Wawrzynek & Weaver CS 61C Sp 2018 Great Ideas in Computer Architecture MT 2

PRINT your name: ______________________, ______________________
(last) (first)

I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that any academic misconduct will be reported to the Center for Student Conduct, and may result in partial or complete loss of credit.

SIGN your name: ____

PRINT your class account login: `cs61c-`______ and SID: ______________________

Your TA's name: ____

Your section time: ____

Exam # for person sitting to your left: ____________ Exam # for person sitting to your right: ____________

You may consult two sheets of paper (double-sided) of notes. You may not consult other notes, textbooks, etc. Calculators, computers, and other electronic devices are not permitted.

You have 110 minutes. There are 5 questions, of varying credit (90 points total). The questions are of varying difficulty, so avoid spending too long on any one question. Parts of the exam will be graded automatically by scanning the **bubbles you fill in**, so please do your best to fill them in somewhat completely. Don't worry—if something goes wrong with the scanning, you'll have a chance to correct it during the regrade period.

If you have a question, raise your hand, and when an instructor motions to you, come to them to ask the question.

Do not turn this page until your instructor tells you to do so.

Question:	1	2	3	4	5	Total
Points:	17	17	19	20	17	90

Exam Clarifications

Sit every other seat

Q2 (b) The second sentence should read: “What’s the minimum number of NAND gates you need to implement A+ B”.
Q2 (b) NAND gates that only have 2 inputs
Q2 (e) You may or may not need all 4 states.
Q3 (e) 6. and Q4d MemRW: 1 (‘Enable Write’). 0 (‘Disable Write, Read Only’)
Q4: On the diagram ALU Execute is annotated with (X), in the table, it is E. Use E for your answer.
Q4 (d): If the value of any signal doesn’t matter, you must write X (“don’t care”).

Figure 1: This Space Deliberately Left Blank

Problem 1 *RISCy Business* (17 points)

Bubble in one answer per question:

(a) Select the stage that computes the offset for a `beq` instruction.

○ Compiler ○ Linker

[✓ ● Assembler] ○ Loader

Solution: Branch instruction offsets are PC-relative, after pseudo-instructions are replaced by real ones, the Assembler can compute by how many instructions to branch.

(b) Select the stage that computes the offset for a `jal` instruction to a function from a different object file.

○ Compiler [✓ ● Linker]

○ Assembler ○ Loader

Solution: Linker (or link editor) takes all the independently assembled machine language programs and "stitches" them together. When it sees a `jal` instruction, it uses the relocation information and symbol table in each object module to compute the jump offset.

(c) Select the stage that creates a symbol table and a relocation table.

○ Compiler ○ Linker

[✓ ● Assembler] ○ Loader

Solution: Assembler keeps track of labels used in branches and data transfer instructions in a symbol table. For instructions and data words that depend on other modules, Assembler would add them into relocation table such that the Linker can finish the relocation process.

(d) Select the stage that emits `add s0, s1, s2`.

[✓ ● Compiler] ○ Linker

○ Assembler ○ Loader

Solution: Compiler transforms high-level programs (such as C programs) into assembly code.

(e) Select the stage that combines several object files into a single executable.

○ Compiler [✓ Linker]

○ Assembler ○ Loader

Solution: Linker (or link editor) takes all the independently assembled machine language programs and "stitches" them together.

(f) Complete the following RISC-V procedure `jal_address_fixing` that handles address relocation for all `jal` instructions. It first calls `find_next_jal` to find a `jal` instruction that does not yet have its offset filled in (the immediate bits are all zeroes), calculates the jump offset, and fills the immediate field of the `jal` instruction.

- Fill in **one** instruction for each of the 5 blanks.
- You can assume `jal_address_fixing` has the ability to modify text segment instruction memory.
- The function `find_next_jal` returns two values: the first is the address of a `jal` instruction stored in `a0`; the second is the address of the target instruction this `jal` instruction is jumping to stored in `a1`. If there are no more `jal` instructions to fill in offsets for, it returns 0 and 0.

```
jal_address_fixing:
    jal ra, find_next_jal
    beq a0, x0, DONE
    sub a1, a1, a0         # set a1 as the jump offset
IMM_20:                      # sets imm[20]
    srai a5, a1, 20         # now a5 has imm[20]
    slli a5, a5, 31      # a5 has imm[20]
IMM_19_12:    # sets imm[19:12]
    li a3, 0xFF000
    and a3, a1, a3
    or a5, a5, a3      # now a5 has imm[20] and imm[19:12]
IMM_10_1:    # sets imm[10:1]
    li a3, 0x7FE
    and a3, a1, a3
    slli a3, a3, 20
    or a5, a5, a3      # now a5 has imm[20], imm[10:1], and imm[19:12]
IMM_11:    # sets imm[11]
```

```
    li a3, 0x800
    and a3, a1, a3
    slli a3, a3, 9
    or a5, a5, a3      # now a5 has imm[20], imm[10:1], imm[11], and imm[19:12])
UPDATE:                # inserts immediate into jal instruction
    lw t0, 0(a0)       # load the current jal instruction (from mem[a0])
    or t0, t0, a5      # update the instruction (add also works)
    sw t0, 0(a0)       # save the updated instruction (to mem[a0])
    j jal_address_fixing          # jump back to fix the next one
DONE:
    ...
```

Solution: The above procedure in C:

```
void jal_address_fixing(int address, int target) {
     int offset = target - address;
     int imm_10_1 = (offset & 0x7FE) >> 1;
     int imm_11 = (offset & 0x800) >> 11;
     int imm_19_12 = (offset & 0xFF000) >> 12;
     int imm_20 = (offset & 0x100000) >> 20;
     offset = (imm_20 << 31) | (imm_10_1 << 21) |
                   (imm_11 << 20) | (imm_19_12 << 12);
     int* addr = (int*) address;
     *addr &= offset;
}
```

(g) The above code works for a `jal` target address that is 2^{16} bytes smaller than the `jal` instruction address.

[✓ True] False

Solution: This code correctly updates the offset for `jal` instruction if the address is 2^{16} bytes smaller.

(h) The above code works for a `jal` target address that is 2^{24} bytes larger than the `jal` instruction address.

True [✓ False]

Solution: To reach an address that is 2^{24} bytes larger, we need two instructions: `auipc` and `jalr`. This code does not work for far jump targets.

Problem 2 ***Tell Us The Truth*** **(17 points)**

X	Y	Z	Out
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(a) Select all of the following expressions that are equivalent to the truth table above.

● $(X + \bar{Y} + Z)(\bar{X} + \bar{Y} + Z)$

○ $\bar{Z} + Y$

● $\bar{Y} + Z$

○ $X\bar{Y}Z + \bar{X}\bar{Y}Z$

● $X\bar{Y} + \bar{X}\bar{Y} + Z\bar{X} + ZX$

● $\bar{Y} + \bar{Y}Z + Z$

Solution: Looking at the truth table, we see that there are more 0's in our output than 1's. This means that Product of Sums will be the easiest way to proceed.

Taking the Product of Sums will give us $(X + \bar{Y} + Z)(\bar{X} + \bar{Y} + Z)$. This is the first bubble.

We will then simplify this further to be: $(X\bar{X} + X\bar{Y} + XZ + \bar{Y}\bar{X} + \bar{Y}\bar{Y} + \bar{Y}Z + Z\bar{X} + \bar{Z}\bar{Y} + ZZ)$. We can use the boolean identities to simplify this to be $(\bar{Y}(X + \bar{X}) + Z(X + \bar{X}) + \bar{Y} + Z)$.

Since $(\bar{Y}(X + \bar{X})) = \bar{Y}$ and $(Z(X + \bar{X}) + \bar{Y} + Z) = Z)$, we can simplify this to be $(\bar{Y} + Z)$, which is option 3. This is the most simplified expression for the truth table.

We can also expand $(\bar{Y}(X + \bar{X})) + Z(X + \bar{X}))$ to be $(X\bar{Y} + \bar{X}\bar{Y} + Z\bar{X} + ZX)$, which is option 5.

For option 6, we see we can simplify by adding another $(\bar{Y}Z)$:

$(\bar{Y} + \bar{Y}Z + Z + \bar{Y}Z) = (\bar{Y}(Z + 1) + Z(\bar{Y} + 1) = \bar{Y} + Z$, which matches our simplified expression.

Option 2 is a simplified expression which does not match $\bar{Y} + Z$, so it must be incorrect. Option 4 is also incorrect, as it takes the Sum of Products approach on the 0 entries in our truth table.

This leaves options 1, 3, 5, and 6 as the correct answers.

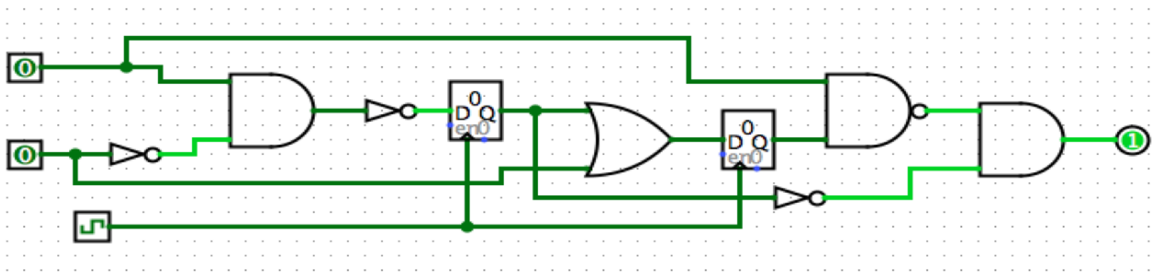
(b) Suppose you wanted to implement $\bar{A} + B$, but the only available gates are NAND gates. What is the minimum number of NAND gates you need to implement the above truth table correctly?

Solution: 2

The boolean equation for A NAND B is $\bar{A} + \bar{B}$. This almost matches our expression $\bar{A} + B$, so we want to do A NAND $\bar{B}$.

If we try doing B NAND B, we see that this gives us $\bar{B}$. So, we can recreate the expression $\bar{A} + B$ by doing A NAND (B NAND B), which uses 2 NAND gates.

Now, consider the following circuit:



You are given the following timing parameters: Register Clk-To-Q: 2ps, Register Setup: 2ps, NOT Gate: 1ps, AND Gate: 4ps, OR Gate: 3ps, NAND Gate: 4ps. **Assume the 2 inputs comes from registers and the output is connected to a register as well.**

(c) What is the minimum clock period at which this circuit can be run?

Solution: 12 ps

We can break this circuit into 3 paths between registers. The minimum clock period will be the maximum path delay, which is calculated by Clk-To-Q + Combinational Logic Delay + Setup Time.

Path 1: Clk-To-Q + NOT + AND + NOT + Setup = 2ps + 1ps + 4ps + 1ps + 2ps = 10ps

Path 2: Clk-To-Q + OR + Setup = 2ps + 3ps + 2ps = 7ps

Path 3: Clk-to-Q + NAND + AND + Setup = 2ps + 4ps + 4ps + 2ps = 12ps

So, the critical path is path 3, and the max delay is 12ps.

(d) What is the maximum hold time that would allow for this circuit to run correctly?

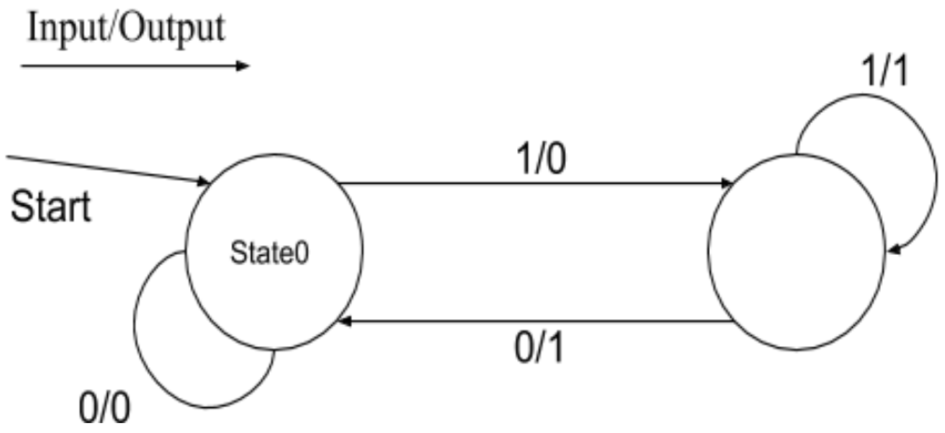
Solution: 5 ps

We find the max hold time by now looking at the shortest path, which is path 2. We need to ensure that Hold Time ¿= Clk-To-Q + Combinational Logic delay.

Path 2: Clk-to-Q + OR = 2ps + 3ps = 5ps. So, our hold time can be no more than 5ps, or there will be a hold time violation in path 2.

(e) Draw the State Transition Diagram for a Finite State Machine that, given a sequence of binary digits, outputs 1 if the second most recently seen digit is 1 and outputs 0 otherwise. For example, an input of 01100111 has the output 00110011. Label all states and transition inputs/outputs you draw. You may or may not need all 6 states.

Solution:



For this question, we need states to track the value of the most recent input bit. If the most recent input bit was a 0, we know that the output for the next transition will be a 0 (as an input of either a 0 or a 1 will cause the 2nd most recent bit to be a 0). If the most recent bit is a 1, we know that after another bit comes in, the 2nd most recent bit will have been a 1, so we should output a 1.

We see that any transition leaving state 0 outputs a 0, and any transition leaving state 1 outputs a 1. While there are possible implementations that could use more states, we only need 2 to represent this problem as the state from which we are leaving will represent the value of the 2nd most recent bit.

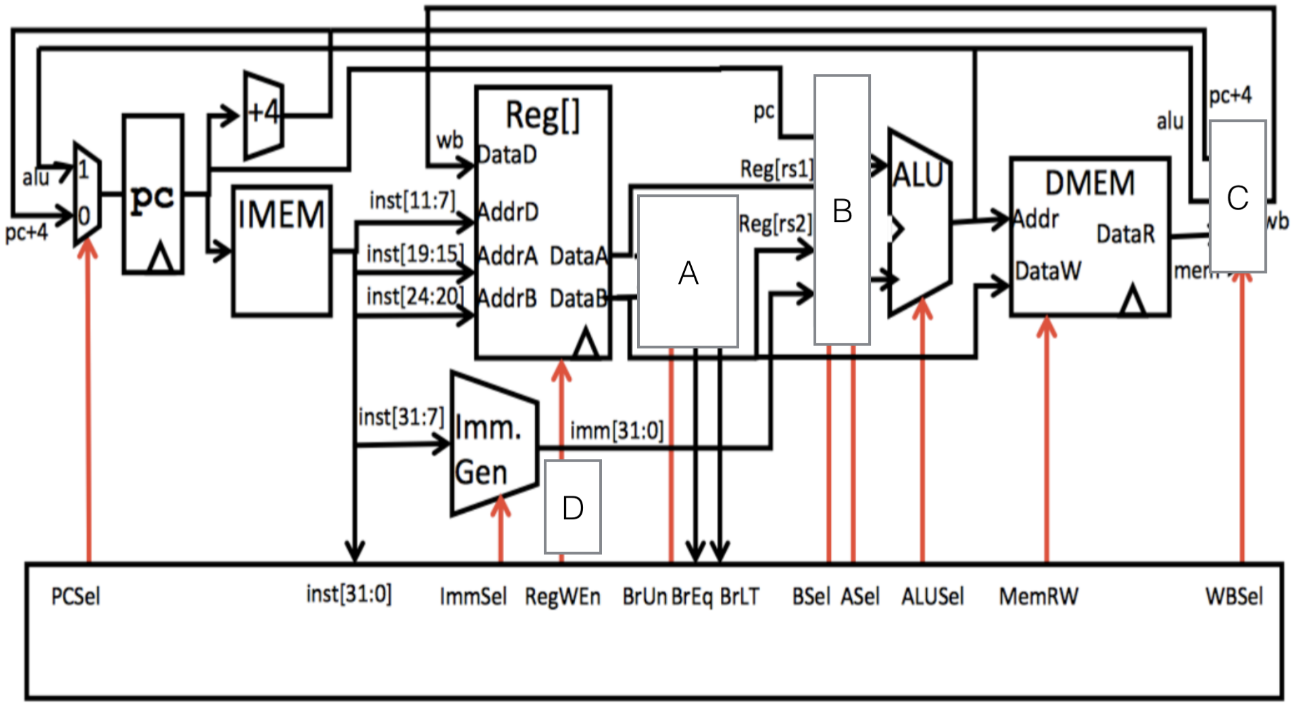
Problem 3 ***Set ... If Zero*** **(19 points)**

We wish to introduce a new instruction into our single-cycle datapath. The instruction **SIZ** (set if zero) works as follows:

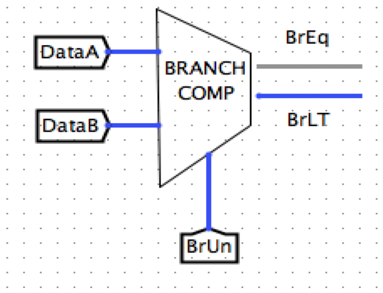
```
if (R[rs2] == 0):
    R[rd] = R[rs1]
```

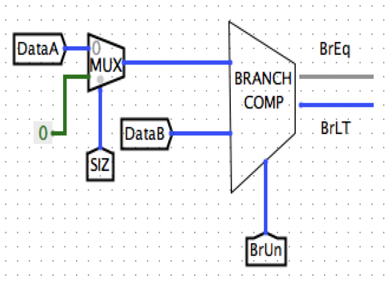
Given the single cycle datapath below, select the correct modifications in parts (a) - (d) such that the datapath executes correctly for this new instruction (and all core instructions!). You can make the following assumptions:

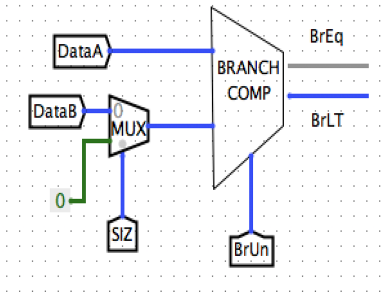
- the `SIZ` signal is 1 if and only if the instruction is `SIZ`
- `ALUSel` is **ADD** when when we a have `SIZ` instruction.
- **the immediate generator outputs ZERO** when we have a `SIZ` instruction.

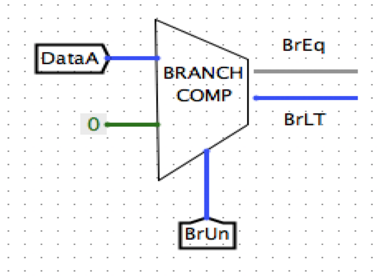


(a) Consider the following modifications to the branch comparator inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set?

○ 

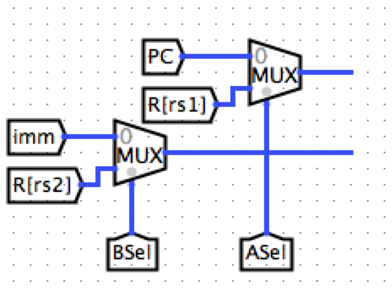
● 

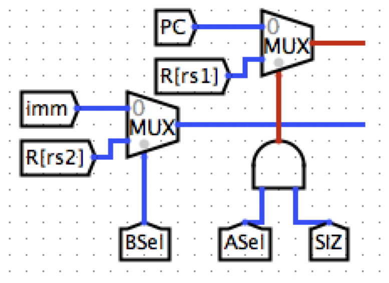
○ 

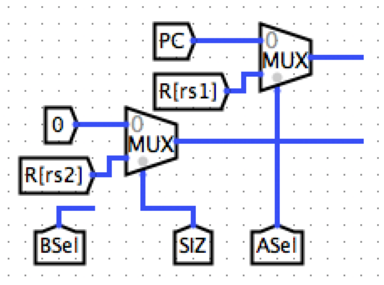
○ 

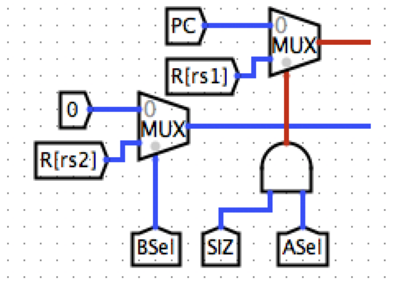
Solution: Note that for this question we have two requirements: the modification we pick must support our new instruction AND it must make it so all other instructions (those in our core instruction set) continue to execute correctly as if no changes were made. If we look at the logic describing the instruction behaviour in the first part of the question, we notice we must compare the value in register rs2 to zero. Unlike normal branch instructions, we are not comparing to another register value; we are comparing to a constant. This eliminates choice A. Noting the conditions of our modification (that it must be able to support core instructions, too) we can eliminate D because it removes the ability to compare DataA and DataB which we need for regular branch instructions. We are left with options B and C. Again, if we revisit the instruction logic, we see the item we're comparing to zero is the value in rs2; this is equal to DataB. We therefore pick option B which adds a MUX on DataA to allow us to select 0 as our second operand in the case of a SIZ instruction.

(b) Consider the following modifications to the ALU inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set? Select the configuration that requires **minimum** modifications to the original datapath. Notice in the bottom left choice `BSel` is unused.

● 

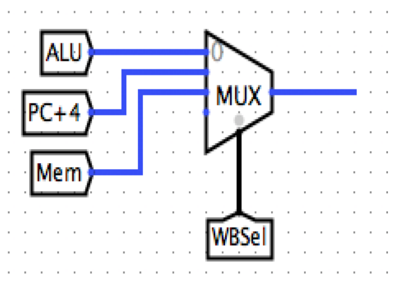
○ 

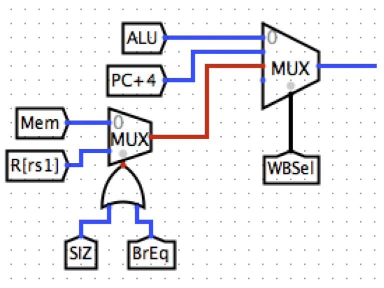
○ 

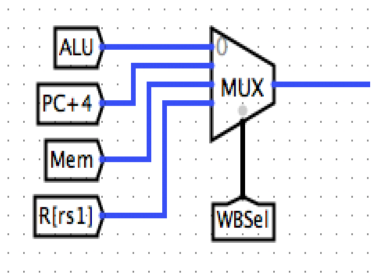
○ 

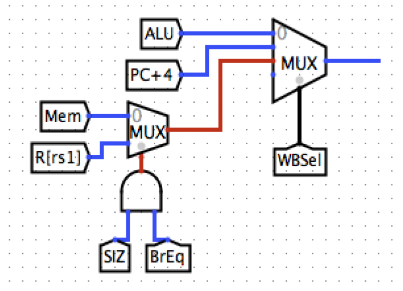
Solution: This modification must also support our new instruction while allowing other instructions to continue executing as normal. This section concerns inputs to our ALU–the output of which we will write. Looking again at the instruction logic, we can see the write we need to make is R[rd] = R[rs1]; the value in register rs1 should be written to the rd register. Therefore, the output of our ALU should be the value in register rs1. If we look at our datapath, this is already possible by manipulating existing controls (ASel, BSel), and so we do not need to make any modifications; A is the correct answer.

(c) Consider the following modifications to the WB mux inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set? Select the configuration that requires **minimum** modifications to the original datapath.

● 

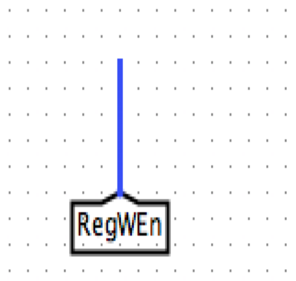
○ 

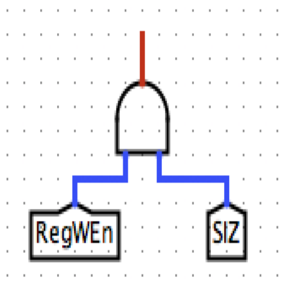
○ 

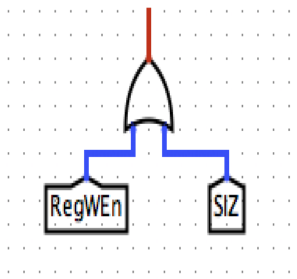
○ 

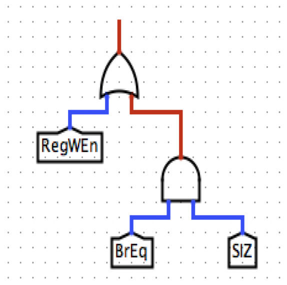
Solution: Similar to the previous question, we do not need to make a modification to the datapath and should therefore select option A. We know our ALU is emitting the value stored in register rs2. Because the standard datapath already allows us to write back the output of our ALU (and because write-back by default writes to our specified destination register rd), the value in rs2 can be written back to rd by setting our existing control bit WBSel to ALU. Again, no modifications are required.

(d) Consider the following modifications to the RegWEn inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set?

○ 

○ 

○ 

● 

Solution: To answer this question we should look at the instruction logic to find out under what conditions the write occurs. Note that, in the SIZ instruction, we should only set R[rd] = R[rs1] in the case that R[rs2] == 0 is true. In order to check that condition, we need to make sure of two things: (a) the instruction we're writing back for should be a SIZ instruction and (b) the result of the branch equality comparison should be true (BrEq). Because we want both of these to be true before we write, we use an AND gate. To preserve existing functionality, we also want to keep our RegWEn control bit around. Because the additional logic we added for the SIZ instruction will be false for all other instructions (add, load, etc.) we use an OR gate to support write-back functionality for our core instructions.

(e) Given your selections above, decide the rest of the control signals for this instruction based on the diagram given at the beginning of the problem. Select `X` when a signal's value doesn't matter. You can assume:

- the `SIZ` signal is 1 if and only if the instruction is `SIZ`
- `ALUSel` is **ADD** when when we a have `SIZ` instruction.
- the immediate generator outputs **ZERO** when we have a `SIZ` instruction.

1. `PCSel`:

   ○ 1 [✓ 0] ○ X

   Solution: Though this instruction is similar to other branch instructions in that it uses the branch comparator to check equality, it does not alter our control flow as a result, therefore we should select PC as PC+4 like we do normally.

2. `RegWEn`:

   ○ 1 (Enable) [✓ 0 (Disable)] ○ X

   Solution: Our write to rd should only happen in the case that our if case is true. We added logic to support this in the previous question and, in order for that check to happen, we have to set RegWEn to false. If it were true, we would write to rd on every SIZ instruction, not just those where R[rs2] == 0.

3. `BrUn`:

   ○ 1 (Signed) ○ 0 (Unsigned) [✓ X]

   Solution: We are doing a comparison to zero. Whether the branch comparison is signed or unsigned has no effect on the outcome.

4. `BSel`:

   ○ 1 ○ 0 [✓ X]

   Solution: We want our ALU to produce rs1 as its output. We do not care what the value of our second operand is (because regardless, it isn't the output we want) and therefore it doesn't matter if we pass in the immediate or DataB.

5. `ASel`:

[✓ 1] ○ 0 ○ X

Solution: Though we don't care about BSel, we do care about ASel because this is where we have the option of passing in rs1 to the ALU. If we want our write to contain the correct information, we must select DataA as our input here.

6. `MemRW`:

○ 1 (Enable) [✓ 0 (Disable)] ○ X

Solution: This instruction makes no modifications to memory and therefore shouldn't write to memory. The reason we can't leave this control bit as an X (don't care) is because modifying memory is a state change; if we "don't care" and modify memory on some SIZ instructions and not on others, we could end up overwriting, deleting, or forcing garbage data into our memory and messing up execution of later instructions.

7. `WBSel`

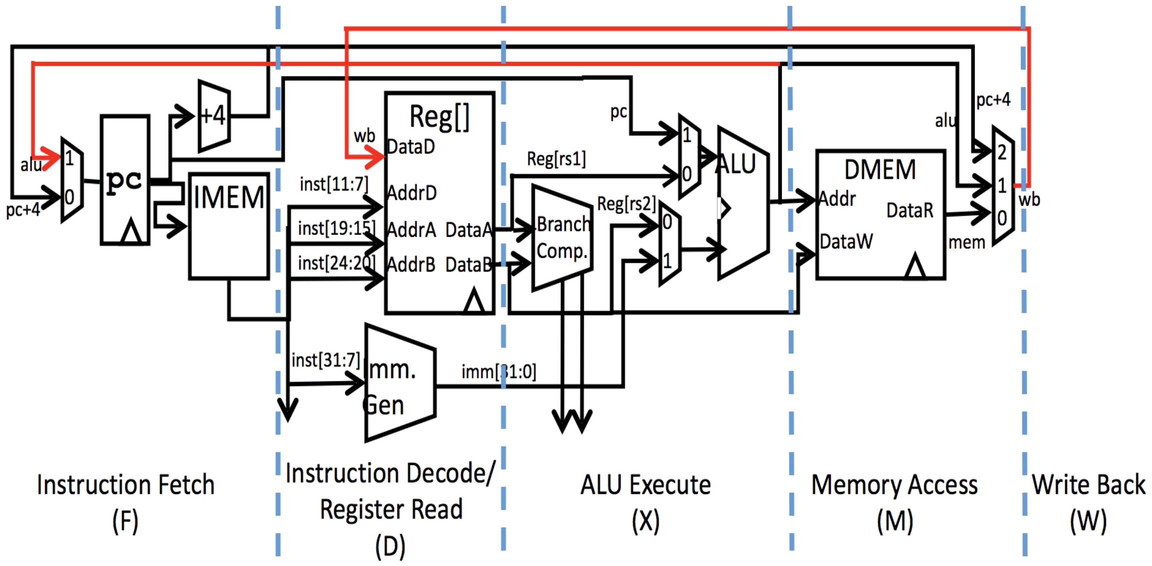
[✓ ALUOut] ○ MemOut

○ PC + 4 ○ Other: Please specify:______________

Solution: See solution to part C above

Problem 4 ***More or Less*** **(20 points)**

Consider a typical 5-stage (**F**etch, **D**ecode, **EX**ecute, **M**emory, **W**rite**B**ack) pipeline. Assume pipeline registers exist where the dotted lines are.



For this question, consider the following parameters:

- Forwarding is not implemented
- The branch predictor always predicts the branch is not taken. Flush the pipeline if prediction is wrong.
- We cannot read and write from the same registers or memory address in the same clock cycle.
- No other optimizations are implemented in this datapath.

(a) Fill in the corresponding pipeline stages for the code sequence below for the 5-stage pipeline. The first instruction is done for you. If you need to stall a cycle, write "*" in that cycle.

```
begin:
        ori s1, x0, 0xF
        andi s2, x0, 0
        beq s1, s2, exit
        lw s1, 0xc(s0)
        xor s1, s1, s2
exit:
        lw s1, 0xc(s0)
```

Solution:

Instructions	Cycles															
	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16
ori s1 x0 0xf	F	D	E	M	W											
andi s2 x0 0		F	D	E	M	W										
beq s1 s2 exit			F	D	*	*	*	E	M	W						
lw s1 0xc(s0)				F	*	*	*	D	E	M	W					
xor s1 s1 s2								F	D	*	*	*	E	M	W	
lw s1 0xc(s0)									F	*	*	*	D	E	M	W

The first thing to notice for this question is that the datapath does not implement bypassing, and a register cannot be simultaneously read and written in the same cycle. Recall that instructions read their registers in stage DE, and write registers in WB. These restrictions mean that if instruction B needs a register that instruction A writes, then B cannot start its DE stage until the cycle after A's WB stage (this is a "data hazard"). The other type of hazards to worry about are "structural hazards", this means that no two instructions can be in the same stage at the same time. Now lets go through the answer instruction-by-instruction:

- **andi s2 x0 0:** This doesn't have any data depencies, so we just need to worry about structural hazards. It can start as soon as the F stage is available (c2).
- **beq s1 s2 exit:** This instruction reads register s2 which was written by the previous instruction. Therefore we must wait until the andi has finished its WB stage before running beq's DE stage (c7).
- **lw s1 0xc(s0):** At this point, the result of the branch doesn't matter because it is always predicted to be taken. Also, the branch doesn't write any registers, so we don't have any data dependencies and can start as soon as the stages are available. In this case, the fetch can start on c4, but the decode has to wait until beq is done with it (c8).

- **xor s1 s1 s2:** We now must consider whether or not the branch was predicted correctly. Fortunately it was, so we don't need to take any action. Next we must look for data hazards; s1 is read by xor, but written by lw, so we must wait for lw to finish WB before starting xor's DE (c12).
- **lw s1 0xc(s0):** There are no data hazards (notice that s1 is not read during the lw, only written, so there isn't a hazard). We need only wait for the stages to become available (structural hazards).

(b) Assume the maximum delays through each stage are: `F: 200ns, D: 150ns, EX: 100ns, MEM: 300ns, WB: 250ns`.

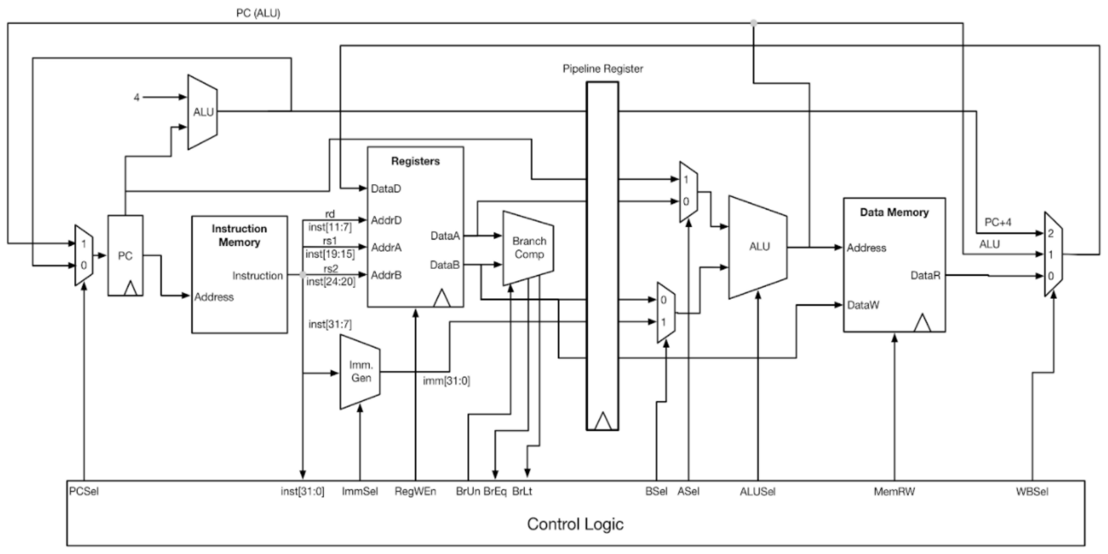
Assume the delays for the pipeline registers are factored into the pipeline stage delays. What is the latency and best case throughput of this 5-stage pipelined CPU? You may leave your answers as fractions. Don't forget the units!

Solution: Latency: 1500 ns Throughput: $\frac{1}{300ns}$

Latency: Latency is the time it takes from when the instruction starts the first stage, to when it exits the last stage. While you may be tempted to sum up the stage latencies, this would not be correct. This is because the CPU must run at a constant clock frequency, and the clock frequency is determined by the slowest stage (MEM in this case). Therefore, the cycle time is 300ns, and it takes 5 cycles to finish all stages, so the latency is $5 * 300 = 1500$ns.

Throughput: Throughtput is rate at which instructions leave the pipeline when there are lots of instructions running (steady-state). Technically, throughput could be affected by data hazards and such, so we ask for the "best-case" throughput (e.g. for an endless series of non-dependent adds). A pipeline completes one instruction every cycle under ideal circumstances, so the throughput of this pipeline is $\frac{1}{cycleTime} = \frac{1}{300ns}$.

You decide to combine certain stages to make a two-stage pipeline by removing certain registers. The two-stage pipelined datapath looks like the following:



Again, consider the following parameters:

- Forwarding is not implemented
- The branch predictor always predicts the branch is not taken. Flush the pipeline if prediction is wrong.
- We cannot read and write from the same registers or memory address in the same clock cycle.
- No other optimization is implemented on this datapath.

(c) Fill in the corresponding pipeline stages for the same code sequence for the 2-stage pipelined CPU. The code sequence is reproduced below. Use "A" to denote stage 1, "B" for stage 2. The first instruction is done for you. If you need to stall a cycle, write "*" in that cycle.

```
begin:
         ori s1, x0, 0xF
         andi s2, x0, 0
         beq s1, s2, exit
         lw s1, 0xc(s0)
         xor s1, s1, s2
exit:
         lw s1, 0xc(s0)
```

Solution:

Instructions	Cycles													
	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
ori s1 x0 0xf	A	B												
andi s2 x0 0		A	B											
beq s1 s2 exit			A	*	B									
lw s1 0xc(s0)					A	B								
xor s1 s1 s2						A	*	B						
lw s1 0xc(s0)								A	B					

This question is extremely similar to 4.a so we will not cover it in quite as much detail. To do this problem, you must first determine what impact data hazards will have. In this case, the registers are written in stage B and read in stage A, so we just need to make sure that instructions that have data dependencies wait until the previous B stage finishes before starting A. The two data dependencies are on the beq and the xor, so those have stalls to ensure their A stage doesn't run until the previous B has finished. Also note that unlike 4.a, we don't introduce long stalls on later instructions due to structural hazards (the lw's start relatively sooner). This results in the program taking 9 cycles instead of 16. This true in general of pipelining: deeper pipelines are more impacted by stalls than shallower pipelines.

(d) Suppose we want to execute three instructions on this two-stage pipelined CPU. The first instruction begins executing at the start of Cycle C0, the second begins at the start of Cycle C1, and the third, C2.
Fill in the correct control signals on each clock cycle in order to execute these instructions correctly:

- Any signals set by earlier instructions (before the first) should be set to "E".
- Any signals set by later instructions (after the third) should be set to "L".
- Indicate `Enable` or `Disable` for write enable signals.
- ImmSel should be set as `I, S, SB, U` or `UJ`.
- All other signals should be set as 0, 1, an ALU operation, or X (doesn't matter).
- The list of available ALU operations are `ADD, AND, OR, SRL, SRL, SLT`.

You may assume that there are no structural or data hazards.

```
Program:
    srl t1, t2, t3
    sw t0, 4(a0)
    bltu s0, t2, 44
```

Cycle	Signals								
	PCSel	ImmSel	RegWEn	BrUn	BSel	ASel	ALUSel	MemRW	WBSel
C0	0	X	E	X	E	E	E	E	E
C1	0	S	1	X	0	0	SRL	1	01
C2	0	SB	0	1	1	0	000	0	X
C3	X	L	0	L	1	1	000	1	X

PCSel is somewhat ambiguous for C3 so we are allowing any answer for that cell

The first step in solving this problem is to identify which control signals are set in which stage. This is because, as a pipelined datapath, two different instructions will be responsible for setting control signals during any given cycle. The assignments are as follows:

- **A:** ImmSel, BrUn
- **B:** RegWEn, BSel, ASel, ALUSel, MemRW, WBSel
- **PCSel:** Is a bit ambiguous. It's not really set by any particular instruction, but rather defaults to 0, but can be overwritten by earlier branches (if they predicted wrong).

Notice that RegWEn is physically on the left, but is actually set in the second stage (write-back). A common error was setting RegWEn in stage A.

Next we need to figure out what instructions are responsible for which control signals on a cycle-by-cycle basis (you could write this down in the margins or on your scratch paper).

- **C0:** SRL / E
- **C1:** SW / SRL
- **C2:** BLTU / SW
- **C3:** L / BLTU

Now we just need to figure out the control signals for each instruction as if it weren't pipelined, and then fill them into the table in the appropriate locations:

- **SRL:**
  - **ImmSel**: Doesn't use immediates (don't care)
  - **BrUn**: Not a branch (don't care)
  - **RegWen:** R-types write a result to the register, so this must be 1
  - **A/BSel:** R-types use registers for input so 0,0
  - **ALUSel**: ALU needs to do an SRL
  - **MemRW:** Not a memory instruction, so must be "read". It's not X because we mustn't write garbage into memory.
  - **WBSel:** R-types write ALU results back to a register so must be 01.
- **SW:**
  - **ImmSel:** Stores use an immediate for the offset, must ask the imm generator to do an S-type
  - **BrUn:** Not a branch, X
  - **RegWen:** Stores are from reg-¿mem, it doesn't write any registers: 0
  - **A/BSel:** Adds the immediate to a register value so B=1, A=0
  - **ALUSel:** Needs to add the offset, so set ALU to ADD
  - **MemRW:** Writes to memory, 1
  - **WBSel:** Doesn't write to register, X
- **BLTU:**
  - **ImmSel:** Uses immediate for branch target, use SB-type imm
  - **BrUn:** Branch is unsigned: 1

  - **RegWen:** Doesn't write to a register: 0
  - **A/BSel:** Branches in RISC-V are PC-relative, so it needs to add an immediate to the PC (1, 1)
  - **ALUSel:** Needs to add imm with PC: ADD
  - **MemRW:** Doesn't write memory: read
  - **WBSel:** Doesn't write registers: X

The last thing to mention here is that the PC is a bit ambiguous in this question.

- **C0:** Could be 0 or E. 0 because that's what seemed to happen (the next instruction in memory ran), but it could be E if you think E was a branch.
- **C1:** Must be 0 because there wasn't an earlier branch and stores don't affect the PC
- **C2:** Must be 0 because we said branches are predicted not taken
- **C3:** Could be anything because we didn't say whether the branch was taken or not.

Problem 5 ***$$$*** **(17 points)**

You are given following RISC-V Code:

```
// a0:  Integer array location
// a1:  End bound of the array
// a2:  Offset to new location in words
// Assume these registers hold the following values
// at the start of the program:
// a0 -> 0x1000, a1 -> 2048, a2 -> 2048

     add t0, x0, x0
     slli t3, a2, 2
loop:
     beq t0, a1, done
     slli t1, t0, 2
     add t1, t1, a0
     lw t2, 0(t1)
     add t1, t1, t3
     sw t2, 0(t1)
     addi t0, t0, 4
     jal x0, loop
done:
     ...
```

Solution: Equivalent C Code:

int a[512]; // At Address 0x1000

...

int b[512]; // At Address 0x1000 + 2048 × 4

for (int i = 0; i < 2048; i += 4) {

    b[i] = a[i];

}

Assume there is enough memory allocated for the array such that there are no memory out of bounds issues. Also assume the code is run on a machine with a 32-bit address space. Questions will only involve the code starting from `loop` and only refer to one data cache. This cache uses a LRU replacement policy and is write allocate unless otherwise stated.

(a) Consider an 8 words/block, 512B **direct-mapped** data cache. The cache starts empty and we run the program above to completion.

(i) Calculate the number of tag, index, and offset bits for this cache.

Solution: Tag: 23 Index: 4 Offset: 5
8 words/block under a 32-bit address space means $4 \times 8 = 2^5$ bytes/block.
As a result, this also means there are $512/(2^5) = 16 = 2^4$ blocks total.
We need a 32-bit address, so the tag bits must be the remaining 23.

(ii) What is the hit rate of this direct-mapped cache?

Solution: 0
In lw t2, 0(t1), t1 at first is just 0 + 0x1000, or 0b1000 0000 00000, so our cache pulls in 0x1000 to 0x101F into Index 0x0.
For the corresponding sw, t1 is the old t1 plus the offset (2048×4), or 0b11000 0000 00000. This maps to the same Index as before, so the previous block is evicted.
The next iteration's lw looks for 4×4 + 0x1000, or 0b1000 0000 10000, which maps to Index 0x0. It evicts the sw's block to make way for its own. This ping-pong of evictions continues, so the cache never returns a hit.

(iii) What types of of cache misses occur? Mark all that apply.

○ Capacity ● Conflict

● Compulsory

Solution: From (ii), we see that there are Compulsory misses if the program hits an array location it has never accessed before. The cycle of evictions is a result of Conflict misses, where the different Tags+matching Indices situation means previously-in-cache information is lost then re-accessed. There are no Capacity misses; adding more blocks won't solve this situation (and our current situation works as-if there was always just one-block).

(iv) Assume the cache is emptied and we re-run the program above to completion, but this time with a cache **block size of 4 words**. What is the hit rate of this new cache?

Solution: 0
No difference. Due to the offset between the two arrays, only the Tag differs between the two array addresses.

(b) Now consider an 8 words/block, 512B **Two-Way Set Associative** data cache. The cache starts empty and we run the program above to completion.

(i) What is the hit rate of this Two-Way Set Associative cache?

Solution: 1/2
We can handle the Tag-difference situation, but what now is important is our stride, or how i changes. A strides of 4 results in lookups like:
(lw) 0b01000 0000 00000
(sw) 0b11000 0000 00000
(lw) 0b01000 0000 10000
(sw) 0b11000 0000 10000
(lw) 0b01000 0001 00000
For each Tag-Index pair, we only see two possible accesses, so the cache helps get the second access faster. (Cache pulls entire block in)

(ii) What types of of cache misses occur? Mark all that apply.

○ Capacity ○ Conflict

● Compulsory

Solution: From (i), there are Compulsory misses if the program hits an array location it has never accessed before (e.g. new Tag-Index). Unlike (a), the lw and sw operations don't Conflict in the cache. There continues to be no Capacity misses; adding more blocks won't help our cache.

(iii) Assume the cache is emptied and we re-run the program above to completion, but this time with a cache **block size of 4 words**. What is the hit rate of this new cache?

Solution: 0
4 words/block = 16 bytes/block (2^4)
512/(2^4) = 32 blocks total (2^5)
Changing block size shifts our T:I:O allocations, so our accesses look like:
(lw) 0b01000 00000 0000
(sw) 0b11000 00000 0000
(lw) 0b01000 00001 0000
(sw) 0b11000 00001 0000
As seen, Tag-Index pairs are now unique; they never repeat. As such, the information in cache is never used, so there can be no hits.

(c) Now consider a 8 words/block, 512B **Four-Way Set Associative** data cache. The cache starts empty and we run the program above to completion.

(i) What is the hit rate of this Four-Way Set Associative cache?

Solution: 1/2
As seen from the memory access pattern in (b)(i), Tag-Index pairs come in two. Increasing Associativity does not change this, so making that cache Four-Way doesn't change the hit rate.

(ii) Assume the Four-Way Set Associative cache is emptied and we re-run the program above to completion, but this time with a **random replacement policy**. How would the hit rate most likely change compared to part c.i?

Increase   Stay the Same   [✓ Decrease]

Solution: The best replacement policy would be the one where the page evicted is the one that will not be accessed for the longest time. LRU is a better approximation of this strategy that Random Replacement, given that generally programs have some form of locality. The worst case for Random Replacement here is if we evict the block we need later, a situation in this scenario can't happen in LRU:

Assuming the Cache has one block left
(lw) 0b01000 0000 00000 # Block granted to 0b01000
(sw) 0b11000 0000 00000 # Cache full; Randomly Evict 0b01000
(lw) 0b01000 0000 10000 # Cache full; Randomly Evict 0b11000
(sw) 0b11000 0000 10000 # Cache full; Randomly Evict 0b01000
(lw) 0b01000 0001 00000

(d) Consider the 8 words/block, 512B **direct-mapped** data cache again. The cache starts empty and we run the program above to completion, **except this time with a2 initialized to 2056**. What is the hit rate of this direct-mapped cache?

Solution: 1/2
The offset is now 2056×4, or 0b10000 0001 00000. As a result, the lw/sw now don't share the same index location, so we don't have the eviction ping pong from (a).
(lw) 0b01000 0000 00000 Compulsory
(sw) 0b11000 0001 00000 Compulsory, but doesn't affect earlier lw info
(lw) 0b01000 0000 10000 Hit
(sw) 0b11000 0001 10000 Hit
(lw) 0b01000 0001 00000 Compulsory