ECE550
Midterm

Name: NetID:

There are 6 questions, with the point values as shown below. You have 75 minutes with a total of 75 points. Pace yourself accordingly.

This exam must be individual work. You may not collaborate with your fellow students. However, you are permitted one page of notes.

I certify that the work shown on this exam is my own work, and that I have neither given nor received improper assistance of any form in the completion of this work.

Signature:

#	Question	Points Earned	Points Possible
1	Combinatorial Logic		15
2	Sequential Logic		10
3	FSMs		10
4	Asm Programming		20
5	Datapaths		10
6	Memory Hierarchy		10
	Total		75
	Percent		100

Question 1 Combinatorial Logic [15 pts]

Given the following truth-table:

a	b	c	x
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

1. Write the sum-of-product formula

2. Simplify the formula

3. Write VHDL that implements the formula

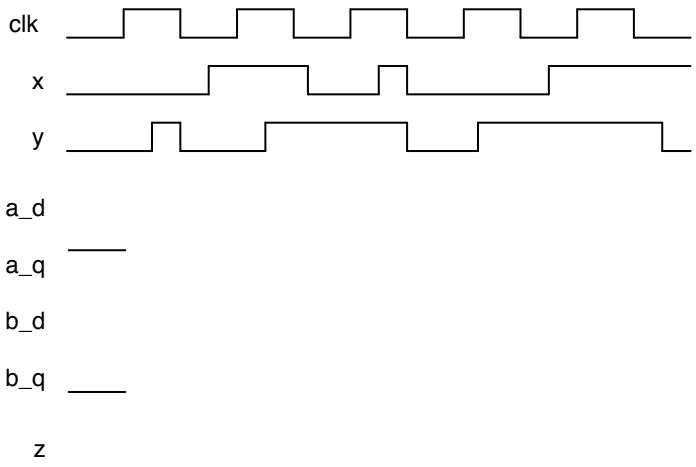
```
entity q1 is
  port (
    a : in  std_logic;
    b : in  std_logic;
    c : in  std_logic;
    x : out std_logic);
end q1;
architecture basic of q1 is
begin



end basic;
```

Question 2 Sequential Logic [10 pts]

Consider the following VHDL fragment, in which `x`, and `y` are inputs, `z` is an output, and there are two DFFs (`a` and `b`) whose `d` inputs are `a_d` and `b_d` respectively, and whose `q` outputs are `a_q` and `b_q` respectively:

```
b_d <= b_q xor x;
a_d <= (b_q and a_q) or y;
z   <= (not a_q) or b_q;
```

Complete the waveform below (assume that the DFFs are triggered by the rising edge of `clk`): Note that `a_q` is initially 1, and `b_q` is initially 0:



Question 3 FSMs [10 pts]

Draw a state machine diagram for a finite state machine which accepts a single bit input (either 0 or 1—you can just label each edge with 0 or 1). This state machine also has a single bit of output, which is initially 0.

- Whenever the FSM receives an input containing two `10` patterns, even if other input bits occur between the first `10` and the second `10` (although the 1 and 0 themselves must be consecutive), the output goes to 1 (after the second occurrence of `10`).
- The output remains at 1 until three *consecutive* 0s are received as input by the FSM, at which point the output returns to 0.
- The process then repeats (another two `1` followed by `0` inputs return the output to 1).

Label each state with the bit it outputs. Be sure to indicate your start state (with an arrow to it from nowhere).

Question 4 Asm Programming [20 pts]

Translate the following C function to MIPS assembly. **Answer on the pages after the MIPS reference** where you have each C-code line written out for you with space to write the MIPS assembly for that line directly under it.

```
int * count(int * ptr, int n) {
  int * ans = calloc(256 * sizeof(int));
  int i = 0 ;
  while (i < n) {
      int temp = ptr[i];
      int idx = getIndex(temp);
      ans[temp] = ans[temp] + 1;
      i++;
  }
  return ans;
}
```

Answer on next pages after the MIPS reference
The next 3 pages are MIPS reference material for your benefit.

MIPS Assembly Instructions

Arithmetic & Logical Instructions
abs Rdest, Rsrc Absolute Value y
add Rdest, Rsrc1, Src2 Addition (with overflow)
addi Rdest, Rsrc1, Imm Addition Immediate (with overflow)
addu Rdest, Rsrc1, Src2 Addition (without overflow)
addiu Rdest, Rsrc1, Imm Addition Immediate (without overflow)
and Rdest, Rsrc1, Src2 AND
andi Rdest, Rsrc1, Imm AND Immediate
div Rsrc1, Rsrc2 Divide (signed)
divu Rsrc1, Rsrc2 Divide (unsigned)
div Rdest, Rsrc1, Src2 Divide (signed, with overflow)
divu Rdest, Rsrc1, Src2 Divide (unsigned, without overflow)
mul Rdest, Rsrc1, Src2 Multiply (without overflow)
mulo Rdest, Rsrc1, Src2 Multiply (with overflow)
mulou Rdest, Rsrc1, Src2 Unsigned Multiply (with overflow)
mult Rsrc1, Rsrc2 Multiply
multu Rsrc1, Rsrc2 Unsigned Multiply
Multiply the contents of the two registers. Leave the low-order word of the product in register
lo and the high-word in register hi.
neg Rdest, Rsrc Negate Value (with overflow)
negu Rdest, Rsrc Negate Value (without overflow)
nor Rdest, Rsrc1, Src2 NOR
not Rdest, Rsrc NOT y
or Rdest, Rsrc1, Src2 OR
ori Rdest, Rsrc1, Imm OR Immediate
rem Rdest, Rsrc1, Src2 Remainder y
remu Rdest, Rsrc1, Src2 Unsigned Remainder
Put the remainder from dividing the integer in register Rsrc1 by the integer in Src2 into register Rdest.
rol Rdest, Rsrc1, Src2 Rotate Left
ror Rdest, Rsrc1, Src2 Rotate Right
sll Rdest, Rsrc1, Src2 Shift Left Logical
sllv Rdest, Rsrc1, Rsrc2 Shift Left Logical Variable
sra Rdest, Rsrc1, Src2 Shift Right Arithmetic
srav Rdest, Rsrc1, Rsrc2 Shift Right Arithmetic Variable
srl Rdest, Rsrc1, Src2 Shift Right Logical
srlv Rdest, Rsrc1, Rsrc2 Shift Right Logical Variable
sub Rdest, Rsrc1, Src2 Subtract (with overflow)
subu Rdest, Rsrc1, Src2 Subtract (without overflow)
xor Rdest, Rsrc1, Src2 XOR
xori Rdest, Rsrc1, Imm XOR Immediate

Constant-Manipulating Instructions
li Rdest, imm Load Immediate y
lui Rdest, imm Load Upper Immediate

Comparison Instructions
seq Rdest, Rsrc1, Src2 Set Equal
Set register Rdest to 1 if register Rsrc1 equals Src2 and to be 0 otherwise.
sge Rdest, Rsrc1, Src2 Set Greater Than Equal
sgeu Rdest, Rsrc1, Src2 Set Greater Than Equal Unsigned y
Set register Rdest to 1 if register Rsrc1 is greater than or equal to Src2 and to 0 otherwise.
sgt Rdest, Rsrc1, Src2 Set Greater Than
sgtu Rdest, Rsrc1, Src2 Set Greater Than Unsigned
Set register Rdest to 1 if register Rsrc1 is greater than Src2 and to 0 otherwise.
sle Rdest, Rsrc1, Src2 Set Less Than Equal y
sleu Rdest, Rsrc1, Src2 Set Less Than Equal Unsigned y
Set register Rdest to 1 if register Rsrc1 is less than or equal to Src2 and to 0 otherwise.
slt Rdest, Rsrc1, Src2 Set Less Than
slti Rdest, Rsrc1, Imm Set Less Than Immediate
sltu Rdest, Rsrc1, Src2 Set Less Than Unsigned
sltiu Rdest, Rsrc1, Imm Set Less Than Unsigned Immediate
Set register Rdest to 1 if register Rsrc1 is less than Src2 (or Imm) and to 0 otherwise.
sne Rdest, Rsrc1, Src2 Set Not Equal
Set register Rdest to 1 if register Rsrc1 is not equal to Src2 and to 0 otherwise.

Branch and Jump Instructions
b label Branch instruction y
Unconditionally branch to the instruction at the label.
bczt label Branch Coprocessor z True
bczf label Branch Coprocessor z False
Conditionally branch to the instruction at the label if coprocessor z's condition flag is true
(false).
beq Rsrc1, Src2, label Branch on Equal
Conditionally branch to the instruction at the label if the contents of register Rsrc1 equals Src2.
beqz Rsrc, label Branch on Equal Zero y
Conditionally branch to the instruction at the label if the contents of Rsrc equals 0.
bge Rsrc1, Src2, label Branch on Greater Than Equal
bgeu Rsrc1, Src2, label Branch on GTE Unsigned y
Conditionally branch to the instruction at the label if the contents of register Rsrc1 are greater
than or equal to Src2.
bgez Rsrc, label Branch on Greater Than Equal Zero
Conditionally branch to the instruction at the label if the contents of Rsrc are greater than or
equal to 0.
bgezal Rsrc, label Branch on Greater Than Equal Zero And Link
Conditionally branch to the instruction at the label if the contents of Rsrc are greater than or
equal to 0. Save the address of the next instruction in register 31.
bgt Rsrc1, Src2, label Branch on Greater Than
bgtu Rsrc1, Src2, label Branch on Greater Than Unsigned
Conditionally branch to the instruction at the label if the contents of register Rsrc1 are greater
than Src2.
bgtz Rsrc, label Branch on Greater Than Zero
Conditionally branch to the instruction at the label if the contents of Rsrc are greater than 0.
ble Rsrc1, Src2, label Branch on Less Than Equal
bleu Rsrc1, Src2, label Branch on LTE Unsigned
Conditionally branch to the instruction at the label if the contents of register Rsrc1 are less than or equal to Src2.
blez Rsrc, label Branch on Less Than Equal Zero
Conditionally branch to the instruction at the label if the contents of Rsrc are less than or equal to 0.
bgezal Rsrc, label Branch on Greater Than Equal Zero And Link
bltzal Rsrc, label Branch on Less Than And Link
Conditionally branch to the instruction at the label if the contents of Rsrc are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.
blt Rsrc1, Src2, label Branch on Less Than
bltu Rsrc1, Src2, label Branch on Less Than Unsigned

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are less
than Src2.
bltz Rsrc, label Branch on Less Than Zero
Conditionally branch to the instruction at the label if the contents of Rsrc are less than 0.
bne Rsrc1, Src2, label Branch on Not Equal
Conditionally branch to the instruction at the label if the contents of register Rsrc1 are not
equal to Src2.
bnez Rsrc, label Branch on Not Equal Zero
Conditionally branch to the instruction at the label if the contents of Rsrc are not equal to 0.
j label Jump
Unconditionally jump to the instruction at the label.
jal label Jump and Link
jalr Rsrc Jump and Link Register
Unconditionally jump to the instruction at the label or whose address is in register Rsrc. Save
the address of the next instruction in register 31.
jr Rsrc Jump Register
Unconditionally jump to the instruction whose address is in register Rsrc.

Load Instructions
la Rdest, address Load Address y
Load computed address, not the contents of the location, into register Rdest.
lb Rdest, address Load Byte
lbu Rdest, address Load Unsigned Byte
Load the byte at address into register Rdest. The byte is sign-extended by the lb, but not the
lbu, instruction.
ld Rdest, address Load Double-Word
Load the 64-bit quantity at address into registers Rdest and Rdest + 1.
lh Rdest, address Load Halfword
lhu Rdest, address Load Unsigned Halfword
Load the 16-bit quantity (halfword) at address into register Rdest. The halfword is sign-extended
by the lh, but not the lhu, instruction
lw Rdest, address Load Word
Load the 32-bit quantity (word) at address into register Rdest.
lwcz Rdest, address Load Word Coprocessor
Load the word at address into register Rdest of coprocessor z (0--3).
lwl Rdest, address Load Word Left
lwr Rdest, address Load Word Right
Load the left (right) bytes from the word at the possibly-unaligned address into register Rdest.
ulh Rdest, address Unaligned Load Halfword
ulhu Rdest, address Unaligned Load Halfword Unsigned
Load the 16-bit quantity (halfword) at the possibly-unaligned address into register Rdest. The halfword is sign-extended by the ulh, but not the ulhu, instruction
ulw Rdest, address Unaligned Load Word
Load the 32-bit quantity (word) at the possibly-unaligned address into register Rdest.

Store Instructions
sb Rsrc, address Store Byte
Store the low byte from register Rsrc at address.
sd Rsrc, address Store Double-Word y
Store the 64-bit quantity in registers Rsrc and Rsrc + 1 at address.
sh Rsrc, address Store Halfword
Store the low halfword from register Rsrc at address.
sw Rsrc, address Store Word
Store the word from register Rsrc at address.
swcz Rsrc, address Store Word Coprocessor
Store the word from register Rsrc of coprocessor z at address.
swl Rsrc, address Store Word Left
swr Rsrc, address Store Word Right
Store the left (right) bytes from register Rsrc at the possibly-unaligned address.
ush Rsrc, address Unaligned Store Halfword
Store the low halfword from register Rsrc at the possibly-unaligned address.
usw Rsrc, address Unaligned Store Word
Store the word from register Rsrc at the possibly-unaligned address.

Data Movement Instructions
move Rdest, Rsrc Move y
Move the contents of Rsrc to Rdest.
The multiply and divide unit produces its result in two additional registers, hi and lo. These instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudoinstructions that make it appear as if this unit operates
on the general registers and detect error conditions such as divide by zero or overflow.
mfhi Rdest Move From hi
mflo Rdest Move From lo
Move the contents of the hi (lo) register to register Rdest.
mthi Rdest Move To hi
mtlo Rdest Move To lo
Move the contents register Rdest to the hi (lo) register.
Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.
mfcz Rdest, CPsrc Move From Coprocessor z
Move the contents of coprocessor z's register CPsrc to CPU register Rdest.
mfc1.d Rdest, FRsrc1 Move Double From Coprocessor 1
Move the contents of floating point registers FRsrc1 and FRsrc1 + 1 to CPU registers Rdest
and Rdest + 1.
mtcz Rsrc, CPdest Move To Coprocessor z
Move the contents of CPU register Rsrc to coprocessor z's register CPdest.

System Call Interface

print int	1	$a0 = integer
print float	2	$f12 = float
print double	3	$f12 = double
print string	4	$a0 = string
read int	5	integer (in $v0)
read float	6	float (in $f0)
read double	7	double (in $f0)
read string	8	$a0 = buffer, $a1 = length
sbrk	9	$a0 = amount address (in $v0)
exit	10	

.align n
Align the next datum on a 2 n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off

automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

.ascii str
Store the string in memory, but do not null-terminate it.

.asciiz str
Store the string in memory and null-terminate it.

.byte b1, ..., bn
Store the n values in successive bytes of memory.

.data <addr>
The following data items should be stored in the data segment. If the optional argument addr is present, the items are stored beginning at address addr .

.double d1, ..., dn
Store the n floating point double precision numbers in successive memory locations.

.extern sym size
Declare that the datum stored at sym is size bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register $gp.

.float f1, ..., fn
Store the n floating point single precision numbers in successive memory locations.

.globl sym
Declare that symbol sym is global and can be referenced from other files.

.half h1, ..., hn
Store the n 16-bit quantities in successive memory halfwords.

.space n
Allocate n bytes of space in the current segment (which must be the data segment in SPIM).

.text <addr>
The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, the items are stored beginning at address addr .

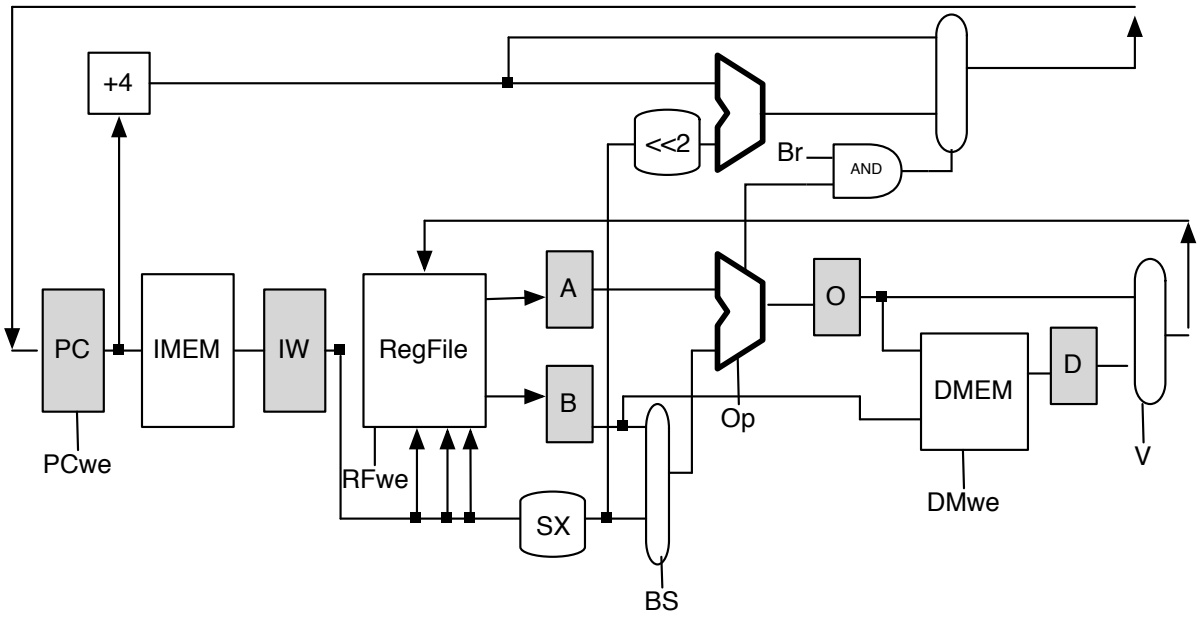
.word w1, ..., wn
Store the n 32-bit quantities in successive memory words.

```
# int * count(int * ptr, int n) {

#   int * ans = calloc(1024); //256 * sizeof(int) = 1024

#   int i = 0 ;

#   while (i < n) {

#       int temp = ptr[i];

#       int idx = getIndex(temp);
```

```
#       ans[temp] = ans[temp] + 1;




#       i++;

#   }



#   return ans;









# }
```

Question 5 Datapaths [10 pts]

Consider the following multi-cycle data path:



Suppose we wanted to add support for a (CISC-style) instruction, `lw-bzr $rt, imm($rs)`. This instruction loads a *word* from memory (at address `$rs` imm+ and tests if it is zero. If the word loaded is zero, this instruction branches to the PC specified in `$rt`. If the word loaded is non-zero, the instruction behaves like a not-taken branch (advancing to PC+4, but having no other effect).

The following three pages show three different proposed modifications to the datapath. For each you will select one of the following four options (you clearly will not use all four options. You may use an option multiple times if needed):

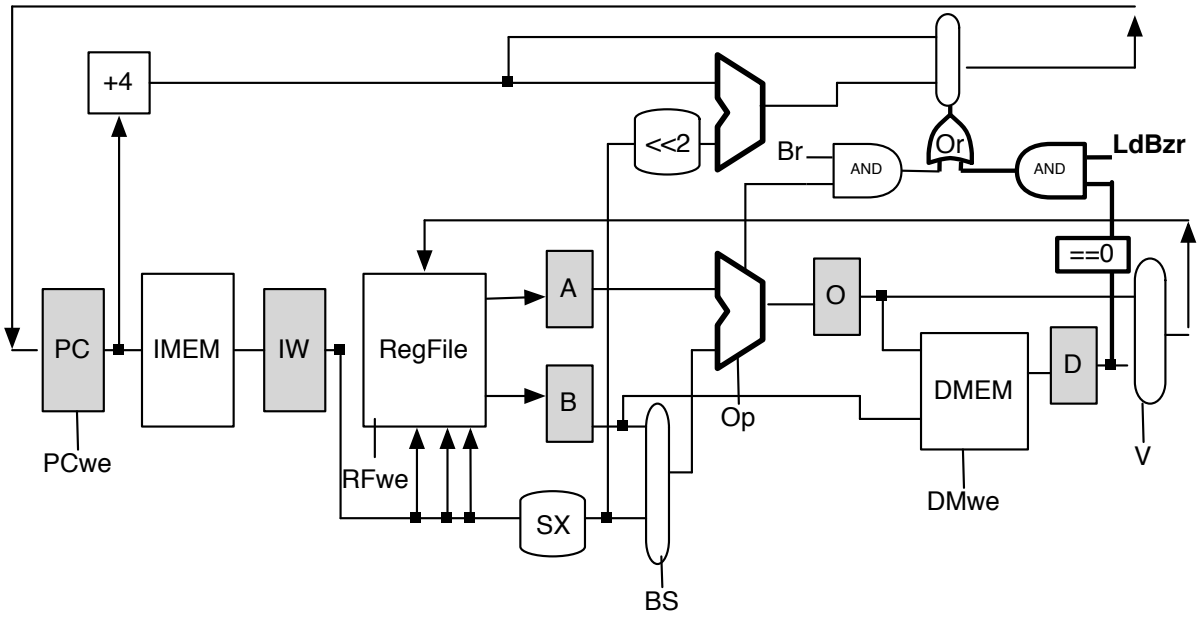
Does not implement this instruction. Select this option if the proposed design does not provide a way to execute the proposed instruction.

Breaks other functionality. Select this option if the proposed design implements the `lw-bzr` instruction, but makes the datapath incapable of correctly executing some other instruction. **Give an example of an instruction that breaks.**

Functionally correct, but impacts clock frequency. Select this option if the proposed design implements the `lw-bzr` instruction and does not break any other instructions, but the implementation would have a dramatically bad impact on the clock frequency of the design.

Correct. Select this option if the proposed design is the correct way to implement the `lw-bzr` instruction. **If you select this option, fill in the control signals required to execute the `lw-bzr` instruction.**

Option 1:



Does not implement this instruction.

Breaks other functionality. Which instruction breaks?

Functionally correct, but impacts clock frequency.

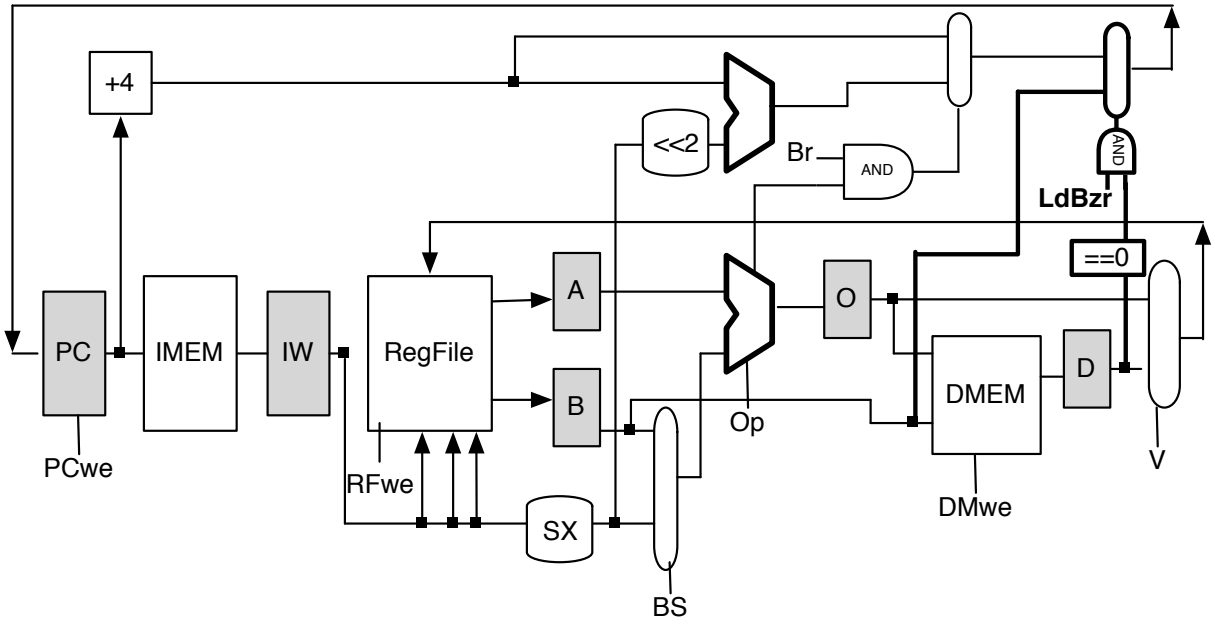
Correct. If you select this option, fill in the table of control signals below:

Cyc	PCwe	RFwe	BS	Op	DMwe	Br	V	LdBrz
1								
2								
3								
4								
5								

A few notes about filling in the table:

- Mux selectors have 0 for the top input, 1 for the bottom.
- Write enables (we) are 0 for disabled, 1 for enabled
- For Op, you can write down the symbol for the mathematical operation you want (`+`, `-`, `*`, `<<`,etc).
- You should write X for "dont care" if (and only if) that control signal does not matter.

Option 2:



Does not implement this instruction.

Breaks other functionality. Which instruction breaks?

Functionally correct, but impacts clock frequency.

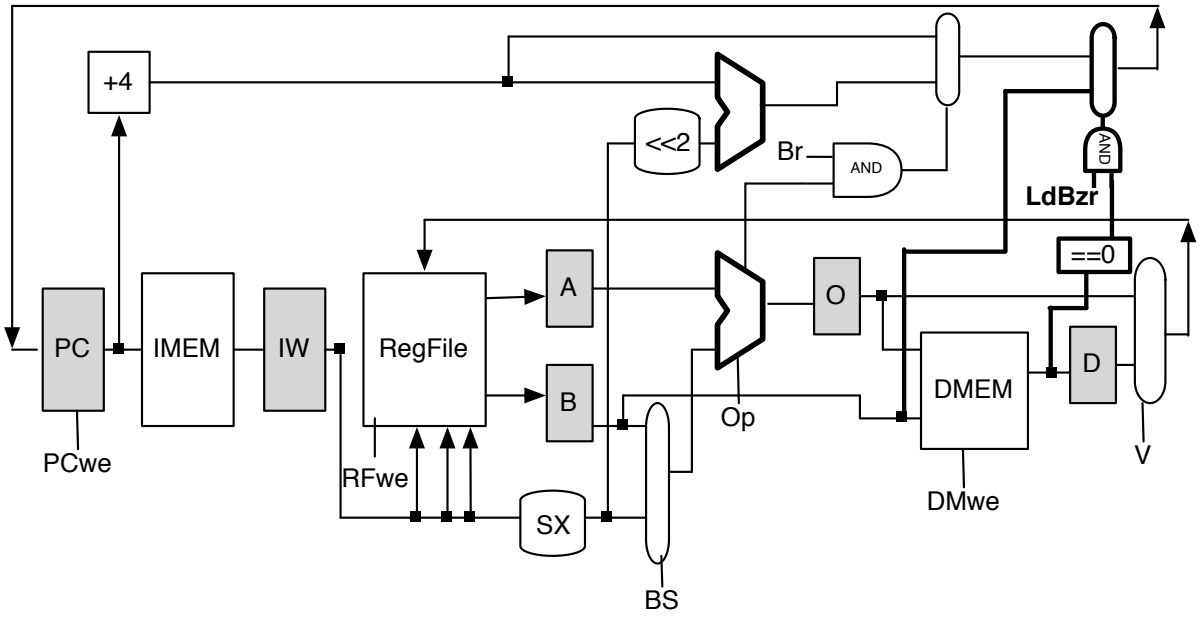
Correct. If you select this option, fill in the table of control signals below:

Cyc	PCwe	RFwe	BS	Op	DMwe	Br	V	LdBrz
1								
2								
3								
4								
5								

A few notes about filling in the table:

- Mux selectors have 0 for the top input, 1 for the bottom.
- Write enables (we) are 0 for disabled, 1 for enabled
- For Op, you can write down the symbol for the mathematical operation you want (`+`, `-`, `*`, `<<`,etc).
- You should write X for "dont care" if (and only if) that control signal does not matter.

Option 3:



Does not implement this instruction.

Breaks other functionality. Which instruction breaks?

Functionally correct, but impacts clock frequency.

Correct. If you select this option, fill in the table of control signals below:

Cyc	PCwe	RFwe	BS	Op	DMwe	Br	V	LdBrz
1								
2								
3								
4								
5								

A few notes about filling in the table:

- Mux selectors have 0 for the top input, 1 for the bottom.
- Write enables (we) are 0 for disabled, 1 for enabled
- For Op, you can write down the symbol for the mathematical operation you want (`+`, `-`, `*`, `<<`,etc).
- You should write X for "dont care" if (and only if) that control signal does not matter.

Question 6 Memory Hierarchy [10 pts]

Suppose that you have a memory hierarchy with the following caches:

L1 Data Cache 2 cycle hit, 10% miss rate

L2 Cache 10 cycle hit, 20% miss rate

Main Memory 150 cycle latency

- What is the average access latency of the L1 Data Cache (in cycles)?

- Suppose that the processor's clock frequency were doubled, what would the new average access time of the L1 Data Cache be (in cycles)?

- Instead of doubling the clock frequency, suppose that an L3 cache were added with a hit latency of 50 cycles. What hit rate is required to make the average access time of the L1 data cache 3.8 cycles?