Garcia, Nikolic CS 61C Midterm
Fall 2020 Machine Structures

Note (August 10, 2022): These are extremely rough drafts of rewritten solutions. They definitely contain errors and unfinished sections, but might have some useful parts for studying.

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

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 ??? minutes. There are 5 questions of varying credit, for a total of 46 points.
For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option
- ● Only one selected option

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

- ■ You can select
- ■ multiple squares

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

Question 1 ***Floating Point*** **(5 points)**

FOR THIS ENTIRE SECTION, YOU ARE NOT ALLOWED TO USE THE C PROGRAMMING LANGUAGE, OR A CALCULATOR, OR A FLOATING POINT SIMULATOR.

Consider a 21-bit floating-point number with the following components (1 sign bit, 11 exponent bits, 9 mantissa bits); i.e.,

SEEEEEEEEEEEMMMMMMMMM

All other properties of IEEE754 apply (bias, denormalized numbers, ∞, NaNs, etc). The bias is the usual $-(2^{E-1}-1)$, which here would be $-(2^{10}-1) = -1023$.

Q1.1 (3 points) What is the bit representation (in hex) of the floating-point number -4.75? **Do NOT include the 0x prefix when writing your answer.**

Solution: `180380`

First write the number in binary: $-4.75 = -$`0b`0100.11

Then write the number in scientific notation by moving the binary point so that only the implicit 1 is on the left of the binary point: $-$`0b`1.0011×2^2

Sign bit: `0b1` (number is negative)

Exponent is 2. Subtract the bias to get $2-(-1023) = 1025$. Write 1025 in 11-bit unsigned binary to get `0b1000 0000 001`

Mantissa bits are `0b1100 0000 0`

Combine the bits together to get `0b1 1000 0000 0011 1000 0000`

Convert to hex: `0x180380`

Q1.2 (2 points) How many floats are there in the range $[8, 12)$, i.e., $8 \leq n < 12$?

You can either simplify your answer or leave it in exponent form. If you leave it in exponent form, you must use `**` for exponentiation. For example, if your answer is 8 (ie 2^3), you can either write it as 8 or 2^3.

Solution: 256

First, convert $8 =$ `0b`1×2^3 to floating-point. The bias is encoded as $3-(-1023) = 1026$. `0b0 1000 0000 010 0000 0000 0`.

Then convert $12 =$ `0b`1.1×2^3 to floating-point: `0b0 1000 0000 010 1000 0000 0`.

In between 8 and 12, the sign bit cannot change, and the exponent bit cannot change. However, we can change the mantissa bits to be anything between `0b0000 0000 0` and `0b1000 0000 0`. There are `0b1000 0000 0` $= 2^8 = 256$ numbers in this range.

Question 2 ***Quest Clobber*** **(10 points)**

Q2.1 (3 points) Help! We have two robots, Alexa and Siri, but they're speaking the wrong languages! Alexa sends sensor data with the bits as `uint8_t`s encoding biased notation (with a bias of −127), which would later be read by Siri. However, Siri can only understand sign-magnitude. Your job is to write a simple function `bias2sm` that converts biased notation (with a bias of −127) to sign-magnitude so that Siri can correctly convert messages received from Alexa. (E.g., if Alexa saw −3 and encoded it as biased notation (with a bias of −127) bits, you'd need to return a new set of bits so that Siri would say "oh, those bits mean −3"). Note that the range of the sensor is such that it would never produce a number that Siri couldn't handle. Also, if you're ever given the choice between storing a +0 or -0, store it as a +0. If you are not able to complete this part, you can still receive full credit on the next two subparts.

(*This subpart is independent from the next two subparts*)

```
1 uint8_t bias2sm(uint8_t bias)
2 {
3   return bias + 1; // <-- replace this with your converter
4 }
```

Solution:

```
1 uint8_t bias2sm(uint8_t bias) {
2     return (bias >= 0x7F) ? bias - 127 : 255 - bias;
3 }
```

Note that in biased notation, 0 is encoded as $0 - (-127) = 127$ in unsigned notation, which is `0b0111 1111` = `0x7F`. Therefore, any biased encoding less than `0x7F` corresponds to a negative biased number, and any biased encoding greater than `0x7F` corresponds to a positive biased number.

If the number is positive, we can convert the biased encoding back to the number by adding the bias (subtracting 127). Now we need to set the most-significant bit of this number to 0 (sign is positive). Note that this number will always have 0 as the most-significant bit: if we take an 8-bit bias encoding between 127 (`0x7F`) and 255 (`0xFF`) and subtract 127, we'll get an 8-bit value between 0 (`0x00`) and 128 (`0x80`). All values in this range already have the most-significant bit as 0, except 128, but 128 is not representable by 8-bit sign-magnitude, so we can ignore it.

In summary, when the number is positive (`bias >= 0x7F`), we subtract 127 from the number (`bias - 127`), and the sign bit is already set to 0 for us.

Sign bit has a value of 128 (for 8-bits, MSB = 1, `0x80` = 128 in unsigned representation) Then, for negative numbers, we need to get their absolute values (since since magnitude is unsigned). We can simply subtract from 0 if the corresponding biased number is negative (if `bias -127` is negative). Thus, 0 - (`bias - 127`) is the absolute value of whatever the biased representation is. Now, add 128 to set the sign bit to 1 (since the biased number is negative in the first place). 128 + 0 - (`bias - 127`) = 128 -bias + 127 = `255 - bias`.

Q2.2 (4 points) After much deliberation, the robots have agreed to use unsigned numbers instead. They have stored some data in a binary node structure, but have realized that all their data is one less than the correct value. Complete the code for `TREE incr_tree(TREE *p)` that returns a duplicate tree in which every number has been incremented by 1. You must use `CS61C_malloc()` instead of `malloc()`. You can assume that every call to `CS61C_malloc()` succeeds.

```
1 typedef struct node {
2     struct node *L;
3     struct node *R;
4     uint8_t N;
5 } TREE;
```

```
1 TREE *incr_tree(TREE *p)
2 {
3   return p; // <-- replace this with your code
4 }
```

Solution:

```
1  TREE *incr_tree(TREE *p) {
2      if (!p) {
3          return NULL;
4      }
5      TREE *q = (TREE *) CS61C_malloc (sizeof(TREE));
6      q->N = p->N + 1;
7      q->L = incr_tree(p->L);
8      q->R = incr_tree(p->R);
9      return q;
10 }
```

At every node of the original tree, we allocate space for the corresponding node in the duplicate tree (line 5) and assign the value in the duplicate tree (line 6).

Then, we recursively call the function on the left and right subtrees of the current node (lines 7-8).

The function should return a pointer to the newly-allocated node (line 9) so that when a parent node calls `incr_tree` on a child subtree, we can store a pointer to the new child subtree in the parent node (lines 7-8).

Base case: if the current subtree is NULL, then return NULL (lines 2-4).

Q2.3 (3 points) Clean up behind yourself! Write a function void `free_tree(TREE *p)` which will free all of the space used by the input tree `p`. You may assume that all of the nodes in `p` were malloc'd properly. You must use `CS61C_free()` instead of `free()`.

```
1 void free_tree(TREE *p)
2 {
3   return; // <-- replace this with your code
4 }
```

Solution:

```
1 void free_tree(TREE *p)
2 {
3   if (p) {
4         free_tree(p->L);
5         free_tree(p->R);
6         CS61C_free(p);
7     }
8 }
```

Recursively free the left and right subtrees of the current node, then free the current node. (We have to free the subtrees before freeing the current node; if we freed the current node first, we'd lose the pointers to the subtrees.)

Base case: If the current subtree is NULL, do nothing. The if statement at line 3 checks for the base case.

Question 3 ***RISC-V Assembly*** **(10 points)**

Q3.1 (2 points) What is the machine code (in hex) of `lb s2 5(s5)`. Do NOT prefix your solution with `0x`. Please pad your answer to a full 4 bytes when submitting if necessary.

Solution: `0x005A8903`
Opcode: `0b000 0011`
Funct3: `0b000`
rd: `s2` is `x18` = `0b10010`
rs1: `s5` is `x21` = `0b10101`
Immediate: 5 is `0b0000 0000 0101`
I-type instruction encodes immediate, then rs1, then funct3, then rd, then opcode:
`0b0000 0000 0101 10101 000 10010 000 0011`
Grouping bits of 4:
`0b0000 0000 0101 1010 1000 1001 0000 0011`
Converting to hex: `0x005A8903`

Q3.2 (8 points) Write a function in RISC-V that takes a string of only letters (uppercase and lowercase) terminated appropriately and lowercases it, returning the length of the string; call it `LowerLen`.

For example (in C notation):

```
1 S = "Cal"; LowerLen(S) => 3 // S is now "cal"; "Cal" had 3 letters
```

You will probably find an ASCII table helpful. You are allowed to use Venus to debug your code, feel free to test it on the string above to see if it works. We strongly encourage you to test your code on other strings. We have also provided a testing framework, which runs your code on the string `"Cal"`. To test different strings, you can modify the data value `"InputString"`. There are also comments in the testing framework in case you want to add more functionality to your code.

Note that this does not require a large amount of code, our staff solution was less than 15 lines of RISC-V code.

As a reminder, you are required to follow standard RISC-V calling convention. You should expect your input string from register `a0`, and output the length of the string in register `a0`.

```
.data

InputString:  .asciiz "Cal"

BeforeString:  .asciiz "Input should be 'Cal', and it is: "

AfterString:  .asciiz "After calling, it should be 'cal'. It is: "

ReturnValueString: .asciiz "Return Value should be 3. It is: "

.text

main:
#Feel free to edit this for your own tests, but this part will not be graded
#Print String before running code
la a0 BeforeString
jal print_str
la a0 InputString
jal print_str
jal print_newline
#Run LowerLen
la a0 InputString
jal LowerLen
mv s0 a0
#Print String after running code
la a0 AfterString
jal print_str
la a0 InputString
jal print_str
jal print_newline
#Print return value
la a0 ReturnValueString
jal print_str
mv a0 s0
jal print_int
jal print_newline
#Exits the program
li a0 10
ecall



LowerLen:
#Reads through the string, and converts the entire string to lowercase
#Returns the length of the string
#YOUR CODE HERE
li a0 0
jr ra
```

```
##########################################
#Utility Functions
##########################################
print_int:
    mv a1, a0
    li a0, 1
    ecall
    jr   ra

print_str:
    mv a1, a0
    li a0, 4
    ecall
    jr   ra

print_newline:
    li a1, '\n'
    li a0, 11
    ecall
    jr   ra
```

Solution:

```
1  addi t0 x0 0
2  Loop:
3      lb t1, 0(a0)
4      beq t1, x0, End
5      addi t0, t0, 1
6      ori t1, t1, 0b0010 0000
7      sb t1, 0(a0)
8      addi a0, a0, 1
9      j Loop
10 End:
11     mv a0, t0
12     jr ra
```

`t0` will be used to hold the length of the string, which starts at 0 (line 1). On every iteration of the loop, we'll increment `t0` to count one more character (line 5).

At the start of the function, `a0` holds the argument to the function, which is the address of the start of the string. On each iteration of the loop, we'll load one character of the string into `t1` (line 3), and increment `a0` to point to the next character of the string (line 8).

When the byte we load is the null terminator, we've reached the end of the string. At this point, we don't want to increment the length counter or convert a character to lowercase, so we should leave the loop before doing those things (line 4).

To convert a character to lowercase, we use the fact that the capital letter and lowercase letter differ exactly by one bit; for example, `a = 0x61 = 0b0110 0001`, and `A = 0x41 = 0b0100 0001`. In other words, to force a letter to be lowercase, we need to set this bit to `0b1`.

Recall that ORing any bit with 1 always produces 1, and ORing any bit with 0 leaves the bit unchanged, so we can set the relevant bit (third from the left) to 1 and leave all other bits unchanged by ORing the byte with `0b0010 0000` (line 7).

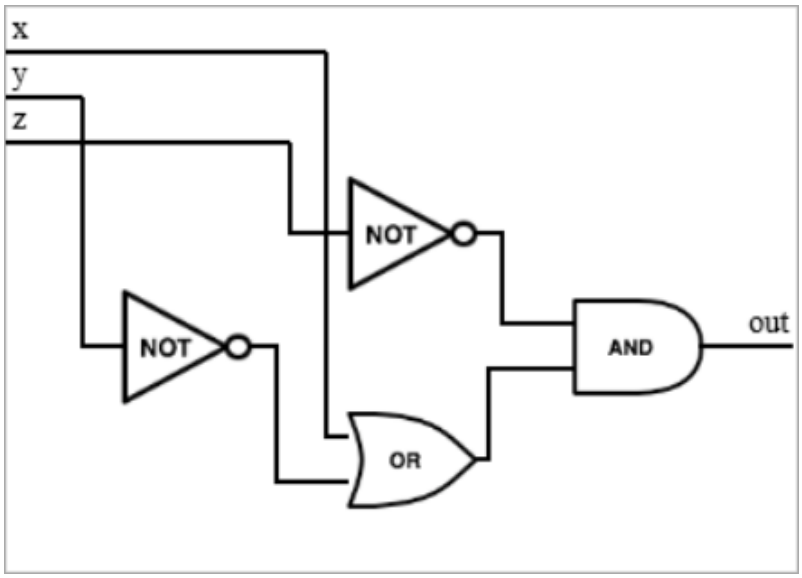
Once we've converted the character to lowercase, we store back into the same address in memory (line 7). We have to use store-byte and not store-word because we only want to modify one byte in memory, not 4 bytes (1 word).

At the end of the function, we need to put the length of the string into `a0` since it's the return value (line 11). Then we can return control back to the caller function (line 12).

Question 4 ***SDS*** **(12 points)**

FOR THIS ENTIRE SECTION, YOU ARE NOT ALLOWED TO USE LOGISIM.

Q4.1 (4 points) Fill out the truth table for the circuit below. For wires that intersect, assume the signal follows a straight path until the wire turns to feed into a logic gate's input.



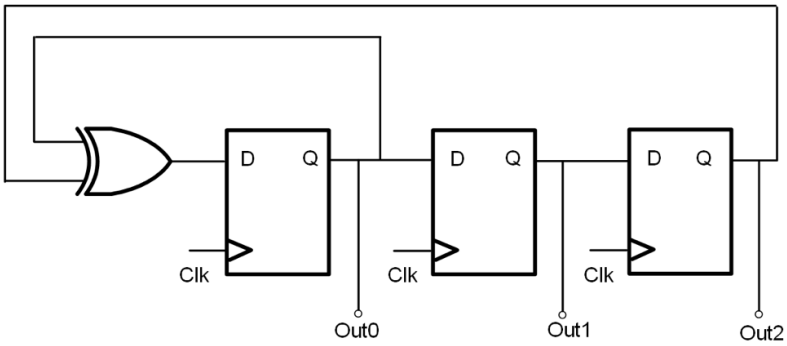
x	y	z	out
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Solution:

x	y	z	out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Q4.2 (3 points) The logic implementation of a state machine is shown in the figure below. How many reachable states does this state machine have?

(Assume that it always starts from `Out0=0`, `Out1=0`, `Out2=1`)



Solution: 7

At each time step, the left-most register (`Out0`) gets updated to be the value of `Out0 XOR Out2`. Also, `Out1` gets updated to be the value of `Out0`, and `Out2` gets updated to be the value of `Out1`. We can run this state machine from the starting state and see how many states we encounter until we hit a cycle.

Starting state: 0, 0, 1

`0 XOR 1 = 1`, so the next state is 1, 0, 0

`1 XOR 0 = 1`, so the next state is 1, 1, 0

`1 XOR 0 = 1`, so the next state is 1, 1, 1

`1 XOR 1 = 0`, so the next state is 0, 1, 1

`0 XOR 1 = 1`, so the next state is 1, 0, 1

`1 XOR 1 = 0`, so the next state is 0, 1, 0

`0 XOR 0 = 0`, so the next state is 0, 0, 1

That's 7 cycles before we start repeating the pattern.

Q4.3 (3 points) In the figure above, flip-flop clk-to-q delay is 50ps. Setup time is 30ps. XOR delay is 20ps. What is the minimum cycle time of operation?

Solution: 100ps

The longest combinatorial logic path between two clocked elements (registers) goes from the output of a register (either `Out0` or `Out2`), through the XOR gate, and to the input of `Out0`.

When a rising edge happens, we have to first wait the clk-to-q delay for the signal to appear at the output of a register. Then we have to wait for the signal to travel through the XOR gate. Then once the signal arrives at the register input, we have to wait the setup time so the signal stabilizes before the next rising edge.

clk-to-q delay + XOR delay + setup time = $50 + 20 + 30 = 100$ ps

Q4.4 (2 points) In the above figure, what is the longest hold time for the flip-flop that allows for correct operation?

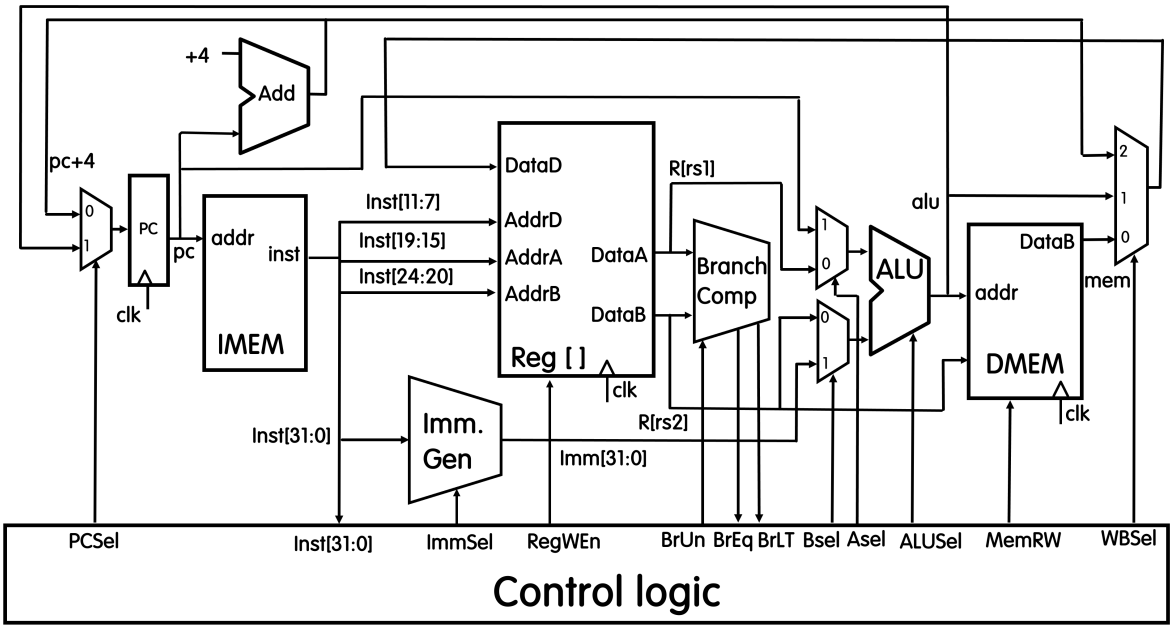
Solution: 50ps

The shortest combinatorial path between two clocked elements (registers) goes through no logic gates (e.g. from the output of the left-most register directly to the input of the middle register).

When a rising edge happens, inputs to registers start changing after the clk-to-q time (i.e. 50ps after the rising edge). Therefore, our hold time (time after the rising edge when the input to registers must stay stable) can only be as long as 50ps.

Question 5 ***RISC-V Datapath, Control, and Pipelining*** **(9 points)**

The datapath below implements the RV32I instruction set.



Q5.1 (3 points) In the RISC-V datapath above, mark what is used by a lb instruction.

Select one choice for each of the following:

PCSel Mux:

- ● **pc + 4 branch**
- ○ ALU branch
- ○ Input dependent
- ○ * (don't care)

ASel Mux:

- ○ pc branch
- ○ * (don't care)
- ● **Reg[rs1] branch**

BSel Mux:

- ○ * (don't care)
- ○ Reg[rs2] branch
- ● **imm branch**

WBSel Mux:

- ● **mem branch**
- ○ * (don't care)
- ○ pc + 4 branch
- ○ ALU branch

Select all that apply for the following:

Datapath Units:

- ■ **Load Extend**
- □ Branch Comp
- ■ **Imm. Gen**

RegFile:

- ■ **Read Reg[rs1]**
- ■ **Write Reg[rd]**
- □ Read Reg[rs2]

Q5.2 (3 points) Specify whether the following proposed instructions can be implemented using this datapath without modifications. If the instruction can be implemented, specify an expression for the listed control signals, by following the example below. If the instruction is not implementable, write "No" in the implementable column and "N/A" in the Control Signals column.

Example:

Instruction	Description	Implementable?	Control Signals
Add `add rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`	Yes	ALUSel = Add WBSel = 1

Instruction	Description	Implementable?	Control Signals
Negate `neg rd, rs1`	`R[rd] = -R[rs1]`		ASel = BSel =
PC-relative load `lwpc rd, imm`	`R[rd] = M[PC + imm]`		ASel = BSel =

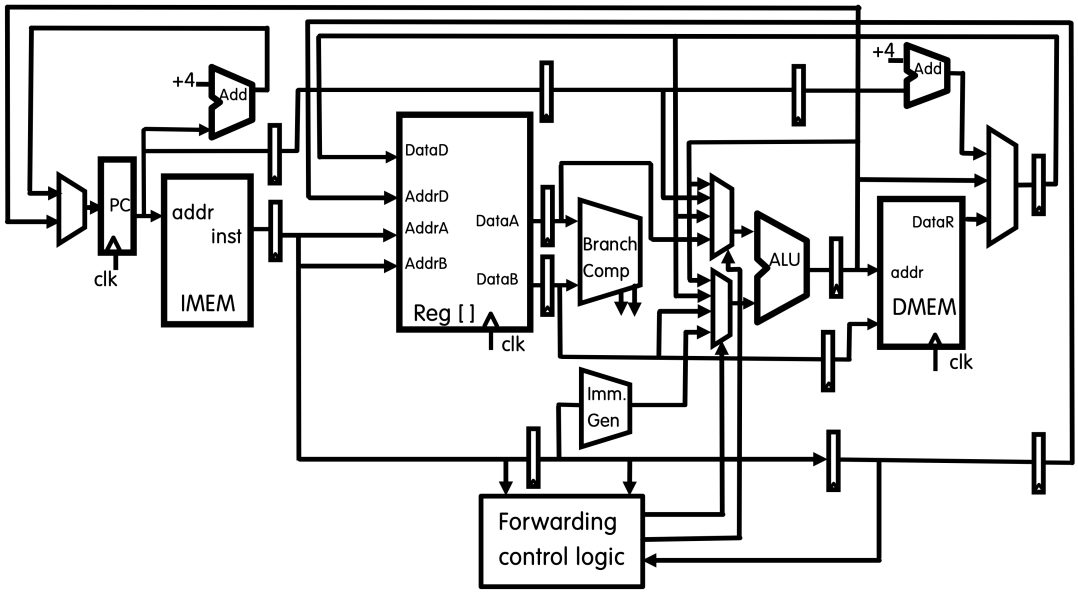
Solution:

Instruction	Description	Implementable?	Control Signals
Negate `neg rd, rs1`	`R[rd] = -R[rs1]`	Yes	ASel = reg BSel = imm
PC-relative load `lwpc rd, imm`	`R[rd] = M[PC + imm]`	Yes	ASel = pc BSel = imm

SOLUTION-TODO: Doesn't reg/imm for neg cause you to compute reg-imm? Don't you want to compute imm-reg to negate the number? Or do you need some two's complement thing

`lwpc` is the same as a normal `lw`, except we the memory address is computed as `PC + imm` instead of `R[rs1] + imm`. We can still use the ALU to compute the memory address (as in normal `lw`), but we send PC to the A input of the ALU instead of the value in `rs1`.

Q5.3 (3 points) Consider the 5-stage pipeline presented in lecture with combinational-read IMEM and DMEM and the forwarding paths as drawn below.



Write the number of NOPs needed between each instruction, and list the hazard that causes the stall. If no hazard occurs, select "None", and list the number of NOPs as 0. Assume you can write to and read from the same address in the register file (Reg [ ]) in the same cycle. If there is a branch, assume that it is not taken, and there is no branch prediction. Consider two cases:

Case 1: Forwarding is not implemented (the same pipeline above part A).

Case 2: The forwarding muxes in the diagram are driven correctly by the forwarding logic. Note: If a hazard is resolved by forwarding and no other hazard is present, select "None" for the hazard

Assume that the branch is not taken and the there is no branch prediction.

```
1         andi x2, x1, 0xF
2         bge x1, x2, label
3         xori x2, x2, 1
4 label:  or x3, x2, x1
```

Case 1, Instructions 1-2 Hazard:

- ○ Structural
- ● **Data**
- ○ Control
- ○ None

Solution: Instruction 1 writes to `x2`, and instruction 2 reads from `x2` before instruction 1 can write to it. This is a data hazard.

Case 1, Instructions 1-2, # of nops:

Solution: 2

The question says to assume that we can write to and read from the same register in the same cycle. Adding two stalls causes the register write (W) stage in instruction 1 to be in the same cycle as the register read (D) stage in instruction 2.

```
1     F D E M W
nop     F D E M W
nop       F D E M W
2           F D E M W
```

Case 1, Instructions 2-3 Hazard:

- ○ Structural
- ○ Data
- ● **Control**
- ○ None

Solution: Instruction 2 is a branch. We have to wait for instruction 2 to figure out whether a branch is taken or not before we can fetch the next instruction. This is a control hazard.

Case 1, Instructions 2-3, # of nops:

Solution: 2

The branch comparator computes whether we need to branch in the execute (E) stage. We need to wait for the cycle after this to fetch the next instruction.

Adding two stalls causes the start of instruction 2 (F) to be just after the execute stage of instruction 1 (E).

```
1     F D E M W
nop     F D E M W
nop       F D E M W
2           F D E M W
```

Case 1, Instructions 3-4 Hazard:

- ○ Structural
- ● **Data**
- ○ Control
- ○ None

Solution: Instruction 3 writes to `x2`. Instruction 4 reads from `x2`. This a data hazard.

Case 1, Instructions 3-4, # of nops:

Solution: 2
Same reasoning as the data hazard between instructions 1-2.

Case 2, Instructions 1-2 Hazard:

- ○ Structural
- ● **Data**
- ○ Control
- ○ None

Solution: Forwarding does not solve our data hazard, because the value in `x2` is needed in the branch comparator in instruction 2. Looking at the datapath diagram, the branch comparator can only get its value from the regfile, so we still need to wait for the value of `x2` to be written to the regfile before we can read from it.

Case 2, Instructions 1-2, # of nops:

Solution: 2
Forwarding didn't help, so we use the same reasoning as case 1.

Case 2, Instructions 2-3 Hazard:

- ○ Structural
- ○ Data
- ● **Control**
- ○ None

Solution: Forwarding doesn't solve control hazards.

Case 2, Instructions 2-3, # of nops:

Solution: Forwarding didn't help, so we use the same reasoning as case 1.

Case 2, Instructions 3-4 Hazard:

- ○ Structural
- ○ Data
- ○ Control
- ● **None**

Solution: Forwarding can remove this data hazard. The result of the XOR computation in line 3 can be sent directly to the ALU so that line 4 can use that value.

Case 2, Instructions 3-4, # of nops:

Solution: 0

Line 3 computes the value to be put in `x2` at the E stage. We forward this value to be used in the E stage of line 4, which is later in time, so no stalls are needed.

```
3  F D E M W
4    F D E M W
```