Wawrzynek, Weaver
Fall 2021

CS 61C

Final

Solutions last updated: Saturday, March 2, 2024

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

PRINT your student ID: _____

Read the following honor code and sign your name.

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: _____

You have 170 minutes. There are 9 questions of varying credit (100 points total).

For questions with **circular bubbles**, you may select only one choice.

- ◯ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares (completely filled).

Anything you write that you ~~cross out~~ will not be graded. Anything you write outside the answer boxes will not be graded.

If an answer requires hex input, make sure you only use capitalized letters! For example, `0xDEADBEEF` instead of `0xdeadbeef`. Please include hex (`0x`) or binary (`0b`) prefixes in your answers. For all other bases, do not add the suffix or prefixes.

This page intentionally left with only one sentence.

Q1 *Potpourri* **(12 points)**

Q1.1 (1 point) True or False: The assembler translates code from a human-readable language (such as C) to an assembly language (such as RISC-V assembly).

○ True ● False

Solution: False, the compiler does this.

Q1.2 (1 point) True or False: Jumps made to statically linked libraries are fully resolved in the linker.

● True ○ False

Solution: True

Q1.3 (1 point) True or False: The OS allows for higher reliability; if a program has a bug, only that program will crash, instead of the entire system.

● True ○ False

Solution: True

Q1.4 (1 point) True or False: For high-performance network devices, polling tends to be used when there's a low data rate, while interrupts tend to be used when there's a high data rate.

○ True ● False

Solution: False; generally, the opposite tends to happen.

Q1.5 (1 point) True or False: A multithreaded program is considered correct as long as at least one order of the threads yields the correct answer, since we can force the scheduler to follow that thread order.

○ True ● False

Solution: False. You can't force schedulers to follow a thread order.

Q1.6 (1 point) True or False: It is cheaper to locate warehouse-scale computers in a cooler climate, in order to reduce total energy consumption.

● True ○ False

Solution: True

We decide to set up 10 1 TiB disks together in a single RAID configuration. What is the effective amount of storage we have if we decide to use:

Q1.7 (0.5 points) RAID 0?

[ ] TiB

Solution: In RAID 0, we don't have any redundancy, so we get a capacity of 10 TiB

Q1.8 (0.5 points) RAID 5?

[ ] TiB

Solution: In RAID 5, we get one parity block for every 9 data blocks, so we get a total capacity of 9 TiB

Q1.9 (1 point) We run the following code on two threads.

```
1  int y = 0;
2  int x = 10;
3  #pragma omp parallel
4  {
5    while(x>0)
6    {
7      y = y+1;
8      x = x-1;
9    }
10 }
```

What is the smallest possible value y can contain after this runs?

[ ]

Solution: This one's a bit tricky. The optimal sequence is:

Thread 1 reads y=0 and goes to sleep.
Thread 2 runs to completion.
Thread 1 wakes up and writes y=1, reads x=0, sets x=-1, then sees x == -1 and stops the loop.

Q1.10 (1 point) Justin purchased his HP Pavilion 15t-cs300 laptop 1,000 days ago. During this time, it has broken twice, and had to be repaired. Each repair took 10 days to complete, during which time the laptop was unusable. What is the mean time to failure (MTTF) of Justin's laptop, in days?

[ ] days

Solution: 490 days. 1,000 days minus 20 broken days = 980 days, divided by 2 failures.

Q1.11 (1 point) What is the availability of Justin's laptop?

Solution: 0.98. 980 days out of 1,000 days.

Author's note: Apparently I made a mistake when writing this question; I've only had my laptop for 2 years, not 3 years. Other than that, this is a fairly accurate representation of my laptop. Not that I'm bitter or anything...

Q1.12 (1 point) We've devised an error-correcting code which is able to fix 1 bit errors. If `0x61C` is a valid codeword, which of the following can NOT be a valid codeword, regardless of the error-correcting scheme we have? Select all that apply.

- ☐ `0x71C`
- ☐ `0xC16`
- ☐ `0x51C`
- ☐ `0x16C`
- ☐ `0x70D`
- ☐ None of the above

Solution: 0x71C and 0x51C cannot be valid codewords.

For 0x71C: If we received the data 0x71C, we would not be able to know if it was a correct codeword, or if we were supposed to receive 0x61C, but a bit got corrupted.

For 0x51C: If we received the data 0x71C, we would not be able to tell if the original word was 0x61C or 0x51C.

All others: The bit distance from 0x61C is far enough that we don't run into the problems above. We can construct an error-correcting code with those as codewords by defining our error-correcting code to have only those two as valid codewords; this is able to fix 1-bit errors, among other things.

Q1.13 (1 point) A program originally takes 1 second to run. We manage to parallelize 90% of our code to be 10 times faster, at the cost of 10 milliseconds of overhead. How many times faster is our new code?

Solution: 5 times faster. 1s / ((1s * 10%) + (1s * 90%) / 10 + 0.01s)

The easiest way to do this question is not to use the formula for Amdahl's law, but rather to treat it as a word problem. 90% of our code is 0.9 seconds worth of runtime, and that gets sped up to 0.09 seconds. We still have 0.1 seconds from our serial section, and add 0.01 seconds of overhead. This adds up to 0.2 seconds, which is 5 times less than our original runtime.

A useful note: Amdahl's law can almost always be solved in this "word problem" manner, by assigning an arbitrary runtime to the original code. I've generally found it more useful to know this instead of the formula itself.

Q2 *Rounding Errop* **(14 points)**

Note: we think Q2 and Q3 are harder questions. Feel free to skip them and come back later.

When working with floating point arithmetic, it is often the case that the exact result can't be stored in the floating point format. In this case, IEEE-754 defines the following rounding rule, which is commonly used: Compute the value precisely, then round to the nearest floating point number. In the event that the number is exactly halfway between two floating point numbers, round to the number with a least significant bit of 0.

For example, if we had a 10-bit minifloat with 5 exponent bits (and standard bias of -15) and 4 significand bits, the numbers 32 and 34 would be precisely representable (with no other representable numbers between them). When evaluating 32 + 0.5 = 32.5, we would round down to 32, while evaluating 34 - 0.5 = 33.5 would round up to 34. The expression 16 +17 = 33 would round to 32, because 32's binary representation as a 10-bit float is `0b0 10100 0000` (which has a 0 as its least significant bit), and 34's binary representation as a 10-bit float is `0b0 10100 0001`.

You may assume that for any two adjacent floating point numbers, one will have a LSB of 1 and the other will have a LSB of 0. Further, you may assume for this question that you will not need to round to infinity. Assume that any division in this question is float division (not integer division).

For the following questions, we will work with a 10-bit floating point representation that follows all conventions of IEEE-754 (including NaNs, denorms, etc.) but with 5 exponent bits (and standard bias of -15) and 4 significand bits.

What is the rounded values of the following (decimal) floating point numbers? You may express your answer either as an decimal value, or as an odd integer multiplied by a power of 2:

Q2.1 (3.5 points) 37

Solution: Answer: 36. The adjacent floating point numbers are 36 (0b 0 10100 0010) and 38 (0b 0 10100 0011)

Q2.2 (3.5 points) 1/3 (whose binary representation is 0b0.0101 0101…)

Solution: Answer: $21 * 2^{-6} = 0.328125$. We can move the binary point to get our floating point representation 0b1.010101... $* 2^{-2}$. The mantissa rounds down, so our float would be 0b1.0101 $*2^{-2}$, or 0b10101 $*2^{-6}$ or $21 * 2^{-6}$

We compute the following infinite sums under this floating point system, using left-association for addition (that is, a+b+c is evaluated in the order ((a+b)+c)), rounding after each addition. Eventually, this converges to some value, after which any further iterations don't change the sum. What is that value? You may express your answer either as an decimal value, or as an odd integer multiplied by a power of 2.

Q2.3 (3.5 points) $1 + (1/2) + (1/4) + \ldots$

Solution: We can look at how our mantissa changes:

0b1.0000

0b1.1000

0b1.1110

0b1.1111

0b1.11111 -> 0b10.000

0b10.000001 -> 0b10.000

Our answer is thus 2, which is mathematically correct.

Q2.4 (3.5 points) $2 + 2 + 2 + 2 + \ldots$

Solution: We don't need to worry about rounding until we get to the first nonrepresentable even number. This occurs at $0b1.00001$, with enough exponent that the right 1 becomes the 2s place. This is $0b1000010 = 66$. This will round down to 64, so we effectively get "stuck" there, and continuously get 64 from then. Thus, our answer is 64.

Q3 *Wait, why was this RISC-y, anyway?* **(14 points)**

Note: we think Q2 and Q3 are harder questions. Feel free to skip them and come back later.

Recall the definition of the function `verifypassword`:

The function `verifypassword` is defined as follows:

- Input: No register input; however, the function receives a string input from `stdin`.
- Output: `a0` returns 1 if the input from `stdin` is exactly `"secretpass"`, and 0 otherwise.

You have access to the following labels defined externally:

- Password: a pointer to a statically-stored string `"secretpass"`
- Get20chars: A function defined as follows:
  - Input: `a0` is a pointer to a buffer
  - Effect: Reads characters from `stdin`, and fills the buffer pointed to by `a0` with the read data, null-terminating the string. Your code may assume that the input is at most 19 characters, not including the null-terminator.
  - Output: None

You are a hacker, and you're currently trying to target the implementation of `verifypassword` presented on the midterm (copied below):

```
1  verifypassword:
2      addi sp, sp, -24 # Space for:
3      sw ra 20(sp)     #     ra
4      mv a0 sp         #     20-byte buffer
5      jal ra Get20chars
6      la t0 Password
7      mv t1 sp
8  Loop:
9      lb t2 0(t0)
10     lb t3 0(t1)
11     bne t2 t3 Fail
12     beq t2 x0 Pass
13     addi t0 t0 1
14     addi t1 t1 1
15     j Loop
16 Pass:
17     addi a0 x0 1
18     j End
19 Fail:
20     mv a0 x0
21 End:
22     lw ra 20(sp)
23     addi sp sp 24
24     jr ra
```

During the course of your testing, you discovered an interesting fact: the function `Get20Chars` doesn't actually work as intended! Instead of truncating at the 20th character, `Get20Chars` continues to write data until the first null terminator in its input. As before, verifypassword is located at `0x1000` and `Get20Chars` is located at `0x0F00`. Further, assume that the stack pointer is located at `0xBFFF F800` at the start of `verifypassword`, our page size is 4 KiB, and that we are currently working on a little-endian system.

Q3.1 (2 points) Our first step in exploiting this program is to find an input that changes the program flow. Submit a string that, if inputted in `stdin`, will cause `verifypassword` to return to the address `0xDEAD BEEF`. You may use the syntax `"A"*10` to denote a string consisting of 10 letter "A"s, and `"0xAB"` to signify the ASCII character corresponding to byte value `0xAB` (so "B" == "0x42"). For example, the answer `"A"*15+"0x42 0x42"+"C"` would correspond to the string `"AAAAAAAAAAAAAAABBC"`. (Hint: What gets changed if we write more than 20 characters?)

Solution: `"A" * 20 + "0xEF 0xBE 0xAD 0xDE"`

The hint reminds us that if we write more than 20 characters, we'll end up writing over whatever is located directly above (at a higher address than) the 20-byte buffer in memory. Looking at the code, we can see that we put the saved value of the ra register directly above the buffer on the stack.

Note that if we write `0xDEADBEEF` into the saved value of ra, then when the function returns, it will restore the saved value of ra on the stack back into the ra register, and the program counter will jump to the value in the ra register, which causes the program to return to the address `0xDEADBEEF`.

In order to do this, we need to do the following:

We need to save 20 bytes of data for the correct offset. It doesn't matter what we write there for now, so we just write "A"s.

We then need to write the return address, keeping in mind that we have a little-endian system. This evaluates to 0xEF 0xBE 0xAD 0xDE.

Note that because of the null terminator, we do end up messing up the word of data immediately before where the stack pointer was originally. We can't fix this here, but since we eventually inject instructions, we can write them to fix whatever data was there.

Scoring note: Note that by the convention set in this homework, we considered "0xDEAD-BEEF" to refer to "The data equivalent of 0xDEADBEEF in the system's endianness". As such, "0xDEADBEEF" was counted as equivalent to "0xEF 0xBE 0xAD 0xDE", but with the penalty for incorrect syntax (since we didn't define the 0x syntax for nonbytes). As a corollary, "0xEFBEADDE" was interpreted as "0xDE 0xAD 0xBE 0xEF".

Q3.2 (2 points) Now that we can move the program counter to an arbitrary location, we would like to jump to some RISC-V code that we've written. In order to do this, we decide to jump to the start of the buffer on the stack. What is the maximum number of RISC-V standard instructions we can inject into this buffer?

Solution: 5

The buffer is 20 bytes long. Each RISC-V instruction is 4 bytes long. In total, we can fit 20/4 = 5 instructions in the buffer.

Q3.3 (4 points) Regardless of your previous answer, assume that we can put up to 5 instructions in the buffer. Unfortunately, that's not really enough instructions to do much. Instead, we decide to inject code that lets us run longer programs, instead of only being limited to 5 instructions.

Complete the following 5-line code which does the above. You may use pseudoinstructions, as long as they resolve to exactly one instruction. Each blank is worth 1 point.

```
# Allocate a buffer of 256 bytes, which does not overlap with any data
# we already are using (such as the instructions injected in part 1)


1: addi _____________________________

# Set the argument of Get20Chars to the start of the allocated buffer


2: mv _________ sp

# Set t1 so the next instruction jumps to Get20Chars


3: lui _____________________________

4: jalr ra t1 -256 # Call Get20Chars

# Jump to the start of the buffer


5: ___________________________________
```

Solution: Line 1: `addi sp sp -280`. We allocate 256 bytes of buffer, plus 24 bytes to avoid overlap with the injected instructions and the saved ra register from part 1, for a total of 280 bytes. (This does mean that we rely on data after our stack pointer to stay constant, which is technically undefined behavior. In reality, the data will stay consistent for long enough that we move the stack pointer back down; since our sp starts in the middle of a page, the data will not be deallocated. The reasoning in the last sentence is not considered in scope, but is relevant in CS 162.)

Line 2: `mv a0 sp`. The `Get20Chars` function expects the address of a buffer as an argument in the `a0` register.

Line 3: `lui t1 1`. This puts the value `0x1000` in `t1`, which then causes the next line to jump to `0x1000 - 256 = 0x0F00`, the address of `Get20Chars`. Note that we needed to use lui here, since addi can only increase by up to 0x7FF. Line 4 could also not have been a jal operation, since our jump distance would have exceeded the size of a 20-bit immediate.

Line 5: `jr sp`. We want to jump to the address in the `sp` register, and we don't care about saving a return address anywhere.

Q3.4 (2 points) Translate the instruction `jalr ra t1 -256` to its hexadecimal machine language encoding.

(Binary answers will not be awarded credit.)

0x

Solution: `0xF00300E7`

Q3.5 (2 points) Briefly explain in 10 words or fewer why we cannot use this instruction in our injected code. (Hint: What would `Get20Chars` do if you tried to send this instruction as input?)

Solution: The instruction contains the byte `0x00`, which is treated as a null terminator, and would thus stop `Get20Chars` from writing the entire instruction, or overwriting the saved ra value on the stack.

The following answers were common enough that they warrant explanation:

"This doesn't call Get20Chars correctly/jumps to the wrong location": As noted in the code written above, it is possible to write code such that Get20Chars gets called. Indeed, this is one of the few ways that this is possible in 2 lines.

"We don't save the ra of verifypassword anywhere, so we end up infinite looping": While this would be the case in most programs, it's actually not the case here, because we get to input a new stdin every time Get20Chars gets run. Thus, if we did have an issue like this, we could run our malicious code on iteration 1, then on iteration 2 write code that just fixed the stack and jumped back to the right spot.

Q3.6 (2 points) Which of the following jump instructions can we use in our injected code? Don't worry about these lines not properly calling `Get20Chars`; we just want a valid RISC-V jump without running into the problem identified in part 5 (Hint: use the conversion you already did in part 4).

Note that +3840 == `0x0000 0F00`.

- [x] `jalr ra t2 -256`
- [x] `jalr ra t0 16`
- [ ] `ret`
- [ ] `jalr s0 x0 3840`
- [ ] `jalr x0 t1 -256`
- [x] `jalr s0 t1 -256`
- [ ] None of the above

Solution:

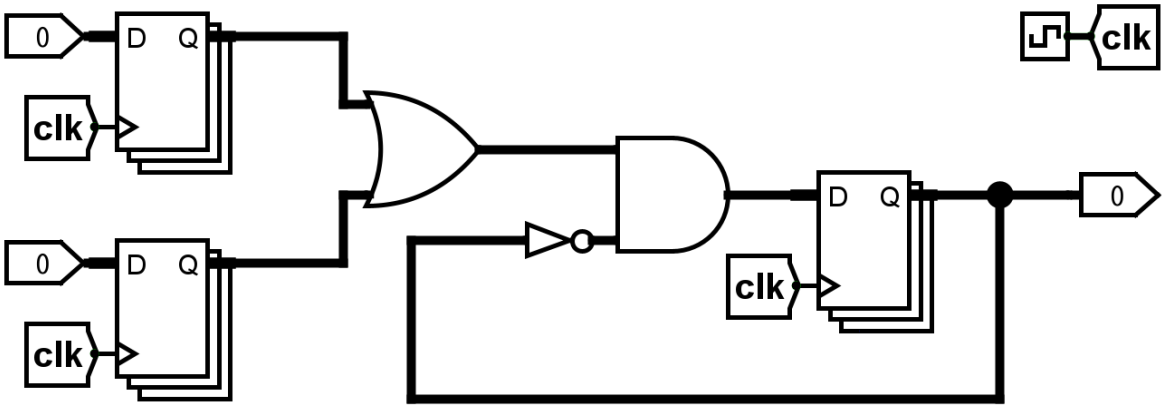
`ret = jr ra = jalr x0 ra 0` and `jalr x0 t1 -256` contain a null byte when translated to machine code: `0x00008067` and `0xF0030067`. Note that we can reuse our translation from part 4 and note only the places that get changed.

`jalr s0 x0 3840` is an invalid instruction because the immediate 3840 is greater than 2047. I-type immediates must be between -2048 and +2047.

`jalr ra t2 -256`, `jalr ra t0 16`, and `jalr s0 t1 -256` are valid, and don't have a null byte; note that `jalr ra t2 -256 = 0xF00380E7` doesn't have a null byte, because it gets split up into bytes as `0xE7 0x80 0x03 0xF0`.

Q4 *Bit of a Delay* **(10 points)**

Consider the following circuit. Assume that AND and OR gates have a delay of 8 ps (picoseconds), NOT gates have a delay of 4 ps, and all registers have a setup time constraint of 6 ps and clock-to-Q delay of 3 ps. Assume all wires are ideal, i.e. they have zero delay.



Q4.1 (2 points) What is the largest combinational delay of all paths in this circuit, in picoseconds?

ps

Solution: 16 ps

The longest path between two registers goes through the AND gate, then the OR gate, for a total delay of 8+8=16 ps.

Q4.2 (2 points) What is the smallest combinational delay of all paths in this circuit, in picoseconds?

ps

Solution: 12 ps

The shortest path between two registers goes through the NOT gate, then the AND gate, for a total delay of 8+4=12 ps.

Q4.3 (2 points) What is the maximum possible hold time constraint for registers to function properly in this circuit, in picoseconds?

ps

Solution: 15 ps

Hold time = smallest combinatorial delay + clock-to-Q delay = 12+3 = 15 ps

Q4.4 (2 points) What is the minimum allowable clock period for this circuit to function properly, in picoseconds?

[ ] ps

Solution: 25 ps

Shortest clock period = clock-to-Q delay + largest combinatorial delay + setup time = 6+16+3 = 25 ps

Q4.5 (2 points) What is the maximum allowable clock frequency for this circuit to function properly, in gigahertz?

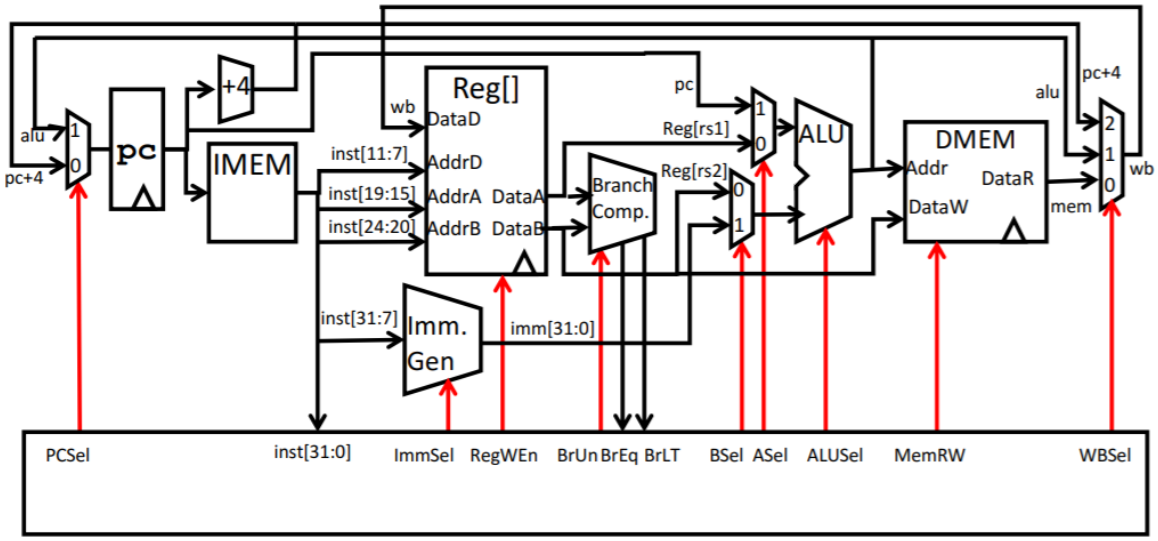
[ ] GHz

Solution: 40 GHz

1 / (25 ps) = 40 GHz

Q5 *Big Mac* **(10 points)**

Below is the standard RISC-V CPU used in Project 3.

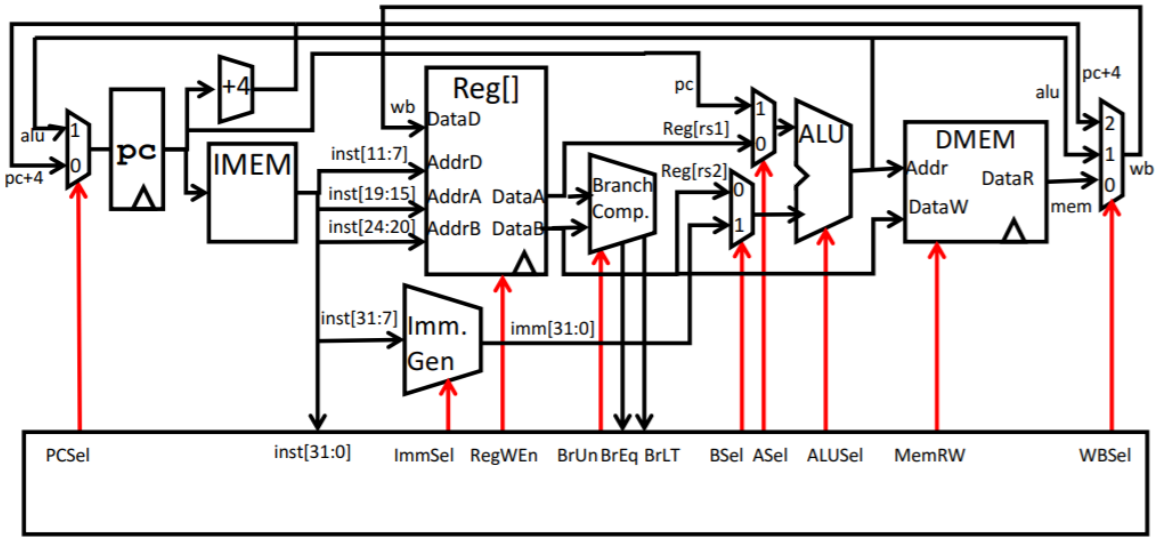


Q5.1 (4.5 points) For the instruction "lw", what are the control signals used? If a control signal doesn't matter, select "Don't Care".

Select one option per box. Each box is worth 0.5 points.

PCSel	● PC+4	○ ALU	○ Don't Care	
ImmSel	● I-type	○ S-type	○ B-type	○ U-type
	○ J-type	○ Don't Care		
ASel	○ PC	● `rs1`	○ Don't Care	
BSel	● Imm	○ `rs2`	○ Don't Care	
RegWEn	● 1	○ 0	○ Don't Care	
BrUn	○ 1	○ 0	● Don't Care	
ALUSel	● `add`	○ `sll`	○ `slt`	○ `xor`
	○ `srl`	○ `or`	○ `and`	○ `mul`
	○ `mulh`	○ `sub`	○ `sra`	○ `bsel`
	○ Don't Care			
MemRW	● Read	○ Write	○ Don't Care	
WBSel	○ ALU	● MEM	○ PC+4	○ Don't Care

The CPU is reproduced here for convenience.



Q5.2 (3 points) We want to add a new instruction `mac` (multiply and accumulate) to our CPU:

`mac rd, rs1, rs2`

Set `rd` to `rd + (rs1 * rs2)`.

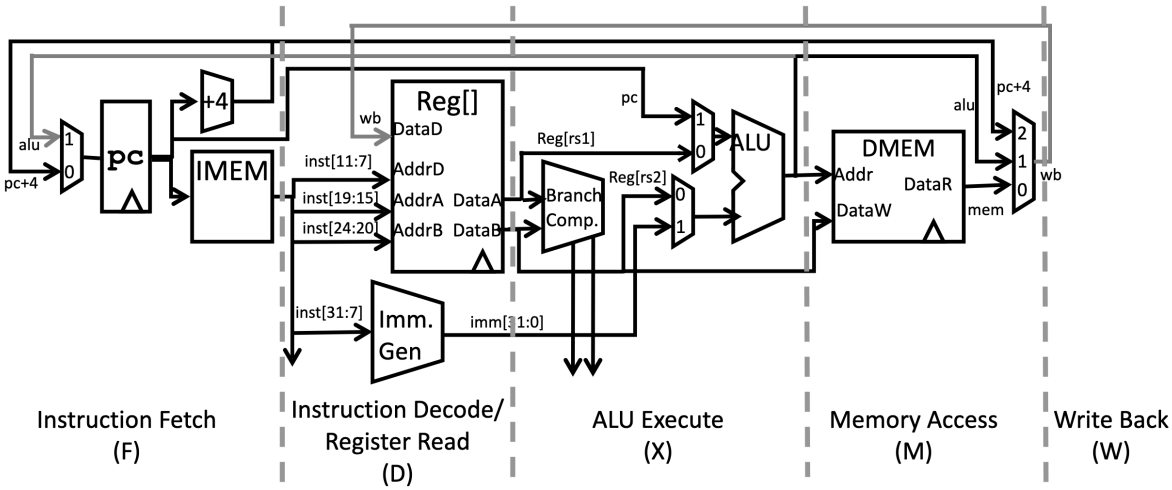
What changes would we need to make to our datapath in order for us to implement this instruction (with as few changes as possible)? Select all that apply.

- ☐ Add a new instruction format
- ☐ Add a new immediate type for the ImmGen
- ☐ Add a new rs3 input to RegFile
- ■ Add a new output to RegFile for a third register value
- ☐ Add a new input to AMux and update the relevant selectors/control logic
- ☐ Add a new input to BMux and update the relevant selectors/control logic
- ■ Add a new ALU input for a third register input
- ■ Add a new ALU operation and update the relevant selectors/control logic
- ☐ Add a new input to WBMux and update the relevant selectors/control logic
- ☐ None of the above

Solution:

We now have to read three values from the RegFile to compute `rd + (rs1 * rs2)`, so we need a third output from RegFile. However, we can reuse the R-type format, and use the rd input to RegFile to determine what the third output should be; as such, we don't need a new instruction format or rs3 input.

We also need to pass three values into the ALU to calculate `rd + (rs1 * rs2)`, and we need a new ALU operation to compute the multiplication and addition together on one cycle. The remaining components do not need to be updated.



Q5.3 (2.5 points) Write a sequence of instructions that causes a hazard in a completely unoptimized 5-stage pipeline (no forwarding, no branch prediction, no synchronous read/writes, etc), but which would not cause a hazard if all mac instructions were changed to mul instructions. If no such sequence exists, write "Not Possible."

Solution: Note that regardless of the values stored in registers, we still need to stall (since we don't change operation based off ID), so we can write code without initializing registers.

The extra hazard that occurs as a result of this instruction is a data hazard on rd; with mul, we don't need to wait for the value of rd to get written back, but we do need to wait for rd for a mac.

Thus, a correct answer required a mac instruction up to 3 instructions after its rd got updated, and no other hazards involving other registers. Our staff solution was:

mac a0 a1 a1

mac a0 a1 a1

Q6 *Boolean Code Golf* **(10 points)**

For this question, consider each truth table of inputs and expected outputs. Write Boolean expressions that, given the inputs W, Y, and Z, evaluate to the given output.

Your answer should consist of the following characters:

W, Y, Z	The inputs
$\sim$	NOT
|	OR
&	AND
$\wedge$	XOR
()	Parentheses
1, 0	Constants

Each question specifies a par score, which is the target number of Boolean operations to use. For full credit, your solution must use at most the par score number of operations ($\sim$, |, &, and $\wedge$ each count as one operation). Partial credit will be awarded for fully correct solutions that slightly exceed the par score. The par score is not necessarily the minimum number of operations required.

Operator precedence will follow standard C operator precedence. We will NOT offer partial credit for assuming incorrect operator precedence, so use parentheses when uncertain.

Q6.1 (2.5 points) Par 1 (1 Boolean operation)

W	Y	Z	Out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Solution: Staff solution: $\sim W$

Other answers may be possible, such as $W \wedge 1$

In this one, we note that our output doesn't depend on Y or Z, so we can make an output using just W.

Q6.2 (2.5 points) Par 2

W	Y	Z	Out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Solution: Staff solution: $\sim (Y \& Z)$

Other answers may be possible.

Similar to the previous, we note that the answer does not depend on W.

Q6.3 (2.5 points) Par 3

An X is used to signify that either 1 or 0 can be outputted for the corresponding input.

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

Solution: Staff solution: $W \wedge Z$

Other answers may be possible (Notably, this question can yield a score well below par).

In this case, we note that there's never a time when Y distinguishes between 1 and 0; as such, Y never affects the result of our output, and we can look at the reduced truth table containing only W and Z.

Q6.4 (2.5 points) Par 4

W	Y	Z	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Solution: Staff solution: $(\sim W \& Y) | (W \& Z)$

Other answers may be possible. Congratulations to the three students who found the following solution in 3 operations: $Y \wedge (W \& (Y \wedge Z))$

In this case, we can't easily do our analysis like we did in earlier questions. This can be either solved by starting with the sum-of-products form and simplifying, or by noting that the patterns when W is on and when W is off are both simple.

Author's note: This question was constructed by random number generator.

Q7 *<insert obligatory money pun here>* **(10 points)**

A program is run on a byte-addressed system with a single-level cache, where memory addresses are 10 bits long. After a while, the entire cache has the following state:

Index	Tag 1	Valid 1	Tag 2	Valid 2
0b00	0b1011	1	0b1101	1
0b01	0b0011	1	0b0010	1
0b10	0b1110	1	0b0111	0
0b11	0b1111	0	0b0001	0

Q7.1 (1 point) What is the associativity of the cache?

Solution: 2

Each index has two cache entries (as seen by the two tags), so the cache is 2-way set associative.

Q7.2 (1.5 points) What is the T:I:O breakdown of memory addresses?

T	I	O

Solution: 4:2:4

From the table, each tag is 4 bits long, and each index is 2 bits long. Since the address is 10 bits long in total, the offset must be 10-4-2=4 bits long.

Q7.3 (1.5 point) How many bytes of data can this cache contain?

Solution: 128

There are 4 different indices (2 index bits = 2^2 possible indices), and each index contains 2 cache entries (2-way set associative), so there are $4 \times 2 = 8$ cache entries in total.

The offset is 4 bits long, so each cache entry holds $2^4 = 16$ bytes of data. In total, there are $16 \times 8 = 128$ bytes of data in this cache.

Q7.4 (6 points) For each of the following memory accesses, determine if each access would be a hit or miss based on the cache state shown above, and if it's a miss, classify the possible miss type(s). If multiple miss types are possible depending on prior memory accesses, select all possible miss types.

Note: For this question, each memory access should be considered in isolation. In particular, do not update the cache state after each memory access.

Address				
`0b 0011011011`	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
`0b 0011001101`	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
`0b 0110100010`	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
`0b 0010111100`	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
`0b 1110100010`	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss
`0b 1111111111`	☐ Hit	☐ Compulsory miss	☐ Capacity miss	☐ Conflict miss

Solution: The crux of this question is that data is always set to some garbage (there's no such thing as blank in binary, and we often don't zero out data if we can avoid it; as such, the cache generally ends up containing whatever data used to be there the last time a program ran), even if the cache block is empty. As such, the valid bit is needed to tell if a block is actually there. If the valid bit is 0, then the block is considered empty, regardless of the corresponding tag.

1: Hit; we notice that Tag 1 matches for our given index, and the valid bit is on.

2: Miss; Our tag isn't in our index. This can either be compulsory or conflict, since we don't know if this block has been accessed before and ejected. This can't be a capacity miss, because there are still empty slots in our cache.

3: Miss; Our tag isn't in our index. This must be compulsory, because we still have an empty slot in this index. We only ever kick out a block when the cache is full, and only to replace that block with a new one; as such, we can't have kicked out a block while there is still empty space in the cache.

4: Miss; as with 3, this must be compulsory

5: Hit; this one hits the valid block in that index.

6: Miss; As noted above, we don't count this as a hit, since our valid bit is off.

Q8 *Deja-VM* **(10 points)**

One useful aspect of virtual memory is that it allows two distinct programs to try and access the same virtual memory addresses, and still get different data locations. Consider a system with 64 MiB of physical memory, which runs programs that have 4 GiB of virtual memory. Our page size is 4 KiB.

Q8.1 (1 point) How many virtual pages do we have? Express your answer as a power of 2.

Solution: $2^{32}/2^{12} = 2^{20}$

Q8.2 (1 point) How many bits long is a physical address?

Solution: $\log_2(64Mi) = 26$

Q8.3 (1 point) How many bits long is a virtual page number?

Solution: $\log_2(2^{20}) = 20$

Regardless of your answer to Q8.2, assume that physical addresses are 20 bits long. We run two programs (with no shared memory), which access the following virtual memory addresses in order. For each memory access, determine the physical address that gets accessed, writing your answer in hexadecimal.

Assume that no physical pages are in use prior to the first memory access, and that physical pages get assigned in order of physical page number (so page 0 is assigned first, then page 1, and so on).

Q8.4 (1 point) Program 1: `0xABCDEFAB`

Solution: This is our first page, so we get the physical page $0x00$. The page offset is the last 12 bits of our address (since our page size is 2^{12} bytes), so we take the last 12 bits of the virtual address: `0x00FAB`.

Q8.5 (1 point) Program 1: `0x12345678`

Solution: This is a different virtual page number, so we get the physical address `0x01678`.

Q8.6 (1 point) Program 1: `0xABCDD312`

Solution: This is also new page (despite the same first 19 bits), so we get the physical address `0x02312`.

Q8.7 (1 point) Program 2: `0xABCDEFAB`

Solution: This is the same address as before, but it's for a new program, so we need a new page. We get the physical address `0x03FAB`.

Q8.8 (1 point) Program 2: `0x12345664`

Solution: As with before, this is a new address for this program, so we get the physical address `0x04664`.

Q8.9 (1 point) Program 1: `0x12345664`

Solution: This is a hit on program 1's version of this page, so we reuse the same page number from before. We get the physical address `0x01664`.

Q8.10 (1 point) Program 2: `0xABCDEFAB`

Solution: This is a hit on program 2's version of this page, so we reuse the same page number from before. We get the physical address `0x03FAB`.

Q9 *Testception 2, 3, 4, AND 5! Now simulcasting!* **(10 points)**

Fred's Factorization Factory has unveiled their latest product: an algorithm that factorizes an array of numbers provided. You want to test their factoring algorithm, so you decide to write the following function:

`int testFactor(uint32_t n, uint64_t *a, uint64_t *b, uint64_t *c);`

- n: The length of each list of integers. For simplicity, you may assume that n is a multiple of 4.
- a, b, c: Pointers to arrays of 64-bit integers.

`testFactor` returns 1 if, for all `i` from 0 to n-1, `a[i]*b[i] == c[i]`. Otherwise, it returns 0.

You have access to the following SIMD instructions:

- `_mm256 vectorLoad(void* ptr)`: Loads four `uint64_t` from `ptr` into a SIMD vector
- `void vectorStore(void* ptr, _mm256 mm)`: Stores the four `uint64_t` in `mm` at `ptr`
- `_mm256 vectorMul(_mm256 a, _mm256 b)`: Multiplies the values in `a` and `b`, and returns the result
- `_mm256 vectorSet0()`: Returns a vector containing only 0s.
- `_mm256 vectorOr(_mm256 a, _mm256 b)`: Computes the bitwise OR of the two vectors, and returns the result.
- `_mm256 vectorXor(_mm256 a, _mm256 b)`: Computes the bitwise XOR of the two vectors, and returns the result.

```
int testFactor(uint32_t n, uint64_t *a, uint64_t *b, uint64 *c)
{
    uint64_t output[4];

    _mm256 total = ____________________________________________;

    for(int i = 0; i < ___________________; i+= ___________________)
    {
        _mm256 adata = vectorLoad(a+i);
        _mm256 bdata = vectorLoad(b+i);
        _mm256 cdata = vectorLoad(c+i);

        _mm256 prod = ____________________________________________;

        _mm256 isequal = ____________________________________________;

        ____________________________________________________________;
    }
    vectorStore(output, total);

    return ____________________________________________________ ? 1 : 0;
}
```

Solution: Blank 1: `vectorSet0()`

Blank 2: `n`

Blank 3: `4`

Blank 4: `vectorMul(adata, bdata)`

Blank 5: `vectorXor(prod, cdata)`

Blank 6: `total = vectorOr(total, isequal)`

Blank 7:

```
(output[0] == 0) && (output[1] == 0) && (output[2] == 0) && (output[3] == 0)
```

Other solutions may exist. Note that the solution:

`output[0]+output[1]+output[2]+output[3] == 0` is incorrect, since we could have received outputs that happened to add to 0, even if they aren't all 0. As an example consider the inputs $A = [0, 0, 0, 0]$, $B = [0, 0, 0, 0]$, $C = [1 << 30, 1 << 30, 1 << 30, 1 << 30]$.

The main idea of this accumulator is noting that two numbers are equal if and only if their XOR is exactly 0. By ORing next, we ensure that the total values are nonzero if any instance of isequal ended up returning a nonzero value. Thus, we can just check if the output vector is all zero at the end.

In general, bitwise operations tend to be much faster than branch comparators, so it is generally preferable to use bitwise and simple arithmetic operations when possible.