ECE 3544: Digital Design 1
Homework Assignment 5 (90 points)

When writing a Verilog module or naming a file, replace YOURPID with your Virginia Tech PID. Name the file according to the module name that you use.

For each design that you represent in a Verilog module, copy your source code into the document. For each design that you simulate in ModelSim, include waveforms displaying the correct operation of each module

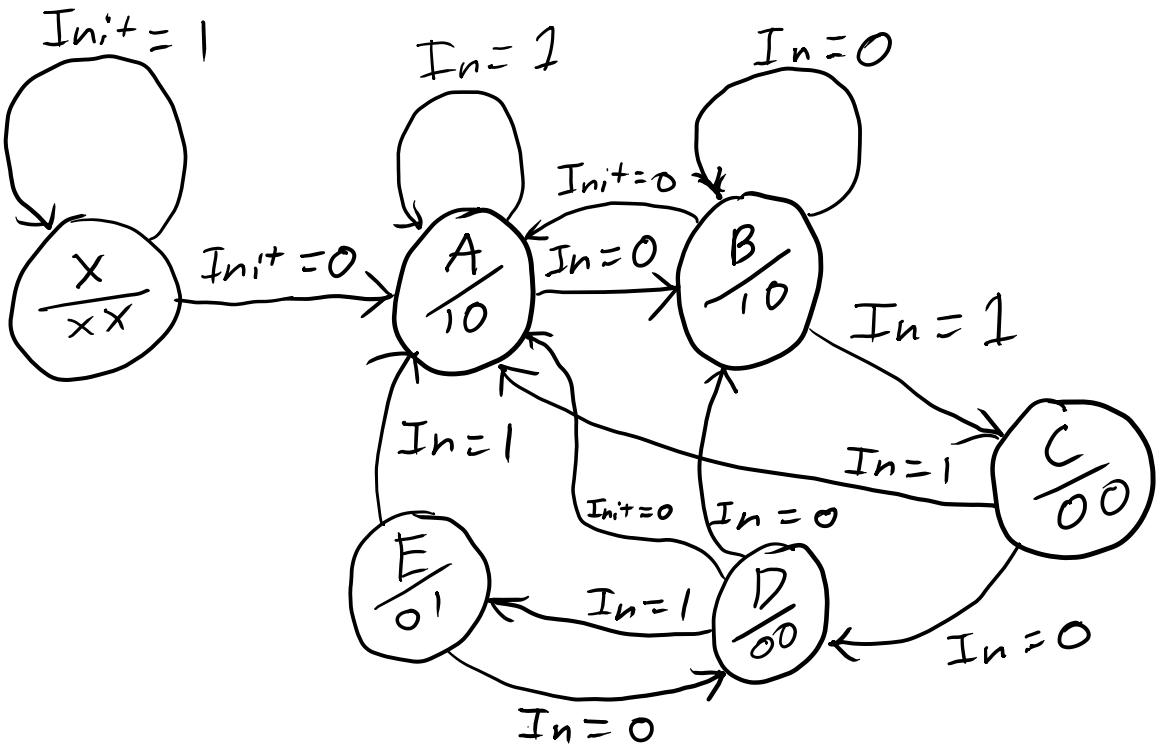
In addition, submit the .v file containing each design and each test bench that you write. Your files should contain header information as described in Project 1 and be neatly formatted and commented. Submit your files separately on Canvas. Do not put them into an archive. *Make sure that you upload all of your files.*

Helpful Hints for Implementing Verilog Models for Finite State Machines
Read the documents "Comparing State Machine Modeling Techniques" and "Modeling Synchronous Finite State Machines" – they are available on the Canvas site!

Problem 1 (10 points)

Draw the state diagram that is the basis for the finite state machine described by the following Verilog behavioral model. *Make sure that you label the initial state.*

```
module problem1(clock, init, in, out);
   input        clock;     // System clock
   input        init;      // Asynchronous active-low init
   input        in;        // FSM input
   output [1:0] out; // FSM output

   reg    [2:0] state;     // FSM state
   reg    [1:0] out; // The output is the target of a procedure.

// Parameters define the FSM states.
   parameter sA = 3'b000, sB = 3'b001, sC = 3'b010, sD = 3'b011, sE = 3'b100;

// State machine
   always@(posedge clock or negedge init) begin
      if(init == 1'b0)
         state <= sA;
      else begin
         case(state)
            sA:      if(in == 1'b0)
                        state <= sB;
            sB:      if(in == 1'b1)
                        state <= sC;
            sC:      if(in == 1'b0)
                        state <= sD;
                     else
                        state <= sA;
            sD:      if(in == 1'b0)
                        state <= sB;
                     else
                        state <= sE;
            sE:      if(in == 1'b0)
                        state <= sD;
                     else
                        state <= sA;
            default: state <= 3'bxxx;
         endcase
      end
   end

// Output machine
   always@(state) begin
      case(state)
         sA:      out = 2'b10;
         sB:      out = 2'b00;
         sC:      out = 2'b00;
         sD:      out = 2'b00;
         sE:      out = 2'b01;
         default: out = 2'bxx;
      endcase
   end

endmodule
```


Init = 1
In = 1
In = 0
Init = 0
Init = 0
In = 0
In = 1
In = 1
In = 1
Init = 0
In = 0
In = 1
In = 0
In = 0
X
xx
A
10
B
10
C
00
D
00
E
01

Problem 2 (15 points)
A state machine has a two-bit input `in[1:0]` and an output `out`.

For reference, here is a Verilog module and port declaration that corresponds to the system being described:

```
module state_machine(clock, init, in, out);
   input         clock
   input         init;
   input  [1:0]  in;
   output        out;
```

The state machine uses an asynchronous active-low INIT to send the state machine to its initial state from any state when the INIT is asserted. In the initial state, `out` = 0.

In this state machine, all output changes (aside from those caused by asserting INIT) occur in synch with the clock. Specifically, the synchronous output changes occur in response to the following input *sequences*. As just indicated, the output should only change on the clock trigger following the second two-bit value in the sequence.

- The clocked input sequence in[1:0] = 01, 00 causes the output to become 0.
- The clocked input sequence in[1:0] = 10, 00 causes the output to become 1.
- The clocked input sequence in[1:0] = 11, 00 causes the output to toggle (complement) its value.

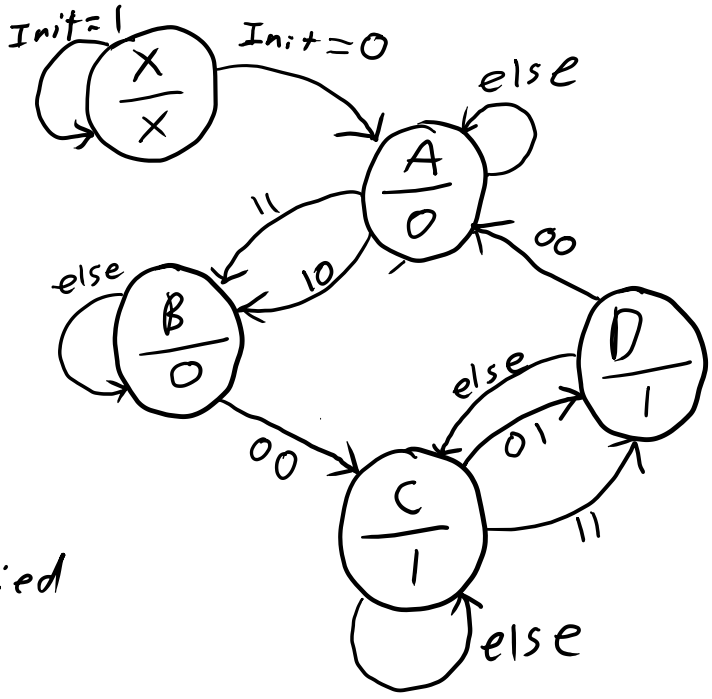
a. Draw the state diagram that models the specification. Show the state table that corresponds to the state diagram.
b. Demonstrate using the partitioning method that your model contains the minimal number of states, or use it to derive the minimal model if your original model is not already minimal. If your original state table is not minimal, you should show the minimal state table, but you need not redraw the diagram.

NOTE: This state diagram/table assumes INIT = 1 for all state values other than X, else each next state transition would be to state A.

a)

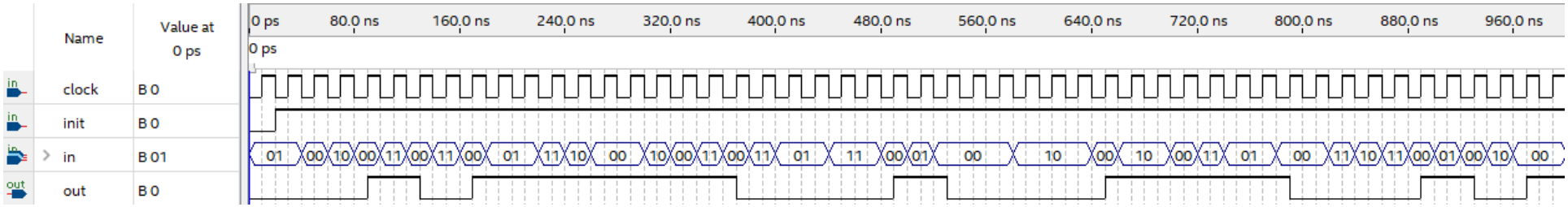
PS	in[1:0] 00	01	10	11	Z
A	A	A	B	B	0
B	C	B	B	B	0
C	C	D	C	D	1
D	A	C	C	C	1

NS



b) States cannot be simplified

Here is a waveform showing the behavior of the circuit from Problem 2. It does not show all possible input sequences, but it is representative of the system's behavior.



Problem 3 (10 points)
Use the partitioning method to reduce each state table to one having a minimum number of states.

a.

PS	x_1x_2 = 00	01	11	10	z
A	A	B	D	C	0
B	D	A	F	E	1
C	E	B	B	E	0
D	B	C	F	A	1
E	C	D	D	A	0
F	F	E	A	C	1

NS

Equivalent States:

$A \equiv C \equiv E$

$B \equiv D$

PS	x_1x_2 = 00	01	11	10	z
A	A	B	B	A	0
B	B	A	C	A	1
C	C	A	A	A	1

NS

b.

PS	x = 0	1
A	B / 1	C / 0
B	D / 1	E / 0
C	F / 1	E / 1
D	G / 1	F / 0
E	D / 1	B / 0
F	H / 0	I / 1
G	H / 1	B / 1
H	F / 0	I / 1
I	A / 0	A / 1

NS / z

Equivalent states:

$B \equiv E$

$C \equiv G$

$F \equiv H$

PS	x = 0	1
A	B/1	C/0
B	D/1	B/0
C	F/1	B/1
D	C/1	F/0
F	F/0	I/1
I	A/0	A/1

In Problems 4, 5, and 6, you must write Verilog code to implement the state machine you modeled in Problem 2. Each problem requires a different approach to the implementation. For Problems 4 and 5 I have included a Verilog module for a D flip-flop with the assignment materials. Study the behavioral description of the flip-flop before you use it. Make certain that you assign appropriate values to the D flip-flop's initializing inputs when you instantiate the flip-flop.

You may show the test results for all three implementations in the same test bench, but you should verify each implementation before you proceed to the next one.

Problem 4 (10 points)
Implement the state machine using a dense state assignment and D flip-flops. The module you write should instantiate the proper number of flip-flops to implement a dense state assignment. Use continuous assignments for the D flip-flop input logic and the output logic.

Use the following model for your module declaration:

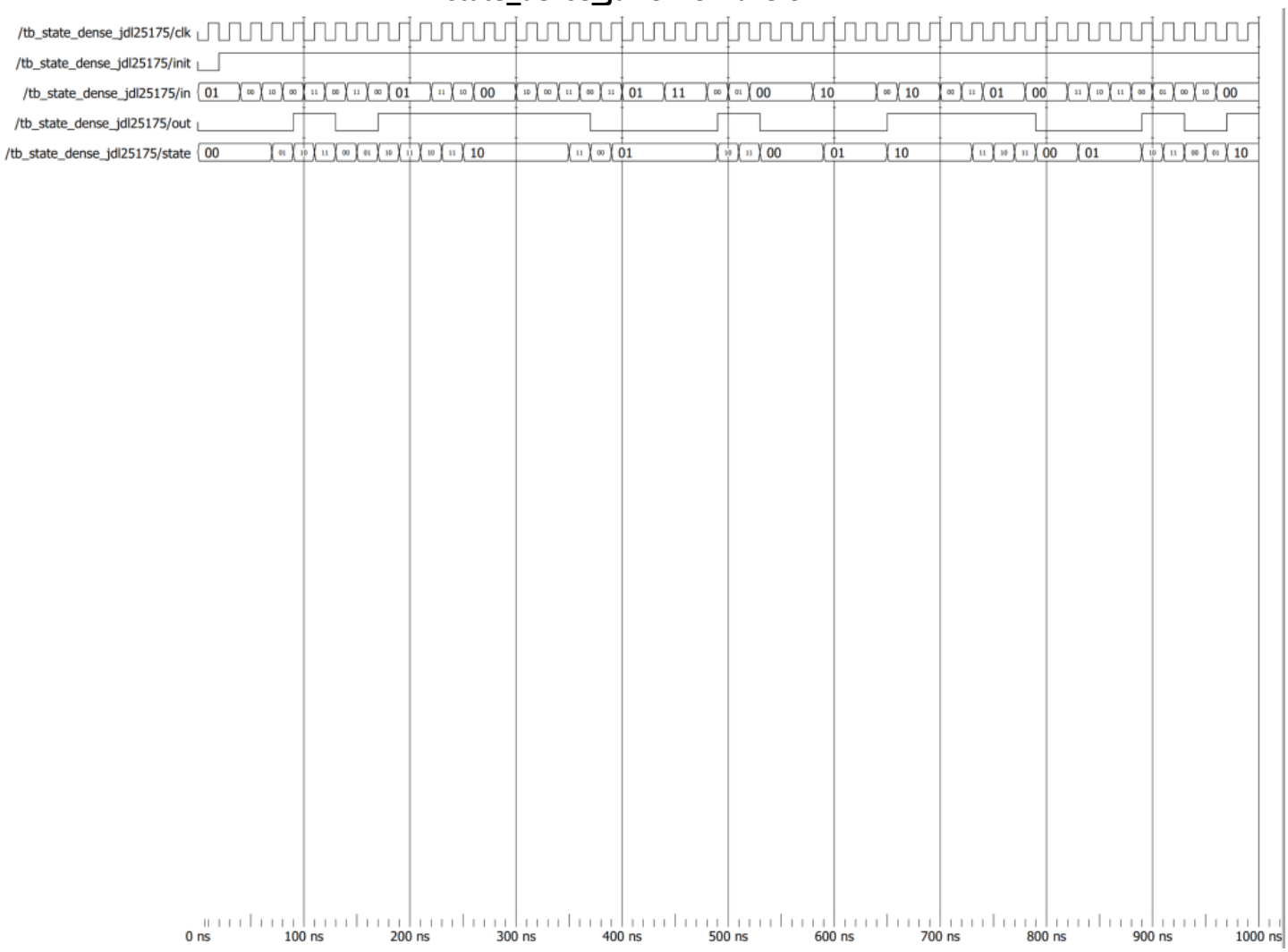
```
module state_dense_YOURPID(clock, init, in, out, state);
   input         clock;   // System clock
   input         init;    // Asynchronous active-low init
   input  [1:0] in;       // System (two-bit) input
   output        out;     // System output
   output [?:0] state;    // System state
```

This is important!

- The question mark is not an element of Verilog syntax. It's me not telling you how many state variables you should have. Your state assignment method should provide you with the information that you need to stick a number in that spot.
- It's not necessary to have the state as an output. I am having you put it there to make a good habit of looking at the state when you want to debug a state machine.

```
module state_dense_jdl25175(clock, init, in, out, state);
   input        clock;   // System clock
   input        init;    // Asynchronous active-low init
   input  [1:0] in;      // System (two-bit) input
   output       out;     // System output
   output [1:0] state;   // System state

   wire  [1:0] newState;

   dff_hw5_set_reset D1(clock, ~init, 1'b0, newState[1], state[1]), D2(clock, ~init, 1'b0, newState[0], state[0]);

   assign newState = (state == 2'b00 && (in == 2'b11 || in == 2'b10)) ? 2'b01:
                     (state == 2'b01 && in == 2'b00) ? 2'b10:
                     (state == 2'b01) ? 2'b01:
                     (state == 2'b10 && (in == 2'b11 || in == 2'b01)) ? 2'b11:
                     (state == 2'b11 && in == 2'b00) ? 2'b00:
                     (state == 2'b11) ? 2'b10 : state;

   assign out = (state == 2'b00) ? 0:
                (state == 2'b01) ? 0:
                (state == 2'b10) ? 1:
                (state == 2'b11) ? 1:
                1'bx;

endmodule
```

state_dense_jdl25175 waveform



Problem 5 (10 points)
Implement the state machine using a one-hot encoding for the state assignment and D flip-flops. The module you write should instantiate the proper number of flip-flops to implement a one-hot state assignment. Use continuous assignments for the D flip-flop input logic and the output logic.

Use the following model for your module declaration:

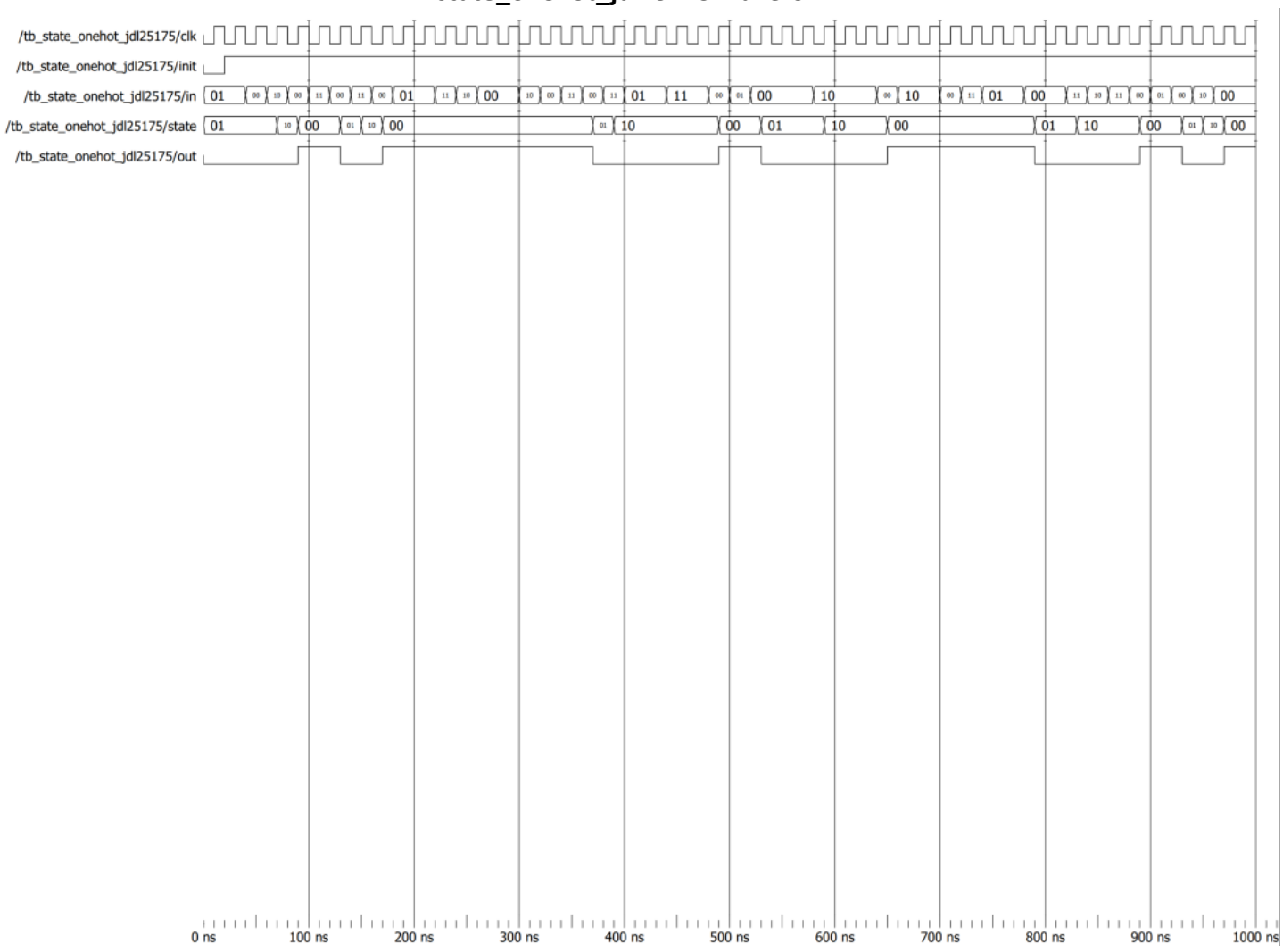
```
module state_onehot_YOURPID(clock, init, in, out, state);
   input         clock;   // System clock
   input         init;    // Asynchronous active-low init
   input  [1:0] in;       // System (two-bit) input
   output        out;     // System output
   output [?:0] state;    // System state
```

This is important!

- When using a one-hot state code, every state – including the initial state – is a one-hot code. Therefore, you must use the same init to **set** the initial state's hot bit that you use to **reset** the bits of the initial state that are not hot.

```
module state_onehot_jdl25175(clock, init, in, out, state);
   input         clock;  // System clock
   input         init;   // Asynchronous active-low init
   input  [1:0] in;     // System (two-bit) input
   output        out;    // System output
   output [3:0] state;  // System state

   wire [3:0] newState;

   dff_hw5_set_reset D1(clock, ~init, 1'b0, newState[3], state[3]), D2(clock, ~init, 1'b0, ne
wState[2], state[2]),
                     D3(clock, ~init, 1'b0, newState[1], state[1]), D4(clock, 1'b0, ~init, ne
wState[0], state[0]);

   assign newState = (state[0] == 1'b1 && (in == 2'b11 || in == 2'b10)) ? 4'b0010:
                     (state[1] == 1'b1 && in == 2'b00) ? 4'b0100:
                     (state[1] == 1'b1) ? 4'b0010:
                     (state[2] == 1'b1 && (in == 2'b11 || in == 2'b01)) ? 4'b1000:
                     (state[3] == 1'b1 && in == 2'b00) ? 4'b0001:
                     (state[3] == 1'b1) ? 4'b0100 : state;

   assign out = (state[0] == 1'b1) ? 0:
                  (state[1] == 1'b1) ? 0:
                  (state[2] == 1'b1) ? 1:
                  (state[3] == 1'b1) ? 1:
                  1'bx;
endmodule
```

state_onehot_jdl25175 waveform



Problem 6 (10 points)
Implement the state machine using the behavioral model template described in class.

Use the following model for your module declaration:

```
module state_procedural_YOURPID(clock, init, in, out, state);
   input         clock;   // System clock
   input         init;    // Asynchronous active-low init
   input  [1:0] in;      // System (two-bit) input
   output        out;     // System output
   output [?:0] state;   // System state
```

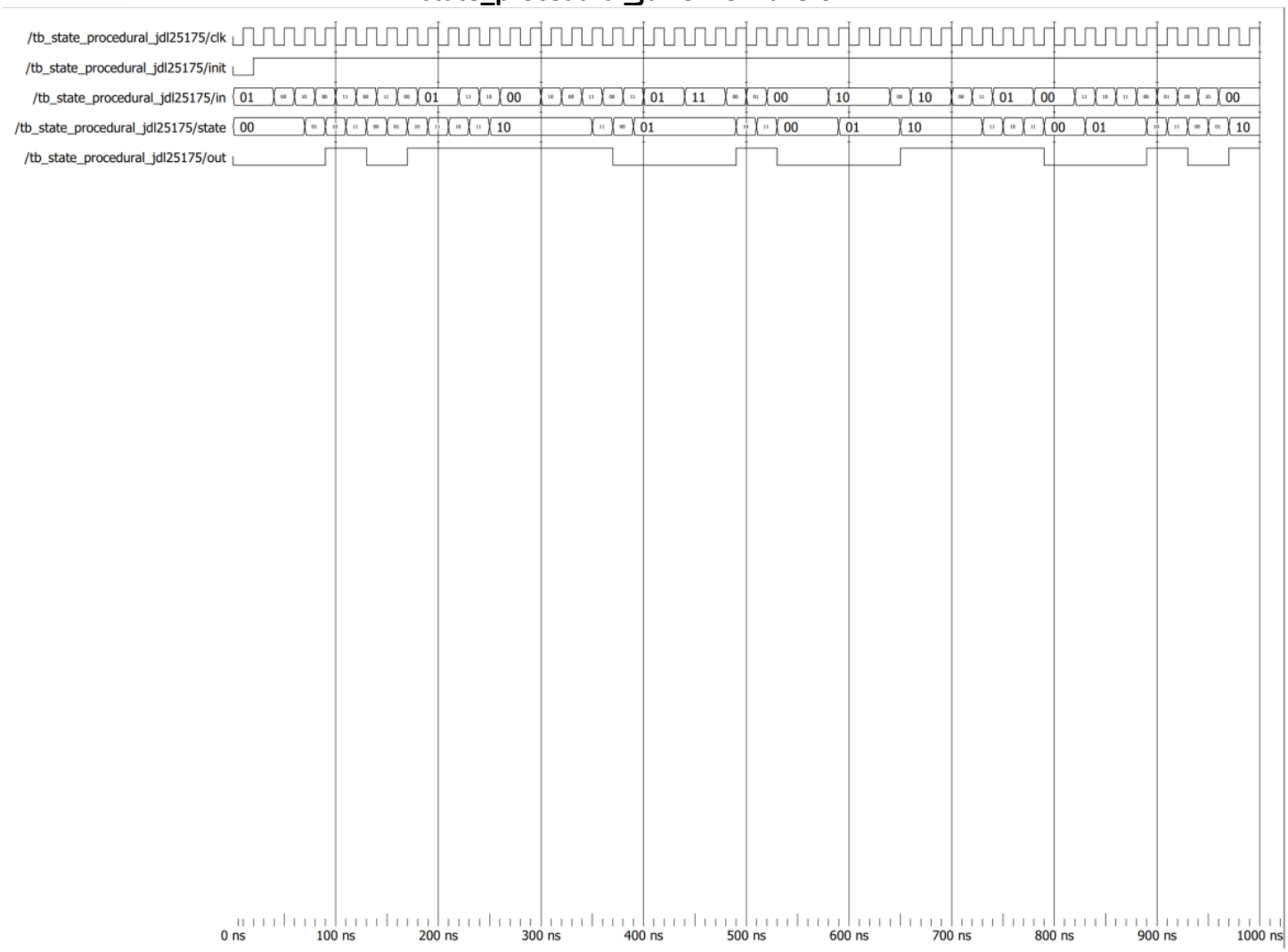
This is important!

- Remember, in this procedural model you will be targeting certain entities for change via procedure. That means you must declare them as regs even if they are already declared as outputs.
- For this implementation, use parameters for the state names in the manner shown in Program 12-26 on page 651 of the text.

```
module state_procedural_jdl25175(clock, init, in, out, state);
   input        clock;   // System clock
   input        init;    // Asynchronous active-low init
   input  [1:0] in;              // System (two-bit) input
   output reg   out;     // System output

   output reg [1:0] state;       // System state
   reg [1:0] nextState;

    always @(posedge clock or negedge init) begin
        if(init == 1'b0)
            state <= 2'b00;
        else
            state <= nextState;
    end

    always @(in or state) begin
        if(state == 2'b00) begin
            if(in == 2'b11 || in == 2'b10)
                nextState = 2'b01;
            else
                nextState = state;
        end
        else if(state == 2'b01) begin
            if(in == 2'b00)
                nextState = 2'b10;
            else
                nextState = state;
        end
        else if(state == 2'b10) begin
            if(in == 2'b11 || in == 2'b01)
                nextState = 2'b11;
            else
                nextState = state;
        end
        else if(state == 2'b11) begin
            if(in == 2'b00)
                nextState = 2'b00;
            else
                nextState = 2'b10;
        end
        else
            nextState = state;
    end
```

```
    always @(state) begin
        if(state == 2'b00)
            out = 1'b0;
        else if(state == 2'b01)
            out = 1'b0;
        else if(state == 2'b10)
            out = 1'b1;
        else if(state == 2'b11)
            out = 1'b1;
        else
            out = 1'bx;
    end

endmodule
```

state_procedural_jdl25175 waveform



Problem 7 (25 points)

A coin-operated pay phone charges twenty-five cents for a phone call. (It's visiting town from the 1980s.) It has a coin slot that accepts nickels, dimes, and quarters. It has a coin return lever that will cause the phone to dump the coins that the user deposited into it prior to making a call. The phone returns change by releasing dimes and nickels; to speed up change dispensing, it releases dimes from two different columns.

Create a model for a synchronous finite state machine that models the pay phone.

The finite state machine has six inputs. Besides the clock and the init, the remaining inputs reflect to input events that affect the FSM state.

- CLOCK: The system clock.
- INIT: An asynchronous active-low input. It returns the state machine to the initial state from any state.
- NICKEL_IN: Represents the user depositing a nickel.
- DIME_IN: Represents the user depositing a dime.
- QUARTER_IN: Represents the user depositing a quarter.
- COIN_RELEASE: Represents the user pulling the coin release lever.

To give an example, if the user deposits a nickel, NICKEL_IN will equal 1 for one clock period. If the user asserts the coin release, COIN_RELEASE will equal 1 for one clock period.

The finite state machine has five outputs, each of which corresponds to an output event.

- MAKE_CALL: Represents a situation where enough money has been deposited to make a phone call. This output equals 1 in the state where the user has deposited at least 25 cents.
- NICKEL_OUT: Represents a situation where a nickel is being dropped *as change*.
- DIME_OUT1: Represents a situation where a dime is being dropped *as change*.
- DIME_OUT2: Represents a situation where a second dime must be dropped *as change*, so that both can be dropped in the same state.
- COIN_DUMP: Represents a situation where the deposited coins are returned to the user. When COIN_DUMP = 1, *don't assert the nickel and dime outputs – a coin dump does not return change in the same way that a successful call does.*

To give examples:

- In the state reached where the user has deposited 25 cents, MAKE_CALL = 1, NICKEL_OUT = 0, DIME_OUT1 = 0, DIME_OUT2 = 0, and COIN_DUMP = 0.
- In the state reached where the user has deposited 40 cents, MAKE_CALL = 1, NICKEL_OUT = 1, DIME_OUT1 = 1, DIME_OUT2 = 0, and COIN_DUMP = 0.
- In the state reached where the user asserted the coin release, MAKE_CALL = 0, NICKEL_OUT = 0, DIME_OUT1 = 0, DIME_OUT2 = 0, and COIN_DUMP = 1.

When the machine makes 10 cents or 15 cents in change, it dispenses a dime from dime_out1. When the machine makes 20 cents change, it dispenses a dime from dime_out1 and from dime_out2. *If these situations don't seem possible, your options are: a) ignore them, b) figure out why they are possible.*

The output events occur on the clock pulse following the input event that should trigger the output – that is, in conjunction with the state that results – and not concurrently with the occurrence of the input. When the state machine reaches a state that performs an output action, it should return to the initial state on the next clock pulse. As indicated above, asserting the RESET input should cause the state machine to return to the initial state from any state. Asserting the reset should not cause a coin dump.

a. Show a complete state diagram for the situation described above. Use the partitioning method to verify that the state diagram is minimal.
b. Write a Verilog model that implements the state diagram, following the template for behavioral state machines described in class. (This Verilog model should be in the style of Problem 6.) Use the following module declaration as a model:

```
module pay_phone_YOURPID(clock, init, events_in, events_out);
   input         clock;         // The system clock
   input         init;          // Asynchronous active-low
   input  [3:0] events_in;      // The input events
                                // - Bit 3: quarter_in
                                // - Bit 2: dime_in
                                // - Bit 1: nickel_in
                                // - Bit 0: coin_release
   output [4:0] events_out;     // The output events
                                // - Bit 4: make_call
                                // - Bit 3: dime_out1
                                // - Bit 2: dime_out2
                                // - Bit 1: nickel_out
                                // - Bit 0: coin_dump
```