Logic Design with Verilog II: Basic Syntax and Combinational Logic

Jae W. Lee (jaewlee@snu.ac.kr)
Department of Computer Science and Engineering
Seoul National University

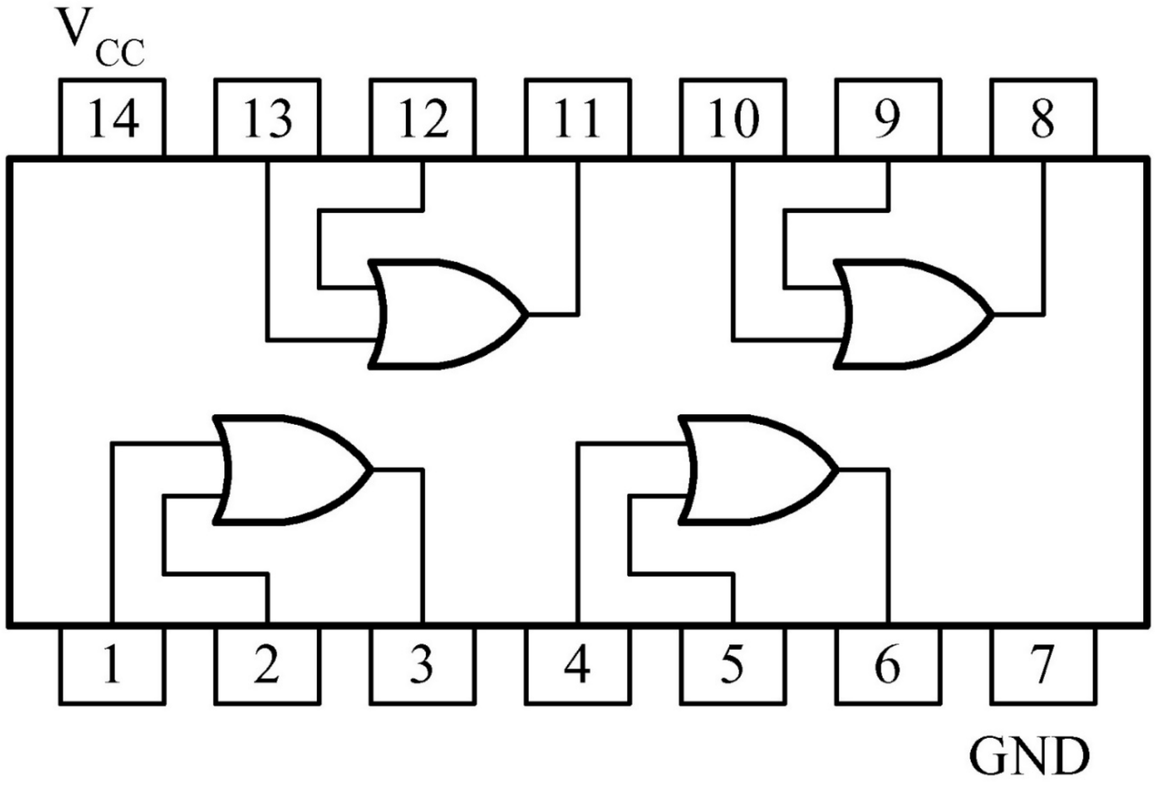
Slide credits: Prof. Ming-Bo Lin (Digital System Designs and Practices Using Verilog HDL and FPGAs)

Outline

- Modules
  - Concept
  - Definitions
  - Parameters
  - Module instantiation
  - Parameter values
- Generate statements
- Module modeling styles
- Simulation

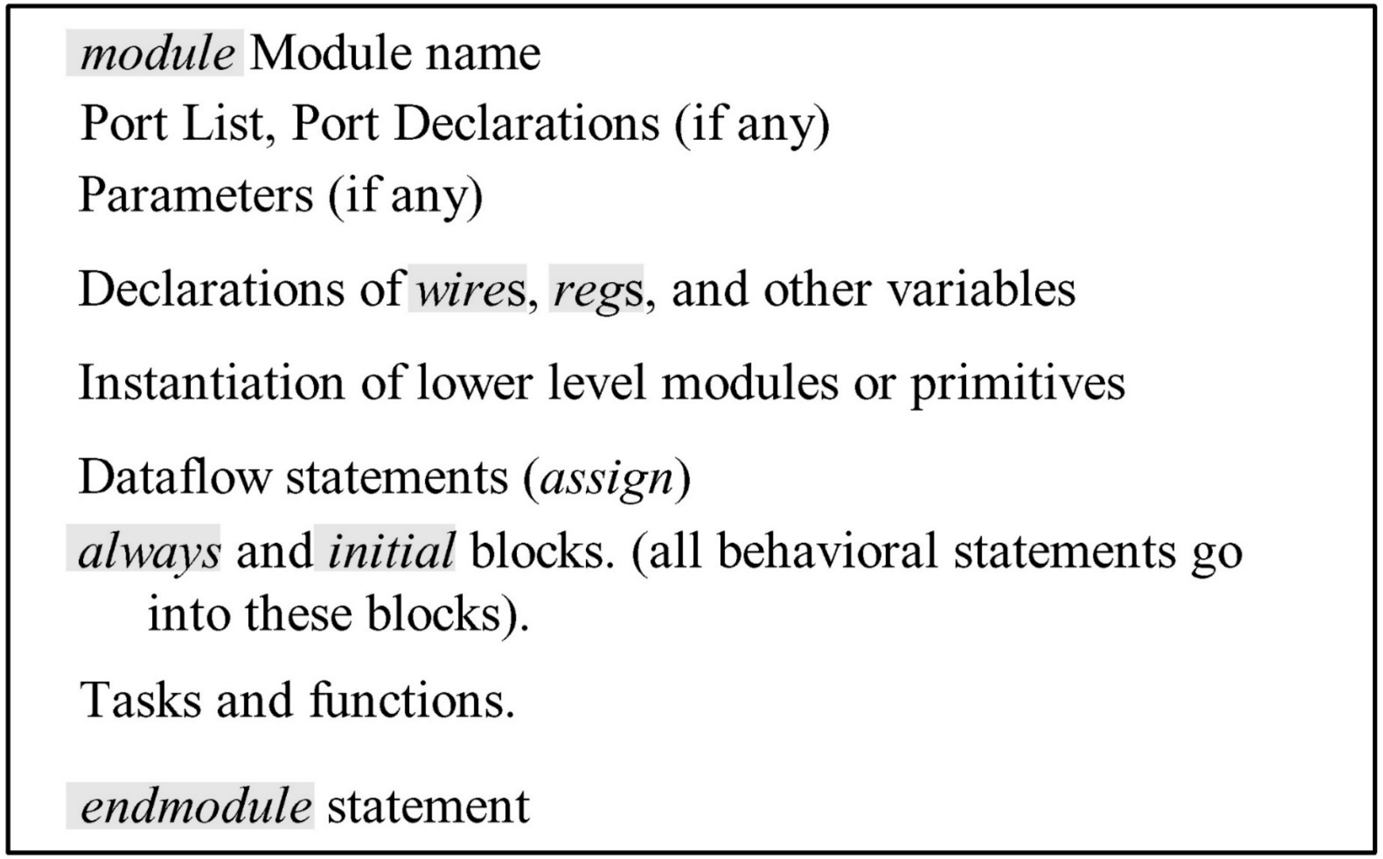
Modules: Concept

- Concept of module
  - A core circuit (called internal or body)
  - An interface (called ports)



Modules: Concept

- Module in Verilog: basic building block



module Module name

Port List, Port Declarations (if any)

Parameters (if any)

Declarations of *wire*s, *reg*s, and other variables

Instantiation of lower level modules or primitives

Dataflow statements (*assign*)

always and *initial* blocks. (all behavioral statements go into these blocks).

Tasks and functions.

endmodule statement

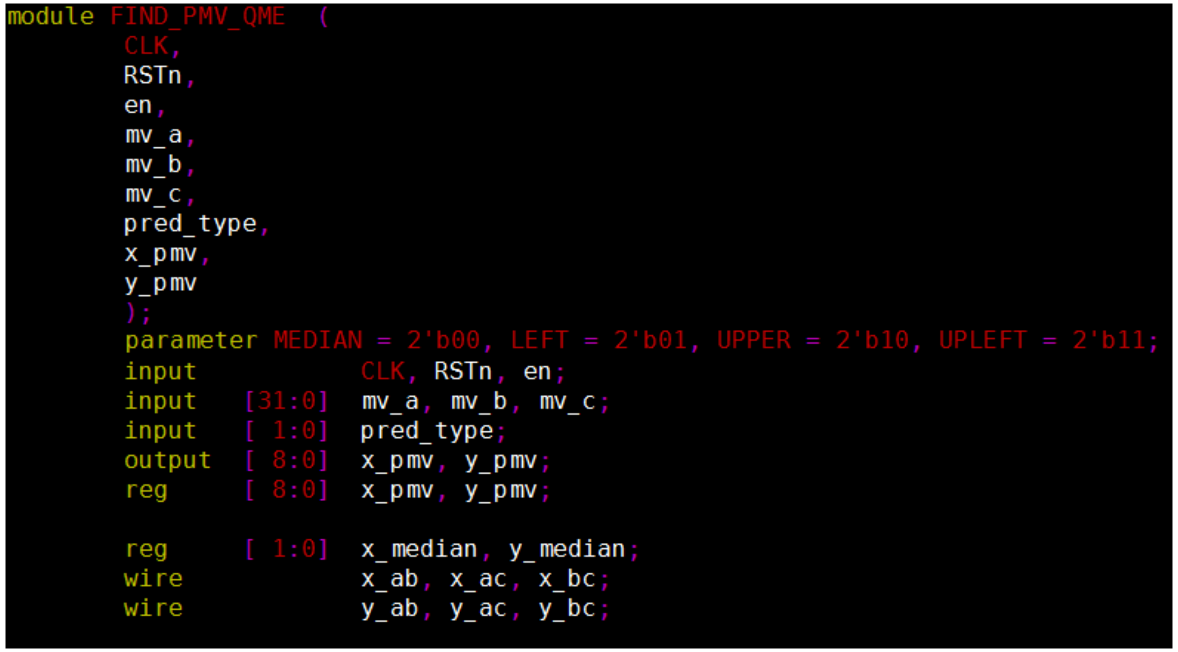
Modules: Definitions

- Module definitions

```
// port list style
module module_name [#(parameter_declarations)][port_list];
parameter_declarations; // if no parameter ports are used
port_declarations;
other_declaration;
statements;
endmodule

// port list declaration style
module module_name [#(parameter_declarations)][port_declarations];
parameter_declarations; // if no parameter ports are used
other_declarations;
statements;
endmodule
```

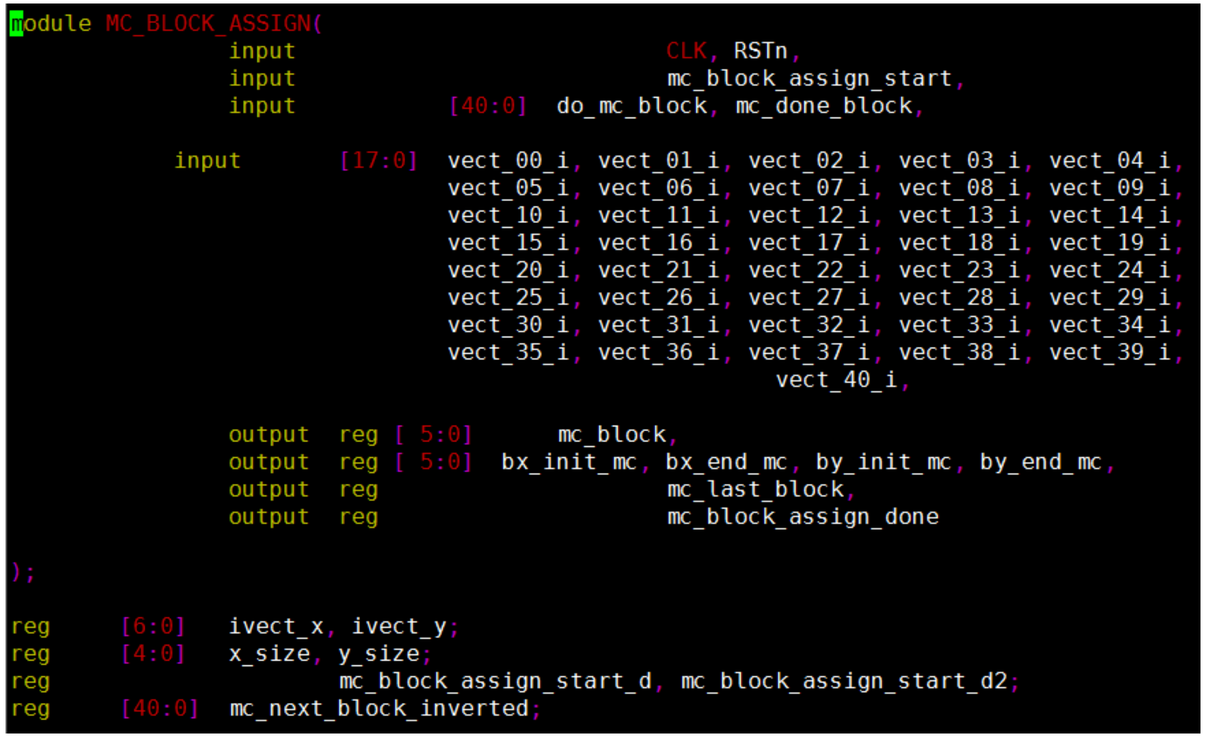
Modules: Definitions

- Example: module definitions



```
module FIND_PMV_QME  (
        CLK,
        RSTn,
        en,
        mv_a,
        mv_b,
        mv_c,
        pred_type,
        x_pmv,
        y_pmv
        );
        parameter MEDIAN = 2'b00, LEFT = 2'b01, UPPER = 2'b10, UPLEFT = 2'b11;
        input           CLK, RSTn, en;
        input   [31:0]  mv_a, mv_b, mv_c;
        input   [ 1:0]  pred_type;
        output  [ 8:0]  x_pmv, y_pmv;
        reg     [ 8:0]  x_pmv, y_pmv;

        reg     [ 1:0]  x_median, y_median;
        wire            x_ab, x_ac, x_bc;
        wire            y_ab, y_ac, y_bc;
```

Port list style

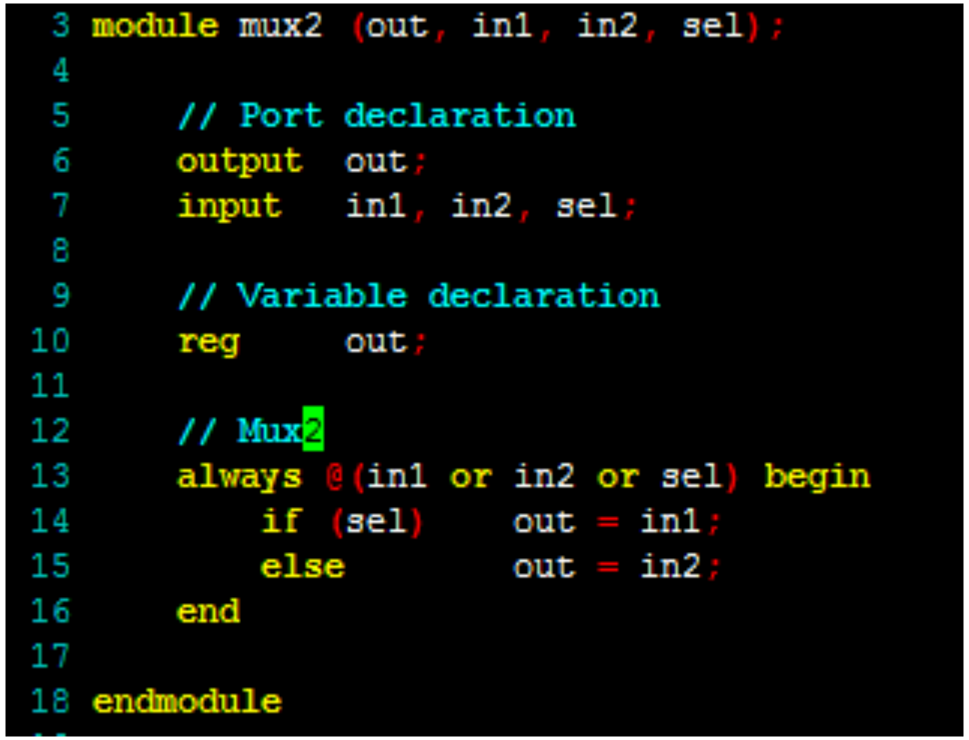


```
module MC_BLOCK_ASSIGN(
        input                   CLK, RSTn,
        input                   mc_block_assign_start,
        input           [40:0]  do_mc_block, mc_done_block,

      input     [17:0]  vect_00_i, vect_01_i, vect_02_i, vect_03_i, vect_04_i,
                        vect_05_i, vect_06_i, vect_07_i, vect_08_i, vect_09_i,
                        vect_10_i, vect_11_i, vect_12_i, vect_13_i, vect_14_i,
                        vect_15_i, vect_16_i, vect_17_i, vect_18_i, vect_19_i,
                        vect_20_i, vect_21_i, vect_22_i, vect_23_i, vect_24_i,
                        vect_25_i, vect_26_i, vect_27_i, vect_28_i, vect_29_i,
                        vect_30_i, vect_31_i, vect_32_i, vect_33_i, vect_34_i,
                        vect_35_i, vect_36_i, vect_37_i, vect_38_i, vect_39_i,
                                    vect_40_i,

        output  reg [ 5:0]      mc_block,
        output  reg [ 5:0]  bx_init_mc, bx_end_mc, by_init_mc, by_end_mc,
        output  reg             mc_last_block,
        output  reg             mc_block_assign_done

);

reg     [6:0]   ivect_x, ivect_y;
reg     [4:0]   x_size, y_size;
reg                 mc_block_assign_start_d, mc_block_assign_start_d2;
reg     [40:0]  mc_next_block_inverted;
```

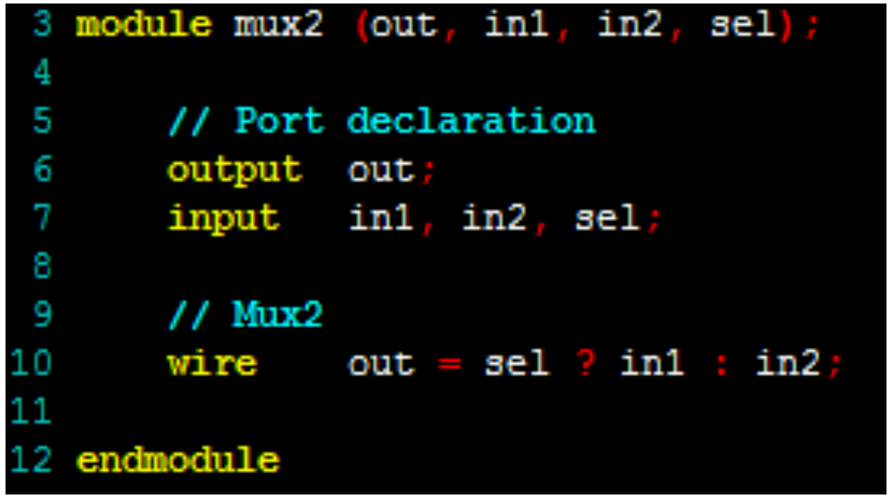
Port list declaration style

Modules: Definitions

- Example: module definitions
  - Variable names
    - All undeclared variables are wires and are one bit wide.
    - Good Practice: Declare all variables!!



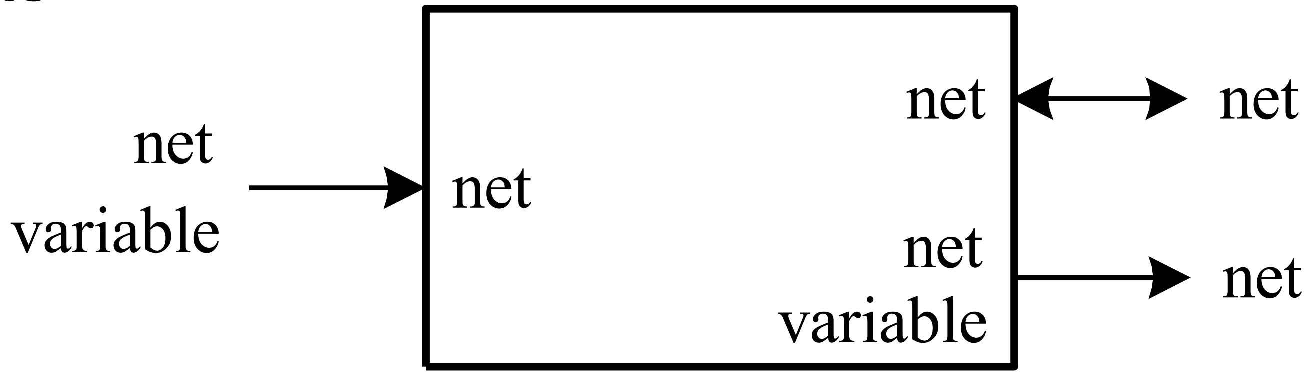
```
module mux2 (out, in1, in2, sel);

    // Port declaration
    output  out;
    input   in1, in2, sel;

    // Variable declaration
    reg     out;

    // Mux2
    always @(in1 or in2 or sel) begin
        if (sel)     out = in1;
        else         out = in2;
    end

endmodule
```



```
module mux2 (out, in1, in2, sel);

    // Port declaration
    output  out;
    input   in1, in2, sel;

    // Mux2
    wire    out = sel ? in1 : in2;

endmodule
```

Modules: Definitions

- Port declaration: Types of ports
  - input
  - output
  - inout



```
module adder(x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output reg [3:0] sum;
output reg c_out;
```

Modules: Parameters

- Types of Parameters
  - Module parameters
    - `parameter`
    - `localparam`
  - Specify parameters

```
parameter SIZE = 7;
parameter WIDTH_BUSA = 24, WIDTH_BUSB = 8;
parameter signed [3:0] mux_selector = 4’b0;
```

Modules: Parameters

- Options for constant specification
  - `define compiler directive
    ```
    `define BUS_WIDTH  8
    ```
    defined at outside of module (applied to entire code)

  - parameter
    ```
    parameter BUS_WIDTH = 8;
    ```
    cannot defined at outside of module

  - localparam
    ```
    localparam BUS_WIDTH = 8;
    ```
    cannot defined at outside of module

Modules: Parameters

- Parameter ports

```
module module_name
#(parameter SIZE = 7,
  parameter WIDTH_BUSA = 24, WIDTH_BUSB = 8,
  parameter signed [3:0] mux_selector = 4’b0
) //parameter ports
(port list or port list declarations)
...
endmodule
```

Modules: Module Instantiation

- Syntax
```
module_name [#(parameters)] instance_name ([ports]);
```

- Port Connection Rules
  - Named association
    ```
    (.port_id1(port_expr1),..., .port_idn(port_exprn))
    ```
  - Positional association
    ```
    (port_expr1, ..., port_exprn)
    ```

Modules: Module Instantiation

- Port connection rules
  - Named association

```
shift_reg shift_reg_1(     //module name and instance of the module
.clk          (clk_50),    //the pin clk is connected to the wire clk_50
.reset_n      (reset_n),   //the "." denotes the pin
.data_ena     (data_ena),  //the value in the parens is the wire
.serial_data  (serial_data),
.parallel_data (shift_reg_out));  //wires and pins do not have to match
[show with arrows and such the various parts]
```

  - Positional association

```
shift_reg shift_reg_1(clk_50,reset_n,data_ena,serial_data,shift_reg_out);
```

How do you know its connected correctly?
What if the module had 50 pins on it?
What if you wanted to add wire in the middle of the list?

Do not use Positional Association!
It saves time once, and costs you dearly afterwords

Modules: Module Instantiation

- Example: Port Connection

```
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// -- half adder body-- //
// instantiate primitive gates
   xor xor1 (s, x, y);
   and and1 (c, x, y);
endmodule

module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1,c1,c2;  // outputs of both half adders
// -- full adder body-- //
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));
   or (cout, c1, c2);
endmodule
```

Can only be connected by using positional association

Instance name is optional.

Connecting by using positional association

Connecting by using named association

Instance name is necessary.

Modules: Parameter Values

- Parameterized modules

```
module adder_nbit(x, y, c_in, sum, c_out);
parameter N = 4;   // set default value
input [N-1:0] x, y;
input c_in;
output [N-1:0] sum;
output c_out;

assign {c_out, sum} = x + y + c_in;
endmodule
```

Modules: Parameter Values

- Overriding parameters: Using `defparam` statement

```
module counter_nbits (clock, clear, qout);
parameter N = 4;  // define counter size
always @(negedge clock or posedge clear)
begin              // qout <= (qout + 1) % 2^n
      if (clear) qout <= {N{1'b0}};
      else         qout <= (qout + 1) ;
end
```

```
// define top level module
…
output [3:0] qout4b;
output [7:0] qout8b;
// instantiate two counter modules
defparam cnt_4b.N = 4, cnt_8b.N = 8;
counter_nbits cnt_4b (clock, clear, qout4b);
counter_nbits cnt_8b (clock, clear, qout8b);
```

Modules: Parameter Values

- Overriding parameters: Using module instance parameter value as signment - one parameter

```
module counter_nbits (clock, clear, qout);
parameter N = 4;  // define counter size
…
always @(negedge clock or posedge clear)
begin                   // qout <= (qout + 1) % 2^n;
      if (clear) qout <= {N{1'b0}};
      else          qout <= (qout + 1) ;
end
```

```
// define top level module
…
output [3:0] qout4b;
output [7:0] qout8b;
// instantiate two counter modules
counter_nbits #(4) cnt_4b (clock, clear, qout4b);
counter_nbits #(8) cnt_8b (clock, clear, qout8b);
```

Modules: Parameter Values

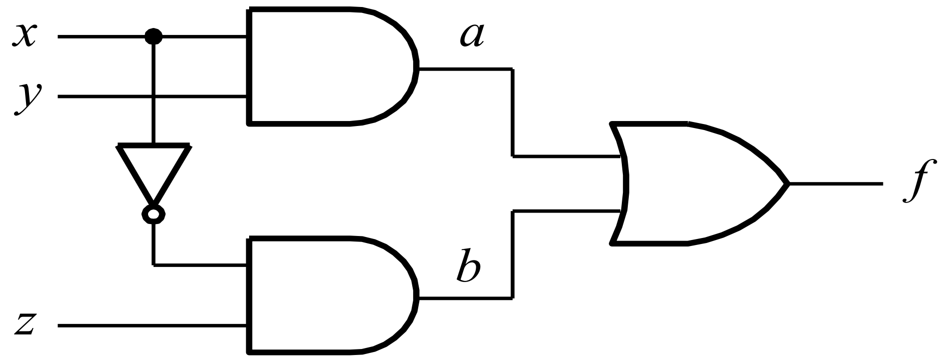
- Overriding parameters: Using module instance parameter value assignment - two parameters

```
module hazard_static (x, y, z, f);
parameter delay1 = 2, delay2 = 5;
…
    and  #delay2 a1 (b, x, y);
    not  #delay1  n1 (a, x);
    and  #delay2 a2 (c, a, z);
    or    #delay2 o2 (f, b, c);
endmodule
```

```
// define top level module
module …
…
hazard_static #(4, 8) example (x, y, z, f);
```

```
hazard_static #(.delay2(4), .delay1(6))
example (x, y, z, f);
```

- parameter value assignment by name --- minimize the chance of error!



Outline

- Module
- Generate statements
  - Generate-loop statements
  - Generate-conditional statements
- Module modeling styles
- Simulation

Generate-loop Statements

- generate block structures
  - Keywords: generate and endgenerate

```
// convert Gray code into binary code
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
genvar i;
generate for (i = 0; i < SIZE; i = i + 1)
begin: bit
      assign bin[i] = ^gray[SIZE-1:i];
end endgenerate
```

Generate-loop Statements

- generate loop construct

```
// convert Gray code into binary code
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
reg    [SIZE-1:0] bin;

genvar i;
generate for (i = 0; i < SIZE; i = i + 1)
begin:bit
     always @(*)
           bin[i] = ^gray[SIZE - 1: i];
end endgenerate
```

Generate-Conditional Statements

- Full adder

```
// define a full adder at dataflow level.
module full_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;
// Specify the function of a full adder.
   assign {c_out, sum} = x + y + c_in;
endmodule
```

Generate-Conditional Statements

- *n*-bit adder

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar  i;
wire  [N-2:0] c;      // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
   if (i == 0)             // specify LSB
      full_adder fa (___________________________);
   else if (i == N-1)      // specify MSB
      full_adder fa (___________________________);
   else                    // specify other bits
      full_adder fa (___________________________);
end endgenerate
```

Generate-Conditional Statements

- *n*-bit adder: An alternative

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar  i;
wire [N-2:0] c;               // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
   if (i == 0)                // specify LSB
      assign {___________} =  x[i] + y[i] + c_in;
   else if (i == N-1)         // specify MSB
      assign {___________} =  x[i] + y[i] + c[i-1];
   else                       // specify other bits
      assign {___________} =  x[i] + y[i] + c[i-1];
end endgenerate
```

Generate-Conditional Statements

- *n*-bit adder: Yet another alternative

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar i;
reg    [N-2:0] c;          // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
   if  (i == 0)            // specify LSB
      always @(*) {___________} =  x[i] + y[i] + c_in;
   else if (i == N-1)      // specify MSB
      always @(*) {___________} =  x[i] + y[i] + c[i-1];
   else                    // specify other bits
      always @(*) {___________} =  x[i] + y[i] + c[i-1];
end endgenerate
```

Outline

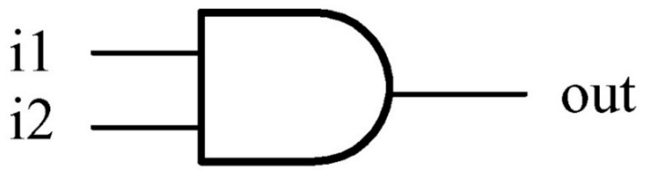
- Module
- Generate statements
- Module modeling styles
- Simulation

Module Modeling Styles

- **Structural style**
  - Gate level
  - Hierarchical design
- Dataflow style
- Behavioral or algorithmic style
- Mixed style

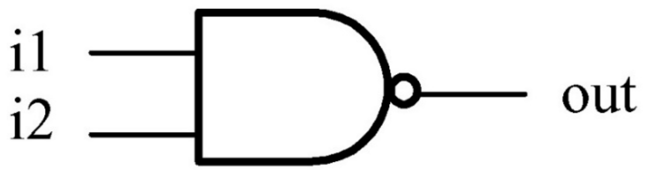
Structural Style: and/nand Gates

- Verilog primitive



and		i2			
		0	1	x	z
i1	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

(a) and gate

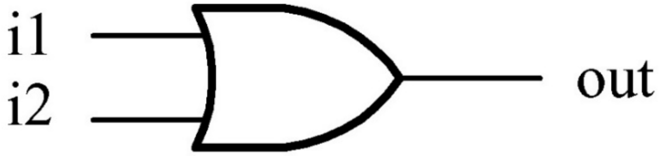


nand		i2			
		0	1	x	z
i1	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

(b) nand gate

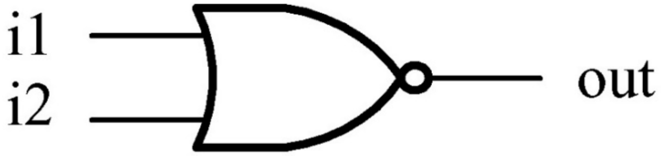
Structural Style: or/nor Gates

- Verilog primitive



or		i2			
		0	1	x	z
i1	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

(c) or gate

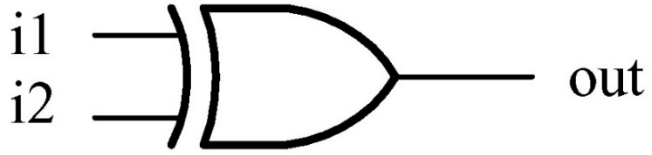


nor		i2			
		0	1	x	z
i1	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

(d) nor gate

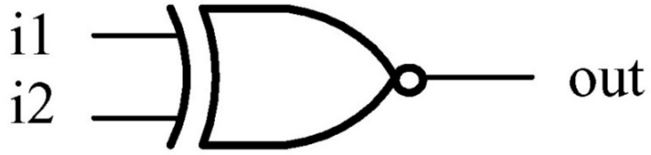
Structural Style: xor/xnor Gates

- Verilog primitive



xor		i2			
		0	1	x	z
i1	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

(e) xor gate

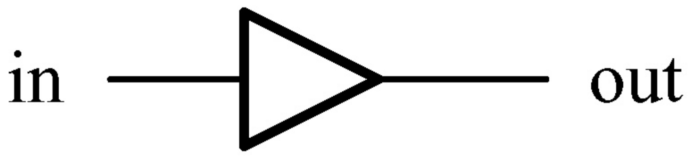


xnor		i2			
		0	1	x	z
i1	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

(f) xnor gate

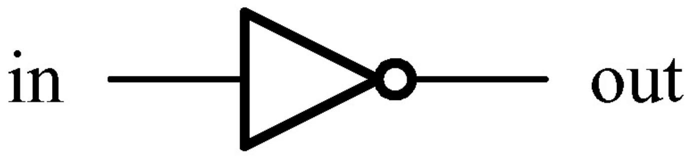
Structural Style: buf/not Gates

- Verilog primitive



in	out
0	0
1	1
x	x
z	x

(a) buffer



in	out
0	1
1	0
x	x
z	x

(b) not gate

Structural Style: Instantiation of Basic Gates

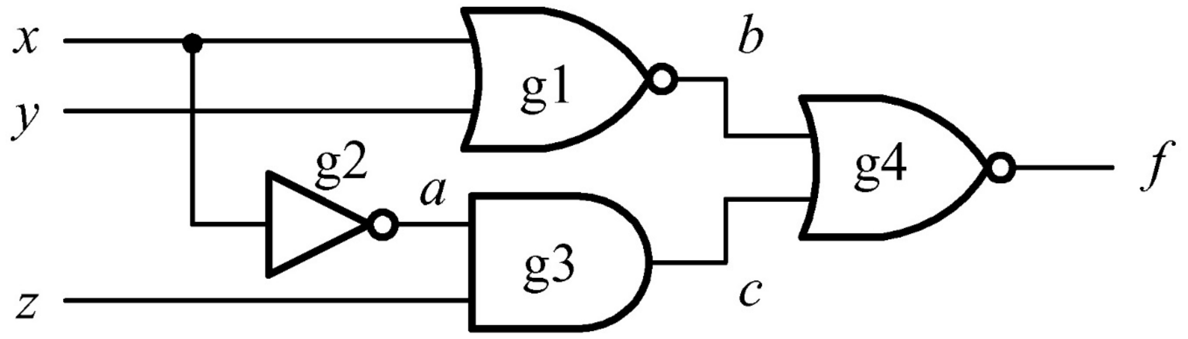
- To instantiate and/or gates

```
gatename [instance_name](output, input1, input2, ..., inputn);
```

  - instance_name is optional

```
module basic_gates (x, y, z, f) ;
input   x, y, z;
output f ;
wire a, b, c;
// Structural modeling
    nor  g1  (b, x, y);
    not  g2  (a, x);
    and g3  (c, a, z);
    nor g4  (f, b, c);
endmodule
```

Optional



Structural Style: Array of Instances

- Array instantiations may be a synthesizer dependent!
  - Suggestion: check this feature before using the synthesizer

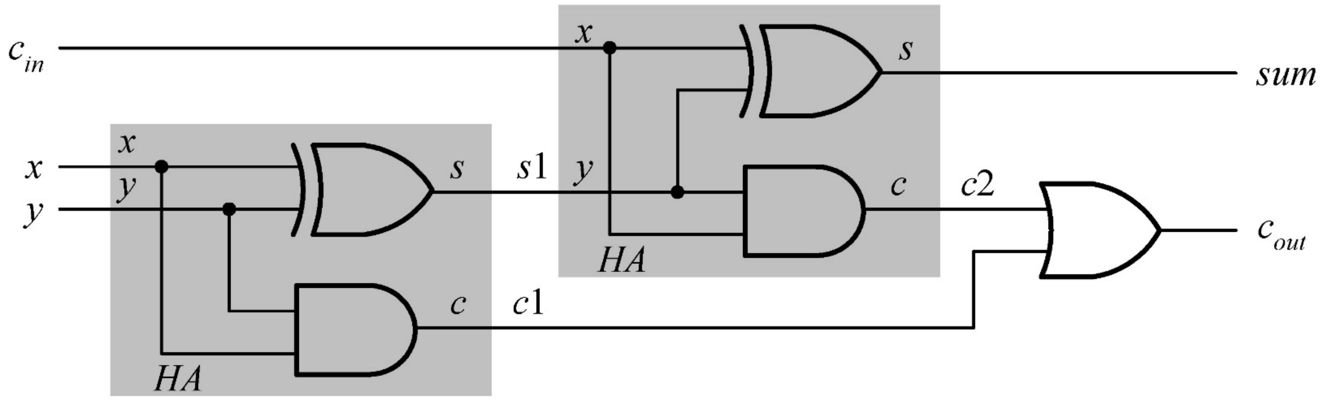
```
wire [3:0] out, in1, in2;
// basic array instantiations of nand gate.
nand n_gate[3:0] (out, in1, in2);

// this is equivalent to the following, this is better
nand n_gate0 (out[0], in1[0], in2[0]);
nand n_gate1 (out[1], in1[1], in2[1]);
nand n_gate2 (out[2], in1[2], in2[2]);
nand n_gate3 (out[3], in1[3], in2[3]);
```

Structural Style: An Example

```
module full_adder_structural(x, y,
c_in, sum, c_out);
// I/O port declarations
input   x, y, c_in;
output  sum, c_out;
wire    s1, c1, c2;
// Structural modeling of the 1-bit
full adder.
   xor  xor_s1(s1, x, y); // compute
sum.
   xor  xor_s2(sum, s1, c_in);
   and  and_c1(c1, x, y); // compute
carry out.
   and  and_c2(c2, s1, c_in);
   or   or_cout(c_out, c1, c2); // can
be xor
endmodule
```

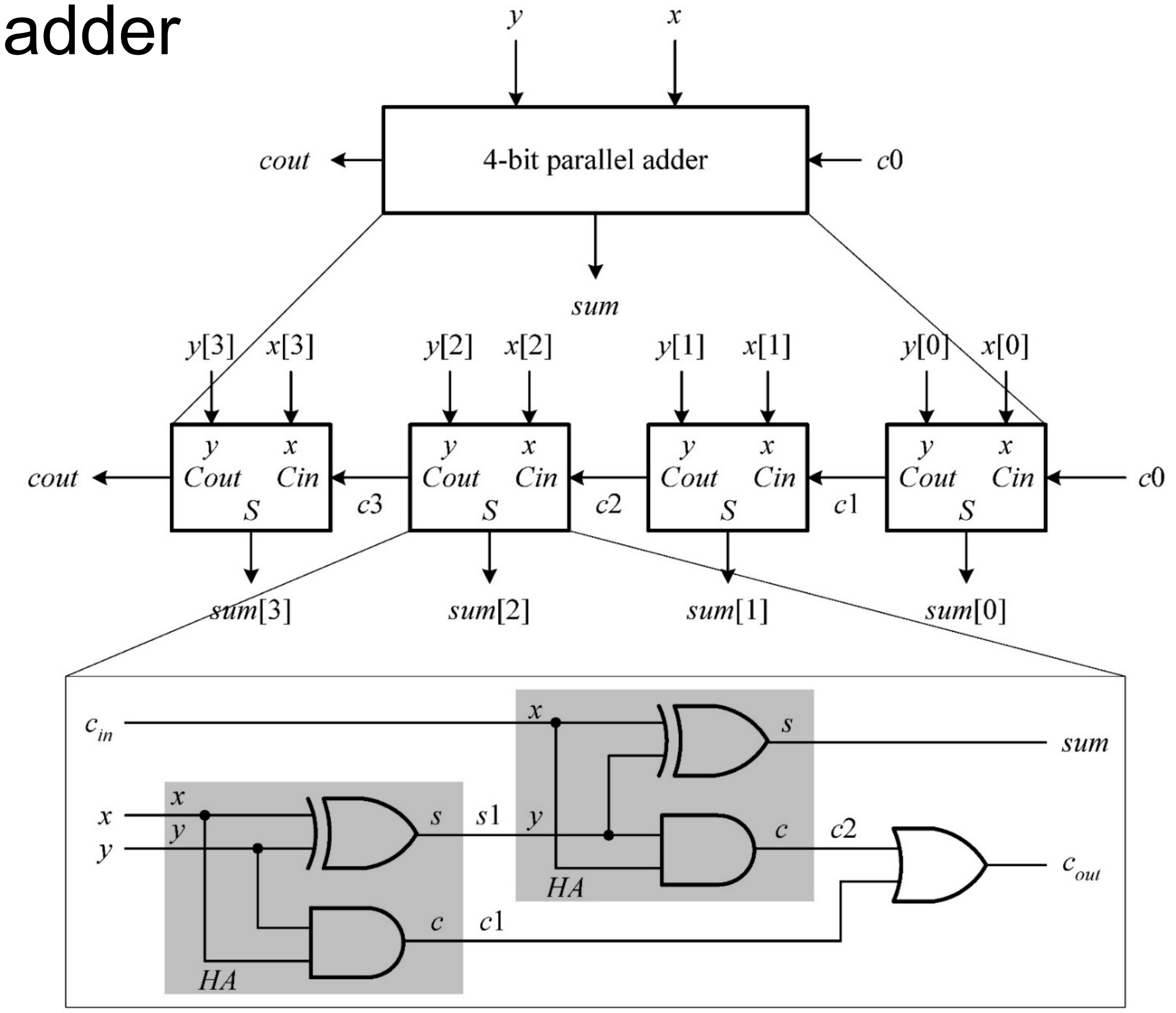
- 1-bit full adder

$$sum = (a \oplus b \oplus cin)$$
$$cout = (a \cdot b) + cin \cdot (a \oplus b)$$



Structural Style: Hierarchical Design

- Example: 4-bit adder



Structural Style: Hierarchical Design

- Example: 4-bit adder

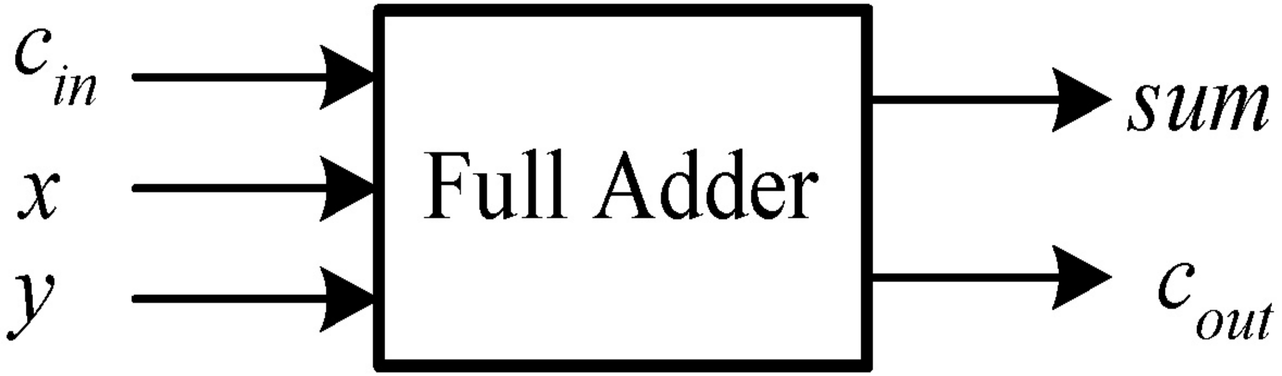
```
// gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output [3:0] sum;
output c_out;
wire c1, c2, c3; // intermediate carries
// four_bit adder body
// instantiate the full adder
full_adder fa_1 (x[0], y[0], c_in, sum[0], c1);
full_adder fa_2 (x[1], y[1], c1, sum[1], c2);
full_adder fa_3 (x[2], y[2], c2, sum[2], c3);
full_adder fa_4 (x[3], y[3], c3, sum[3], c_out);
endmodule
```

Module Modeling Styles

- Structural style
- **Dataflow style**
- Behavioral or algorithmic style
- Mixed style

Dataflow Style

```
module full_adder_dataflow(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
// specify the function of a full adder
assign {c_out, sum} = x + y + c_in;
endmodule
```



Module Modeling Styles

- Structural style
  - Gate level
- Dataflow style
- **Behavioral or algorithmic style**
- Mixed style

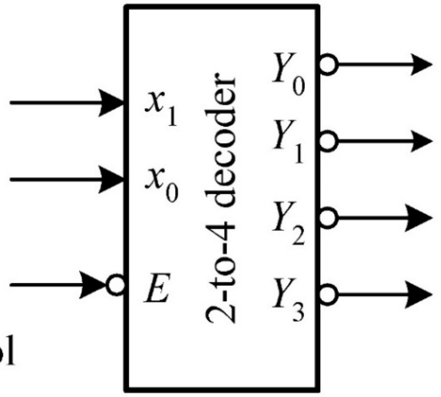
Behavioral Style

- Similar with a C-code
- Flip-Flop implementation

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out;  // need to be declared as reg types, output reg

// specify the function of a full adder
always @(x, y, c_in) // always @(*) or always@(x or y or c_in)
  {c_out, sum} = x + y + c_in;
endmodule
```

Behavioral Style: Example #1

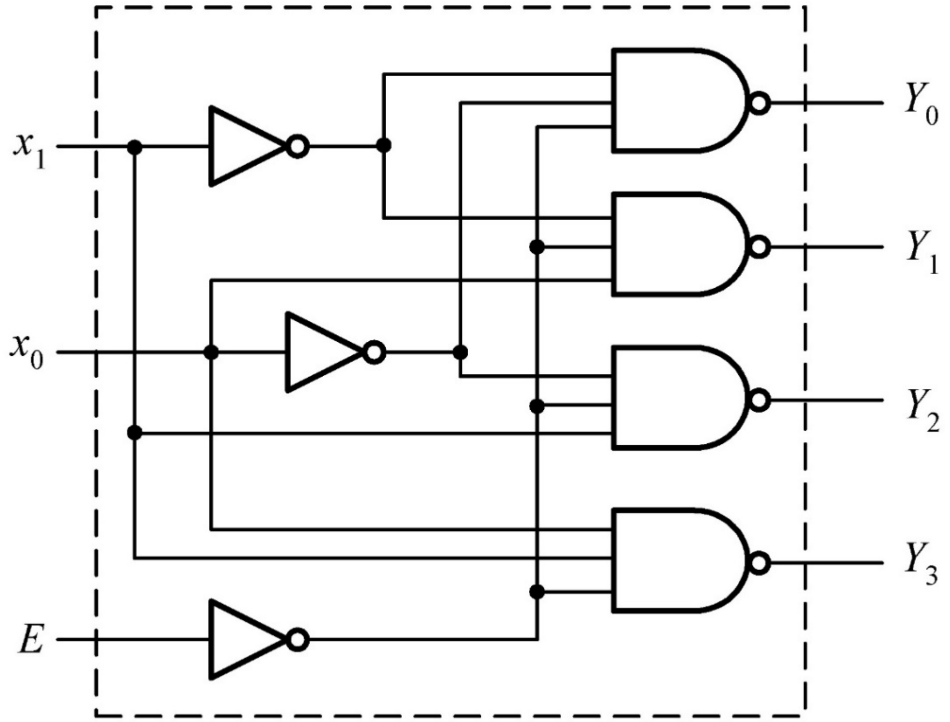
- 2-to-4 decoder



(a) Logic symbol

E	x_1	x_0	Y_3	Y_2	Y_1	Y_0
1	ϕ	ϕ	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

(b) Function table

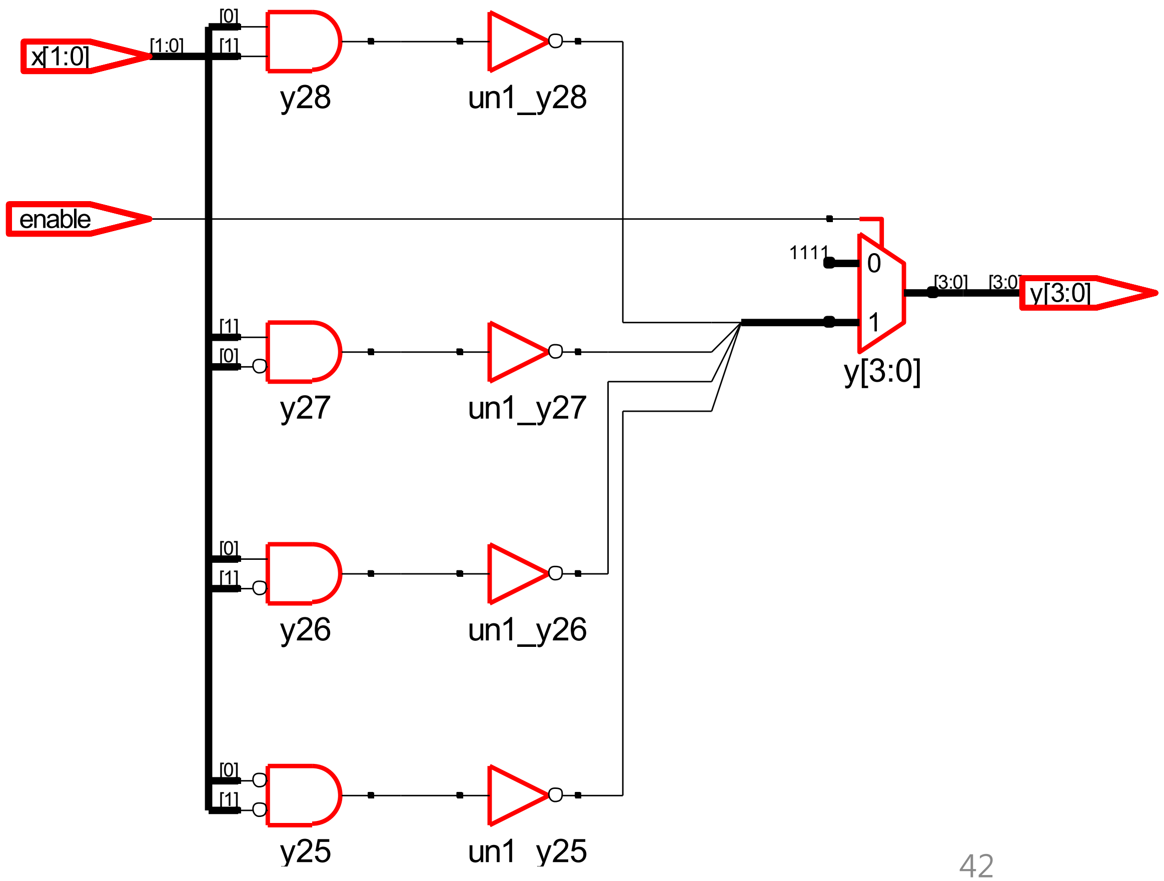


(c) Logic circuit

Behavioral Style: Example #1

- 2-to-4 decoder

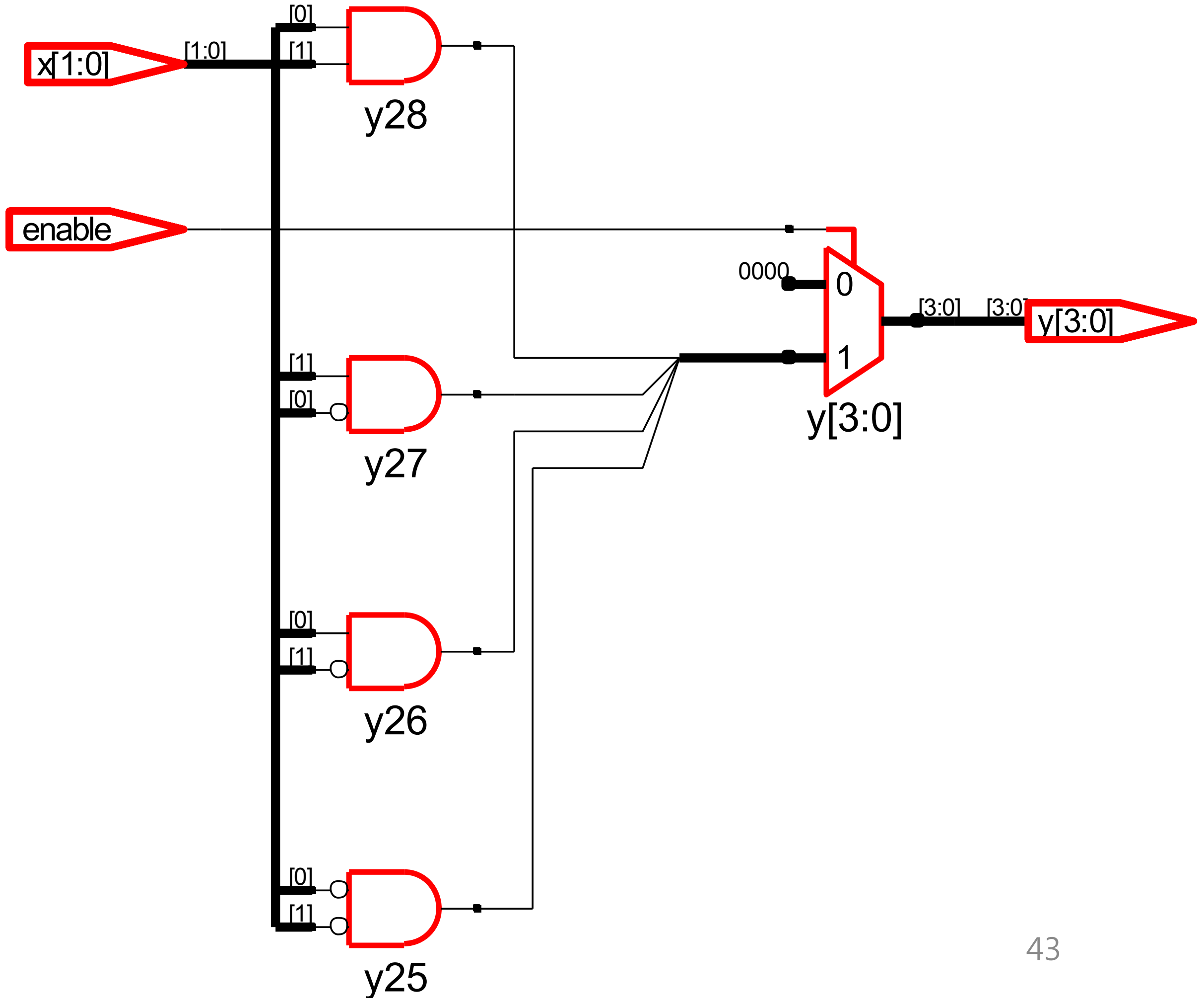
```
// a 2-to-4 decoder with active low output
always @(x or enable_n)
    if (enable_n)
       y = 4'b1111;
    else
        case (x)
            2'b00 : y = 4'b1110;
            2'b01 : y = 4'b1101;
            2'b10 : y = 4'b1011;
            2'b11 : y = 4'b0111;
       default : y = 4'b1111;
endcase
```



Behavioral Style: Example #1'

- 2-to-4 decoder with enable control

```
// a 2-to-4 decoder with active-
high output
always @ (x or enable)
   if (!enable) y = 4'b0000;
   else
       case (x)
            2'b00 : y = 4'b0001;
            2'b01 : y = 4'b0010;
            2'b10 : y = 4'b0100;
            2'b11 : y = 4'b1000;
         default : y = 4'b0000;
endcase
```

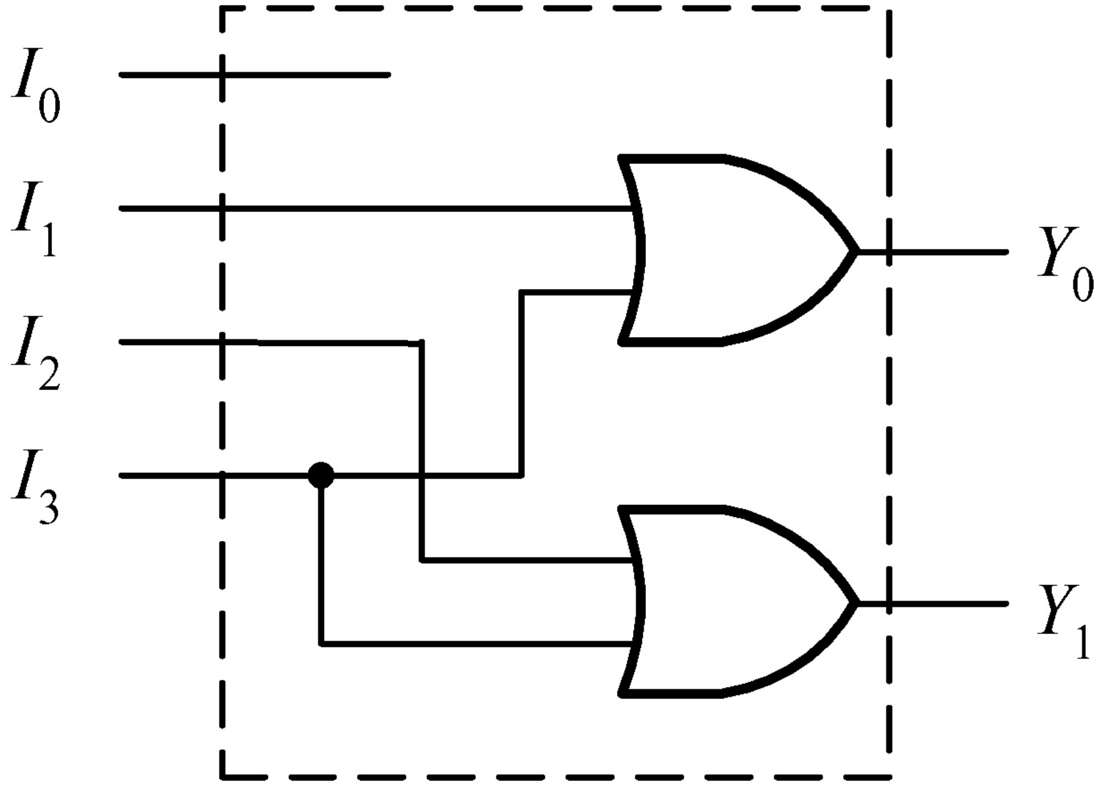


Behavioral Style: Example #2

- 4-to-2 encoder

I_3	I_2	I_1	I_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Function table

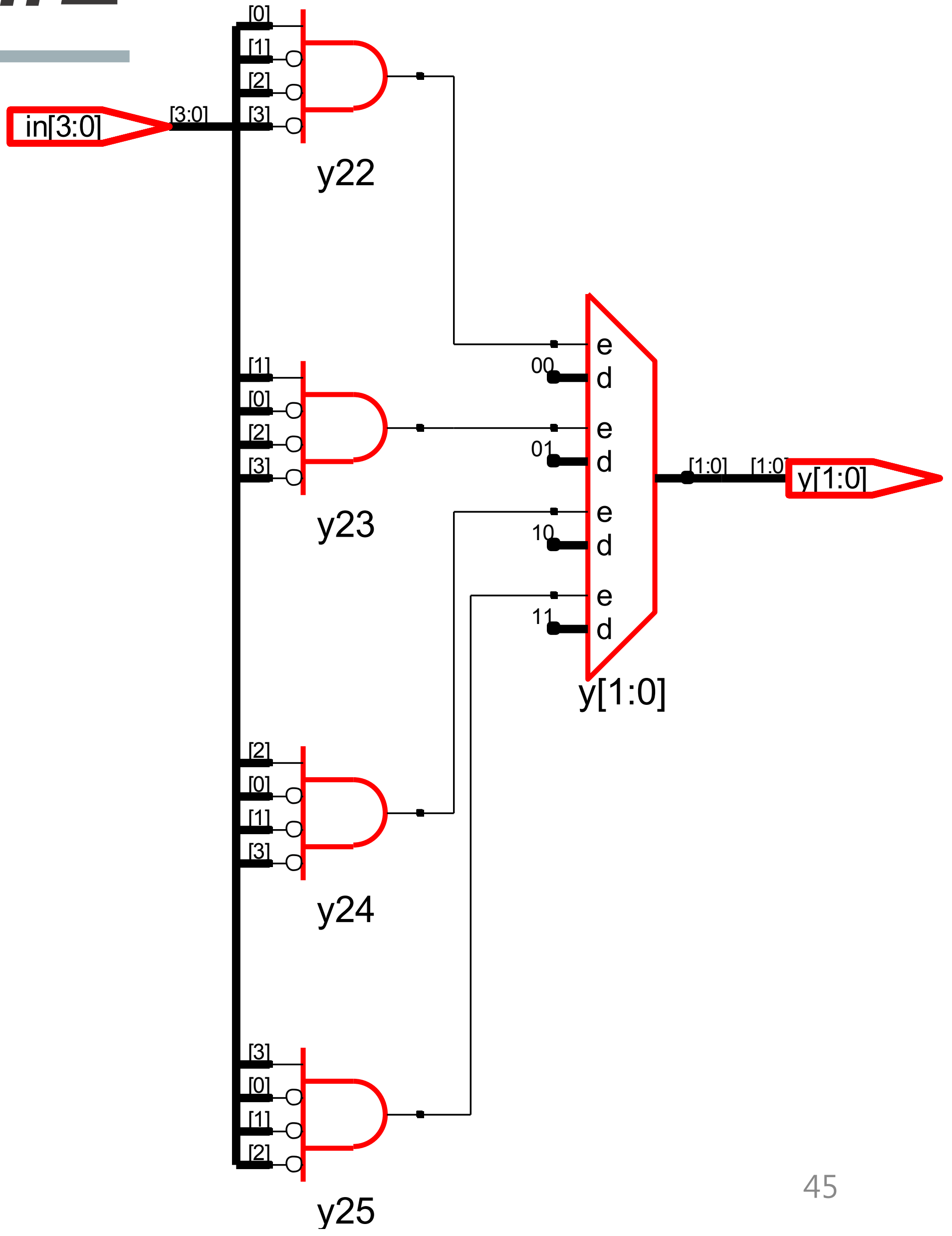


(b) Logic circuit

Behavioral Style: Example #2

- 4-to-2 encoder: `if ... else`

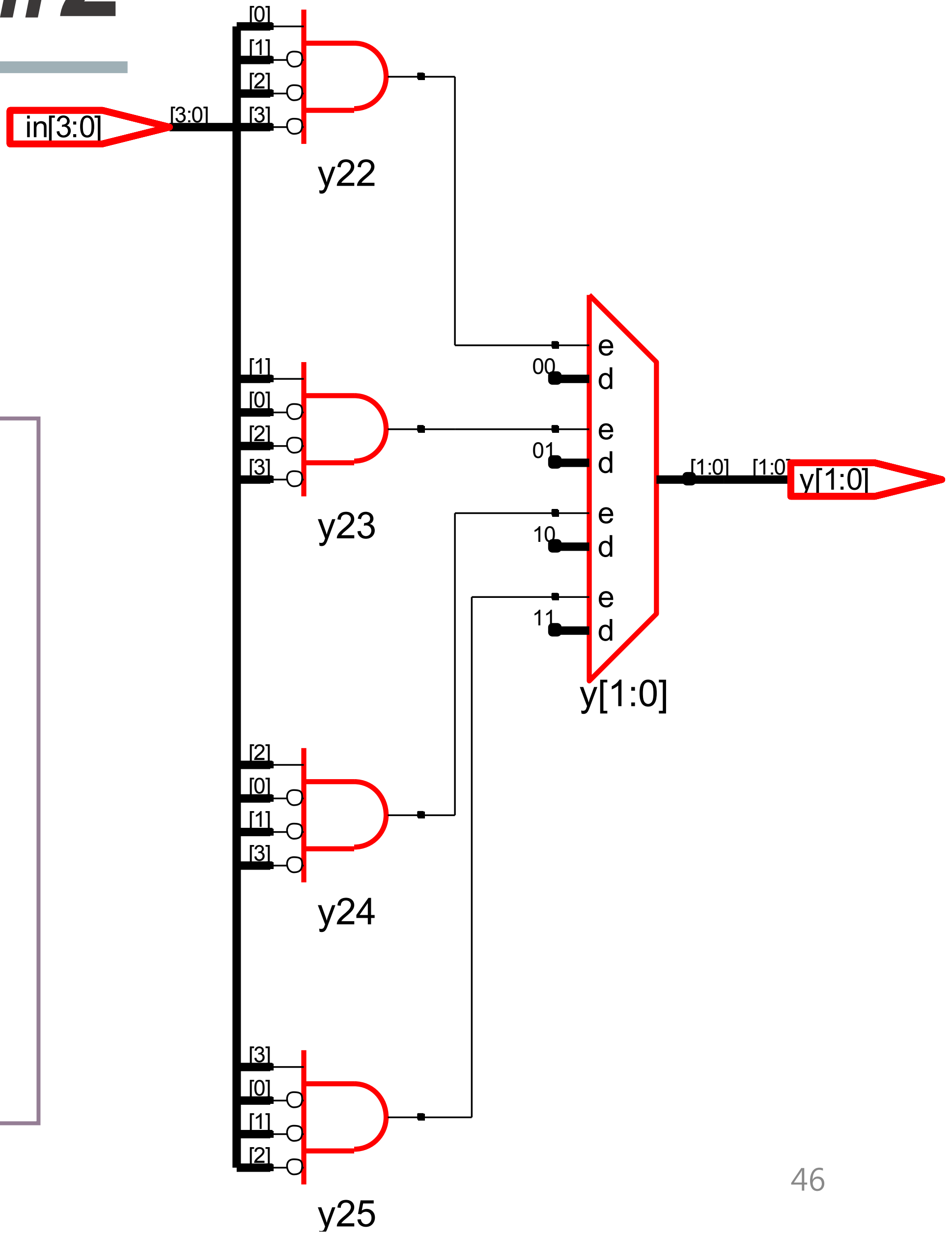
```
// a 4-to-2 encoder using if ... else
structure
always @ (in) begin
   if (in == 4'b0001) y = 0;
   else if (in == 4'b0010) y = 1;
   else if (in == 4'b0100) y = 2;
   else if (in == 4'b1000) y = 3;
   else y = 2'bx;
end
```



Behavioral Style: Example #2

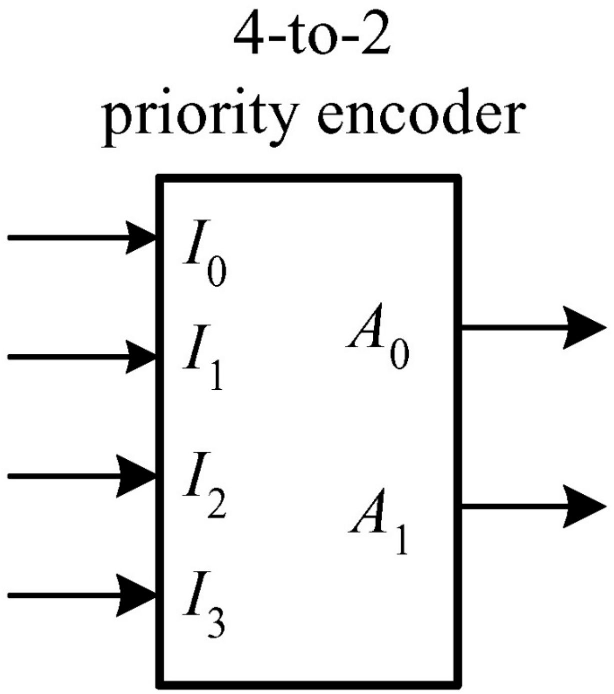
- 4-to-2 encoder: `case` statement

```
// a 4-to-2 encoder using case structure
always @ (in)
   case (in)
      4'b0001 : y = 0;
      4'b0010 : y = 1;
      4'b0100 : y = 2;
      4'b1000 : y = 3;
      default : y = 2'bx;
   endcase
```



Behavioral Style: Example #3

- 4-to-2 priority encoder



(a) Block diagram

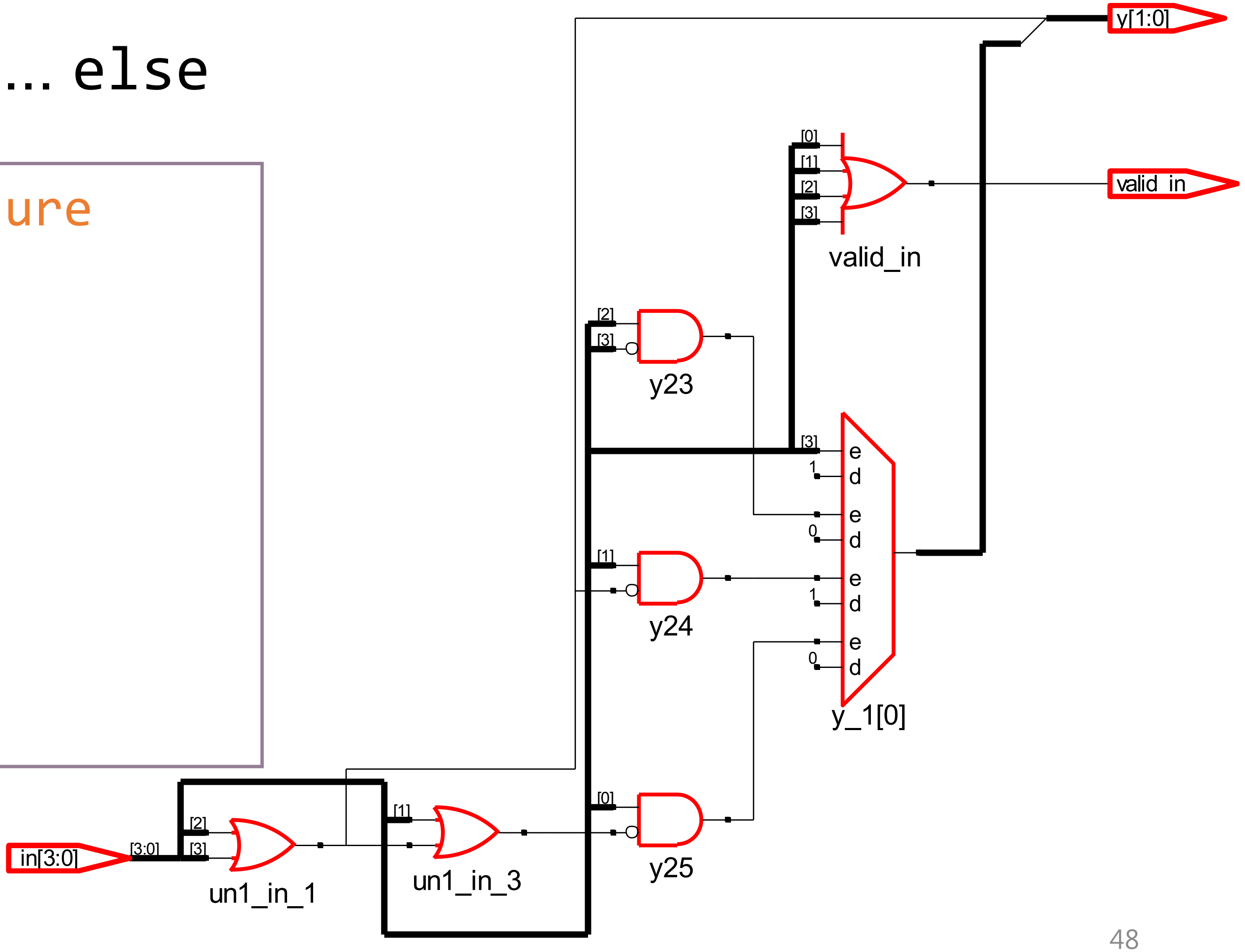
Input				Output	
I_3	I_2	I_1	I_0	A_1	A_0
0	0	0	1	0	0
0	0	1	ϕ	0	1
0	1	ϕ	ϕ	1	0
1	ϕ	ϕ	ϕ	1	1

(b) Function table

Behavioral Style: Example #3

- 4-to-2 priority encoder: `if ... else`

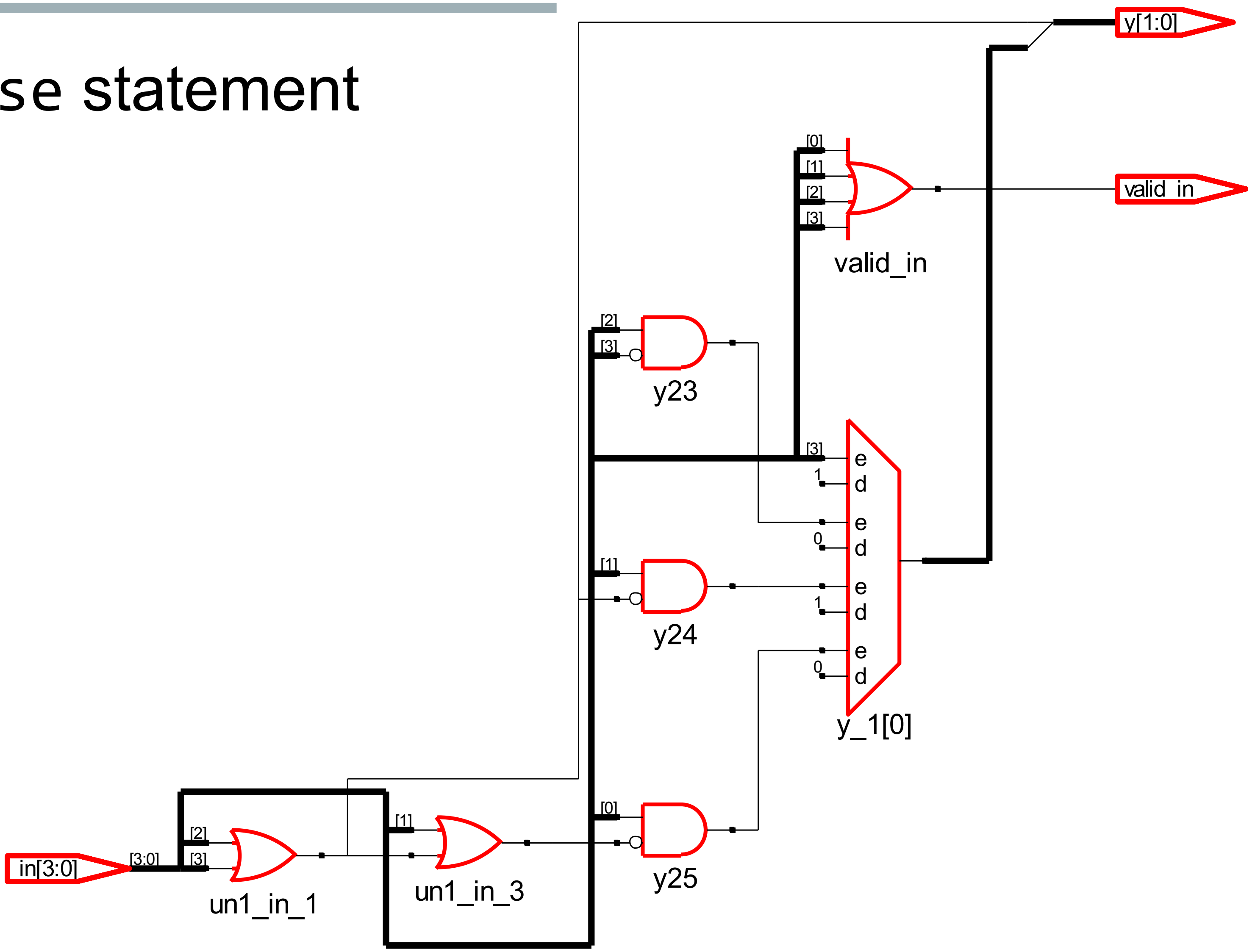
```
// using if ... else structure
assign valid_in = |in;
always @(in) begin
   if (in[3]) y = 3;
   else if (in[2]) y = 2;
   else if (in[1]) y = 1;
   else if (in[0]) y = 0;
   else y = 2'bx;
end
```



Behavioral Style: Example #3

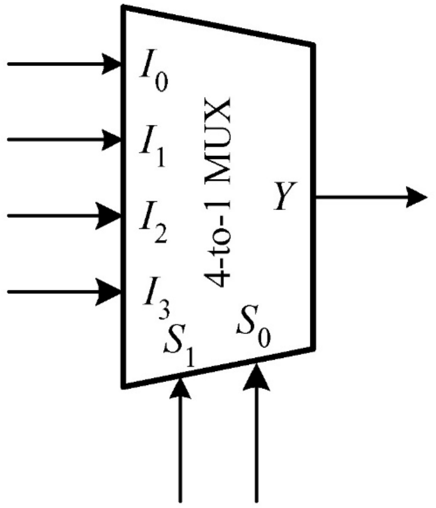
- 4-to-2 priority encoder: `case` statement

```
// using casex structure
assign valid_in = |in;
always @ (in)
   casex (in)
      4'b1xxx: y = 3;
      4'b01xx: y = 2;
      4'b001x: y = 1;
      4'b0001: y = 0;
      default: y = 2'bx;
  endcase
```



Behavioral Style: Example #4

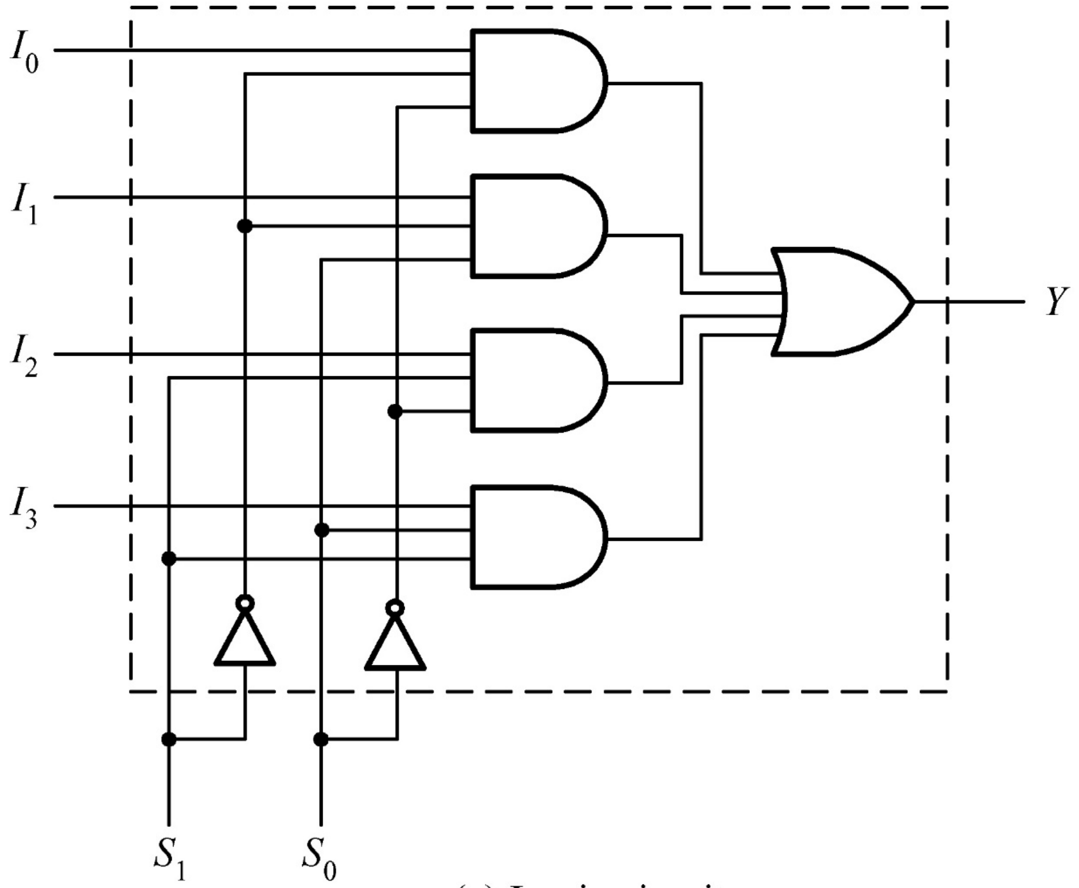
- 4-to-1 multiplexer



(a) Logic symbol

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

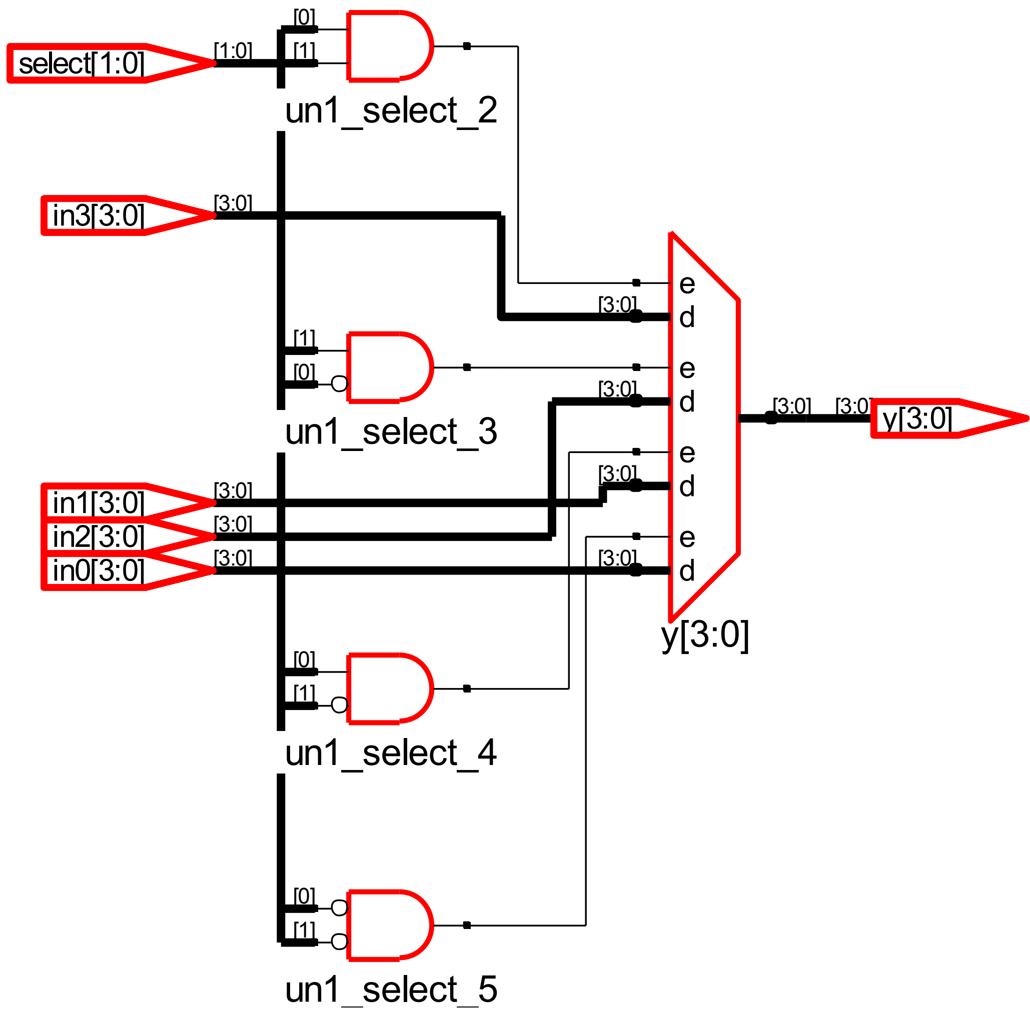


(c) Logic circuit

Behavioral Style: Example #4'

- *n*-bit 4-to-1 Multiplexer

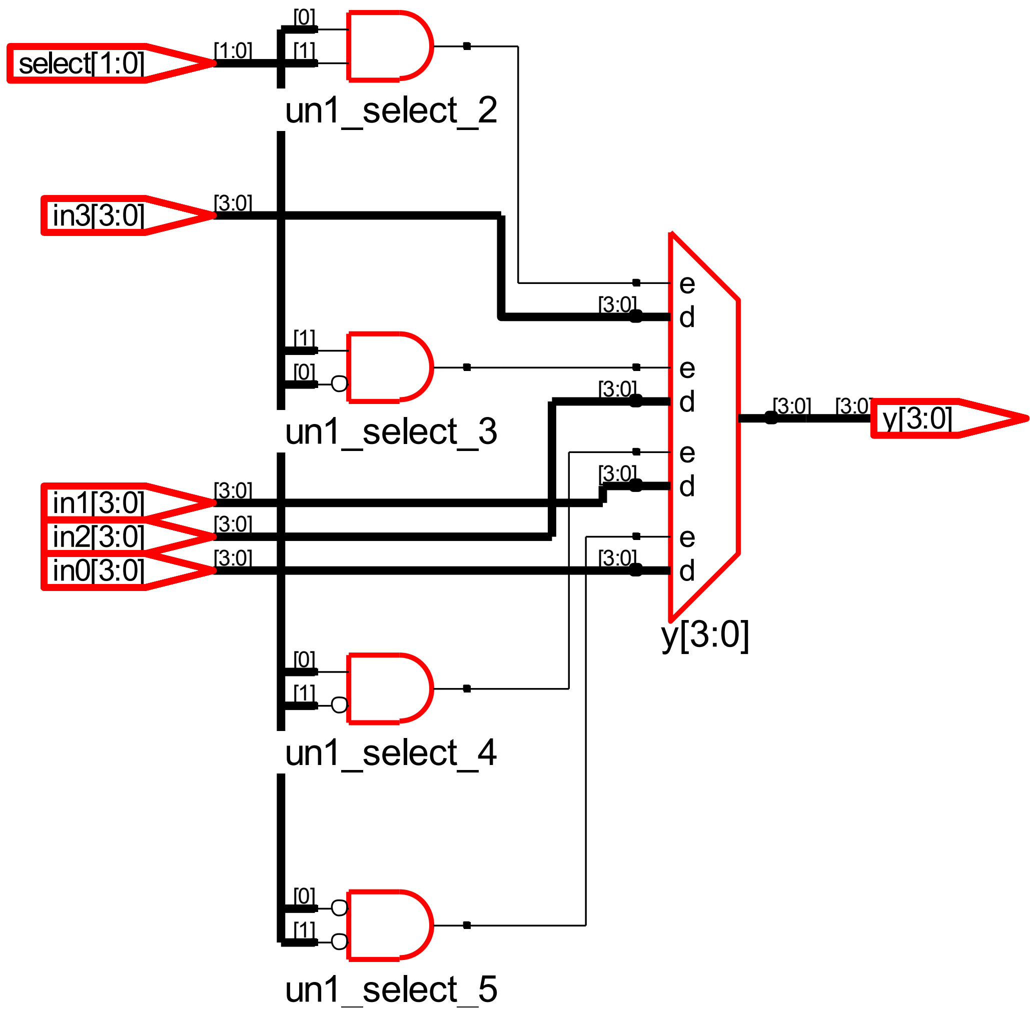
```
// an N-bit 4-to-1 multiplexer using
conditional operator
parameter N = 4;
input [1:0] select;
input [N-1:0] in3, in2, in1, in0;
output [N-1:0] y;
assign y = select[1] ?
           (select[0] ? in3 : in2) :
           (select[0] ? in1 : in0) ;
```



Behavioral Style: Example #4"

- *n*-bit 4-to-1 multiplexer with enable

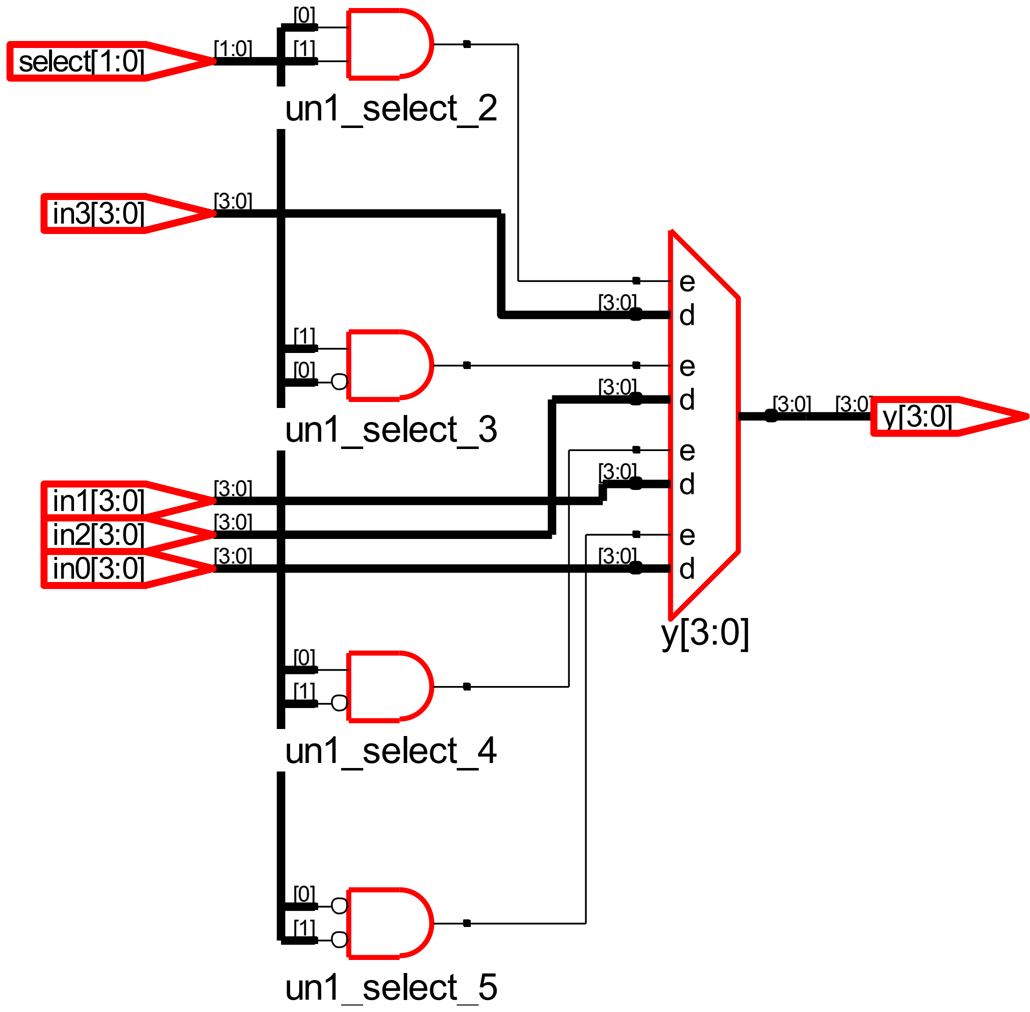
```
// an N-bit 4-to-1 multiplexer with enable
control
parameter N = 4;
input [1:0] select;
input enable;
input [N-1:0] in3, in2, in1, in0;
output reg [N-1:0] y;

always @(select or enable or in0 or in1 or in2
or in3)
     if (!enable) y = {N{1’b0}};
     else y = select[1] ?
                 (select[0] ? in3 : in2) :
                 (select[0] ? in1 : in0) ;
```



Behavioral Style: Example #4"

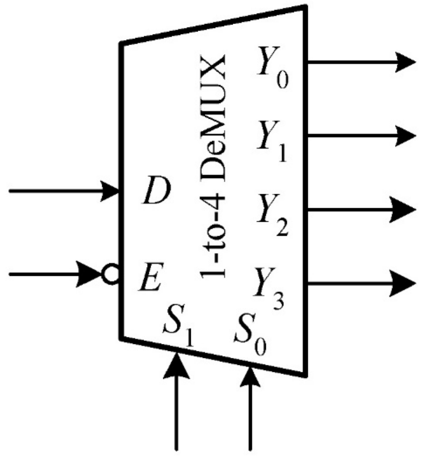
- *n*-bit 4-to-1 multiplexer with enable: An alternative using `case`

```
// an N-bit 4-to-1 multiplexer using case
parameter N = 8;
input [1:0] select;
input [N-1:0] in3, in2, in1, in0;
output reg [N-1:0] y;
always @(*)
      case (select)
            2'b11: y = in3 ;
            2'b10: y = in2 ;
            2'b01: y = in1 ;
            2'b00: y = in0 ;
            default:y = N{1'b0};
      endcase
```



Behavioral Style: Example #5

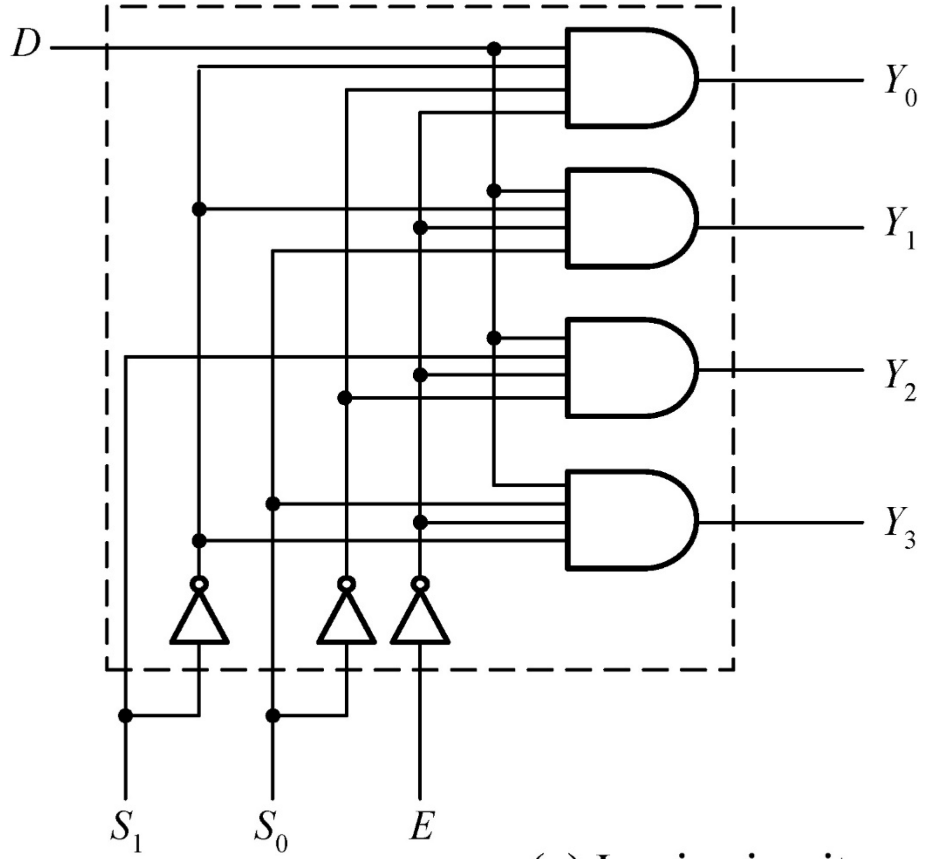
- 1-to-4 demultiplexer (DEMUX)



(a) Logic symbol

E	S_1	S_0	Y_3	Y_2	Y_1	Y_0
1	ϕ	ϕ	0	0	0	0
0	0	0	0	0	0	D
0	0	1	0	0	D	0
0	1	0	0	D	0	0
0	1	1	D	0	0	0

(b) Function table

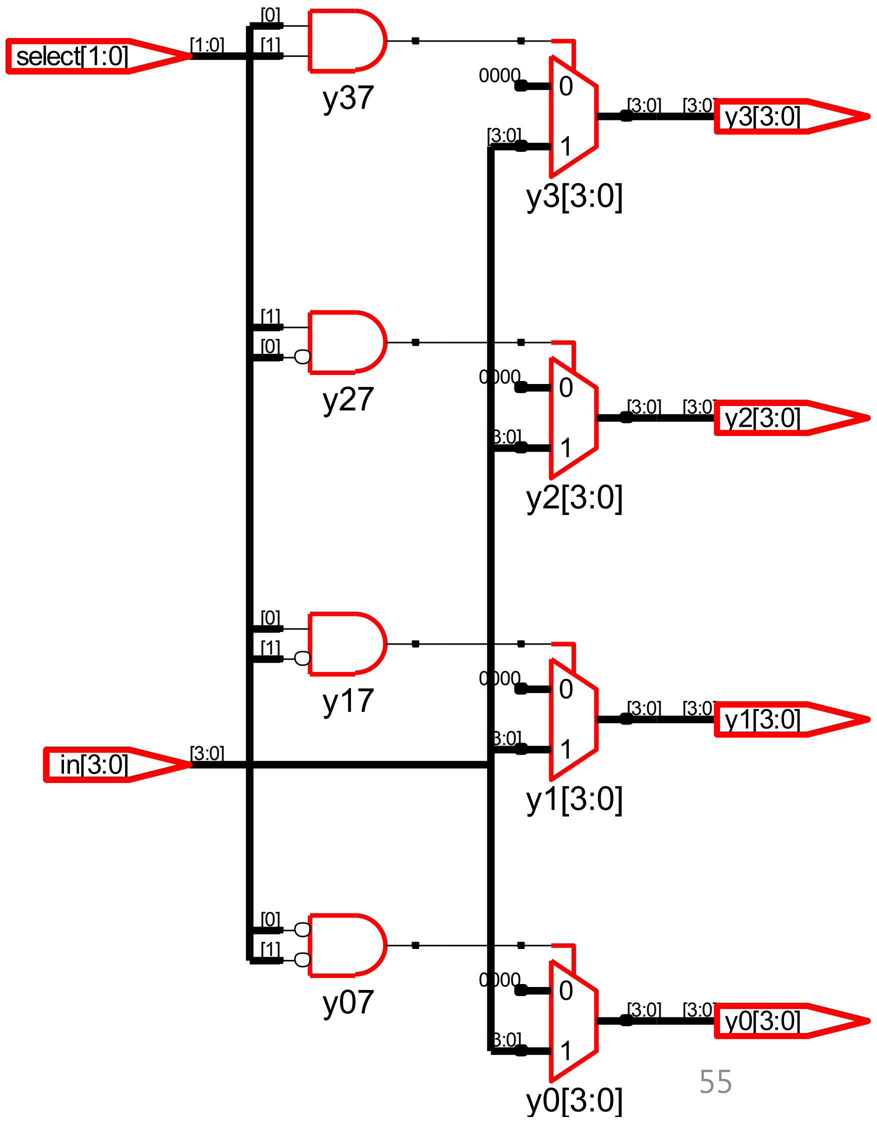


(c) Logic circuit

Behavioral Style: Example #5

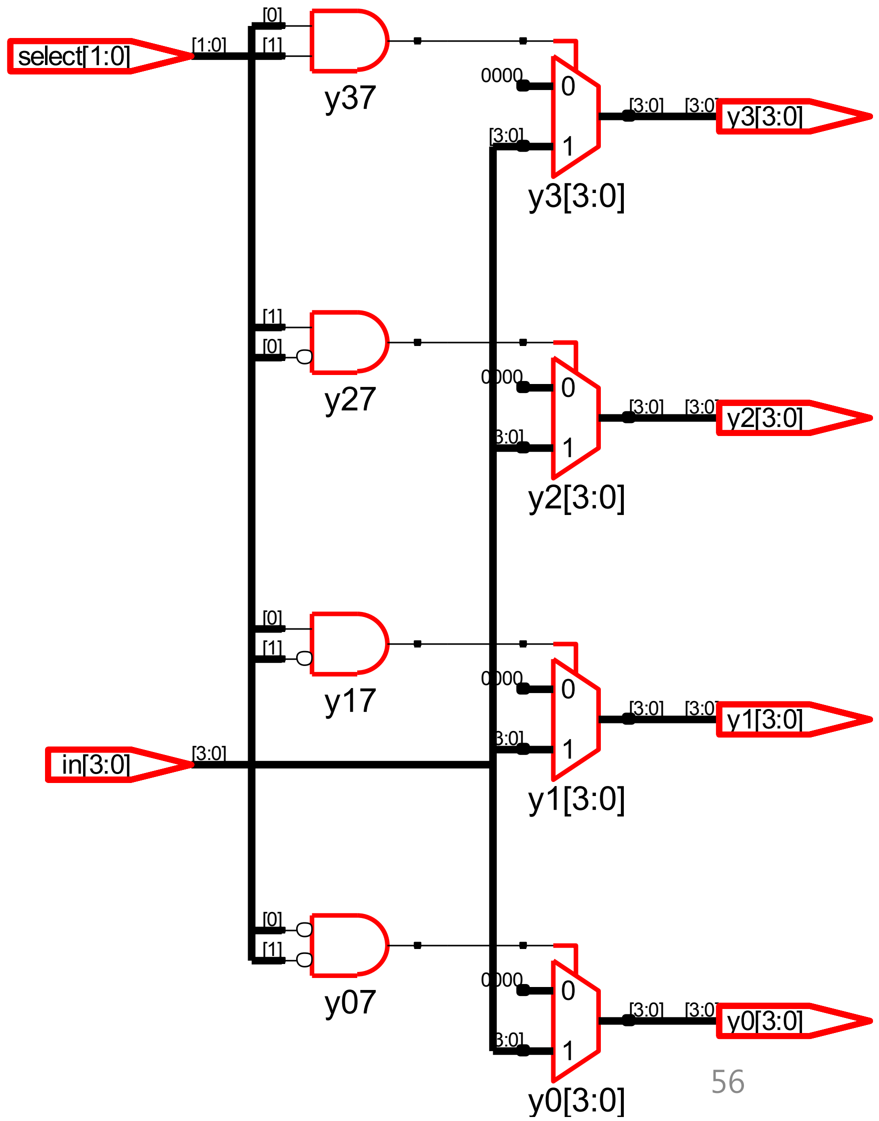
- 1-to-4 DEMUX: `if ... else`

```
// an N-bit 1-to-4 demultiplexer using if ...
else structure
parameter N = 4;  // default width
input      [1:0] select;
input      [N-1:0] in;
output reg [N-1:0] y3, y2, y1, y0;

always @(select or in) begin
   if (select == 3) y3 = in; else y3 = {N{1'b0}};
   if (select == 2) y2 = in; else y2 = {N{1'b0}};
   if (select == 1) y1 = in; else y1 = {N{1'b0}};
   if (select == 0) y0 = in; else y0 = {N{1'b0}};
end
```



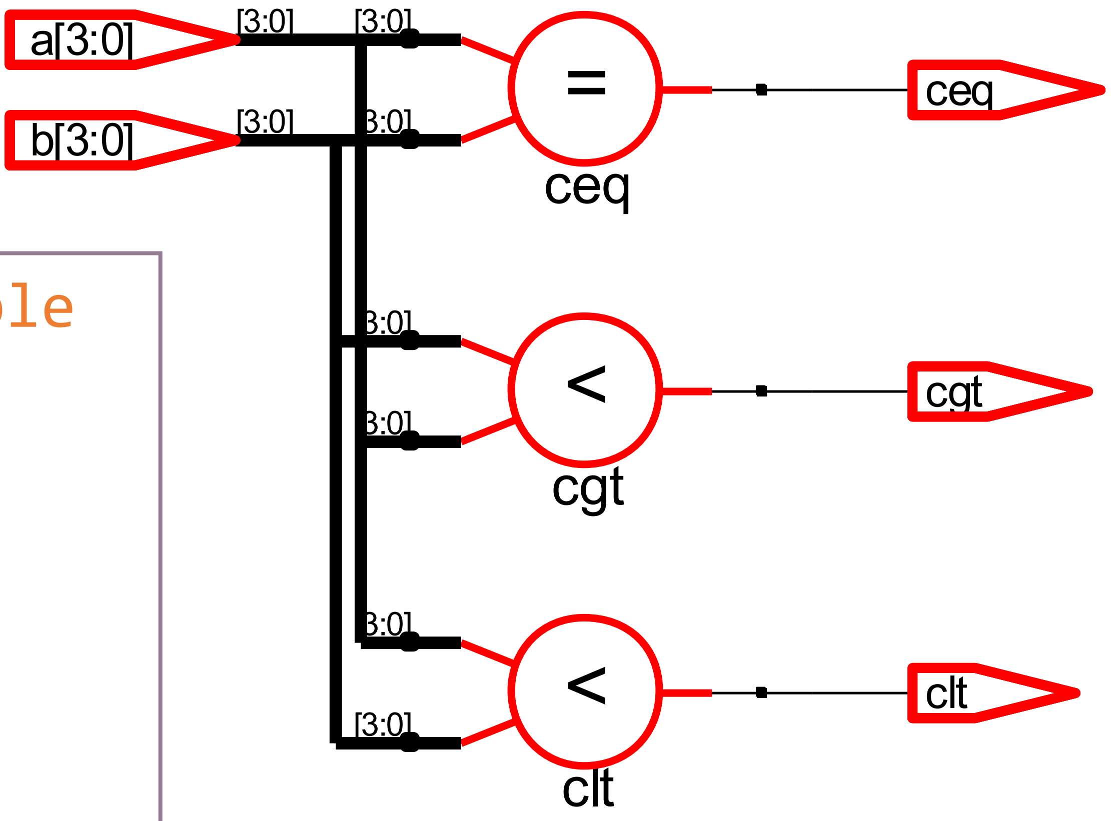
Behavioral Style: Example #5

```
// an N-bit 1-to-4 demultiplexer with enable control
parameter N = 4;  // Default width
…
output reg [N-1:0] y3, y2, y1, y0;
always @(select or in or enable) begin
   if (enable) begin
      if (select == 3) y3 = in; else y3 = {N{1'b0}};
      if (select == 2) y2 = in; else y2 = {N{1'b0}};
      if (select == 1) y1 = in; else y1 = {N{1'b0}};
      if (select == 0) y0 = in; else y0 = {N{1'b0}};
   end
   else begin
      y3 = {N{1'b0}};
      y2 = {N{1'b0}};
      y1 = {N{1'b0}};
      y0 = {N{1'b0}}; end
   end
```



Behavioral Style: Example #6

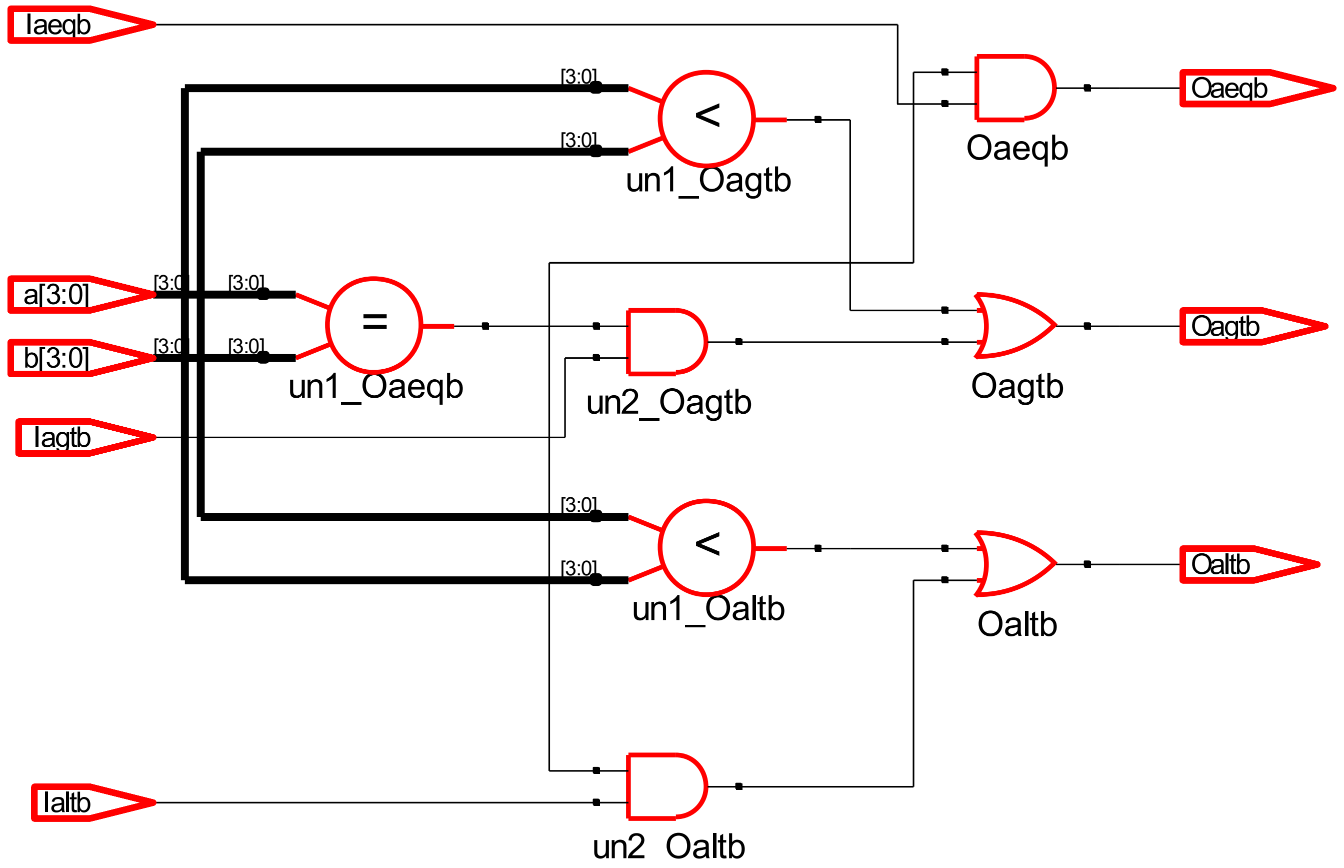
- Simple comparator



```
// an N-bit comparator module example
parameter N = 4;    // default size
input  [N-1:0] a, b;
output cgt, clt, ceq;

assign cgt = __________;
assign clt = __________;
assign ceq = __________;
```

Behavioral Style: Example #6'

- Cascadable comparator



Behavioral Style: Example #6'

- Cascadable comparator

```
parameter N = 4;
// I/O port declarations
input     Iagtb, Iaeqb, Ialtb;
input     [N-1:0] a, b;
output   Oagtb, Oaeqb, Oaltb;

// dataflow modeling using relation operators
assign Oaeqb =_______________________;  // =
assign Oagtb =_______________________;  // >
assign Oaltb =_______________________;  // <
```

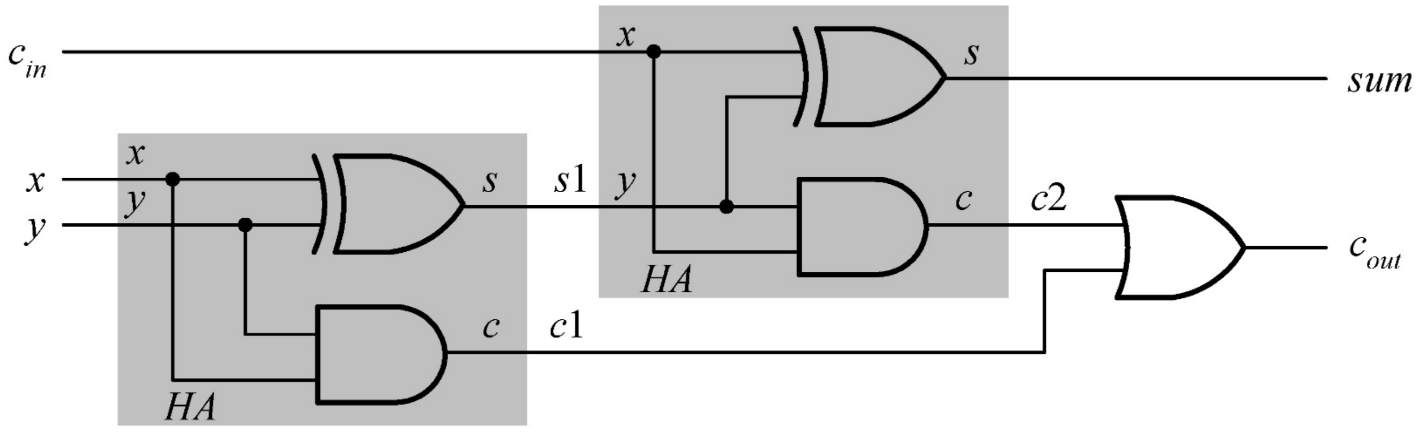
Module Modeling Styles

- Structural style
  - Gate level
- Dataflow style
- Behavioral or algorithmic style
- **Mixed style**

Mixed Style

- Example: Full adder

```
module full_adder_mixed_style(x, y, c_in, s, c_out); // I/O port declarations
input x, y, c_in;
output s, c_out;
reg c_out;
wire s1, c1, c2;
// structural modeling of HA 1
xor xor_ha1 (s1, x, y);
and and_ha1(c1, x, y);
// dataflow modeling of HA 2
assign s = c_in ^ s1;
assign c2 = c_in & s1;
// behavioral modeling of output OR gate
always @(c1, c2) // always @(*)
c_out = c1 | c2;
endmodule
```



Mixed Style

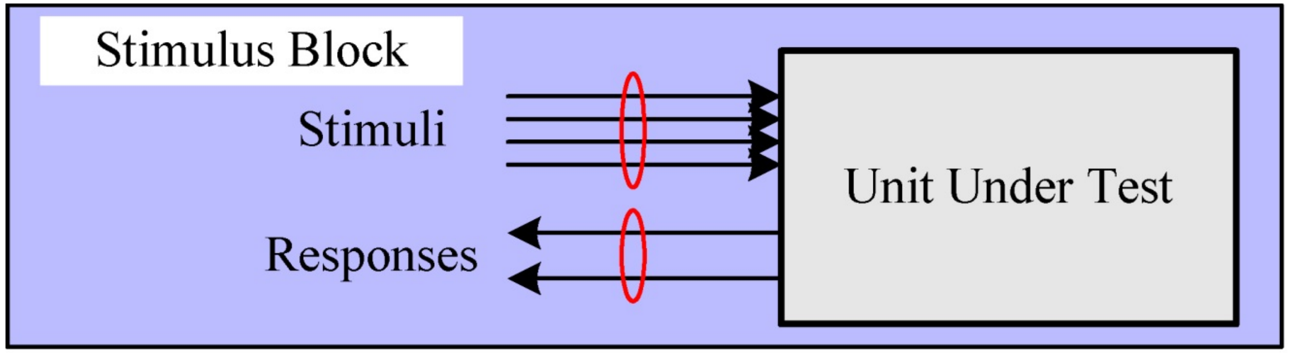
- Mixed style modeling
  - Using two or more different styles of description
  - Common mixed style: Register Transfer-Level (RTL)
    - RTL = synthesizable behavioral + dataflow constructs
    - Synthesized into gate-level design for chip design or FPGA emulation

Outline

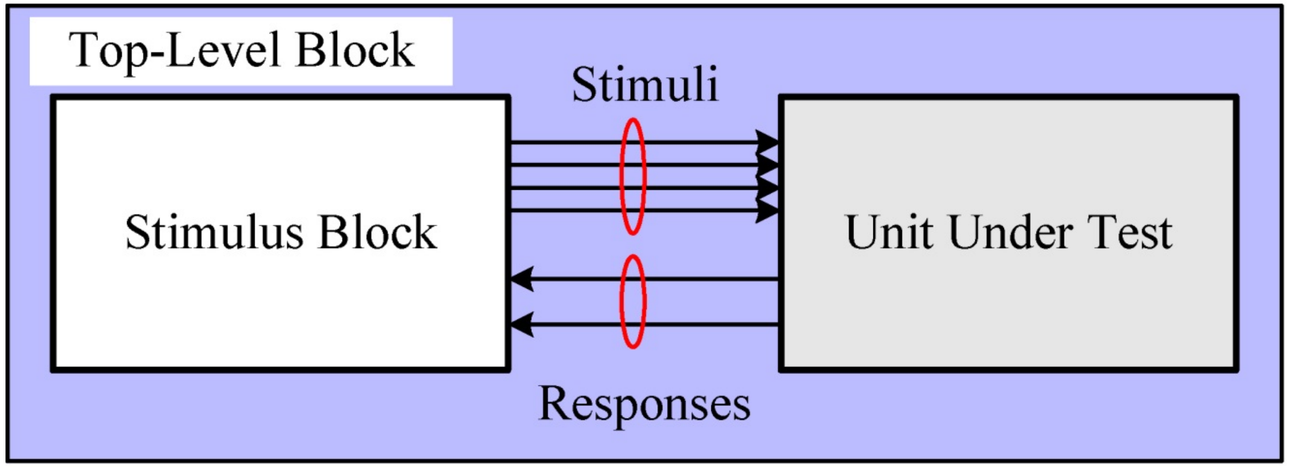
- Module
- Generate statements
- Module modeling styles
- Simulation

Simulation

- Basic simulation constructs
  - Stimulus Block = Test Bench



(a) Stimulus block at the top-level module.



(b) Stimulus block is considered as a separate module.

Simulation: Example

- Unit Under Test (UUT): 4-bit adder

```
// Gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input   [3:0] x, y;
input   c_in;
output [3:0] sum;
output c_out;
wire   C1,C2,C3;  // Intermediate carries
// -- four_bit adder body--
// Instantiate the full adder
   full_adder fa_1 (x[0],y[0],c_in,sum[0],C1);
   full_adder fa_2 (x[1],y[1],C1,sum[1],C2);
   full_adder fa_3 (x[2],y[2],C2,sum[2],C3);
   full_adder fa_4 (x[3],y[3],C3,sum[3],c_out);
endmodule
```

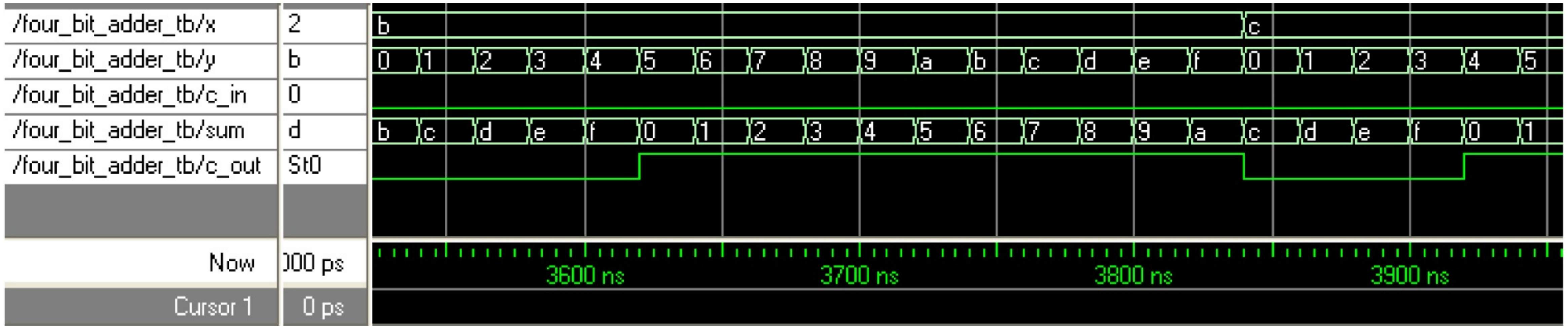
Simulation: Example

- Test bench of 4-bit adder

```
`timescale 1ns / 100ps  // time unit is in ns.
module four_bit_adder_tb;
// internal signals declarations:
reg [3:0] x;
reg [3:0] y;
reg c_in;
wire [3:0] sum;
wire c_out;
// Unit Under Test port map
four_bit_adder UUT (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
reg [7:0] i;

initial
   for (i = 0; i <= 255; i = i + 1) begin
        x[3:0] = i[7:4]; y[3:0] = i[3:0]; c_in =1'b0;
   #20 ;   end
```

Simulation: Example

- Simulation results



/four_bit_adder_tb/x
2
/four_bit_adder_tb/y
b
/four_bit_adder_tb/c_in
0
/four_bit_adder_tb/sum
d
/four_bit_adder_tb/c_out
St0
Now
000 ps
Cursor 1
0 ps
b c
0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5
b c d e f 0 1 2 3 4 5 6 7 8 9 a c d e f 0 1
3600 ns
3700 ns
3800 ns
3900 ns