

Using Push Buttons

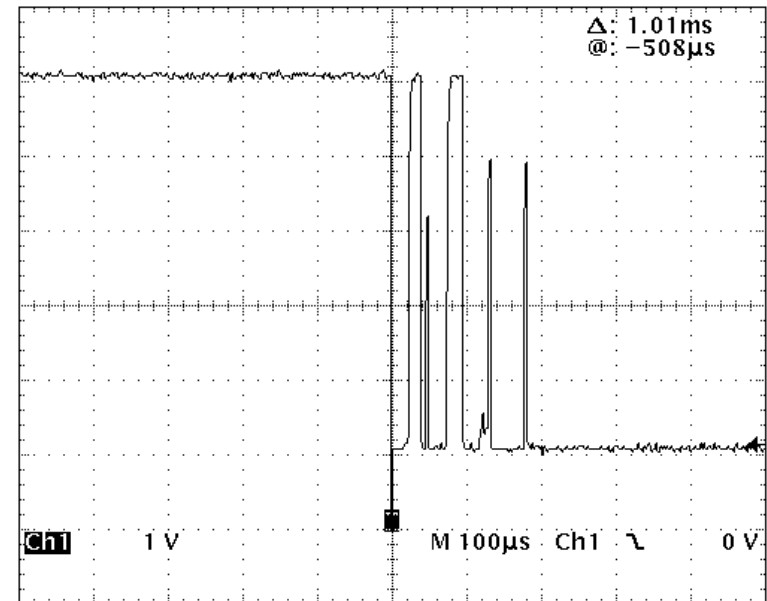
Hsi-Pin Ma

<http://lms.nthu.edu.tw/course/43639>

Department of Electrical Engineering
National Tsing Hua University

Switch Contact Bounce

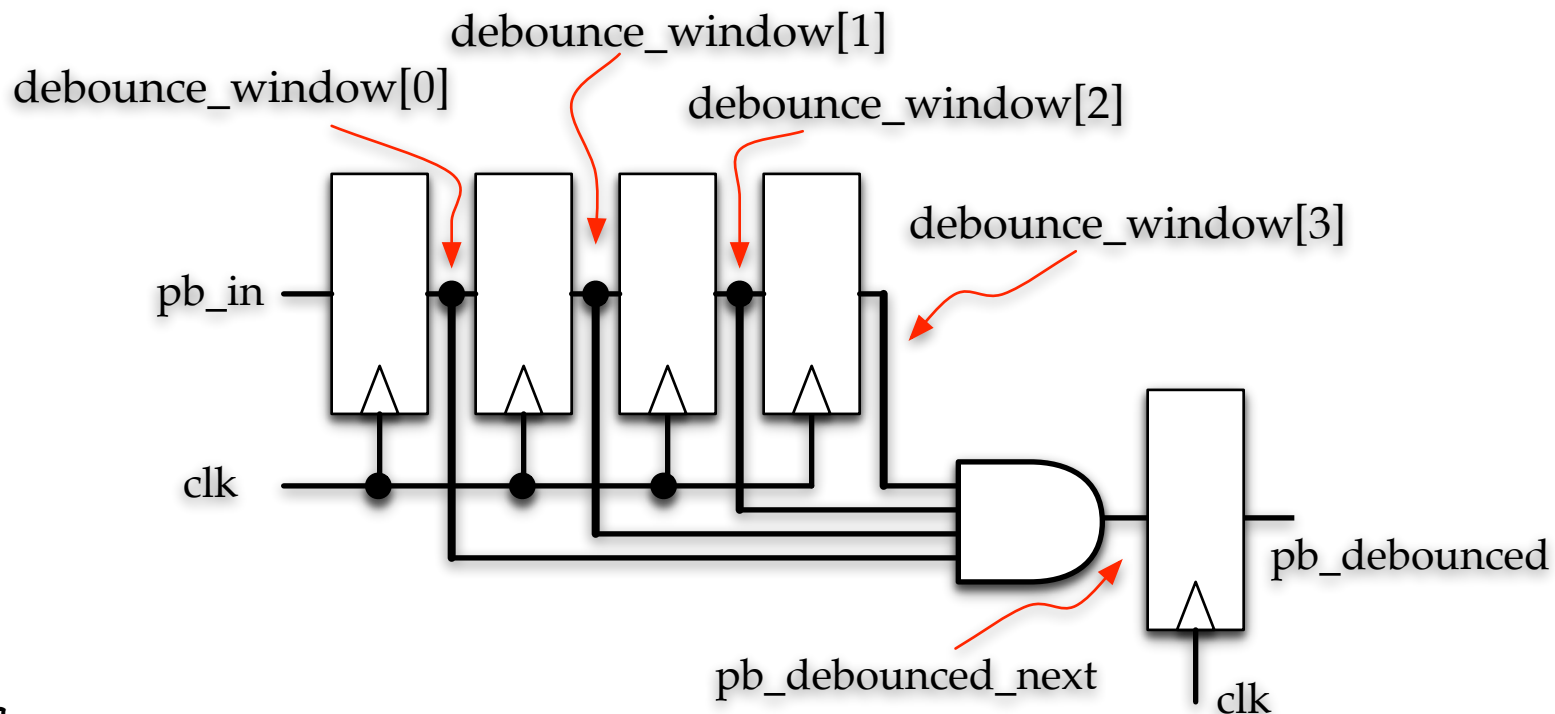
- Push button contains a meta spring which will cause signal bounce when switching
 - A random number of unwanted signal pulses
 - Usually in μs range
 - FPGA is sensitive to pulses down to **nsec** range



Debounce Circuits

- Spec

- 4-bit shift register clocked at 100Hz
- Input is the push button input
- When all 4 bits of the registers are high the output of the debounce circuit changes to high



debounce_circuit.v

```
`include "global.v"
module debounce_circuit(
    clk, // clock control
    rst_n, // reset
    pb_in, //push button input
    pb_debounced // debounced push button output
);

// declare the I/Os
input clk; // clock control
input rst_n; // reset
input pb_in; //push button input
output pb_debounced; // debounced push button output
reg pb_debounced; // debounced push button output

// declare the internal nodes
reg [3:0] debounce_window; // shift register flip flop
reg pb_debounced_next; // debounce result
```

```
// Shift register
always @(posedge clk or negedge rst_n)
    if (~rst_n)
        debounce_window <= 4'd0;
    else
        debounce_window <= {debounce_window[2:0], pb_in};

// debounce circuit
always @*
    if (debounce_window == 4'b1111)
        pb_debounced_next = 1'b1;
    else
        pb_debounced_next = 1'b0;

always @(posedge clk or negedge rst_n)
    if (~rst_n)
        pb_debounced <= 1'b0;
    else
        pb_debounced <= pb_debounced_next;

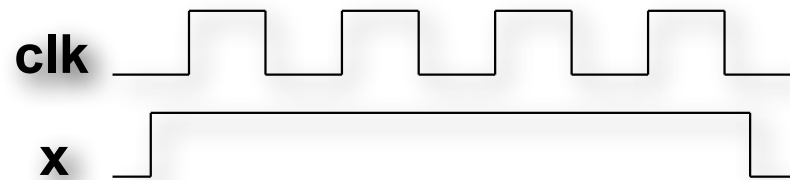
endmodule
```

Level Control vs. Pulse Control

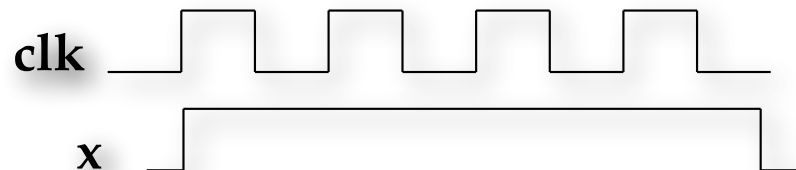
- You can use *level* to control circuits to operate at certain function continuously for a period of time
 - *enable* control
- You can also use *pulse* to trigger circuits for one certain function once
 - state transition control in FSM
 - no pulse: remain at the same state
 - pulse received: jump to next state

One-Pulse Generation

- When one presses the push button for a short moment, the time that the switch is closed (ms range) is usually much longer than one clock period (μs or ns range)



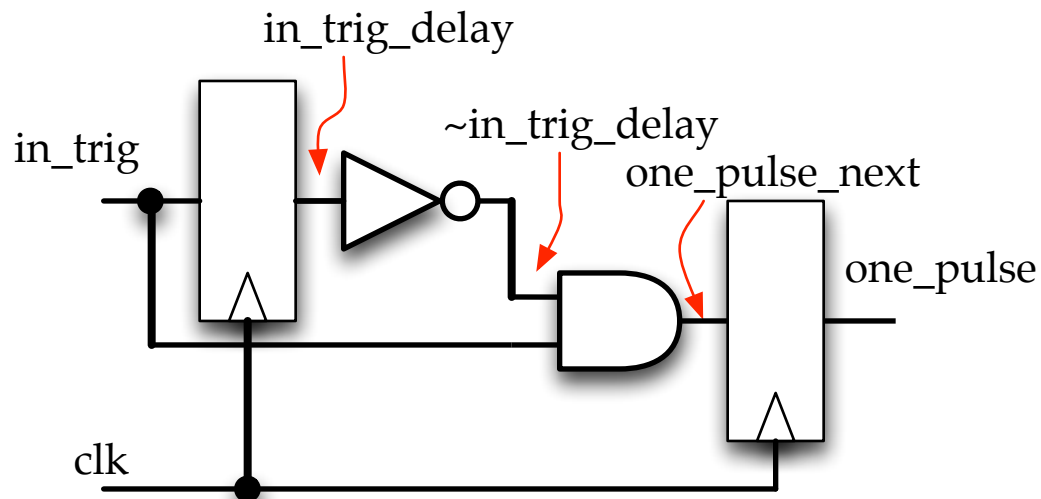
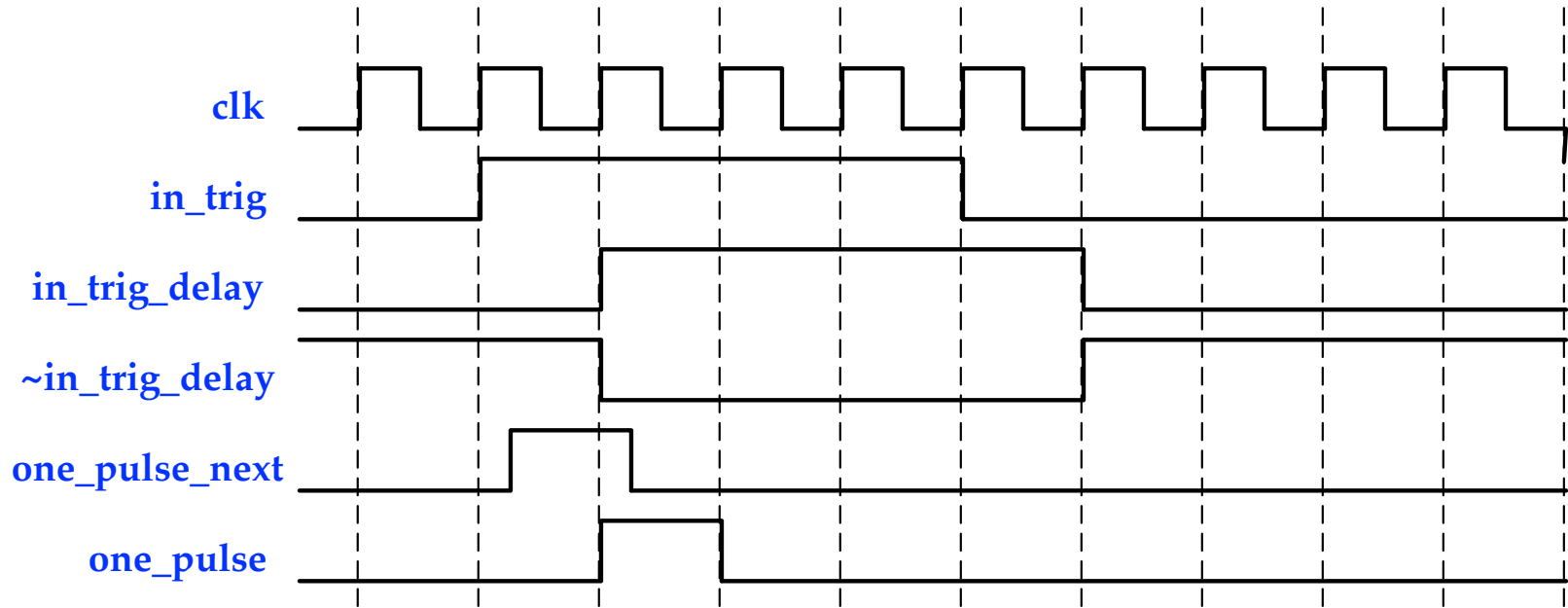
- The one-pulse circuit generates only a one-clock-period-long pulse every time the push button is hit, independent of the time one keeps the button pressed



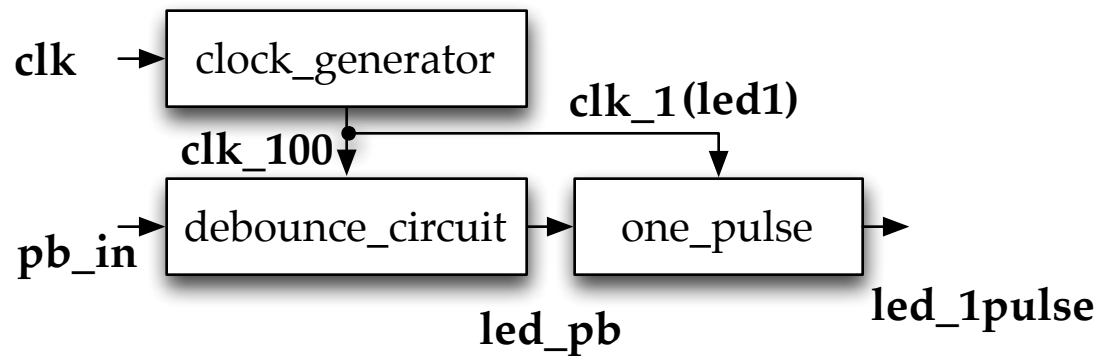
one pulse control signal



One-Pulse Generation



example: one_pulse



clock_generator.v

```

`include "global.v"
module clock_generator(
    clk, // clock from crystal
    rst_n, // active low reset
    clk_1, // generated 1 Hz clock
    clk_100 // generated 100 Hz clock
);

// Declare I/Os
input clk; // clock from crystal
input rst_n; // active low reset
output clk_1; // generated 1 Hz clock
output clk_100; // generated 100 Hz clock
reg clk_1; // generated 1 Hz clock
reg clk_100; // generated 100 Hz clock

// Declare internal nodes
reg [DIV_BY_50M_BIT_WIDTH-1:0]
count_50M, count_50M_next;
reg [DIV_BY_500K_BIT_WIDTH-1:0]
count_500K, count_500K_next;
reg clk_1_next;
reg clk_100_next;

```

```

// Clock Divider: Counter operation
always @*
    if (count_50M == `DIV_BY_50M-1)
        begin
            count_50M_next = `DIV_BY_50M_BIT_WIDTH'd0;
            clk_1_next = ~clk_1;
        end
    else
        begin
            count_50M_next = count_50M + 1'b1;
            clk_1_next = clk_1;
        end

// Counter flip-flops
always @(posedge clk or negedge rst_n)
    if (~rst_n)
        begin
            count_50M <= `DIV_BY_50M_BIT_WIDTH'b0;
            clk_1 <= 1'b0;
        end
    else
        begin
            count_50M <= count_50M_next;
            clk_1 <= clk_1_next;
        end
    ...
endmodule

```

Remember to use clk_100 in real design !!!

one_pulse.v

```

module one_pulse(
  clk, // clock input
  rst_n, //active low reset
  in_trig, // input trigger
  out_pulse // output one pulse
);

// Declare I/Os
input clk; // clock input
input rst_n; //active low reset
input in_trig; // input trigger
output out_pulse; // output one pulse
reg out_pulse; // output one pulse

// Declare internal nodes
reg in_trig_delay;

// Buffer input
always @(posedge clk or negedge rst_n)
  if (~rst_n)
    in_trig_delay <= 1'b0;
  else
    in_trig_delay <= in_trig;

```

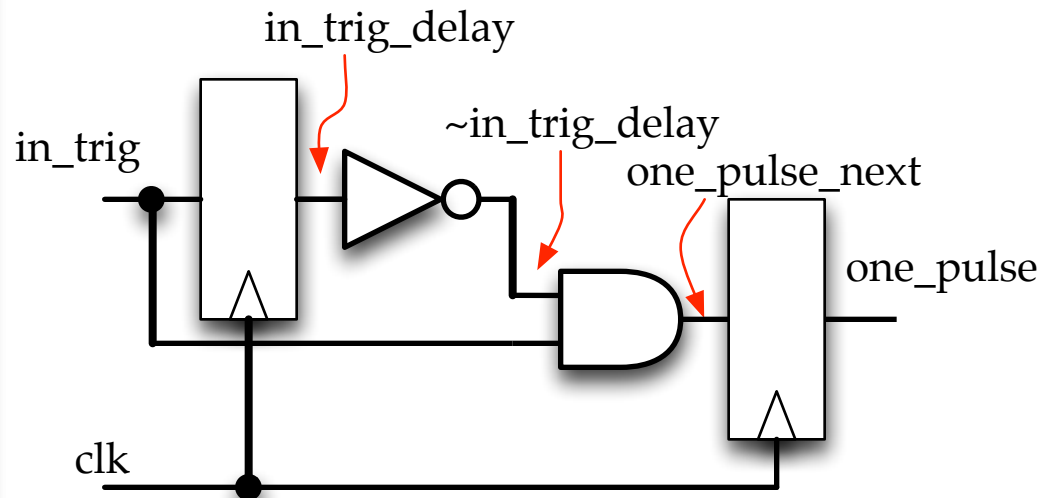
```

// Pulse generation
assign out_pulse_next = in_trig &
(~in_trig_delay);

always @(posedge clk or negedge rst_n)
  if (~rst_n)
    out_pulse <= 1'b0;
  else
    out_pulse <= out_pulse_next;

endmodule

```



top.v

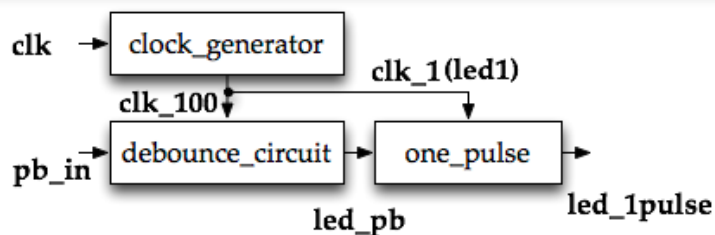
```

module top(
  clk, // Clock from crystal
  rst_n, //active low reset
  pb_in, //push button input
  led_1, // 1Hz divided clock
  led_pb, // LED display output (push button)
  led_1pulse // LED display output (1 pulse)
);

// Declare I/Os
input clk; // Clock from crystal
input rst_n; //active low reset
input pb_in; //push button input
output led_1; // 1Hz divided clock
output led_pb; // LED display output (push button)
output led_1pulse; // LED display output (1 pulse)

// Declare internal nodes
wire pb_debounced; // push button debounced out

```



```

// Clock generator module
clock_generator U_cg(
  .clk(clk), // clock from crystal
  .rst_n(rst_n), // active low reset
  .clk_1(led_1), // generated 1 Hz clock
  .clk_100(clk_100) // generated 100 Hz clock
);

// debounce circuit
debounce_circuit U_dc(
  .clk(clk_100), // clock control
  .rst_n(rst_n), // reset
  .pb_in(pb_in), //push button input
  .pb_debounced(led_pb) // debounced push button out
);

// 1 pulse generation circuit
one_pulse U_op(
  .clk(led_1), // clock input
  .rst_n(rst_n), //active low reset
  .in_trig(led_pb), // input trigger
  .out_pulse(led_1pulse) // output one pulse
);
endmodule

```



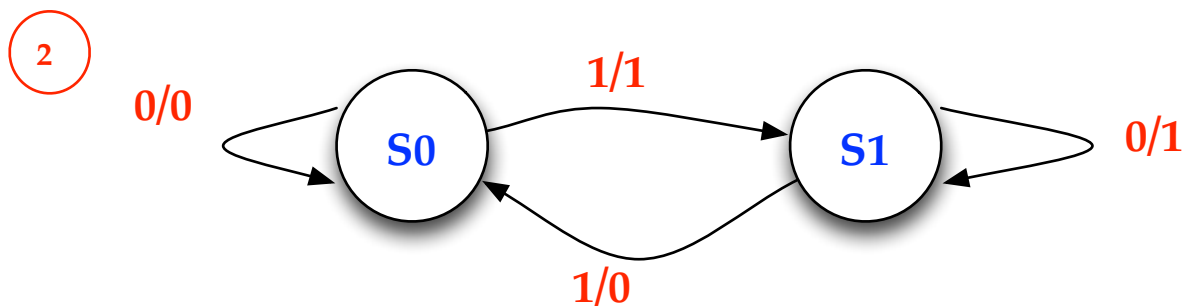
Stopwatch Example

Stopwatch with FSM

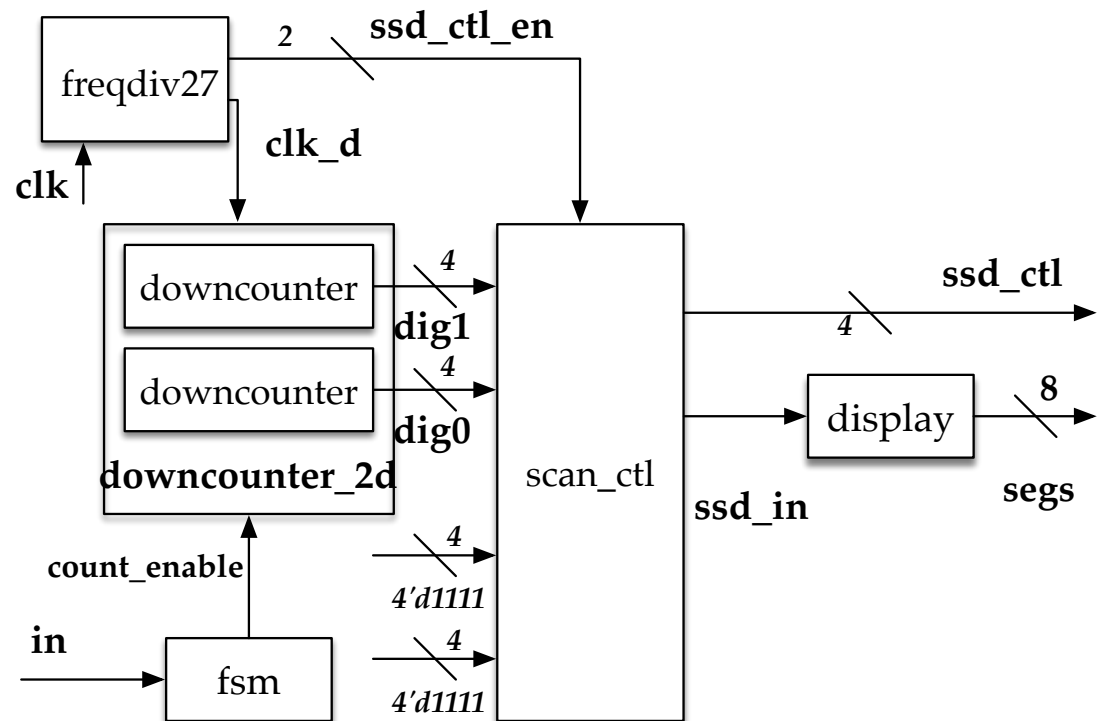
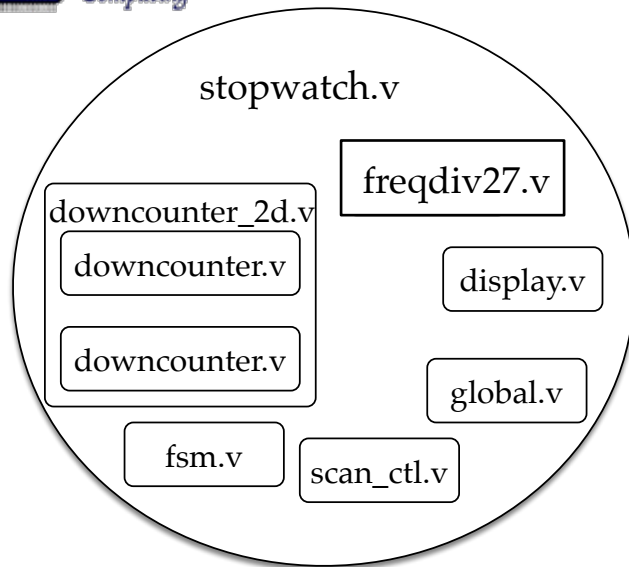
- 1 • stopwatch function with 1-bit control for stop (S0), start(S1)

Inputs: pressed

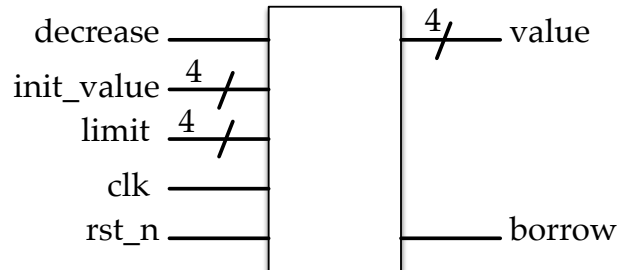
Outputs: count_enable



stopwatch



downcounter.v (1 / 2)



```

`define BCD_BIT_WIDTH 4
`define BCD_ZERO 4'd0
`define INCREMENT 1'b1
module downcounter(
    value, // counter value
    borrow, // borrow indicator for counter to next stage
    clk, // global clock
    rst_n, // active low reset
    decrease, // decrease input from previous stage of counter
    init_value, // initial value for the counter
    limit // limit for 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_n; // active low 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

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

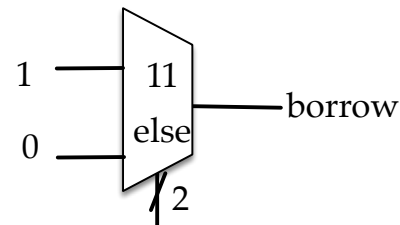
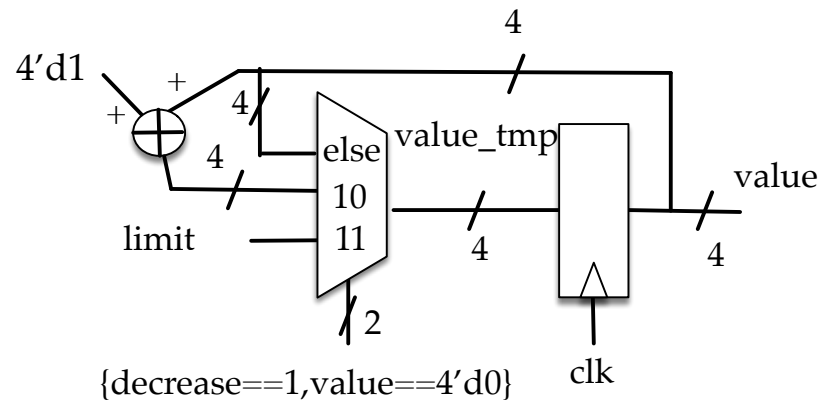
```

downcounter.v (2/2)

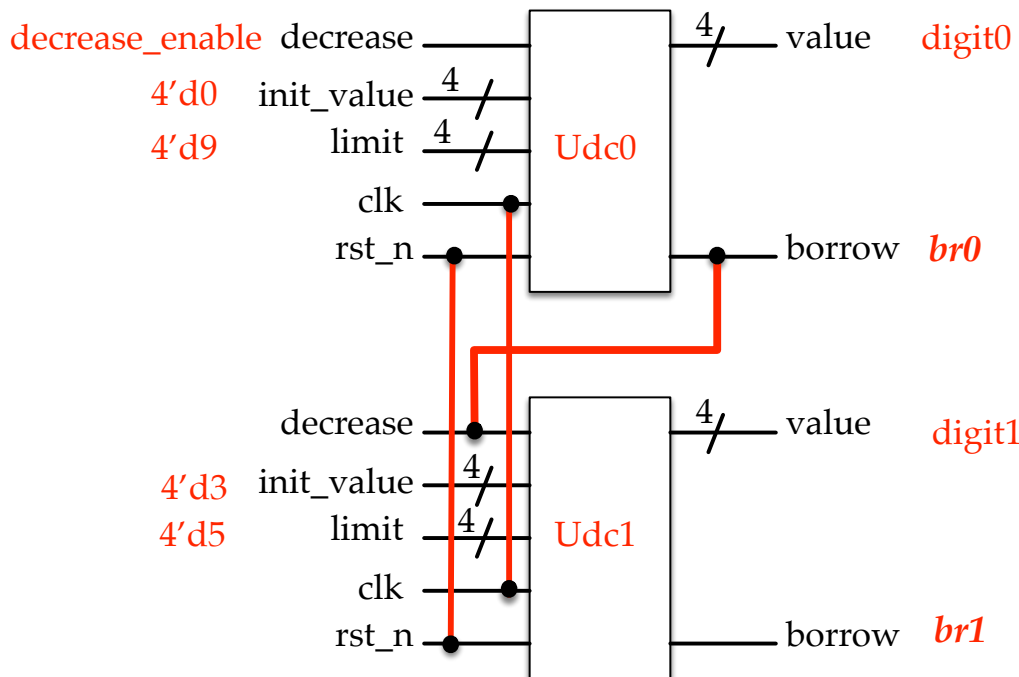
```
// Combinational logics
always @*
  if (value==`BCD_ZERO && decrease)
    begin
      value_tmp = limit;
      borrow = `ENABLED;
    end
  else if (decrease)
    begin
      value_tmp = value - `INCREMENT;
      borrow = `DISABLED;
    end
  else
    begin
      value_tmp = value;
      borrow =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

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

endmodule
```



downcounter_2d.v (1/3)



```

`define BCD_BIT_WIDTH 4
`define ENABLED 1
`define DISABLED 0
`define INCREMENT 1'b1
`include "global.v"
module downcounter_2d(
    digit1, // 2nd digit of the down counter
    digit0, // 1st digit of the down counter
    clk, // global clock
    rst_n, // active low reset
    en // enable/disable for stopwatch
);

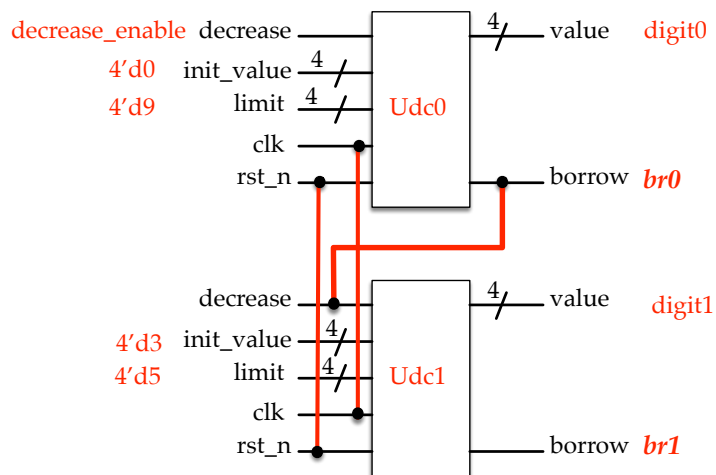
    output [`BCD_BIT_WIDTH-1:0] digit1;
    output [`BCD_BIT_WIDTH-1:0] digit0;
    input clk; // global clock
    input rst_n; // active low reset
    input en; // enable/disable for stopwatch

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

    assign decrease_enable = en &&
        (~((digit0==`BCD_ZERO) &&
            (digit1==`BCD_ZERO) &&
            (digit2==`BCD_ZERO)));

```

downcounter_2d.v (2/2)



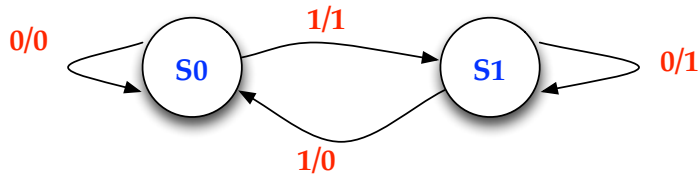
```

// 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_n(rst_n), // low active reset
    .decrease(decrease_enable), // decrease input from previous stage of counter
    .init_value(^BCD_ZERO), // initial value for the counter
    .limit(^BCD_NINE) // limit for the counter
);

downcounter Udc1(
    .value(digit1), // counter value
    .borrow(br1), // borrow indicator for counter to next stage
    .clk(clk), // global clock signal
    .rst_n(rst_n), // low active reset
    .decrease(br0), // decrease input from previous stage of counter
    .init_value(^BCD_THREE), // initial value for the counter
    .limit(^BCD_FIVE) // limit for the counter
);

```

fsm.v



```

`include "global.v"
module fsm(
    count_enable, // if counter is enabled
    in, //input control
    clk, // global clock signal
    rst_n // low active reset
);

// outputs
output count_enable; // if counter is enabled

// inputs
input clk; // global clock signal
input rst_n; // low active reset
input in; //input control

reg count_enable; // if counter is enabled
reg state; // state of FSM
reg next_state; // next state of FSM

```

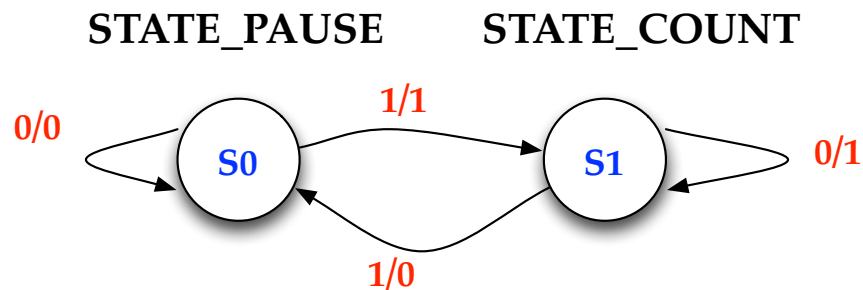
fsm.v

```
// FSM state decision
always @*
case (state)
`STAT_PAUSE:
  if (in)
  begin
    next_state = `STAT_COUNT;
    count_enable = `ENABLED;
  end
  else
  begin
    next_state = `STAT_PAUSE;
    count_enable = `DISABLED;
  end
`STAT_COUNT:
  if (in)
  begin
    next_state = `STAT_PAUSE;
    count_enable = `DISABLED;
  end
  else
  begin
    next_state = `STAT_COUNT;
    count_enable = `ENABLED;
  end
end
```

```
default:
begin
  next_state = `STAT_DEF;
  count_enable = `DISABLED;
end
Endcase
```

```
// FSM state transition
always @(posedge clk or negedge rst_n)
if (~rst_n)
  state <= `STAT_PAUSE;
else
  state <= next_state;

endmodule
```



stopwatch.v

```
`include "global.v"
module stopwatch(
    segs, // 7 segment display control
    ssd_ctl, // scan control for 7-segment display
    clk, // clock
    rst_n, // low active reset
    in // input control for FSM
);

output [^SSD_BIT_WIDTH-1:0] segs; // 7-segment display control
output [^SSD_DIGIT_NUM-1:0] ssd_ctl; // scan control for ssd
input clk; // clock
input rst_n; // low active reset
input in; // input control for FSM

wire [^SSD_SCAN_CTL_BIT_WIDTH-1:0] ssd_ctl_en; // divided output for ssd ctl
wire clk_d; // divided clock

wire count_enable; // if count is enabled

wire [^BCD_BIT_WIDTH-1:0] dig0,dig1; // second counter output
```

stopwatch.v

```

//*****
// Functional block
//*****
// frequency divider 1/(2^27)
freqdiv27 U_FD(
    .clk_out(clk_d), // divided clock
    .clk_ctl(ssd_ctl_en), // divided clock for scan control
    .clk(clk), // clock from the crystal
    .rst_n(rst_n) // low active reset
);

FSM U_fsm(
    .count_enable(count_enable), // if counter is enabled
    .in(in), //input control
    .clk(clk_d), // global clock signal
    .rst_n(rst_n) // low 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_n(rst_n), // low active reset
    .en(count_enable) // enable/disable for the stopwatch
);

```

```

//*****
// Display block
//*****
// Scan control
scan_ctl U_SC(
    .ssd_ctl(ssd_ctl), // ssd display control signal
    .ssd_in(ssd_in), // output to ssd display
    .in0(4'b1111), // 1st input
    .in1(4'b1111), // 2nd input
    .in2(dig1), // 3rd input
    .in3(dig0), // 4th input
    .ssd_ctl_en(ssd_ctl_en) // divided clock for scan control
);

// binary to 7-segment display decoder
display U_display(
    .segs(segs), // 7-segment display output
    .bin(ssd_in) // BCD number input
);

endmodule

```