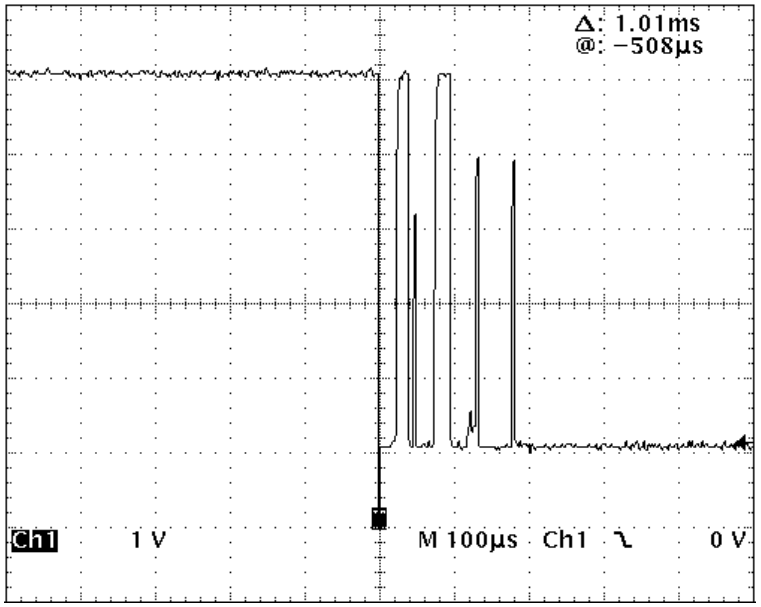
Using Push Buttons

Hsi-Pin Ma

http://lms.nthu.edu.tw/course/24953
Department of Electrical Engineering
National Tsing Hua University

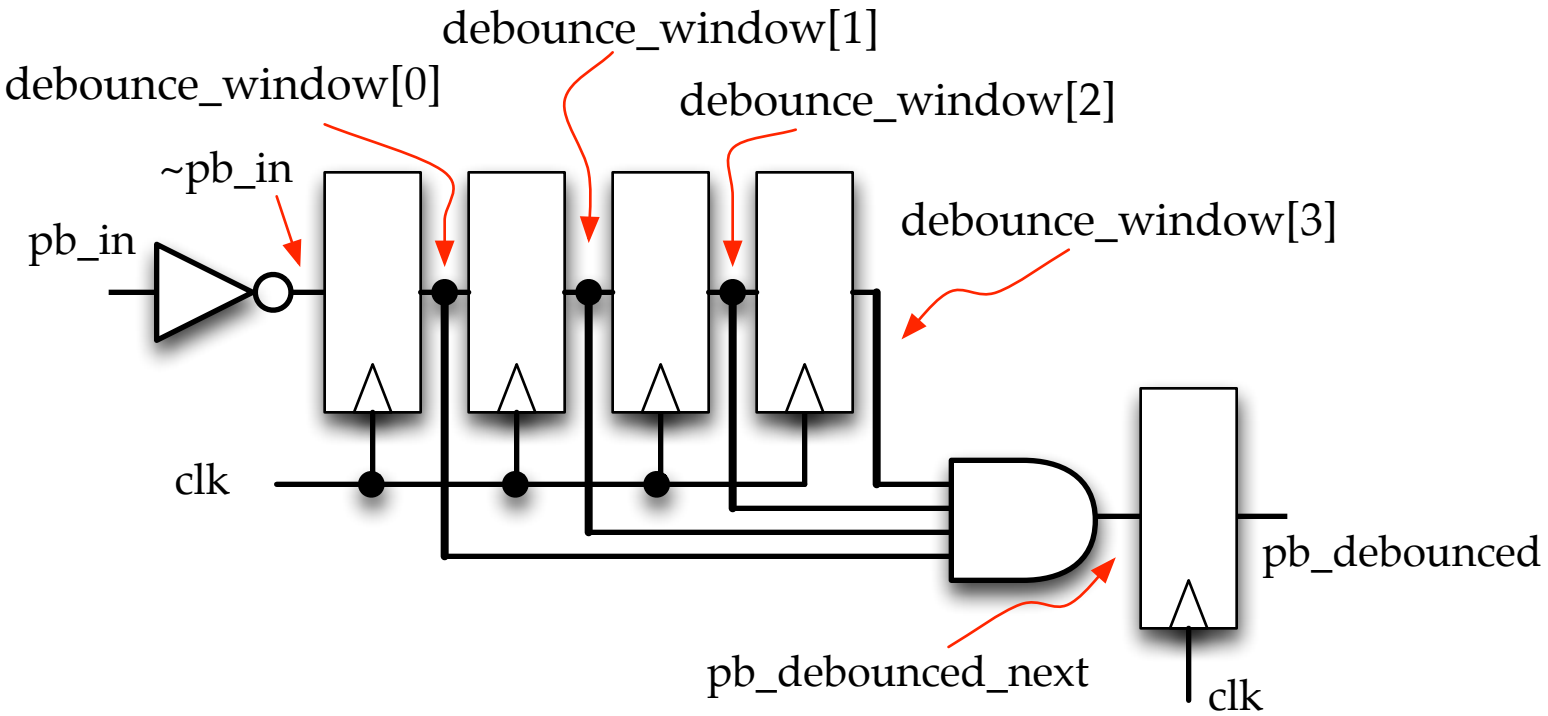
Switch Contact Bounce

- 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 ns range



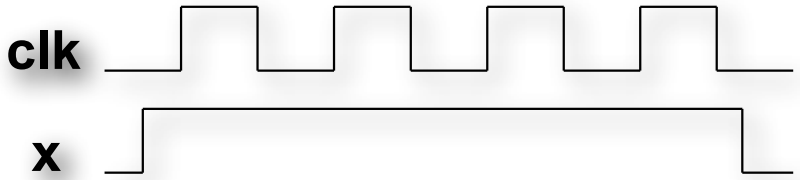
Debounce Circuits

- 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

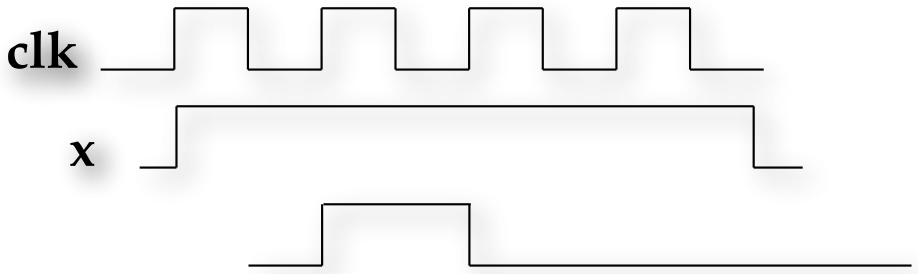


One-Pulse Generation

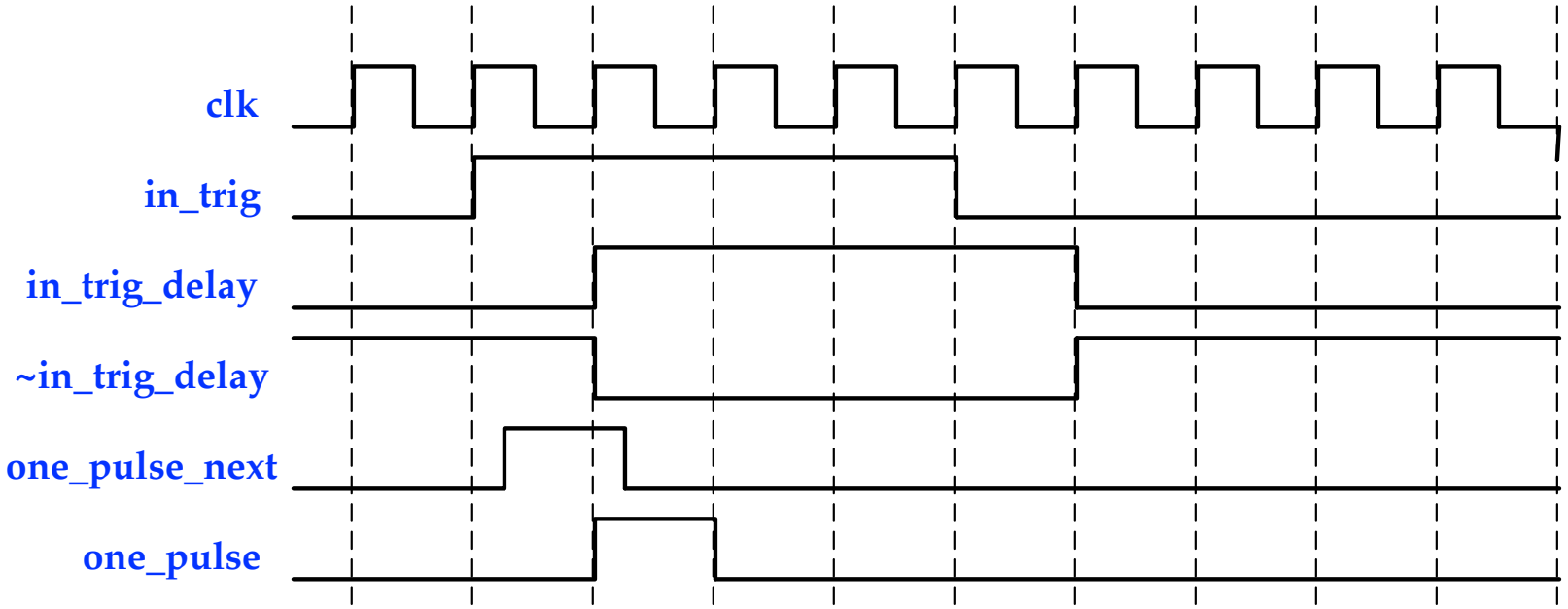
- When one presses the push button for a short moment, the time that the switch is closed (ms range) is usually much longer than one clock period (μs or ns range)

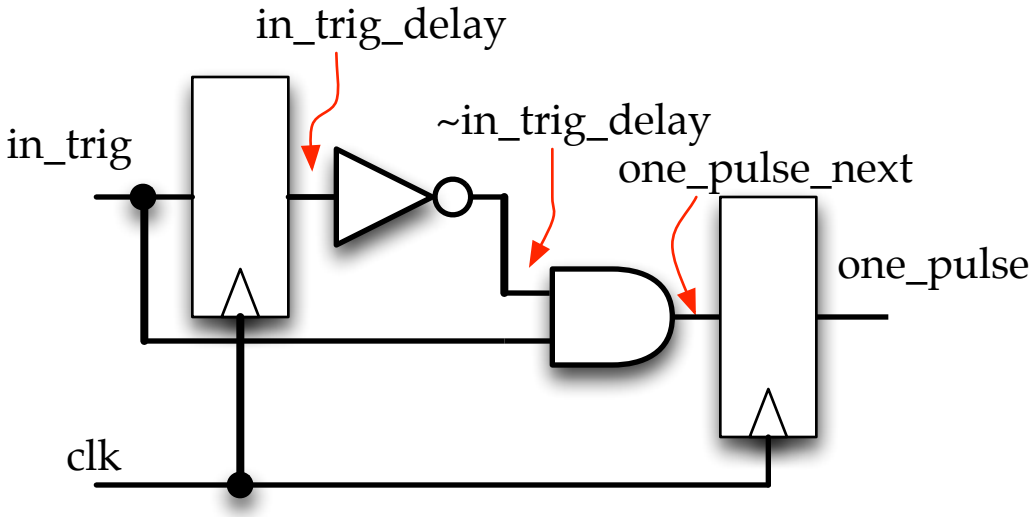


- The one-pulse circuit generates only a one-clock-period-long pulse every time the push button is hit, independent of the time one keeps the button pressed

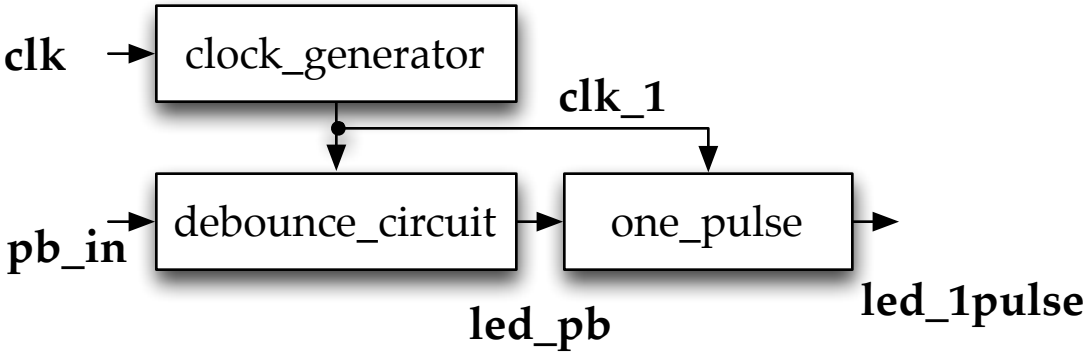


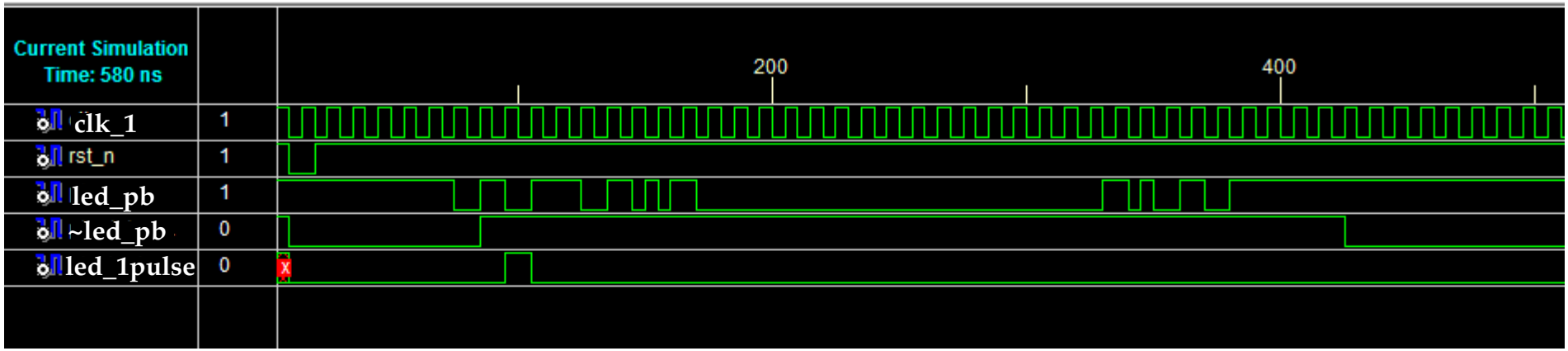
One-Pulse Generation





Simulation


clk
clock_generator
clk_1
pb_in
debounce_circuit
one_pulse
led_pb
led_1pulse


Current Simulation Time: 580 ns
200
400
clk_1
1
rst_n
1
led_pb
1
~led_pb
0
led_1pulse
0

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_20M_BIT_WIDTH-1:0]
count_20M, count_20M_next;
reg [`DIV_BY_200K_BIT_WIDTH-1:0]
count_200K, count_200K_next;
reg clk_1_next;
reg clk_100_next;
```

Remember to use clk_100 in real design !!!

```
// Clock Divider: Counter operation
always @*
  if (count_20M == `DIV_BY_20M-1)
  begin
    count_20M_next = `DIV_BY_20M_BIT_WIDTH'd0;
    clk_1_next = ~clk_1;
  end
  else
  begin
    count_20M_next = count_20M + 1'b1;
    clk_1_next = clk_1;
  end

// Counter flip-flops
always @(posedge clk or negedge rst_n)
  if (~rst_n)
  begin
    count_20M <=`DIV_BY_20M_BIT_WIDTH'b0;
    clk_1 <=1'b0;
  end
  else
  begin
    count_20M <= count_20M_next;
    clk_1 <= clk_1_next;
  end
...
endmodule
```

debounce_circuit.v

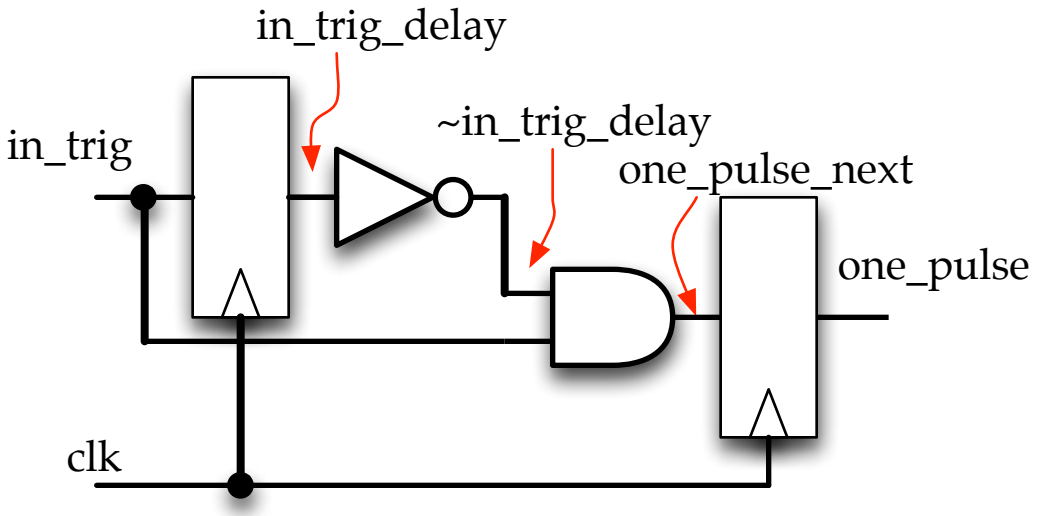
```
`include "global.v"
module debounce_circuit(
  clk, // clock control
  rst_n, // reset
  pb_in, //push button input
  pb_debounced // debounced push button output
);

// 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
```

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
```



Stopwatch Example

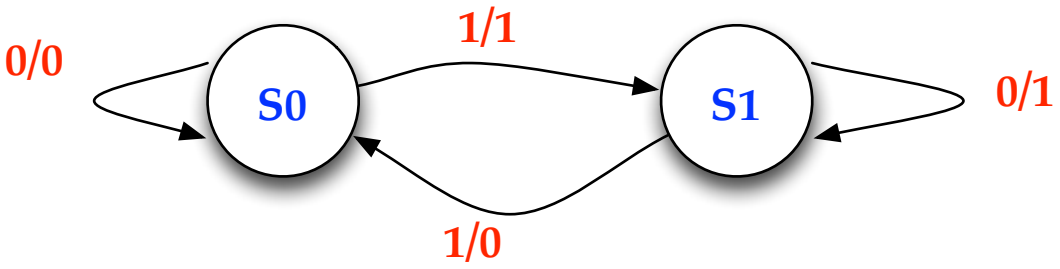
Stopwatch with FSM

1

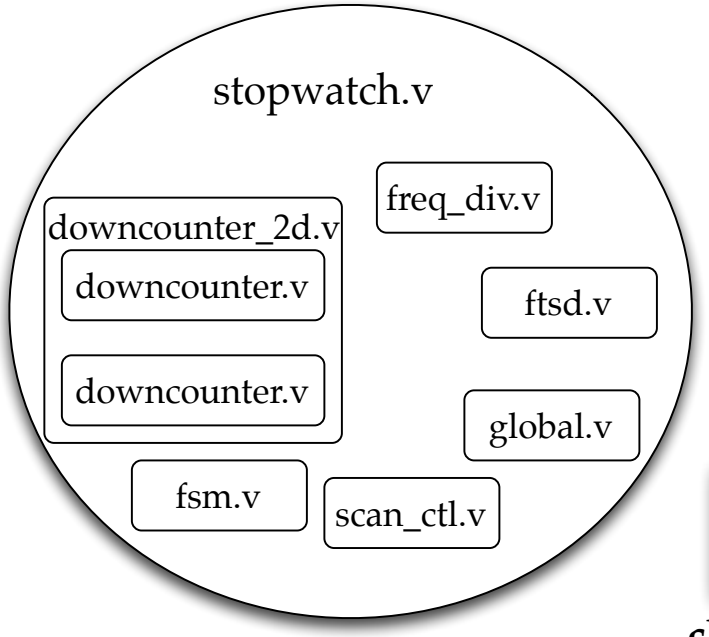
- stopwatch function with 1-bit control for stop (S0), start(S1)

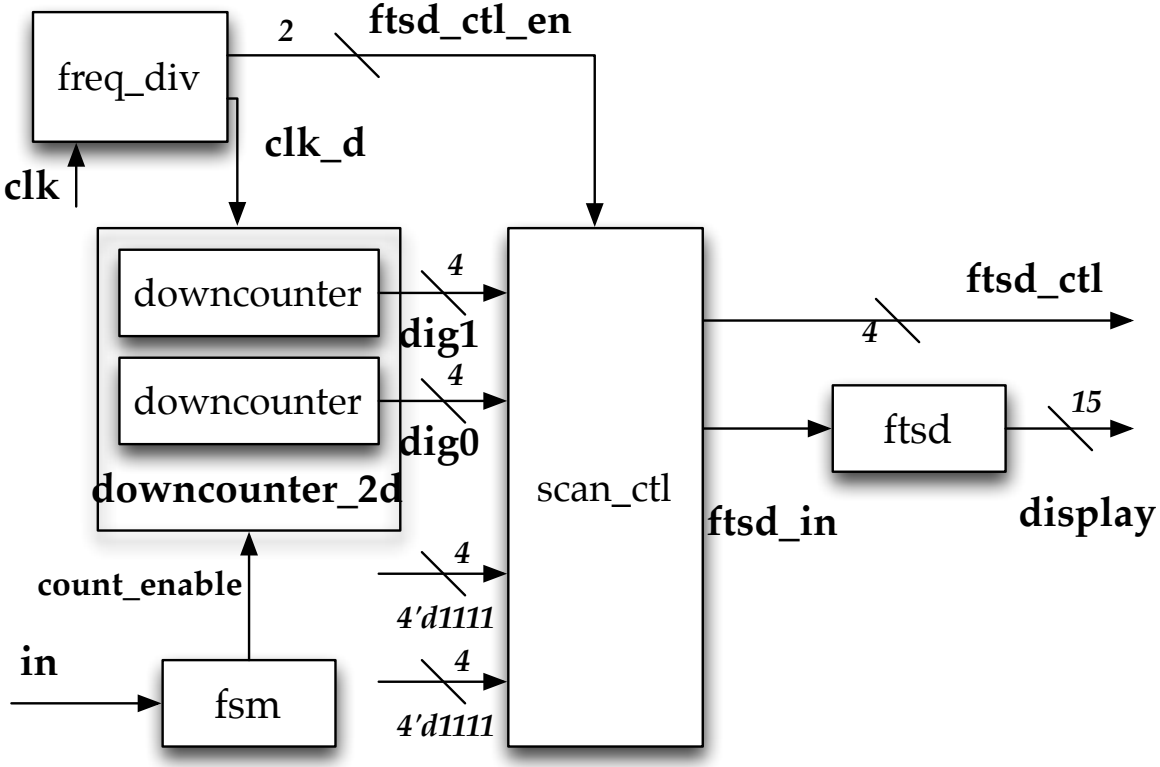
Inputs: pressed **Outputs: count_enable**

2

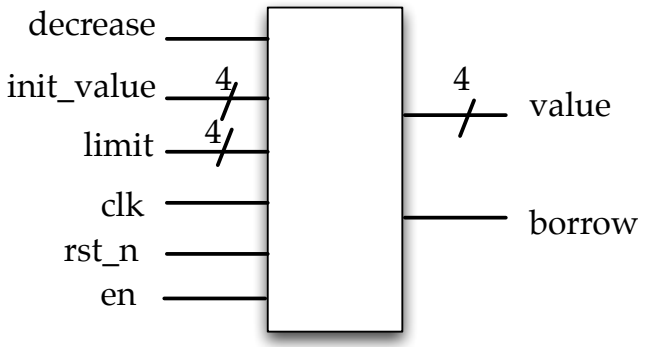


stopwatch


stopwatch.v
freq_div.v
downcounter_2d.v
downcounter.v
downcounter.v
ftsd.v
global.v
fsm.v
scan_ctl.v


freq_div
2
ftsd_ctl_en
clk_d
clk
downcounter
downcounter
4
dig1
4
dig0
downcounter_2d
scan_ctl
ftsd_ctl
4
ftsd
15
display
ftsd_in
count_enable
4
4'd1111
in
fsm
4
4'd1111

downcounter.v (1/2)

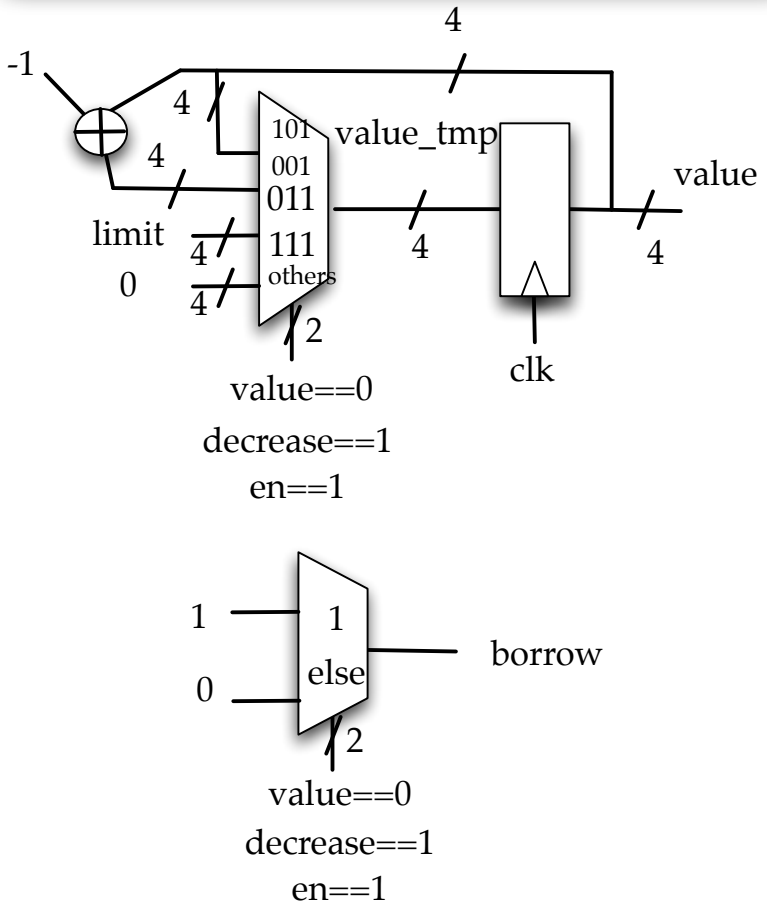



```
`define BCD_BIT_WIDTH 4
`define BCD_ZERO 4'd0
`define INCREMENT 1'b1
module downcounter(
  value,  // counter value
  borrow,  // borrow indicator for counter to next stage
  clk,  // global clock
  rst_n,  // active low reset
  decrease,  // decrease input from previous stage of counter
  init_value,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

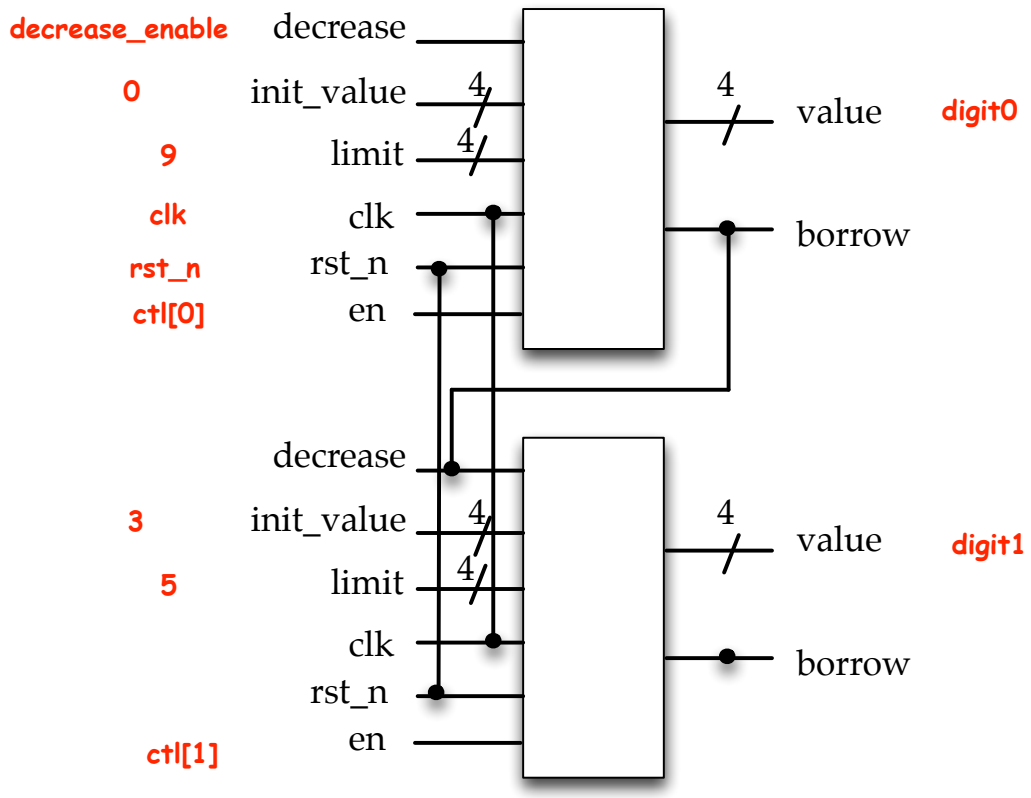
downcounter.v (2/2)

```
// 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 negedge rst_n)
  if (~rst_n) value <= init_value;
  else value <= value_tmp;

endmodule
```



downcounter_2d.v (1/3)

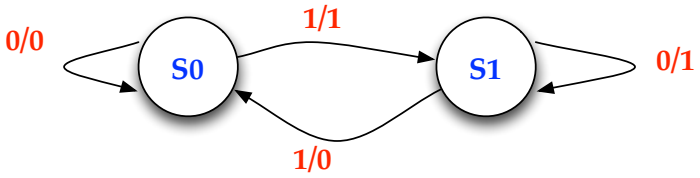



```
`define BCD_BIT_WIDTH 4
`define ENABLED 1
`define DISABLED 0
`define INCREMENT 1'b1
`include "global.v"
module downcounter_2d(
  digit1,  // 2nd digit of the down counter
  digit0,  // 1st digit of the down counter
  clk,  // global clock
  rst_n,  // active low reset
  en  // enable/disable for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

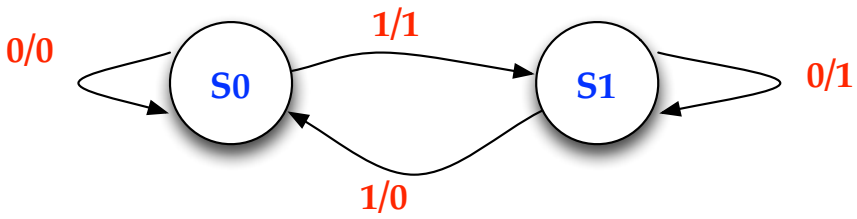
fsm.v




```
`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_DEF:
      if (in)
      begin
        next_state = `STAT_COUNT;
        count_enable = `ENABLED;
      end
      else
      begin
        next_state = `STAT_DEF;
        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
    `STAT_PAUSE:
      if (in)
      begin
        next_state = `STAT_COUNT;
        count_enable = `ENABLED;
      end
      else
      begin
        next_state = `STAT_PAUSE;
        count_enable = `DISABLED;
      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_DEF;
  else
    state <= next_state;

endmodule
```



stopwatch.v

```
`include "global.v"
module stopwatch(
  display, // 14 segment display control
  ftsd_ctl, // scan control for 14-segment display
  clk, // clock
  rst_n, // low active reset
  in // input control for FSM
);

output [`FTSD_BIT_WIDTH-1:0] display; // 14 segment display control
output [`FTSD_DIGIT_NUM-1:0] ftsd_ctl; // scan control for ftsd
input clk; // clock
input rst_n; // low active reset
input in; // input control for FSM

wire [`FTSD_SCAN_CTL_BIT_WIDTH-1:0] ftsd_ctl_en; // divided output for ftsd 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/(2^25)
freq_div 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(
  .ftsd_ctl(ftsd_ctl), // ftsd display control signal
  .ftsd_in(ftsd_in), // output to ftsd display
  .in0(4'b1111), // 1st input
  .in1(4'b1111), // 2nd input
  .in2(dig1), // 3rd input
  .in3(dig0),  // 4th input
  .ftsd_ctl_en(ftsd_ctl_en) // divided clock for scan control
);

// binary to 14-segment display decoder
ftsd U_display(
  .display(display), // 14-segment display output
  .in(ftsd_in)  // BCD number input
);

endmodule
```