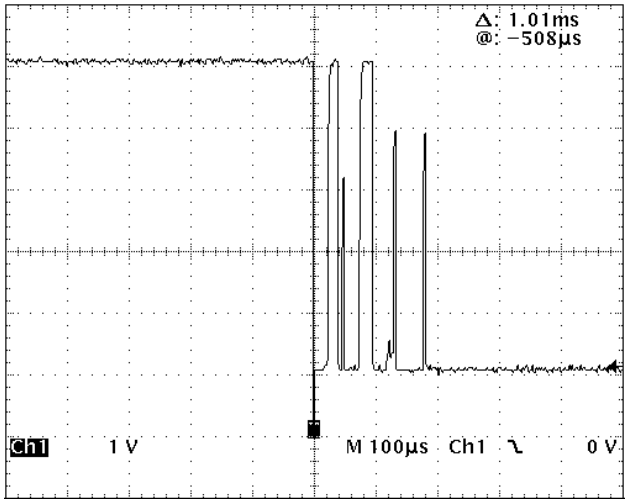
Finite-State Machine Button and Stopwatch

黃元豪
Yuan-Hao Huang

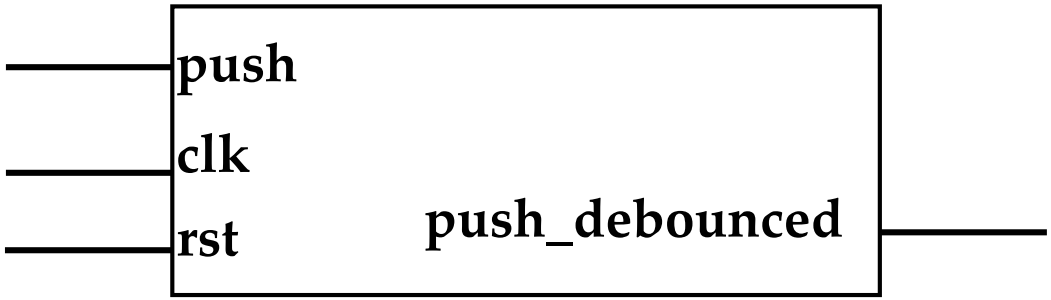
國立清華大學電機工程學系
Department of Electrical Engineering
National Tsing-Hua University

Switch Contact Bounce

- Pushbutton contains a metal spring
  - The switch contact will bounce several times before stabilization when buttons are pressed and released.
  - The random pulses causes instable results to the design.
  - The random pusles ranges in μs, but the FPGA is sensitive to pulses down to ns range.



Debounce Circuit



- N-bit shift register triggered with clock can be used as debounce circuit.
- When all 4 bits of the register are high (low), the output of the debounce circuit changes to high (low).

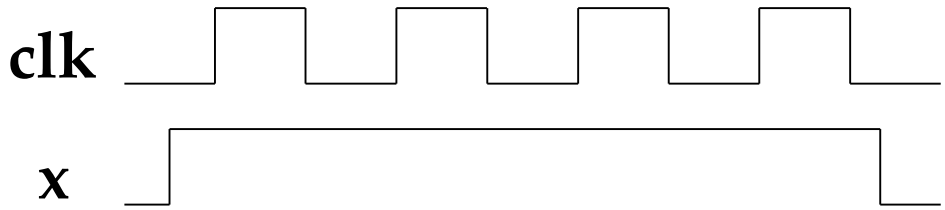
Debounce Design

```
module debounce (
 rst, clk, push, push_debounced );

input rst;
input clk;
input push;
output push_debounced;

// declare the outputs
reg push_debounced;

// declare the shifting registers
reg[3:0] push_window;
```

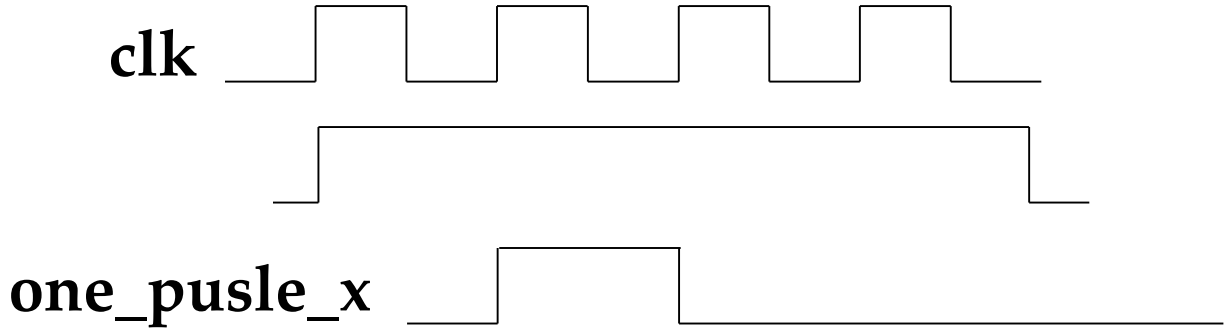
```
 always  @(posedge clk or posedge rst)
  begin
    if (rst) begin
      push_window <= 4'b0;
      push_debounced <= 1'b0;
    end else begin
      push_window<={push, push_window[3:1]};

     if (push_window[3:0] == 4'b1111) begin
        push_debounced <= 1'b1;
      end else begin
        push_debounced <= 1'b0;
      end
    end
  end
endmodule
```

One-Pulse Generation

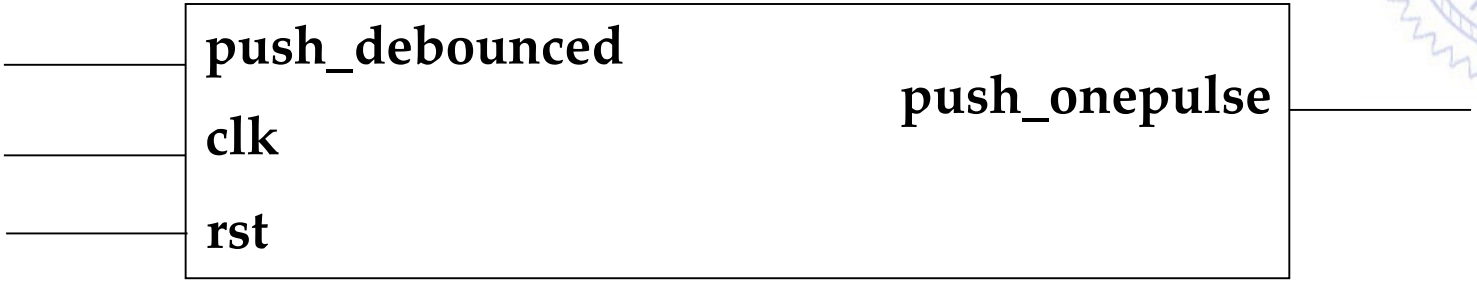
- Since the state of pressed is much longer than one clock period, the push-button usually recognizes one "pressed" event as successive cycles of "pressed".



- Once a pushbutton is pressed (long or short), the one-pulse circuit generates only a one-clock-period-long pulse.



One-Pulse Generation



- Inputs
  - clk, rst : system clock and reset.
  - push_debounced: debounced pushbutton signal
- Output
  - push_onepulse: one-clock-cylce impulse when pushbuttone is pressed.

One-Pulse Design

```
module onepulse (
rst, clk, push_debounced, push_onepulse
);
  input clk, rst;
  input push_debounced;
  output push_onepulse;

  // internal registers
  reg push_onepulse_next;
  reg push_debounced_delay;

  always @* begin
    push_onepulse_next  =  push_debounced
     & ~push_debounced_delay;
  end
```

```
  always @(posedge clk or posedge rst)
     begin
    if (rst) begin

      push_onepulse <= 1'b0;
      push_debounced_delay <= 1'b0;

    end else begin

 push_onepulse <= push_onepulse_next;
push_debounced_delay <=push_debounced;

    end
  end
endmodule
```

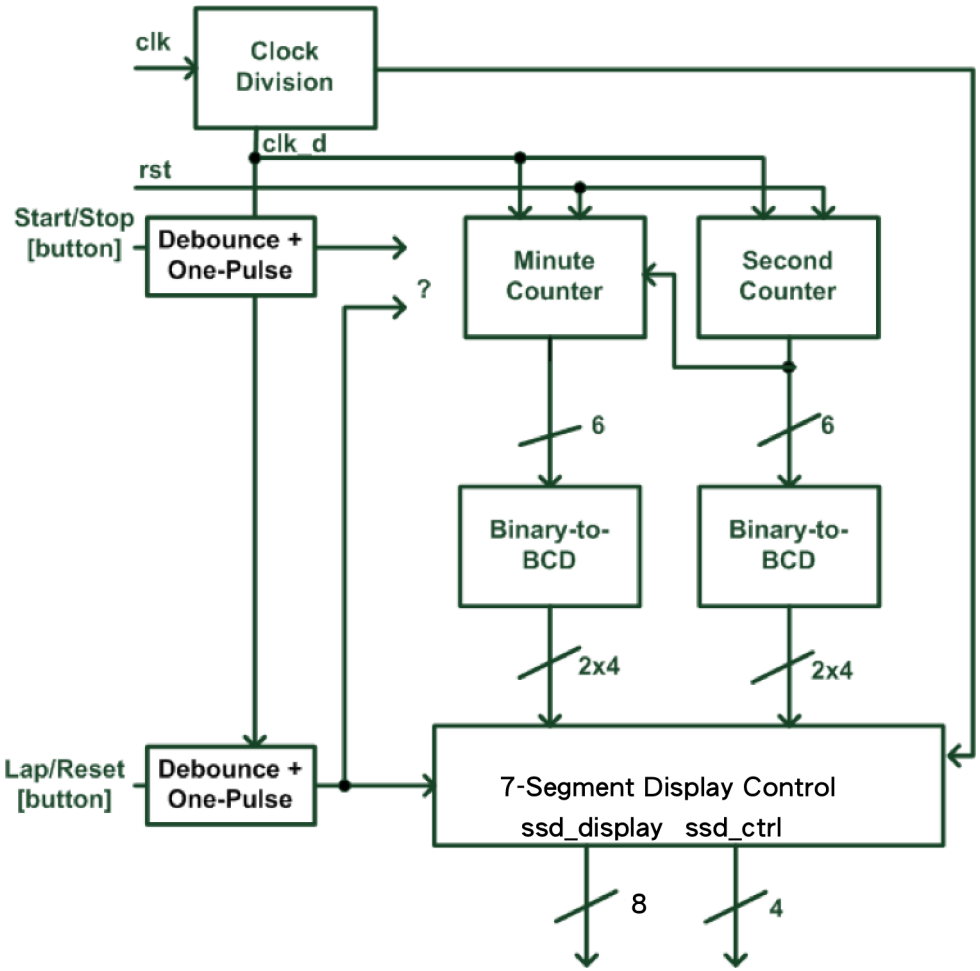
Timers and Stopwatches

Timer

- Timer is a counter with carry propagation.
  - Second : 0~59
  - Minute : 0~59
  - Hour : 0~23

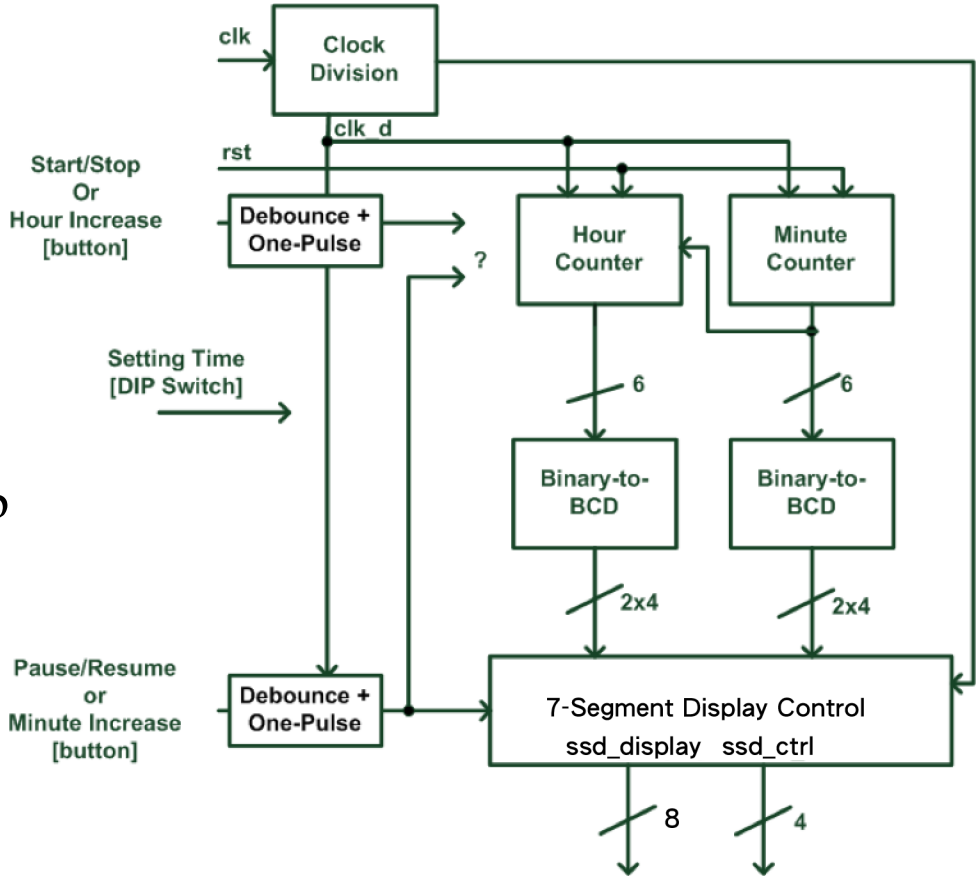
Stopwatch

- Stopwatch
  - Start/Stop function
  - When counting
    - Lap/Reset button freezes the 7-segment LED display
  - When no counting (i.e. stop state)
    - Lap/Reset button resets the counter to zero.

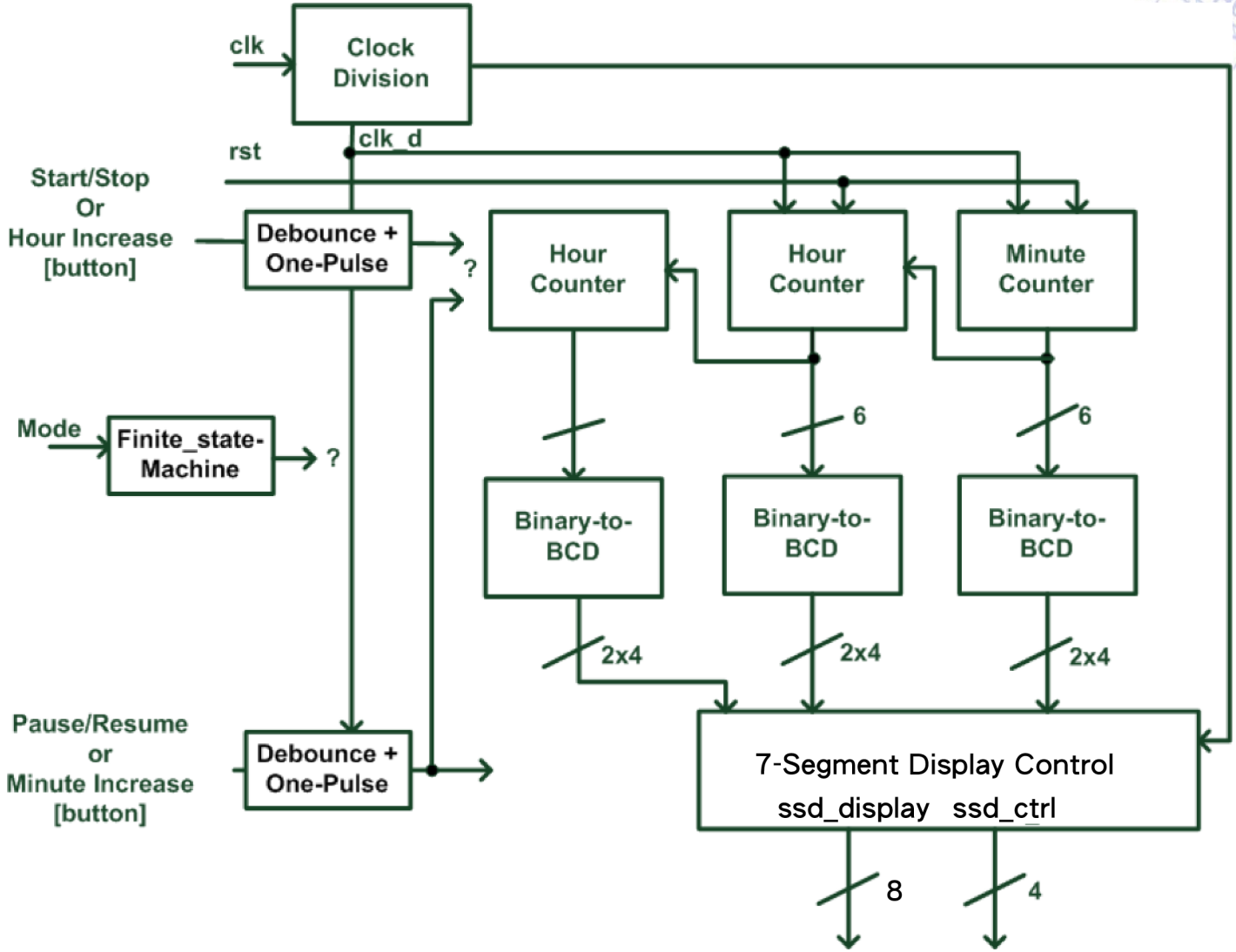


Timer

- Timer (Hour : Minute)
  - DIP switch is on
    - Press Button 1 to increase hour counter
    - Press Button 2 to increase minute counter
  - DIP switch is off
    - Press Button 1 for start/stop control
    - Press Button 2 for pause/resume control



Stopwatch and Timer

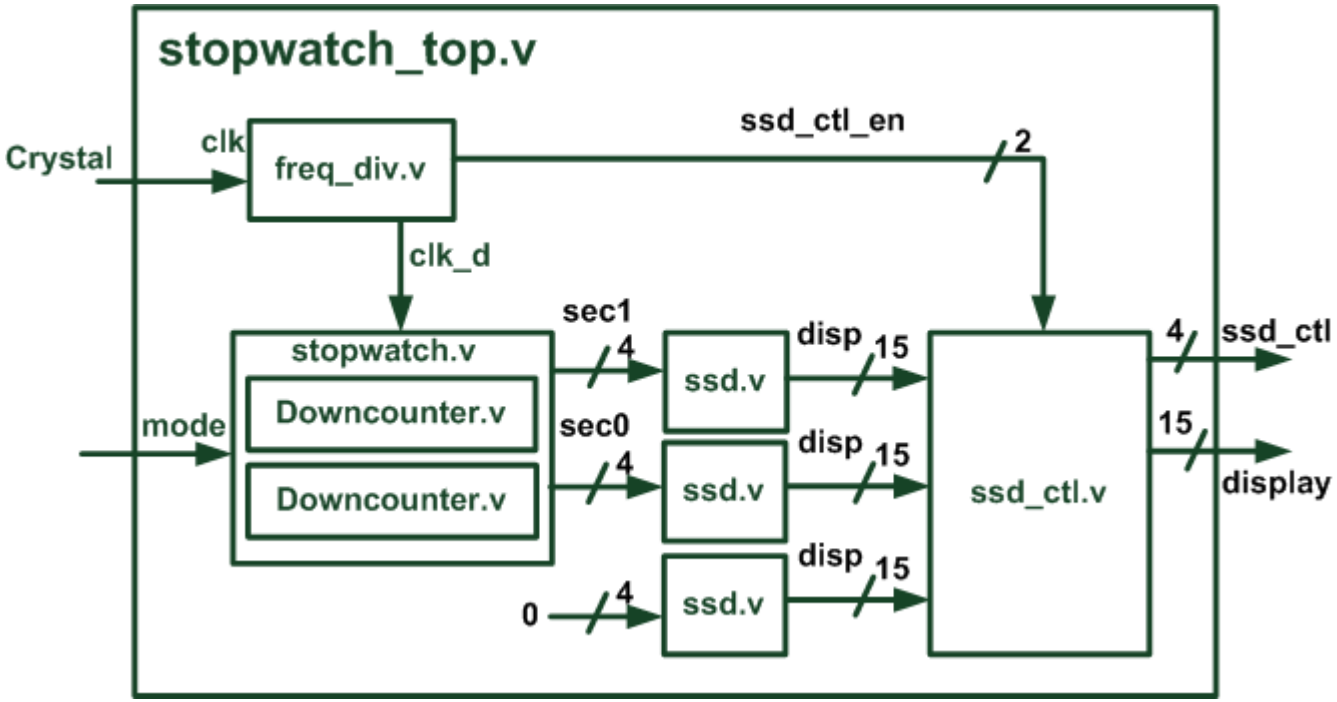


Stopwatch

without Finite State Machine

stopwatch

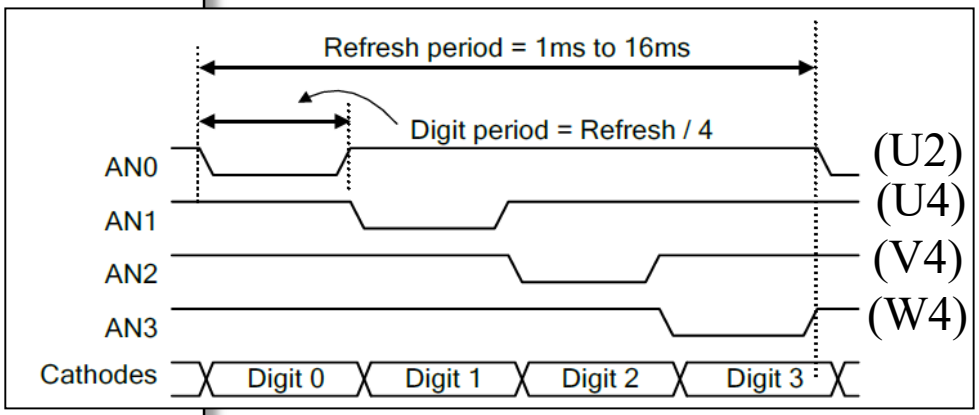
en=0	disable counting
en=0, mode=0	initial at 15
en=0, mode=1	initial at 30
en=1	enable down counting



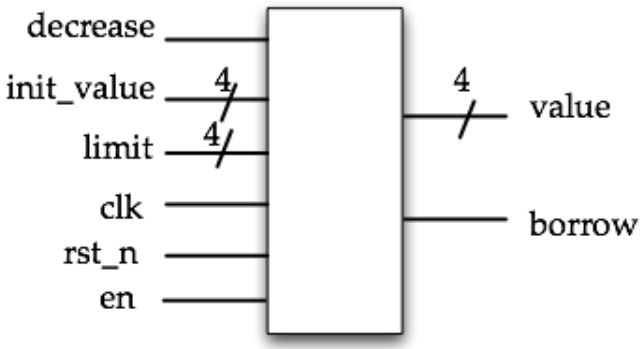
global.v

```
`define BCD_BIT_WIDTH 4 // bit width for BCD counter
`define ENABLED 1 // ENABLE indicator
`define DISABLED 0 // EIDABLE indicator
`define INCREMENT 1'b1 // increase by 1
`define BCD_ZERO 4'd0 // 0 for BCD
`define BCD_ONE 4'd1 // 1 for BCD
`define BCD_TWO 4'd2 // 2 for BCD
`define BCD_THREE 4'd3 // 3 for BCD
`define BCD_FOUR 4'd4 // 4 for BCD
`define BCD_FIVE 4'd5 // 5 for BCD
`define BCD_SIX 4'd6 // 6 for BCD
`define BCD_SEVEN 4'd7 // 7 for BCD
`define BCD_EIGHT 4'd8 // 8 for BCD
`define BCD_NINE 4'd9 // 9 for BCD
`define BCD_DEF 4'd15 // all 1

// For BCD to seven segment display decoder
`define SSD_BIT_WIDTH 8 // SSD display control bit width
`define SSD_ZERO  `SSD_BIT_WIDTH'b0000001_1 // 0
`define SSD_ONE   `SSD_BIT_WIDTH'b1001111_1 // 1
`define SSD_TWO   `SSD_BIT_WIDTH'b0010010_1 // 2
`define SSD_THREE `SSD_BIT_WIDTH'b000110_1 // 3
`define SSD_FOUR  `SSD_BIT_WIDTH'b1001100_1 // 4
`define SSD_FIVE  `SSD_BIT_WIDTH'b0100100_1 // 5
`define SSD_SIX   `SSD_BIT_WIDTH'b0100000_1 // 6
`define SSD_SEVEN `SSD_BIT_WIDTH'b0001111_1 // 7
`define SSD_EIGHT `SSD_BIT_WIDTH'b0000000_1 // 8
`define SSD_NINE  `SSD_BIT_WIDTH'b0001100_1  // 9
`define SSD_DEF   `SSD_BIT_WIDTH'b0000000_0 // default
```

```
// For adder
`define ADDER_IN_BIT_WIDTH 3 // For decoder
`define CTL_BIT_WIDTH 3
`define CTL_BIT_WIDTH_L 2
`define BCD_BIT_EXT 3 // BCD bit width

// SSD display scan control
`define SSD_DIGIT_NUM 4 // number of SSD digit
`define SSD_SCAN_CTL_BIT_NUM 2

// number of bits for ssd scan control
`define SSD_SCAN_CTL_DISP1 4'b1110 // set the value of SSD 0
`define SSD_SCAN_CTL_DISP2 4'b1101 // set the value of SSD 1
`define SSD_SCAN_CTL_DISP3 4’b1011 // set the value of SSD 2
`define SSD_SCAN_CTL_DISP4 4'b0111 // set the value of SSD 3
```



downcounter.v (1/2)

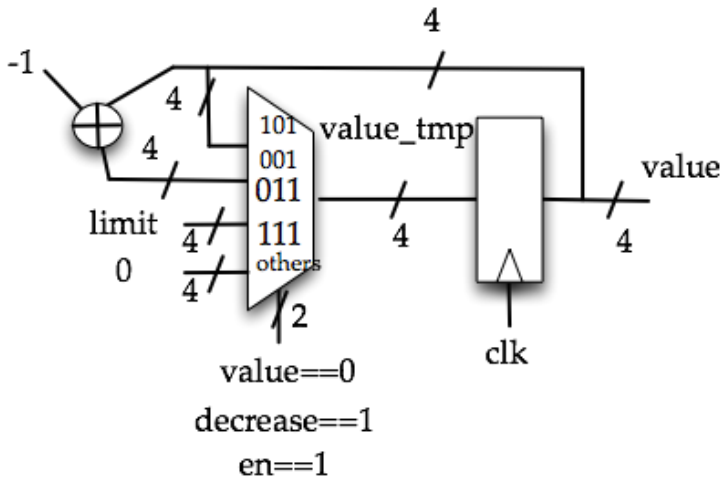


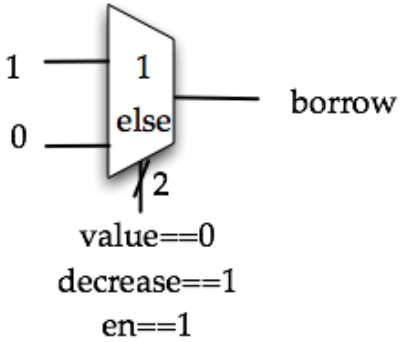

```
`include "global.v"
module downcounter(
  value,  // counter value
  borrow,  // borrow indicator for counter to next stage
  clk,  // global clock
  rst,  // active high reset
  decrease,  // decrease input from previous stage of counter
  init_value,  // initial value for the counter
  limit,  // limit for the counter
  en  // enable/disable of the counter
);

output [`BCD_BIT_WIDTH-1:0] value;  // counter value
output  borrow;  // borrow indicator for counter to next stage
input clk;  // global clock
input rst;  // active high reset
input decrease;  // decrease input from previous stage of counter
input [`BCD_BIT_WIDTH-1:0] init_value;  // initial value for the counter
input [`BCD_BIT_WIDTH-1:0] limit; // limit for the counter
input en;  // enable/disable of the counter

reg [`BCD_BIT_WIDTH-1:0] value; // output (in always block)
reg [`BCD_BIT_WIDTH-1:0] value_tmp; // input to dff (in always block)
reg borrow; //  borrow indicator for counter to next stage
```

downcounter.v (2/2)

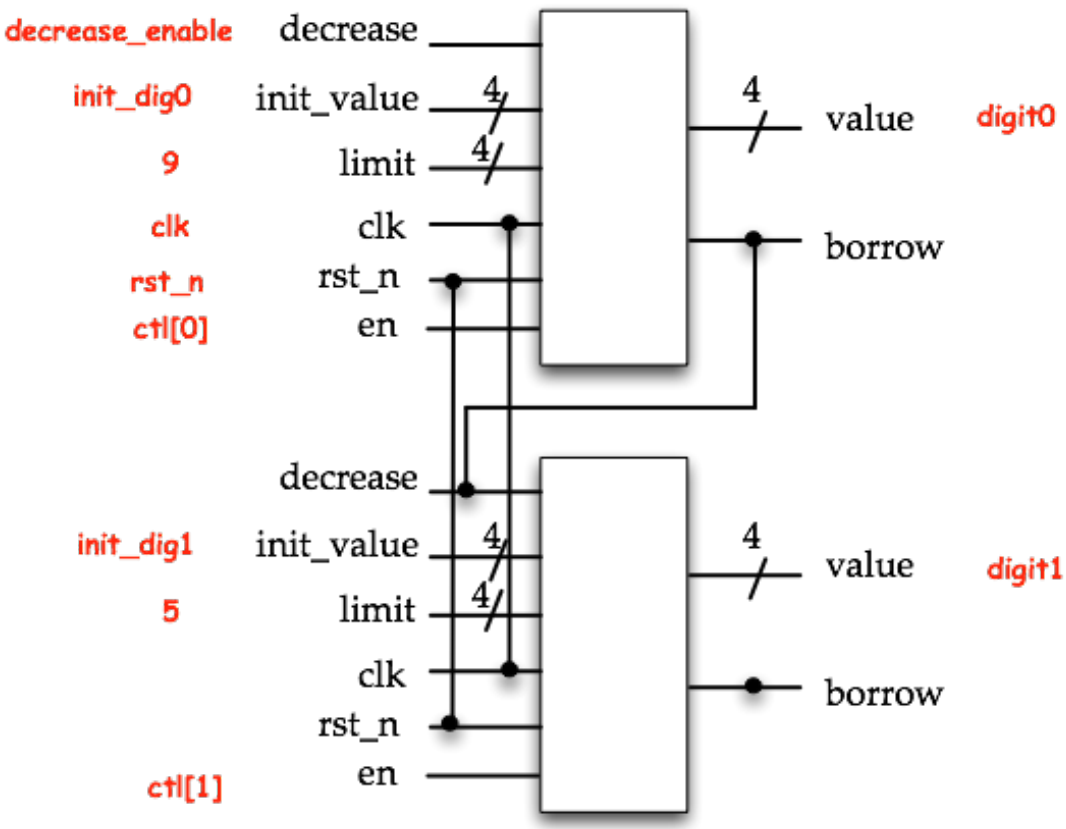
```
// Combinational logics
always @(value or decrease or en or limit)
 if (value==`BCD_ZERO && decrease && en)
   begin
     value_tmp = limit;
     borrow = `ENABLED;
   end
 else if (decrease && en)
   begin
     value_tmp = value - `INCREMENT;
     borrow = `DISABLED;
   end
 else if (en)
   begin
     value_tmp = value;
     borrow = `DISABLED;
   end
else
   begin
     value_tmp = `BCD_ZERO;
     borrow = `DISABLED;
   end
```

```
// register part for BCD counter
always @(posedge clk or posedge rst)
if (rst)
      value <= init_value;
else
      value <= value_tmp;
endmodule
```





stopwatch.v (1/2)




```
`include "global.v"
module stopwatch(
  digit1,  // 2nd digit of the down counter
  digit0,  // 1st digit of the down counter
  clk,  // global clock
  rst,  // active high reset
  mode,  // mode selection for 15 sec, 30-sec
  en  // enable/disable for stopwatch
);

output [`BCD_BIT_WIDTH-1:0] digit1;
output [`BCD_BIT_WIDTH-1:0] digit0;
input clk;  // global clock
input rst;  // active high reset
input mode;  // mode selection for 15 sec, 30-sec
input en;  // enable/disable for stopwatch

wire br0, br1;  // borrow indicator
wire decrease_enable;

reg [1:0] ctl;
reg [`BCD_BIT_WIDTH-1:0] init_dig0;
reg [`BCD_BIT_WIDTH-1:0] init_dig1;
```

stopwatch.v (2/2)

```
// mode selection: 15-sec or 30-sec
always @(mode)
  case (mode)
    1'b0:
    begin
      ctl = 2'b11;
      init_dig0 = `BCD_FIVE;
      init_dig1 = `BCD_ONE;
    end
    1'b1:
    begin
      ctl = 2'b11;
      init_dig0 = `BCD_ZERO;
      init_dig1 = `BCD_THREE;
    end
    default:
    begin
      ctl = 2'b00;
      init_dig0 = `BCD_ZERO;
      init_dig1 = `BCD_ZERO;
   end
  endcase

assign decrease_enable = en &&
    (~((digit0==`BCD_ZERO) &&
    (digit1==`BCD_ZERO)));
```

```
// 30 sec down counter
downcounter Udc0(
.value(digit0),  // counter value
.borrow(br0),  // borrow indicator for counter to next stage
.clk(clk), // global clock signal
.rst(rst),  // high active reset
.decrease(decrease_enable),  // decrease input from
previous stage of counter
.init_value(init_dig0),  // initial value for the counter
.limit(`BCD_NINE),  // limit for the counter
.en(ctl[0])  // enable/disable of the counter
);

downcounter Udc1(  .value(digit1),  // counter value
.borrow(br1),  // borrow indicator for counter to next stage
.clk(clk), // global clock signal
.rst(rst),  // high active reset
.decrease(br0),  // decrease input from previous stage of
counter
.init_value(init_dig1),  // initial value for the counter
.limit(`BCD_FIVE),  // limit for the counter
.en(ctl[1])  // enable/disable of the counter
);
endmodule
```

Stopwatch with Finite State Machine

Stopwatch with Finite State Machine

1 • stopwatch function (30, 15) with 1-bit control for stop(S0), start(S1), and pause display(S2)

inputs
1st : pressed
2nd : press_latch==111
3rd : press_latch==001

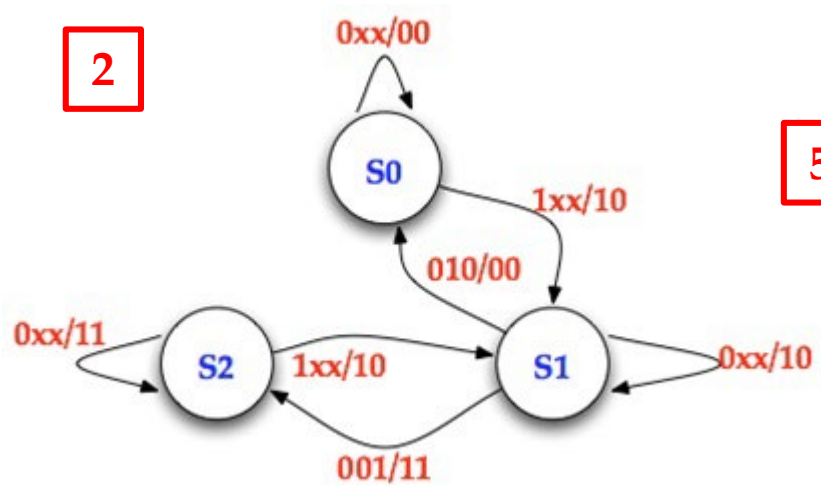
outputs
1st : count_enable
2nd : freeze_enable

3

Present State	Input			Next State	Output	
	in	case1	case2		count_enable	freeze_enable
S0	0	x	x	S0	0	0
S0	1	x	x	S1	1	0
S1	0	x	x	S1	1	0
S1	0	1	0	S0	0	0
S1	0	0	1	S2	1	1
S2	0	x	x	S2	1	1
S2	1	x	x	S1	1	0

4

2



5

Present State (state)	Input			Next State (state)	Output	
	in	case1	case2		count_enable	freeze_enable
00	0	x	x	00	0	0
00	1	x	x	01	1	0
01	0	x	x	01	1	0
01	0	1	0	00	0	0
01	0	0	1	10	1	1
10	0	x	x	10	1	1
10	1	x	x	01	1	0

Stopwatch with FSM

5

$(state_1(t), state(t)_0)$

Present State (state)	Input			Next State (state)	Output	
	in	case1	case2		count_enable	freeze_enable
00	0	x	x	00	0	0
00	1	x	x	01	1	0
01	0	x	x	01	1	0
01	0	1	0	00	0	0
01	0	0	1	10	1	1
10	0	x	x	10	1	1
10	1	x	x	01	1	0

$(state_1(t+1), state(t+1)_0)$

* unused state 11

6 **Next state equations**

$$state_0(t+1) = D_0(state_1(t), state_0(t), in, case_1, case_2) = \sum(13,16,17,18,19)$$

$$state_1(t+1) = D_1(state_1(t), state_0(t), in, case_1, case_2) = \sum(4,5,6,7,8,9,10,11,20,21,22,23)$$

10

11

Output equations

$$count_enable = \sum(4,5,6,7,8,9,10,11,13,16,17,18,19,20,21,22,23)$$

$$freeze_enable = \sum(4,5,6,7,8,9,10,11,20,21,22,23)$$

7 Choose D FFs (default)

8 **Excitation equations**

$$D_1 = state_1' \cdot state_0 \cdot in \cdot case_1' \cdot case_2' + state_1 \cdot state_0' \cdot in'$$

$$D_0 = state_0' \cdot in + state_1' \cdot state_0 \cdot in'$$

9 Your Logic Diagram

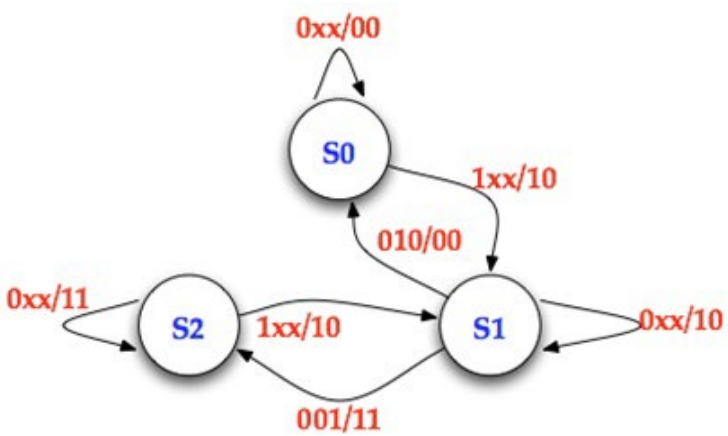
Output equations

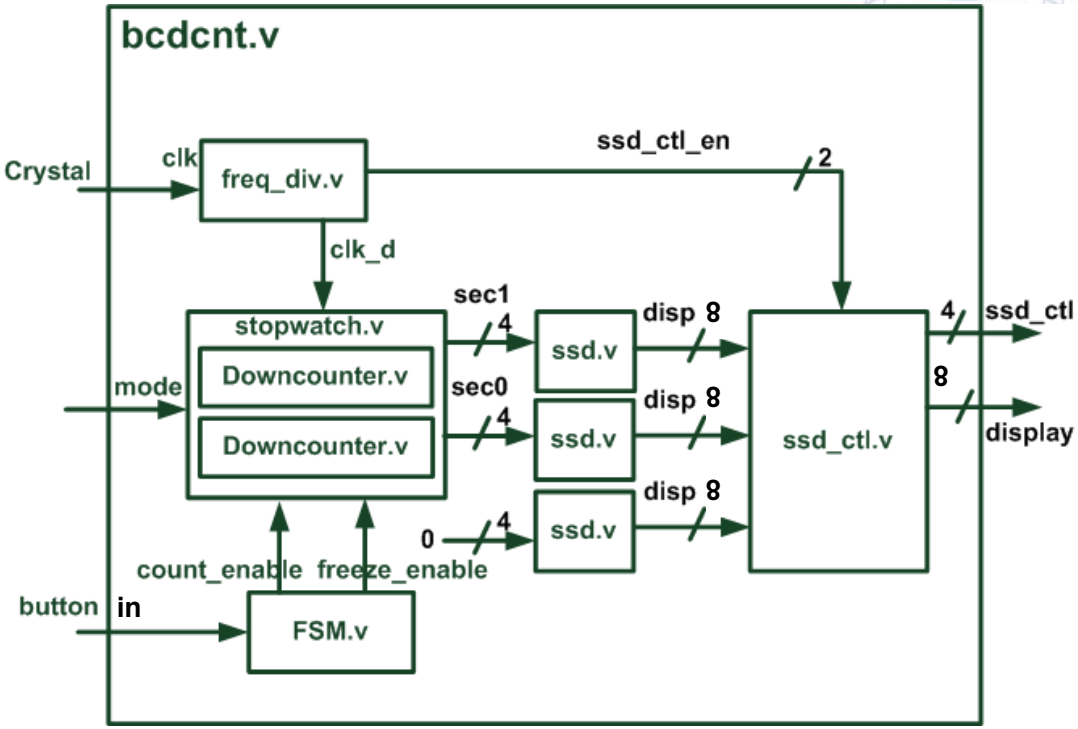
$$count_enable = state_1 \cdot state_0' + state_1' \cdot (state_0 \otimes in) + state_1' \cdot state_0 \cdot in \cdot case_1' \cdot case_2$$

$$freeze_en = state_1' \cdot state_0 \cdot in \cdot case_1' \cdot case_2' + state_1 \cdot state_0' \cdot in'$$

Stopwatch with FSM

en=0	disable counting
en=0, mode=0	initial at 15
en=0, mode=1	initial at 30
en=1	enable down counting



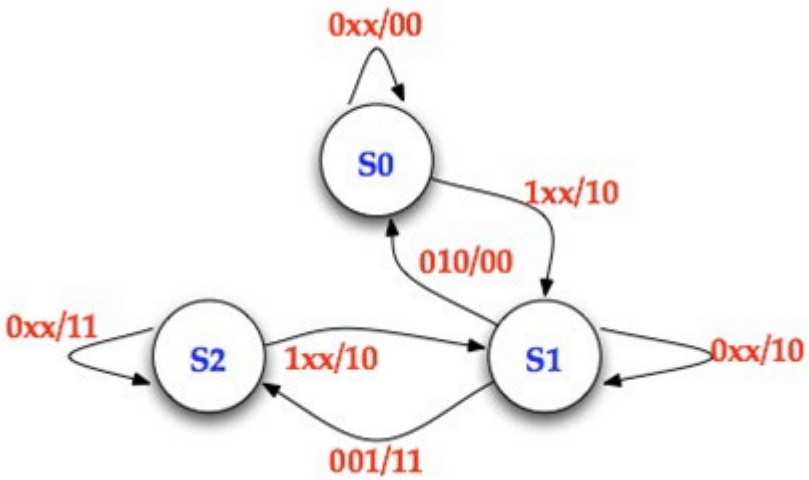


global.v

```
`define BCD_BIT_WIDTH 4 // bit width for BCD counter
`define ENABLED 1 // ENABLE indicator
`define DISABLED 0 // EIDABLE indicator
`define INCREMENT 1'b1 // increase by 1
`define BCD_ZERO 4'd0 // 0 for BCD
`define BCD_ONE 4'd1 // 1 for BCD
`define BCD_TWO 4'd2 // 2 for BCD
`define BCD_THREE 4'd3 // 3 for BCD
`define BCD_FOUR 4'd4 // 4 for BCD
`define BCD_FIVE 4'd5 // 5 for BCD
`define BCD_SIX 4'd6 // 6 for BCD
`define BCD_SEVEN 4'd7 // 7 for BCD
`define BCD_EIGHT 4'd8 // 8 for BCD
`define BCD_NINE 4'd9 // 9 for BCD
`define BCD_DEF 4'd15 // all 1

// For BCD to seven segment display decoder
`define SSD_BIT_WIDTH 8 // SSD display control bit width
`define SSD_ZERO  `SSD_BIT_WIDTH'b0000001_1 // 0
`define SSD_ONE   `SSD_BIT_WIDTH'b1001111_1 // 1
`define SSD_TWO   `SSD_BIT_WIDTH'b0010010_1 // 2
`define SSD_THREE `SSD_BIT_WIDTH'b000110_1 // 3
`define SSD_FOUR  `SSD_BIT_WIDTH'b1001100_1 // 4
`define SSD_FIVE  `SSD_BIT_WIDTH'b0100100_1 // 5
`define SSD_SIX   `SSD_BIT_WIDTH'b0100000_1 // 6
`define SSD_SEVEN `SSD_BIT_WIDTH'b0001111_1 // 7
`define SSD_EIGHT `SSD_BIT_WIDTH'b0000000_1 // 8
`define SSD_NINE  `SSD_BIT_WIDTH'b0001100_1  // 9
`define SSD_DEF   `SSD_BIT_WIDTH'b0000000_0 // default
```

```
// For adder
`define ADDER_IN_BIT_WIDTH 3// For decoder
`define CTL_BIT_WIDTH 3
`define CTL_BIT_WIDTH_L 2
`define BCD_BIT_EXT 3 // BCD bit width

// SSD display scan control
`define SSD_DIGIT_NUM 4 // number of SSD digit
`define SSD_SCAN_CTL_BIT_NUM 2

// number of bits for ssd scan control
`define SSD_SCAN_CTL_DISP1 4'b1110 // set the value of SSD 1
`define SSD_SCAN_CTL_DISP2 4’b1101 // set the value of SSD 2
`define SSD_SCAN_CTL_DISP3 4'b1011 // set the value of SSD 3
`define SSD_SCAN_CTL_DISP4 4'b0111 // set the value of SSD 4
```

FSM.v

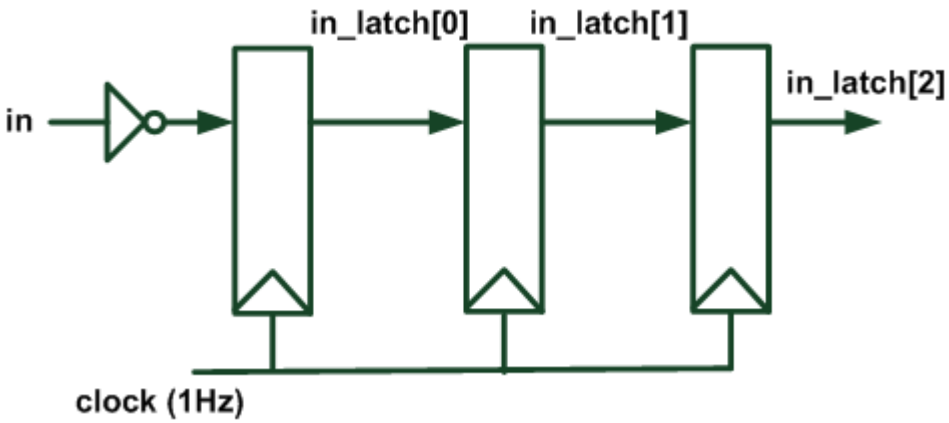



```
`include "global.v"

module FSM(  count_enable,  // if counter is enabled
freeze_enable,  // if ssd is freezed
in, //input control
clk, // global clock signal
rst  // high active reset
);

// outputs
output count_enable;  // if counter is enabled
output freeze_enable;  // if ssd is freezed
reg freeze_enable;

// inputs
input clk; // global clock signal
input rst; // high active reset
input in; //input control

reg count_enable;  // if counter is enabled
reg freeze_enable;  // if ssd is freezed
reg [1:0] state; // state of FSM
reg [1:0] next_state; // next state of FSM
reg [2:0] in_latch; // input latch
reg [2:0] in_latch_temp; // input latch
```

FSM.v

- Press duration control

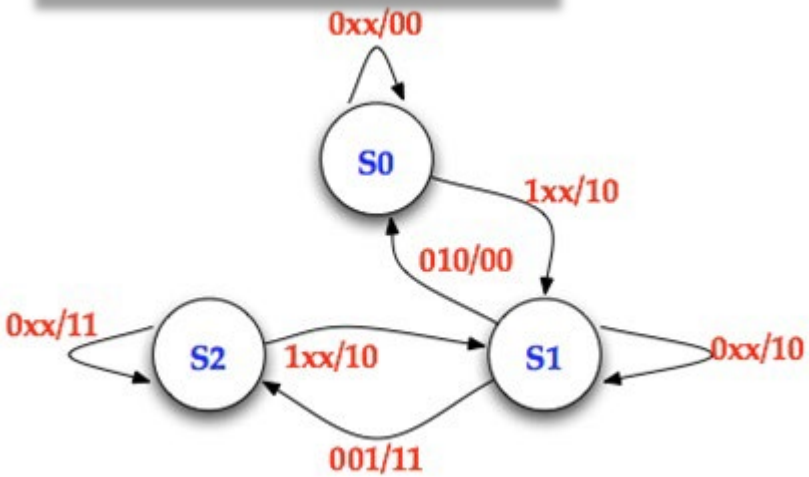
```
// State Definition
`define STAT_DEF        2'b00
`define STAT_COUNT      2'b01
`define STAT_FREEZE     2'b10


// FSM state transition

always @(posedge clk or posedge rst)
if (rst)    state <= `STAT_DEF;
else    state <= next_state;

// ************************
// Press counting
// ************************
always @(*)
if((in) && (state != `STAT_COUNT))
 in_latch_temp = 3'b000;
else
 in_latch_temp = {in_latch[1:0],~in};

always @(posedge clk or posedge rst)
if (rst)    in_latch <= 3'b000;
else    in_latch <= in_latch_temp;
```



FSM.v

```
// FSM state decision
always @(state or in or in_latch)
case (state)
`STAT_DEF:
if (in)     begin
next_state = `STAT_DEF;
count_enable = `DISABLED;
freeze_enable = `DISABLED;
end
else     begin
next_state = `STAT_COUNT;
count_enable = `ENABLED;
freeze_enable = `DISABLED;
end
`STAT_COUNT:
if ((in) && (in_latch==3'b111))     begin
next_state = `STAT_DEF;
count_enable = `DISABLED;
freeze_enable = `DISABLED;
end
else if ((in) && (in_latch==3'b001))   begin
next_state = `STAT_FREEZE;
count_enable = `ENABLED;
freeze_enable = `ENABLED;
end
else     begin
next_state = `STAT_COUNT;
count_enable = `ENABLED;
freeze_enable = `DISABLED;
end
```

```
`STAT_FREEZE:
if (~in)     begin
next_state = `STAT_COUNT;
count_enable = `ENABLED;
freeze_enable = `DISABLED;
end
else     begin
next_state = `STAT_FREEZE;
count_enable = `ENABLED;
freeze_enable = `ENABLED;
end
default:   begin
next_state = `STAT_DEF;
count_enable = `DISABLED;
freeze_enable = `DISABLED;
end  endcase

endmodule
```



stopwatch_disp.v

```
`include "global.v“

module stopwatch_disp(  display, // seven segment display control
ssd_ctl, // scan control for seven-segment display
clk, // clock
rst, // high active reset
mode, // mode selection for the stopwatch
in // input control for FSM
);

output [0:`SSD_BIT_WIDTH-1] display; // seven segment display control
output [`SSD_DIGIT_NUM-1:0] ssd_ctl; // scan control for ssd

input clk; // clock
input rst; // high active reset
input mode; // mode selection for the stopwatch
input in; // input control for FSM

wire [`SSD_SCAN_CTL_BIT_NUM-1:0] ssd_ctl_en; // divided output for ssd ctl
wire [0:`SSD_BIT_WIDTH-1] disp0,disp1,disp2,disp3; // ssd control
wire clk_d; // divided clock
wire count_enable; // if count is enabled
wire freeze_enable; // if ssd is enabled
wire [`BCD_BIT_WIDTH-1:0] dig0,dig1; // two counter outputs

reg [`BCD_BIT_WIDTH-1:0] dig0_latch,dig1_latch;  // two counter latches
reg [`BCD_BIT_WIDTH-1:0] dig0_out,dig1_out;  // output to ssd
```

stopwatch_disp.v

```
//*************************************************************
// Functional block
//*************************************************************

//frequency divider 1/(2^25)

freq_div U_FD(  .clk_out(clk_d), // divided clock
.ssd_ctl_en(ssd_ctl_en), // divided clock for scan control
.clk(clk), // clock from the crystal
.rst(rst) // high active reset
);

FSM U_fsm(  .count_enable(count_enable),  // if counter is enabled
.freeze_enable(freeze_enable),  // if ssd is freezed
.in(in), //input control
.clk(clk_d), // global clock signal
.rst(rst)  // high active reset
);

// stopwatch module

stopwatch U_sw(  .digit1(dig1),  // 2nd digit of the down counter
.digit0(dig0),  // 1st digit of the down counter
.clk(clk_d),  // global clock
.rst(rst),  // high active reset
.mode(mode),  // mode selection of 15-sec, 30-sec
.en(count_enable) // enable/disable for the stopwatch
);
```

```
// Display latch
always @(posedge clk_d or posedge rst)
if (rst)  begin
dig0_out <= 4'd0;
dig1_out <= 4'd0;
end
else  begin
dig0_out <= dig0_latch;
dig1_out <= dig1_latch;
end

// Whether the display is freezed
always @(freeze_enable or dig0 or dig1 or dig0_latch
or dig1_latch or dig0_out or dig1_out)
if (freeze_enable)  begin
dig0_latch = dig0_out;
dig1_latch = dig1_out;
end  else
begin
dig0_latch = dig0;
dig1_latch = dig1;
end
```

stopwatch_disp.v

```
//*************************************************************
// Display block
//*************************************************************
// BCD to seven segment display decoding

// seven-segment display decoder for DISP3
ssd U_SSD3(
.display(disp3), // SSD display output
.bcd(`BCD_DEF) // BCD input
);

// seven-segment display decoder for DISP2
ssd U_SSD2(
.display(disp2), // SSD display output
.bcd(`BCD_DEF) // BCD input
);

// seven-segment display decoder for DISP1
ssd U_SSD1(
.display(disp1), // SSD display output
.bcd(dig1_out) // BCD input
);

// seven-segment display decoder for DISP0
ssd U_SSD0(
.display(disp0), // SSD display output
.bcd(dig0_out) // BCD input
);
```

```
// SSD display scan control
ssd_ctl U_CTL(
.display_c(ssd_ctl), // ssd display control signal
.display(display), // output to ssd display
.ssd_ctl_en(ssd_ctl_en), // divided clock for scan control
.display0(disp0), // 1st input
.display1(disp1), // 2st input
.display2(disp2), // 3st input
.display3(disp3) // 4st input
);

endmodule
```