Final Solutions

CSEE W3827 - Fundamentals of Computer Systems — Dec 17, 2018
Fall 2018 — Prof. Rubenstein

This final contains 4 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 UNI, yours is sorted incorrectly because of a lack of clarity, you lose the 5 points.

Question 1 involves the game rock-paper-scissors. A brief description of the game is as follows: 2 players play and simultaneously select one of three configurations: (R)ock, (P)aper or (S)cissors. If both players select the same configuration (e.g., both select "rock"), then the game is a (D)raw. If they select different configurations, then there will always be one (W)inner and one (L)oser, with rock beating (smashing) scissors, scissors beating (cutting) paper, and paper beating (covering) rock. The game can be played for multiple rounds, providing hours of entertainment.

1. (30 pts) You are to build a sequential circuit that implements the strategy of one player (who we call the *strategic* player) over multiple rounds. In each round (which takes a clock cycle), the circuit does three things:

   - It chooses a configuration (R,P,S) that the strategic player will play for that round
   - It reads in a 2-bit input XY that specifies its opponent's configuration (R,P,S) for that round
   - It outputs a 2-bit result R_1R_0 that specifies the strategic player's outcome (W,L,D) against the opponent.

   The digital representations of the configuration and outcome are specified as:

Opponent Configuration (XY)		Outcome (R_1R_0)	
00	Rock	0X	Draw
01	Paper	10	Lose
10	Scissors	11	Win

   (Note for the outcome, the high-order bit R_1 indicates whether someone won the round, and if so, the low order bit then indicates whether the strategic player won).

   The configuration played in the $t+1$st round (clock cycle) by the strategic player depends on what both players played in the tth round, and on the outcome (W,L,D).

   - If the strategic player lost in round t, the configuration in round $t+1$ matches the configuration in round t.
   - If the strategic player wins or draws in round t, the configuration in round $t+1$ is what would have lost against the opponent in round t.

   The following table enumerates the 9 scenarios that the 2 players could have played in the round t, and shows the outcome and what the strategic player will play in round $t+1$:

Strategic player round t configuration:	R	R	R	P	P	P	S	S	S
Opponent round t config:	R	P	S	R	P	S	R	P	S
Strategic player outcome [(W)in, (L)ose or (D)raw]:	D	L	W	W	D	L	L	W	D
Strategic Player's round $t+1$ config:	S	R	P	S	R	P	S	R	P

   (NOTE: The above is not an example of the game being played over a series of rounds, but shows the outcome and next move for all possible configurations of the current round).

   (a) (20 points) Draw a state machine that implements the sequential circuit described above.

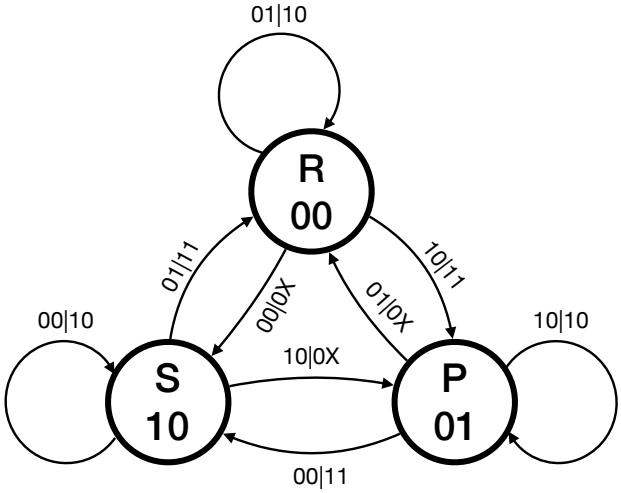
   (b) (10 pts) Give simplified expressions for the sequential circuit using using JK flip-flops. For full credit, provide simplified expressions for both output bits and inputs to each of the inputs to the JK flip-flops.

   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	$\overline{Q(t)}$

   NOTE: Part (a) is 2/3 of the points for this problem, and is a lot less grunt work. Be sure to look at other problems before sinking too much time into part (b).

Answer: The table that enumerates the 9 scenarios can be translated directly into a state machine. While the states can be selected arbitrarily, the most straightforward approach is to have a state represent the move the strategic player plays in the tth clock cycle. Simply mapping the letters to the earlier table yields the state machine.



If we refer to the high order bit of each state as A and the low-order bit as B, the excitation table is:

$A(t)$	$B(t)$	X	Y	R_1	R_0	$A(t+1)$	J_A	K_A	$B(t+1)$	J_B	K_B
0	0	0	0	0	X	1	1	X	0	0	X
0	0	0	1	1	0	0	0	X	0	0	X
0	0	1	0	1	1	0	0	X	1	1	X
0	0	1	1	X	X	X	X	X	X	X	X
0	1	0	0	1	1	1	1	X	0	X	1
0	1	0	1	0	X	0	0	X	0	X	1
0	1	1	0	1	0	0	0	X	1	X	0
0	1	1	1	X	X	X	X	X	X	X	X
1	0	0	0	1	0	1	X	0	0	0	X
1	0	0	1	1	1	0	X	1	0	0	X
1	0	1	0	0	X	0	X	1	1	1	X
1	0	1	1	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X

R_1:

	Y			
A	0	1	X	1
	1	0	X	1
	X	X	X	X
	1	1	X	0
			X	

B

$R_1 = A\bar{X} + \bar{A}X + B\bar{Y} + \bar{B}Y = (B \oplus Y) + (A \oplus X)$
(don't need to express with XOR for credit)

R_0:

	Y			
A	X	0	X	1
	1	X	X	0
	X	X	X	X
	0	1	X	X
			X	

B

$R_0 = B\bar{X} + \bar{B}X + AY = B \oplus X + AY$
(don't need to express with XOR for credit)

J_A:

A \ Y	0	1	3	2
0	1	0	X	0
1	1	0	X	0
3	X	X	X	X
2	X	X	X	X

B; X

$J_A = \bar{X}\bar{Y}$

K_A:

A \ Y	0	1	3	2
0	X	X	X	X
1	X	X	X	X
3	X	X	X	X
2	0	1	X	1

B; X

$K_A = X + Y$

J_B:

A \ Y	0	1	3	2
0	0	0	X	1
1	0	0	X	1
3	X	X	X	X
2	X	X	X	X

B; X

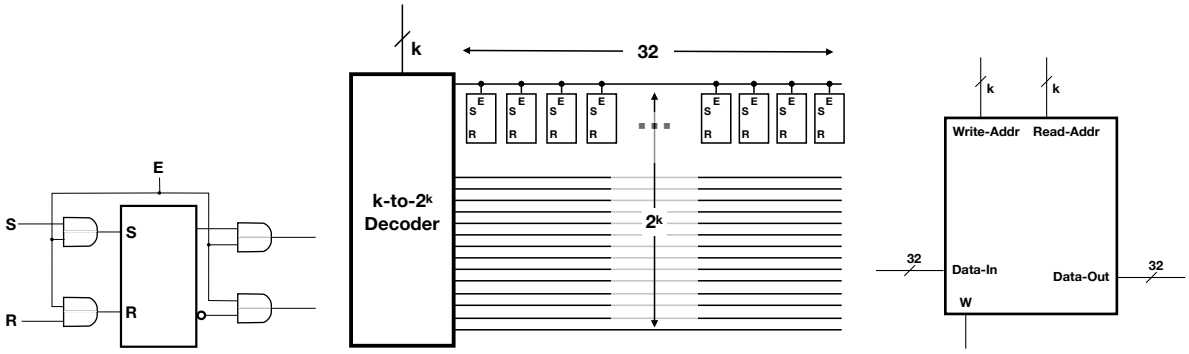
$J_B = X$

K_B:

A \ Y	0	1	3	2
0	X	X	X	X
1	1	1	X	0
3	X	X	X	X
2	X	X	X	X

B; X

$K_B = \bar{X}$

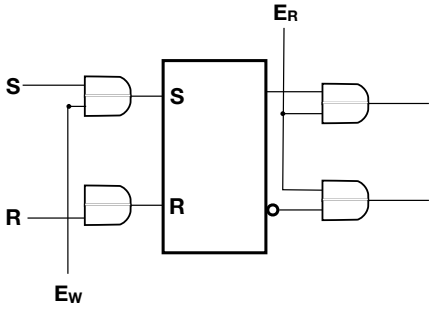


2. (30 pts) The traditional memory cell (an SR latch with enable) is depicted above on the left, and the manner in which these cells are enabled in a memory chip with 2^k 32-bit words is shown on the figure in the middle (the read and write logic is not shown).

   Suppose we wish to design a memory chip, as depicted in the right figure, that takes 2 addresses A_W and A_R such that the chip can simultaneously write data to address A_W while it reads from address A_R.

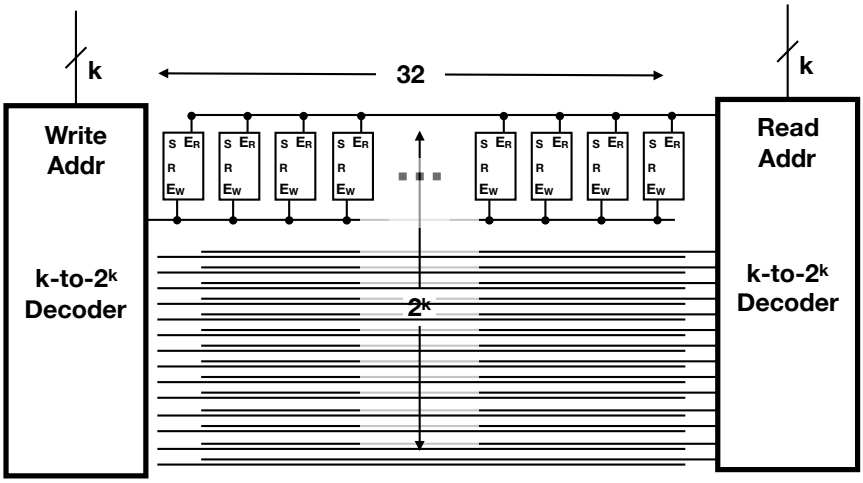
   (a) (15 pts) Show how to modify the traditional memory cell to implement this functionality (i.e., how would you redraw the left figure?)

   Answer: In the original, a single enable signal enabled both the write (left-hand side) and read (right-hand side) of the SR-latch. This simply needs to be split into 2 separate enable signals, as depicted below:



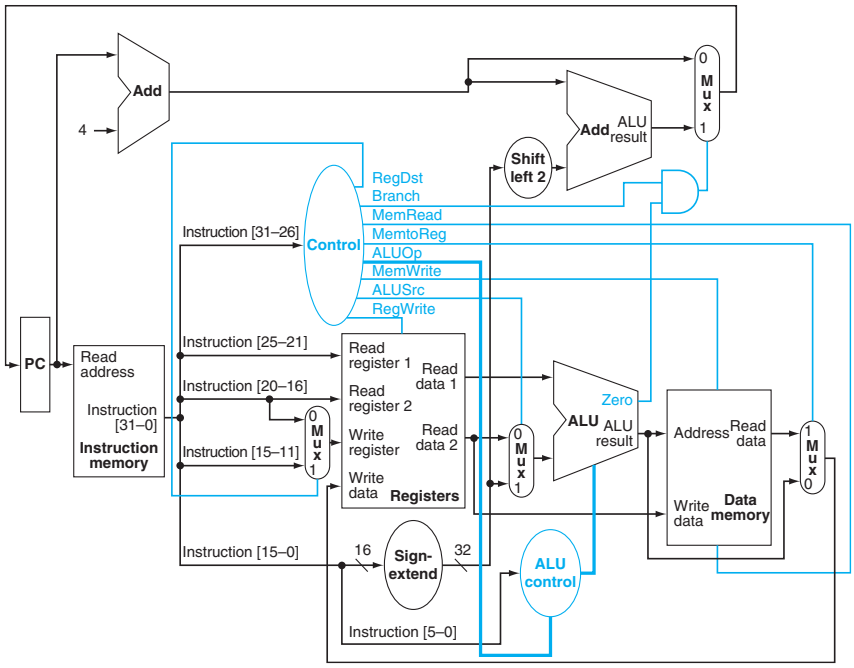
(b) (15 pts) Show how to modify the memory chip's enable circuitry such that you can separately select an address for writing and another for reading (i.e., how would you redraw the middle figure?)

Answer: The solution is to have two decoders, a "Write" decoder that attaches to all the SR-latches' Write-enable selectors, and a "Read" decoder that attaches to all the SR-latches' Read-enable selectors.



Note that for the write side, this will enable a only row for writing. The W input on the chip that enables writing can be implemented as before, enabling the values being fed into the S and R inputs of all the latches.

You do not need to show the Read or Write Logic. You only need to show the "enable" portion of the memory.



3. (30 pts) Consider the single-cycle architecture as pictured above.

   (a) (10 pts) Suppose we wish to add a new instruction, bme which stands for branch-if-memory-equal, branching only when the specified register's stored value equals the value stored in the specified memory location, and has the form:

   ```
   bme $r1, C($r2), LABEL
   ```

   It performs the *exact same* functionality as the following snippet using existing MIPS instructions:

   ```
   lw $t0, C($r2)
   beq $t0, $r1, LABEL
   ```

   There are at least three main challenges to implementing this instruction into the existing single-cycle architecture. Describe two of them.

   NOTE: for full credit, the two you describe must be correct. If you list three and n of them are right, you get at most $n/3$ of the credit.

   Answer: Three problems are:

   - bme requires 2 constants (one for the branch offset, one for the constant to be added to the base address). If the instruction is expected to have the exact same functionality, then these must both be 16-bit constants, leaving no room for any other information, such as an opcode to indicate what instruction is being represented.
   - The ALU is needed for two purposes: to add the constant to the base address, and to compare the value in register $r1 to the value in the specified memory location (i.e., to perform a subtract).
   - The value pulled from memory must be compared to the value in $r1, but there is no path from memory back into the ALU to perform the comparison.

(b) (20 pts) Suppose we wish to add a different new instruction, sbz, which stands for subtract and branch-if-zero, and has the form:

```
sbz $r1, $r2, LABEL
```

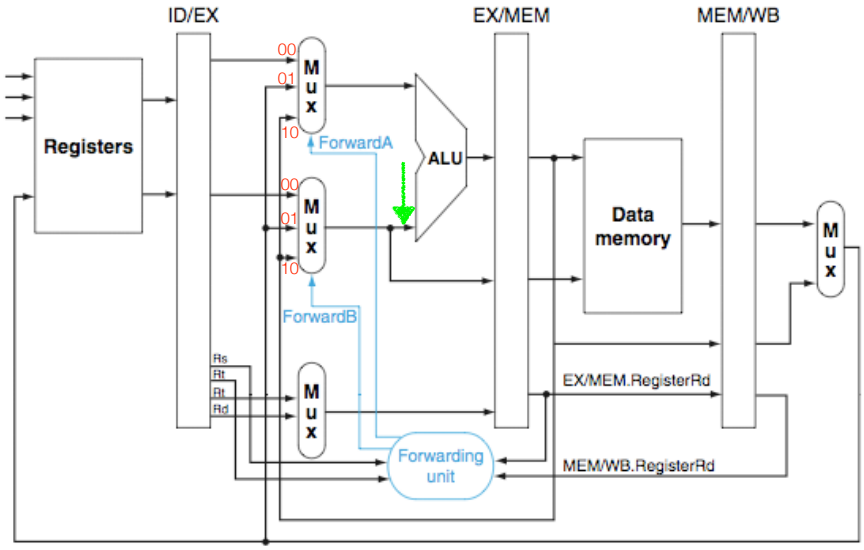
and performs the functionality as the following existing snippet:

```
sub $r2, $r1, $r2
beq $r2, $zero, LABEL
```

Starting with the an existing MIPS instruction (other than sbz), explain how to alter that instruction to arrive at sbz. Specifically, what signals would you change coming out of Control? And what hardware would you need to add, if any?

Answer: By starting with the beq instruction, the ALU is already performing a substraction to compare the values initially stored in $r1 and $r2. Normally, beq does not write to the register file, so the only changes to enforce are:

- Set RegWrite to 1 so that the register file is written to
- Set MemToReg to 0 so that the ALU's result is forwarded to the register file
- BONUS (not required for full credit, but can earn points if credit was lost elsewhere on part b): Set RegDest to 0 so that the register that gets written to is the same as $r2. This should be obvious from the figure since the bits of the "Read Register 2" input in the register file are the same as those in sent to "Write register".



4. (30 pts) The above figure shows a portion of the pipeline architecture with data-forwarding implemented. The green arrow points to a region where a MUX that allows one to select between the second register value and a constant exists (it was the same location as it would be in the single-cycle architecture).

   Consider the following snippet of code:

```
add $s0, $zero, $zero
addi $s1, $zero, 40000
lw $s0, 0($s1)
sw $s0, 1000($s1)
```

   and suppose that the value stored in memory at address 40000 equals 100.

   Suppose the designer figures that because data forwarding is implemented, he/she doesn't need to stall any of the above instructions when running them through the pipeline.

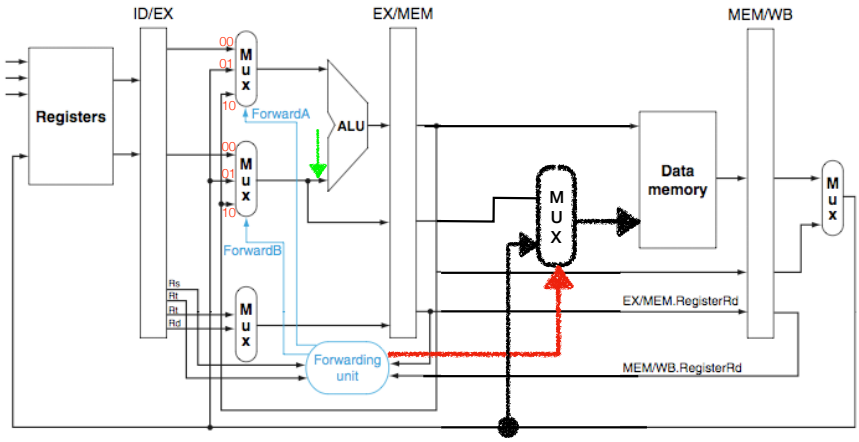
   (a) (10 pts) What registers and/or memory locations will have incorrect values stored as a result of not implementing the stall, and what (incorrect) value could occur? Explain in 1 sentence how that particular value occurred instead of the correct value.

   Answer: Only memory location 41000 will have an incorrect value, since as designed, lw will only complete reading from memory after it has finished the MEM stage of the pipeline, and forwarding would occur when lw enters the WB stage. If sw is in the EXE stage when lw is in the MEM stage, then it will be in the MEM stage when lw is in the WB stage, and has already bypassed the point at which forwarding can happen.

Without stalling, the value that sw would store into memory location would be the value originally in $s0 prior to the lw, which would be 0 (if no forwarding was performed when sw enters the EXE stage). It's possible that forwarding is performed, in which case the value might be the result of the ALU forwarded from the previous instruction, which would also be 0. Or it could be the result of the ALU from 2 instructions ago, which would be 40,000. So either 0 or 40,000 would be a reasonable value.

(b) (10 pts) It is possible to add additional forwarding logic such that a stall would not be necessary. Indicate where and how. Either draw a picture of the circuitry where a change is being made, or simply provide a short, clear description of the change.

Answer: Forwarding must be implemented from the beginning of the WB stage to the beginning of the memory stage. In the above snippet, the value to be written into $s0 by lw is forwarded to the sw instruction as it enters the memory stage. The high-level view of the change is depicted below, with the novel components drawn with fuzzy lines:



(c) (10 pts) Does the above change in part (b) prevent all stalls due to data dependencies? If so, give a short, 1 sentence explanation as to why. If not, give an example MIPS code snippet that would still require a stall due to a data dependence.

Answer: No, there can still be stalls. For instance:

```
lw $s0, 20($s1)
add $s2, $s0, $s3
```

will still stall, since the add instruction would need the result from lw prior to entering the ALU. The additional fix proposed above would not solve this issue.

If a solution were proposed where the result from Memory were forwarded back around to in front of the ALU, this would fix stalling, but would significantly increase the clock cycle time to accomodate a value exiting memory being available to then be run through the ALU. It would, however, technically prevent stalling.

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