Verilog HDL (2)

Hsi-Pin Ma

http://lms.nthu.edu.tw/course/24953

Department of Electrical Engineering

National Tsing Hua University

Lexical Conventions

White Space and Comments

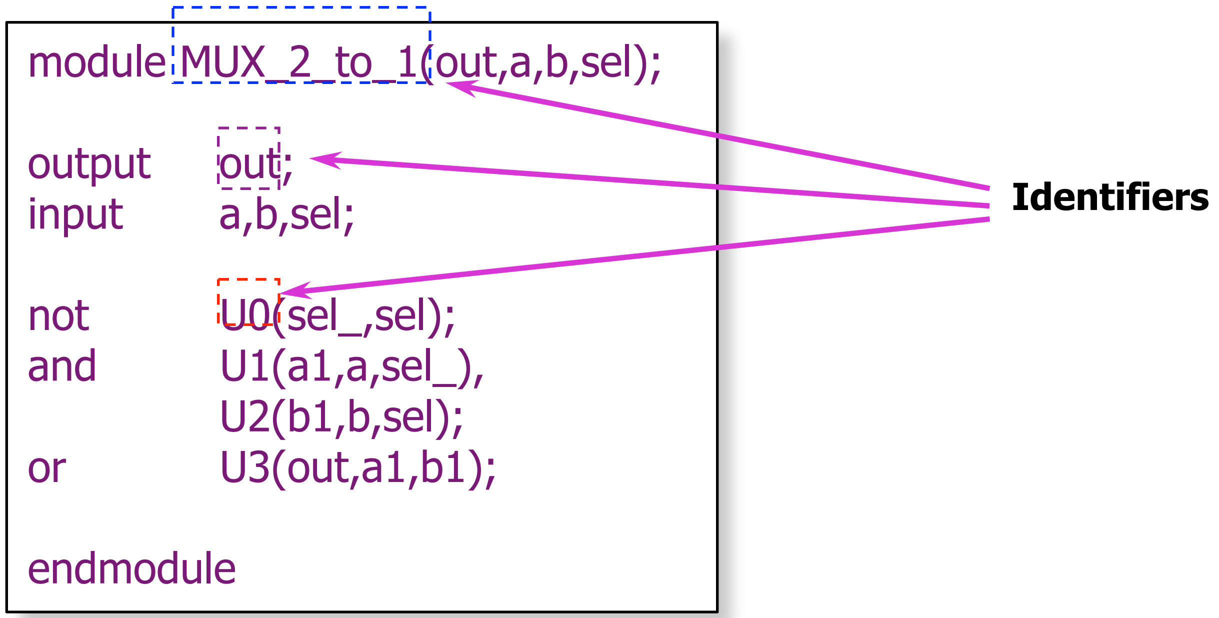
- White space makes code more readable
  - Include blank space (\b), tabs (\t), and carriage return (\n).
- Comments
  - /* … */ : mark more than one line
  - // : mark only one line.

Identifiers

- Identifiers are user-provided names for Verilog objects within a description.
- Legal characters in identifiers:
  - a-z, A-Z, 0-9, _, $
- The first character of an identifier must be an alphabetical character (a-z, A-Z) or an underscore (_).
- Identifiers can be up to 1023 characters long.

Identifiers

- Names of modules, ports, and instances are identifiers.



```
module MUX_2_to_1(out,a,b,sel);

output     out;
input      a,b,sel;

not        U0(sel_,sel);
and        U1(a1,a,sel_),
           U2(b1,b,sel);
or         U3(out,a1,b1);

endmodule
```

Keywords

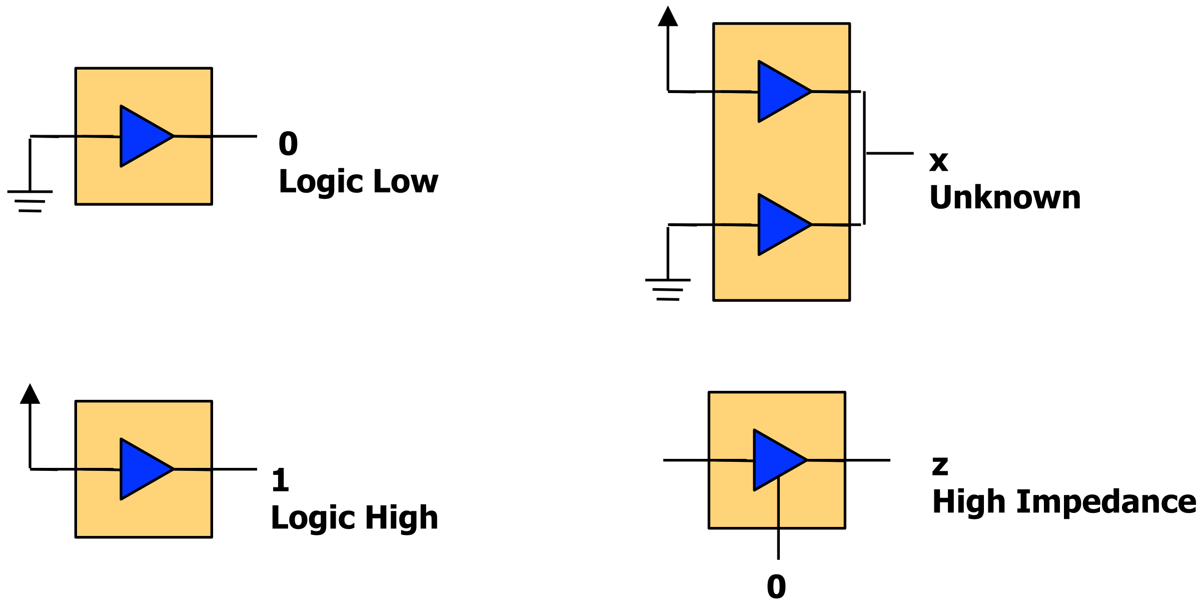
- Pre-defined non-escaped identifiers that used to define the language construct.
- All keywords are defined in lower cases.
- Examples
  - module, endmodule
  - input, output, inout
  - reg, integer, real, time
  - not, and, or, nand, nor, xor
  - parameter
  - begin, end
  - fork, join
  - always, for
  - …

Case Sensitivity

- Verilog is a case sensitive language.
- Use "-u" option in command line option for case-insensitive.

Value Sets

- 4-value logic system in Verilog

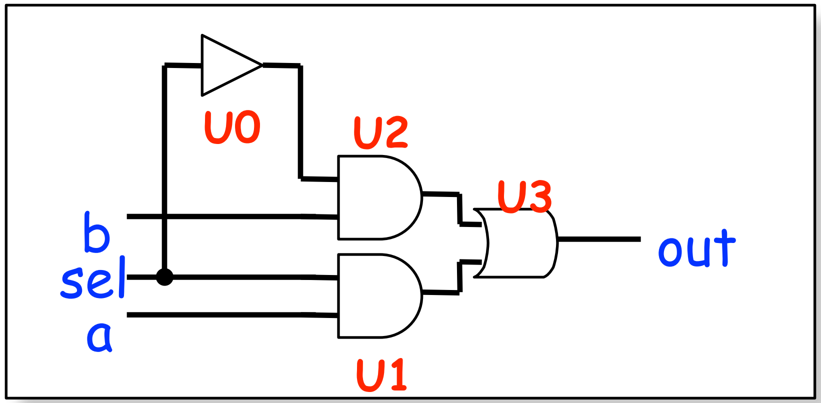


Verilog HDL Synthesis

Some Interpretations (1/3)

- **?:** : Conditional operators (multiplexer)

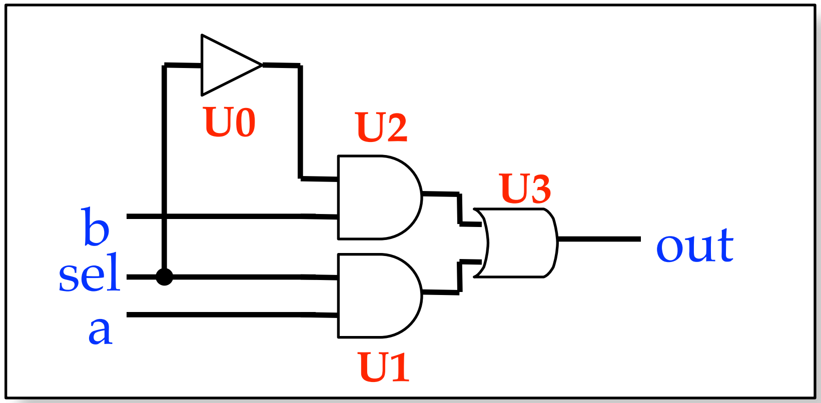
```
module SMUX(out,a,b,sel);
output out;
input a,b,sel;

assign out = (sel) ? a : b ;

endmodule
```



Some Interpretations (2/3)

- **if-else** statements
  - Not for large multiplexers

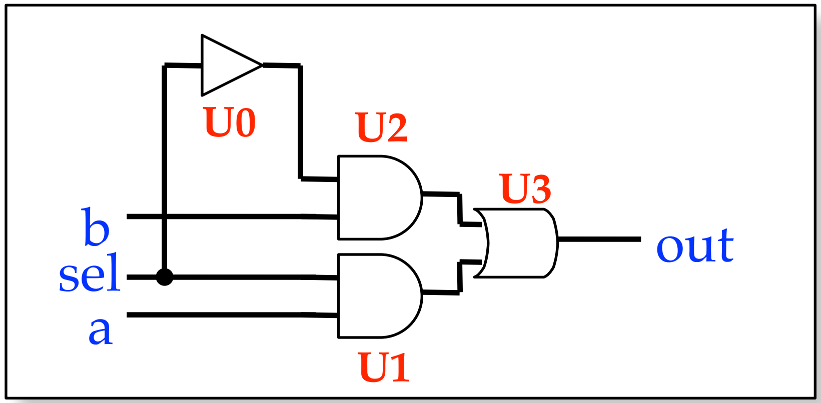
```
module SMUX(out,a,b,sel)
output out;
input a,b,sel;

always @*
  if (sel)
    out = a;
  else
    out = b ;

endmoudle
```



Some Interpretations (3/3)

- **case** statements
  - Usually for large multiplexers

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

always @*
  case (sel)
    1'b0: out = b;
    1'b1: out = a;
    default: out=b;
  endcase

endmodule
```



Some Notes about Xilinx ISE

- Use one project for one exp.
- Use the same name for project, top module
- Do not use number for the first character of the module/project name

Integer and Real Numbers

- Numbers can be integer or real numbers.
- Integer can be sized or unsized. Sized integer can be represented as
  - <size>’<base><value>
    - size : size in bits
    - base : can be b(binary), o(octal), d(decimal), or h(hexadecimal)
    - value : any legal number in the selected base and x, z, ?.
- Real numbers can be represented in decimal or scientific format.

Integer and Real Numbers

- 16 : 32 bits decimal
- 8’d16
- 8’h10
- 8’b0001_0000
- 8’o20
- 32’bx : 32 bits x
- 2’b1? : ? represents a high impedance bit
- 6.3
- 5.3e-4
- 6.2e3

Concatenation and Replication Operators

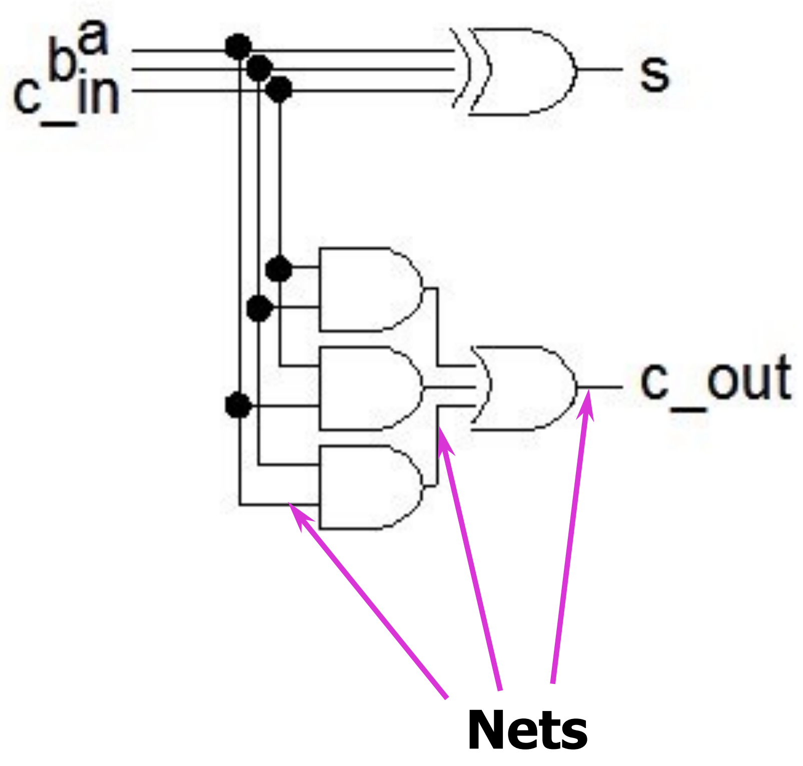
- Bit replication for 01010101
  - assign byte = {4{2'b01}};
- Sign extension
  - assign word = {{8{byte[7]}},byte};

Major Data Type Class

- Nets
  - Physical connection between devices
- Registers
  - Represent abstract storage elements
- Parameters
  - Configure module instances

Nets

- Physical connections between structural entities.
- Must be driven by a driver, such as a **gate instantiation** or **continuous assignment**
- As the driver changes its value, Verilog automatic propagates the value onto a net.
- Default value is **z** if no drivers are connected to net



Registers

- Registers represent **abstract** storage elements.
- A register holds its value until a new value is assigned to it.
- Registers are used extensively in behavior modeling and in applying stimuli.
- Default value is X.

Type of Registers

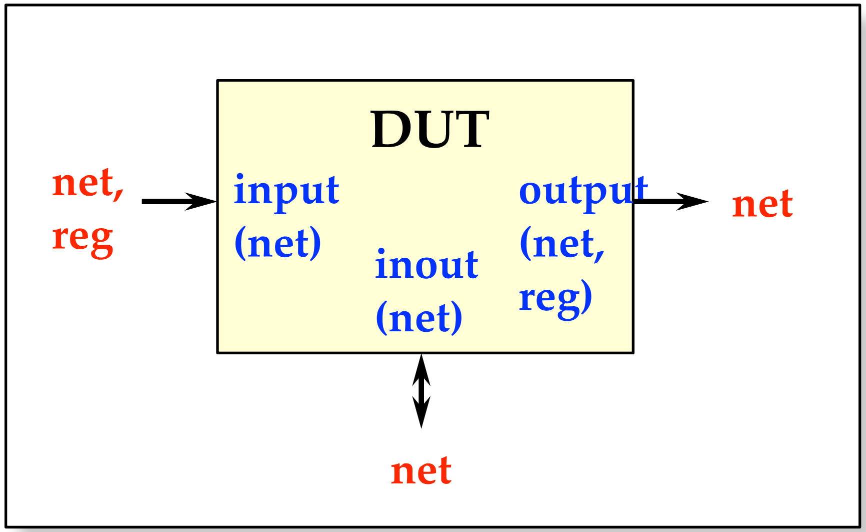
- reg
  - Unsigned integer variable of varying bit width
- integer
  - Signed integer variable, 32-bit wide. Arithmetic operations produce 2's complement results.
- real
  - Signed floating point variable, double precision
- time
  - Unsigned integer variable, 64-bit wide.
- Do not confuse register data type with structural storage element (e.g. D-type FF)

Declaration Syntax of Verilog Registers

- reg <range> ? <name> <,<name>>*;
- Example
  - reg a;
  - reg [5:2] b,c;

Choosing the Correct Data Types

- An **input** or **inout** port must be a net.
- An **output** port can be a register data type.
- A signal assigned a value in a procedural block must be a register data type.



Common Mistakes in Choosing Data Types

- Make a procedural assignment to a net

  wire [7:0] databus;

  always @(read or addr) databus=read ? mem[addr] : 'bz;

  Illegal left-hand-side assignment

- Connect a register to an instance output

  reg myreg;

  and (myreg, net1, net2);

  Illegal output port specification

- Declare a module **input** port as a register

  input myinput;

  reg myinput;

  Incompatible declaration

Procedural Assignments

```
module assignment_test;
reg [3:0] a,b;
wire [4:0] sum1;
reg [4:0] sum2;

assign sum1 = a + b ;

initial
begin
      a=4'b1010;b=4'b0110;
      sum2 = a + b;
      $display("a b sum1 sum2);
      $monitor(a,b,sum1,sum2);
      #10   a=4'b0001;
end

endmodule
```

Continuous assignment

Procedural assignment

```
module FA(s,co,a,b,ci);
input a,b,ci;
output s,co;
reg s;

s=a^b^ci;

always @*
begin
      assign co=(a&b)|(b&ci)|
      (a&ci);
end

endmodule
```

Error! Illegal left-hand-side continuous assignment.

Error! Illegal left-hand-side in assign statement.

Examples

Design Procedure

1 • From the ***specifications***, determine the inputs, outputs, and their symbols.

2 • Derive the ***truth table*** (***functions***) from the relationship between the inputs and outputs

3 • Derive the ***simplified Boolean functions*** for each output function.

4 • Draw the logic diagram.

5 • Construct the Verilog code according to the logic diagram.

6 • Write the testbench and verify the design.

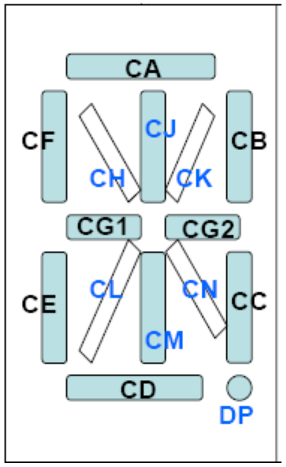
Seven-Segment Display Decoder (1/2)

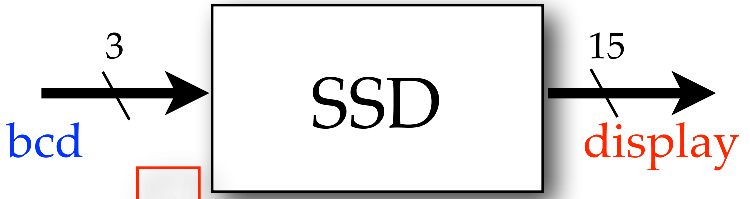
1 input: bcd[3:0] output: display[14:0]

0000	-> 0000_0011_1111_111
0001	-> 1111_1111_1011_011
0010	-> 0010_0100_1111_111
0011	-> 0000_1100_1111_111
0100	-> 1001_1000_1111_111
0101	-> 0100_1000_1111_111
0110	-> 0100_0000_1111_111
0111	-> 0001_1111_1111_111
1000	-> 0000_0000_1111_111
1001	-> 0000_1000_1111_111
others	-> 1111_1111_1111_111

2

3





4

Laboratory for Reliable Computing

Seven-Segment Display Decoder (2/2)

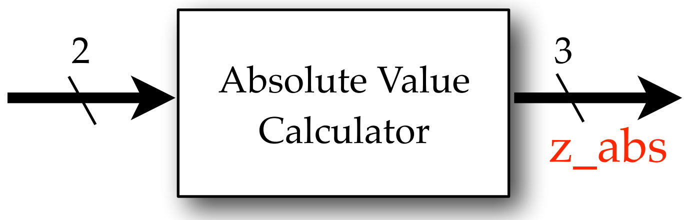
5

```
module bcd2ftsegdec(
 display,  // 14-segment display output
 bcd  // BCD input
);

output [14:0] display;  // SSD display output
input [3:0] bcd;  // BCD input

reg [14:0] display;  // SSD display output (in always)

// Combinational logics:
always @*
 case (bcd)
  4’d0: display = 15‘b0000_0011_1111_111; //0
  4’d1: display = 15‘b1111_1111_1011_011; //1
  4’d2: display = 15‘b0010_0100_1111_111; //2
  4’d3: display = 15‘b0000_1100_1111_111; //3
  4’d4: display = 15‘b1001_1000_1111_111; //4
  4’d5: display = 15‘b0100_1000_1111_111; //5
  4’d6: display = 15‘b0100_0000_1111_111; //6
  4’d7: display = 15‘b0001_1111_1111_111; //7
  4’d8: display = 15‘b0000_0000_1111_111; //8
  4’d9: display = 15‘b0000_1000_1111_111; //9
  default: display = 15‘b1111_1111_1111_111; //DEF
 endcase

endmodule
```

Laboratory for Reliable Computing

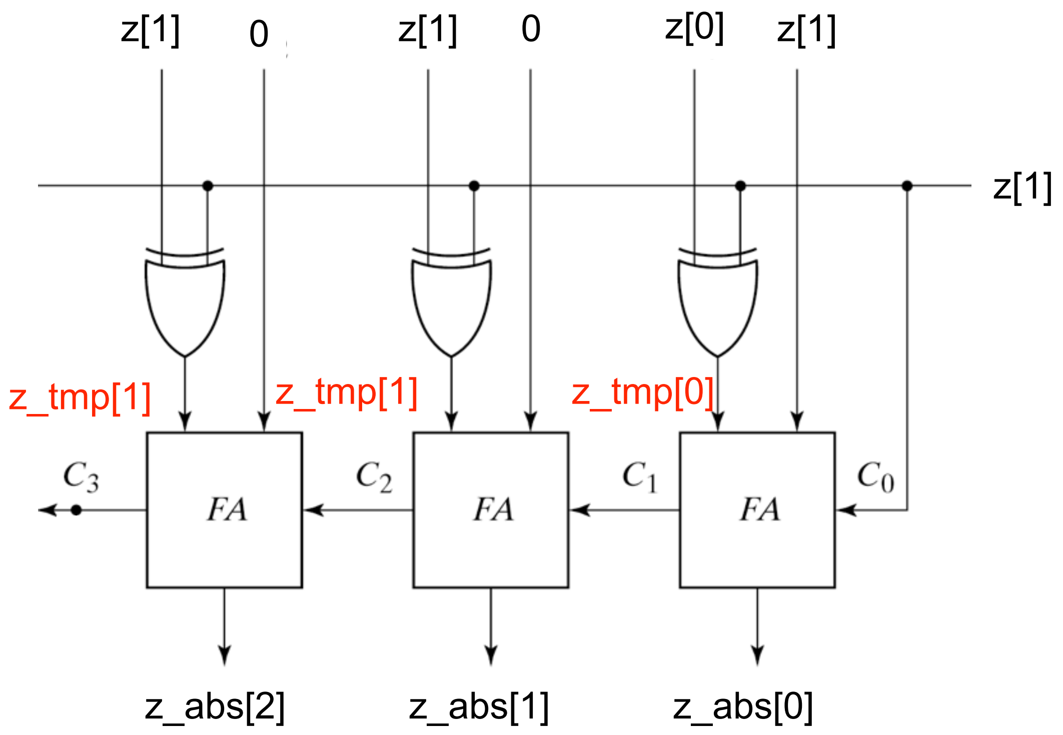
2-bit Absolute Value Calculator (1/2)

1 input: z[1:0]   output: z_abs[2:0]



2 If z is negative (MSB is 1), complement every bit and add 1.
If z is positive (MSB is 0), keep all bits the same.
Use XOR for MSB and every bit.

3



2-bit Absolute Value Calculator (2/2)

Module (abs.v)

5

```
module abs(
  z_abs, // absolute value of z
  z  //  original value
);

output [2:0] z_abs;  // absolute value of z
input [1:0] z;  // original value

reg [1:0] z_tmp;  // XOR output
reg [2:0] z_abs;  // register for Z

// Combinational logics:
always @*
begin
  z_tmp[1]=z[1]^z[1];
  z_tmp[0]=z[0]^z[1];
  z_abs={z_tmp[1],z_tmp}+{2'b0,z[1]};
end

endmodule
```

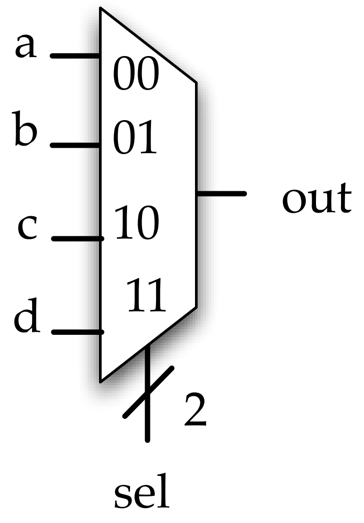
Testbench (t_abs.v)

6

```
module t_abs;

wire [2:0] z_abs;  // absolute value of z
reg [1:0] z;  // original value

abs U0(.z_abs(z_abs),.z(z));

initial
begin
  z=2'b00;
  #5 z=2'b01;
  #5 z=2'b10;
  #5 z=2'b11;
  #5 z=2'b00;
end

endmodule
```

MUX 1

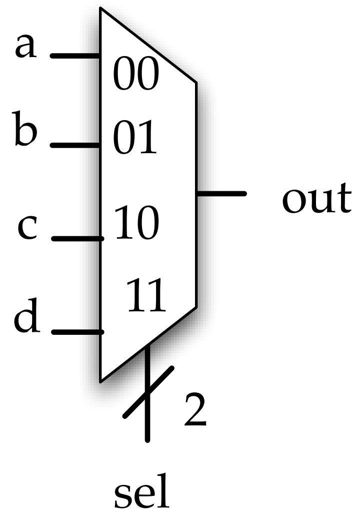
```
module mux(
  out,  // output
  a,  // input a
  b,  // input b
  c,   // input c
  d,  // input d
  sel // selection control signal
);

output out;  // output
input a;  // input a
input b;  // input b
input c;  // input c
input d;  // input d
input [1:0] sel;  // selection control signal
reg out; // output (in always block)

always @*
  if (sel==2'b00)  out = a;
  else if (sel==2'b01) out = b;
  else if (sel==2'b10) out=c;
  else out=d;

endmodule
```



should have final "*else*"

MUX 2

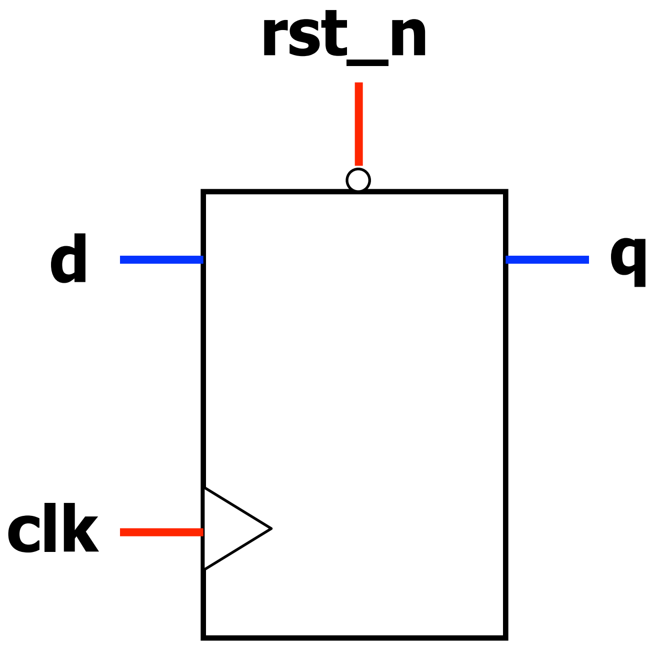
```
module mux(
 out,  // output
 a,  // input a
 b,  // input b
 c,   // input c
 d,  // input d
 sel // selection control signal
);

output out;  // output
input a;  // input a
input b;  // input b
input c;  // input c
input d;  // input d
input [1:0] sel;  // selection control signal
reg out; // output (in always block)

always @*
 case (sel)
  2'b00: out = a;
  2'b01: out = b;
  2'b10: out = c;
  2'b11: out = d;
  default: out = 0;
endcase
endmodule
```



should have final "*default*"

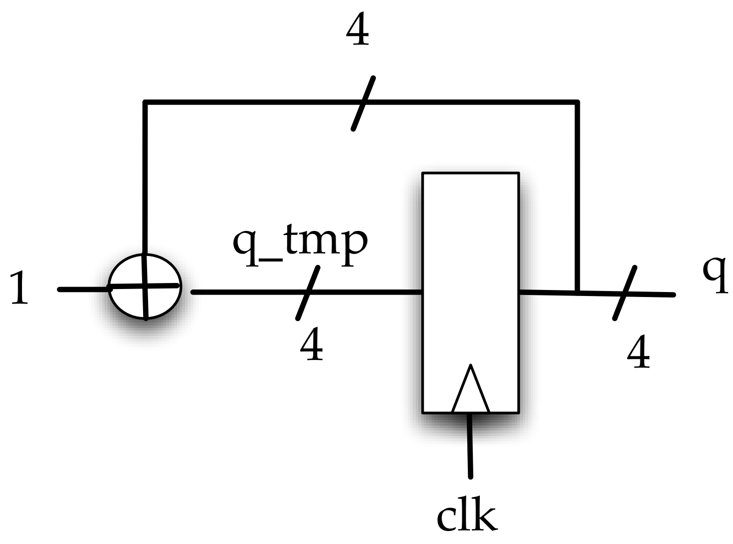
D-type Flip Flop

```
module dff(
  q,  // output
  d,  // input
  clk,  // global clock
  rst_n // active low reset
);

output q;  // output
input d;  // input
input clk;  // global clock
input rst_n;  // active low reset

reg q; // output (in always block)

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

endmodule
```



Binary Up Counter

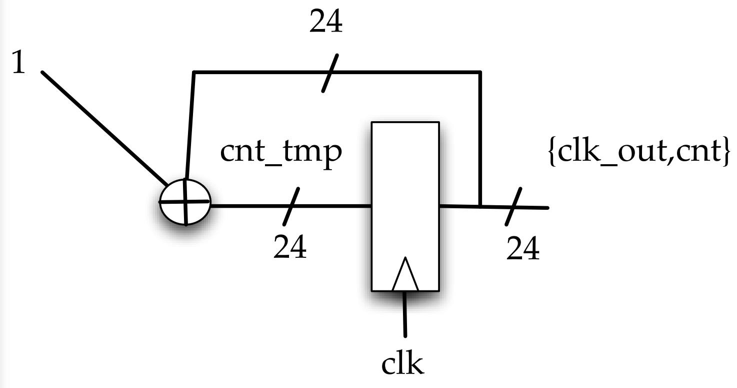
```
`define BCD_BIT_WIDTH 4
`define BCD_ZERO 4'd0
`define BCD_ONE 4'd1
`define BCD_NINE 4'd9
module bcdcounter(
  q,  // output
  clk,  // global clock
  rst_n  // active low reset
);

output [`BCD_BIT_WIDTH-1:0] q;  // output
input clk;  // global clock
input rst_n;  // active low reset

reg [`BCD_BIT_WIDTH-1:0] q; // output (in always block)
reg [`BCD_BIT_WIDTH-1:0] q_tmp; // input to dff (in always block)

// Combinational logics
always @*
  q_tmp = q + `BCD_ONE;

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n) q<=`BCD_BIT_WIDTH'd0;
  else q<=q_tmp;

endmodule
```



NTHU EE Laboratory for Reliable Computing

Frequency Divider

```
`define FREQ_DIV_BIT 24
module freq_div(
  clk_out,  // divided clock output
  clk,  // global clock input
  rst_n  // active low reset
);

output clk_out;  // divided output
input clk;  // global clock input
input rst_n;  // active low reset

reg clk_out; // clk output (in always block)
reg [`FREQ_DIV_BIT-2:0] cnt; // remainder of the counter
reg [`FREQ_DIV_BIT-1:0] cnt_tmp; // input to dff (in always block)

// Combinational logics: increment, neglecting overflow
always @*
  cnt_tmp = {clk_out,cnt} + 1'b1;

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n) {clk_out, cnt}<=`FREQ_DIV_BIT'd0;
  else {clk_out,cnt}<=cnt_tmp;

endmodule
```

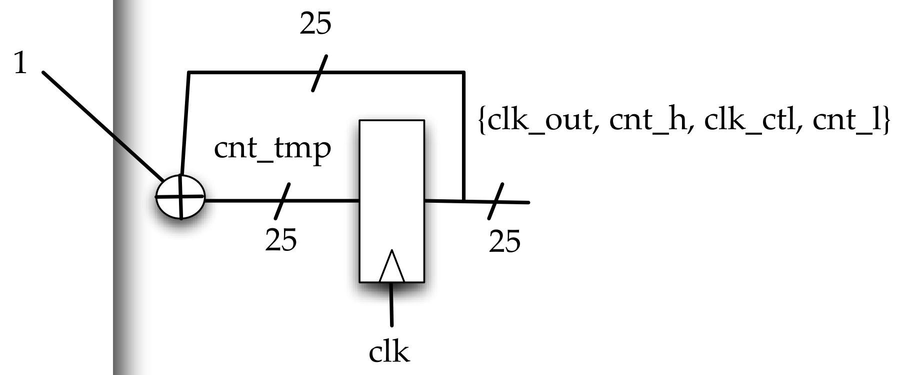


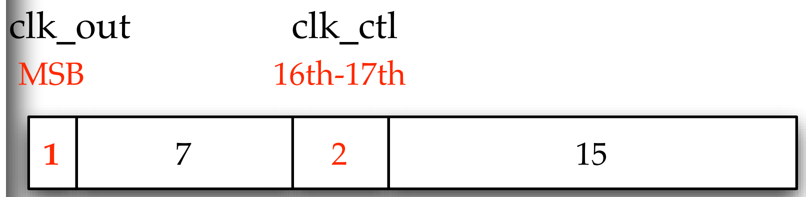
cnt_tmp[23:0]

cnt[22:0]

Frequency Divider

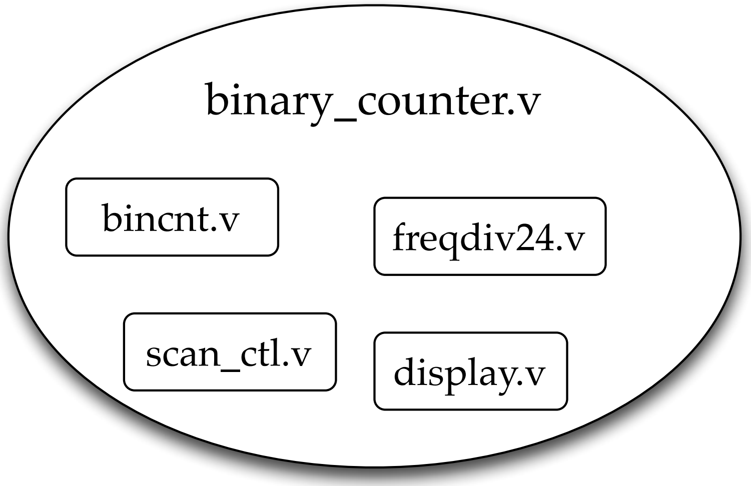
```
`define FREQ_DIV_BIT 25
module freq_div(
  clk_out,  // divided clock output
  clk_ctl,  // divided clock output for scan freq
  clk,  // global clock input
  rst_n  // active low reset
);

output clk_out;  // divided output
output [1:0] clk_ctl;  // divided output for scan freq
input clk;  // global clock input
input rst_n;  // active low reset

reg clk_out; // clk output (in always block)
reg [1:0] clk_ctl; // clk output (in always block)
reg [14:0] cnt_l; // temp buf of the counter
reg [6:0] cnt_h; // temp buf of the counter
reg [`FREQ_DIV_BIT-1:0] cnt_tmp; // input to dff (in always block)

// Combinational logics: increment, neglecting overflow
always @*
  cnt_tmp = {clk_out,cnt_h,clk_ctl,cnt_l} + 1’b1;

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n) {clk_out, cnt_h, clk_ctl, cnt_l}<=`FREQ_DIV_BIT’d0;
  else {clk_out,cnt_h, clk_ctl, cnt_l}<=cnt_tmp;

endmodule
```

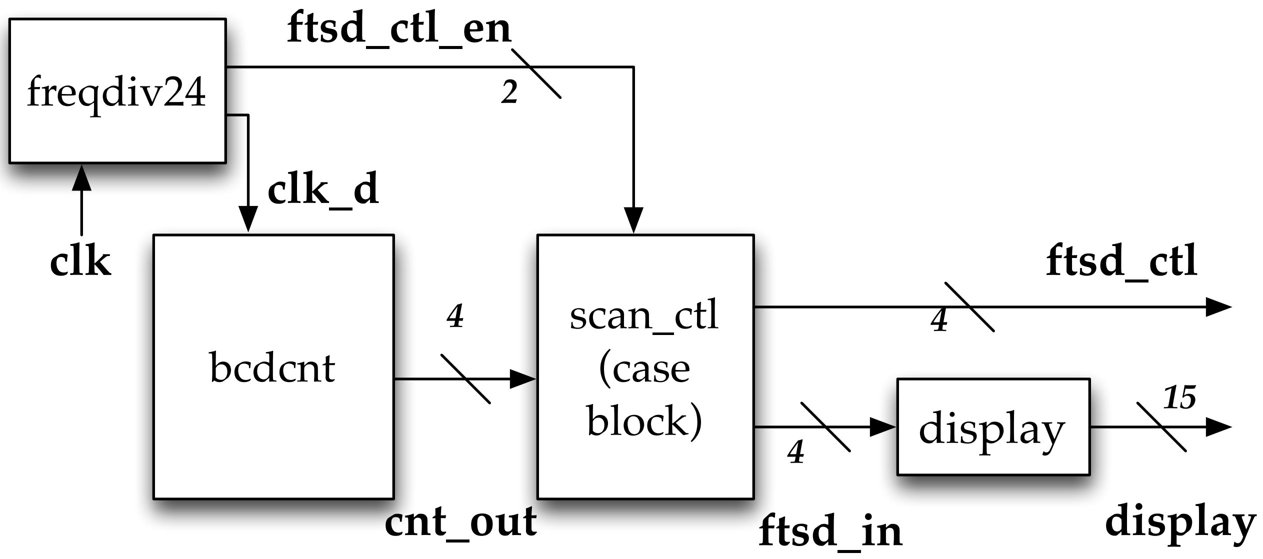




Modularized Binary Counter

Binary Up Counter



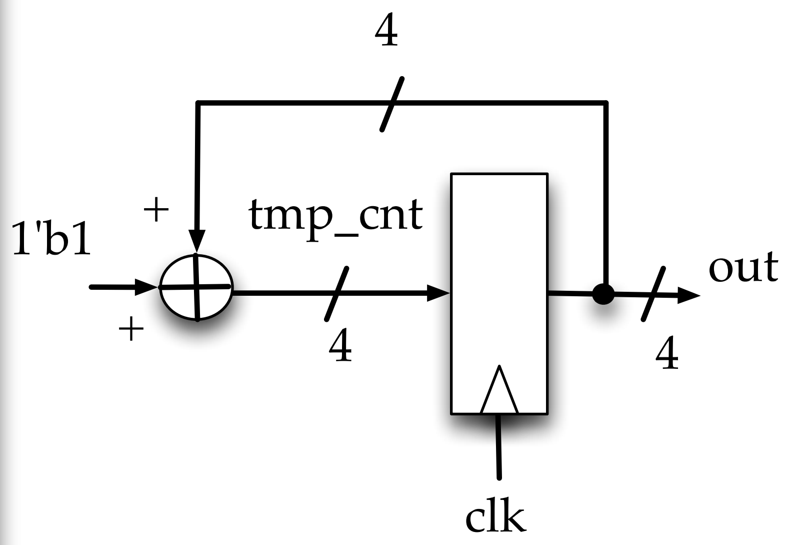


NTHU EE

Laboratory for Reliable Computing

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(
  ftsd_ctl, // ftsd display control signal
  ftsd_in, // output to ftsd display
  in0, // 1st input
  in1, // 2nd input
  in2, // 3rd input
  in3,  // 4th input
  ftsd_ctl_en // divided clock for scan control
);

output [`BCD_BIT_WIDTH-1:0] ftsd_in; // Binary data
output [`FTSD_NUM-1:0] ftsd_ctl; // scan control for 14-segment display
input [`BCD_BIT_WIDTH-1:0] in0,in1,in2,in3; // binary input control for the four digits
input [`FTSD_SCAN_CTL_BIT_WIDTH-1:0] ftsd_ctl_en; // divided clock for scan control

reg [`FTSD_NUM-1:0] ftsd_ctl; // scan control for 14-segment display (in the always block)
reg [`BCD_BIT_WIDTH-1:0] ftsd_in; // 14 segment display control (in the always block)
```

2

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

endmodule
```