



## San Francisco Bay University

### EE461 Verilog-HDL Homework #5

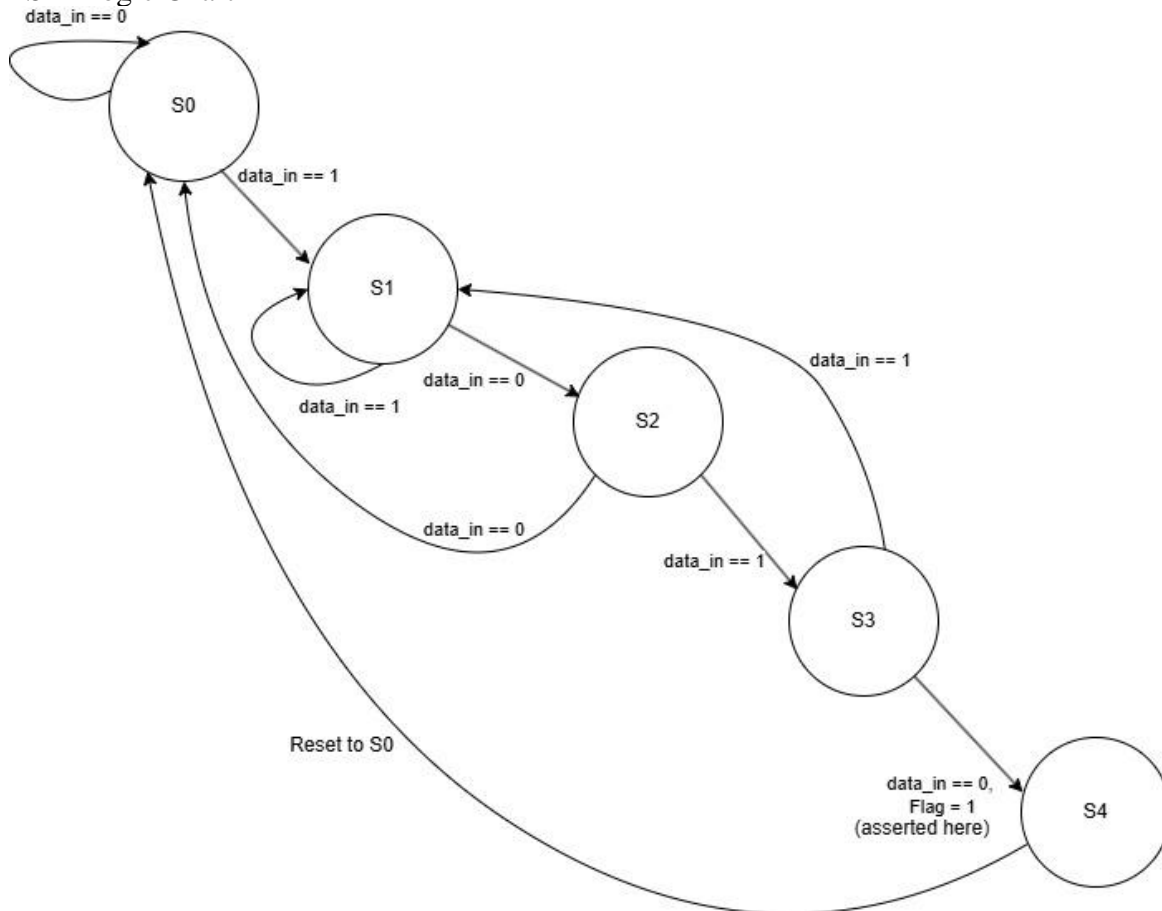
Due day: 11/27/2024

#### Instruction:

- Push answer sheets/source code to Github
- Please follow the code style rule like programs on handout.
- Overdue homework submission could not be accepted.
- Takes academic honesty and integrity seriously (Zero Tolerance of Cheating & Plagiarism)

1. Draw FSM logic chart to detect bit stream pattern “1010” and design FSM in Verilog to output flag signal with the testbench module.

#### FSM Logic Chart



## MODULE

```
module FSM_1010_detector(
    input clk,
    input rst,
    input data_in,
    output reg flag
);
    // State encoding
    parameter S0 = 3'b000,
               S1 = 3'b001,
               S2 = 3'b010,
               S3 = 3'b011,
               S4 = 3'b100;

    reg [2:0] current_state, next_state;

    // State transition logic
    always @(posedge clk or posedge rst) begin
        if (rst)
            current_state <= S0; // Reset to initial state
        else
            current_state <= next_state; // Move to the next state
    end

    // Next state logic
    always @(*) begin
        case (current_state)
            S0: next_state = (data_in) ? S1 : S0;
            S1: next_state = (data_in) ? S1 : S2;
            S2: next_state = (data_in) ? S3 : S0;
            S3: next_state = (data_in) ? S1 : S4;
            S4: next_state = (data_in) ? S1 : S0;
            default: next_state = S0;
        endcase
    end

    // Output logic: Flag is combinational when transitioning to S4
    always @(*) begin
        flag = (current_state == S3 && !data_in); // Detecting transition into S4
    end
endmodule
```

## TESTBENCH

```
module tb_FSM_1010_detector;
    reg clk, rst, data_in;
    wire flag;
```

```
// Instantiate the FSM
FSM_1010_detector uut (.clk(clk), .rst(rst), .data_in(data_in), .flag(flag));

// Clock generation
initial clk = 0;
always #5 clk = ~clk;

// Test sequence
initial begin
    rst = 1; data_in = 0;
    #10 rst = 0;

    // Input sequence: 10101011010
    #10 data_in = 1;
    #10 data_in = 0;
    #10 data_in = 1;
    #10 data_in = 0; // Detects "1010", flag should go high
    #10 data_in = 1;
    #10 data_in = 0;
    #10 data_in = 1;
    #10 data_in = 0;

    #50 $finish;
end

initial $monitor("Time: %0d, Input: %b, Flag: %b", $time, data_in, flag);
endmodule
```

## SCREENSHOT

```

1 module tb_FSM_1010_detector;
2   reg clk, rst, data_in;
3   wire flag;
4
5   // Instantiate the FSM
6   FSM_1010_detector uut (.clk(clk), .rst(rst),
7     .data_in(data_in), .flag(flag));
8
9   // Clock generation
10  initial clk = 0;
11  always #5 clk = ~clk;
12
13  // Test sequence
14  initial begin
15    rst = 1; data_in = 0;
16    #10 rst = 0;
17
18    // Input sequence: 10101011010
19    #10 data_in = 1;
20    #10 data_in = 0;
21    #10 data_in = 1;
22    #10 data_in = 0; // Detects "1010", flag should go
23    high
24    #10 data_in = 1;
25    #10 data_in = 0;
26    #10 data_in = 1;
27    #10 data_in = 0;
28    #50 $finish;
29  end
30
31  initial $monitor("Time: %0d, Input: %b, Flag: %b",
32    $time, data_in, flag);
33 endmodule
34
35
36
37
38
39
40
41

```

```

1 module FSM_1010_detector(
2   input clk,
3   input rst,
4   input data_in,
5   output reg flag
6 );
7 // State encoding
8 parameter S0 = 3'b000,
9           S1 = 3'b001,
10          S2 = 3'b010,
11          S3 = 3'b011,
12          S4 = 3'b100;
13
14 reg [2:0] current_state, next_state;
15
16 // State transition logic
17 always @(posedge clk or posedge rst) begin
18   if (rst)
19     current_state <= S0; // Reset to initial state
20   else
21     current_state <= next_state; // Move to the next state
22 end
23
24 // Next state logic
25 always @(*) begin
26   case (current_state)
27     S0: next_state = (data_in) ? S1 : S0;
28     S1: next_state = (data_in) ? S1 : S2;
29     S2: next_state = (data_in) ? S3 : S0;
30     S3: next_state = (data_in) ? S1 : S4;
31     S4: next_state = (data_in) ? S1 : S0;
32     default: next_state = S0;
33   endcase
34 end
35
36 // Output logic: Flag is combinational when transitioning to S4
37 always @(*) begin
38   flag = (current_state == S3 && !data_in); // Detecting transition into S4
39 end
40 endmodule
41

```

[2024-12-16 14:10:48 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out  
 Time: 0, Input: 0, Flag: 0  
 Time: 20, Input: 1, Flag: 0  
 Time: 30, Input: 0, Flag: 0  
 Time: 40, Input: 1, Flag: 0  
 Time: 50, Input: 0, Flag: 1  
 Time: 55, Input: 0, Flag: 0  
 Time: 60, Input: 1, Flag: 0  
 Time: 70, Input: 0, Flag: 0  
 Time: 80, Input: 1, Flag: 0  
 Time: 90, Input: 0, Flag: 1  
 Time: 95, Input: 0, Flag: 0  
 testbench.sv:27: \$finish called at 140 (1s)  
 Done

2. Design FSM in Verilog to detect bit streams which could be divided by 7 and output flag signal with the testbench module.

#### MODULE

```
module FSM_divisible_by_7 (  
    input clk,  
    input rst,  
    input data_in,  
    output reg flag  
);  
    reg [2:0] remainder;    // Current remainder  
    wire [2:0] next_remainder; // Next remainder (combinational logic)  
  
    // Combinational logic for next remainder  
    assign next_remainder = (remainder * 2 + data_in) % 7;  
  
    // Sequential logic for remainder and flag update  
    always @(posedge clk or posedge rst) begin  
        if (rst) begin  
            remainder <= 3'b000; // Reset remainder  
            flag <= 0;           // Reset flag  
        end else begin  
            remainder <= next_remainder; // Update remainder  
            flag <= (next_remainder == 3'b0); // Flag = 1 if next remainder is 0  
        end  
    end  
endmodule
```

#### TESTBENCH

```
module tb_FSM_divisible_by_7;  
    reg clk, rst, data_in;  
    wire flag;  
  
    // Instantiate the FSM  
    FSM_divisible_by_7 uut (  
        .clk(clk),  
        .rst(rst),  
        .data_in(data_in),  
        .flag(flag)  
    );  
  
    // Clock generation  
    initial clk = 0;  
    always #5 clk = ~clk; // 10ns clock period
```

```

// Test sequence
initial begin
    rst = 1; data_in = 0;
    #10 rst = 0;

    // Input sequence: bits representing numbers
    #10 data_in = 1; // Input 1
    #10 data_in = 0; // Input 10
    #10 data_in = 1; // Input 101
    #10 data_in = 0; // Input 1010
    #10 data_in = 1; // Input 10101

    #50 $finish;
end

initial $monitor("Time: %0d, Input: %b, Remainder: %d, Flag: %b", $time,
data_in, uut.remainder, flag);
endmodule

```

## SCREENSHOT

```

1 module tb_FSM_divisible_by_7;
2   reg clk, rst, data_in;
3   wire flag;
4
5   // Instantiate the FSM
6   FSM_divisible_by_7 uut (
7     .clk(clk),
8     .rst(rst),
9     .data_in(data_in),
10    .flag(flag)
11  );
12
13  // Clock generation
14  initial clk = 0;
15  always #5 clk = ~clk; // 10ns clock period
16
17  // Test sequence
18  initial begin
19    rst = 1; data_in = 0;
20    #10 rst = 0;
21
22    // Input sequence: bits representing numbers
23    #10 data_in = 1; // Input 1
24    #10 data_in = 0; // Input 10
25    #10 data_in = 1; // Input 101
26    #10 data_in = 0; // Input 1010
27    #10 data_in = 1; // Input 10101
28
29    #50 $finish;
30  end
31
32  initial $monitor("Time: %0d, Input: %b, Remainder: %d,
33  Flag: %b", $time, data_in, uut.remainder, flag);
34 endmodule

```

```

1 module FSM_divisible_by_7 (
2   input clk,
3   input rst,
4   input data_in,
5   output reg flag
6 );
7   reg [2:0] remainder; // Current remainder
8   wire [2:0] next_remainder; // Next remainder
9   (combinational logic)
10
11  // Combinational logic for next remainder
12  assign next_remainder = (remainder * 2 + data_in) % 7;
13
14  // Sequential logic for remainder and flag update
15  always @(posedge clk or posedge rst) begin
16    if (rst) begin
17      remainder <= 3'b000; // Reset remainder
18      flag <= 0; // Reset flag
19    end else begin
20      remainder <= next_remainder; // Update
21      flag <= (next_remainder == 3'b0); // Flag = 1
22    end
23  end
24 endmodule

```

```

[2024-12-16 14:46:04 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out
Time: 0, Input: 0, Remainder: 0, Flag: 0
Time: 15, Input: 0, Remainder: 0, Flag: 1
Time: 20, Input: 1, Remainder: 0, Flag: 1
Time: 25, Input: 1, Remainder: 1, Flag: 0
Time: 30, Input: 0, Remainder: 1, Flag: 0
Time: 35, Input: 0, Remainder: 2, Flag: 0
Time: 40, Input: 1, Remainder: 2, Flag: 0
Time: 45, Input: 1, Remainder: 5, Flag: 0
Time: 50, Input: 0, Remainder: 5, Flag: 0
Time: 55, Input: 0, Remainder: 3, Flag: 0
Time: 60, Input: 1, Remainder: 3, Flag: 0
Time: 65, Input: 1, Remainder: 0, Flag: 1
Time: 70, Input: 1, Remainder: 0, Flag: 0
Time: 75, Input: 1, Remainder: 1, Flag: 0
Time: 80, Input: 1, Remainder: 3, Flag: 0
Time: 85, Input: 1, Remainder: 0, Flag: 1
Time: 90, Input: 1, Remainder: 1, Flag: 0
Time: 95, Input: 1, Remainder: 1, Flag: 0
Time: 100, Input: 1, Remainder: 1, Flag: 0
Time: 105, Input: 1, Remainder: 1, Flag: 0
testbench.sv:29: $finish called at 110 (1s)
Done

```

3. Design Verilog module to find the duty cycle of PWM (Pulse Width Modulation) signal with a higher clock frequency compared to the PWM signal. After that, verify your design.

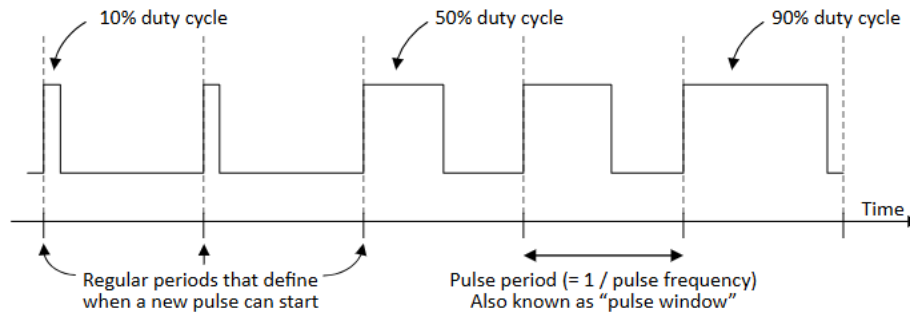


Figure 1. PWM signals and definitions

MODULE

```
module PWM_duty_cycle (
    input clk,          // High-frequency clock
    input rst,          // Reset signal
    input pwm_in,       // PWM signal input
    output reg [7:0] duty_cycle // Duty cycle percentage
);
```

```
    reg [15:0] high_count, total_count; // Counters for high and total time
    reg pwm_prev; // To detect edges
    reg update_flag; // Signal to update duty cycle
    reg first_period; // Skip first period calculation
```

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        high_count <= 0;
        total_count <= 0;
        duty_cycle <= 0;
        pwm_prev <= 0;
        update_flag <= 0;
        first_period <= 1; // Enable warm-up phase
    end else begin
        // Count total time and high time
        total_count <= total_count + 1;
        if (pwm_in) high_count <= high_count + 1;

        // Detect falling edge (end of PWM period)
        if (pwm_prev && ~pwm_in) begin
            if (!first_period && total_count > 0) begin
```

```
        duty_cycle <= (high_count * 100) / total_count; // Duty cycle calculation
    end
    first_period <= 0; // Disable warm-up after first period

    // Reset counters for next period
    high_count <= 0;
    total_count <= 0;
end

    pwm_prev <= pwm_in; // Update previous state
end
end
endmodule
```

#### TESTBENCH

```
module tb_PWM_duty_cycle;
    reg clk, rst, pwm_in;
    wire [7:0] duty_cycle;

    // Instantiate the module
    PWM_duty_cycle uut (
        .clk(clk),
        .rst(rst),
        .pwm_in(pwm_in),
        .duty_cycle(duty_cycle)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk; // 10ns clock period

    // Generate PWM signal
    initial begin
        rst = 1; pwm_in = 0; #20; // Reset
        rst = 0;

        // PWM signal with varying duty cycles
        repeat (5) begin
            pwm_in = 1; #50; // High for 50 time units (50% duty cycle)
            pwm_in = 0; #50; // Low for 50 time units
        end

        repeat (5) begin
            pwm_in = 1; #30; // High for 30 time units (30% duty cycle)
            pwm_in = 0; #70; // Low for 70 time units
        end
    end
end
```



```
repeat (5) begin
    pwm_in = 1; #10; // High for 10 time units (10% duty cycle)
    pwm_in = 0; #90; // Low for 90 time units
end

#1000 $finish;
end

// Monitor output
initial $monitor("Time: %0d, PWM In: %b, Duty Cycle: %0d%%", $time, pwm_in,
duty_cycle);
endmodule
```

## SCREENSHOT

The screenshot displays two panels from a Verilog simulation environment. The left panel, titled 'testbench.sv', contains the testbench code. It initializes a clock 'clk' to 0, sets a 10ns clock period, and generates a PWM signal 'pwm\_in' with varying duty cycles: 10% (lines 18-20), 50% (lines 23-25), and 30% (lines 28-30). The right panel, titled 'design.sv', shows the 'PWM\_duty\_cycle' module. It takes 'clk' and 'pwm\_in' as inputs and outputs a 7-bit 'duty\_cycle' percentage. The module uses counters 'high\_count' and 'total\_count' to calculate the duty cycle based on the input signal's high and low periods. The bottom panel shows the simulation log output, which tracks the 'Time', 'PWM In' state, and 'Duty Cycle' percentage over time. The output shows the duty cycle changing from 0% to 10%, then to 55%, then to 33%, and finally to 11% as the testbench signal changes. The simulation ends at 2520 time units with the message 'testbench.sv:38: \$finish called at 2520 (1s)'.

```
testbench.sv
14 initial clk = 0;
15 always #10 clk = ~clk; // 10ns clock period
16
17 // Generate PWM signal
18 initial begin
19     rst = 1; pwm_in = 0; #20; // Reset
20     rst = 0;
21
22     // PWM signal with varying duty cycles
23     repeat (5) begin
24         pwm_in = 1; #10; // High for 10 time units
25         pwm_in = 0; #90; // Low for 90 time units
26     end
27
28     repeat (5) begin
29         pwm_in = 1; #30; // High for 30 time units
30         pwm_in = 0; #70; // Low for 70 time units
31     end
32
33     repeat (5) begin
design.sv
1 module PWM_duty_cycle (
2     input clk, // High-frequency clock
3     input rst, // Reset signal
4     input pwm_in, // PWM signal input
5     output reg [7:0] duty_cycle // Duty cycle percentage
6 );
7
8     reg [15:0] high_count, total_count; // Counters for high and total time
9     reg pwm_prev; // To detect edges
10    reg update_flag; // Signal to update duty cycle
11    reg first_period; // Skip first period calculation
12
13    always @(posedge clk or posedge rst) begin
14        if (rst) begin
15            high_count <= 0;
16            total_count <= 0;
17            duty_cycle <= 0;
18            pwm_prev <= 0;
19            update_flag <= 0;
20            first_period <= 1; // Enable warm-up phase
21        end else begin
22            // Count total time and high time
[2024-12-16 15:04:29 UTC] iverilog -Wall -g2012 design.sv testbench.sv && unbuffer vvp a.out
Time: 0, PWM In: 0, Duty Cycle: 0%
Time: 20, PWM In: 1, Duty Cycle: 0%
Time: 70, PWM In: 0, Duty Cycle: 0%
Time: 120, PWM In: 1, Duty Cycle: 0%
Time: 170, PWM In: 0, Duty Cycle: 0%
Time: 175, PWM In: 0, Duty Cycle: 55%
Time: 220, PWM In: 1, Duty Cycle: 55%
Time: 270, PWM In: 0, Duty Cycle: 55%
Time: 320, PWM In: 1, Duty Cycle: 55%
Time: 370, PWM In: 0, Duty Cycle: 55%
Time: 420, PWM In: 1, Duty Cycle: 55%
Time: 470, PWM In: 0, Duty Cycle: 55%
Time: 520, PWM In: 1, Duty Cycle: 55%
Time: 550, PWM In: 0, Duty Cycle: 55%
Time: 555, PWM In: 0, Duty Cycle: 42%
Time: 620, PWM In: 1, Duty Cycle: 42%
Time: 650, PWM In: 0, Duty Cycle: 42%
Time: 655, PWM In: 0, Duty Cycle: 33%
Time: 720, PWM In: 1, Duty Cycle: 33%
Time: 750, PWM In: 0, Duty Cycle: 33%
Time: 820, PWM In: 1, Duty Cycle: 33%
Time: 850, PWM In: 0, Duty Cycle: 33%
Time: 920, PWM In: 1, Duty Cycle: 33%
Time: 950, PWM In: 0, Duty Cycle: 33%
Time: 1020, PWM In: 1, Duty Cycle: 33%
Time: 1030, PWM In: 0, Duty Cycle: 33%
Time: 1035, PWM In: 0, Duty Cycle: 14%
Time: 1120, PWM In: 1, Duty Cycle: 14%
Time: 1130, PWM In: 0, Duty Cycle: 14%
Time: 1135, PWM In: 0, Duty Cycle: 11%
Time: 1220, PWM In: 1, Duty Cycle: 11%
Time: 1230, PWM In: 0, Duty Cycle: 11%
Time: 1320, PWM In: 1, Duty Cycle: 11%
Time: 1330, PWM In: 0, Duty Cycle: 11%
Time: 1420, PWM In: 1, Duty Cycle: 11%
Time: 1430, PWM In: 0, Duty Cycle: 11%
testbench.sv:38: $finish called at 2520 (1s)
Done
```