硬體描述語言

Chapter 04

Synthesis of Combinational Logic

Ren-Der Chen (陳仁德)
Department of Computer Science and Information Engineering
National Changhua University of Education
E-mail: rdchen@cc.ncue.edu.tw
Fall, 2024

HDL-Based Synthesis

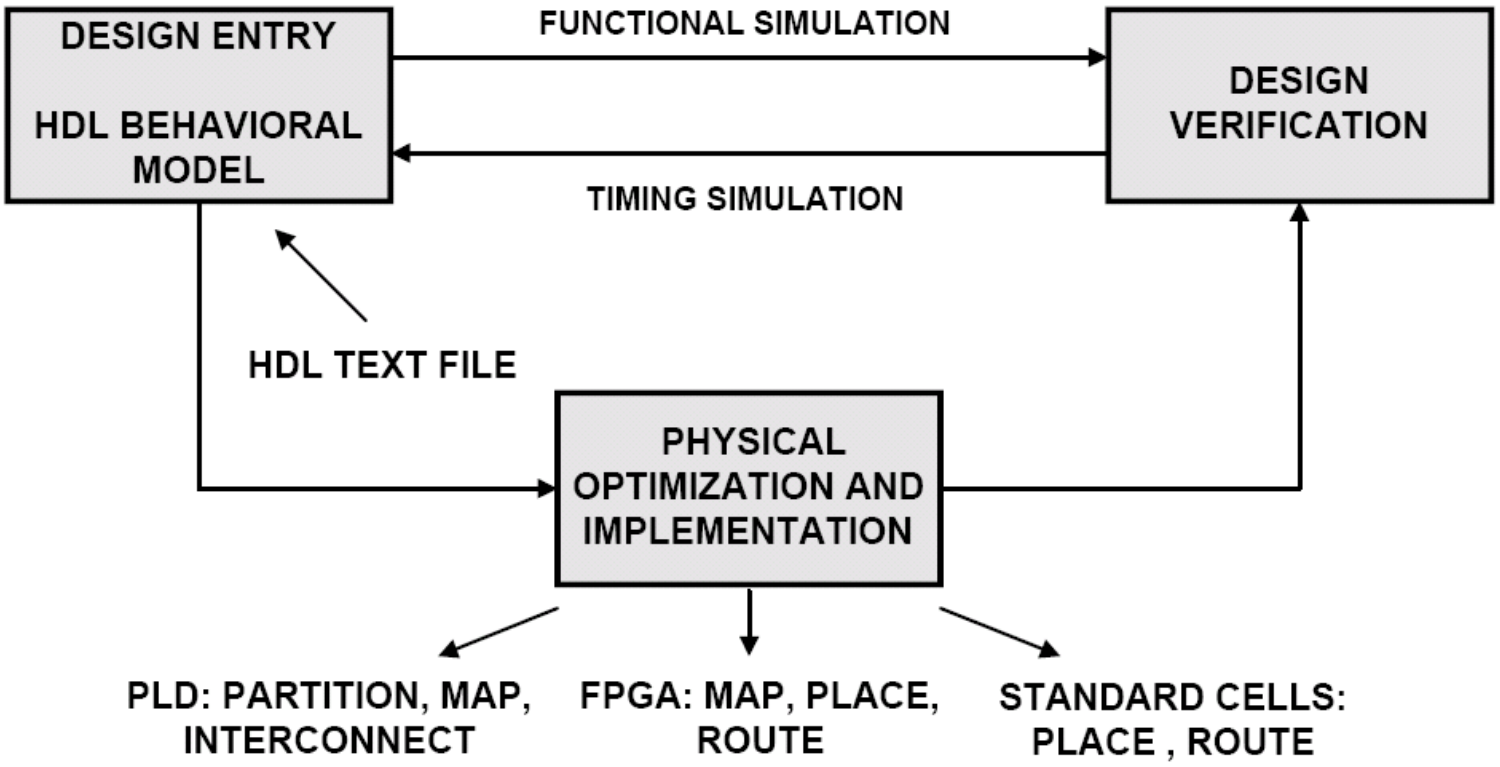
- HDL-based synthesis provides
  - Alternative to gate-level design
  - Higher level of design abstraction
  - Description of overall architecture
  - Description of functionality
- Synthesis tools provide
  - Automated gate-level representation
  - Optimal representation
  - Architectural exploration

HDL Descriptive Style

- HDL source style strongly influences synthesized product.
- Rules for descriptive styles are vendor-specific.
- Clocking rules must be observed for asynchronous and synchronous sequential behavior.
- Area-delay tradeoffs are made within the context of a vendor's tool and available technology.

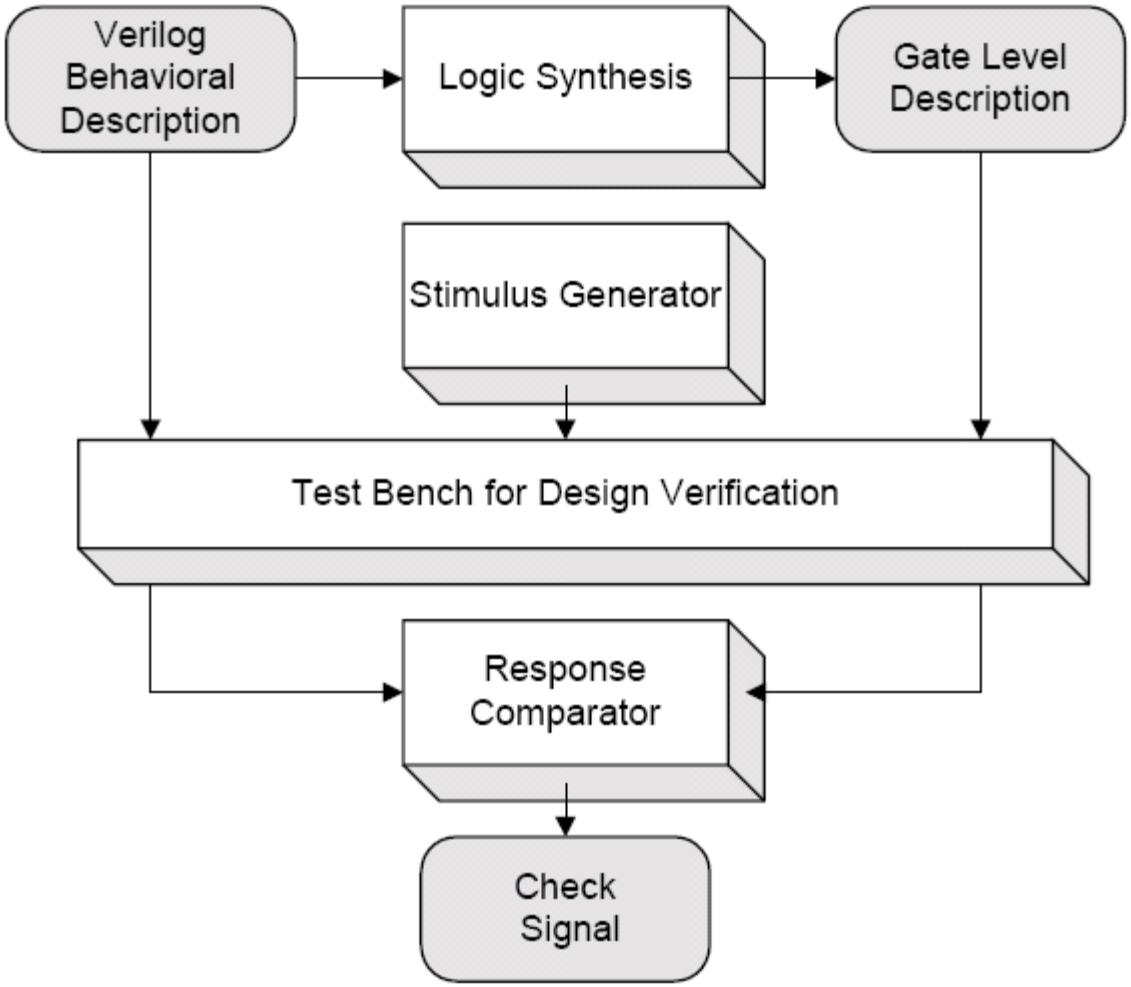
Not all algorithms are synthesizable!

HDL-Based Design Flow



In theory, synthesis tools "automatically" create an optimal gate-level realization. In practice, results are dependent upon the skill of the designer using the tool.

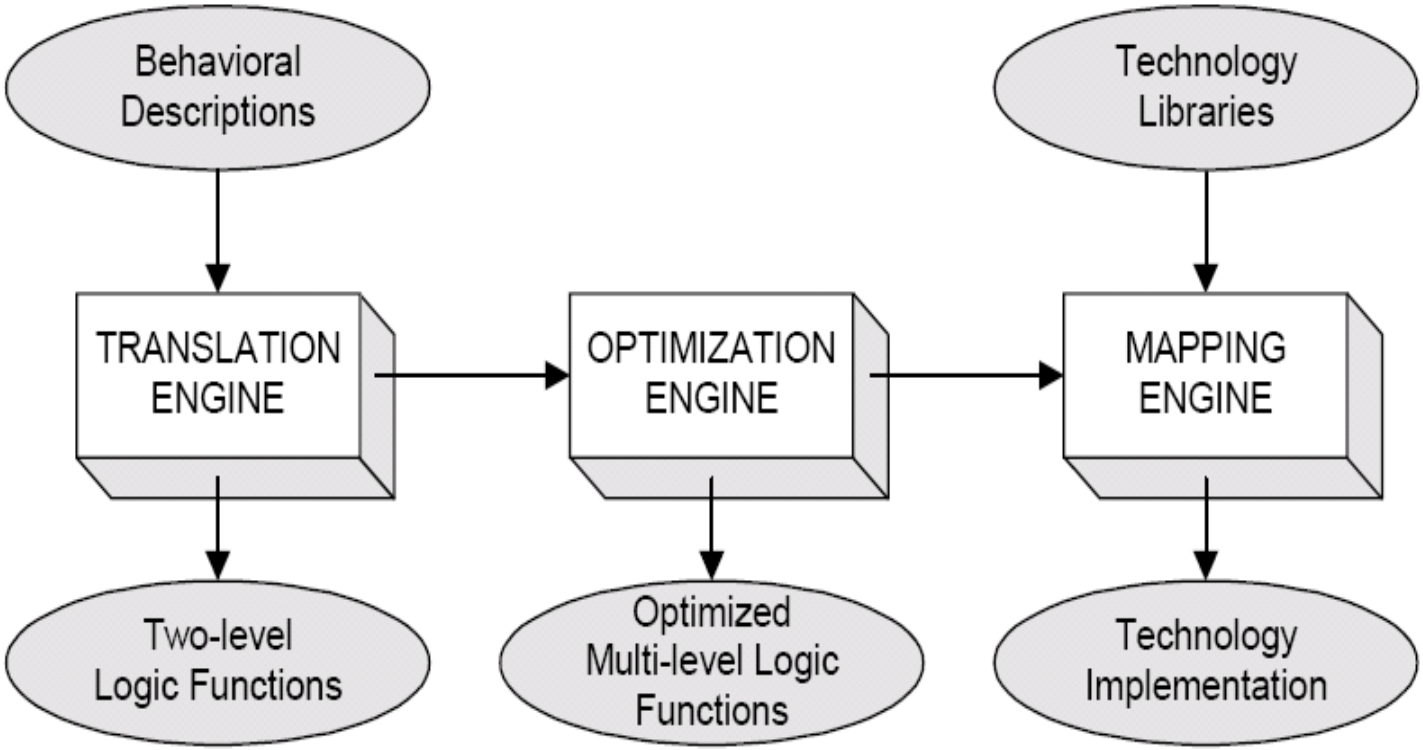
Testbench for Post-Synthesis Design Verification



Levels of Synthesis

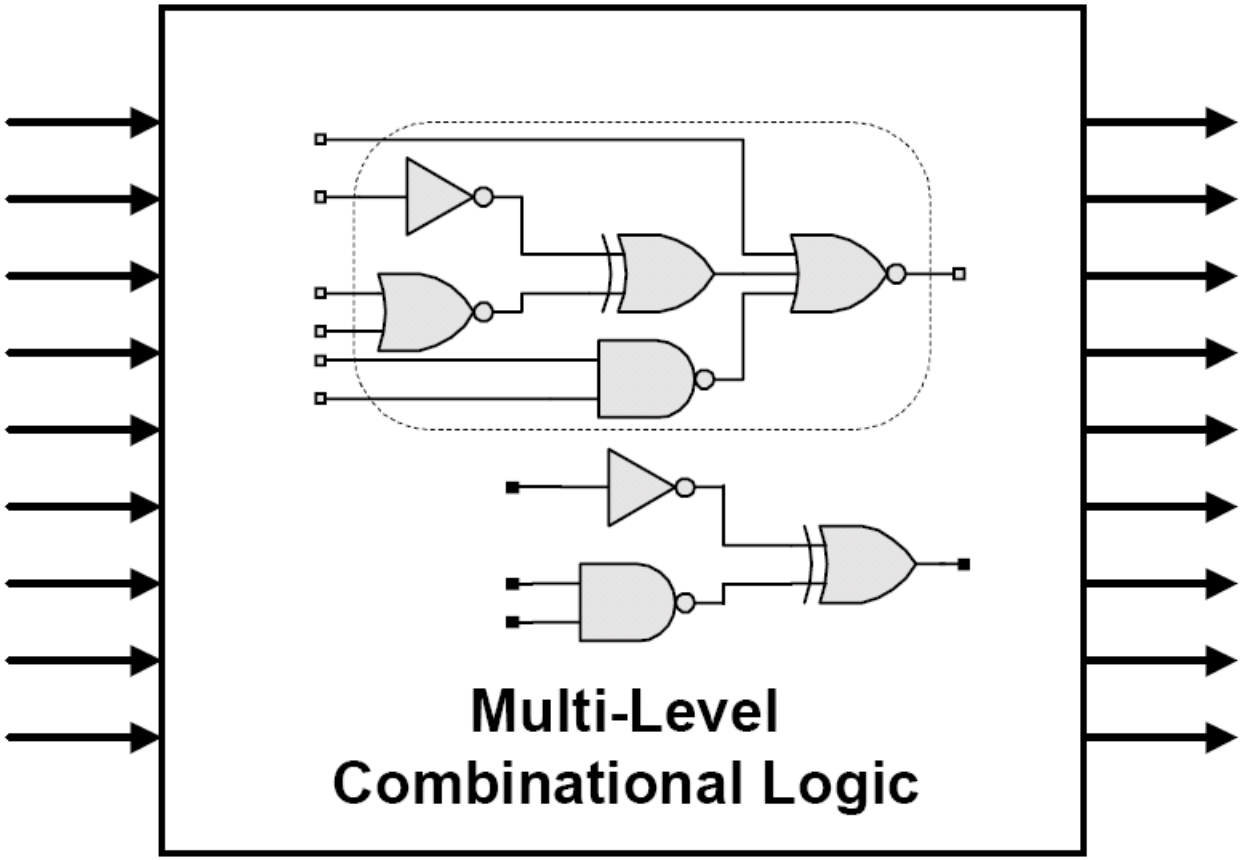
- **Logic synthesis**
  - Realizes an optimal circuit from a Boolean description.
- **RTL synthesis**
  - Creates a Boolean description from an RTL description.
- **Behavioral synthesis**
  - Creates a RTL description from an algorithmic model of functionality.

Synthesis Tool Organization



Synthesis of optimal combinational logic is subject to area, speed, or power, etc.

Multi-Level Combinational Logic



Logic synthesis treats a set of individual Boolean input-output equations as a multi-level circuit.

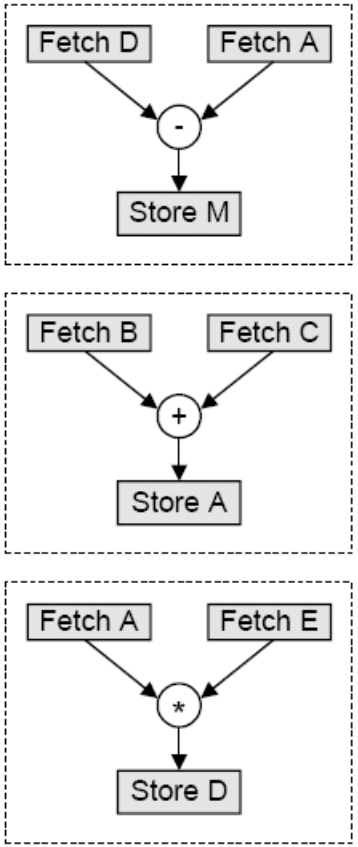
Register Transfer Level (RTL) Synthesis

- Transforms a behavior described in terms of operations on registers, signals, and constraints.
- Synthesizes an optimal realization in an assumed architecture.
- Begins with the assumptions that a set of hardware resources are available, and that the scheduling and allocation of resources have been determined by a data-flow graph (DFG).

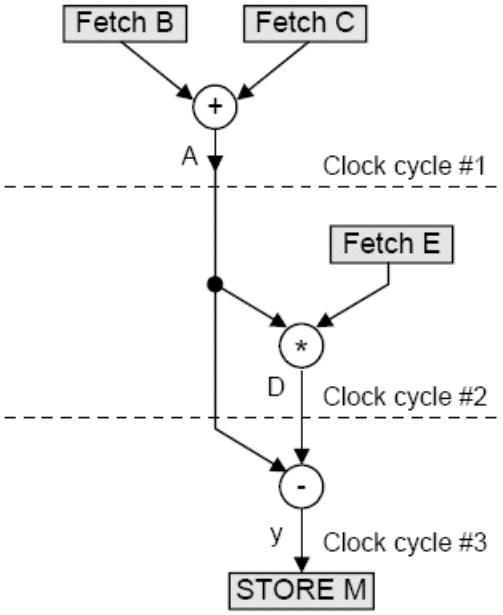
Behavioral Synthesis

- Creates an architecture whose resources can be scheduled and allocated to implement an algorithm.

M = D – A;
A = B + C;
D = A * E;



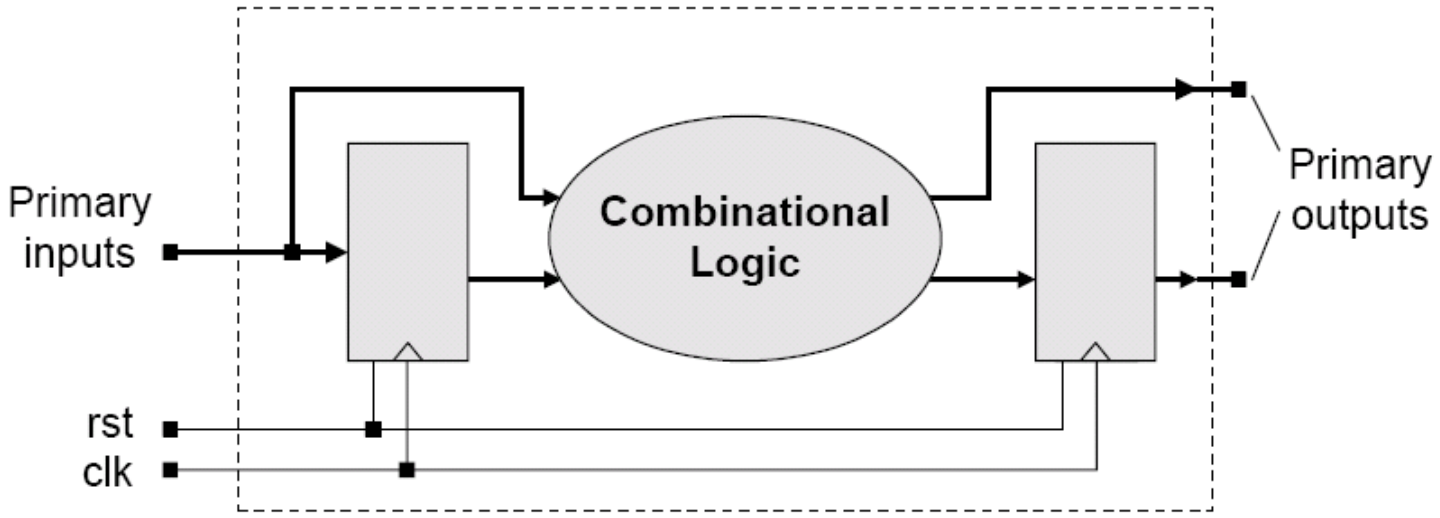
Parse trees



Data-flow graph

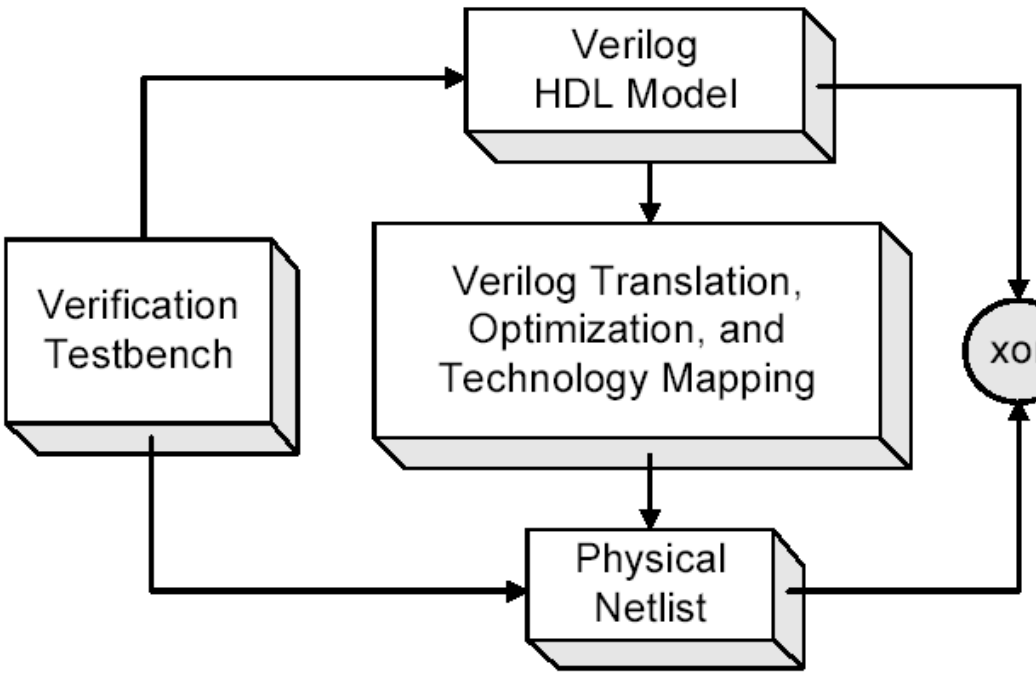
Structure for Synchronous Logic

- Storage elements are triggered by external clock/reset.
- Inputs to combinational logic are primary inputs or are from storage elements.
- The combinational logic is assumed to settle in one clock cycle.



Match of Pre- and Post-Synthesis Behaviors

- Functionality is independent of delays in the models.
- The outputs of the source description and the synthesized physical netlist must match at the clock edge boundaries.



tsim	clk	reg_A	reg_B
0	0	0000	1100
50	0	0000	1100
100	1	1011	0011
150	1	1011	0011

0000
0000
0000
0000

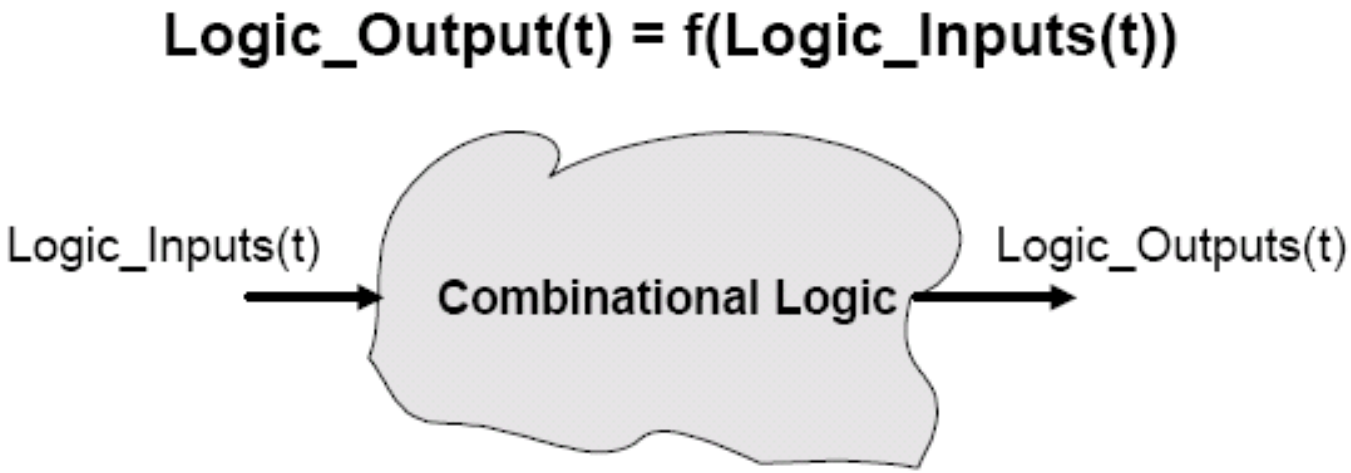
tsim	clk	reg_A	reg_B
0	0	0000	1100
50	0	0000	1100
100	1	1011	0011
150	1	1011	0011

Benefits of Synthesis

- Fast generation of the gate-level description from the HDL.
- Reduced effort to debug the gate-level design.
- Efficient gate-level implementation.
- Consistency between RTL and gate-level descriptions.
- Technology migration (FPGA, ASIC standard cell).
- Top-down organization with language-based description and documentation.

Synthesis of Combinational Logic

- A Boolean function whose output depends only on the instantaneous inputs to the logic, not its history.
- No storage elements or memory.

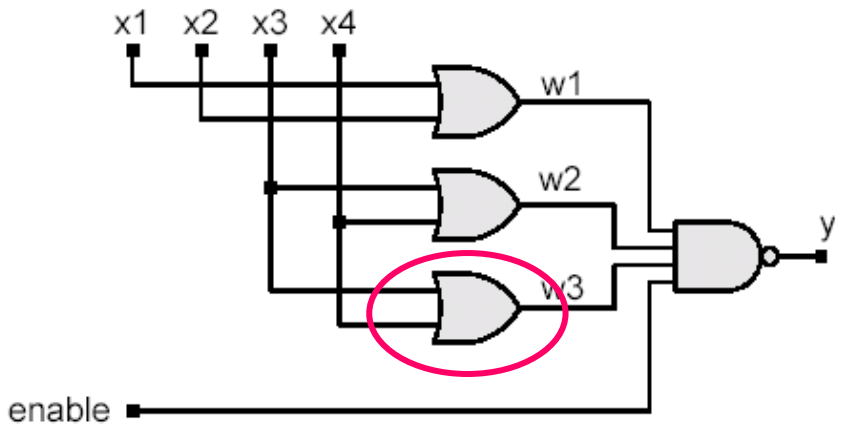


Rules

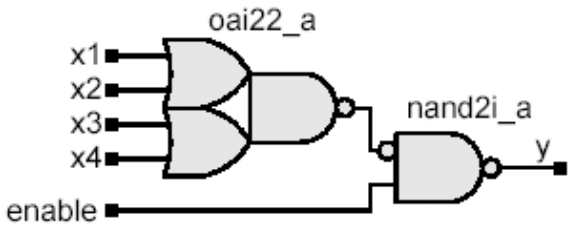
- Avoid technology dependent modeling; i.e. implement functionality, not timing.
- The combinational logic *must not have feedback*.
- Specify the output of a combinational behavior for all possible cases of its inputs.
- Logic that is not combinational will be synthesized as sequential.

Combinational Synthesis from a Netlist of Primitives

```
module or_nand_1 (enable, x1, x2, x3, x4, y);
    input       enable, x1, x2, x3, x4;
    output      y;
    wire        w1, w2, w3;

    or          (w1, x1, x2);
    or          (w2, x3, x4);
    or          (w3, x3, x4);   // redundant
    nand        (y, w1, w2, w3, enable);
endmodule
```



Pre-synthesis

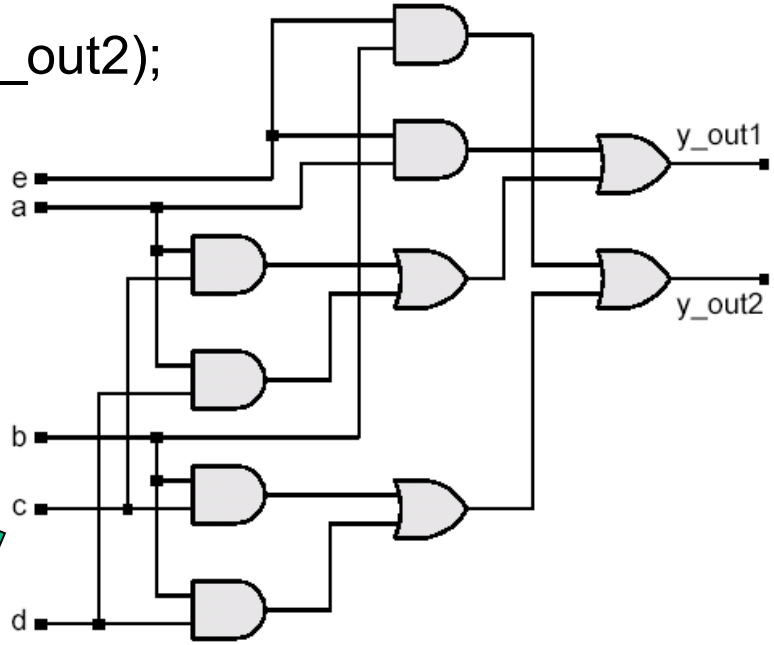


Post-synthesis

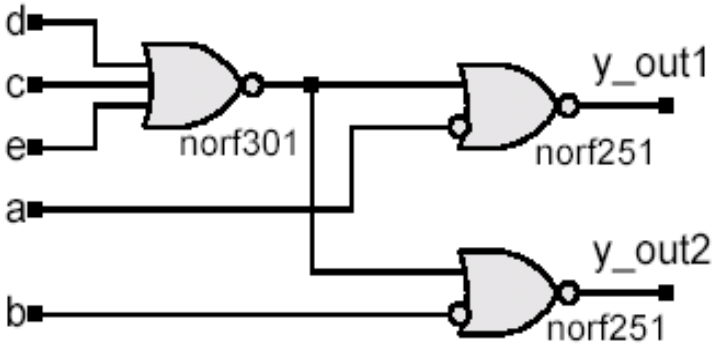
✍ Even if a design is expressed as a netlist of generic primitives, it can still be synthesized to remove any redundant logic before mapping into a technology.

Example 1

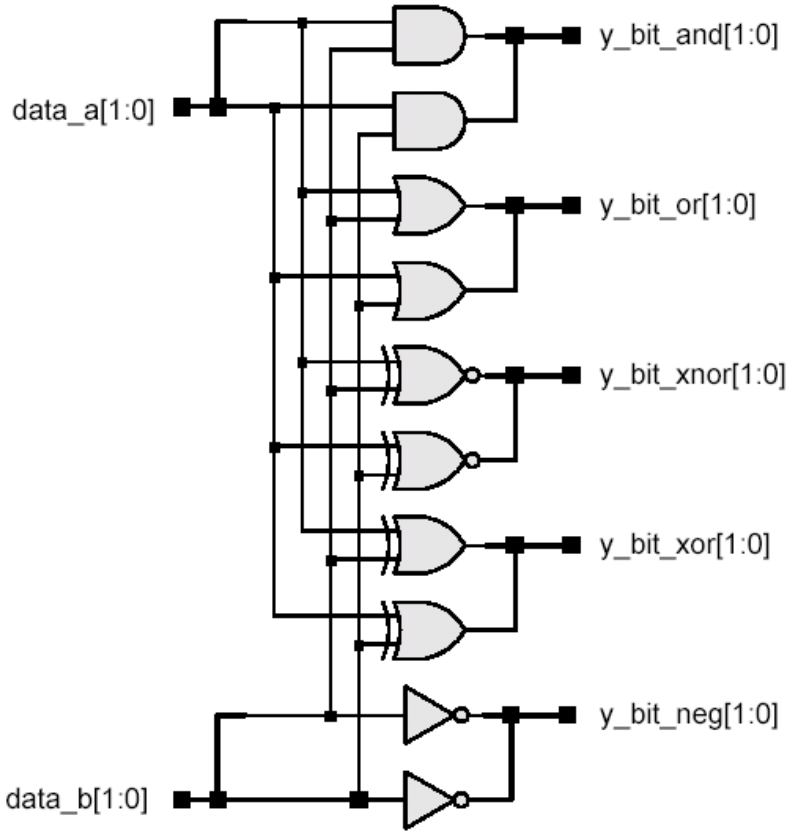
```
module boole_opt (a, b, c, d, e, y_out1, y_out2);
    input       a, b, c, d, e;
    output      y_out1, y_out2;
    and  (y1, a, c);
    and  (y2, a, d);
    and  (y3, a, e);
    or   (y4, y1, y2);
    or   (y_out1, y3, y4);
    and  (y5, b, c);
    and  (y6, b, d);
    and  (y7, b, e);
    or   (y8, y5, y6);
    or   (y_out2, y7, y8);
endmodule
```



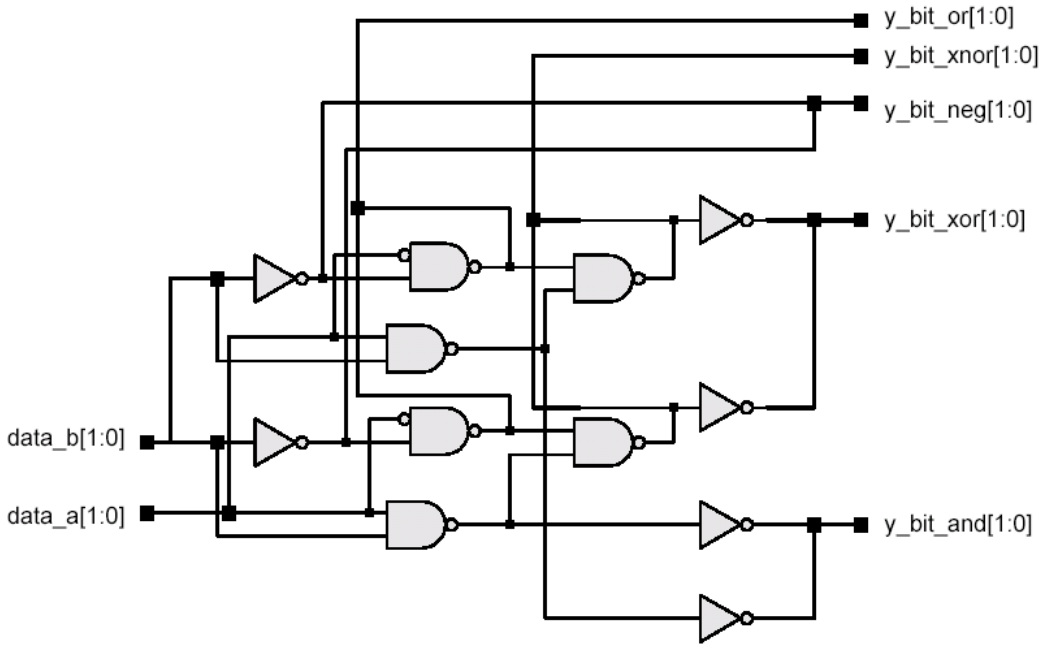
Logic synthesis



Example 2



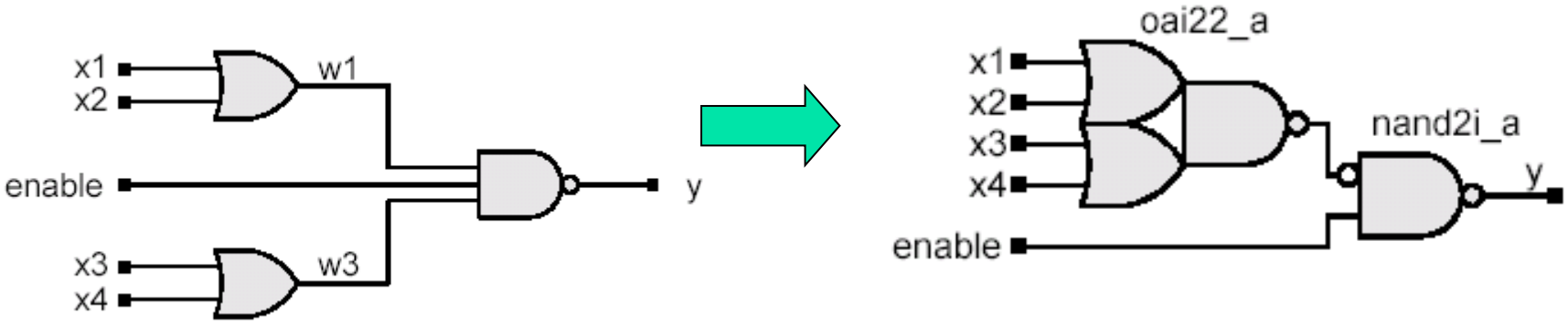
Generic implementation of bitwise operations



Synthesized multi-level structure

Combinational Synthesis from Continuous Assignments

```
module or_nand_2 (enable, x1, x2, x3, x4, y);
    input       enable, x1, x2, x3, x4;
    output      y;

    assign      y = ~ (enable & (x1 | x2) & (x3 | x4));
endmodule
```



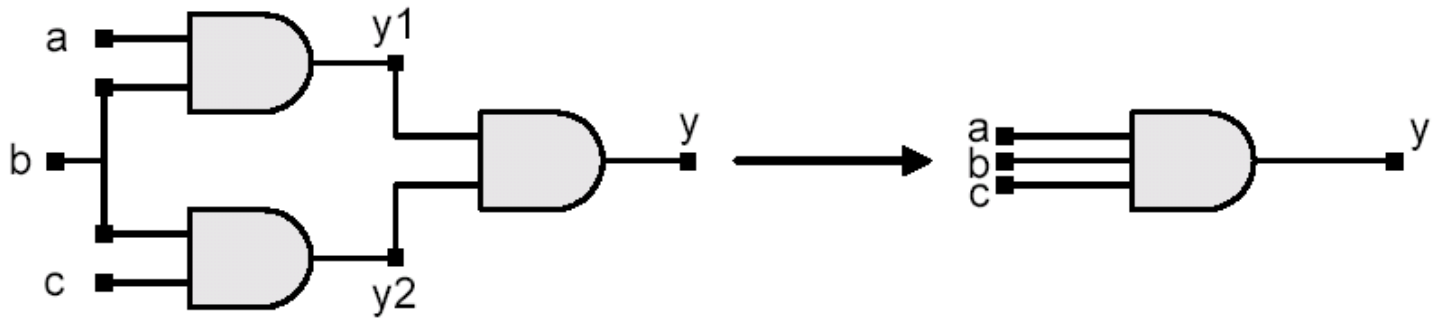
Pre-synthesis

Post-synthesis

Example

```
module and_gate (a, b, c, y);
    input       a, b, c;
    output      y;
    wire        y1, y2;

    assign      y1 = a & b;
    assign      y2 = c & b;
    assign      y = y1 & y2;
endmodule
```

- An internal net declared in the source description may be eliminated by the optimization process.
- A net that is declared as a primary input or output will be retained in the synthesized design.

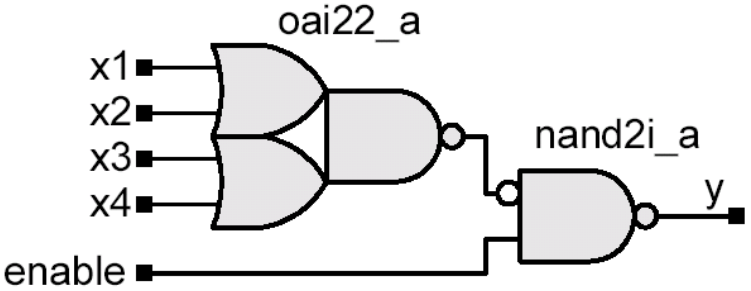


Combinational Synthesis from a Cyclic Behavior

- The behavior must assign value to the output under all events that affect the right-hand side expression of the assignment; otherwise, *an unwanted latches will be produced*.
- All inputs to a behavior that is to implement combinational logic must be included in the event control expression, either explicitly or implicitly.

Example 1

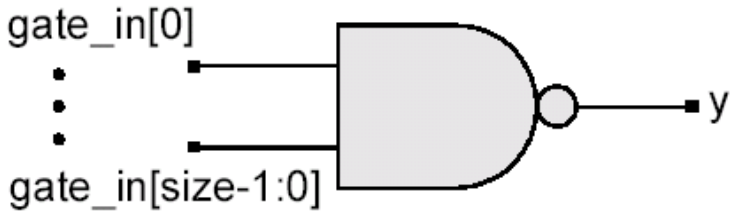
```
module or_nand_3 (enable, x1, x2, x3, x4, y);
    input       enable, x1, x2, x3, x4;
    output      y;
    reg         y;

    always @ (enable or x1 or x2 or x3 or x4)
        y = ~ (enable & (x1 | x2) & (x3 | x4));
endmodule
```



- The use of a register variable does not imply that the associated behavior is sequential.
- Nor does it imply that the synthesized hardware will have a storage device.

Example 2

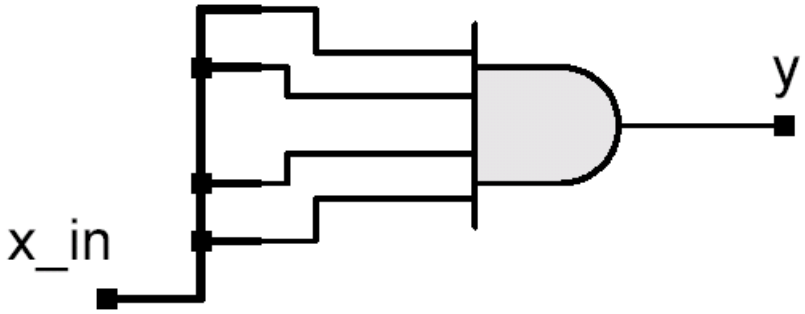
```
module n_gate1 (gate_in, y);
parameter          size = 3;
input    [size-1: 0] gate_in;   output  y;
reg                  y_int;     integer k;
assign  y = y_int;
always @ (gate_in) begin : look_for_0
    y_int = 0;
    for (k=0; k <= size-1; k=k+1)
      if (gate_in[k] == 0) begin
          y_int = 1;
          disable look_for_0;
      end
end
endmodule
```



- A register variable that is a primary output at the top level of a design will appear in the synthesized result (not necessarily as a storage element).
- A register variable declared in the source description may be eliminated by the optimization process (e.g., y_int, k).

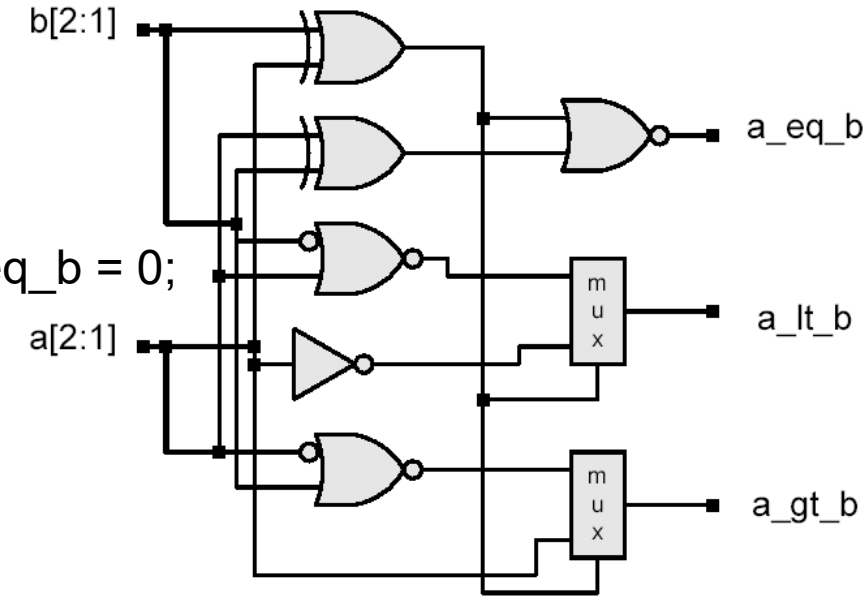
Example 3

```
module and4_behav (y, x_in);
parameter       word_length = 4;
input   [word_length-1:0] x_in;   output  y;
reg                 y;                    integer k;

always @ (x_in) begin: check_for_0
    y = 1;
    for (k = 0; k <= word_length -1; k = k+1)
      if (x_in[k] == 0) begin
          y = 0;
          disable check_for_0;
      end
end
endmodule
```



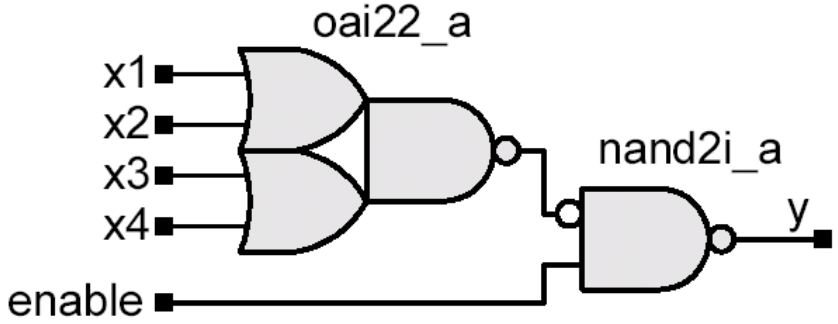
Example 4

```
module comparator (a, b, a_gt_b, a_lt_b, a_eq_b);
parameter          size = 2;
input      [size: 1]  a, b;
output   a_gt_b, a_lt_b, a_eq_b;     reg   a_gt_b, a_lt_b, a_eq_b;
integer  k;
always @ (a or b) begin: compare_loop
    for (k = size; k > 0; k = k-1) begin
      if (a[k] != b[k]) begin
          a_gt_b = a[k];  a_lt_b = ~a[k];  a_eq_b = 0;
          disable compare_loop;
      end
    end
    a_gt_b = 0; a_lt_b = 0; a_eq_b = 1;
end
endmodule
```



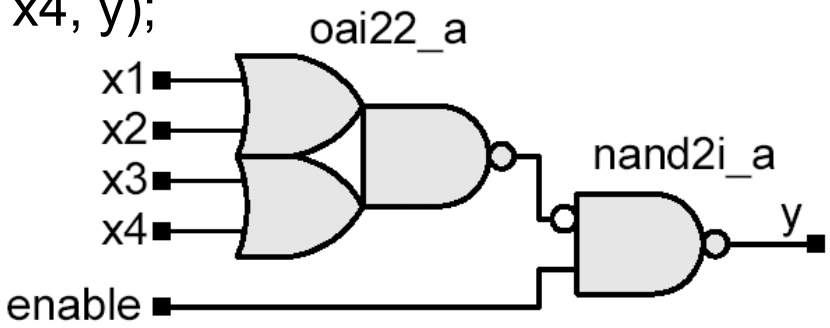
Combinational Synthesis from a Function

```
module or_nand_4 (enable, x1, x2, x3, x4, y);
    input       enable, x1, x2, x3, x4;
    output      y;

    assign      y = or_nand (enable, x1, x2, x3, x4);

    function or_nand;
        input enable, x1, x2, x3, x4;
        or_nand = ~ (enable & (x1 | x2) & (x3 | x4));
    endfunction
endmodule
```



Combinational Synthesis from a Task

```
module or_nand_5 (enable, x1, x2, x3, x4, y);
    input       enable, x1, x2, x3, x4;
    output      y;
    reg         y;

    always @ (enable or x1 or x2 or x3 or x4)
        or_nand (enable, x1, x2, x3, x4, y);

    task or_nand;
        input   enable, x1, x2, x3, x4;
        output  y;
        y = ~ (enable & (x1 | x2) & (x3 | x4));
    endtask
endmodule
```

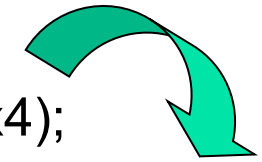


Simulation Efficiency

- Use of procedural-continuous assignment (PCA) reduces the size of the event sensitivity list and improves the simulation efficiency.
- **assign** is used without **deassign** to establish a binding between the output and the inputs.
- **deassign** is used only in sequential models (e.g., latches), when a previously assigned binding of the register variable is to be released.
- Some tools do not support PCA for synthesis.
- *Two models might be used and switched between simulation and synthesis.*

Example 1

```
module or_nand_6 (enable, x1, x2, x3, x4, y);
    input       enable, x1, x2, x3, x4;
    output      y;
    reg         y;

    always @ (enable)
        if (enable)
            assign y = (x1 | x2) & (x3 | x4);
        else
            assign y = 1;
endmodule
```



```
always @ (enable or x1 or x2 or x3 or x4)
    if (enable)
      y = (x1 | x2) & (x3 | x4);
    else
      y = 1;
```

Example 2

```
always @ (opcode or a or b)
    case (opcode)
        3'b111: out_y = a & b;
        3'b011: out_y = a | b;
        3'b001: out_y = a ^ b;
        default: out_y = 2'bx;
    endcase
```

Preferred for synthesis

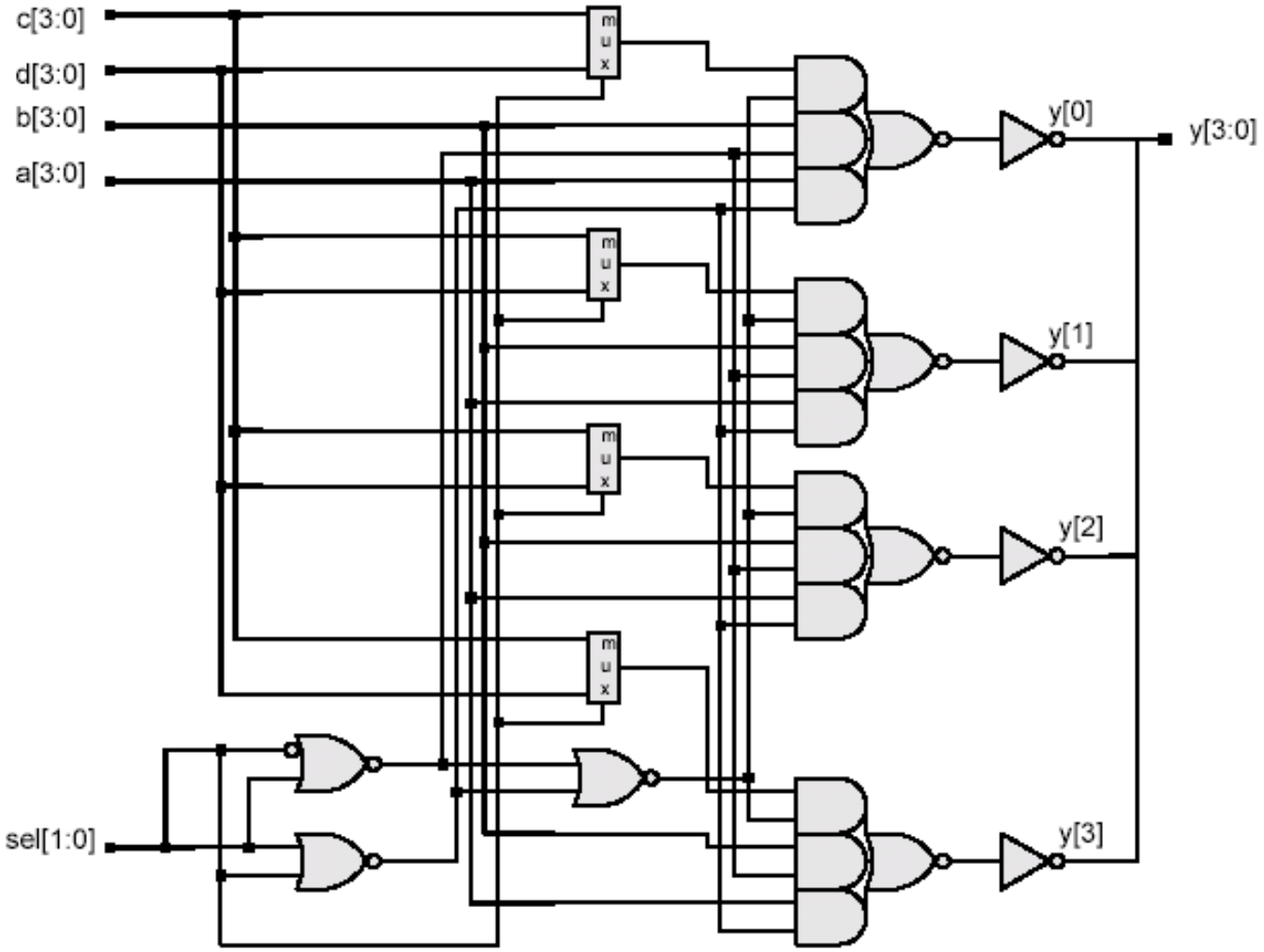
```
always @ (opcode)
    case (opcode)
        3'b111: assign out_y = a & b;
        3'b011: assign out_y = a | b;
        3'b001: assign out_y = a ^ b;
        default: assign out_y = 2'bx;
    endcase
```

Preferred for simulation

Synthesis of Control Logic for Multiplexed Datapaths

```
// assign / conditional
module syn1_mux_4bits (a, b, c, d, sel, y);
    input       [3:0] a, b, c, d;
    input       [1:0] sel;
    output      [3:0] y;

    assign y =  (sel == 0) ? a :
                (sel == 1) ? b :
                (sel == 2) ? c :
                (sel == 3) ? d : 4'bx;
endmodule
```

Synthesis Result for “syn1_mux_4bits”



Under some specific synthesis tool and technology

Example 1

```
// case / PCA, may not be synthesized
module syn2_mux_4bits
        (a, b, c, d, sel, y);
input   [3:0] a, b, c, d;
input   [1:0] sel;
output  [3:0] y;
reg     [3:0] y;

always @(sel)
    case (sel)
        0:      assign y = a;
        1:      assign y = b;
        2:      assign y = c;
        3:      assign y = d;
        default: assign y = 4'bx;
    endcase
endmodule
```

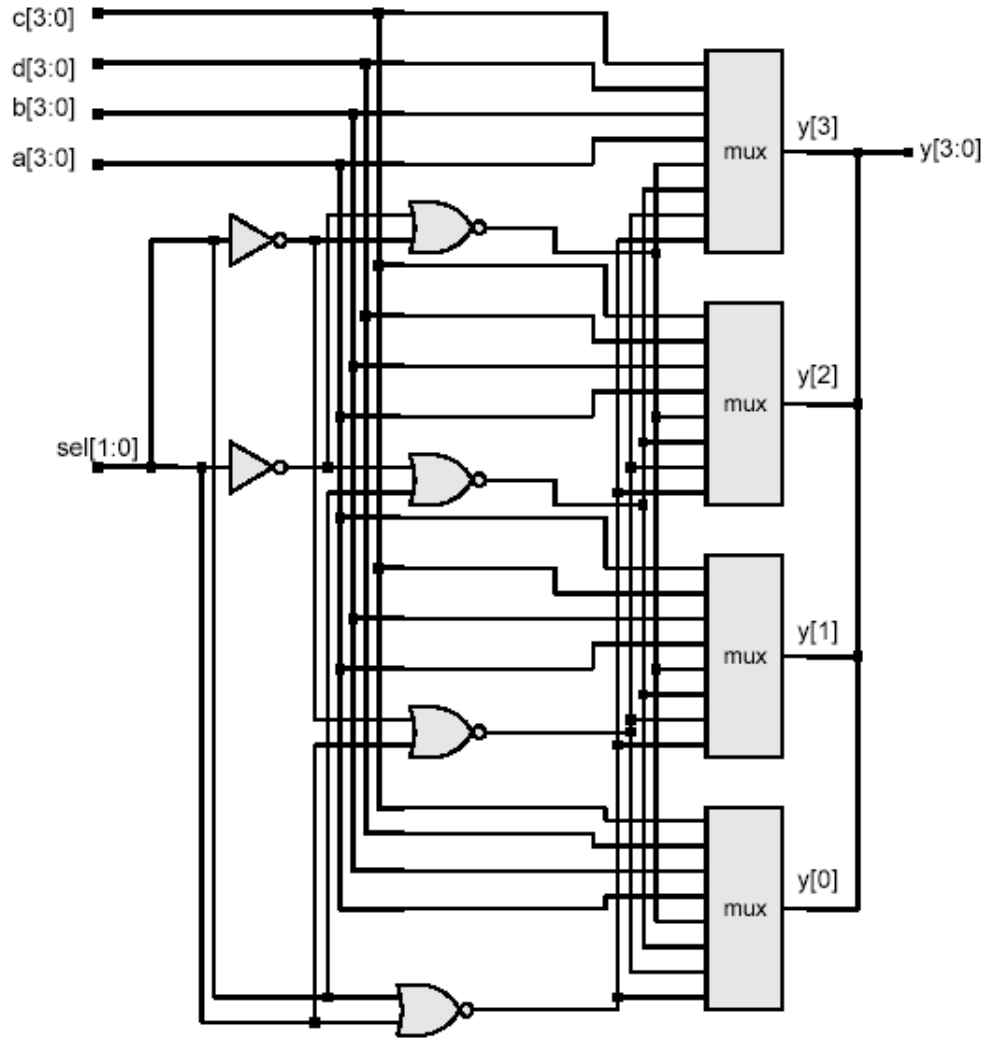
```
// synthesis friendly
module syn2b_mux_4bits
        (a, b, c, d, sel, y);
input   [3:0] a, b, c, d;
input   [1:0] sel;
output  [3:0] y;
reg     [3:0] y;

always @ (a or b or c or d or sel)
    case (sel)
        0:      y = a;
        1:      y = b;
        2:      y = c;
        3:      y = d;
        default: y = 4'bx;
    endcase
endmodule
```

Example 2

```
// if … else / PCA,
// may not be synthesized
module syn3_mux_4bits
            (a, b, c, d, sel, y);
input       [3:0] a, b, c, d;
input       [1:0] sel;
output      [3:0] y;
reg         [3:0] y;

always @ (sel)
    if (sel == 0)       assign y = a;
    else if (sel == 1) assign y = b;
    else if (sel == 2) assign y = c;
    else if (sel == 3) assign y = d;
    else                assign y = 4'bx;
endmodule
```

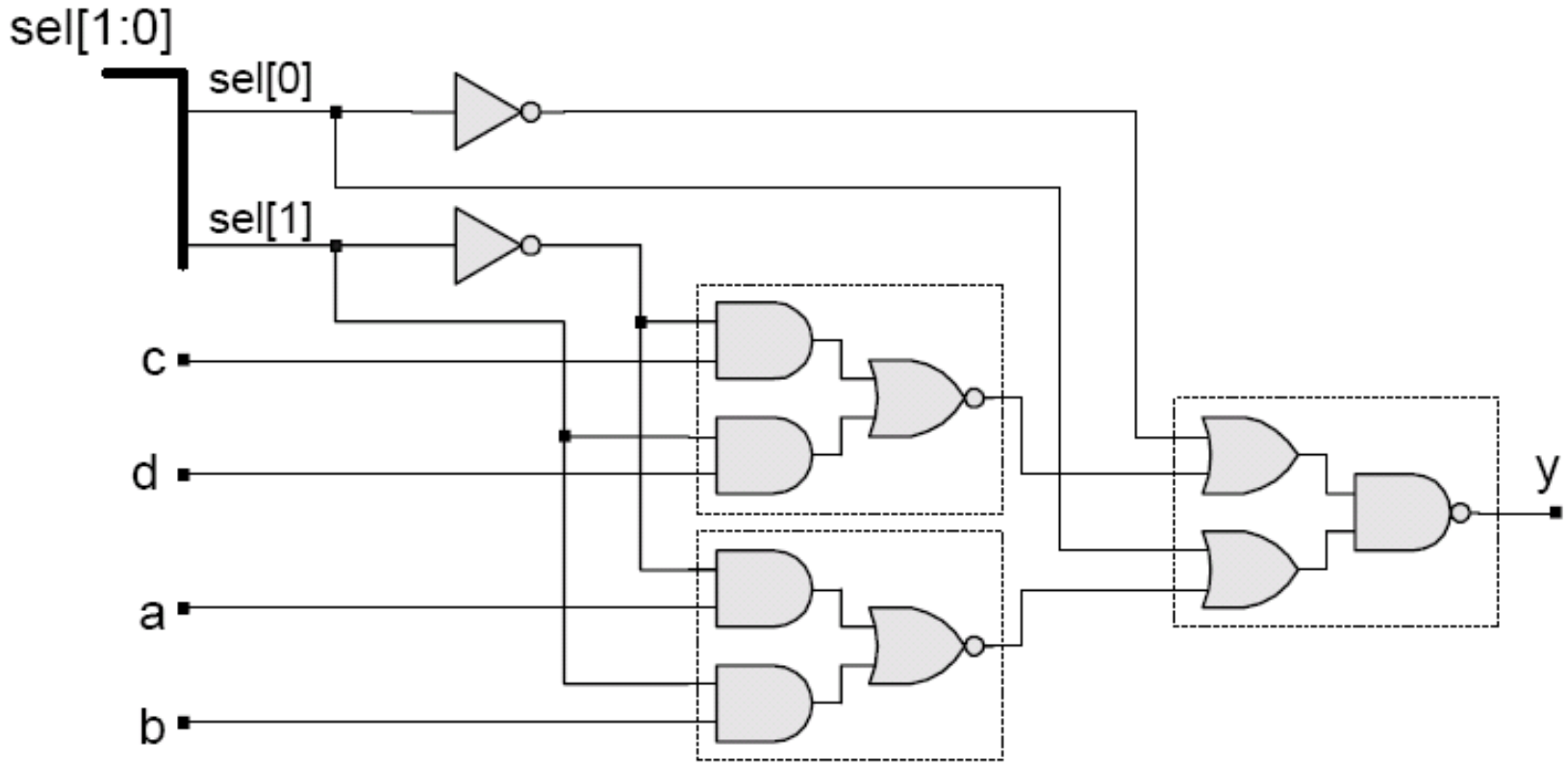
```
// synthesis friendly
module syn3b_mux_4bits
            (a, b, c, d, sel, y);
input       [3:0] a, b, c, d;
input       [1:0] sel;
output      [3:0] y;
reg         [3:0] y;

always @ (a or b or c or d or sel)
    if (sel == 0)               y = a;
    else if (sel == 1)          y = b;
    else if (sel == 2)          y = c;
    else if (sel == 3)          y = d;
    else                        y = 4'bx;
endmodule
```

Synthesis Result for “syn2b_mux_4bits” and “syn3b_mux_4bits”



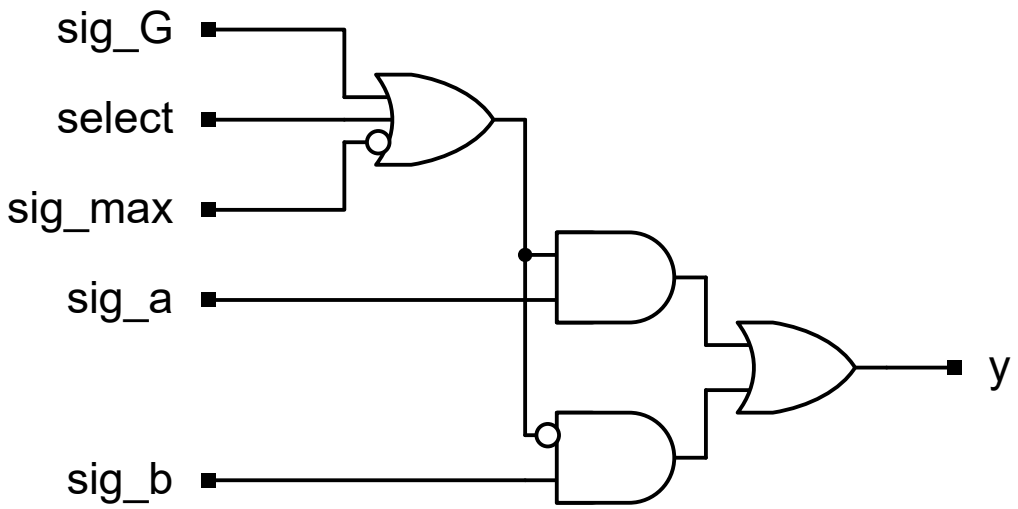
Under some specific synthesis tool and technology

Synthesis Result for all Three Versions with a Scalar Channel



Multiplexer with Selector Logic

```
module mux_logic (select, sig_G, sig_max, sig_a, sig_b, y);
    input select, sig_G, sig_max, sig_a, sig_b;
    output y;

    assign y = (select == 1) || (sig_G == 1) || (sig_max == 0) ?
                    sig_a : sig_b;
endmodule
```



Unexpected and Unwanted Latches

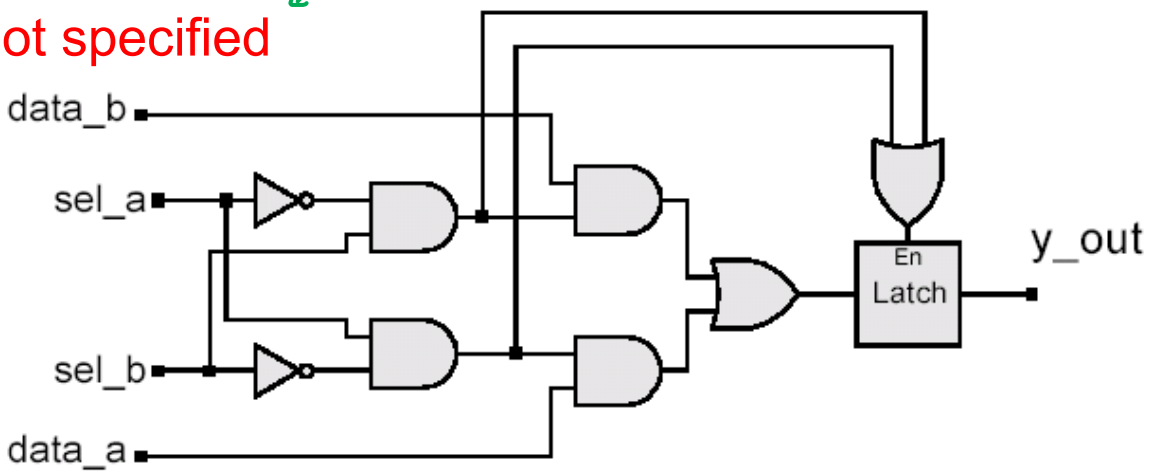
不完全指定

- Incompletely specified **case** statements and conditionals (**if** statements) may lead to the synthesis of unwanted latches.
- When a **case** statement does not specify an output for all possible inputs, the synthesis tool infers an implicit latch.
- The description implies that the output should retain its residual value under the conditions that were left unspecified.

Example

```
module mux_latch (sel_a, sel_b, data_a, data_b, y_out);
    input       sel_a, sel_b, data_a, data_b;
    output      y_out;
    reg         y_out;
    always @ (sel_a or sel_b or data_a or data_b)
        case ({sel_a, sel_b})
            2'b10: y_out = data_a;
            2'b01: y_out = data_b;
            // 00, 11: not specified
        endcase
endmodule
```

若無指定
會產生鎖 (Latch)



Ex: ALU1

```
module alu_incomplete1a (data_a, data_b, enable, opcode, alu_out);
input    [3:0]    data_a, data_b;
input             enable;
input    [2:0]    opcode;
output            alu_out; // Note: scalar, for convenience of illustration
reg      [3:0]    alu_reg;

assign alu_out = (enable == 1) ? alu_reg : 4'b0;

always @ (opcode)
    case (opcode)
        3'b001:   alu_reg = data_a | data_b;
        3'b010:   alu_reg = data_a ^ data_b;
        3'b110:   alu_reg = ~data_b;
    endcase
endmodule
```

Incomplete event control expression and **case** statement; no default **case** assignment.

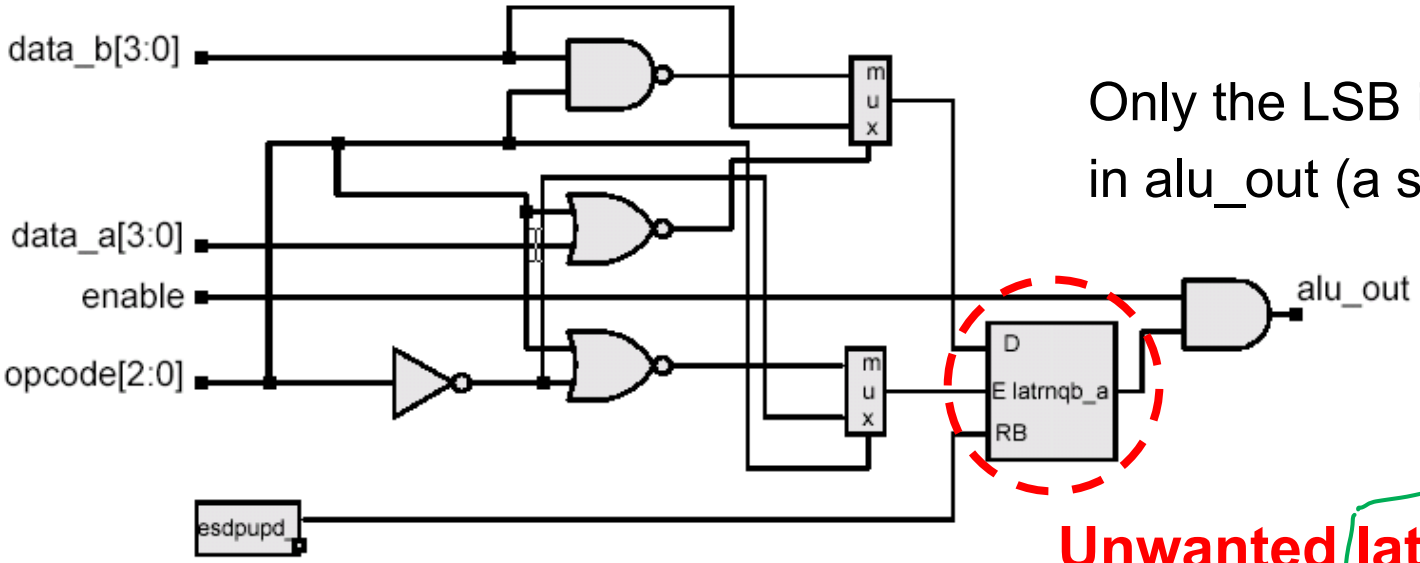
Ex: ALU2

```
module alu_incomplete2a (data_a, data_b, enable, opcode, alu_out);
input     [3:0]     data_a, data_b;
input               enable;
input     [2:0]     opcode;
output              alu_out; // Note: scalar, for convenience of illustration
reg       [3:0]     alu_reg;

assign alu_out = (enable == 1) ? alu_reg : 4'b0;

always @ (opcode or data_a or data_b)
    case (opcode)
        3'b001:     alu_reg = data_a | data_b;
        3'b010:     alu_reg = data_a ^ data_b;
        3'b110:     alu_reg = ~data_b;
    endcase
endmodule
```

Complete event control expression, but incomplete **case** statement; no default **case** assignment.

Synthesis Result for ALU1 & ALU2

- ALU1 and ALU2 synthesize to the same circuit, even though they have different event control expressions.
- They have different pre-synthesis behavior because ALU1 ignores events on the datapath.



Only the LSB is shown in alu_out (a scalar)

Unwanted latch!

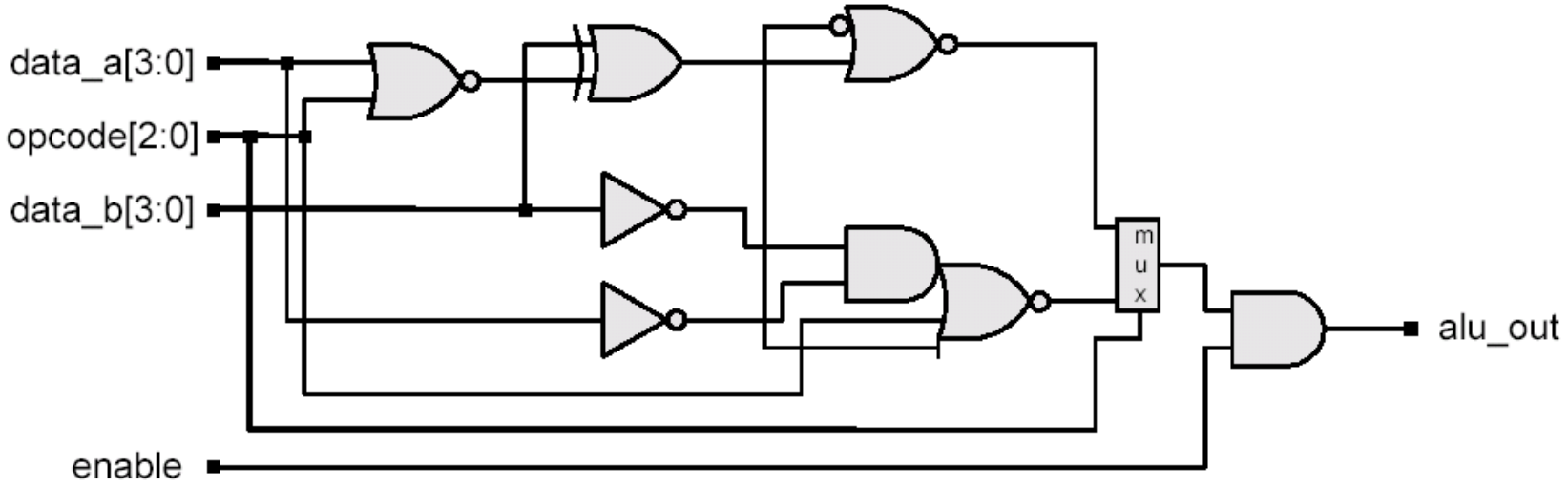
Under some specific synthesis tool and technology

Ex: ALU3

```
module alu_complete (data_a, data_b, enable, opcode, alu_out);
input     [3:0]     data_a, data_b;
input               enable;
input     [2:0]     opcode;
output              alu_out; // Note: scalar, for convenience of illustration
reg       [3:0]     alu_reg;

assign alu_out = (enable == 1) ? alu_reg : 4'b0;
always @ (opcode or data_a or data_b)
    case (opcode)
        3'b001:     alu_reg = data_a | data_b;
        3'b010:     alu_reg = data_a ^ data_b;
        3'b110:     alu_reg = ~data_b;
        default:    alu_reg = 4'b0;
    endcase
endmodule
```

Complete event control expression and default **case** assignment.

Synthesis Result for ALU3



Fully combinational !

Alternative Description for ALU3

```
module alu_complete2 (data_a, data_b, enable, opcode, alu_out);
input    [3:0]    data_a, data_b;
input             enable;
input    [2:0]    opcode;
output            alu_out; // Note: scalar, for convenience of illustration
reg      [3:0]    alu_reg;

assign alu_out = (enable == 1) ? alu_reg : 4'b0;
always @ (opcode)
    case (opcode)
        3'b001:   assign alu_reg = data_a | data_b;
        3'b010:   assign alu_reg = data_a ^ data_b;
        3'b110:   assign alu_reg = ~data_b;
        default:  assign alu_reg = 4'b0;
    endcase
endmodule
```

Complete event control expression and default **case** assignment.

Ex: ALU4

```
module alu_z1 (data_a, data_b, enable, opcode, alu_out);
input     [3:0]     data_a, data_b;
input               enable;
input     [2:0]     opcode;
output              alu_out; // Note: scalar, for convenience of illustration
reg       [3:0]     alu_reg;

assign alu_out = (enable == 1) ? alu_reg : 4'bz;
always @ (opcode or data_a or data_b)
    case (opcode)
        3'b001:     alu_reg = data_a | data_b;
        3'b010:     alu_reg = data_a ^ data_b;
        3'b110:     alu_reg = ~data_b;
        default:    alu_reg = 4'b0;
    endcase
endmodule
```

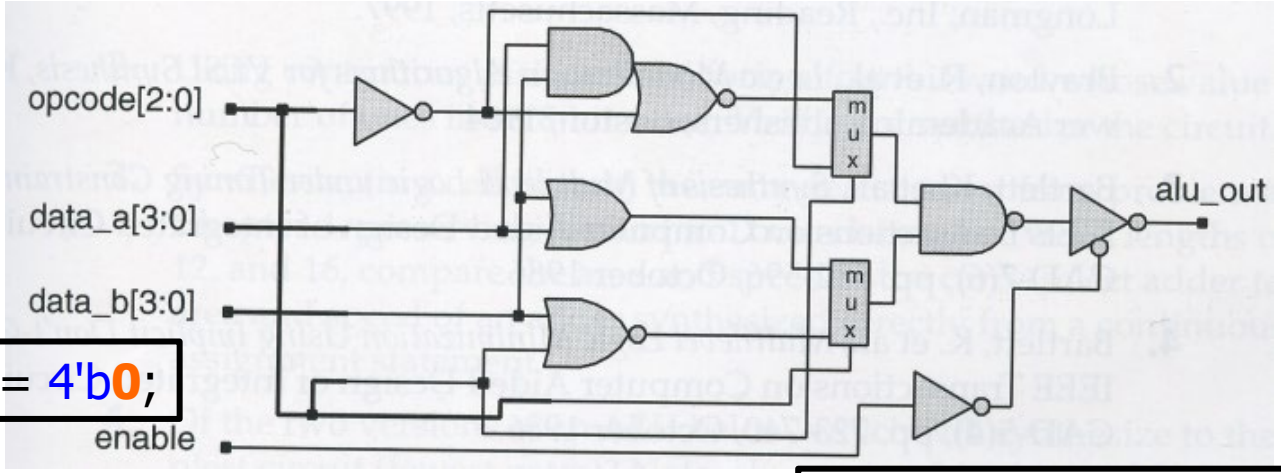
高阻抗 -> bus

Ex: ALU5

```
module alu_z2 (data_a, data_b, enable, opcode, alu_out);
input     [3:0]     data_a, data_b;
input               enable;
input     [2:0]     opcode;
output              alu_out; // Note: scalar, for convenience of illustration
reg       [3:0]     alu_reg;

assign alu_out = (enable == 1) ? alu_reg : 4'bz;
always @ (opcode or data_a or data_b)
    case (opcode)
        3'b001:     alu_reg = data_a | data_b;
        3'b010:     alu_reg = data_a ^ data_b;
        3'b110:     alu_reg = ~data_b;
        default:    alu_reg = 4'bx;
    endcase
endmodule
```

若用不到可設為X
更易化簡→面積更小

Synthesis Results

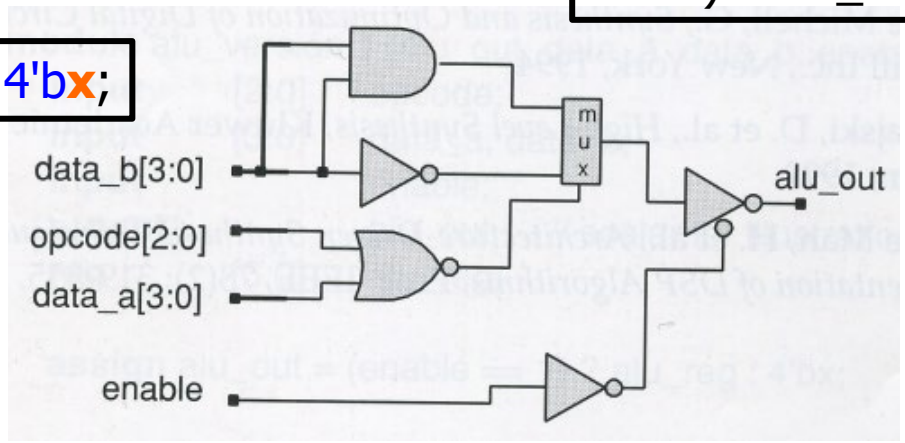


default: alu_reg = 4'b**0**;

alu_z1

assign alu_out = (enable == 1) ? alu_reg : 4'bz;

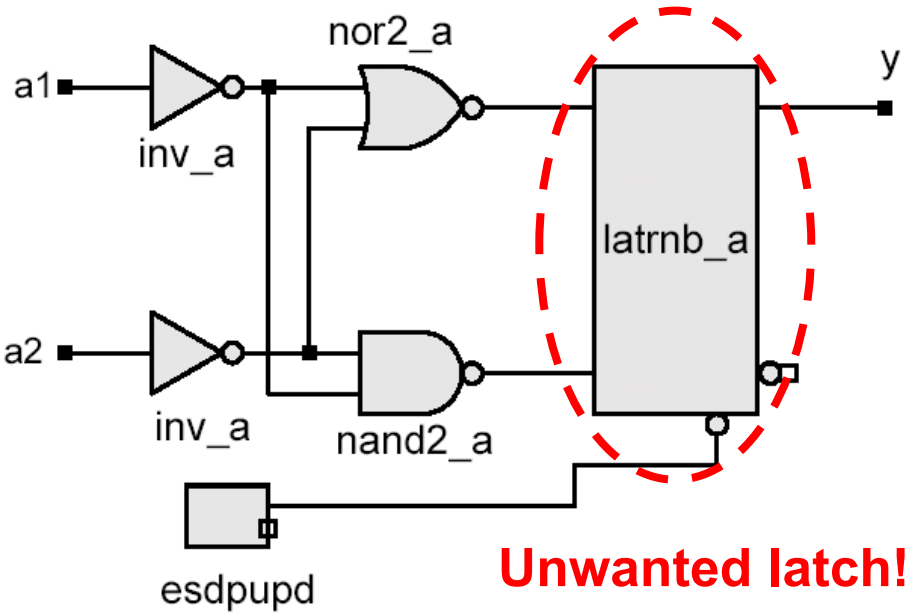
default: alu_reg = 4'b**x**;



alu_z2

Example

```
module incomplete_and (a1, a2, y);
input   a1, a2;
output  y;
reg     y;

always @ (a1 or a2) begin
    if ({a2, a1} == 2'b11) y = 1; else
    if ({a2, a1} == 2'b01) y = 0; else
    if ({a2, a1} == 2'b10) y = 0;
end
endmodule
```

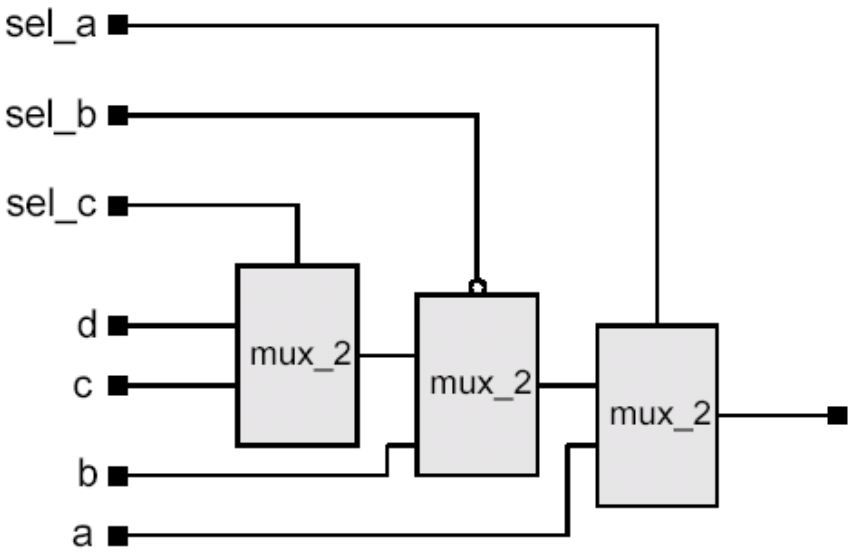


Synthesis of Priority Structures

- A **case** statement implicitly attaches higher priority to the first case_item than to the last one.
- An **if** statement implies higher priority to the first branch than to the remaining branches.
- A synthesis tool will determine whether or not the case items of a **case** statement are mutually exclusive.
- If they are, the synthesis tool will treat them as though they had equal priority and will synthesize a mux rather than priority structure.

Example

```
module mux_4pri (a, b, c, d, sel_a, sel_b, sel_c, y);
    input       a, b, c, d, sel_a, sel_b, sel_c;
    output      y;
    reg         y;

    always @ (a or b or c or d or sel_a or sel_b or sel_c) begin
        if (sel_a == 1) y = a; else
        if (sel_b == 0) y = b; else
        if (sel_c == 1) y = c; else
                        y = d;
    end
endmodule
```



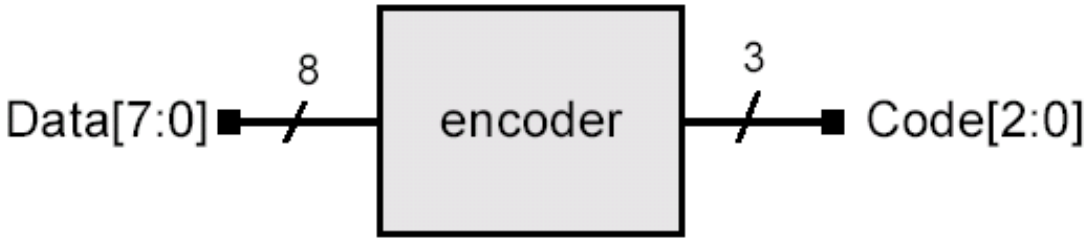
Treatment of Default Conditions

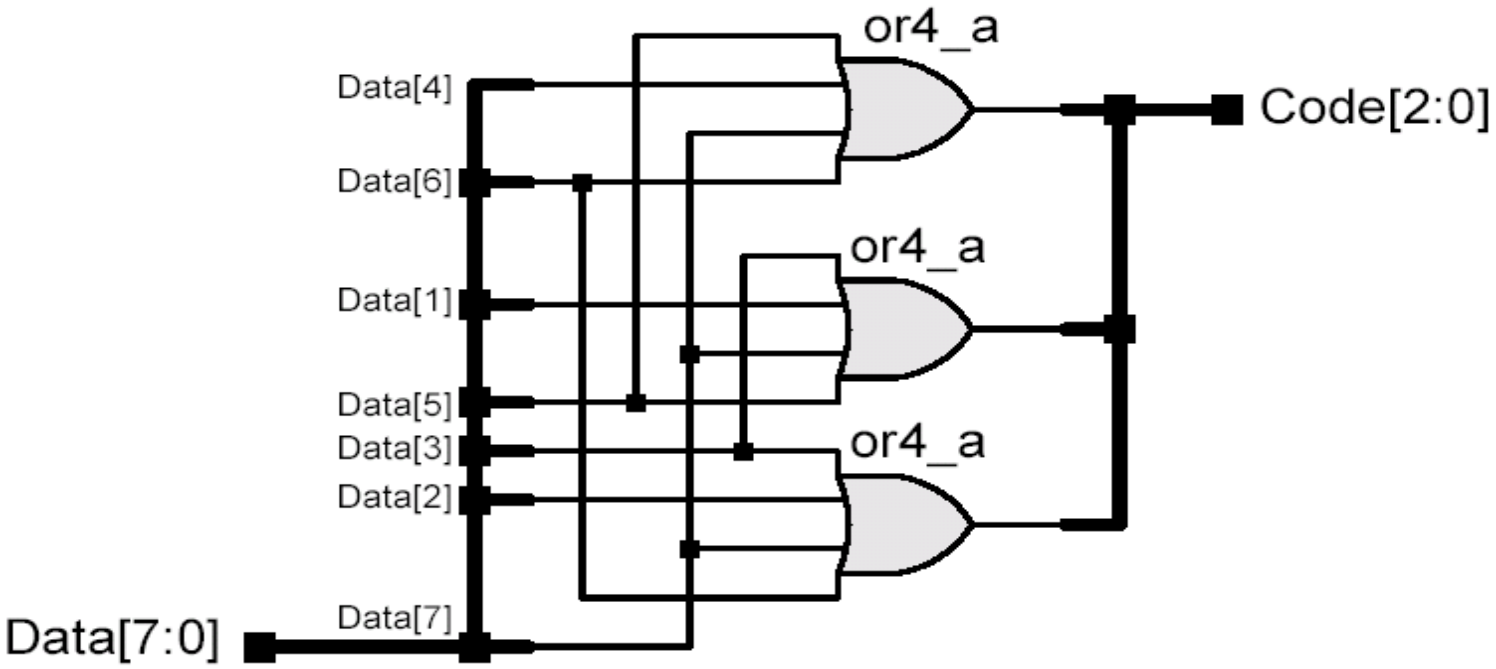
```
module encoder (Data, Code);
    input       [7:0]    Data;
    output      [2:0]    Code;
    reg   [2:0]    Code;
    always @ (Data)
      if (Data == 8'b00000001) Code = 0; else
      if (Data == 8'b00000010) Code = 1; else
      if (Data == 8'b00000100) Code = 2; else
      if (Data == 8'b00001000) Code = 3; else
      if (Data == 8'b00010000) Code = 4; else
      if (Data == 8'b00100000) Code = 5; else
      if (Data == 8'b01000000) Code = 6; else
      if (Data == 8'b10000000) Code = 7; else
                                  Code = 3'bx;
endmodule
```

```
always @ (Data)
  case (Data)
    8'b00000001 : Code = 0;
    8'b00000010 : Code = 1;
    8'b00000100 : Code = 2;
    8'b00001000 : Code = 3;
    8'b00010000 : Code = 4;
    8'b00100000 : Code = 5;
    8'b01000000 : Code = 6;
    8'b10000000 : Code = 7;
    default: Code = 3'bx;
  endcase
```

The default assignment is necessary to prevent synthesis of a latch, and the default condition will be "don't cares" in synthesis.

Synthesis Result





8:3 Priority Encoder 编码

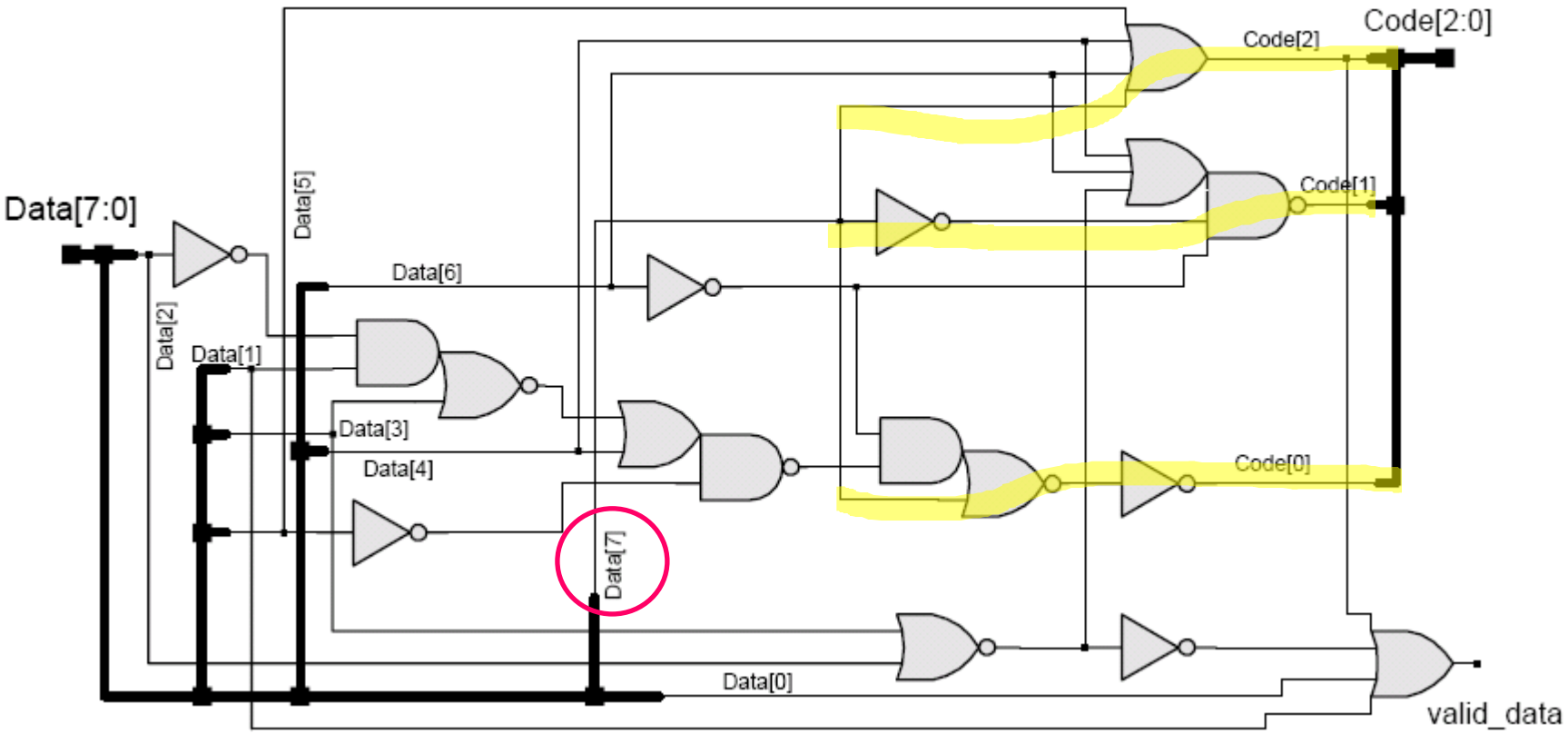
```
module priority (Data, Code, valid_data);
    input       [7:0]   Data;
    output      [2:0]   Code;
    output              valid_data;
    reg         [2:0]   Code;
    assign      valid_data = |Data;
    always @ (Data)
      if (Data[7]) Code = 7; else
      if (Data[6]) Code = 6; else
      if (Data[5]) Code = 5; else
      if (Data[4]) Code = 4; else
      if (Data[3]) Code = 3; else
      if (Data[2]) Code = 2; else
      if (Data[1]) Code = 1; else
      if (Data[0]) Code = 0; else
                   Code = 3'bx;
endmodule
```

优先高

优先低

```
always @ (Data)
  casex (Data)
    8'b1xxxxxxx : Code = 7;
    8'b01xxxxxx : Code = 6;
    8'b001xxxxx : Code = 5;
    8'b0001xxxx : Code = 4;
    8'b00001xxx : Code = 3;
    8'b000001xx : Code = 2;
    8'b0000001x : Code = 1;
    8'b00000001 : Code = 0;
    default:      Code = 3'bx;
  endcase
```

Synthesis Result

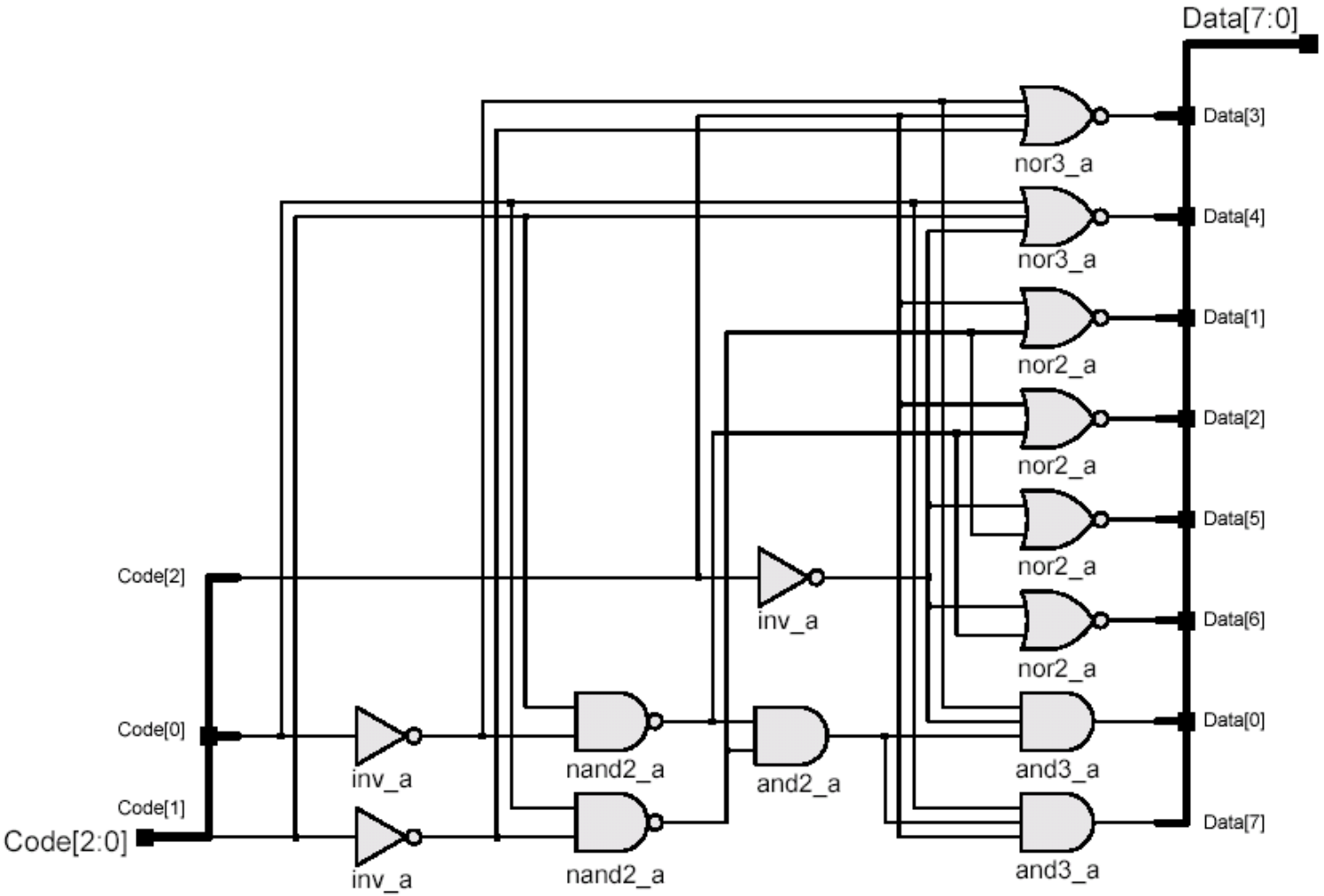


3:8 Decoder

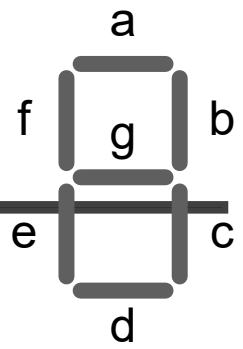
```
module decoder (Code, Data);
    input       [2:0]    Code;
    output      [7:0]    Data;
    reg         [7:0]    Data;
    always @ (Code) begin
      if (Code == 0) Data = 8'b00000001; else
      if (Code == 1) Data = 8'b00000010; else
      if (Code == 2) Data = 8'b00000100; else
      if (Code == 3) Data = 8'b00001000; else
      if (Code == 4) Data = 8'b00010000; else
      if (Code == 5) Data = 8'b00100000; else
      if (Code == 6) Data = 8'b01000000; else
      if (Code == 7) Data = 8'b10000000; else
                     Data = 8'bx;
    end
endmodule
```

```
always @ (Code)
    case (Code)
          0 : Data = 8'b00000001;
          1 : Data = 8'b00000010;
          2 : Data = 8'b00000100;
          3 : Data = 8'b00001000;
          4 : Data = 8'b00010000;
          5 : Data = 8'b00100000;
          6 : Data = 8'b01000000;
          7 : Data = 8'b10000000;
          default: Data = 8'bx;
    endcase
```

Synthesis Result



Combinational 7-Segment Display



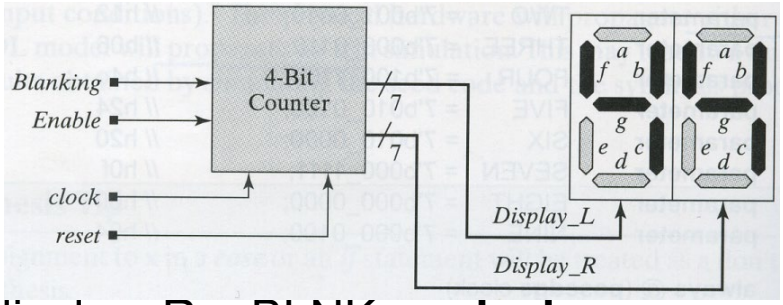

```
module sev_seg_display (data_in, data_out);
    input       [3:0]   data_in;
    output      [6:0]   data_out;
    reg         [6:0]   data_out;
    // active low output, abc_defg
    parameter BLNK      = 7'b111_1111;
    parameter ZERO      = 7'b000_0001;
    parameter ONE       = 7'b100_1111;
    parameter TWO       = 7'b001_0010;
    parameter THREE     = 7'b000_0110;
    parameter FOUR      = 7'b100_1100;
    parameter FIVE      = 7'b010_0100;
    parameter SIX       = 7'b010_0000;
    parameter SEVEN     = 7'b000_1111;
    parameter EIGHT     = 7'b000_0000;
    parameter NINE      = 7'b000_0100;
```

```
always @ (data_in)
    case (data_in)
        0: data_out = ZERO;
        1: data_out = ONE;
        2: data_out = TWO;
        3: data_out = THREE;
        4: data_out = FOUR;
        5: data_out = FIVE;
        6: data_out = SIX;
        7: data_out = SEVEN;
        8: data_out = EIGHT;
        9: data_out = NINE;
        default: data_out = BLNK;
    endcase
endmodule
```

Latched 7-Segment Display (1/2)

```
module latched_sev_seg_display
(blanking, enable, clock, reset, display_L, display_R,);
input          blanking, enable, clock, reset;
output  [6: 0] display_L, display_R;  reg [6: 0]  display_L, display_R;
reg            [3: 0]  count;
// active low output
parameter BLNK          = 7'b111_1111;
parameter ZERO          = 7'b000_0001;
parameter ONE           = 7'b100_1111;
parameter TWO           = 7'b001_0010;
parameter THREE         = 7'b000_0110;
parameter FOUR          = 7'b100_1100;
parameter FIVE          = 7'b010_0100;
parameter SIX           = 7'b010_0000;
parameter SEVEN         = 7'b000_1111;
parameter EIGHT         = 7'b000_0000;
parameter NINE          = 7'b000_0100;
```

Latched 7-Segment Display (2/2)




```
always @ (posedge clock)
    if (reset) count <= 0;
    else if (enable) count <= count +1;
always @ (count or blanking)
    if (blanking) begin display_L = BLNK; display_R = BLNK; end  else
        case (count)
        0:      begin display_L = ZERO;  display_R = ZERO; end
        2:      begin display_L = ZERO;  display_R = TWO; end
        4:      begin display_L = ZERO;  display_R = FOUR; end
        6:      begin display_L = ZERO;  display_R = SIX; end
        8:      begin display_L = ZERO;  display_R = EIGHT; end
        10:     begin display_L = ONE;   display_R = ZERO; end
        12:     begin display_L = ONE;   display_R = TWO; end
        14:     begin display_L = ONE;   display_R = FOUR; end
        // default: begin display_L = BLNK; display_R = BLNK; end
        endcase
endmodule
```

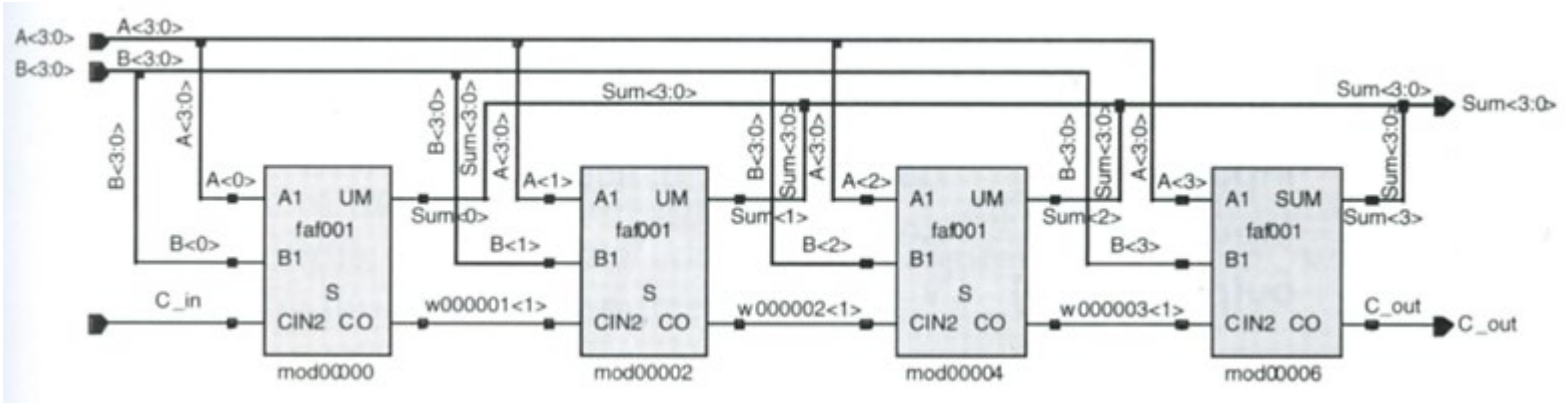
The absence of default assignments latches the output and will cause latches to be implemented by a synthesis tool.

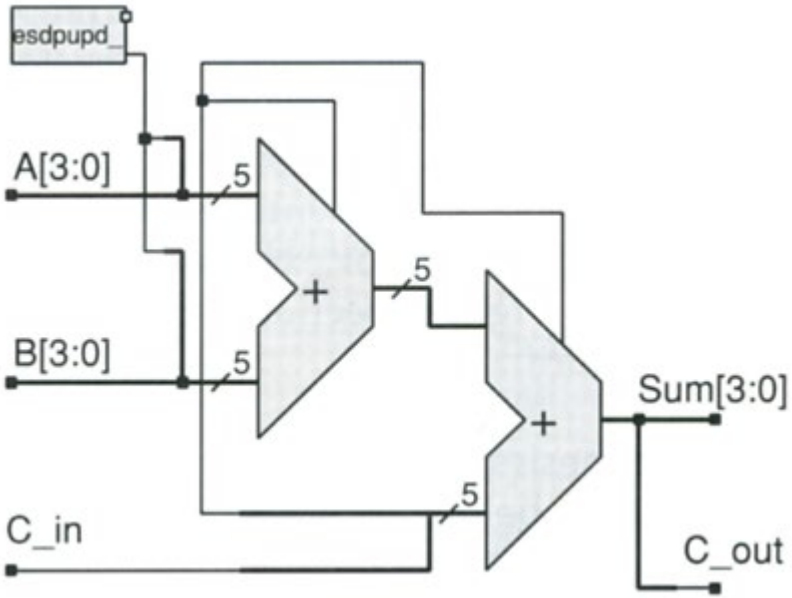
Technology Mapping and Shared Resources

```
module badd_4 (a, b, c_in, sum, c_out);
    input       [3:0]    a, b;
    input                c_in;
    output      [3:0]    sum;
    output               c_out;

    assign      {c_out, sum} = a + b + c_in;
endmodule
```

Synthesis tools include a technology mapping engine that covers the generic, optimized multi-level Boolean description of the circuit by the physical gates in a technology library.

Synthesis Results





Using full-adder cells

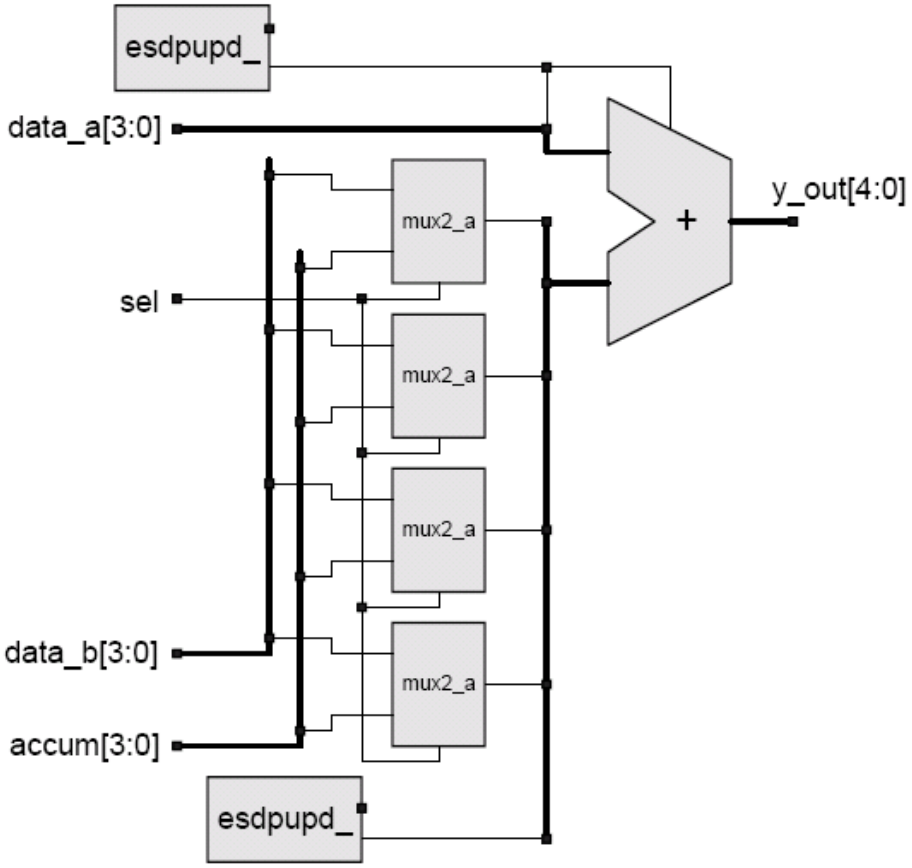
Using 5-bit adder cells depending on the speed goal

Under some specific synthesis tool and technology

Example

assign y_out = sel ? data_a + accum: data_a + data_b;

➔ **assign** y_out = data_a + (sel ? accum : data_b);



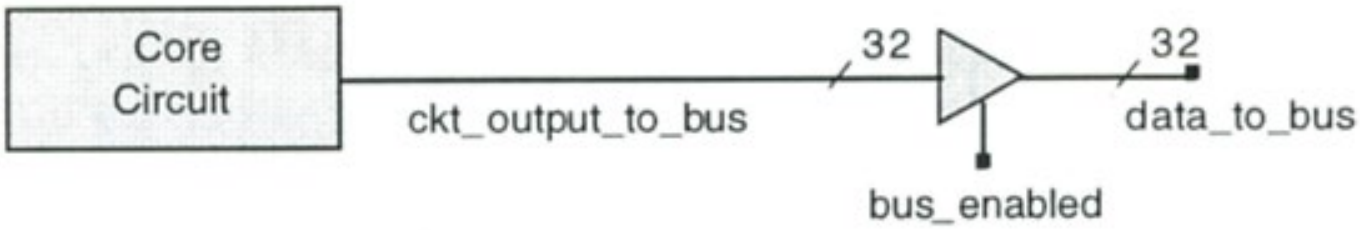
Three-State Buffers

- Three-state devices are controlled by a signal whose value determines whether an input signal is connected to the output.
- This functionality is important in physical circuits having multiple drivers, e.g., bus.

Buses

```
module stuff_to_bus1 (bus_enabled, clk, data_to_bus);
    input                bus_enabled, clk;
    output       [31:0]  data_to_bus;
    reg          [31:0]  ckt_output_to_bus;

    assign data_to_bus = (bus_enabled) ? ckt_output_to_bus : 32'bz;
    …
endmodule
```

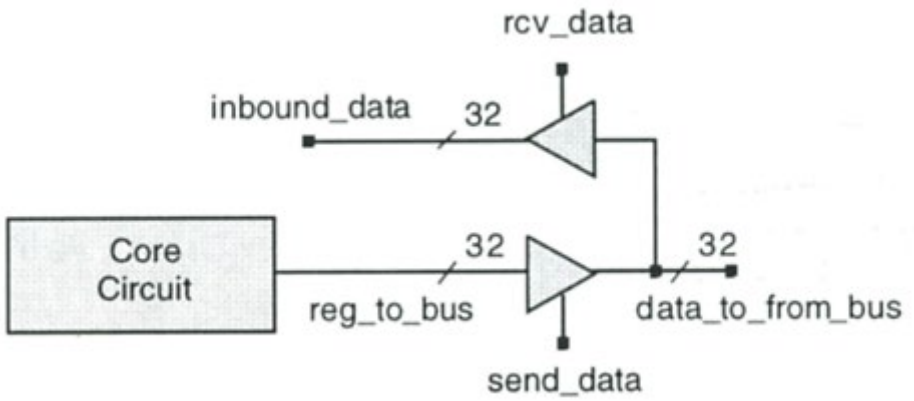
Unidirectional interface to a bus



The speed at which a bus can operate is limited by the capacitive loading placed on it.

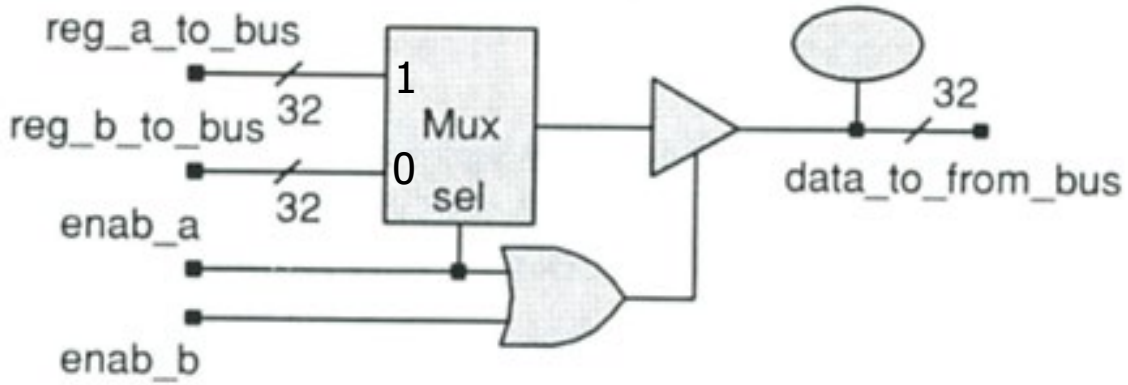
Bi-Directional Bus Drivers

```
module stuff_to_bus2 (clk, send_data, rcv_data, data_to_from_bus);
    input                clk, send_data, rcv_data;
    inout       [31:0]   data_to_from_bus;
    reg         [31:0]   reg_to_bus;
    wire        [31:0]   data_to_from_bus, inbound_data;

    assign inbound_data = (rcv_data) ? data_to_from_bus : 32'bz;
    assign data_to_from_bus = (send_data) ? reg_to_bus:
                              data_to_from_bus;
…
endmodule
```



Multiplexed Bus Drivers

```
module stuff_to_bus3 (enab_a, enab_b, clk, rcv_data,
    data_to_from_bus);
    input                enab_a, enab_b, clk, rcv_data;
    inout       [31:0]   data_to_from_bus;
    reg         [31:0]   reg_a_to_bus, reg_b_to_bus;

    assign data_to_from_bus = (enab_a) ? reg_a_to_bus : (enab_b) ?
    reg_b_to_bus : 32'bz;

    …
endmodule
```



Reference

1. 教師自製
2. Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL, Michael D. Ciletti, ISBN: 0139773983, Prentice Hall, 1999.