CS 61C Summer 2024 — Andrew, Eddy, Jedi, Nikhil — Final

Solutions last updated: Sunday, August 11th, 2024

PRINT Your Name: ______________________________

PRINT Your Student ID: ______________________________

You have 170 minutes. There are 11 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Points:	15	9	8	8	14	5	6	14	12	9	0	100

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

- ○ Unselected option (completely unfilled)
- ● Only one selected option (completely filled)
- ✓ Don't do this (it will be graded as incorrect)

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

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

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation.

A list of statements that apply to the entire exam is on the last page.

Write the statement below in the same handwriting you will use on the rest of the exam.

I have neither given nor received help on this exam (or quiz), and have rejected any attempt to cheat; if these answers are not my own work, I may be deducted up to 0x0123 4567 89AB CDEF points.

SIGN your name: ______________________________

Clarifications made during the exam:
Q1.2: … to **two's complement** 8-bit hexadecimal.
Q3: The adder performs two's complement addition.

Q1 *Potpourri* (15 points)

Q1.1 (1.5 points) Convert the unsigned number 2101_3 to 8-bit hexadecimal.

0x40

Solution: `0x40.` Since we're working in base 3, we can first convert this to a decimal number as $1 * 3^0 + 0 * 3^1 + 1 * 3^2 + 2 * 3^3 = 64$. We can divide 64 by 16 four times, giving us 0x40, which is 8-bits long.

Q1.2 (1.5 points) Convert the decimal number −33 to 8-bit hexadecimal.

0xDF

Solution: `0xDF.` To convert a decimal number to hex, we can first translate it into binary. Since we're working in two's complement, we can use the trick of first representing 33 in binary, then invert the bits and adding one to yield -33. 33 in binary is `0b0010 0001`. Flipping the bits and adding one yields `0b1101 1111`. We can then translate this to hex to yield our final answer.

Q1.3 (1 point) Return of attack of the flippy bit! Convert the 8-bit hexadecimal `0xEB` to binary.

0b1110 1011

Solution: `0b1110 1011.` Attack of the flippy bit!! (only students from Summer 2024 will know what this means...)

Q1.4 (2 points) Given an IEEE-754 standard floating point representation with 10 bits: 1 sign bit, 4 exponent bits (with a standard bias of -7), and 5 significand bits, how many NaNs can be represented?

62

Solution: `62.` In order to have a NaN representation, two conditions must be met: first, we must have an exponent of all 1's, and the significand must be nonzero. Therefore, the sign bit can take on one of two possibilities. There is only one configuration of the exponent bits that can yield all 1's, and finally, given five significand bits, there are 2^5 possible values, and $2^5 - 1$ non-zero representations. Multiplying this out, you get $2 * 1 * (2^5 - 1)$, yielding 62 as our final answer.

Q1.5 (2 points) Convert the following RISC-V machine code into its corresponding instruction. Express immediates in decimal and use the appropriate register names instead of numbers (i.e. `s5` instead of `x21`).

`0x00F9 0E63`

`beq s2 a5 28`

Solution: `beq s2 a5 28.` Translating the machine code into binary, we get `0b0000 0000 1111 1001 0000 1110 0110 0011.` We can examing the lower 7 bits to obtain the opcode `110 0011`, which tells us that we have a B-type instruction. Using this information, we can split up the instruction into the following fields:

imm[12|10:5]: `0000000`

rs2: `01111`

rs1: `10010`

funct3: `000`

imm[4:1|11]: `11100`

opcode: `1100011`

Given the funct3 combined with the fact that we are working with a B-type instruction, we know we are working with a **`beq`** instruction.

rs1 corresponds with register x18, which maps to s2. rs2 corresponds with register x15, which maps to a5.

In order to reconstruct the immediate, we can piece together the immediate to obtain 0000 0000 1110 (given the immediate in the machine code). After adding the implicit 0 in the LSB, we get 0000 0000 1110 0. This corresponds to a byte-offset of 28.

We can put all this together to yield the instruction `beq s2 a5 28`.

Q1.6 (1 point) True or False: Given the single-cycle datapath, PCSel should be 1 for all branch instructions.

True [✓ False]

Solution: False. From the reference card, `PCSel = 1` means that the Program Counter is set to the output of the ALU, which means that the branch is taken. However, the branch is not taken when the condition is not met (e.g. if we have a beq instruction and rs1 and rs2 are not equal, we should set PC to PC + 4, not the output of the ALU) and PCSel should be set to 0.

Q1.7 (1 point) True or False: Given the 5-stage pipelined datapath with double-pumping and **no** forwarding, all data hazards will require at most 2 stalls.

[✓ True] False

Solution: True. Data hazards occur when there is some data dependency. The earliest stage needed to retrieve information is Instruction Decode (from registers), and the latest stage needed to store modified information is Write Back (to registers). Since we have double pumping enabled, we can write at the rising edge of the clock, and read at the falling edge. Therefore, we only need two stalls in order for the Write Back stage of the first instruction to match the same clock cycle as the Instruction Decode stage of the next instruction.

Q1.8 (1 point) True or False: Pipelining increases the clock frequency by decreasing instruction latency.

True [✓ False]

Solution: False. Pipelining increases instruction throughput in terms of instructions per unit time, but instruction latency is always longer in a pipelined datapath since your clock frequency is limited by the slowest stage.

Q1.9 (1 point) True or False: Stack-allocated variables declared within a `#pragma omp parallel` section are shared amongst all threads.

True [✓ False]

Solution: False. Stack-allocated variables are private to all threads inside of a `#pragma omp parallel` section.

Q1.10 (1.5 points) Suppose that a memory system with the following hit time and hit rates has an AMAT (average memory access time) of 97ns. What is the hit time of DRAM?

	Hit Time	**Hit Rate**
L1 Cache	10ns	70%
L2 Cache	50ns	80%
DRAM	?	100%

1200 ns

Solution: `1200 ns.` We can calculate the hit time of DRAM using the formula AMAT = HitTime + MissRatio * MissPenalty. Since the hit rate of the caches + DRAM exceeds 100%, we know we're dealing with local hit rates.

Therefore, our formula is 97 = 10 + 0.3 (50) + 0.3 * 0.2 (DRAM Hit Time). You can also set up the equation as 97 = 10 + 0.3(50 + 0.2(DRAM Hit Time)). Both yield a DRAM Hit Time of 1200 ns.

Q1.11 (1.5 points) Suppose we have a program that takes 100 minutes to run. If 50% of the code can be sped up by a factor of 5, what is the runtime of the optimized program?

60 minutes

Solution: `60 minutes.` We can apply Amdahl's law to find the overall speedup, which is calculated as 1 / ((1 - 0.5) + (0.5 / 5)). This simplifies to 1 / 0.6, or a 5/3x speedup. We can use this to find the runtime of the optimized problem - with a 5/3x speedup, we can invert the speedup to find the optimized runtime relative to the runtime of the old program. 100 x 3/5 = 60 minutes.

Q2 *Cones* (9 points)

Implement `count_ones`, a RISC-V function, defined as follows:

`count_ones`: Count the number of `1`'s in a number's binary representation		
Arguments	`a0`	A 32-bit number.
Return value	`a0`	The number of `1`'s in `a0`'s binary representation.

For example, calling `count_ones` on `0b1000 0110 0001 1100 1011 1110 1110 1111` would return 19 in `a0`.

```
 1  count_ones:
 2      beq a0 x0 zero_case
 3      addi sp sp -8
              Q2.1
 4      sw ra 0(sp)
             Q2.2
 5      sw s7 4(sp)
             Q2.3
 6      slt s7 a0 x0
                 Q2.4
 7      slli a0 a0 1
              Q2.5
 8      jal ra count_ones
 9      add a0 a0 s7
                 Q2.6
10      lw ra 0(sp)
             Q2.7
11      lw s7 4(sp)
             Q2.8
12      addi sp sp 8
              Q2.9
13      jr ra
14  zero_case:
15      li a0 0
16      jr ra
```

Solution: Since the skeleton code used the `s7` register which is callee saved, and we notice a recursive call in line 8 modifying `ra`, we need to store two registers on the stack. We begin by decrementing our stack pointer by 8 bytes to handle the two registers, then storing the values of both registers at the appropriate offsets. We later load in both values and increment the stack pointer accordingly.

In order to count the number of ones, we can set the value in s7 equal to 1 if the MSB is 1, and 0 otherwise. We do this by comparing the input to zero (since if the MSB is 1, we can interpret the input as a negative number). We can then shift a0 left once so that we can eventually hit our base case, which checks if a0 is equal to zero. We then call `count_ones` recursively, and take our recursive leap of faith by adding s7 to the return value of the recursive call.

This page intentionally left (mostly) blank.

The exam continues on the next page.

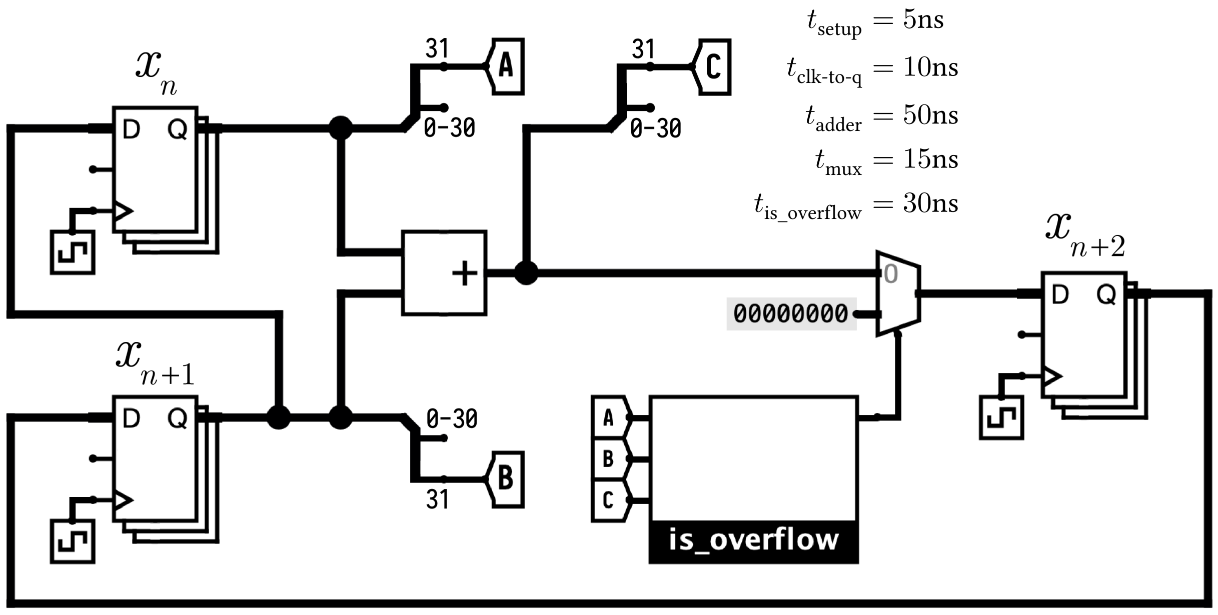
Q3 *Jeddy Sequence* (8 points)

Suppose an Eddy sequence of integers is $x_{n+2} = x_{n+1} + x_n$. For example, given $x_0 = 0$ and $x_1 = -1$, then the Eddy sequence would be as follows:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	$\cdots$
0	-1	-1	-2	-3	-5	-8	$\cdots$

Jedi built the following circuit to calculate the next number in an Eddy sequence. At $t = 0$, assume register x_n has value x_0, register x_{n+1} has value x_1, and register x_{n+2} has value x_2. Assume components have the combinational logic delays listed, and other components have negligible delay.

The `is_overflow` subcircuit outputs a `1` if the adder overflows and `0` otherwise. If the adder overflows, the circuit should store `0x00000000` in the x_{n+2} register.


$t_{\text{setup}} = 5\text{ns}$
$t_{\text{clk-to-q}} = 10\text{ns}$
$t_{\text{adder}} = 50\text{ns}$
$t_{\text{mux}} = 15\text{ns}$
$t_{\text{is\_overflow}} = 30\text{ns}$
x_n
x_{n+1}
x_{n+2}
D Q
31
0-30
A
B
C
+
0
00000000
is_overflow

Q3.1 (2 points) What is the **minimum clock period** for this circuit to function properly, in nanoseconds?

110 ns

Solution: `110 ns.` The minimum clock period is calculated as clk-to-q + maximum combinational logic delay + setup time. The maximum combinational logic delay goes through the adder, which feeds into the `C` tunnel. Then, that input feeds into the `is_overflow` subcircuit, then into the mux. This yields a total of 50 + 30 + 15 = 95 ns combinational delay. Combined with the clk-to-q of 10 ns and setup time of 5 ns, we get 110 ns.

Partial credit was given for a clock period of 80 ns, calculated as just going through the adder and the mux.

Q3.2 (2 points) What is the **maximum hold time** the registers can have such that there are no hold time violations, in nanoseconds?

10 ns

Solution: `10 ns.` The maximum hold time is a function of how quickly the input to a register can change. Since we can see a wire going directly from the x_{n+2} register to the x_{n+1} register, the value going into the x_{n+1} register can change after the output value of the x_{n+2} register changes, which occurs after the clk-to-q delay. Therefore, the answer is 10 ns.

Q3.3 (2 points) Write a simplified Boolean expression that describes the behavior of the `is_overflow` subcircuit. You may use at most 7 operators. NOT (`!`), AND (`*`), OR (`+`) each count as one operator. We will assume standard C operator precedence, so use parentheses when uncertain.

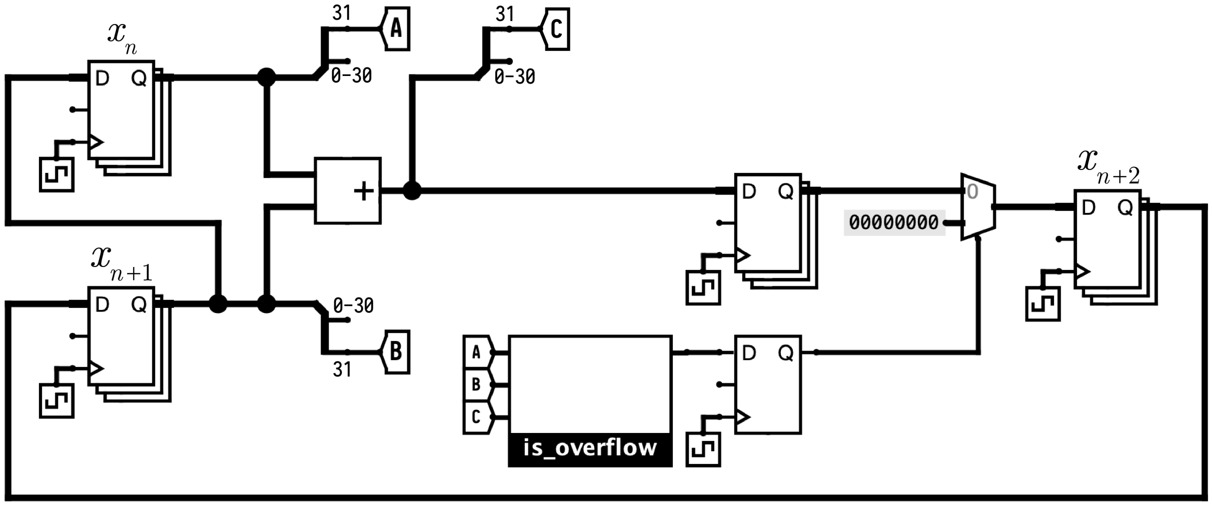
Your answer may consist of the following:

Inputs	NOT	AND	OR	Constants	Parentheses
`A, B, C`	`!`	`*`	`+`	`0, 1`	`()`

```
C*!(A+B) + !C*A*B
```

Solution: `C*!(A+B) + !C*A*B`. There are two cases where the sum overflows. Either the MSB of the two inputs is 0 and the output is 1, or the MSB of the two inputs is 1 and the output is 0. The MSB of the two inputs is captured in A and B, and the MSB of the output is captured in C. Therefore, we can write out the initial boolean expression `C*!A*!B + !C*A*B`. However, this requires 8 operators. To simplify this down to 7 operators, we can apply DeMorgan's to the first term to yield our final answer.

Jedi wants to generate Eddy sequence numbers faster by pipelining the circuit! The proposed pipeline registers are shown in the diagram below, dividing the circuit into two stages.



Q3.4 (2 points) True or False: Will the added pipeline registers improve the maximum throughput in additions/cycle of the circuit? Briefly explain your answer in 15 words or less.

True [✓ False]

False. The circuit still performs one addition per cycle.

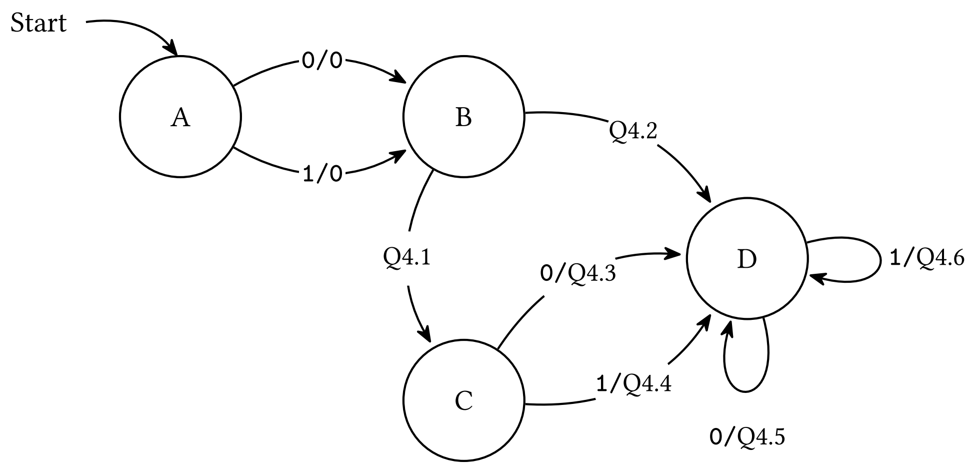
Q4 *iFPSM* **(8 points)**

The iFPSM is a finite state machine that identifies if a floating point number is $+\infty$ or $-\infty$. The IEEE-754 standard representation of the floating point number will be passed bitwise as the input into the iFPSM starting from the MSB (the sign bit).

Input	Output
$+\infty$ or $-\infty$	An all zero bitstring
Non-infinite value	Any non-zero bitstring

Below is an iFPSM for the IEEE-754 standard floating point representation with 6 bits: 1 sign bit, 2 exponent bits (with a standard bias of -1), and 3 significand bits.

For Q4.1 – Q4.6, select the state transitions such that the iFPSM matches the described behavior.



Q4.1 (1 point)

○ 0/0 ○ 0/1 [✓ 1/0] ○ 1/1

Q4.2 (1 point)

○ 0/0 [✓ 0/1] ○ 1/0 ○ 1/1

Q4.3 (0.5 point)

○ 0 [✓ 1]

Q4.4 (0.5 point)

[✓ 0] ○ 1

Q4.5 (0.5 point)

[✓ 0] ○ 1

Q4.6 (0.5 point)

○ 0 [✓ 1]

Solution: A floating point number represents infinity if the exponent is all ones and the significand is all zeros. In that case, we should always output a zero. The transitions from A to B indicate the sign bit, since we should output a 0 for both negative and positive infinity. We observe D as a terminal state (with two self loops). Thus, if we detect a 0 in our exponent bits, we can move directly into the terminal state and output a 1. Since we have two exponent bits, we need two states (B and C) to check that both exponent bits are 1. Once we reach state D, if we detect a 1 at any time in the significand, we should output a 1. Otherwise, we should output a 0.

Let's extend our iFPSM to be compatible with any number of floating point bits.

Assume we are working with an IEEE-754 standard floating point representation with 1 sign bit, e exponent bits (with a standard bias of b), and s significand bits.

Q4.7 (2 points) What is the fewest number of states that a correct iFPSM must have? Your answer may include e, b, and/or s.

For example, the iFPSM from Q4.1 – Q4.6 has 4 states: A, B, C, and D.

$e + 2$ states

Solution: Generalizing the above solution, we need 1 state for the sign bit, `e` states for the exponent bits (since each exponent bit needs to be 1), and finally, 1 state for the significand bits (since we can handle both cases where the significand is zero or nonzero there). `1 + e + 1` simplifies to `e + 2`.

Q4.8 (2 points) The nFPSM is a finite state machine like the iFPSM except it identifies if a floating point number is NaN.

Input	Output
NaN	An all zero bitstring
Non-NaN value	Any non-zero bitstring

What is the fewest number of states that a correct nFPSM must have? Your answer may include e, b, and/or s.

$e + s + 1$ states

Solution: We use similar logic for the sign bit and the exponent bits. However, for NaN, the significand must be nonzero. Consequently, we need one state for every significand bit in order to confirm that the significand is indeed nonzero (consider the case where the significand bits are 00...000 - you would need to process every single bit until you reach the last one to confirm that the input is indeed zero, meaning that the output should be nonzero. Thus, you can draw a FSM where the last state outputs a 1 if you see 0 (meaning you have a significand of all 0s), and a self loop at all other states with 1/0. This gives us 1 state for the sign bit, **e** states for the exponent bits, and **s** states for the significand bits.

Q5 *Jump and Lychee* (14 points)

For the entirety of this question, assume we are working with the single-cycle datapath.

A `ja` instruction in RISC-V is a new instruction described as follows:

`ja rd rs1 imm`

```
PC = rs1 & imm
rd = PC + 4
```

For each of the following control signals, indicate the value it should always have for `ja`.

Q5.1 (1 point) PCSel

○ 0 ○ None of the above [✓ 1]

Solution: PC is set to `rs1 & imm`. Therefore, PCSel should be 1 to retrieve the output of the ALU.

Q5.2 (1 point) ASel

[✓ 0] ○ None of the above ○ 1

Solution: We need the value of rs1 to compute `rs1 & imm`. Therefore, ASel should be 0.

Q5.3 (1 point) BSel

○ 0 ○ None of the above [✓ 1]

Solution: We need the output of the immediate generator to compute `rs1 & imm`. Therefore, BSel should be 1.

Q5.4 (1 point) ALUSel

○ `add` ○ `or` ○ `sub` [✓ `and`] ○ `mul` ○ None of the above

Solution: The arithmetic operation we are performing is an **and** operation. The addition in `PC + 4` is handled by the adder in the IF stage.

Q5.5 (1 point) MemRW

[✓ Memory read]

Memory write

None of the above

Solution: Since we are not writing to memory, we should set MemRW to memory read.

Q5.6 (1 point) WBSel

`0`

`1`

[✓ `2`]

None of the above

Solution: Since we want to write PC + 4 into the RegFile, WBSel should be set to 2.

Q5.7 (1 point) RegWEn

Register write

Register read

None of the Above

Solution: This question was dropped since Register write may have implied that we could not read from the register file.

Given a new instruction **`lieqi`** with the following description:

`lieqi rd rs1 rs2 imm`

```
if (rs1 == rs2) {
    rd = imm
}
```

For Q5.8 – Q5.9, write a simplified Boolean expression that describes the given control logic signal for **`lieqi`**. Your answer may consist of the following:

NOT	AND	OR	Constants	Parentheses	Variables
`!`	`*`	`+`	`0, 1`	`()`	Any control logic input or output signal

Q5.8 (1.5 points) PCSel = 0

Solution: Since we are not branching or jumping elsewhere in code, we can set PCSel to 0 to set PC = PC + 4.

Q5.9 (1.5 points) RegWEn = BrEq

Solution: RegWEn should be set to 1 if we want to write to rd, and 0 otherwise. Since we know that we only execute the body of the if statement if rs1 and rs2 are equivalent, we can just set RegWEn directly to be the value of the BrEq signal that's outputted from the Branch Comparator.

Q5.10 (4 points) What modifications must be made to the single cycle datapath with as few changes as possible? Select all that apply.

- [✓] Create a new instruction type and update the ImmGen.
- ☐ Add a new read input to the RegFile for a third register value.
- ☐ Add a new WriteData and WriteIndex input to the RegFile.
- ☐ Add a new input to the AMux and update any relevant selector/control logic.
- ☐ Add a new input to the BMux and update any relevant selector/control logic.
- ☐ Add a third input into the ALU and update any relevant selector/control logic.
- ☐ Allow the ALU to send out more than 1 output and update any relevant selector/control logic.
- ☐ Add a new ALU operation and update any relevant selector/control logic.
- ☐ Allow the DMEM to be able to read and write at the same clock cycle and update any relevant selector/control logic.
- ☐ Add a new WBMux and update any selector/control logic.
- ☐ None of the above

Solution: Currently, there are no instructions that support the ability to have an `rd`, `rs1`, `rs2`, and `imm`. As a result, we'll need a new instruction format and immediate generator. Everything else in the datapath can remain the same. We can use the branch comparator to determine if `rs1` == `rs2` and the ALU already supports an operation to output just the B input.

Q6 *Hazardous Waters* **(5 points)**

For the entirety of this question, assume that we are using the 5-stage pipelined datapath with double pumping and **no** forwarding.

Q6.1 (3 points) Rearrange the following instructions to minimize the number of stalls while maintaining the same behavior.

```
lw t0 0(s0) # Instruction A
sw t0 0(s1) # Instruction B
lw t1 4(s0) # Instruction C
sw t1 4(s1) # Instruction D
lw t2 8(s0) # Instruction E
sw t2 8(s1) # Instruction F
```

Format your answer as a comma-separated list. For example, the instruction order in the code block above would be described by "A, B, C, D, E, F".

A,C,E,B,D,F

Solution: There are three data hazards here: Instruction B depends on A, D depends on C, and F depends on E. Since you want at least two instructions in between each dependency to avoid a data hazard, there were six total possible solutions. The load instructions must come before the corresponding store instruction, and each dependency must be separated by exactly two instructions to remove all the data hazards.

Q6.2 (2 points) Consider a new instruction, `jellyfish`, that writes the output of the ALU to the RegFile during the EX stage. You may assume that the 5-stage pipelined datapath has been modified to implement this instruction without resolving any potential hazards.

Which hazard would the `jellyfish` instruction cause in the following program?

```
addi s0 x0 1
addi t4 x0 5
jellyfish t0 t1 t2
```

[✓ Structural] Control
Data None of the above

Solution: Since `jellyfish` writes the output of the ALU to the RegFile during the EX stage, we run into a structural hazard when considering any other instruction that writes to the RegFile during the WB stage. In the code snippet provided, the `addi` instruction in line 1 would enter the WB stage when the `jellyfish` instruction in line 3 enters the EX stage. Since we only have one write port and index available, we need to add more hardware to support two writes simultaneously.

Q7 *Person-Level Parallelism* (6 points)

Riley's emotions need help to keep everything running and complete the following tasks:

Task Number	**Task Name**	**Time (hours)**	**Prerequisites**
0	Analyze new feelings	3	-
1	Build headquarters	4	-
2	Create memories	2	0
3	Direct emotions	5	1
4	Express joy	4	2, 3
5	Form an identity	2	1, 2
6	Grow friendships	3	4, 5
7	Highlight happy moments	3	6

Q7.1 (2 points) What is the minimum time it takes for Joy to do all the tasks, assuming she can only work on one task at a time?

26 hours

Solution: Since there's only one emotion (process) working, we add up the time to perform all the tasks in serial.

Joy needs some help, so she asks Nostalgia to help her.

For Q7.2 – Q7.3, assume both Joy and Nostalgia are working on the tasks, each can work on their own task simultaneously, and each can work on one task at a time.

Q7.2 (2 points) What is the minimum time it takes for Joy and Nostalgia to complete all the tasks?

19 hours

Solution: Joy: tasks 0, 2, 5, 4 (once task 3 has been completed), 6, then 7

Nostalgia: tasks 1 then 3

There are alternate solutions. The recognition here is that we are limited by the fact that task 7 is dependent on task 6, which is dependent on tasks 4 and 5 being completed. This limits the amount of parallelism we can do.

Q7.3 (2 points) Suppose Riley's emotions have a new task as follows:

Task Number	Task Name	Time (hours)	Prerequisites
8	Inspire belief	1	4

What is the minimum time it takes for Joy and Nostalgia to do all the tasks?

19 hours

Solution: Joy: tasks 0, 2, 5, 4 (once task 3 has been completed), 6, then 7

Nostalgia: tasks 1, 3, then 8 (after task 4 has been completed).

There are alternate solutions. After Joy completes task 4, Nostalgia can complete task 8 while Joy works on task 6. Parallelism!!

Q8 *(Delayed|Train)-Level Parallelism* **(14 points)**

BART trains often arrive much later than their expected arrival times. The `total_delay` function is defined as follows:

`total_delay`: Calculates the sum of all delays between a set of expected and actual arrival times.		
Arguments	`int32_t n`	The number of trains. You may assume `n` is a multiple of 4.
	`int32_t *expected`	An array of `n` expected arrival times.
	`int32_t *actual`	An array of `n` actual arrival times.
Return value	`int32_t`	The sum of all delays.

If a train arrives earlier or at its expected time, the delay should be 0. You may also assume that a train expected to arrive at `expected[i]` actually arrives at `actual[i]`.

For example, given `n = 4` and the following `expected` and `actual` arrays, the return value should be `5 + 3 + 0 + 0 = 8`:

```
expected:  [10, 20, 30, 40]
  actual:  [15, 23, 29, 40]
   delay:  [ 5,  3,  0,  0]
```

You may use the following SIMD operations, with a maximum of one SIMD operation per blank. A `vector` is a 128-bit vector register capable of holding 4 32-bit signed integers.

- `vector vec_load(int32_t *A)`: Loads 4 integers at memory address `A` into a vector
- `vector vec_setnum(int32_t num)`: Creates a vector where every element is equal to `num`
- `vector vec_cmpgt(vector A, vector B)`: Computes whether each element in `A` is greater than the corresponding element in `B`, and returns a new vector with the results (`0xFFFFFFFF` if true, `0` otherwise, for each element in the vector)
- `vector vec_and(vector A, vector B)`: Computes the bitwise **and** between each pair of corresponding vector elements in `A` and `B`, and returns a new vector with the result
- `vector vec_or(vector A, vector B)`: Computes the bitwise **or** between each pair of corresponding vector elements in `A` and `B`, and returns a new vector with the result
- `vector vec_add(vector A, vector B)`: Adds `A` and `B` together elementwise, and returns a new vector with the result
- `vector vec_mul(vector A, vector B)`: Multiplies `A` and `B` together elementwise, and returns a new vector with the result
- `int32_t vec_sum(vector A)`: Adds all elements of the vector together, and returns the sum

Implement `total_delay` to match the described behavior using SIMD and OpenMP.

```
 1 int32_t total_delay(int32_t n, int32_t *expected, int32_t *actual) {
 2   int32_t result =  0  ;
       Q8.1           Q8.2
 3   vector zero_vec = vec_setnum(0);
 4   vector negate_vec = vec_setnum(-1);
                              Q8.3
 5   #pragma omp parallel
                Q8.4
 6   {
 7     vector result_vec = vec_setnum(0);
 8     #pragma omp for
                 Q8.5
 9     for (int32_t i =  0  ; i < n; i += 4) {
                        Q8.6   Q8.7
10       vector expected_vec = vec_load(expected + i);
                                             Q8.8
11       vector actual_vec = vec_load(actual + i);
                                          Q8.9
12       vector temp_vec = vec_mul(expected_vec, negate_vec);
                                          Q8.10
13       temp_vec = vec_add(actual_vec, temp_vec);
                                Q8.11
14       vector compared_vec = vec_cmpgt(temp_vec, zero_vec);
                                                     Q8.12
15       temp_vec = vec_and(temp_vec, compared_vec);
                                 Q8.13
16       result_vec = vec_add(result_vec, temp_vec);
                            Q8.14
17     }
18     #pragma omp critical
               Q8.15
19     result += vec_sum(result_vec);
                    Q8.16
20   }
21   return result;
22 }
```

Solution: Since we know that **n** is a multiple of four, there's no need for a tail case. Since we don't have a `vec_sub` instruction, we need to negate the expected values by creating a vector of all -1s and multiplying that to the `expected_vec` and add that to the actual values in order to find the delay. Once we have the delay in the `temp_vec`, we can remove all the negative numbers in the delay (representing trains that arrive earlier than expected) by comparing `temp_vec` to `zero_vec` to make a mask, then **and** together the mask with the delays to yield a vector with just the delays, and 0 otherwise.

Lastly, we need to avoid data races by adding a compiler directive when adding to the shared variable result. An alternate solution would be to use the reduction keyword in line 5 `#pragma omp parallel reduction(+:result)`. Using the reduction keyword in line 8 was penalized, as the `result` variable was out of the scope.

(Question 8 continued…)

Q9 *Virtual Memory* **(12 points)**

Q9.1 (1.5 points) Suppose we have 16 MiB of virtual memory, 32 KiB of physical memory, and 64B pages. How many bits are our page offset, virtual page number (VPN), and physical page number (PPN)?

VPN: 18	PPN: 9	Offset: 6

Solution: Our virtual memory space is 16 MiB, or $2^4 * 2^{20} = 2^{24}$ bytes. You need $\log_2(2^{24})$ bits to address the whole virtual memory space, meaning that your virtual address size is 24 bits. Apply similar logic to the physical memory to obtain 15 physical address bits. Lastly, since we have 64B pages, we need $\log_2 64 = 6$ page offset bits.

With 24 bits for the virtual address and 6 bits for the page offset, that leaves us with 24 - 6 = 18 VPN bits. Similarly, with 15 bits for the physical address and 6 bits for the page offset, that leaves us with 15 - 6 = 9 PPN bits.

For Q9.2 – Q9.6, assume we have 8-bit page offsets, 24-bit VPNs, and 8-bit PPNs. We have a 4 entry, fully associative TLB with LRU replacement policy and one free physical page with PPN `0x62`.

The current state of the TLB and the first 6 entries of the page table are given below.

TLB

Valid	VPN	PPN
0	0x00 0004	0x28
1	0x00 0001	0x15
1	0x00 0005	0x90
0	0x00 0002	0x46

Page table

0x3040CA80
0xA7708515
0x63A01986
0xBA025068
0xBF586042
0xD5762290
...

Each page table entry (PTE) is 32 bits:

31	30 8	7 0
Valid	Other status bits	PPN

For each of the following virtual addresses, translate it to its corresponding physical address, and answer whether it will result in a TLB hit, TLB miss and page table hit, or Page fault. Assume each access occurs independently, not sequentially.

Q9.2 (1.5 points) `0x0000 0280`

0x6280

○ TLB hit

○ TLB miss and page table hit

[✓ Page fault]

Solution: Since we have 24-bit VPNs, the first 6 hexdigits of the VA represent the VPN, and the remaining 2 hexdigits represent the page offset. Looking for 0x00 0002 in the TLB, it is present in the TLB, but the Valid bit is set to 0. Therefore, we get a TLB miss.

Going into the page table, we look in the 2nd index (0-indexing) of the page table to find that the valid bit of that entry is 0 (since 0x6 translates to 0b0101, and the MSB represents the valid bit). Therefore, we get a page fault.

We use our free physical page of 0x62 and concatenate that with the page offset of 0x80 to yield the physical address 0x6280.

Q9.3 (1.5 points) `0x0000 03AF`

0x68AF

○ TLB hit
[✓ TLB miss and page table hit]
○ Page fault

Solution: The VPN 0x00 0003 is not in the TLB. TLB Miss. However, the third index of the page table has a valid mapping (since 0xB translates to 0b1011). Since the PPN is the lower 8 bits, we have a page table hit with a PPN of 0x68. This yields our answer 0x68AF.

Q9.4 (1.5 points) `0x0000 0512`

0x9012

[✓ TLB hit]
○ TLB miss and page table hit
○ Page fault

Solution: The VPN 0x00 0005 is in the TLB, and the valid bit is on. TLB Hit! We can retrieve the mapping to yield 0x90 and concatenate that with the page offset.

Q9.5 (1.5 points) `0x0000 0484`

0x4284

○ TLB hit
[✓ TLB miss and page table hit]
○ Page fault

Solution: The VPN 0x00 0004 is in the TLB, but the valid bit is off. However, the page table contains a valid mapping, so we get a page table hit.

Q9.6 (1.5 points) `0x0000 006A`

0x626A

○ TLB hit
○ TLB miss and page table hit
[✓ Page fault]

Solution: The VPN 0x00 0000 is not in the TLB. The page table does not contain a valid mapping, so we get a page fault. We use the free physical page 0x62.

For Q9.7 – Q9.8, assume we have a 2-level page table with 16-bit page offsets, 16-bit VPNs, 8-bit PPNs, and no TLB.

Assume that we divide the VPN bits equally between the L1 and L2 page tables, and the L1 VPN bits are the most significant bits.

Q9.7 (1 point) For the address `0xF944 0620`, what are the L1 VPN and L2 VPN?

L1: `0xF9`

L2: `0x44`

Solution: Since we know that the address is 32 bits long, and 16 bits are dedicated towards the page offset, that leaves us with 16 bits for the L1 and L2 VPNs. The layout of the VA concatenates the L1 bits, L2 bits, and page offset bits in that order. Split evenly, this means that the L1 bits are the first two hexdigits, the L2 bits are the next two hexdigits, and the last four hexdigits are the page offset. This yields our final solution of 0xF9 for the L1 VPN bits and 0x44 for the L2 VPN bits.

Q9.8 (2 points) How many different L2 page tables are accessed in total over the following sequence of memory accesses?

- `0xFFFF F100`
- `0xFFFF F000`
- `0x1000 8000`
- `0xFFFC 6500`
- `0x4000 8000`
- `0x100A B608`
- `0x10FD 0904`

3

Solution: The L1 VPN bits tells us which L2 page table we should look at. Therefore, we count all the unique sequences of L1 VPN bits exist in these accesses, yielding `0xFF`, `0x10`, and `0x40`.

Q10 *Caches* (9 points)

Q10.1 (1.5 points) What is the T:I:O breakdown for a 128B direct mapped cache with a block size of 16B on a 16-bit system?

T: 9 bits	I: 3 bits	O: 4 bits

Solution: We need 4 offset bits since our block size is 16B. To address all 16 bytes, we take $log_2(16)$ to yield 4 offset bits.

Since our cache is 128 bytes, and each block is 16 bytes, we will have a total of 8 blocks in our cache. In order to index into 8 blocks, we need $log_2(8)$ = 3 index bits.

Since we're working with a 16-bit system, this leaves us with 16 - 4 - 3 = 9 tag bits.

For Q10.2 – Q10.4, assume we are using a 2-way set-associative cache with an MRU replacement policy, and each address has 11 tag bits, 2 index bits, and 3 offset bits.

```
#define ARR_SIZE 100
int32_t arr[ARR_SIZE]; // arr starts at address 0x2020
for (register int32_t i = 0; i < ARR_SIZE - 5; i++) {
  arr[i] = arr[i] + 0x6;
  arr[i + 2] = arr[i] + 0x1;
  arr[i + 4] = arr[i] + 0xC;
}
```

Q10.2 (1.5 points) How many memory accesses are there in each iteration of the `for` loop?

6

Solution: Each array access goes into memory, for a total of 6 accesses.

Q10.3 (2 points) How many cache hits are there in the first iteration of the `for` loop (`i = 0`)?

3

Solution: With 3 offset bits, we know each block can hold 8 bytes (or 64 bits). Therefore, each block can hold two 32-bit integers.

After writing out the address of each memory access (keeping in mind that `arr` starts at address 0x2020), we find that the first access to `arr[0]` is a compulsory miss, as our cache starts off cold. Due to spatial locality, we also end up bringing in `arr[1]`. Every subsequent access to `arr[0]` is a cache hit (and there are three of them). However, `arr[2]` and `arr[4]` both result in a cache miss, since only `arr[0]` and `arr[1]` are in our cache at that time. This ends up bringing in `arr[3]` and `arr[5]` as well.

For the first iteration of the for loop, we get 3 cache hits.

Q10.4 (2 points) How many cache hits are there in the third iteration of the `for` loop (`i = 2`)?

5

Solution: Read the solution to the previous part if you're confused!

The first iteration of the for loop brings in the 0th through the 5th elements of `arr` into the cache.

The second iteration of the for loop has six cache hits, since all the array accesses are already inside of the cache.

The third iteration of the for loop tries to access `arr[6]` and finds that it is not in the cache, which is the only cache miss.

There are no cache replacements necessary at any point in the first three iterations of the program.

Q10.5 (2 points) For a fully associative cache, what is the maximum block size for which the cache would always miss for the following program? Assume that the block size must be a power of 2.

```c
#define ARR_SIZE 100
int32_t arr[ARR_SIZE]; // arr starts at address 0x8000
register int32_t sum = 0;
for (register int32_t i = 0; i < ARR_SIZE; i += 32) {
    sum += arr[i];
}
```

128	bytes

Solution: We see that the for loop is incrementing **i** by 32 every iteration. Therefore, in order to guarantee that we will always miss, we need the block size of the cache to be exactly the size of 32 integers (e.g. if we access **arr[0]**, we can fit at most **arr[0]** until **arr[31]** in one block to guarantee a miss when we access **arr[32]** in the next iteration). Here, we see that the integers are 32 bits each (or 4 bytes). Therefore, the maximum block size to guarantee that the cache would always miss is 32 * 4 = 128 bytes.

Q11 *The Finish Line* (0 points)

These questions will not be assigned credit; feel free to leave them blank.

Q11.1 (0 points) For each rating on a scale of 1 (worst) to 6 (best), list a nearby restaurant that matches the rating.

1:	2:
3:	4:
5:	6: Momo Masalas

Q11.2 (0 points) If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.

Solution: ¯_(ツ)_/¯

Unless otherwise specified, the following instructions apply to the entire exam:

- You may assume the following:
  - All provided code compiles successfully and runs without errors
  - All necessary standard libraries are imported
  - Dynamic memory allocation never fails
  - Caches and TLB start off cold
  - The single-cycle and the 5-stage pipelined datapaths are referring to the datapaths on the CS 61C Reference Card
  - No compiler optimizations are performed
  - All expressions follow C operator precedence
- Answer Formats
  - If in hexadecimal, use only capitalized letters (`0xDEADBEEF` instead of `0xdeadbeef`)
  - If the answer is in binary or hexadecimal, include prefixes and do not truncate leading 0's
  - If answer is not in binary or hexadecimal, do not add any prefixes or suffixes
  - Answers must be in the simplest form
  - If no base is specified and the answer is a number or mathematical expression, the answer must be in base 10
- Coding Questions
  - Blanks may be left empty, but do not use more blanks than provided
  - Each blank may contain at most one statement or instruction
  - Programs must be as memory efficient as possible
  - In C, curly braces must be included for `if`/`for`/`while`/`do-while`/`switch` statements, and a block's body may not be on the same line as the opening or closing curly brace
    - For example, an `if` statement requires at least 3 lines, and an `if-else` statement requires at least 5 lines
  - In RISC-V, programs must follow proper calling convention