Verilog HDL (2)

Hsi-Pin Ma

http://lms.nthu.edu.tw/course/38127
Department of Electrical Engineering
National Tsing Hua University

Lexical Conventions

White Space and Comments

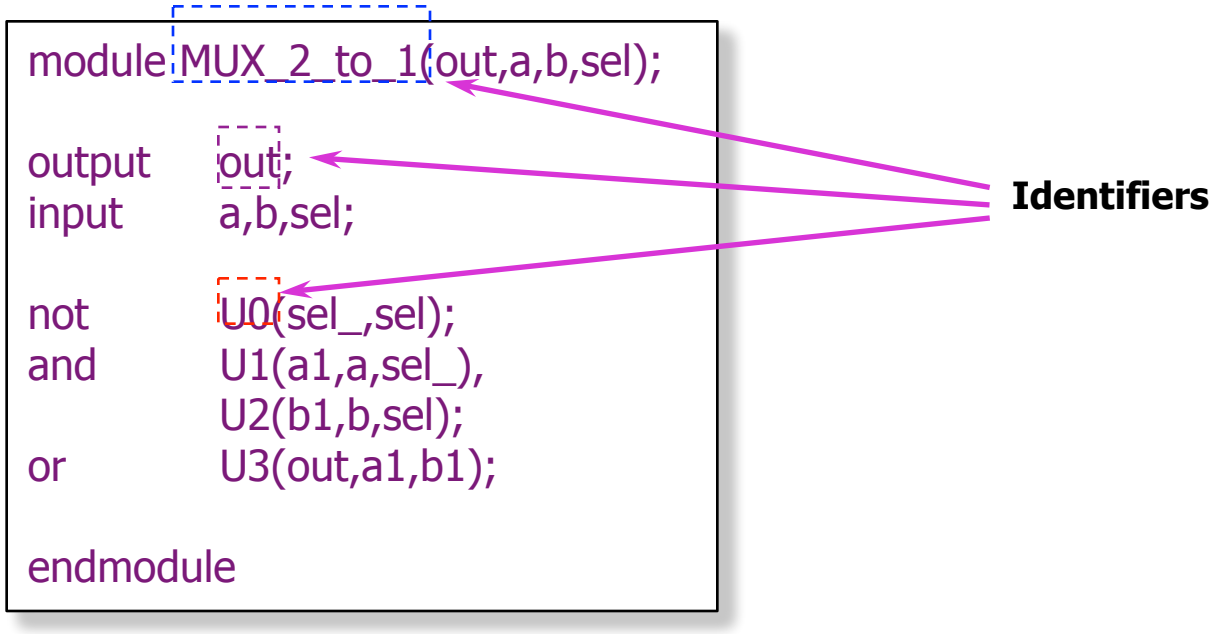
- White space makes code more readable
  - Include blank space (\b), tabs (\t), and carriage return (\n).
- Comments
  - /* … */ : mark more than one line
  - // : mark only one line.

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.



Keywords

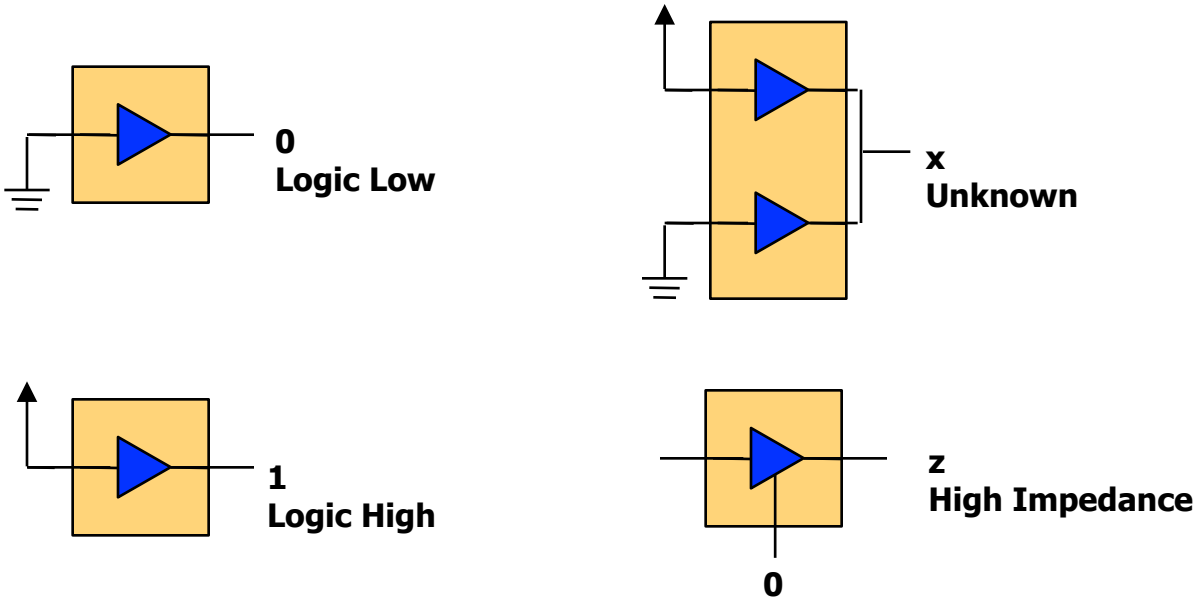
- Pre-defined non-escaped identifiers that used to define the language construct.
- All keywords are defined in lower cases.
- Examples
  - module, endmodule
  - input, output, inout
  - reg, integer, real, time
  - not, and, or, nand, nor, xor
  - parameter
  - begin, end
  - fork, join
  - always,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

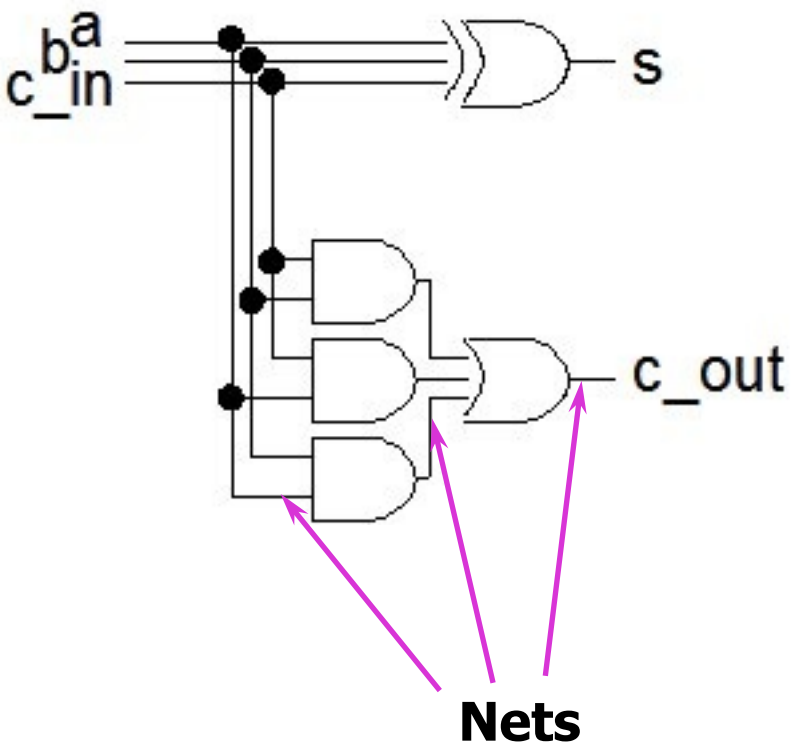
- Bit replication for 01010101
  - assign byte = {4{2'b01}};
- Sign extension
  - assign word = {{8{byte[7]}},byte};

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

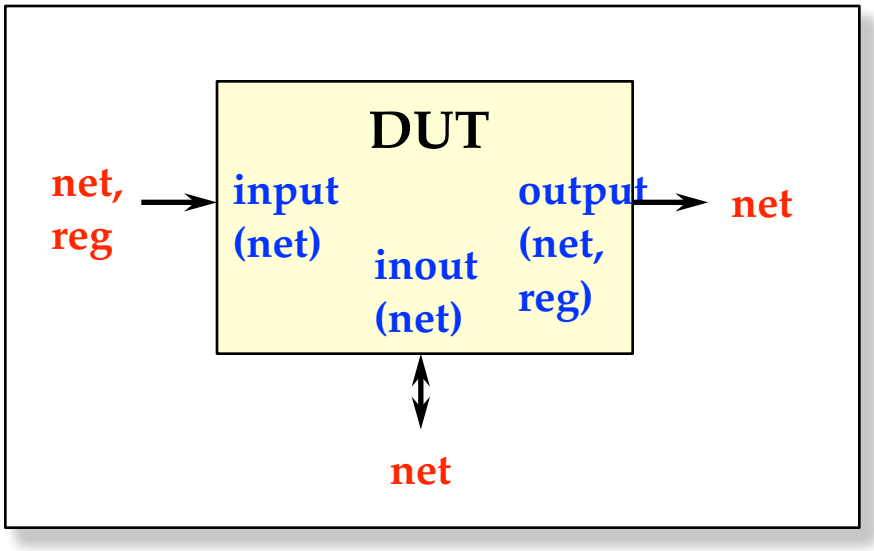
- reg
  - Unsigned integer variable of varying bit width
- integer
  - Signed integer variable, 32-bit wide. Arithmetic operations produce 2's complement results.
- real
  - Signed floating point variable, double precision
- time
  - Unsigned integer variable, 64-bit wide.
- Do not confuse register data 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.

Behavior Modeling

At every positive edge of CLOCK

If PRESET and CLEAR are not low

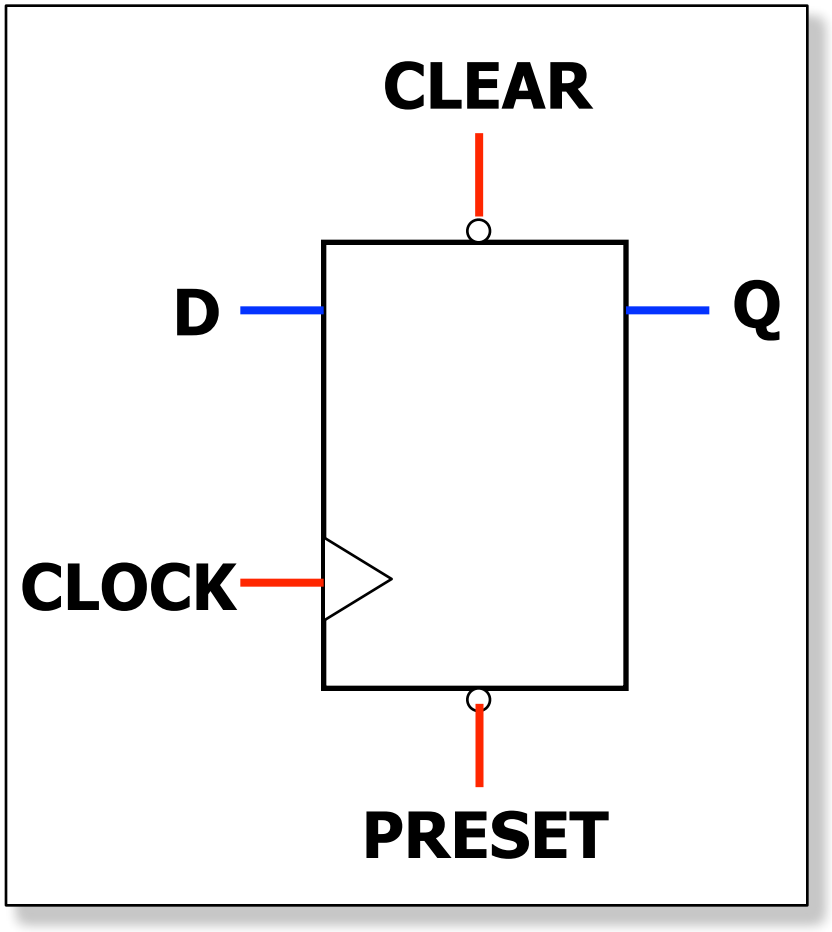
set Q to the value of D

Whenever PRESET goes low

set Q to logic 1

Whenever CLEAR goes low

set Q to logic 0

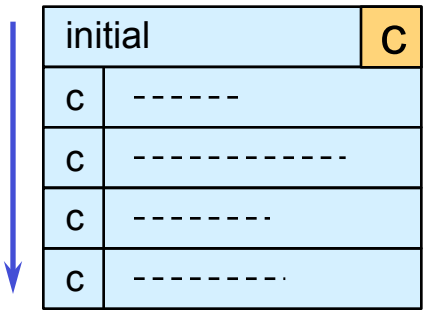


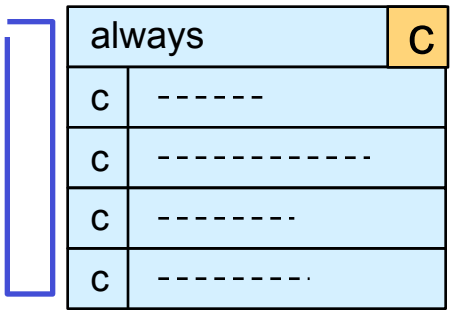
Behavior Modeling

- In behavior modeling, you must specify your circuit's
  - Action
    - How to model the circuit's behavior
  - Timing control
    - Timing
    - Condition
- Verilog supports the following structures for behavior modeling
  - Procedural block
  - Procedural assignment
  - Timing control
  - Control statement

Procedural Blocks

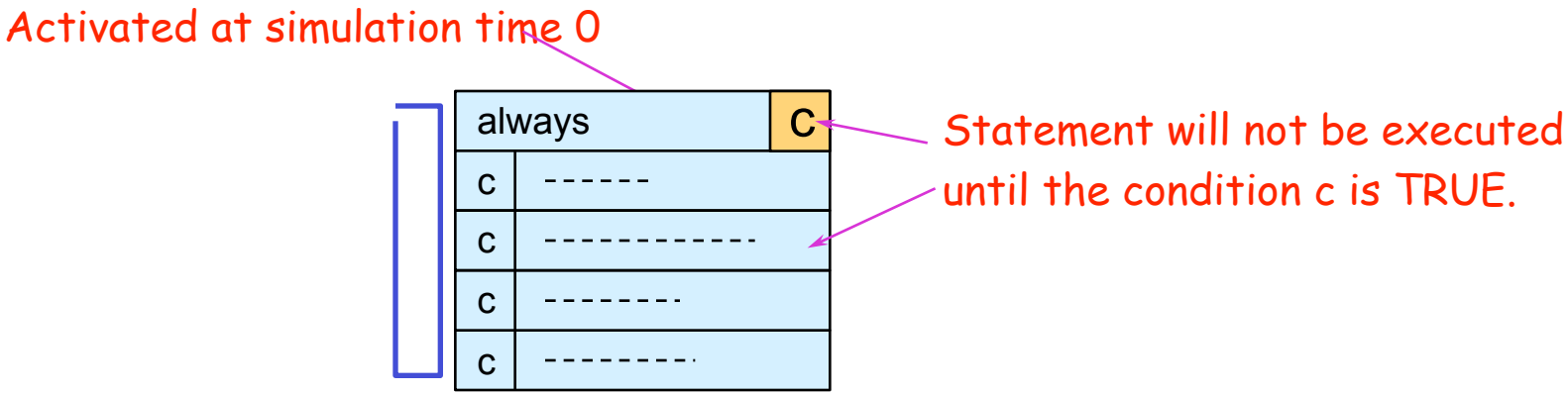
- In Verilog, procedural blocks are basis of behavior modeling
- Procedural blocks are of two types
  - initial procedural block, which execute only once
  - always procedural block, which execute in a loop





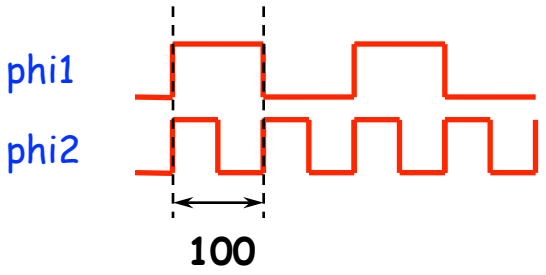
Procedural Blocks

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



Procedural Blocks

```
module clock_gen(phi1,phi2);
output      phi1,phi2;
reg  phi1,phi2;

initial
begin
    phi1=0;phi2=0
end

always
    #100 phi1=~phi1;

always      @(posedge phi1)
begin
    phi2=1;
    #50     phi2=0;
    #50     phi2=1;
    #50     phi2=0;
end
endmodule
```



These procedural blocks are activated and executed at simulation time 0

This procedural block is activated at simulation time 0 but executed at positive edge of phi1

Procedural Blocks

- Three components
  - Procedural assignment statements
  - High-level programming language constructs
  - Timing controls
- Using the first two components to model the actions of the circuit.
- Using timing controls to model when should these actions happen.

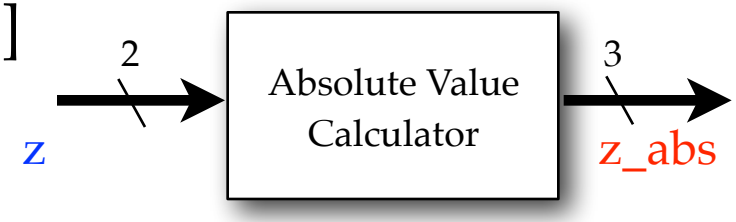
Procedural Timing Control

- Three types
  - Simple delay control
    - #50 clk=~clk;
  - Event control
    - @* sum=a+b+ci;
    - @(posedge clk) q<=d;
  - Level-sensitive timing control

Examples

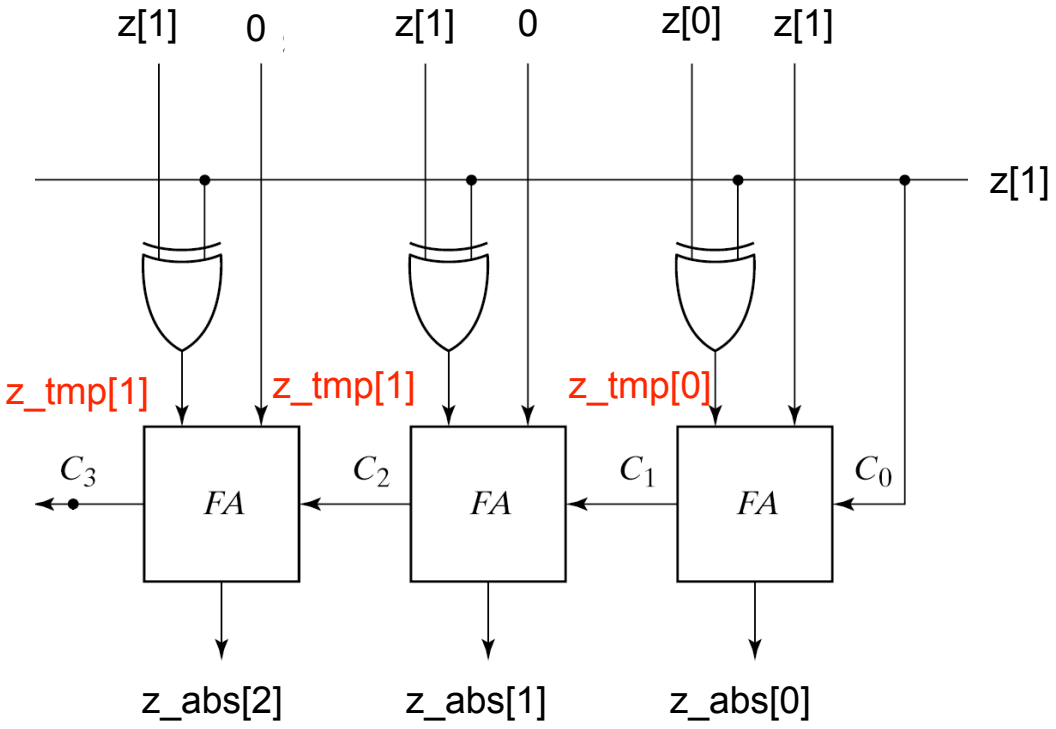
2-bit Absolute Value Calculator (1/2)

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)

5

Module (abs.v)

```
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
```

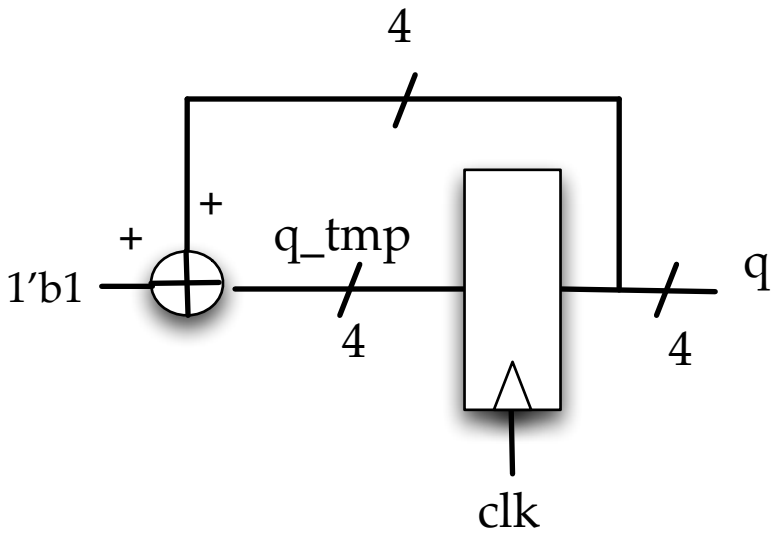
6

Testbench (t_abs.v)

```
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
```

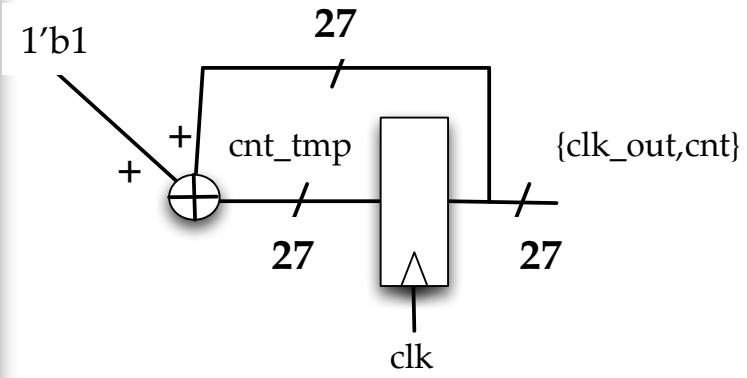
Binary Up Counter

```
`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
```

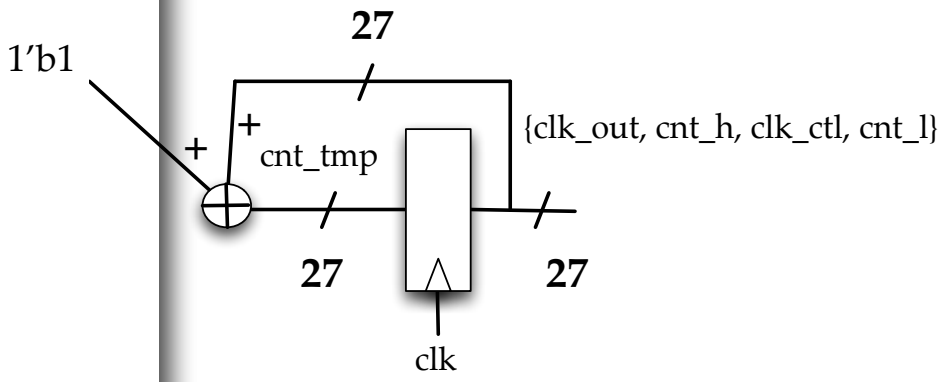


cnt_tmp[26:0]

cnt[26:0]

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
```

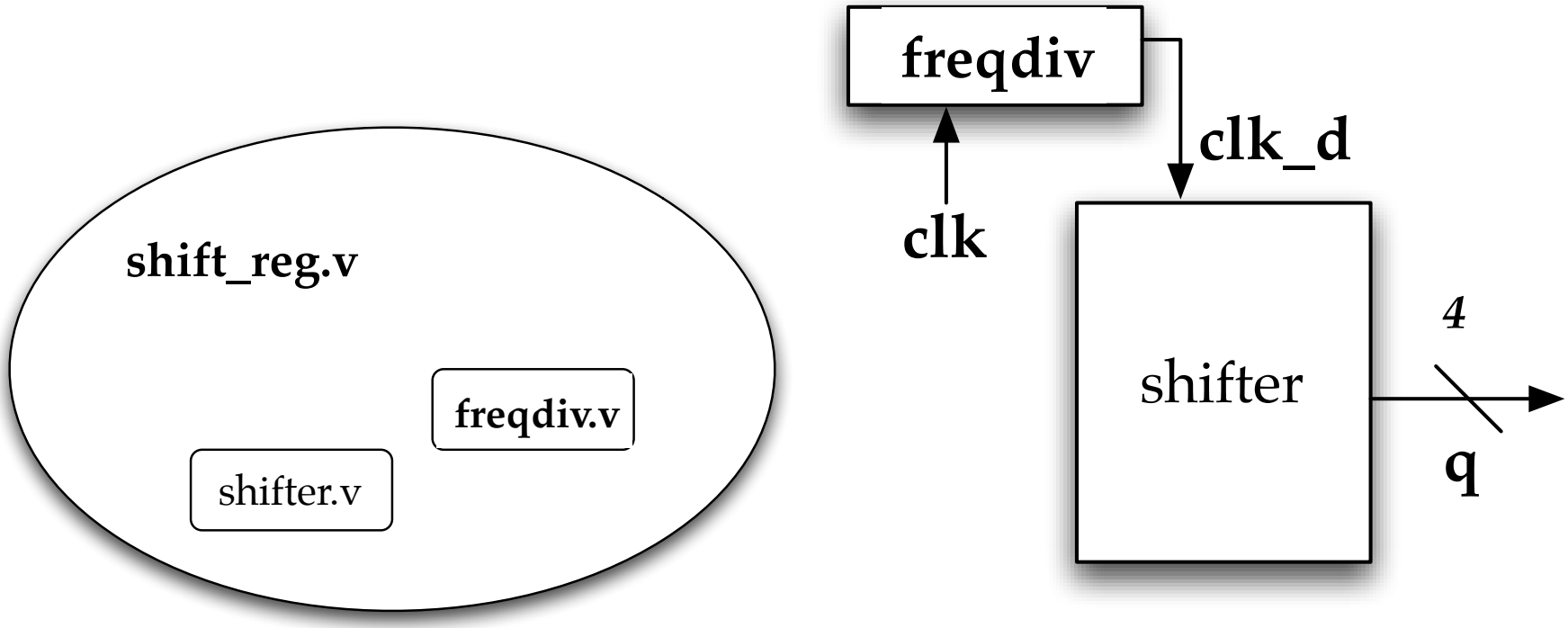


clk_out MSB clk_ctl 16th-17th

1	9	2	15

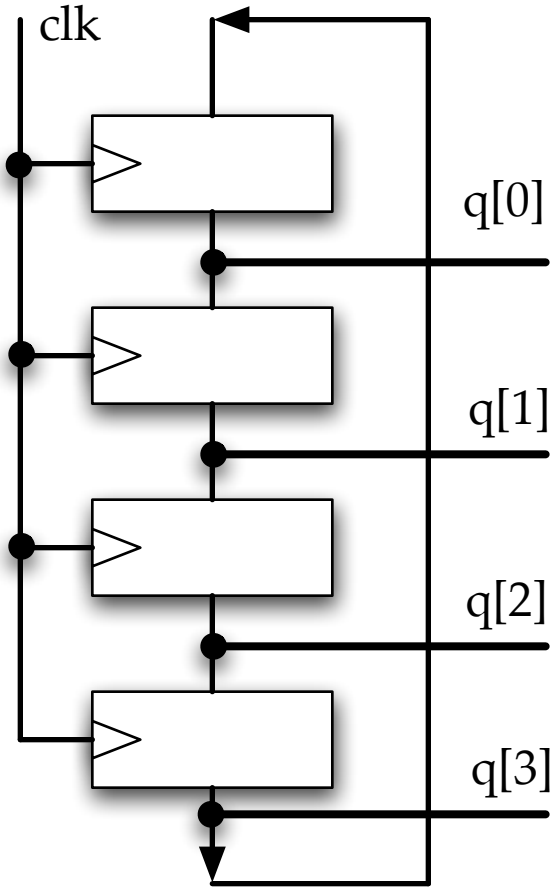
Modularized Shift Register

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                    initial value 0001
  else
 begin
    q[0]<=q[3];
    q[1]<=q[0];
    q[2]<=q[1];
    q[3]<=q[2];
  end
endmodule
```



Top Module (shift_reg.v)

1

```
`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
```

2

```
// 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
```

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]}]
```