives from the MEM stage to the WB stage, and (when needed) data forwarding to other stages. In other words, compared to the original MIPS 5-stage pipeline, this 6-stage pipeline has a fifth stage that doesn't perform any computation, read, or write to memory or register file. If a given program required N clock cycles to complete in the original 5-stage architecture, how many clock cycles will it require in this revised 6-stage architecture?

   Answer: With this new stage not doing anything, each instruction takes an additional clock cycle to go through the pipeline. Since this additional stage happens after MEM, it doesn't alter if and when instructions would have to stall. Hence, the instructions proceed through the pipeline as before, and each instruction will exit the extended pipeline exactly one clock cycle after it would have in the original pipeline. Hence the running time is $N + 1$ clock cycles.

   (b) (10 pts) Now suppose that the ALU can be utilized in this new fifth stage of the 6-stage pipeline. While this additional stage alleviates some stalls as shown above, some stalls can still occur. Give an example snippet of code that would still stall in this new architecture. Provide a short explanation (1 or 2 sentences) of why your example snippet must still cause a stall.

   Answer:
   lw $s0, 0($s1)
   sw $s2, 16($s0)

   The above will cause a stall. The sw requires the value of $s0 returned by lw, so the lw instruction must already proceed through the MEM phase. Without stalling the sw, the sw instruction will enter the EXE phase prior to having this needed information. Also, the sw needs to perform its write to memory prior to the second EXE stage, so it cannot use the latter stage to compute the address it will be writing to.

   The latter instruction could also be a lw instruction.

You have reached the end of the exam. Have a great summer!