1 Verilog

Please answer the following four questions about Verilog.

(a) Does the following code result in a sequential circuit or a combinational circuit? Explain why.

```verilog
module concat (input clk, input data_in1, input data_in2,
                               output reg [1:0] data_out);
 always @ (posedge clk, data_in1)
   if (data_in1)
      data_out = {data_in1, data_in2};
   else if (data_in2)
      data_out = {data_in2, data_in1};
endmodule
```

Answer and concise explanation:

(b) The input `clk` is a clock signal. What is the hexadecimal value of the output `c` right after the third positive edge of `clk` if initially `c = 8'hE3` and `a = 4'd8` and `b = 4'o2` during the entire time?

```verilog
module mod1 (input clk, input [3:0] a, input [3:0] b, output reg [7:0] c);
always @ (posedge clk)
  begin
    c <= {c, &a, |b};
    c[0] <= ^c[7:6];
  end
endmodule
```

Please answer below. Show your work.

(c) Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```verilog
module 1nn3r ( input [3:0] d, input op, output[1:0] s);
  assign s = op ? (d[1:0] - d[3:2]) :
                     (d[3:2] + d[1:0]);
endmodule

module top ( input wire [6:0] instr, input wire op, output reg z);

  reg[1:0] r1, r2;

  1nn3r i0 (.instr(instr[1:0]), .op(instr[7]), .z(r1) );
  1nn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
  assign z = r1 | r2;

endmodule
```

Answer and concise explanation:

(d) Does the following code correctly implement a counter that counts from 1 to 11 by increments of 2 (e.g., 1, 3, 5, 7, 9, 11, 1, 3 ...)? If so, say "Correct". If not, correct the code with minimal modification.

```verilog
module odd_counter (clk, count);
   input wire clk;
   output reg[2:0] count;
   reg[2:0] count_next;

   always@*
   begin
      count_next = count;
      if(count != 11)
         count_next = count_next + 2;
      else
         count_next <= 1;
   end

   always@(posedge clk)
      count <= count_next;
endmodule
```

Answer and concise explanation:

(e) Does the following code correctly instantiate a 4-bit adder? If so, say "Correct". If not, correct the code with minimal modification.

```verilog
module adder(input a, input b, input c, output sum, output carry);
assign sum = a ^ b ^ c;
assign carry = (a&b) | (b&c) | (c&a);
endmodule


module adder_4bits(input [3:0] a, input [3:0] b, output [3:0] sum, output carry);
wire [2:0]s;

adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
adder u1 (a[1],s[0],b[1],sum[1],s[1]);
adder u2 (a[2],s[1],b[2],sum[2],s[2]);
adder u3 (a[3],s[2],b[3],sum[3],carry);
endmodule
```

Answer and concise explanation:

2 Finite State Machine

You are given the following FSM with two one-bit input signals (T_A and T_B) and one two-bit output signal (O). You need to implement this FSM, but you are unsure about how you should encode the states. Answer the following questions to get a better sense of the FSM and how the three different types of state encoding we dicussed in the lecture (i.e., one-hot, binary, output) will affect the implementation.
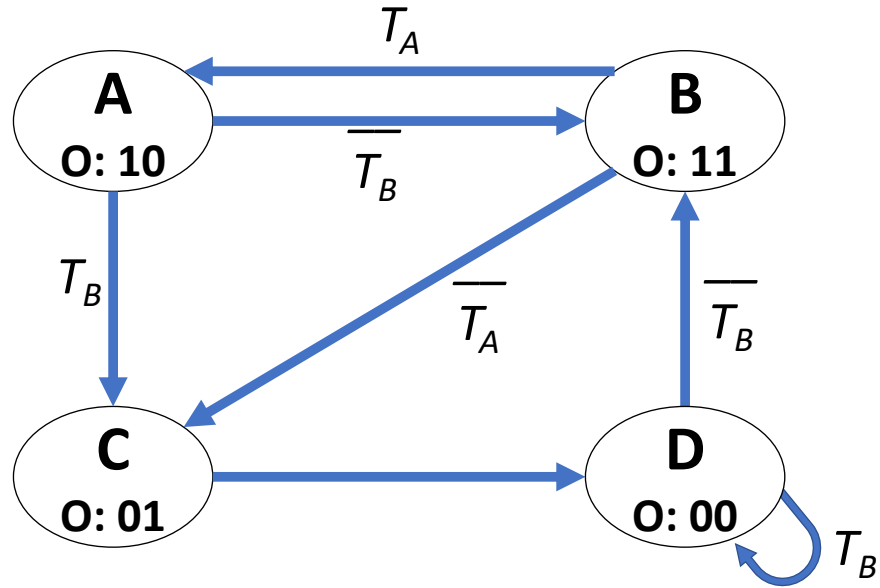
(a) There is one critical component of an FSM that is *missing* in this diagram. Please write what is missing in the answer box below.

(b) What kind of an FSM is this?

(c) List one major advantage of each type of state encoding below.

One-hot encoding

Binary (i.e., fully) encoding

Output encoding

(d) Fully describe the FSM with equations given that the states are encoded with **one-hot** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with one-hot encoding. Indicate the values you assign to each state and simplify all equations:

(e) Fully describe the FSM with equations given that the states are encoded with **binary** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with binary encoding. Indicate the values you assign to each state and simplify all equations:

(f) Fully describe the FSM with equations given that the states are encoded with **output** encoding. Use the **minimum** possible number of bits to represent the states with output encoding. Indicate the values you assign to each state and simplify all equations:

(g) Assume the following conditions:

- We can only implement our FSM with 2-input AND gates, 2-input OR gates, and D flip-flops.
- 2-input AND gates and 2-input OR gates occupy the *same* area.
- D flip-flops occupy 3x the area of 2-input AND gates.

Which state-encoding do you choose to implement in order to minimize the total area of this FSM?

3 Big versus Little Endian Addressing

Consider the 32-bit hexadecimal number 0xcafe2b3a.

1. What is the binary representation of this number in *little endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

2. What is the binary representation of this number in *big endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

4 The MIPS ISA

4.1 Warmup: Computing a Fibonacci Number

The Fibonacci number F_n is recursively defined as

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the MIPS assembly for the `fib(n)` function, which computes the Fibonacci number $F(\texttt{n})$:

```
int fib(int n)
{
  int a = 0;
  int b = 1;
  int c = a + b;
  while (n > 1) {
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  return c;
}
```

Remember to follow MIPS calling convention and its register usage (just for your reference, you may not need to use all of these registers):

- The argument `n` is passed in register $4.
- The result (i.e., `c`) should be returned in $2.
- $8 to $15 are caller-saved temporary registers.
- $16 to $23 are callee-saved temporary registers.
- $29 is the stack pointer register.
- $31 stores the return address.

Note: A summary of the MIPS ISA is provided at the end of this handout.

4.2 MIPS Assembly for REP MOVSB

MIPS is a simple ISA. Complex ISAs—such as Intel's x86—often use one instruction to perform the function of many instructions in a simple ISA. Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which is specified as follows.

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The "repeat" (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

(a) Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.

(b) What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

(c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0xFEE1DEAD
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same fuction as the single REP MOVSB in x86 accomplishes for the given register state?

(d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0x00000000
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, answer the same question in (c) for the above register values.

5 Data Flow Programs

Draw the data flow graph for the `fib(n)` function from Question 4.1. You may use the following data flow nodes in your graph:

- \+ (addition)
- \> (left operand is greater than right operand)
- `Copy` (copy the value on the input to both outputs)
- `BR` (branch, with the semantics discussed in class, label the True and False outputs)

You can use constant inputs (e.g., 1) that feed into the nodes. Clearly label all the nodes, program inputs, and program outputs. Try to the use fewest number of data flow nodes possible.

6   Microarchitecture vs. ISA

a) Briefly explain the difference between the *microarchitecture* level and the *ISA* level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile a given program correctly?

b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

Microarchitecture?	ISA?	Attribute
		The machine does not have a subtract instruction
		The ALU of the machine does not have a subtract unit
		The machine does not have condition codes
		A 5-bit immediate can be specified in an ADD instruction
		It takes n cycles to execute an ADD instruction
		There are 8 general purpose registers
		A 2-to-1 mux feeds one of the inputs to ALU
		The register file has one input port and two output ports

7   Performance Metrics

- If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)

- If a processor executes more of a given program's instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)

8 Performance Evaluation

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA A, the best compiled code for this benchmark performs at the rate of 10 IPC. That processor has a 500 MHz clock. On the processor implementing ISA B, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 600 MHz clock.

- What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA A?

- What is the performance in MIPS of the processor implementing ISA B?

- Which is the higher performance processor: A B Don't know
 Briefly explain your answer.

9 Single-Cycle Processor Datapath

In this problem, you will modify the single-cycle datapath we built up in Lecture 11 to support the JAL instruction. The datapath that we will start with is provided below. Your job is to implement the necessary data and control signals to support the JAL instruction, which we define to have the following semantics:

$$\text{JAL}: \quad \text{R31} \leftarrow \text{PC} + 4$$
$$\text{PC} \leftarrow \text{PC}_{31\ldots28} \parallel \text{Immediate} \parallel 0^2$$

Add to the datapath on the next page the necessary data and control signals to implement the JAL instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).
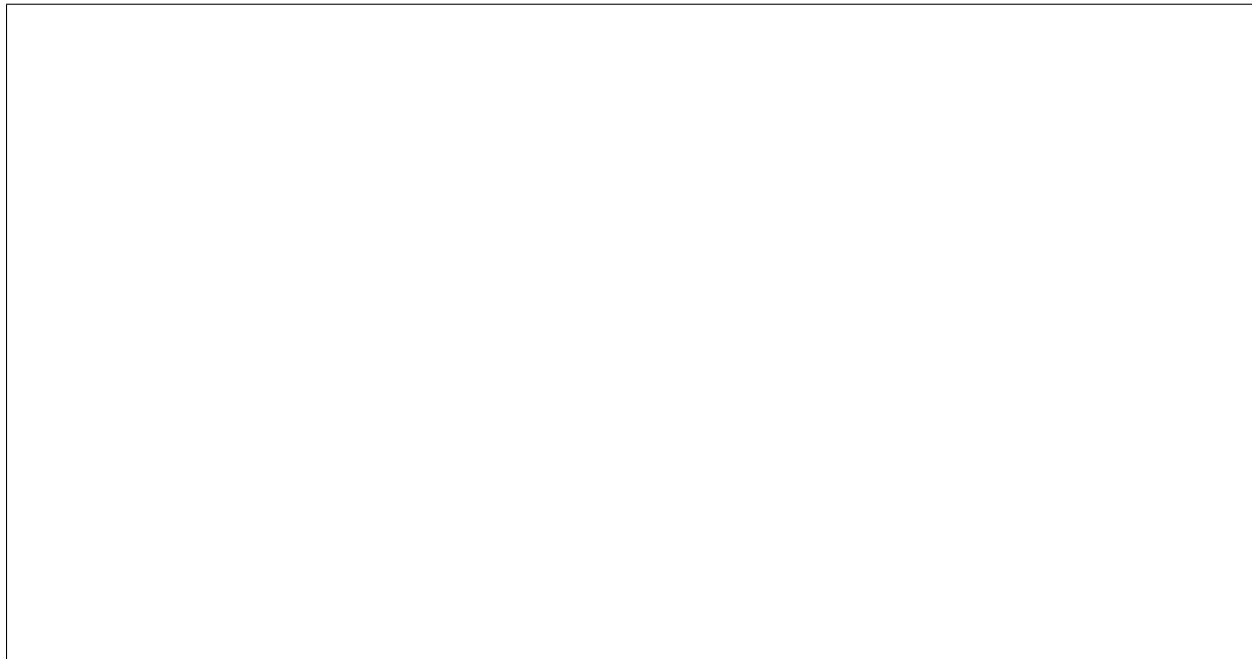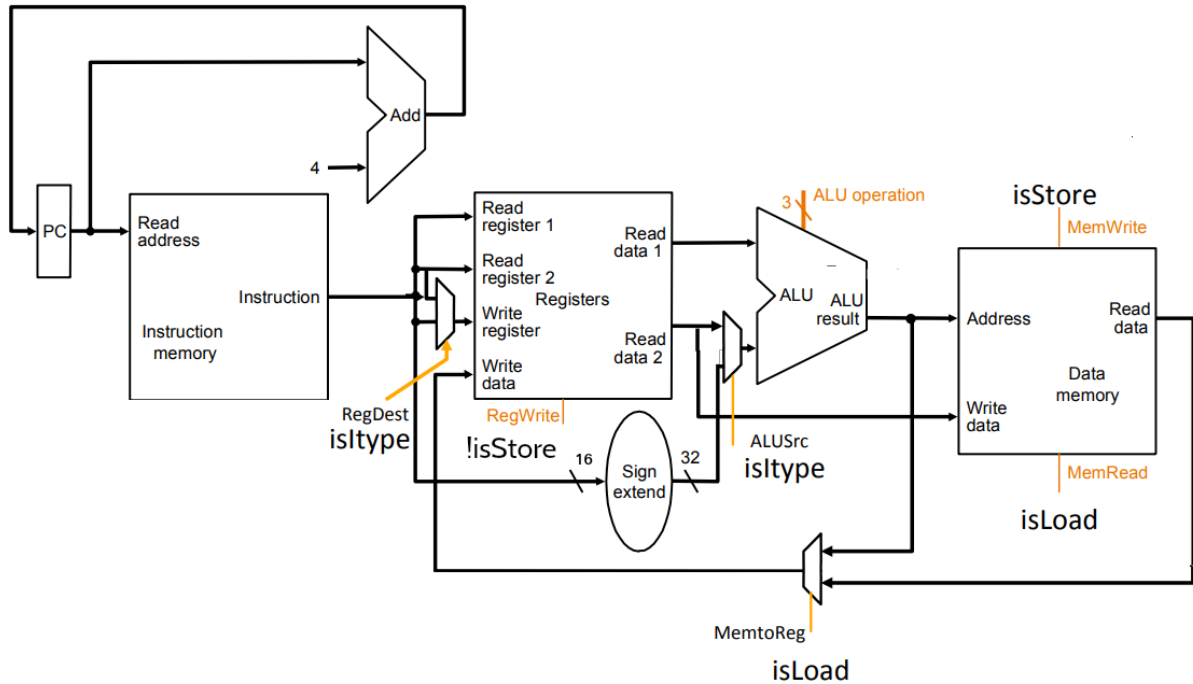
10 REP MOVSB

Let's say you are the lead architect of the next flagship processor at Advanced Number Devices (AND). You have decided that you want to use the LC-3b ISA for your next product, but your customers want a smaller semantic gap and marketing is on your case about it. So, you have decided to implement your favorite x86 instruction, REP MOVSB, in LC-3b.

Specifically, you want to implement the following definition for REP MOVSB (in LC-3b parlance): REP-MOVSB SR1, SR2, DR which is encoded in LC-3b machine code as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DR			SR1			0	0	0	SR2		

REPMOVSB uses three registers: SR1 (count), SR2 (source), and DR (destination). It moves a byte from memory at address SR2 to memory at address DR, and then increments both pointers by one. This is repeated SR1 times. Thus, the instruction copies SR1 bytes from address SR2 to address DR. Assume that the value in SR1 is greater than or equal to zero.

1. Complete the state diagram shown below, using the notation of the LC-3b state diagram. Describe inside each bubble what happens in each state and assign each state an appropriate state number. Add additional states not present in the original LC-3b design as you see fit.
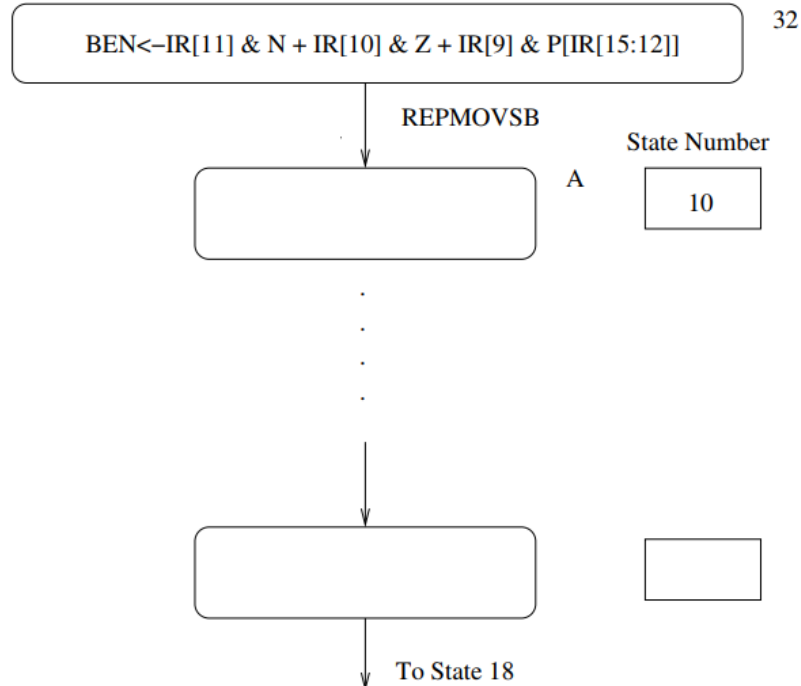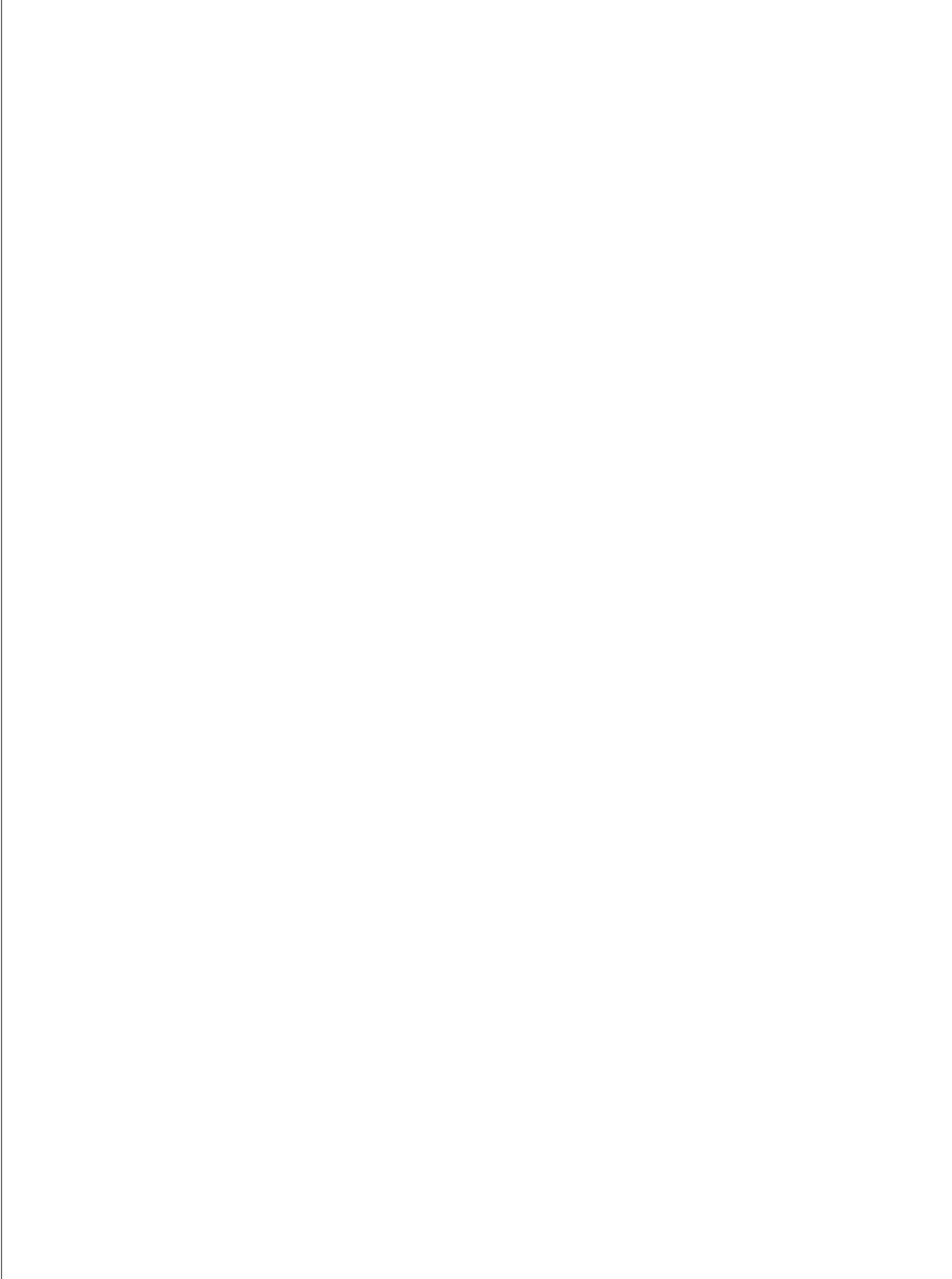
2. Add to the LC-3b datapath any additional structures and any additional control signals needed to implement REPMOVSB. Clearly label your additional control signals with descriptive names. Describe what value each control signal would take to control the datapath in a particular way.

3. Describe any changes you need to make to the LC-3b microsequencer. Add any additional logic and control signals you need. Clearly describe the purpose and function of each signal and the values it would take to control the microsequencer in a particular way.

MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	add $1, $2, $3	$1 = $2 + $3
sub	sub $1, $2, $3	$1 = $2 - $3
add immediate	addi $1, $2, 100	$1 = $2 + 100
add unsigned	addu $1, $2, $3	$1 = $2 + $3
subtract unsigned	subu $1, $2, $3	$1 = $2 - $3
add immediate unsigned	addiu $1, $2, 100	$1 = $2 + 100
multiply	mult $2, $3	hi, lo = $2 * $3
multiply unsigned	multu $2, $3	hi, lo = $2 * $3
divide	div $2, $3	lo = $2/$3, hi = $2 mod $3
divide unsigned	divu $2, $3	lo = $2/$3, hi = $2 mod $3
move from hi	mfhi $1	$1 = hi
move from low	mflo $1	$1 = lo
and	and $1, $2, $3	$1 = $2 & $3
or	or $1, $2, $3	$1 = $2 | $3
and immediate	andi $1, $2, 100	$1 = $2 & 100
or immediate	ori $1, $2, 100	$1 = $2 | 100
shift left logical	sll $1, $2, 10	$1 = $2 « 10
shift right logical	srl $1, $2, 10	$1 = $2 » 10
load word	lw $1, 100($2)	$1 = memory[$2 + 100]
store word	sw $1, 100($2)	memory[$2 + 100] = $1
load upper immediate	lui $1, 100	$1 = 100 « 16
branch on equal	beq $1, $2, label	if ($1 == $2) goto label
branch on not equal	bne $1, $2, label	if ($1 != $2) goto label
set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set on less than immediate	slti $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
set on less than unsigned	sltu $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set on less than immediate	sltui $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
jump	j label	goto label
jump register	jr $31	goto $31
jump and link	jal label	$31 = PC + 4; goto label