Digital Systems

Computer assignment 5

Part a

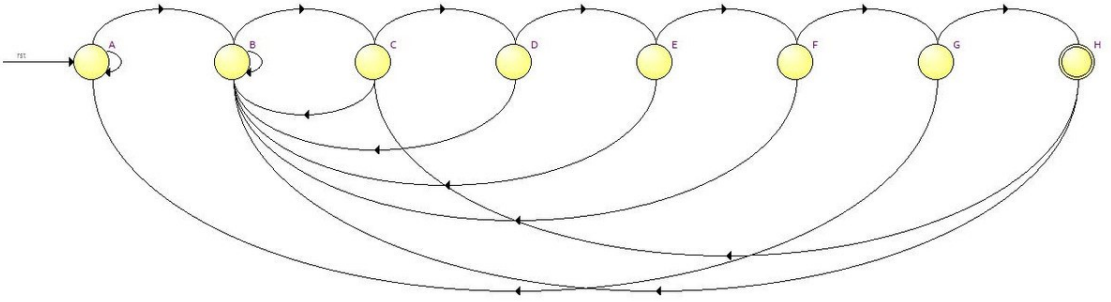
i

In this part, we are going to design a sequence detector circuit which detects `0111110`, this is the Verilog description for this circuit:

```
`timescale 1ns/1ns
module SeqDet(input clk, serIn, rst, output seqValid);
  parameter [2:0] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100,
  reg [2:0] state = A, nextState = A;

  always @(serIn, state) begin
    case (state)
      A: nextState = serIn ? A : B;
      B: nextState = serIn ? C : B;
      C: nextState = serIn ? D : B;
      D: nextState = serIn ? E : B;
      E: nextState = serIn ? F : B;
      F: nextState = serIn ? G : B;
      G: nextState = serIn ? A : H;
      H: nextState = serIn ? C : B;
      default: nextState = A;
    endcase
  end

  always @(posedge clk, posedge rst) begin
    if (rst)
      state <= A;
    else
      state <= nextState;
  end

  assign seqValid = state == H;
endmodule
```

ii

This is the state diagram of our state machine:



To test our circuit, we'll write a testbench for it:

```
`timescale 1ns/1ns
module SeqDetTB;
  reg clk = 0;
  reg serIn;
  reg rst = 0;
  wire seqValidPre, seqValidPost, diff;

  SeqDet UUT1(clk, serIn, rst, seqValidPre);
  SeqDetPost UUT2(clk, serIn, rst, seqValidPost);

  assign diff = seqValidPost ^ seqValidPre;

  initial begin
    repeat (20) #(200) clk = ~clk;
  end

  initial begin
    serIn = 0;
    #(200 * 1.5) serIn = 1;
    #(10 * 200 + 200 * 0.5) serIn = 0;
    #(200 * 2);

    $stop;
  end
```

```
endmodule
```

The number of cells after compilation:

```
Total logic elements    7 / 14,400 ( < 1 % )
Total registers         7
Total pins              4 / 81 ( 5 % )
Total virtual pins      0
Total memory bits       0 / 552,960 ( 0 % )
```

iii

After synthesizing the Verilog code using Quartus and running the testbench on two instances, we'll get the following waveform:

After entering the valid sequence, we see that there is a glitch in `diff` signal which is the `xor` product of `seqValidPre` and `seqValidPost` outputs. This means that our post-synthesis code has a delay value applied to itself through `.sdo` file.

Part b

i

The Verilog description of circuit in part b will be like this:

```
`timescale 1ns/1ns
module GetInputNumber(input iz_cnt, cen, shen, clk, rst, serin, output co, out
```

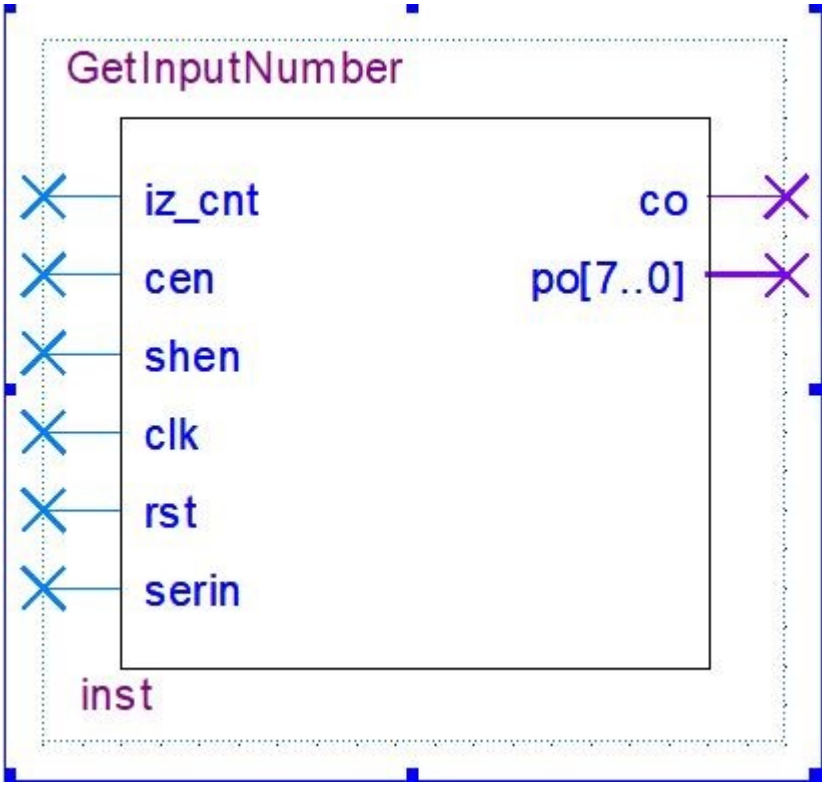
```
    reg [2:0] cnt_po = 3'b0;
    always @(posedge clk, posedge rst) begin
        if (rst)
            cnt_po <= 3'b0;
        else if (iz_cnt)
            cnt_po <= 3'b0;
        else begin
            if (cen)
                cnt_po <= cnt_po + 1;
            else
                cnt_po <= cnt_po;
        end
    end

    assign co = &{cnt_po};

    reg [7:0] shf_po = 8'b0;
    always @(posedge clk, posedge rst) begin
        if (rst)
            shf_po <= 8'b0;
        else begin
            if (shen)
                shf_po <= {serin, shf_po[7:1]};
            else
                shf_po <= shf_po;
        end
    end

    assign po = shf_po;
endmodule
```

The description consists of two main parts:

1. 3 bit counter (mod 8)
2. 8 bit shift register

The `co` output of this circuit will be used as a controlling signal in controller, but the `po` output will be loaded to our 8 bit down counter in datapath (which we'll see in part c)

The symbol for our circuit in this part is like:



The used cells for this circuit:

```
Total logic elements    13 / 14,400 ( < 1 % )
Total registers         11
Total pins              15 / 81 ( 19 % )
Total virtual pins      0
Total memory bits       0 / 552,960 ( 0 % )
```

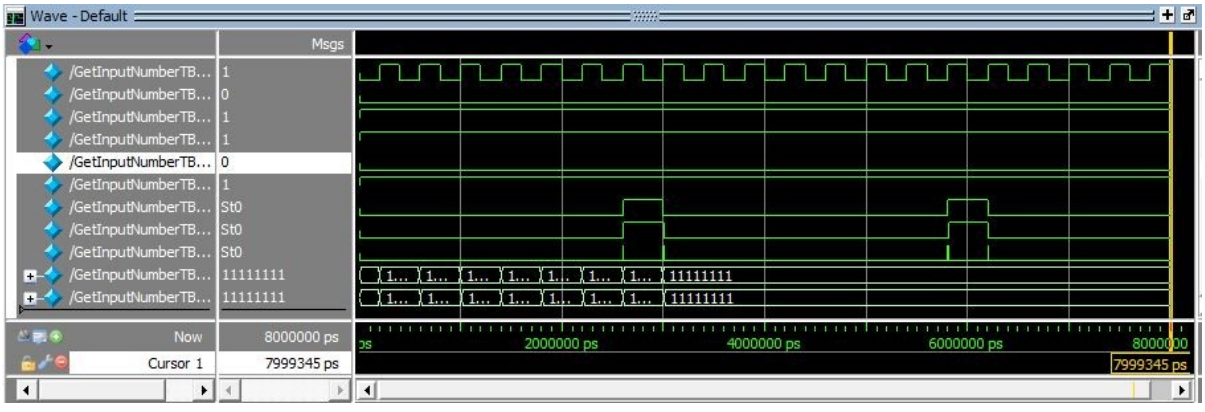
ii

In this part, we'll write a testbench for our circuit, here's the testbench:

```
`timescale 1ns/1ns
module GetInputNumberTB;
 reg clk = 0;
 reg iz_cnt = 0;
 reg shen = 1;
 reg cen = 1;
 reg rst = 0;
 reg serin = 1;
 wire COPre, COPost, diff;
 wire [7:0] POPre;
```

```
  wire [7:0] POPost;

  GetInputNumber UUT1(iz_cnt, cen, shen, clk, rst, serin, COPre, POPre);
  GetInputNumberPost UUT2(iz_cnt, cen, shen, clk, rst, serin, COPost, POPost);

  assign diff = COPre ^ COPost;

  initial begin
    repeat (40) #(200) clk = ~clk;
  end

  initial begin
    #(200 * 40)

    $stop;
  end
endmodule
```

And after simulation, this is the resulted waveform:



Once again as you can see, there is a glitch in `diff` signal which shows that our post-synthesise file has delay applied to it.

Part c

i

In this part, we'll need an 8 bit down counter register with parallel load. Here's the description for it:

```
`timescale 1ns/1ns
```

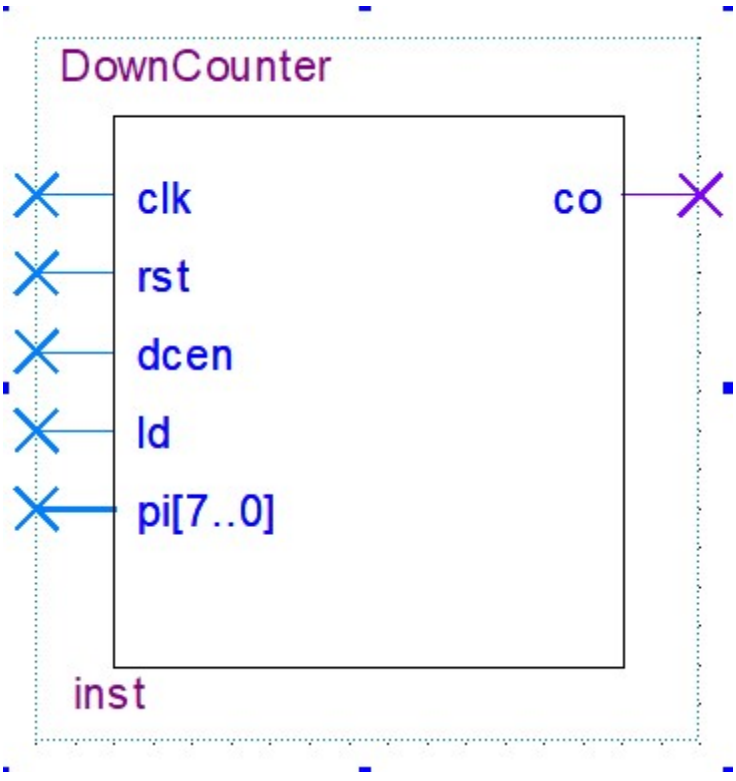
```
module DownCounter(input clk, rst, dcen, ld, input [7:0] pi, output co);
    reg [7:0] po = 8'b0;
    always @(posedge clk, posedge rst) begin
        if (rst)
            po <= 8'b0;
        else if (ld)
            po <= pi;
        else begin
            if (dcen)
                po <= po - 1;
            else
                po <= po;
        end
    end

    assign co = po == 1;
endmodule
```

Note: the `co` output is `1` when the data in registers is `00000001`, because of the timing of the clock, we need to set `co` like this so we wont waste any inputs in clocks.

The cells used by Quartus:

```
Total logic elements    12 / 14,400 ( < 1 % )
Total registers         8
Total pins              13 / 81 ( 16 % )
Total virtual pins      0
Total memory bits       0 / 552,960 ( 0 % )
```

The symbol for out circuit:



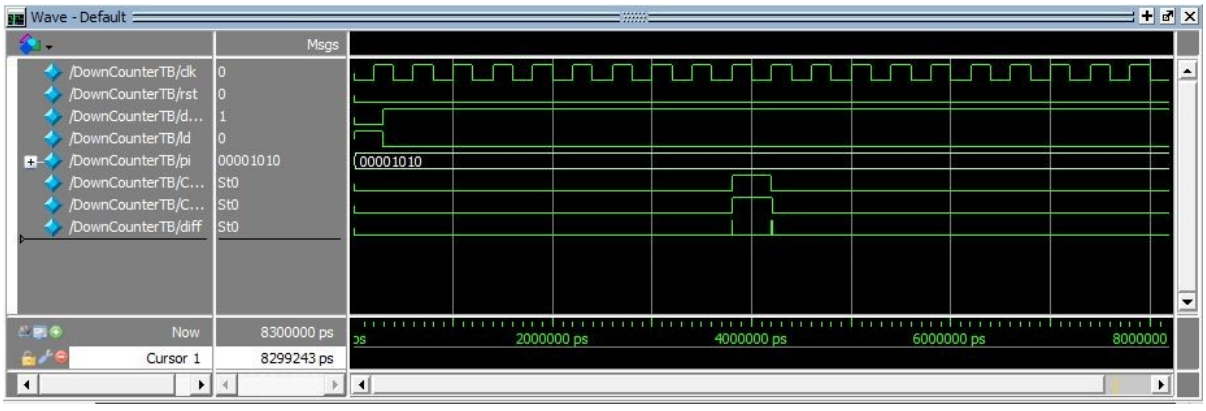
ii

The testbench for circuit of this part is like this. We'll set `pi` to `10` just to see if it works fine:

```
`timescale 1ns/1ns
module DownCounterTB;
  reg clk = 0;
  reg rst = 0;
  reg dcen = 0;
  reg ld = 1;
  reg [7:0] pi = 10;
  wire COPre, COPost, diff;

  DownCounter UUT1(clk, rst, dcen, ld, pi, COPre);
  DownCounterPost UUT2(clk, rst, dcen, ld, pi, COPost);

  assign diff = COPre ^ COPost;

  initial begin
    repeat (40) #(200) clk = ~clk;
  end

  initial begin
```

```
    #300 ld = 0;
    dcen = 1;
    #(200 * 40)

    $stop;
  end
endmodule
```

And the waveform is like this:



Notice the glitchs in the `diff` signal again. It's because of the delay that post-synthesised file has. Also, not that exactly after 9 clocks, the `co` output becomes `1`.

Part d

i

This part consists of two main parts:

1. Designing controller
2. Bussing and connecting all the pieces together

For out controller, we'll use a state machine with 3 states:

1. Sequence detection
2. Reading input number
3. Transmitting output

Here's the Verilog description of the controller:

```
`timescale 1ns/1ns
module Controller(input clk, rst, seqValid, co3, co8, output iz_cnt, cen3, she
    parameter [1:0] seqDetection = 2'b00, readingNumber = 2'b01, transmittingO
    reg [1:0] state = seqDetection, nextState = seqDetection;

    reg iz_cnt_reg = 0, cen3_reg = 0, shen_reg = 0, dcen_reg = 0, ld_reg = 0;

    assign iz_cnt = iz_cnt_reg;
    assign cen3 = cen3_reg;
    assign shen = shen_reg;
    assign dcen = dcen_reg;
    assign ld = ld_reg;

    always @(state, seqValid, co3, co8) begin
        iz_cnt_reg = 0;
        cen3_reg = 0;
        shen_reg = 0;
        dcen_reg = 0;
        ld_reg = 0;
        case (state)
            seqDetection: begin
                iz_cnt_reg = 1;
                if (seqValid) begin
                    shen_reg = 1;
                    nextState = readingNumber;
                end
            end

            readingNumber: begin
                cen3_reg = 1;
                shen_reg = 1;
                if (co3) begin
                    nextState = transmittingOutput;
                    ld_reg = 1;
                end
            end

            transmittingOutput: begin
                dcen_reg = 1;
                if (co8) begin
                    nextState = seqDetection;
                end
            end

            default: nextState = seqDetection;
```

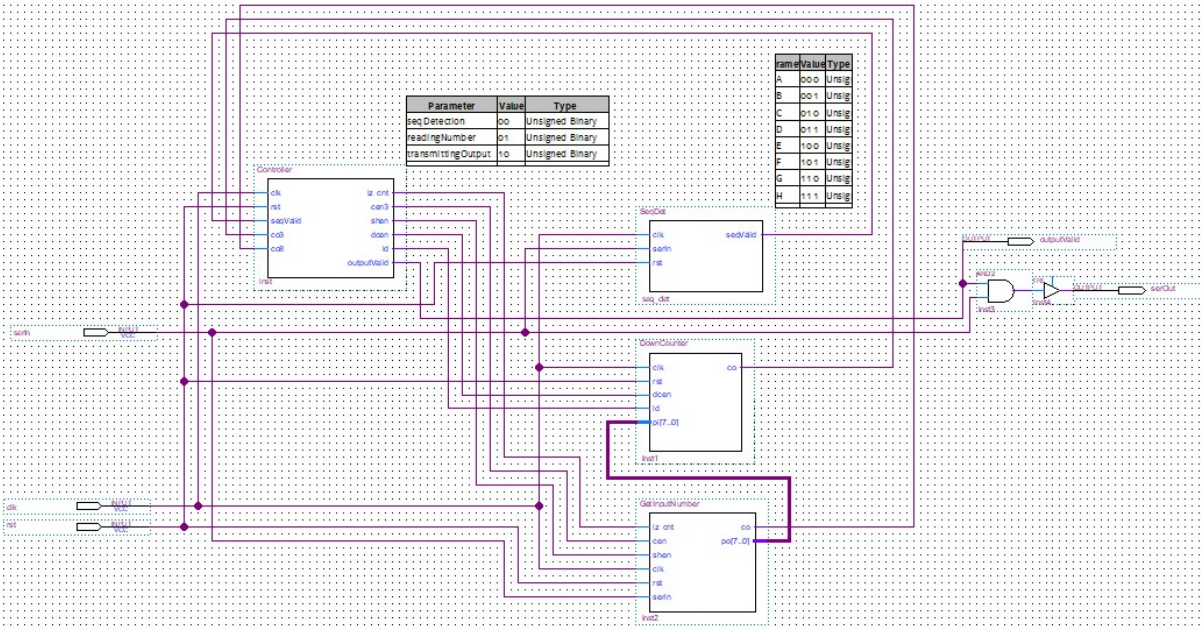
```
        endcase
    end

    assign outputValid = state == transmittingOutput;

    always @(posedge clk, posedge rst) begin
        if (rst)
            state <= seqDetection;
        else
            state <= nextState;
    end
endmodule
```

Note: because we don't want our outputs to be `reg`s, I defined some helper `reg`s and used `assign` statement to assign the output wires their `reg` signal values.

Here we have 2 options to connect our circuits with eachother. One way is to use a Verilog description:

```
`timescale 1ns/1ns
module MainCircuit(input serIn, clk, rst, output serOut, outputValid);
    wire seqValid, iz_cnt, cen3, shen, co3, dcen, ld, co8;
    wire [7:0] po;

    SeqDet seq_det(clk, serIn, rst, seqValid);
    GetInputNumber get_inp_numb(iz_cnt, cen3, shen, clk, rst, serIn, co3, po);
    DownCounter down_cnt(clk, rst, dcen, ld, po, co8);
    Controller cont(clk, rst, seqValid, co3, co8, iz_cnt, cen3, shen, dcen, ld

    assign serOut = outputValid ? serIn : 1'bz;
endmodule
```

And the other way is to use Quartus's graphical tool:



ii

Testbench:

```
`timescale 1ns/1ns
module MainCircuitTB;
  reg clk = 0;
  reg rst = 0;
  reg serIn = 0;
  wire serOutPre, serOutPost, outputValidPre, outputValidPost, diff;

  MainCircuit UUT1(serIn, clk, rst, serOutPre, outputValidPre);
  MainCircuitPost UUT2(serIn, clk, rst, serOutPost, outputValidPost);

  assign diff = serOutPre ^ serOutPost;

  initial begin
    repeat (1000) #(300) clk = ~clk;
  end

  initial begin
    serIn = 0;
    #(300 * 1.5) serIn = 1;
    #(10 * 300 + 300 * 0.5) serIn = 0;
    #(300 * 2);
    repeat (1000) #320 serIn = $random();
```

```
    $stop;
  end
endmodule
```

In both methods if we synthesise our circuit we'll get the following results: