NTHU EE

Laboratory for Reliable Computing

Verilog 2

Hsi-Pin Ma

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

Department of Electrical Engineering

National Tsing Hua University

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
```

Identifiers

Keywords

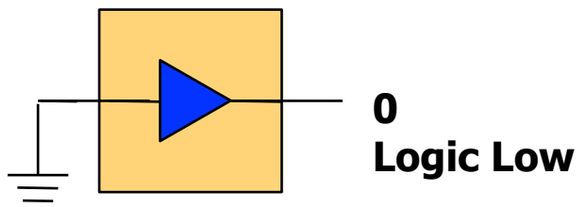
- Pre-defined non-escaped identifiers that used to define the language construct.
- All keywords are defined in lower cases.
- Examples
  - module, endmodule
  - input, output, inout
  - reg, integer, real, time
  - not, and, or, nand, nor, xor
  - parameter
  - begin, end
  - fork, join
  - always,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

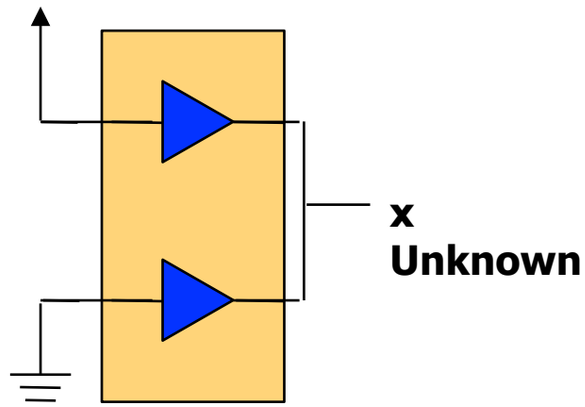
Case Sensitivity

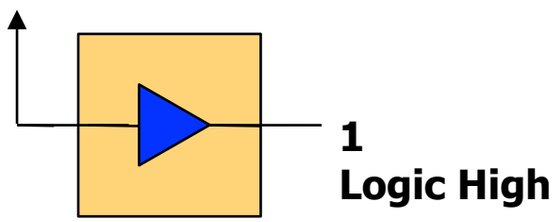
- Verilog is a case sensitive language.
- Use “-u” option in command line option for case-insensitive.

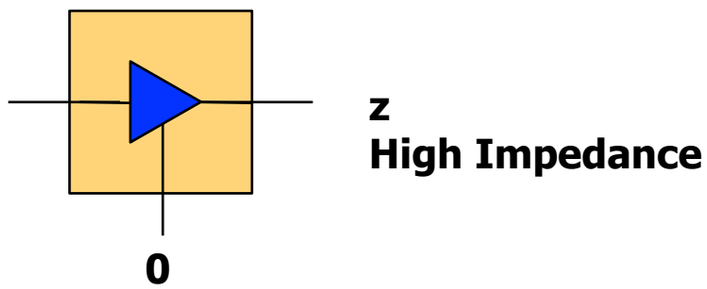
Value Sets

- 4-value logic system in Verilog









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

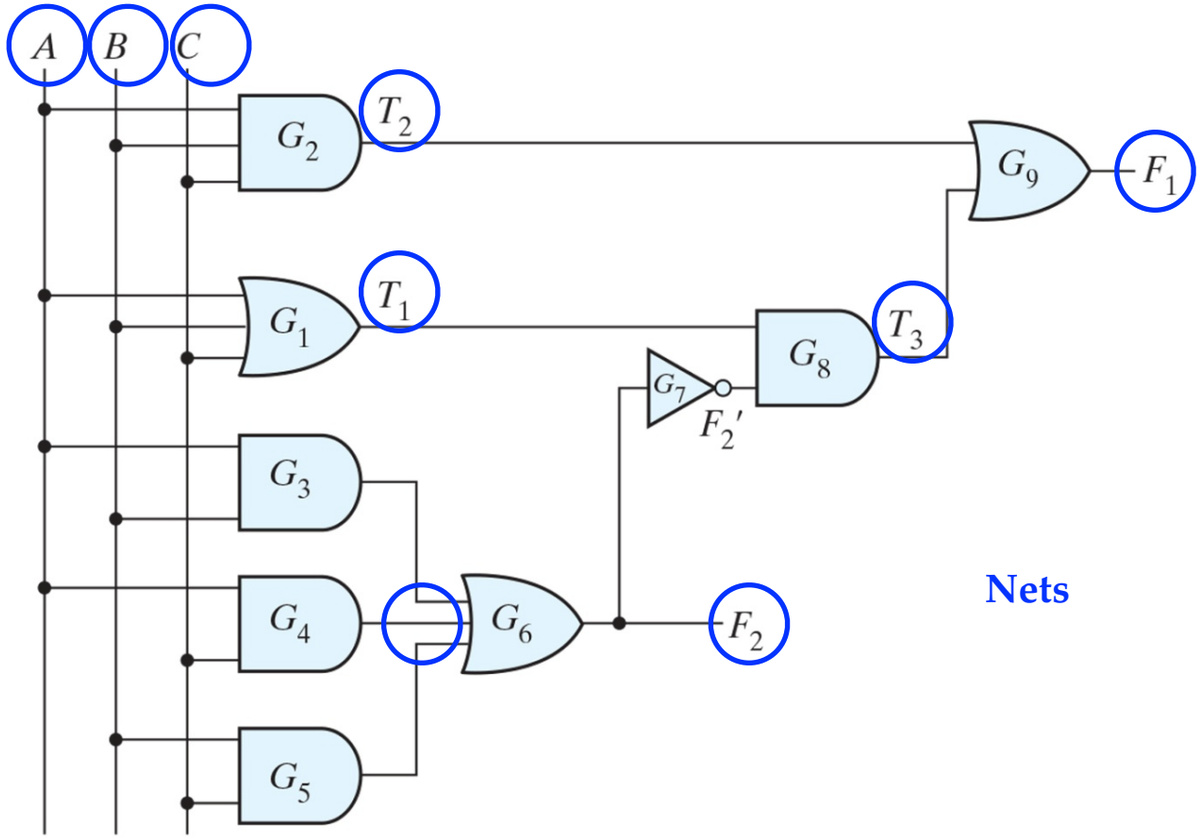
- Bit replication for 01010101
  - assign byte = {4{2'b01}};
- Sign extension
  - assign word = {{8{byte[7]}},byte};

Data Types

- Nets
  - Physical wire or group of wires connecting hardware elements in a module or between modules
- Registers
  - Represent a variable that can contain a value
  - Represent abstract storage elements

Nets (1/2)

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



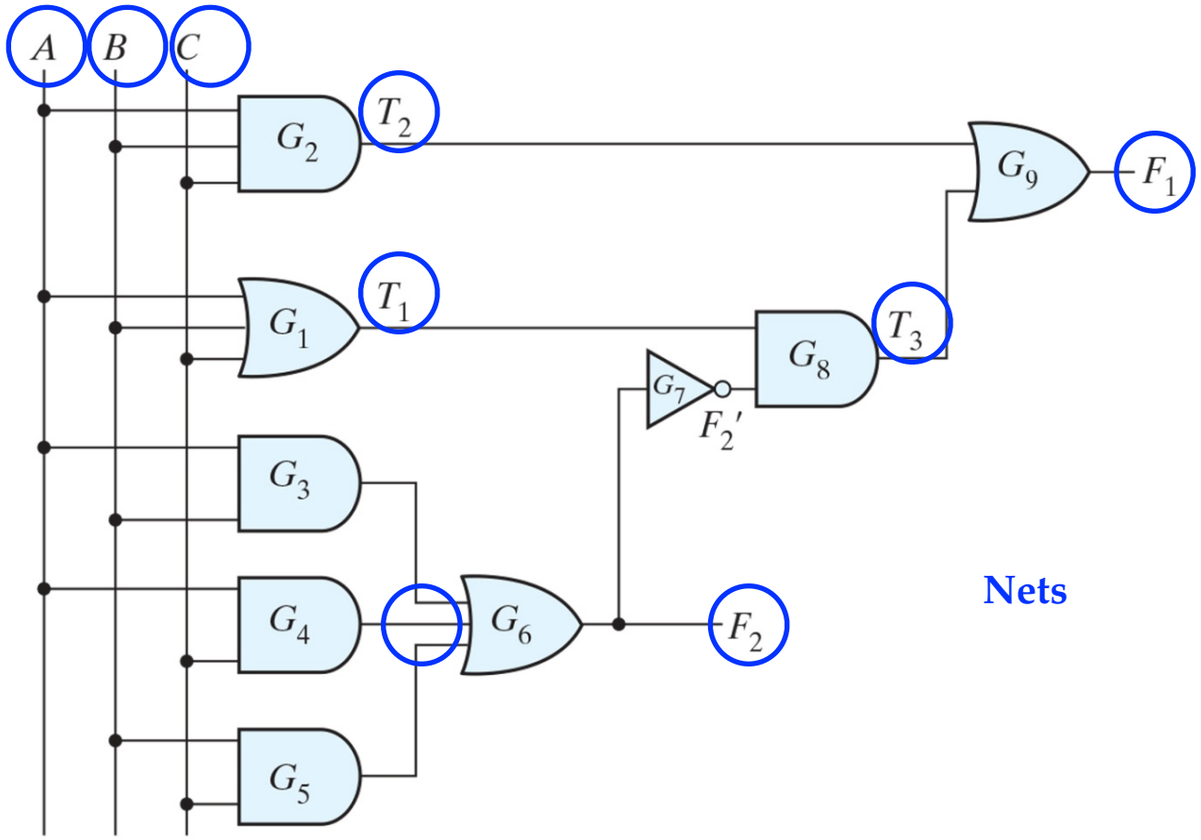
Nets

- F_2=AB+AC+BC
- T_1=A+B+C
- T_2=ABC
- T_3=F_2'T_1
- F_1=T_3+T_2F_1=T_3+T_2=F_2'T_1+ABC=(AB+AC+BC)'(A+B+C)+ABC =A'BC'+A'B'C+AB'C'+ABC

Nets (2/2)

Internal nets

```
module ex_net(F1, F2, A, B, C);
output F1, F2;
input A, B, C;
wire T1, T2, T3;

assign F2 = A&B | B&C | C&A;
assign T1 = A | B | C;
assign T2 = A & B & C;
assign T3 = (~F2) & T1;
assign F1 = T3 | (T2 & F1);

endmodule
```



Nets

- F_2=AB+AC+BC
- T_1=A+B+C
- T_2=ABC
- T_3=F_2'T_1
- F_1=T_3+T_2F_1=T_3+T_2=F_2'T_1+ABC=(AB+AC+BC)'(A+B+C)+ABC =A'BC'+A'B'C+AB'C'+ABC

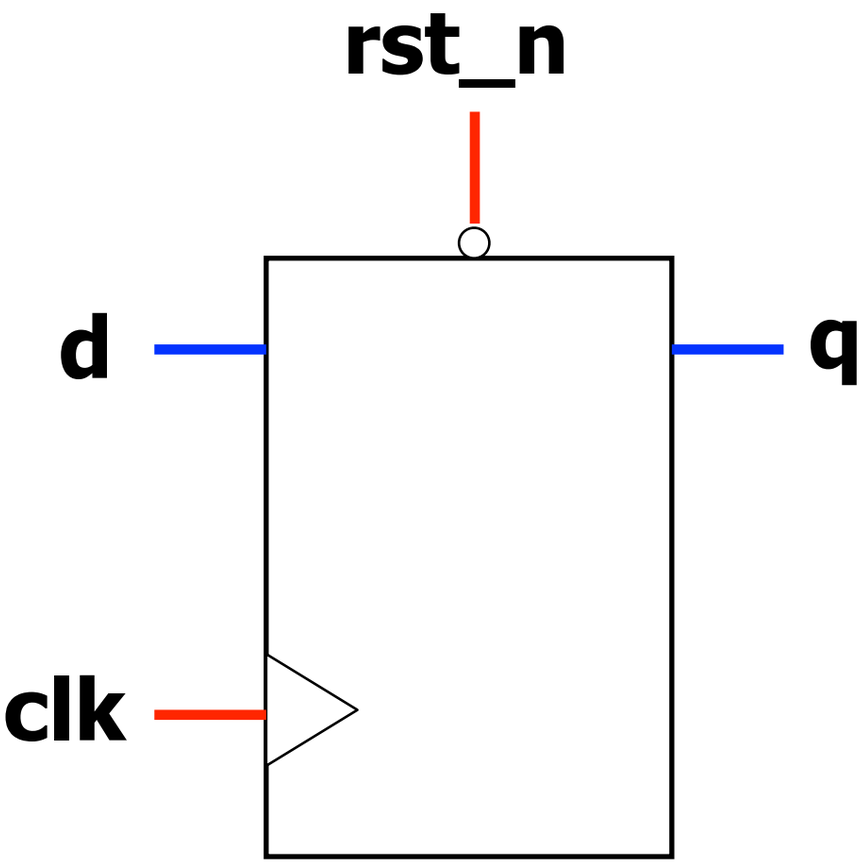
Registers (1/3)

- 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**

Registers (2/3)

```
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
```

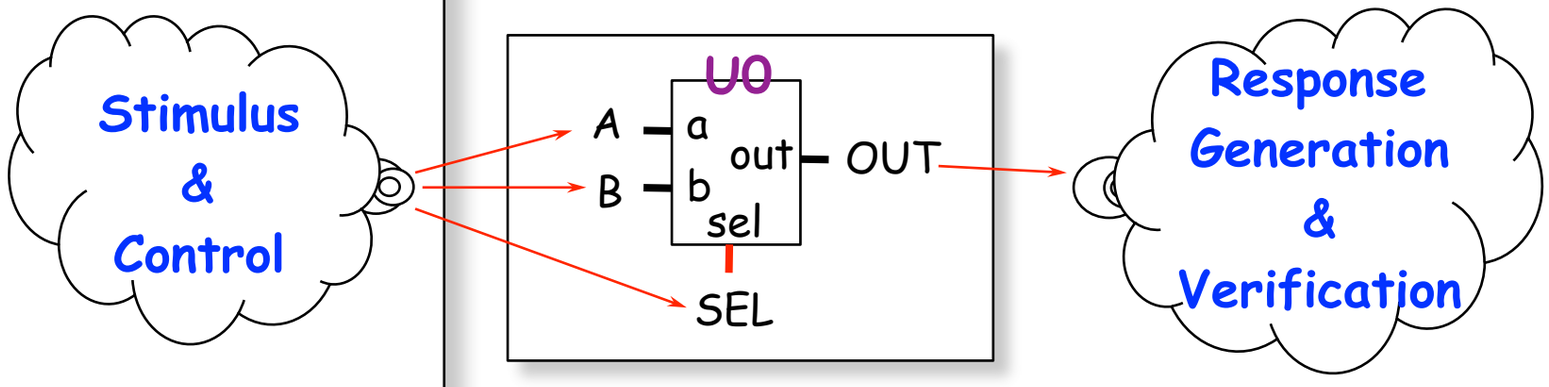
D-type Flip-flop



Registers (3/3)

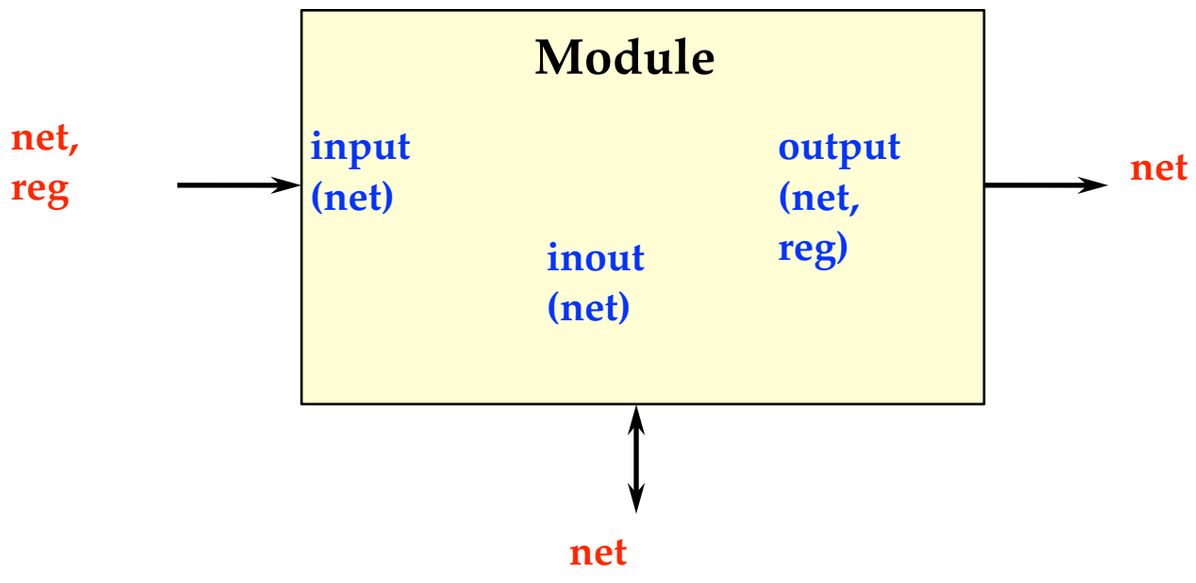
```
module test_smux;
wire OUT;
reg A,B,SEL;

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

initial
begin
  A=0;B=0;SEL=0;
  #10 A=0;B=0;SEL=1;
  #10 A=0;B=1;SEL=0;
  #10 A=0;B=1;SEL=1;
  #10 A=1;B=0;SEL=0;
  #10 A=1;B=0;SEL=1;
  #10 A=1;B=1;SEL=0;
  #10 A=1;B=1;SEL=1;
end

endmodule
```

Testbench for SMUX



Choosing the Correct Data Types

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

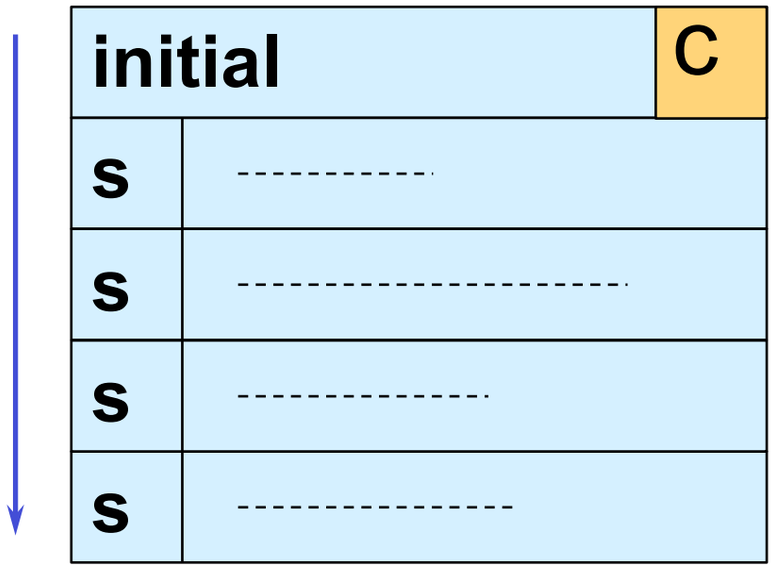


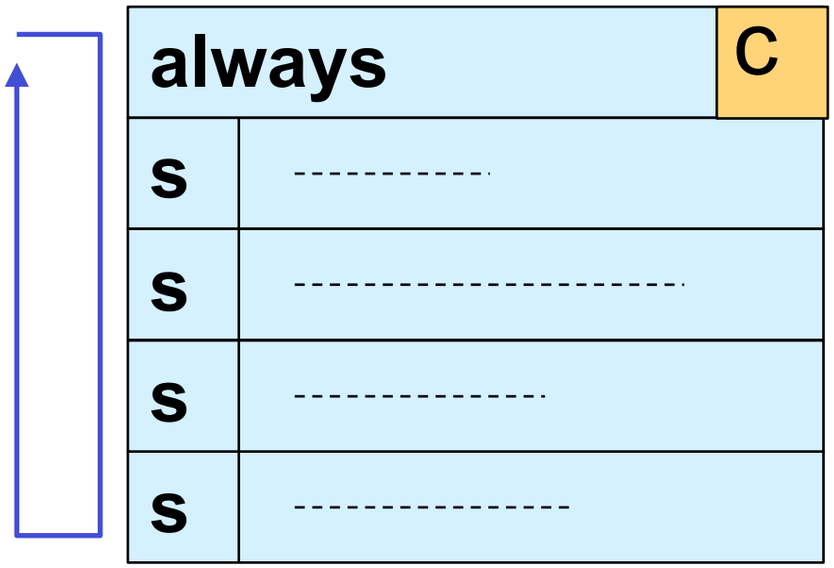
RTL Modeling

- Dataflow modeling
  - A higher level of abstraction the gate-level modeling
  - The expression that can be used in continuous assignment is dataflow modeling
- Behavioral modeling
  - Algorithmic approach to hardware implementation
  - The constructs in behavioral modeling closely resemble those used in the C programming language
  - The behavior of the design is described using procedural blocks (**initial** and **always** statement)

Procedure Blocks (1/2)

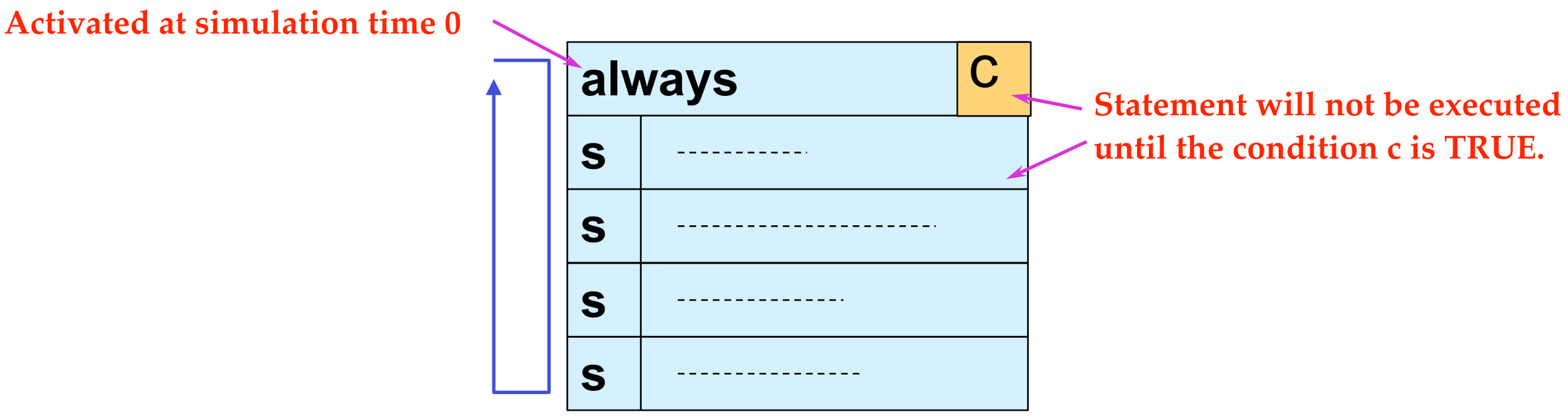
- Basis of behavioral modeling
- Two types
  - **initial**: execute only once
  - **always**: execute in a loop





Procedure Blocks (2/2)

- All procedure blocks are activated at simulation time 0
  - The block will not be executed until the enabling condition evaluates TRUE
  - Without the enabling condition, the clock will be executed immediately



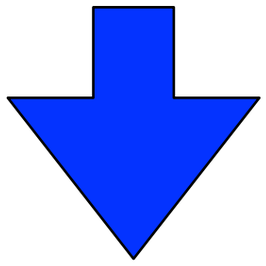
Procedure Assignments

- Blocking assignments (=)
  - The assignment completes execution before the next statement executes
- Non-block assignments (<=)
  - Allow the scheduling of assignments without blocking execution of the following statements in a **sequential** procedure block
  - To synchronize assignment statements so that they appear to execute at the same time

Blocking v. Non-blocking Assignments

```
always @(posedge clk)
  x1 = x2;

always @(posedge clk)
  x2 = x3;
```

Racing condition

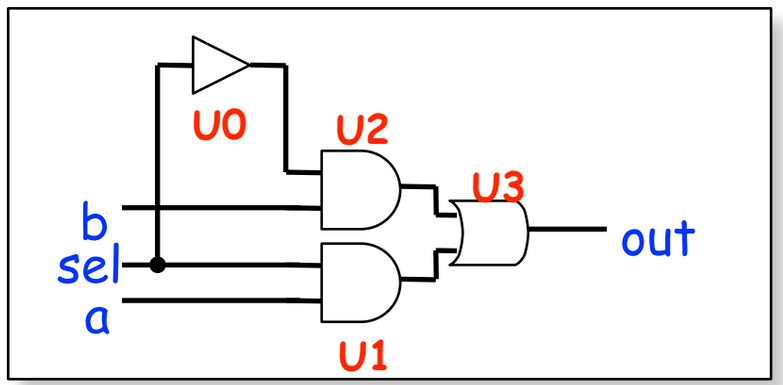


```
always @(posedge clk)
  x1 <= x2;

always @(posedge clk)
  x2 <= x3;
```

Conditional Statements

- **if-else** statement
  - Not for large MUXs (synthesizable)

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

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

endmoudle
```

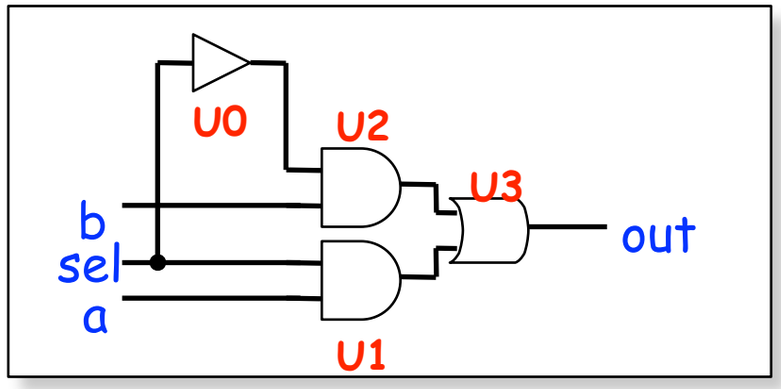



```
if (cond0)
   exp1;
else if (cond1)
  exp2;
else if (cond2)
  exp3;
else
  exp4;
```

Case Statements

- **case** statement
  - Usually for large MUXs (synthesizable)

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

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

endmodule
```



Loop Statements

- Two meanings
  - For testbench: repeated execution of procedural statements
  - For synthesis: multiple of building blocks
- Four types of loop statements
  - for
  - while
  - repeat
  - forever
- Must be placed within an **initial** or **always** block

Loop Statements

for (initial control variable assignment; test expression; control variable assignment)
  procedural statement or block of procedural statements

while (expression)
  procedural statement or block of procedural statements

repeat (loop count expression)
  procedural statement or block of procedural statements

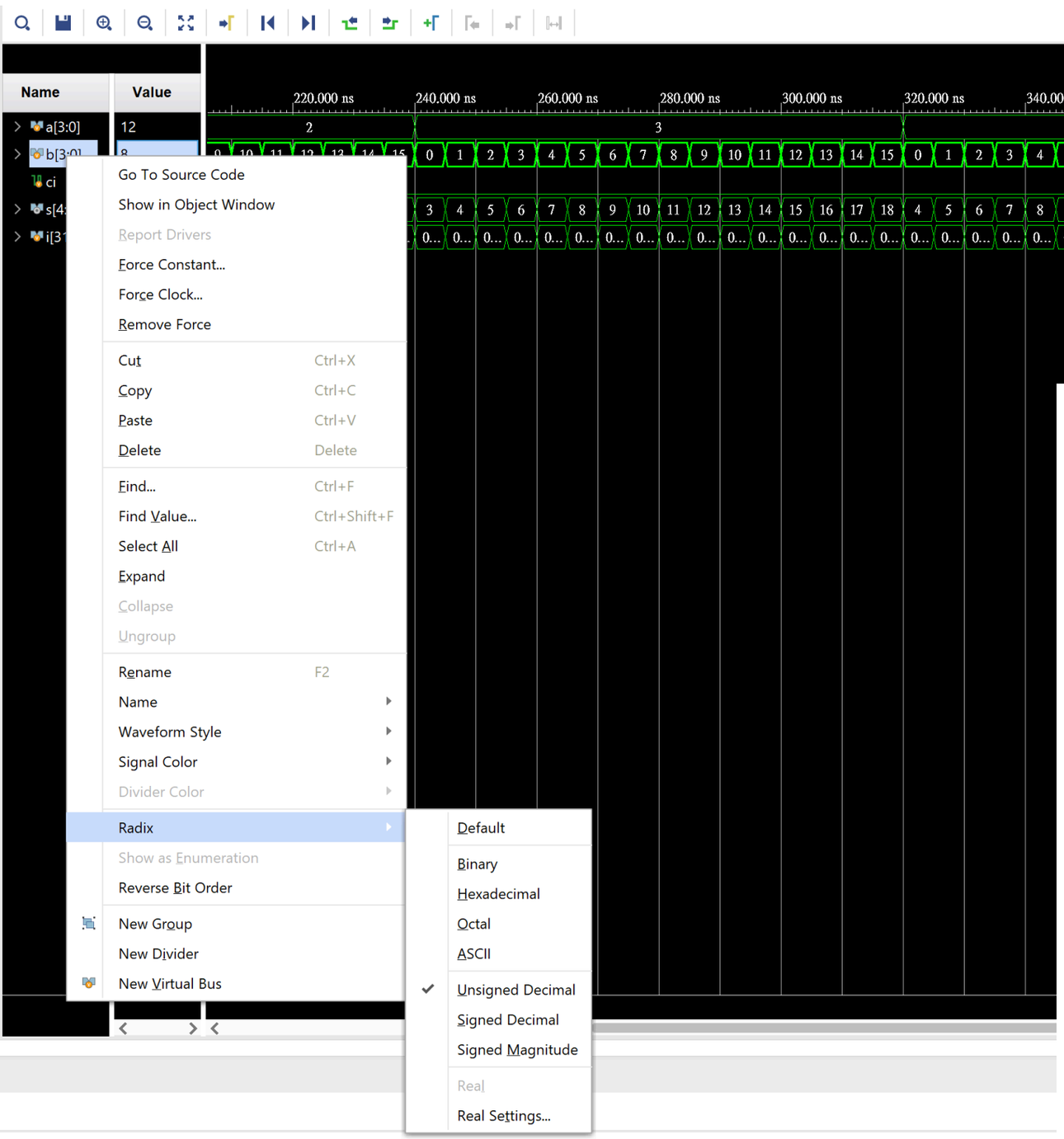
forever
  procedural statement

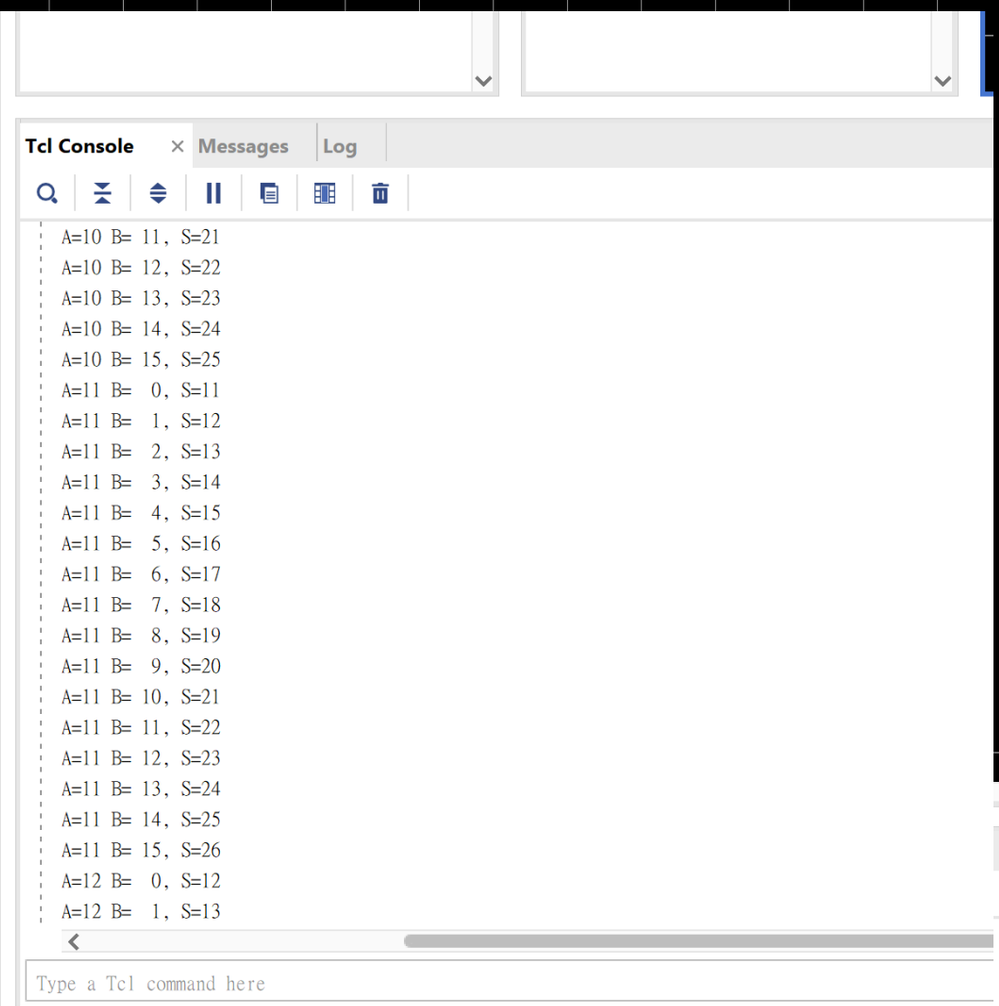
4-bit + 4-bit Ripple Carry Adder

```
module rca(
  s,  // sum
  a,  // input a
  b,  // input b
  ci // carry input
);

output [4:0] s;
input [3:0] a, b;
input ci

assign s = a + b + ci;

endmodule
```

```
module t_rca;
wire [4:0] s;
reg [3:0] a, b;
reg ci;
integer i;

rca U0(.s(s),.a(a),.b(b),.ci(ci));

initial
begin
  ci=0;
  $monitor("A=%d B= %d, S=%d", a, b, s);
  for (i=0;i<256;i=i+1)
  begin
    {a,b} = i;
    #5;
  end
end
endmodule
```

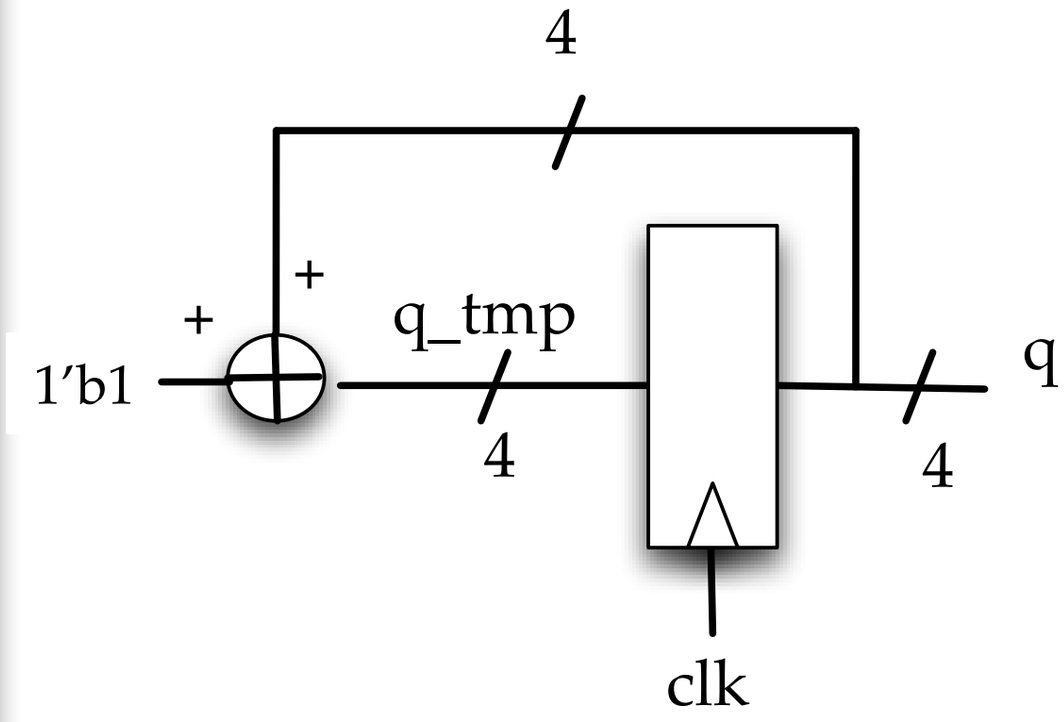
Simulation Results





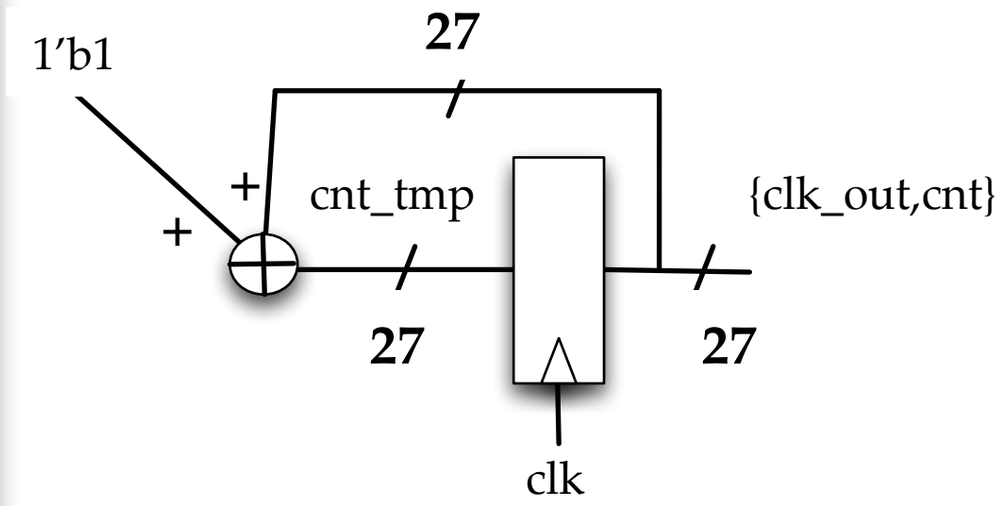
Binary Up Counter

```
`define CNT_BIT_WIDTH 4
module bincnt(
  q,  // output
  clk,  // global clock
  rst_n  // active low reset
);

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

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

// Combinational logics
always @*
  q_tmp = q + 1'b1;

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

endmodule
```



Frequency Divider

```
`define FREQ_DIV_BIT 27
module freqdiv(
  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
```

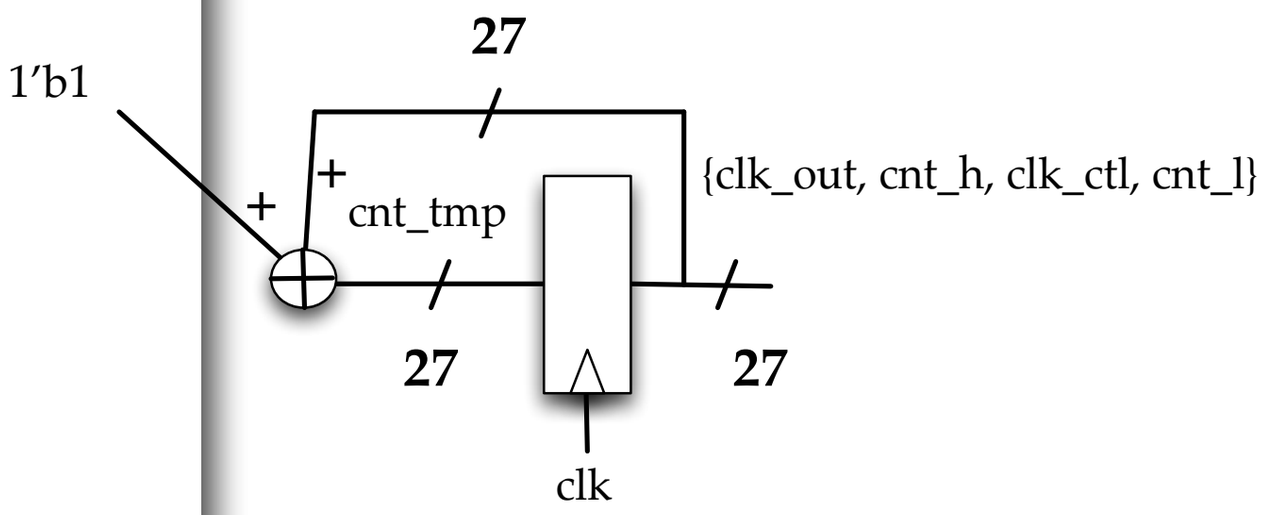


cnt_tmp[26:0]

cnt[25:0]

Frequency Divider

```
`define FREQ_DIV_BIT 27
module freqdiv27(
  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 [8:0] 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
```

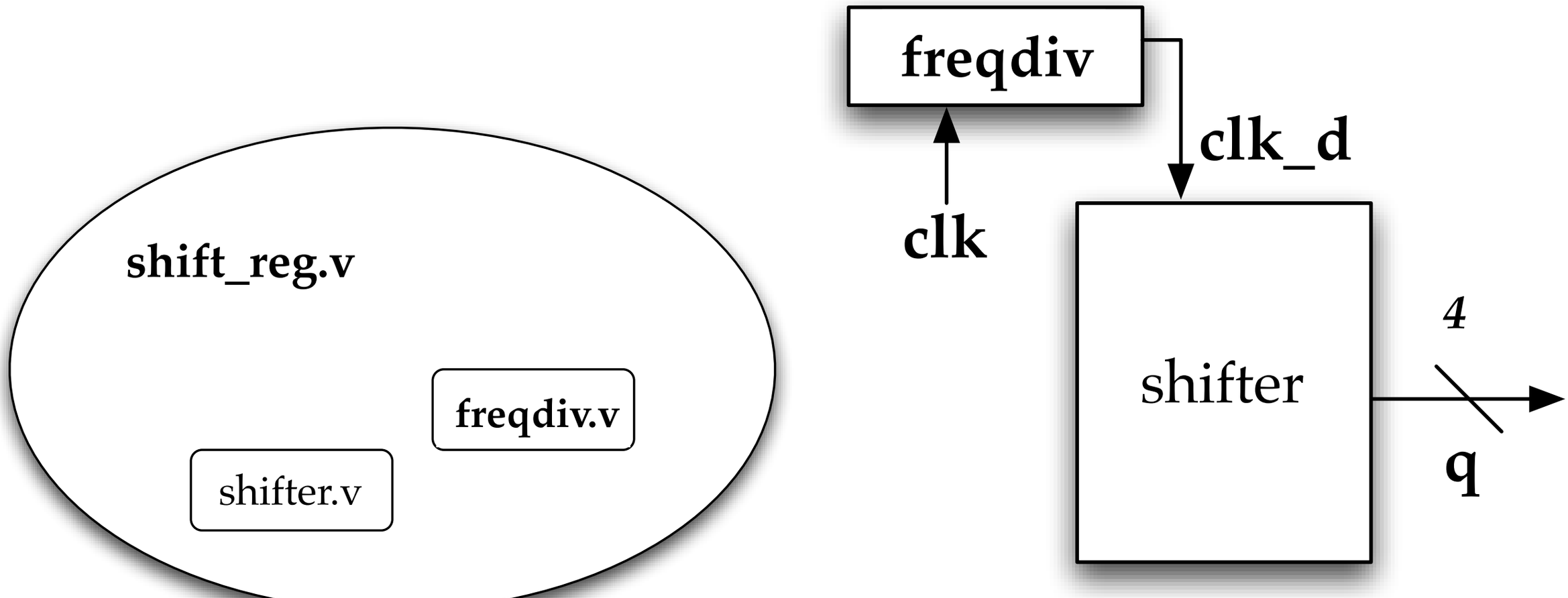


clk_out MSB　　clk_ctl 16th-17th

1	9	2	15

Modularized Shift Register

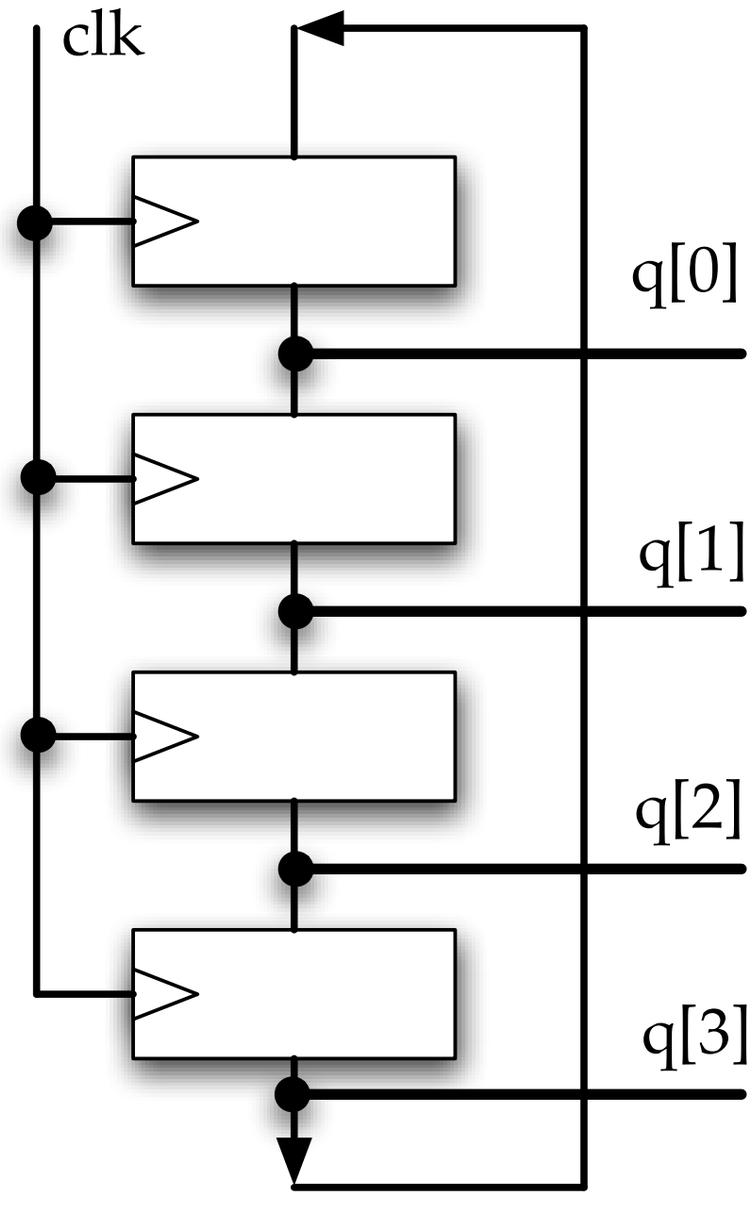
Shift Register



shifter.v

```
`define BIT_WIDTH 4
module shifter(
  q,  // shifter output
  clk,  // global clock
  rst_n  // active low reset
);

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

reg [`BIT_WIDTH-1:0] q;  // output

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
 if (~rst_n)
 begin
   q<=`BIT_WIDTH'b0001;
 end
 else
 begin
   q[0]<=q[3];
   q[1]<=q[0];
   q[2]<=q[1];
   q[3]<=q[2];
 end
endmodule
```

initial value 0001



Top Module (shift_reg.v)

```
`define BIT_WIDTH 4
module shift_reg(
  q,  // LED output
  clk, // global clock
  rst_n  // active low reset
);

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

wire clk_d; // divided clock
wire [`BIT_WIDTH-1:0] q;  // LED output
```

1

```
// Insert frequency divider (freq_div.v)
freqdiv U_FD(
  .clk_out(clk_d),  // divided clock output
  .clk(clk),  // clock from the crystal
  .rst_n(rst_n)  // active low reset
);

// Insert shifter (shifter.v)
shifter U_D(
  .q(q),  // shifter output
  .clk(clk_d),  // clock from the frequency divider
  .rst_n(rst_n)  // active low reset
);

endmodule
```

2

shift_reg.xdc

```
# Clock
set_property PACKAGE_PIN W5 [get_ports {clk}]
set_property IOSTANDARD LVCMOS33 [get_ports {clk}]

# active low reset
set_property PACKAGE_PIN V17 [get_ports {rst_n}]
set_property IOSTANDARD LVCMOS33 [get_ports {rst_n}]

# LEDs
set_property PACKAGE_PIN U16 [get_ports {q[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {q[0]}]
set_property PACKAGE_PIN E19 [get_ports {q[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {q[1]}]
set_property PACKAGE_PIN U19 [get_ports {q[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {q[2]}]
set_property PACKAGE_PIN V19 [get_ports {q[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {q[3]}]
```