End-of-term Exam Solutions

CSEE W3827 - Fundamentals of Computer Systems — April 30, 2019 and May 2, 2019
Spring 2019 — Prof. Rubenstein

This portion of the final contains 3 questions (not counting question 0), totaling 90 points. Question 0 gives an additional 10 points. **BOOKS, NOTES, ELECTRONIC DEVICES ARE NOT PERMITTED!** The time allowed is 150 minutes.

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.

SECTION 1 students: Please put your UNI on your exam copy and submit it at the end. Do not take with you!

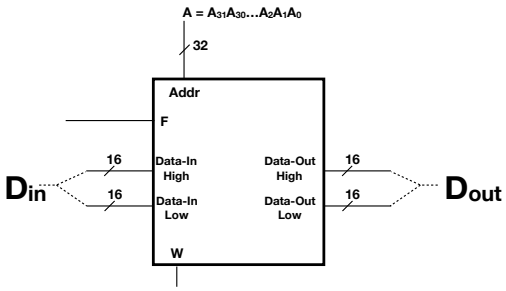
YOUR UNI: _____

SECTION 2 students: You may keep your copy of the exam.

Some advice:

- Be sure to leave some time to work on each problem. The right answer to each problem does not require a very long answer.
- Be sure to start every problem. And take some time to think about how to set the problem up before you start writing.

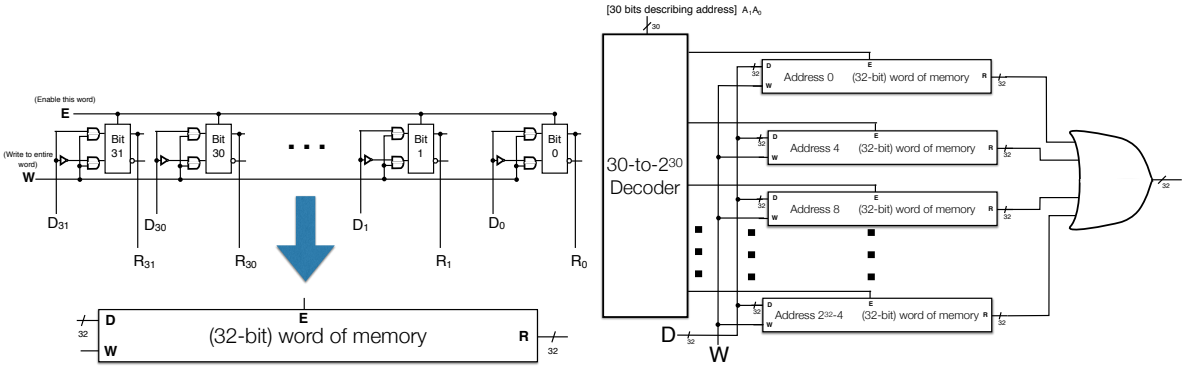
0. (5 pts) Do the following in the blue book:
   (a) CLEARLY write your
      - name **and UNI**
      - **lecture section number** (1 for 10:10 class, 2 for 11:40 class) on the front cover.
      - P-credit TA and section number (or day/time)
   (b) start each numbered question's solution on a new page. So question 2 should start on a new page, question 3 on a new page, etc.
   (c) label solutions (e.g., 2a, 2b, 2c or 2a, b, c)



1. (30 pts) The memories designed in class could read and write one **full word** of data (32 bits) at a time. To extend the design to also handle **half-words** (16 bits), an additional 1-bit input F is added to the chip as shown above.

   - When $F = 1$ the chip operates in traditional **full-word mode**, with $A = A_{31}A_{30} \cdots A_3A_2A_1A_0$ mapping to the **4 byte addresses** associated with **word address** $A_{31}A_{30} \cdots A_3A_200$. When $W = 1$, 32-bit D_{in} is written to the word of memory at this address. 32-bit D_{out} outputs the value stored at this address.
   - When $F = 0$, the chip operates in **half-word mode**. Here, address bit A_1 now plays an important role.
     - $A_1 = 1$ indicates that the 16 high-order bits of the word at address $A = A_{31}A_{30} \cdots A_3A_200$ should be accessed.[1] These bits should be output through Data-Out High (with 0's padding Data-Out Low), and when $W = 1$, the 16 high-order bits of D_{in} (passed into Data-In High) should be written to these 16 memory bits.
     - $A_1 = 0$ indicates that the 16 low-order bits of the word at address $A = A_{31}A_{30} \cdots A_3A_200$ should be accessed.[2] These bits should be output through Data-Out Low (with 0's padding Data-Out High), and when $W = 1$, the 16 low-order bits of D_{in} (passed into Data-In Low) should be written to these 16 memory bits.

   The two figures below depict internals of memory (from class) that only permits full-word read and store:



   (a) (15 pts) Indicate what changes/additions need to be made **internally to the chip** to enable the above. Just show places in the circuitry where you are making changes and explain what is needed to someone who knows what the right answer looks like. You should not need to redraw the entire memory diagram.

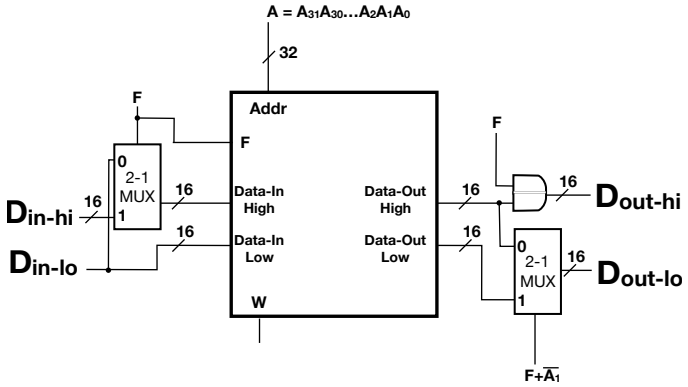
[1] which would be byte addresses $A = A_{31}A_{30} \cdots A_3A_210$ and $A = A_{31}A_{30} \cdots A_3A_211$.
[2] which would be byte addresses $A = A_{31}A_{30} \cdots A_3A_200$ and $A = A_{31}A_{30} \cdots A_3A_201$.

Answer: All that needs to be altered is the enable signal to each latch. High-order bits of a word should be enabled when $F = 1$ or when $F = 0$ and $A_1 = 1$, and low-order bits of a word should be enabled only when $F = 1$ or $A_1 = 0$. These can respectively be represented by algebraic expressions $F + A_1$ and $F + \overline{A_1}$.

Modification to the enable input for the ith most significant bit of a word of memory.

(b) (15 pts) In MIPS, half-word loads and stores only utilize the low-order bits of D_{in} and D_{out}, so while A_1 still indicates which bits of memory are accessed, half-words are input to the chip through Data-In Low, and are output through Data-Out Low (with Data-Out High always being all 0's when in half-word mode). Construct additional circuitry **external to the chip** such that full-word mode can use all 32 bits of input/output data and that half-word mode always uses the 16 least significant of input/output when interfacing with the memory.

Answer: For the input side when in full-word mode ($F = 1$), Data-In High should take in the high-order bits D_{in} and Data-In Low should take in low-order bits. Whereas when $F = 0$, it's either the case that $A_1 = 1$ and Data-In High should take in the low-order bits of D_{in}, or $A_1 = 0$ and Data-In Low should take in the low-order bits of D_{in}. In general, when $F = 0$, we can be feeding the low-order bits of D_{in} into both the High and Low inputs of Data-In.
For the output side, the high output bits come from Data-Out High only when $F = 1$, and should otherwise be 0, so these output bits just need to be ANDed with F. The low-order output bits are taken from Data-Out Low when either $F = 1$ **OR** when ($A_1 = 0$ **AND** $F = 0$). Hence we can MUX on the Data-Out High (tied to the selector 0 input of the MUX) And Low (tied to the selector 1 input of the MUX) with $F + \bar{F}\overline{A_1}$ as the selector, with the observation that this can be simplified to $F + \overline{A_1}$ (i.e., recall in general that $X + \bar{X}Y = X + Y$).



2. There are three important rules in MIPS code design, one of which we discussed, the other two we did not:

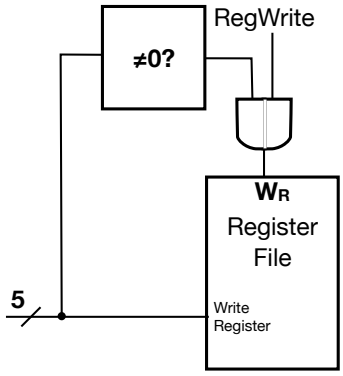
   - Register number 0 in the register file (a.k.a. the $zero register) should always equal 0 and never be changed (i.e., it should never be the destination register of an instruction).
   - Branches should not select themselves as their destination, e.g., don't do:
     `MYLABEL: beq $s0, $s1, MYLABEL`
     since a true conditional will create an endless self-loop.
   - Memory addresses 0 through 65535 (i.e., $2^{16} - 1$) are read-only and should not be written to.

   (a) (6 pts) Suppose a procedure is assembled. Which of the above rules might an assembler not be able to detect a violation of? Give a **short one sentence explanation** as to why it may fail to detect the breaking of each such rule.

   (b) (24 pts) Even though some of the above violations are always detectable by the assembler, it would still make sense for hardware to be added to the single cycle architecture to enforce these rules. On the page 9 of this exam is a figure depicting the MIPS single cycle architecture presented in class. In three separate diagrams (8 pts each), modify the architecture such that:

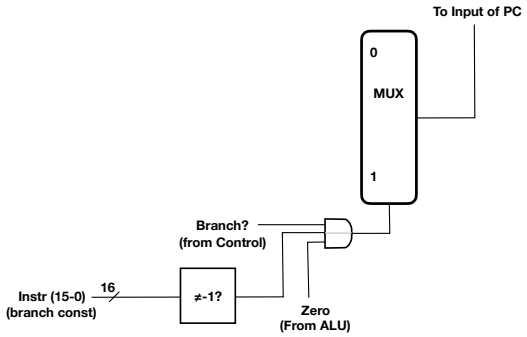
      i. A MIPS instruction that attempts to alter the value in the $zero register winds up doing nothing (i.e., becomes a NOP)

Answer: If the 5-bit write register equals 0, then W_R (writing to register file) should be disabled for this clock cycle. So AND the conditional of (Write Register not being 0) with Control's decision (from reading the OpCode) of the RegWrite signal.



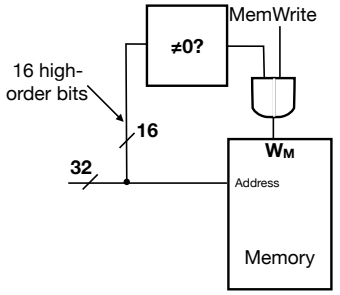
ii. A branch instruction that attempts to branch to itself is simply treated as a NOP and the instruction immediately below the branch is processed in the subsequent clock cycle (don't worry about fixing jumps).

Answer: If the constant equals -1 and this is a branch instruction, then we want to override any decision to branch. Hence, we include an additional input into the AND gate that is confirming that this is a branch and the conditional is true (i.e., zero? from ALU) that the constant is not -1.

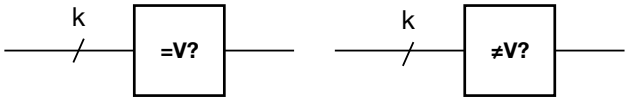


iii. Any MIPS instruction that tries to write to memory will have no effect on state when the specified memory address is less than 65,536.

Answer: The address is less than 65,536 when and only when the 16-high order bits of the address are all 0's. So we need to disable writing to memory in only this case by ANDing the MemWrite signal with a check that the 16-high order bits are not 0.



In addition to the typical circuits (MUXes, ANDs, ORs, Decoders, etc.), you may also use circuits of the following form:



which takes a k-bit input (you can choose k) and returns a single bit which is 1 when the input, viewed as a 2's-complement value, equals V (or alternatively does not equal V).

Remember, we know what the right answers look like. **You only need to draw where you are making changes to the architecture, and each part can be drawn in a separate diagram.** Don't spend time redrawing stuff we already know or trying to merge the three parts together into one diagram. But make sure when use the equalsV circuit above that you specify k and V.

3. (30 pts) Recall that procedures in MIPS can be invoked via a call to `jal PROCNAME` where `PROCNAME` is the label indicating the starting address of the procedure, and the procedure can return control to its caller via `jr $ra`. Consider the implementation of these two instructions in the MIPS pipelined architecture. Since both perform a functionality similar to a jump, assume that, like a jump, both instructions make their modification to the program counter ($PC) when in the ID stage. Also, it may be useful to recall that the $ra register is a register in the register file: it's number 31 (binary 11111).

   On page 10 of the exam is the circuit diagram of the MIPS pipeline with the branch instruction completing its conditional evaluation and also making its modification to the $PC (when the conditional is true) within the ID stage. The circuitry to compute the jump instruction, also taking place in the ID stage, has been added and drawn in red.

   (a) (8 pts) For each pipeline stage, describe what `jal` must do (at most one sentence per stage) to be correctly implemented. Note that if there is another instruction that does some of what `jal` must do, you can start with "everything that [other instruction] does and...".

   Answer:

   - IF stage: instruction is just pulled from instruction memory
   - ID stage: everything jump does but also continue to forward X =$PC+4 down the pipeline
   - EX stage: nothing other than keep forwarding X down the pipeline.
   - MEM stage: nothing other than keep forwarding X down the pipeline
   - WB stage: write the value of X into register $ra.

   (b) (7 pts) For each pipeline stage, describe what `jr` must do (at most one sentence per stage) to be correctly implemented (same rules as part (a)).

   Answer:

   - IF stage: instruction is just pulled from instruction memory
   - ID stage: everything jump does but also pull the contents of the $ra register from the register file and store this result in the $pc (involves adding one more MUX).
   - EX stage: nothing
   - MEM stage: nothing
   - WB stage: nothing

   (c) (15 pts) Consider the following 8-line code snippet:

```
   sw $ra, -4($sp)        #line 1
   addi $sp, $sp, -4      #line 2
   jal LABEL              #line 3
   lw $ra, 0($sp)         #line 4
   addi $sp, $sp, 4       #line 5
   jr $ra                 #line 6
LABEL:                    #line 7
   jr $ra                 #line 8
```

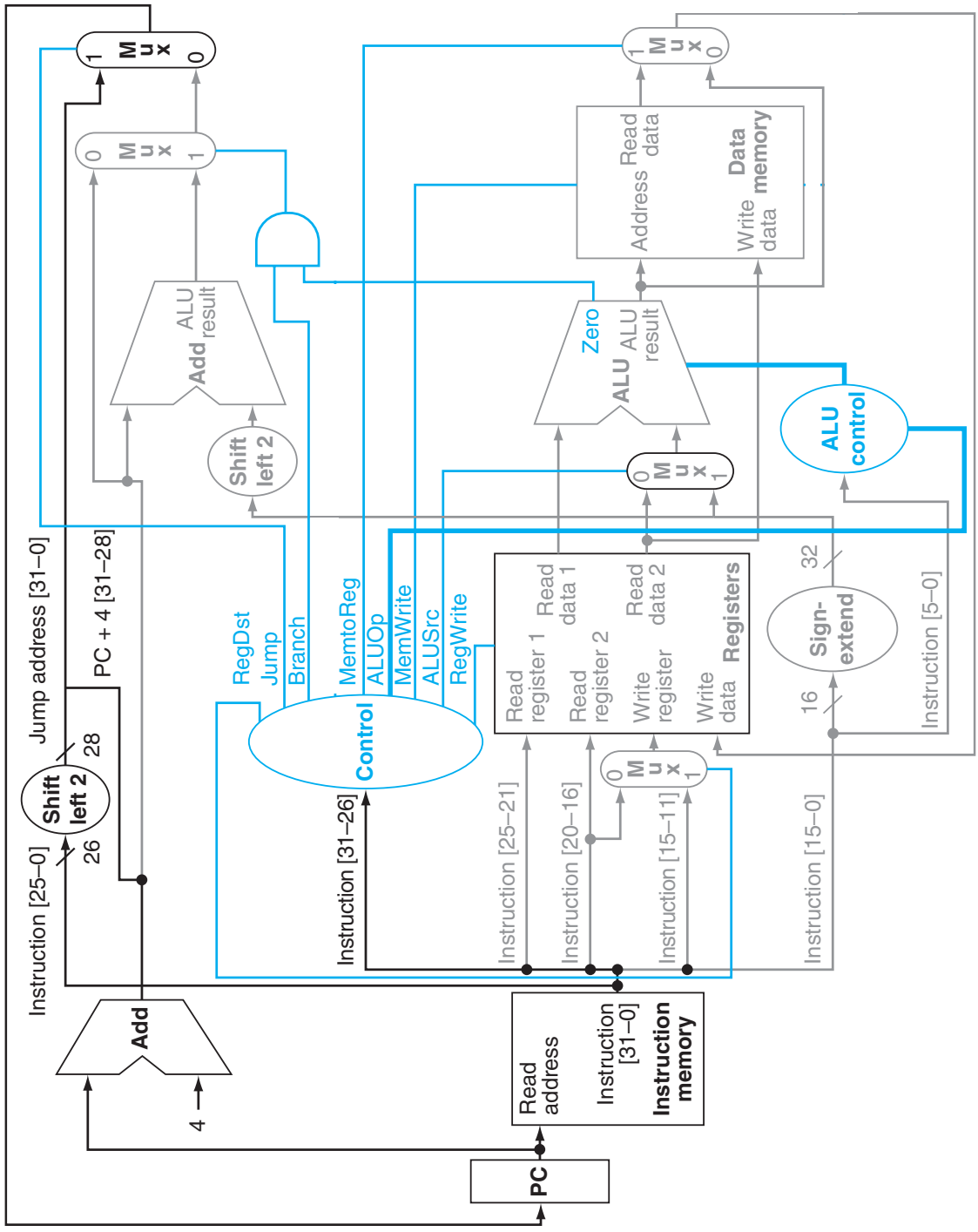
   Describe the problem this code will experience (i.e., what will not go as intended) when this code is run on the pipeline architecture with the modifications for `jr` and `jal` implemented in parts (a) and (b). A 1 to 2 sentence explanation should suffice per problem.

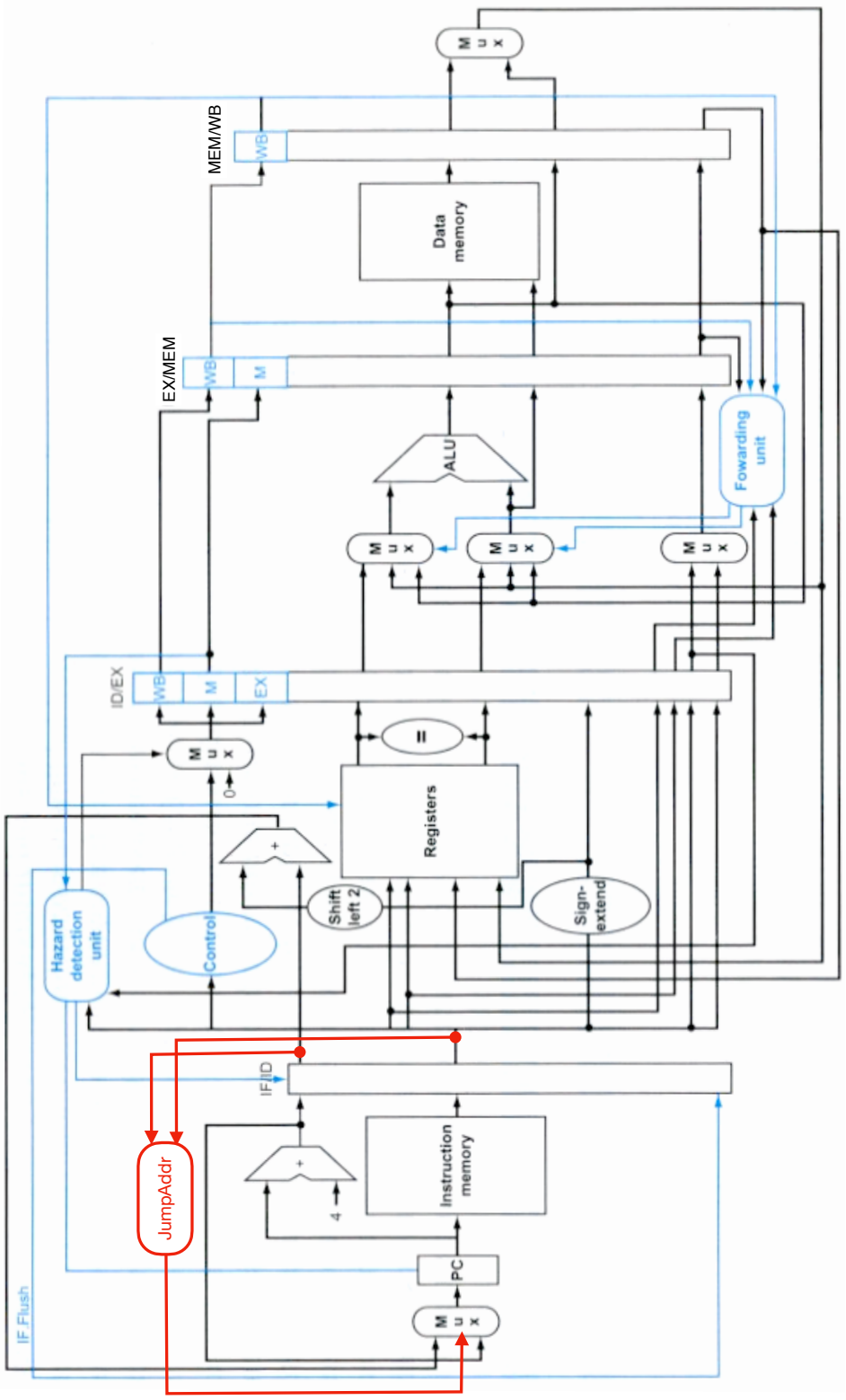
Answer: Line 8 will be 2 instructions behind line 3 in the pipeline, and will reach the ID stage when line 3 reaches the MEM stage, before writing back to the register file. Since it is in this clock cycle that line 8 pull the value from $ra in the register file and modify the $pc, it will pull the wrong value, performing the return that line 6 should be doing (and the stack owill never be popped as it needs to be).

This is the end of the last question. Have a great summer!

(This page intentionally left blank)

The Single Cycle Architecture





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