Digital Design and Computer Architecture (252-0028-00L), Spring 2023

Optional HW 2: Sequential Logic and Verilog

SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Juan Gomez Luna, Mohammad Sadrosadati, Mohammed Alser, Ataberk Olgun, Giray Yaglikci, Can Firtina, Geraldo De Oliveira Junior, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Nisa Bostancı, Rakesh Nadig, Joel Lindegger, İsmail Emir Yüksel, Haocong Luo, Yahya Can Tuğrul, Julien Eudine

Released: Wednesday, March 29, 2023

1 Verilog (I)

Please answer the following three questions about Verilog.

(a) Does the following code result in a D Flip-Flop with a synchronous active-low reset? Please explain your answer.

```
1  module mem (input clk, input reset, input [1:0] d, output reg [1:0] q);
2  always @ (posedge clk or negedge reset)
3      begin
4        if (!reset) q <= 0;
5        else q <= d;
6      end
7  endmodule
```

No.
The code implements *two* D Flip-Flops, not *one*. Each D Flip-Flop works with an *asynchronous* active-low reset signal.

Explanation:

- D and Q signals are two-bit-wide. Therefore, this code implements two D flip-flops.
- The reset input is included in the sensitivity list, therefore it is not synchronous.
- The code resets the output if the reset signal is low. Thus, the reset signal is active-low.

(b) Does the following code result in a sequential circuit or a combinational circuit? Please explain your answer.

```
1  module Mask (input [1:0] data_in, input mask, output reg [1:0] data_out);
2  always @ (*)
3      begin
4          data_out[1] = data_in[1];
5          if (mask)
6              data_out[0] = 0;
7      end
8  endmodule
```

Sequential circuit.
Explanation:
This code results in a sequential circuit, as all the left-hand side signals are not assigned in every possible condition. For example, `data_out[0]` is not assigned when mask signal equals to zero.

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

```
1  module fulladd(input a, b, c, output reg s, c_out);
2     assign s = a^b;
3     assign c_out = (a & b) | (b & c) & (c & a);
4  endmodule
5
6  module top ( input wire [5:0] instr, input wire op, output z);
7
8    reg[1:0] r1, r2;
9    wire [3:0] w1, w2;
10
11   fulladd FA1 (.a(instr[0]), .b(instr[1]), .c(instr[2]),
12                              .c_out(r1[1]), .z(r1[0]));
13   fulladd FA2 (.a(instr[3]), .b(instr[4]), .c(instr[5]),
14                              .z(r2[0]), .c_out(r2[1]));
15
16   assign z = r1 | op;
17   assign w1 = r1 + 1;
18   assign w2 = r2 << 1;
19   assign op = r1 ^ r2;
20
21 endmodule
```

The code is *not* syntactically correct.

Explanation:

- 'r1' and 'r2' have to be declared as *wires*.
- 'op' signal is connected to multiple drivers. It gets assigned from the input port and in line 19.
- The module 'fulladd' does not have ports named 'z'. Those need to be changed to 's'.
- The output signals 's' and 'c_out' have to be declared as *wires* but not as *regs*, since they are driven by *assign* statements.

2 Verilog (II)

Please answer the following four questions about Verilog.

(a) Does the following code result in a D Flip-Flop with asynchronous reset? Please explain why.

```
module dff (input clk, input reset, input [3:0] d, output reg [3:0] q);
always @ (posedge clk)
    begin
      if (reset == 0) q <= 0;
      else q <= d;
    end
endmodule
```

No.

Explanation:
Since the reset input is not included in the sensitivity list, this code will implement a synchronous D Flip-Flop.

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

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

Combinational circuit.

Explanation:
This code results in a combinational circuit because sensitivity list does include all inputs of the circuit: data_in1 and data_in2.

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

```
1  module Inn3r ( input [3:0] d, input op, output s);
2    assign s = op ? (d[1:0] - d[3:2]) :
3                        (d[3:2] + d[1:0]);
4  endmodule
5
6  module top ( input wire [6:0] instr, input wire op, output reg z);
7
8    reg[1:0] r1, r2, r3;
9    wire [3:0] w1, w2;
10
11   Inn3r i0 (.instr(instr[1:0]), .op(instr[6]), .z(r1) );
12   Inn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
13
14   assign z = r1 | r2;
15   assign w1 = r1 + 1;
16   assign w2 = r2 << 1;
17
18   top t (.instr({w1, w2, w1==w2}), .op(z), .z(r3));
19
20   assign op = r1 ^ r2 ^ r3;
21
22
23 endmodule
```

The code is not syntactically correct.

Explanation:

- Modules cannot be instantiated recursively.
- 'r1' and 'r2' have to be declared as 'wire's.
- The module 'Inn3r' does not have ports named 'instr' and 'z'. Those need to be changed to 'd' and 's', respectively.
- Output 'z' is driven by two circuits: lines 14 and 18.
- The output signal 'z' has to be declared as a 'wire' but not 'reg'.

(d) Does the following code correctly implement a counter that counts down from 10 to 1 (e.g., 10, 9, 8, ..., 2, 1, 10, 9, ...)? If so, say "Correct". If not, correct the code with minimal modification.

```
module the_final_count_down (clk, count);
  wire clk;
  reg[3:0] count = 10;
  reg[3:0] count_next;

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


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

Answer and concise explanation:

No, the implementation is not correct.

Explanation:
The correct implementation:

```
module the_final_count_down (clk, count);
  wire clk;
  reg[3:0] count = 10;
  reg[3:0] count_next;

  always @ * begin
    //count_next <= count;
    if(count != 1)
      count_next <= count - 1;
    else
      count_next <= 10;
  end

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

(e) Which of the combinational logic blocks does the following verilog code implement?

```
module mystery(input select, input enable, output result);
        wire [3:0] result;
        wire [1:0] select;
        wire enable;

        assign result = enable << (select);
endmodule
```

A 2:4 decoder.

Explanation:
The code basically shifts *enable* by *select* positions. For example, if *enable* is 1 and *select* is 10, *result* will be *0100*.

3 Verilog (III)

Please answer the following questions about Verilog.

3.1 Blocking vs. Non-Blocking Assignments

(a) What is the difference between a blocking and a non-blocking assignment?

Blocking assignments execute in series, so the results of one statement may affect the results of the next statement.
Nonblocking assignments execute in parallel, so the results of one statement do not affect the results of another statement.

3.2 Verilog Synthesis

For each circuit that results from the following code segments, select and write all applicable relevant words from the **word bank** below. If there are any syntactical or semantic issues in a code sequence, list them all in the code blocks' corresponding answer box.

word bank: *asynchronous, synchronous, active-low, active-high, reset, D Flip-Flop, sequential, combinational, inferred latch, trimmed signal, multiple drivers, race condition, tri-state logic*

(a) Code Block 1

```
  module A (input clk, input rst, input [2:0] d, output wire [1:0] q)
    always @ (posedge clk or negedge rst) begin
          if (!rst) q <= 0;
          else q <= d;
    end
  endmodule
```

Does not compile.
q needs to be instantiated as a reg. module needs to end with a semi-colon.

(b) Code Block 2

```
module B (input clk, input rst, input [1:0] d, input m, output reg [1:0] q);
  always @ (*) begin
        q[1] = d[1];
        if (m)
          q[0] = 0;
  end
endmodule
```

sequential circuit, inferred latch.

Explanation:
This code results in an asynchronous sequential circuit, as all the left-hand side signals are not assigned in every possible condition. For example, `q[0]` is not assigned when mask signal equals to zero.

(c) Code Block 3

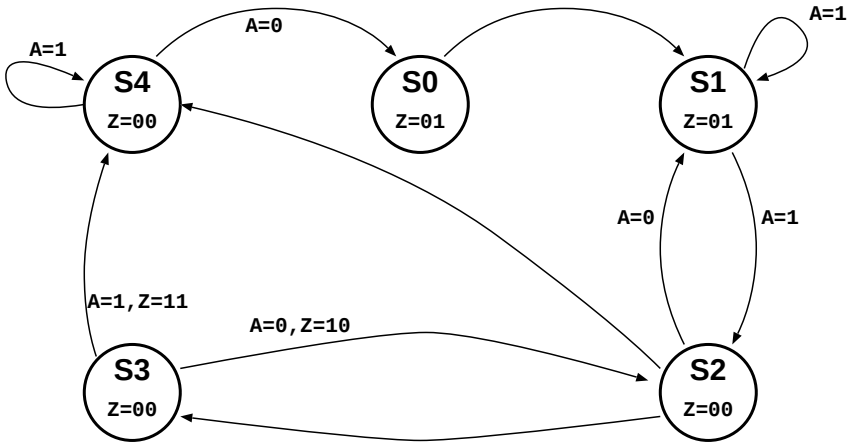
```
module C (input clk, input rst, input d, output reg q);
  always @ (posedge clk or negedge rst) begin
        if (!rst) q <= 0;
        else q <= d;
  end
endmodule
```

Sequential circuit, D Flip-Flop with asynchronous active-low reset.

4 Finite State Machines (FSM) (I)

This question has three parts.

(a) An engineer has designed a deterministic finite state machine with a one-bit input (A) and a two-bit output (Z). He started the design by drawing the following state transition diagram:



Although the exact functionality of the FSM is not known to you, there are **at least three mistakes** in this diagram. Please list **all** the mistakes.

There are four problems with this diagram

(a) Most states have a Moore labelling (output state in the bubble), one has a Mealy type labelling (output given with input transitions)

(b) There are two different transitions both with $A = 1$ from state $S1$. What will happen with $A = 0$ is missing

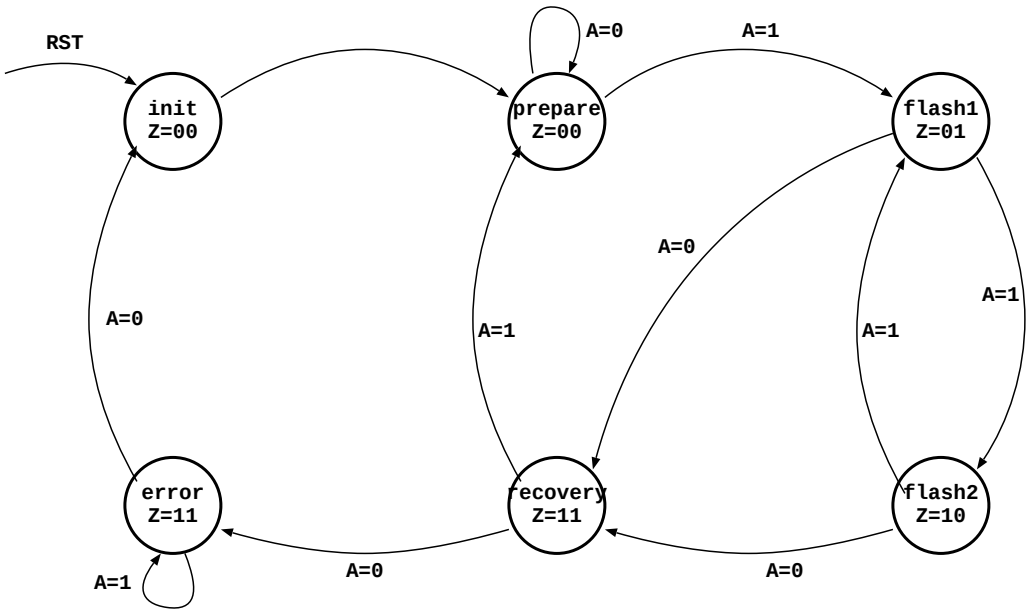
(c) There are two different transitions from state $S2$, without labeling which input triggers them

(d) There is no reset state

(b) After learning from his mistakes, your colleague has proceeded to write the following Verilog code for a much better (and **different**) FSM. The code has been verified for syntax errors and found to be OK.

```
1  module fsm (input CLK, RST, A, output [1:0] Z);
2
3    reg [2:0] nextState, presentState;
4
5    parameter start   = 3'b000;
6    parameter flash1 = 3'b010;
7    parameter flash2 = 3'b011;
8    parameter prepare   = 3'b100;
9    parameter recovery = 3'b110;
10   parameter error = 3'b111;
11
12   always @ (posedge CLK, posedge RST)
13      if (RST)  presentState <= start;
14      else      presentState <= nextState;
15
16   assign Z = (presentState == recovery) ? 2'b11 :
17              (presentState == error)    ? 2'b11 :
18              (presentState == flash1)   ? 2'b01 :
19              (presentState == flash2)   ? 2'b10 : 2'b00;
20
21   always @ (presentState, A)
22     case (presentState)
23       start    : nextState <= prepare;
24       prepare  : if (A) nextState <= flash1;
25       flash1   : if (A) nextState <= flash2;
26                  else   nextState <= recovery;
27       flash2   : if (A) nextState <= flash1;
28                  else   nextState <= recovery;
29       recovery : if (A) nextState <= prepare;
30                  else   nextState <= error;
31       error    : if (~A) nextState <=start;
32       default  : nextState <= presentState;
33     endcase
34
35 endmodule
```

Draw a proper state transition diagram that corresponds to the FSM described in this Verilog code.

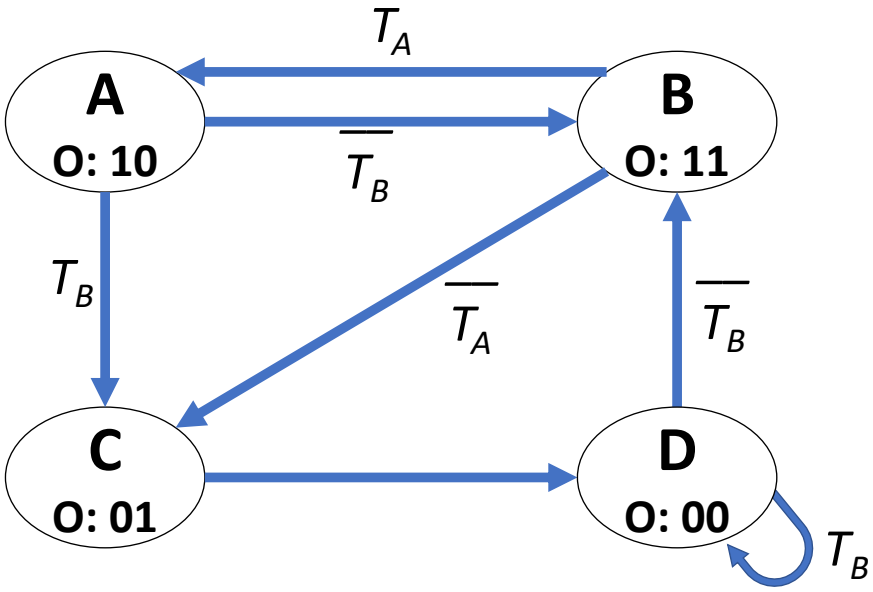


(c) Is the FSM described by the previous Verilog code a Moore or a Mealy FSM? Why?

Moore, the output Z only depends on the state ($presentState$) and not on the input (A).

5 Finite State Machines (FSM) (II)

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.

The reset line or indication for initial state.

(b) What kind of an FSM is this?

Moore

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

One-hot: reduces next-state logic Binary: reduces FFs to hold state Output: reduces output logic

(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:

State assignments: A: 0001, B: 0010, C: 0100, D: 1000
NS[3] = TB * TS[3] + TS[2]
NS[2] = TB * TS[0] + $\overline{TA}$ * TS[1]
NS[1] = $\overline{TB}$ * (TS[0] + TS[3])
NS[0] = TS[1] * TA
O[1] = TS[0] + TS[1]
O[0] = TS[1] + TS[2]

(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:

State assignments: A: 00, B: 01, C: 10, D: 11
NS[1] = $\overline{TS[1]}$ * ($\overline{TS[0]}$ * TB + TS[0] $\overline{TA}$) + TS[1] * ($\overline{TS[0]}$ + TS[0] * TB)
NS[0] = $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$ + TS[1]
O[1] = TS[1]
O[0] = TS[1] XOR TS[0]

(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:

State assignments: A: 10, B: 11, C: 01, D: 00
NS[1] = TS[1] * $\overline{TS[0]}$ * $\overline{TB}$ + TS[1] * TS[0] * TA + $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$
= $\overline{TS[0]}$ * $\overline{TB}$ + TS[1] * TS[0] * TA
NS[0] = TS[1] * $\overline{TS[0]}$ + TS[1] * TS[0] * $\overline{TA}$ + $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$
= TS[1] * ($\overline{TS[0]}$ + TS[0] * $\overline{TA}$) + $\overline{TS[1]}$ * $\overline{TS[0]}$ * $\overline{TB}$
O[1] = TS[1]
O[0] = TS[0]

(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?

one-hot: 10 logic gates, 4 FFs
binary: 16 logic gates, 2 FFs
output: 10 logic gates, 2 FFs

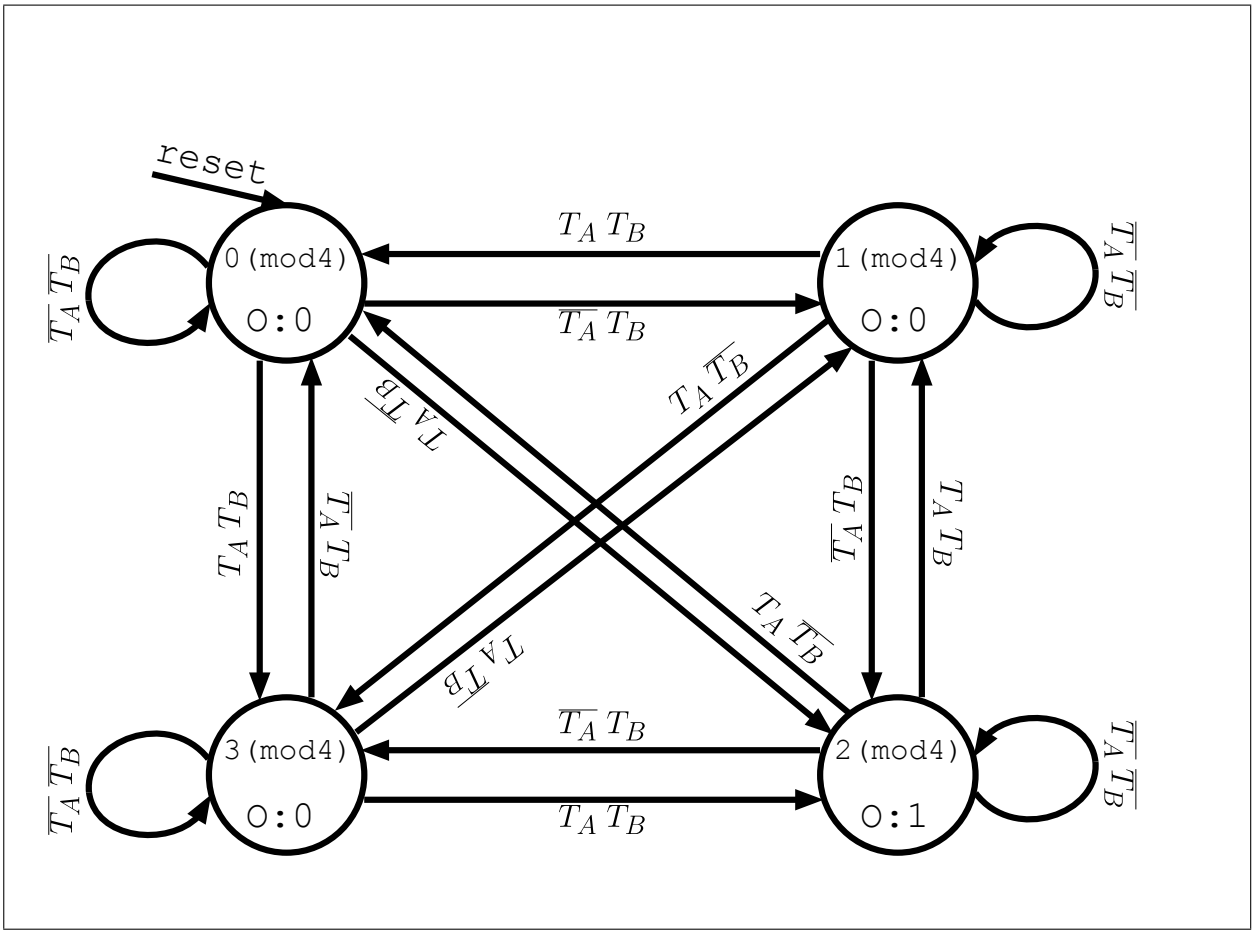
Output encoding has the least amount of circuitry elements.

6 Finite State Machines (FSM) (III)

You are given two *one-bit* input signals (T_A and T_B) and one *one-bit* output signal (O) for the following modular equation: $2N(T_A) + N(T_B) \equiv 2 \pmod 4$. In this modular equation, $N(T_A)$ and $N(T_B)$ represent the **total number of times** the inputs T_A and T_B are high (i.e., logic 1) at each positive clock edge, respectively. The one-bit output signal, O, is set to 1 when the modular equation is satisfied (i.e., $2N(T_A) + N(T_B) \equiv 2 \pmod 4$), and 0 otherwise. An example that sets $O = 1$ at the end of the fourth cycle would be:

- (1^{st} cycle) $T_A = 0$ ($N(T_A) = 0$), $T_B = 0$ ($N(T_B) = 0$), $2N(T_A) + N(T_B) \equiv 0 \pmod 4 \Rightarrow O = 0$
- (2^{nd} cycle) $T_A = 1$ ($N(T_A) = 1$), $T_B = 1$ ($N(T_B) = 1$), $2N(T_A) + N(T_B) \equiv 3 \pmod 4 \Rightarrow O = 0$
- (3^{rd} cycle) $T_A = 1$ ($N(T_A) = 2$), $T_B = 0$ ($N(T_B) = 1$), $2N(T_A) + N(T_B) \equiv 1 \pmod 4 \Rightarrow O = 0$
- (4^{th} cycle) $T_A = 0$ ($N(T_A) = 2$), $T_B = 1$ ($N(T_B) = 2$), $2N(T_A) + N(T_B) \equiv 2 \pmod 4 \Rightarrow O = 1$

(a) You are given a partial **Moore** machine state transition diagram that corresponds to the modular equation described above. However, the input labels of most of the transitions are still missing in this diagram. Please label the transitions with the correct inputs so that the FSM correctly implements the above specification.



(b) Describe the FSM with Boolean equations assuming that the states are encoded with **one-hot encoding**. Assign state encodings while using the **minimum** possible number of bits to represent the states. Please indicate the values you assign to each state.

State assignments: 0 (mod 4): 0001, 1 (mod 4): 0010, 2 (mod 4): 0100, 3 (mod 4): 1000
CS denotes current states, and NS denotes next states.

$NS[0] = CS[0]\ \overline{T_A}\ \overline{T_B} + CS[1]\ T_A\ T_B + CS[2]\ T_A\ \overline{T_B} + CS[3]\ \overline{T_A}\ T_B$
$NS[1] = CS[1]\ \overline{T_A}\ \overline{T_B} + CS[2]\ T_A\ T_B + CS[3]\ T_A\ \overline{T_B} + CS[0]\ \overline{T_A}\ T_B$
$NS[2] = CS[2]\ \overline{T_A}\ \overline{T_B} + CS[3]\ T_A\ T_B + CS[0]\ T_A\ \overline{T_B} + CS[1]\ \overline{T_A}\ T_B$
$NS[3] = CS[3]\ \overline{T_A}\ \overline{T_B} + CS[0]\ T_A\ T_B + CS[1]\ T_A\ \overline{T_B} + CS[2]\ \overline{T_A}\ T_B$
$O[0] = CS[2]$

(c) Describe the FSM with Boolean equations assuming that the states are encoded with **binary encoding** (i.e., fully encoding). Assign state encodings while using the **minimum** possible number of bits to represent the states. Please indicate the values you assign to each state.

State assignments: 0 (mod 4): 00, 1 (mod 4): 01, 2 (mod 4): 10, 3 (mod 4): 11
CS denotes current states, and NS denotes next states.
$NS[0] = \overline{CS[0]}\ T_B + CS[0]\ \overline{T_B}$
$NS[1] = CS[0]\ (CS[1]\ \text{XOR}\ T_A\ \text{XOR}\ T_B) + \overline{CS[0]}\ (T_A\ \text{XOR}\ CS[1])$
$O[0] = CS[1]\ \overline{CS[0]}$

(d) Consider an implementation of the FSM assuming that the states are encoded with **output encoding**. What is the **minimum** number of bits required to encode the states with output encoding?

A minimum of three bits are required to represent the state 2 (mod 4) uniquely so that the output logic layer is minimized. If we consider that the state assignments are as follows:
0 (mod 4): 000, 1 (mod 4): 001, 2 (mod 4): 100, 3 (mod 4): 011
Then the output logic would be:
$O[0] = CS[2]$
There is no way of achieving this minimization using two bits, as we did with fully encoding, because $CS[1] = 1$ for both states 2 (mod 4) and 3 (mod 4). Since only 2 (mod 4) should set $O = 1$, we would need to make sure that $CS[0] = 0$ while $CS[1] = 1$. Thus, it is impossible to have the same minimization as we did for output encoding with three-bit representation of the states.

7 Finite State Machines (FSM) (IV)

7.1 Mealy Machine and Moore Machine

Figure 1 depicts a Mealy state machine corresponding to a digital circuit design that receives *one input* and produces *one output*. All state transitions in the diagram are labelled with the corresponding *input/output* values. Answer the following questions for this state diagram.

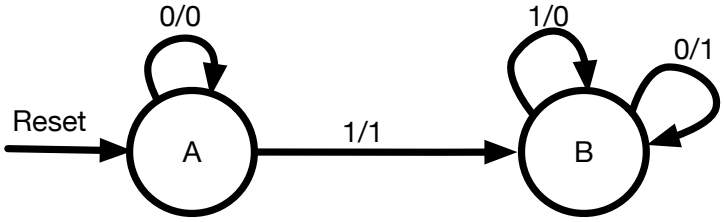
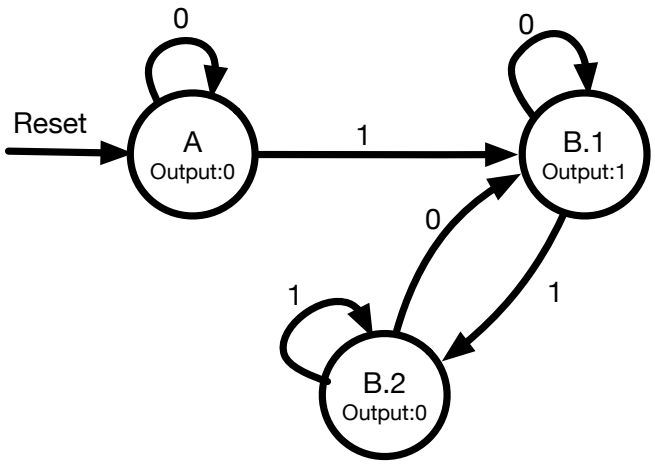


Figure 1: A Mealy Machine.

(a) Can this Mealy machine be converted to its equivalent Moore machine? If so, please convert it to its equivalent Moore machine. Draw the converted Moore machine. If not, explain why it is not possible.

Yes, any Mealy FSM can be converted to a corresponding Moore machine.
For each Mealy FSM, its outputs depend on the current state and the inputs; For a Moore FSM, its outputs depend only on the current state.
Therefore, the equivalent Moore machine is:



(b) Assume the state machine in Figure 1 is used to process binary numbers, from their least significant bit to their most significant bit. You observe an output bit stream from this FSM, as shown in Figure 2.

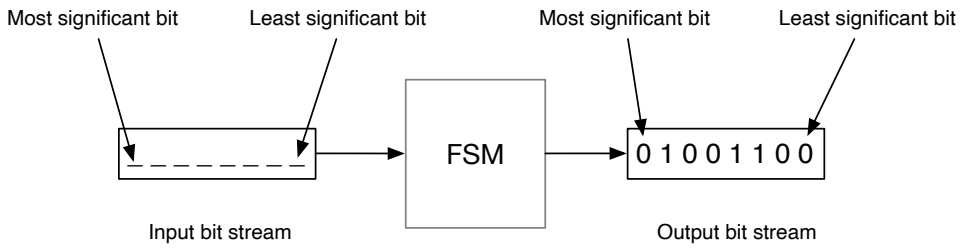


Figure 2: Usage of the Mealy machine to process a bit stream.

What was the input bit stream supplied to this FSM? Show your work.

"10110100"
When processing a bit stream from the least significant bit to the most significant bit, this state machine keeps the bits unchanged until the first "1" comes. After the first "1" is observed, all subsequent bits are flipped (not including the first "1"). The input is therefore "10110100".

7.2 Designing an FSM

Design a Moore finite state machine (FSM) with one input and one output. The input provides an unsigned binary number in a bit-serial manner, from the most-significant bit to the least-significant bit. The output should be logic-1 in a clock cycle if the provided input made up of all the bits received so far is divisible by 3 (i.e., [the input number] `mod` $3 = 0$). (Hint: Recall that the output depends only on the current state in a Moore FSM.)

Below are some example bit-streams that should output logic-1.

- 11
- 110
- 1001
- 1100
- 1111

Draw the state diagram and explain why it works. Your state machine should use as few states as possible and each state should have a comprehensive definition.

When a number is divided by 3, the remainder takes one of the following three values:

- The remainder equals 0.
- The remainder equals 1.
- The remainder equals 2.

When the remainder equals 0 ($r = 0$), the binary string is divisible by 3. We define this state as S_0. Analogously, we define S_1 and S_2 for remainder values $r = 1$ and $r = 2$. We can then design the following Moore machine, whose output O is "1" for state S_0 and "0" for all other states: