Question 1 (10 points): Write a program that computes t0×t1 and leaves the result in register t2. You can use only the following instructions: addi, add, sub, sll, srl, nand, beq. Do not worry about efficiency or following MIPS calling conventions. Assume that before your program runs t0 and t1 contain the values you need to multiply. Furthermore, you can assume that both t0 and t1 are positive integers. Do not worry about overflow.

```
ADD     $t2, $0, $0               # clear $t2
BEQ     $t0, $0, finished         # if either operand=0 → multiplication gives 0
BEQ     $t1, $0, finished

loop:
ADD     $t2, $t2, $t0             # increment $t2 by $t0
ADDi    $t1, $t1, -1              # decrement ‘counter’ ($t1)
BEQ     $t1, $0, finished         # finished if counter reach 0
BEQ     $0, $0, loop              # continue

finished:
```

Question 2 (10 points): Write a program that computes t0 × t1 and leaves the result in register t2. You can use only the following instructions: add, sub, sll, srl, nand, beq. Do not worry about efficiency or following MIPS calling conventions. Assume that before your program runs t0 and t1 contain the values you need to multiply. Furthermore, you can assume that both t0 and t1 are positive integers. Do not worry about overflow. (Note this question is different than question 1).

```
ADD     $t2, $0, $0             # clear $t2
BEQ     $t0, $0, finished       # if either operand=0 → multiplication gives 0
BEQ     $t1, $0, finished
NAND    $t3, $0, $0             # $t3 gets all 1s
SLL     $3, $3, 31              # erase all higher bit 1s
SRL     $3, $3, 31              # 1 stored in $t3

loop:
ADD     $t2, $t2, $t0           # increment $t2 by $t0
SUB     $t1, $t1, $t3           # decrement ‘counter’ ($t1)
BEQ     $t1, $0, finished       # finished if counter reach 0
BEQ     $0, $0, loop            # continue

finished:
```

Question 3 (10 points): Write a program that computes t0 × t1 and leaves the result in register t2. You can use only the following instructions: add, sll, srl, nand, beq. Do not worry about efficiency or following MIPS calling conventions. Assume that before your program runs t0 and t1 contain the values you need to multiply. Furthermore, you can assume that both t0 and t1 are positive integers. Do not worry about overflow. (Note this question is different than question 2).

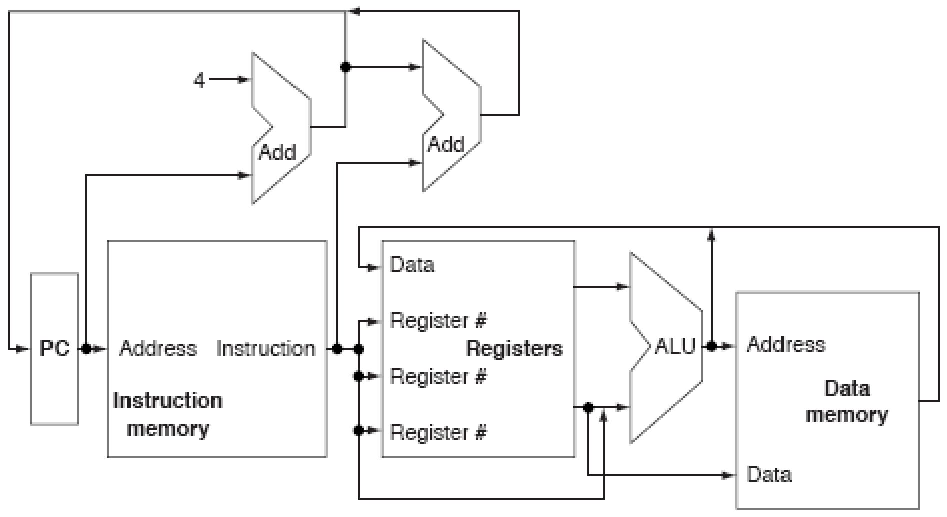
```
ADD     $t2, $0, $0             # clear $t2
BEQ     $t0, $0, finished       # if either operand=0 → multiplication gives 0
BEQ     $t1, $0, finished
NAND    $t3, $0, $0             # all 1s in reg is equivalent to -1

loop:
ADD     $t2, $t2, $t0           # increment $t2 by $t0
ADD     $t1, $t1, $t3           # decrement ‘counter’ ($t1)
BEQ     $t1, $0, finished       # finished if counter reach 0
BEQ     $0, $0, loop            # continue

finished:
```

Question 4 (10 points): Write a program that computes t0 × t1 and leaves the result in register t2. You can use only the following instructions: add, sll, srl, nand. Do not worry about efficiency or following MIPS calling conventions. Assume that before your program runs t0 and t1 contain the values you need to multiply. Furthermore, you can assume that both t0 and t1 are positive integers. Do not worry about overflow. (Note this question is different than question 3).

```
ADD     $t2, $0, $0                # $t2 = final result
ADD     $t5, $0, $0

# repeat 1
SLL     $t3, $t1, 31               # get the lowest bit from $t1
SRL     $t3, $t3, 31
NAND    $t4, $t0, $t3              # 2 NANDs → AND
NAND    $t4, $t4, $t4
ADD     $t5, $t4, $t5

# repeat 2
SLL     $t3, $t3, 1
NAND    $t4, $t0, $t3
NAND    $t4, $t4, $t4
ADD     $t5, $t4, $t5
.
.
.
# repeat the above 32 times → multiplication result of lowest bit of $t1 w/ entire $t0

ADD     $t2, $t2, $t5              # update result reg w/ $t5
SLL     $t0, $t0, 1                # go on to bit of $t1 → need to shift $t0
SLL     $t3, $t1, 30               # ↓shift amount each iteration
SRL     $t3, $t3, 32
.
.
.
# repeat the above 32 repeats 32 times → multiplication result of all bits of $t1 w/ entire $t2
```

```
e.g.            0101    → $t0
             X   101    → $t1
                0101    → $t5: mult result of t1's lowest bit w/ t0 (32 repeats b/c 32 bits in reg)
               0000     → …second bit
              0101      → …third bit
              011001    → $t2
```

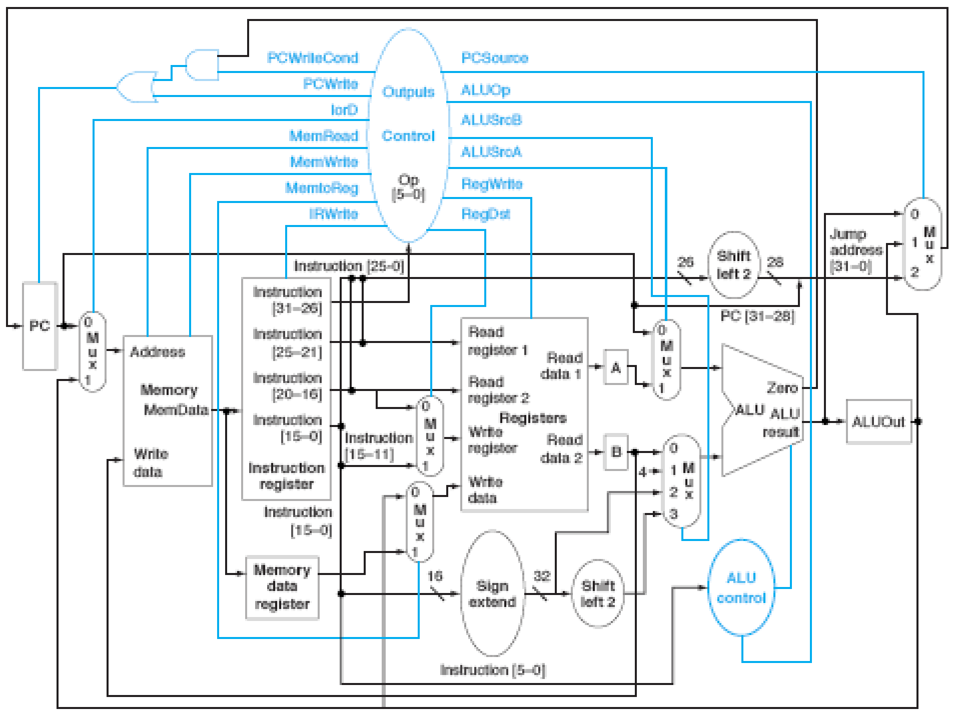
Question 5 (20 points): Modify the following machine to support the following new instruction: JRLZ. It stands for Jump-to-register if less than zero. The instruction is used in assembly like (as an example):
jrlz $t0, $t1
The result of the instruction is to cause the machine to branch to the address specified in $t0 if $t1 is less than zero.



components to add:	[2:1] mux
	AND gate
	SLL component

PC+4 addr and $t0 addr (after SLL by 2 to get actual inst. addr) are selected through a mux and fed into the PC component as next instruction addr. The mux-select bit is from an AND gate that ANDs the output of ALU (is $t1 < 0?) with instruction op (is inst. JRLZ?)

Question 6 (20 points): The following diagram looks a lot like Lab 1, but is subtly different. Explain how to make this processor support loads and stores. Write the control signals required to do this and explain their operation.



It takes 2 cycles to carry out either LW or SW:

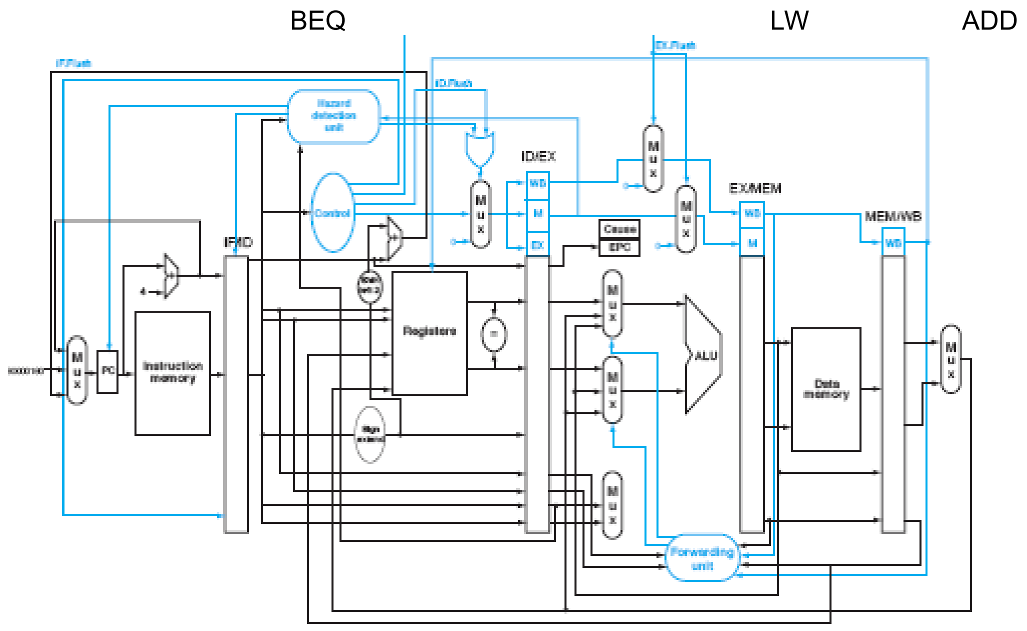
1. fetch instruction
2. execute instruction

	LW	SW
On Fetch	PCWriteCond: 0	PCWriteCond: 0
	PCWrite: 1	PCWrite: 1
	MemRead: 1	MemRead: 1
	MemWrite: 0	MemWrite: 0
	MemtoReg: 0	MemtoReg: 0
	IRWrite: 1	IRWrite: 1
	PCSource: 0	PCSource: 0
	ALUsrcB: 1	ALUsrcB: 1
	ALUsrcA: 0	ALUsrcA: 0
	RegWrite: 0	RegWrite: 0
On Execute	PCWrite: 0	PCWrite: 0
	MemRead: 1	MemRead: 0
	MemWrite: 0	MemWrite: 1
	MemtoReg: 1	MemtoReg: 0
	IRWrite: 0	IRWrite: 0
	ALUsrcB: 0	ALUsrcB: 0
	ALUsrcA: 1	ALUsrcA: 1
	RegWrite: 1	RegWrite: 0
	RegDst: 0	

Question 7 (20 points): Suppose the following instructions were fetched:

```
        add     $t1, $t3, $t4
        lw      $t0, 0($t1)
        beq     $t0, $t2, somewhere
```

Suppose the lw instruction is in the memory stage. Draw on the diagram where the beq and add instructions are (which stage). Label all forwarding networks that are being used in that clock cycle and indicate what data values are being forwarded. It is possible you will have to modify the diagram to support your answer and make the processor properly execute. If this is the case, draw these modifications too.



Add a mux that outputs to the comparator:
input = select between dataMem output or regFile output
mux-select bit = output from the forwarding unit that recognizes LW/BEQ dependence