Counters

Hsi-Pin Ma

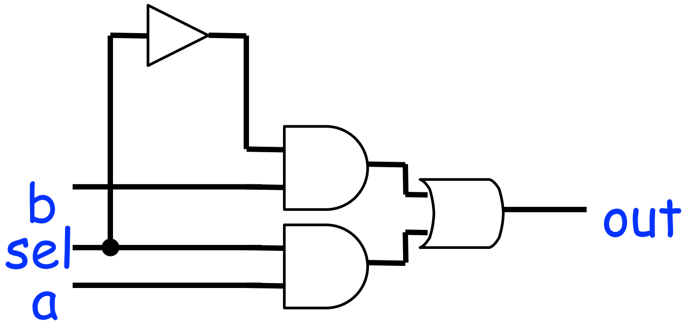
http://lms.nthu.edu.tw/course/43639

Department of Electrical Engineering
National Tsing Hua University

Testbenches

- Self-checking testbenches
- Self-checking testbench with testvectors

Self-Checking Testbenches (1/2)



```
module smux(out, a, b, sel);
output out;
input a,b,sel;

assign out = (a&sel) | (b&(~sel));

endmodule
```

Correct Version

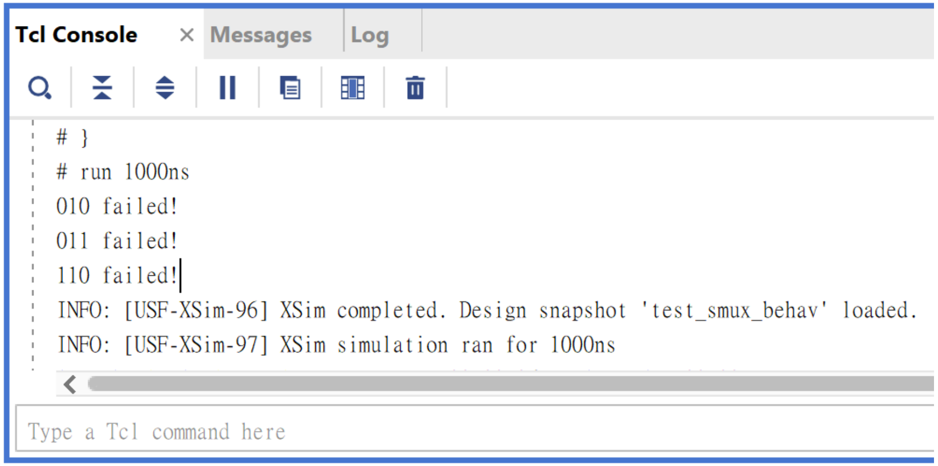
```
module smux(xout, a, b, sel);
output xout;
input a,b,sel;

assign xout = (a&sel) | (b&sel);

endmodule
```

Wrong Version

Self-Checking Testbenches (2/2)

```
module test_smux;
wire OUT;
reg A,B,SEL;

smux U0(.out(OUT),.a(A),.b(B),.sel(SEL));

initial
begin
  A=0;B=0;SEL=0; #10;  //apply input; wait
  if (OUT != 0) $display (“000 failed”); // check
  A=0;B=0;SEL=1; #10;
  if (OUT != 0) $display (“001 failed”);
  A=0;B=1;SEL=0; #10;
  if (OUT != 1) $display (“010 failed”);
  A=0;B=1;SEL=1; #10;
  if (OUT != 0) $display (“011 failed”);
  A=1;B=0;SEL=0; #10;
  if (OUT != 0) $display (“100 failed”);
  A=1;B=0;SEL=1; #10;
  if (OUT != 1) $display (“101 failed”);
  A=1;B=1;SEL=0; #10;
  if (OUT != 1) $display (“110 failed”);
  A=1;B=1;SEL=1; #10;
  if (OUT != 1) $display (“111 failed”);
end
endmodule
```

A	B	SEL	OUT	XOUT
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1



```
# }
# run 1000ns
010 failed!
011 failed!
110 failed!
INFO: [USF-XSim-96] XSim completed. Design snapshot 'test_smux_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Self-Checking Testbench with Testvectors

- Testvector file
  - Text file containing vectors of input_output from truth table

golden.txt

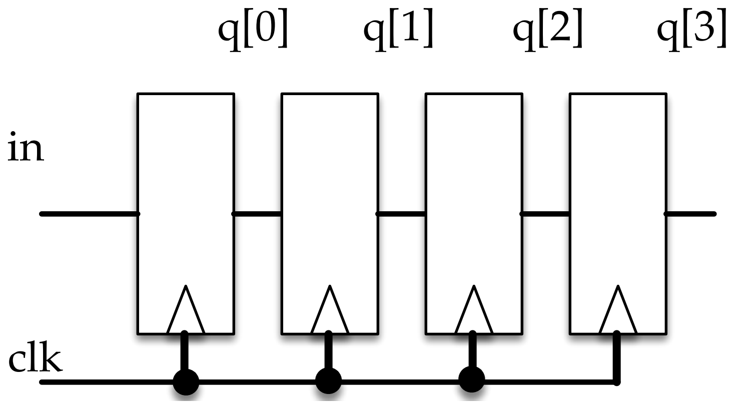
```
//a b sel_out
000_0
001_0
010_1
011_0
100_0
101_1
110_1
111_1
```

Self-Checking Testbench with Testvectors

absolute directory of your golden pattern file

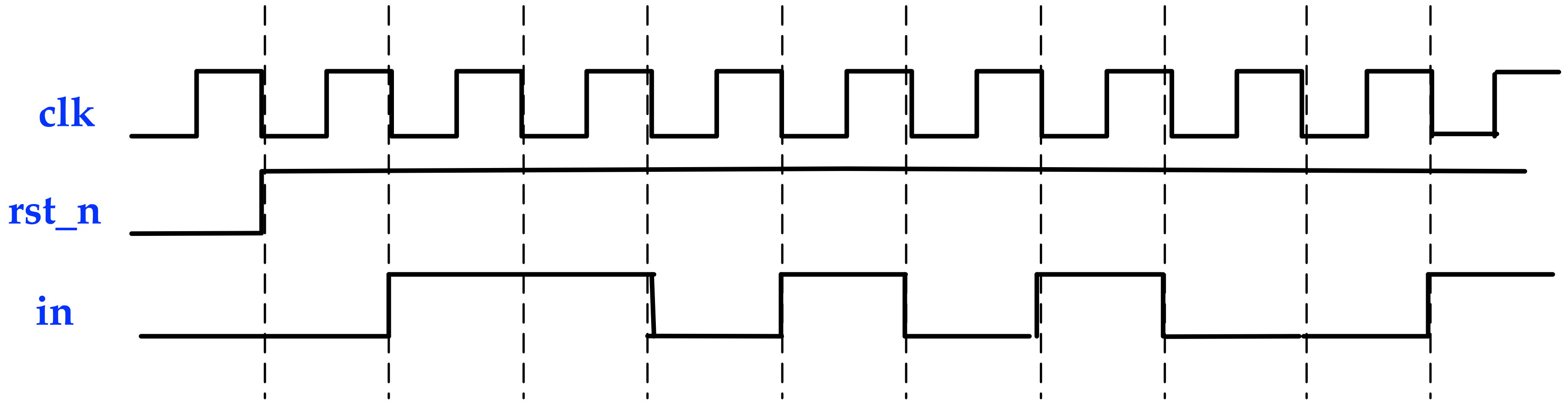
```
module test_smux;
wire OUT;
reg A,B,SEL,OUT_EXPECTED;
reg [7:0] vectornum;
reg [3:0] testvectors[7:0];

smux U0(.out(OUT),.a(A),.b(B),.sel(SEL));

initial
begin
 $readmemb("C:/Users/hp/LD/smux_testvector/smux_testvector.srcs/sim_1/new/golden.txt", testvectors);
 vectornum=0;
end

initial
begin
 for (vectornum=0; vectornum<8; vectornum=vectornum+1)
 begin
  {A,B,SEL,OUT_EXPECTED} = testvectors[vectornum]; $10; //apply input; wait
  if (OUT != OUT_EXPECTED) $display("%b%b%b failed", A, B, SEL);// check
 end
end
endmodule
```

Sequential Logic Testbench



```
module shiftref(q, in, clk, rst_n);
output [3:0] q;
input in,clk,rst_n;

always@(posedge clk or negedge rst_n)
 begin
   if (~rst_n)
     q <= 4’d0;
   else
     q <= {q[2:0],in];
 end

endmodule
```

Testbenches (1/3)



Testbenches (2/3)

```
module t_shiftref;
wire [3:0] q;
reg in, clk, rst_n;
reg [3:0] vectornum;
reg [4:0] testvectors[9:0];
reg [3:0] q_expected;

shiftreg U0(.q(q),.in(in),.clk(clk),.rst_n(rst_n));

initial
begin
 $readmemb(“C:/Users/hp/LD/shiftreg/shiftreg.srcs/sim_1/new/golden.txt”, testvectors);
 vectornum=0;
end
```

golden.txt

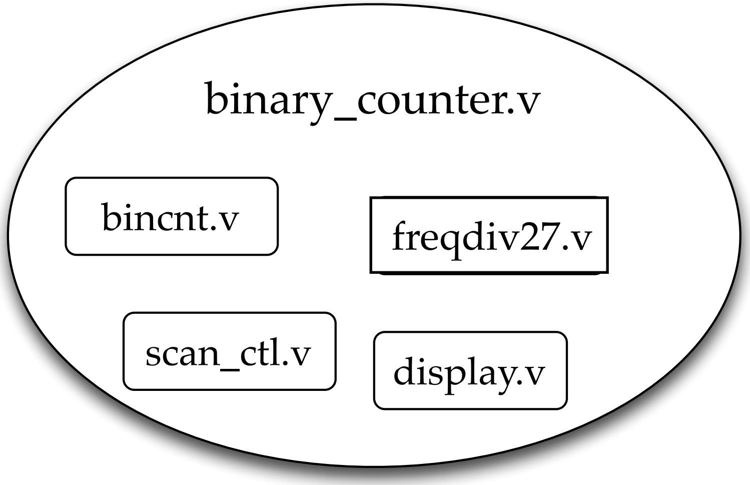
```
//in q_expected
0_0000
1_0000
1_0001
0_0011
1_0110
0_1100
1_1000
0_0001
0_0010
1_0100
```

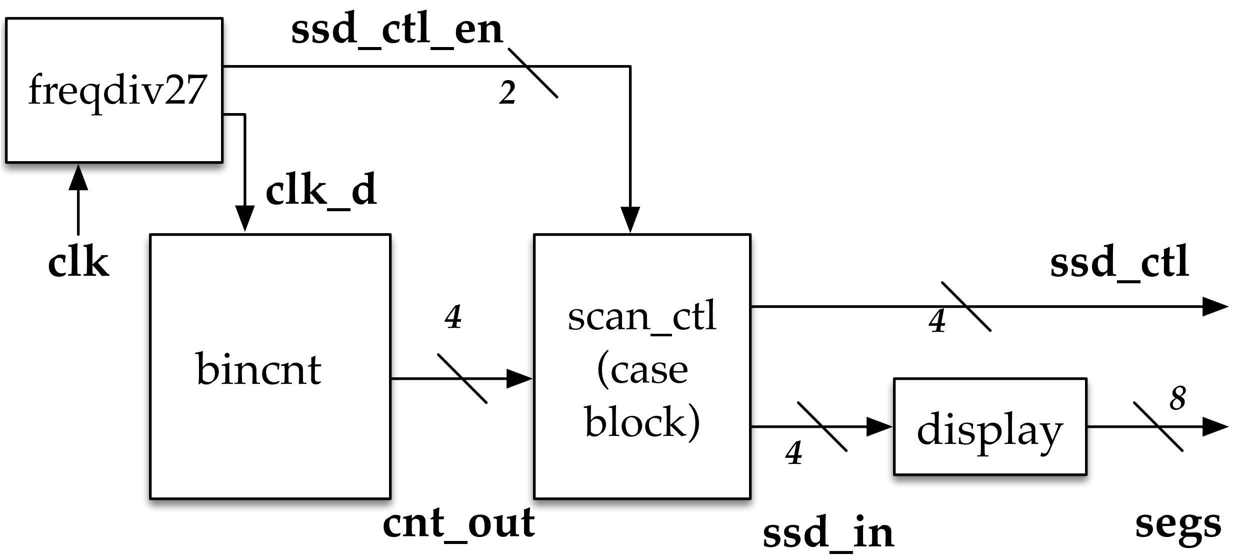
Testbenches (3/3)

```
always #5 clk=~clk;

initial
begin
  clk=0; rst_n=0; in=0;
end

initial
begin
  #10 rst_n=1;
  for (vectornum=0; vectornum<10; vectornum=vectornum+1)
  begin
    #10 {in,q_expected} = testvectors[vectornum]; //apply input; wait
    if (q != q_expected) $display(“%b: %b failed”, in, q);// check
  end
end
endmodule
```

Modularized Binary Counter

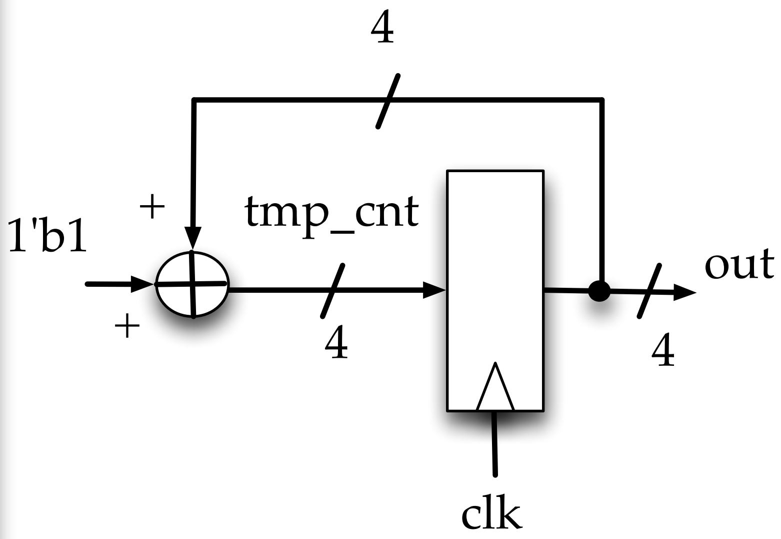
Binary Up Counter





Binary Up Counter (bincnt.v)

```
`include “global.v”
module bincnt(
  out,  // counter output
  clk,  // global clock
  rst_n  // active low reset
);

output [`CNT_BIT_WIDTH-1:0] out;  // counter output
input clk;  // global clock
input rst_n;  // active low reset

reg [`CNT_BIT_WIDTH-1:0] out; // counter output (in always block)
reg [`CNT_BIT_WIDTH-1:0] tmp_cnt; // input to dff (in always block)

// Combinational logics
always @*
  tmp_cnt = out + 1’b1;

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n)
     out<=0;
  else
     out<=tmp_cnt;

endmodule
```



Scan Control

1

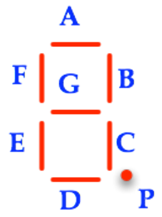
```
`include “global.v”
module scan_ctl(
  ssd_ctl, // ssd display control signal
  ssd_in, // output to ssd display
  in0, // 1st input
  in1, // 2nd input
  in2, // 3rd input
  in3,  // 4th input
  ssd_ctl_en // divided clock for scan control
);

output [`BCD_BIT_WIDTH-1:0] ssd_in; // Binary data
output [`SSD_NUM-1:0] ssd_ctl; // scan control for 7-segment display
input [`BCD_BIT_WIDTH-1:0] in0,in1,in2,in3; // binary input control for the four digits
input [`SSD_SCAN_CTL_BIT_WIDTH-1:0] ssd_ctl_en; // divided clock for scan control

reg [`SSD_NUM-1:0] ssd_ctl; // scan control for 7-segment display (in the always block)
reg [`BCD_BIT_WIDTH-1:0] ssd_in; // 7 segment display control (in the always block)
```

2

```
always @*
  case (ssd_ctl_en)
    2'b00:
    begin
      ssd_ctl=4'b0111;
      ssd_in=in0;
    end
    2'b01:
    begin
      ssd_ctl=4'b1011;
      ssd_in=in1;
    end
    2'b10:
    begin
      ssd_ctl=4'b1101;
      ssd_in=in2;
    end
    2'b11:
    begin
      ssd_ctl=4'b1110;
      ssd_in=in3;
    end
    default:
    begin
      ssd_ctl=4'b0000;
      ssd_in=in0;
    end
  endcase

endmodule
```

A Binary to Seven-Segment Display Decoder

```
`include “global.v”
module display(
  segs, // 7-segment display
  bin // binary input
);

output reg [`SSD_BIT_WIDTH-1:0] segs; // 7-segment display out
input [`BCD_BIT_WIDTH-1:0] bin; // binary input

// Combinatioanl Logic
always @*
  case (bin)
    `BCD_BIT_WIDTH'd0: segs = `SSD_ZERO;
    `BCD_BIT_WIDTH'd1: segs = `SSD_ONE;
    `BCD_BIT_WIDTH'd2: segs = `SSD_TWO;
    `BCD_BIT_WIDTH'd3: segs = `SSD_THREE;
    `BCD_BIT_WIDTH'd4: segs = `SSD_FOUR;
    `BCD_BIT_WIDTH'd5: segs = `SSD_FIVE;
    `BCD_BIT_WIDTH'd6: segs = `SSD_SIX;
    `BCD_BIT_WIDTH'd7: segs = `SSD_SEVEN;
    `BCD_BIT_WIDTH'd8: segs = `SSD_EIGHT;
    `BCD_BIT_WIDTH'd9: segs = `SSD_NINE;
    `BCD_BIT_WIDTH'd10: segs = `SSD_A;
    `BCD_BIT_WIDTH'd11: segs = `SSD_B;
    `BCD_BIT_WIDTH'd12: segs = `SSD_C;
    `BCD_BIT_WIDTH'd13: segs = `SSD_D;
    `BCD_BIT_WIDTH'd14: segs = `SSD_E;
    `BCD_BIT_WIDTH'd15: segs = `SSD_F;
     default: segs = `SSD_DEF;
  endcase

endmodule
```



Top Module

```
`include “global.v”
module binary_counter(
  segs, // 7-segment display
  ssd_ctl, // scan control for 7-segment display
  clk, // clock from oscillator
  rst_n // active low reset
);

output [`SSD_BIT_WIDTH-1:0] segs; // 7-segment display
output [`SSD_NUM-1:0] ssd_ctl; // scan control for 7-segment display
input clk; // clock from oscillator
input rst_n; // active low reset

wire clk_d; // frequency-divided clock
wire [`CNT_BIT_WIDTH-1:0] cnt_out; // binary counter output
wire [`SSD_SCAN_CTL_BIT_WIDTH-1:0] ssd_ctl_en;
wire [`CNT_BIT_WIDTH-1:0] ssd_in;
```

```
// Frequency Divider
freqdiv27 U_FD0(
  .clk_out(clk_d), //divided clock output
  .clk_ctl(ssd_ctl_en), // divided scan clock for 7-segment display scan
  .clk(clk), // clock from the 40MHz oscillator
  .rst_n(rst_n) // low active reset
);

// Binary Counter
bincnt U_BC(
  .out(cnt_out), //counter output
  .clk(clk_d), // clock
  .rst_n(rst_n) //active low reset
);
```

```
// Scan control
scan_ctl U_SC(
  .ssd_ctl(ssd_ctl), // ssd display control signal
  .ssd_in(ssd_in), // output to ssd display
  .in0(cnt_out), // 1st input
  .in1(4'b1111), // 2nd input
  .in2(4'b1111), // 3rd input
  .in3(4'b1111),  // 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
```

global.v

```
// Frequency divider
`define FREQ_DIV_BIT 27
`define SSD_SCAN_CTL_BIT_WIDTH 2 // scan control bit with for 7-segment display

// Counter
`define CNT_BIT_WIDTH 4 //number of bits for the counter

// 14-segment display
`define SSD_BIT_WIDTH 8 // 7-segment display control
`define SSD_NUM 4 //number of 7-segment display
`define BCD_BIT_WIDTH 4 // BCD bit width
`define SSD_ZERO   `SSD_BIT_WIDTH'b0000_0011 // 0
`define SSD_ONE    `SSD_BIT_WIDTH'b1001_1111 // 1
`define SSD_TWO    `SSD_BIT_WIDTH'b0010_0101 // 2
`define SSD_THREE  `SSD_BIT_WIDTH'b0000_1101 // 3
`define SSD_FOUR   `SSD_BIT_WIDTH'b1001_1001 // 4
`define SSD_FIVE   `SSD_BIT_WIDTH'b0100_1001 // 5
`define SSD_SIX    `SSD_BIT_WIDTH'b0100_0001 // 6
`define SSD_SEVEN  `SSD_BIT_WIDTH'b0001_1111 // 7
`define SSD_EIGHT  `SSD_BIT_WIDTH'b0000_0001 // 8
`define SSD_NINE   `SSD_BIT_WIDTH'b0000_1001 // 9
`define SSD_A   `SSD_BIT_WIDTH'b0000_0101 // a
`define SSD_B   `SSD_BIT_WIDTH'b1100_0001 // b
`define SSD_C   `SSD_BIT_WIDTH'b1110_0101 // c
`define SSD_D   `SSD_BIT_WIDTH'b1000_0101 // d
`define SSD_E   `SSD_BIT_WIDTH'b0110_0001 // e
`define SSD_F   `SSD_BIT_WIDTH'b0111_0001 // f
`define SSD_DEF    `SSD_BIT_WIDTH'b0000_0000 // default, all LEDs being lighted
```

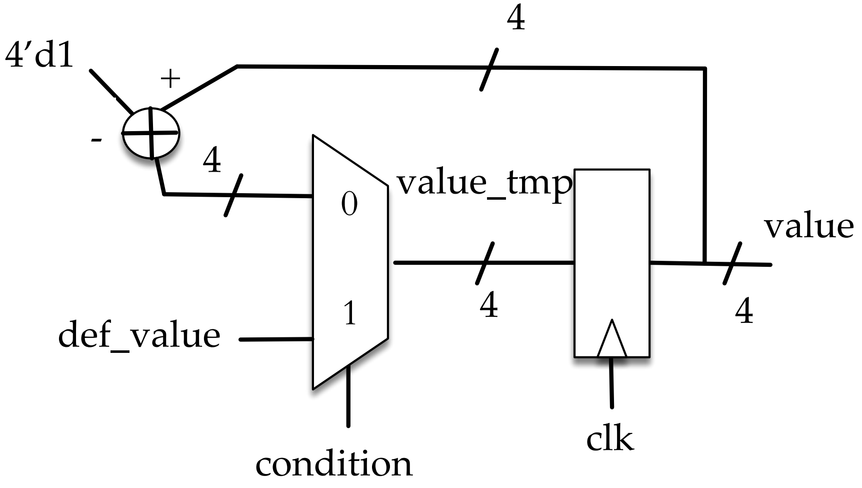
Modularized BCD Counter

Load Default Value for DFFs

- at reset of DFFs

```
always @(posedge clk or negedge rst_n)
  if (~rst_n)
    q <= 0;
  else
    q <= d;
```

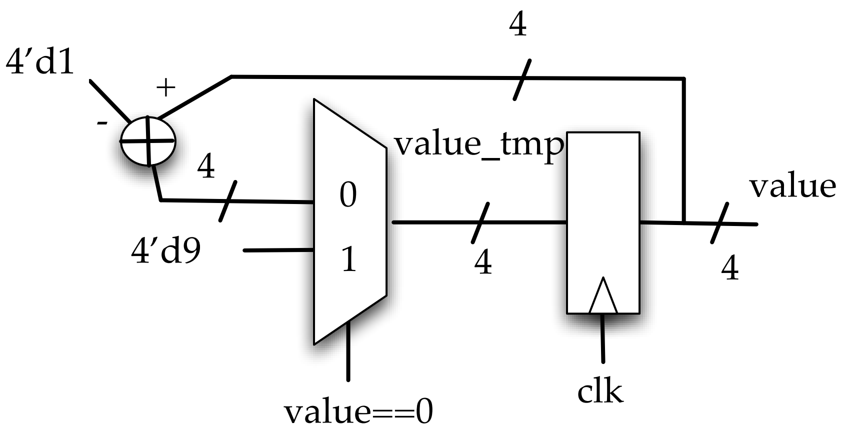
- Use MUX for conditional load to the DFFs

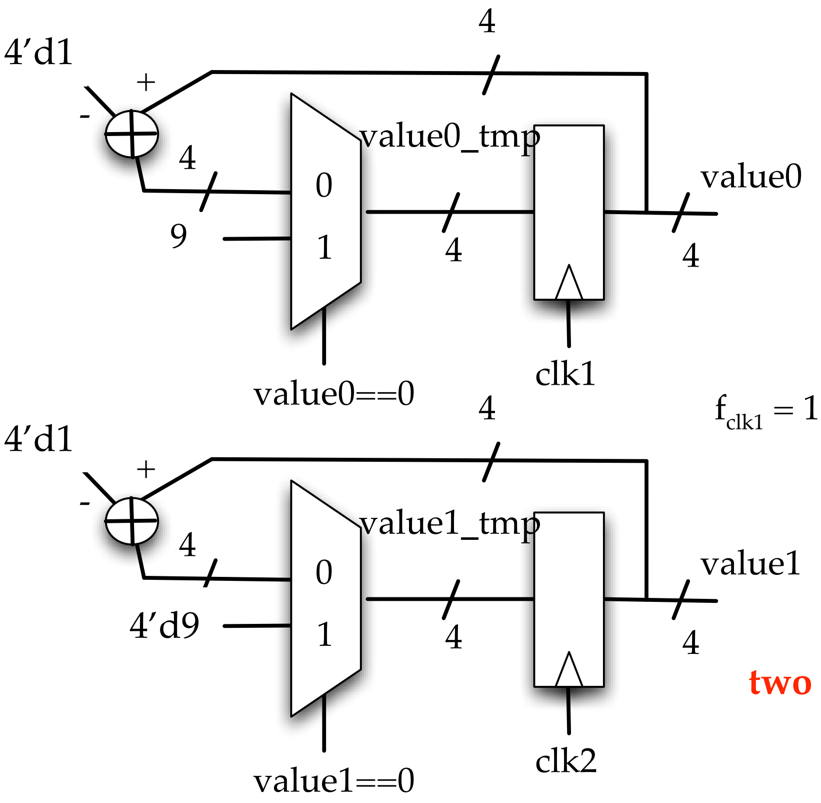


- **Do not** use *initial*

BCD Down-Counter

single digit





$f_{clk1} = 10 \times f_{clk2}$

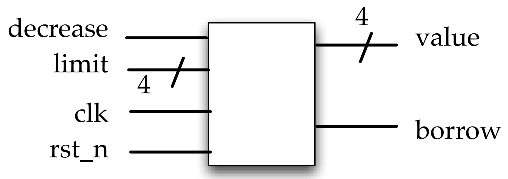
two digits

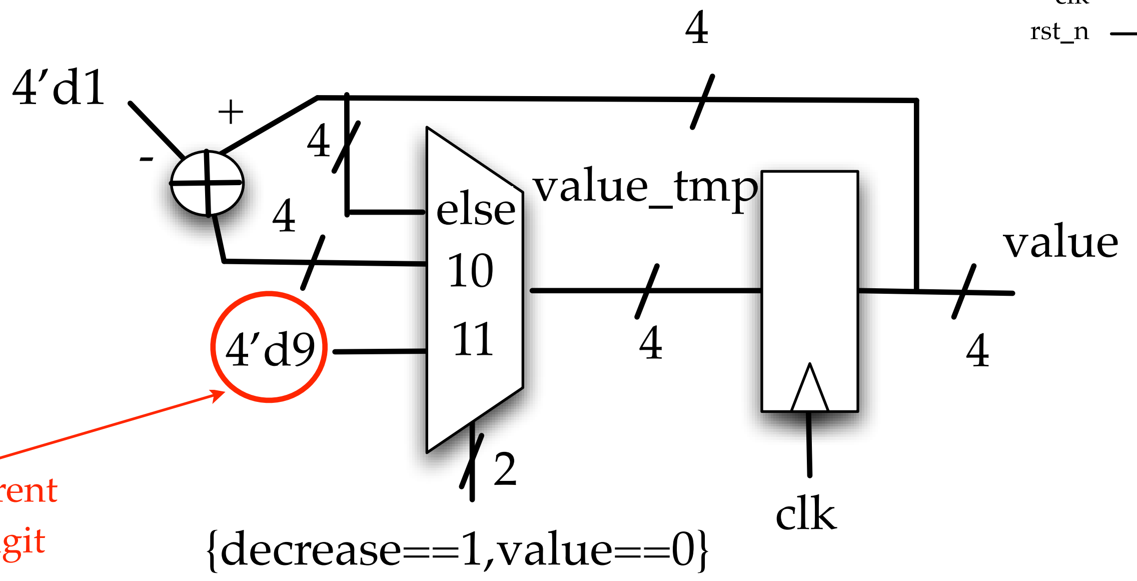
```
// Combinational logics
always @*
  if (value==`BCD_ZERO)
    value_tmp = `BCD_NINE;
  else
    value_tmp = value - `INCREMENT;

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

clk1 and clk2 needed to be synchronized in frequency and phase!!

BCD Down-Counter

How to use one single clock for different digit counting?

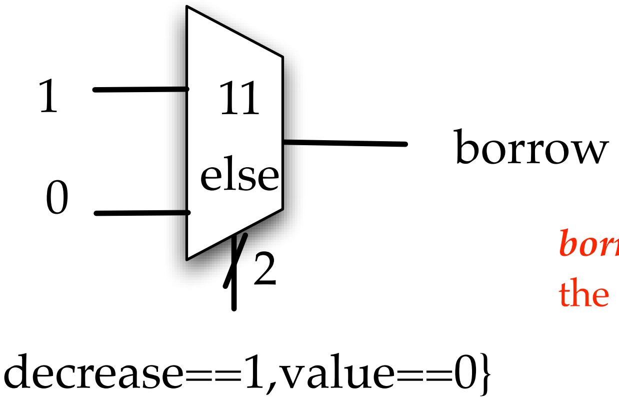




Use *limit* as different ceiling for each digit

`ENABLED

`DISABLED



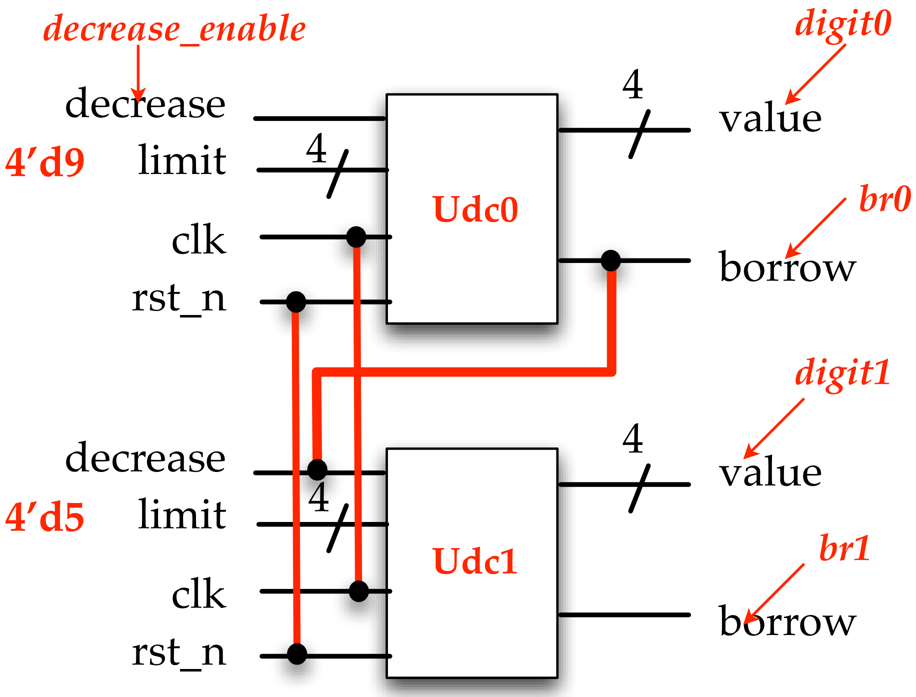
borrow signal in lower digit is the *decrease* control in the upper digit

BCD Down-Counter

```
module downcounter(
  value,  // counter output
  borrow,  // borrow indicator
  clk, // global clock
  rst_n, // active low reset
  decrease, // counter enable control
  limit // limit for the counter
);
... ...
// Combinational logics
always @*
  if (value==`BCD_ZERO && decrease)
    begin
      value_tmp = limit;
      borrow = `ENABLED;
    end
  else if (value!=`BCD_ZERO && decrease)
    begin
      value_tmp = value - `INCREMENT;
      borrow = `DISABLED;
    end
  else
    begin
      value_tmp = value;
      borrow = `DISABLED;
    end
```

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

endmodule
```

2-digit BCD Down-Counter



```
// 30 sec down counter
downcounter Udc0(
  .value(digit0),  // counter value
  .borrow(br0),  // borrow indicator
  .clk(clk), // global clock signal
  .rst_n(rst_n),  // low active reset
  .decrease(decrease_enable),  // counter enable control
  .limit(`BCD_NINE)  // limit for the counter
);

downcounter Udc1(
  .value(digit1),  // counter value
  .borrow(br1),  // borrow indicator
  .clk(clk), // global clock signal
  .rst_n(rst_n),  // low active reset
  .decrease(br0),  // counter enable control
   .limit(`BCD_FIVE)  // limit for the counter
);
```

counting from 59 to 0