Introduction to Verilog HDL

Hardware Description Language (HDL)

- High-Level Programming Language
- Special constructs to model microelectronic circuits
  - Describe the operation of a circuit at various levels of abstraction
    - Behavioral
    - Functional (Register Transfer Level)
    - Structural
  - Express the concurrency of circuit operation
  - Serve as input to synthesis

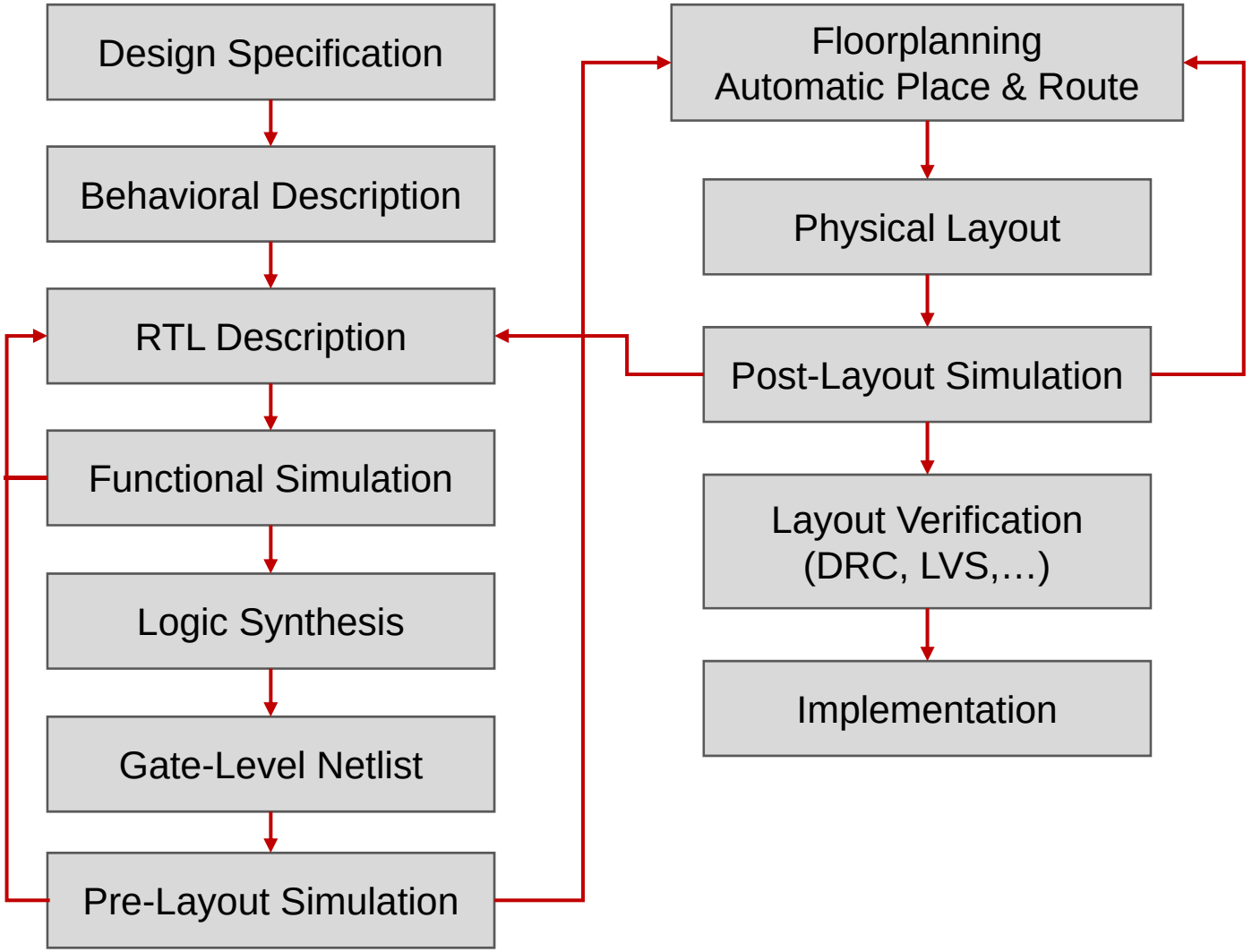
Major HDLs

- Verilog HDL
  - Started by Gateway in 1984
  - Became open to public by Cadence in 1990
  - IEEE standard 1364 in 1995
  - Slightly better at gate/transistor level
  - Language style close to C/C++
  - Pre-defined data type, easy to use
- VHDL
  - Started by VHSIC project in 1980s
  - IEEE standard 1076 in 1987, IEEE 1164 in 1993
  - Slightly better at system level
  - Language style close to Ada/Pascal
  - User-defined data type, more flexible
- Equally effective, personal preference

Why HDL

- Facilitate a top-down design methodology using synthesis
  - Design at a high implementation-independent level
  - Delay decision on implementation details
  - Easily explore design alternatives
  - Automatically map high-level description to a technology-specific implementation
- Provides greater flexibility
  - Reuse earlier design components
  - Move designs between multiple vendors and tools

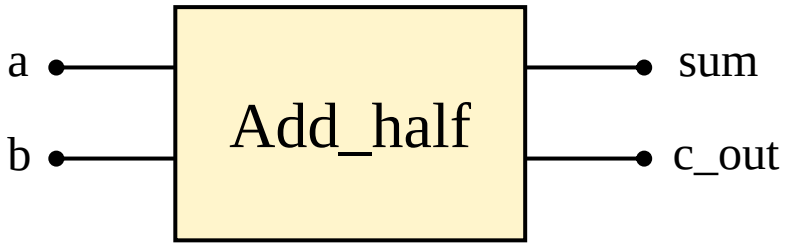
Typical Design Flow



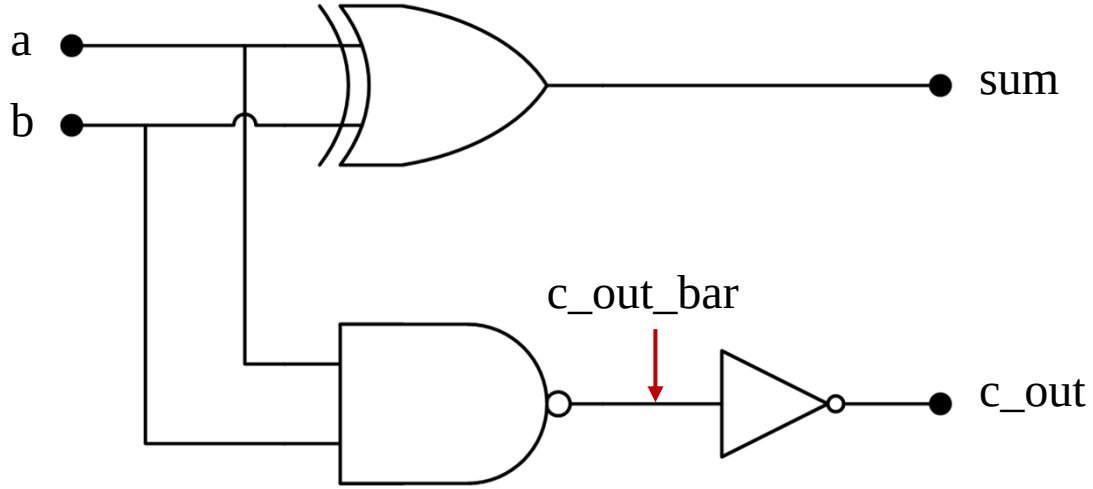
Verilog-Supported Levels of Abstraction

- Behavioral
  - Describes a system by the flow of data between its functional blocks
  - Defines signal values when they change
- Functional or Register Transfer Level (RTL)
  - Describes a system by the flow of data and control signals between and within its functional blocks
  - Defines signal values with respect to a clock
- Structural
  - Describes a system by connecting predefined components
  - Uses technology-specific, low-level components when mapping from an RTL description to a gate-level netlist, such as during synthesis.

Block Diagram & Schematic



block diagram



schematic

Modules

- A module is the basic building block in Verilog
  - Every module starts with the keyword **module**, has a name, and ends with the keyword **endmodule**
- A module can be
  - A design block
  - A simulation block, i.e., a testbench

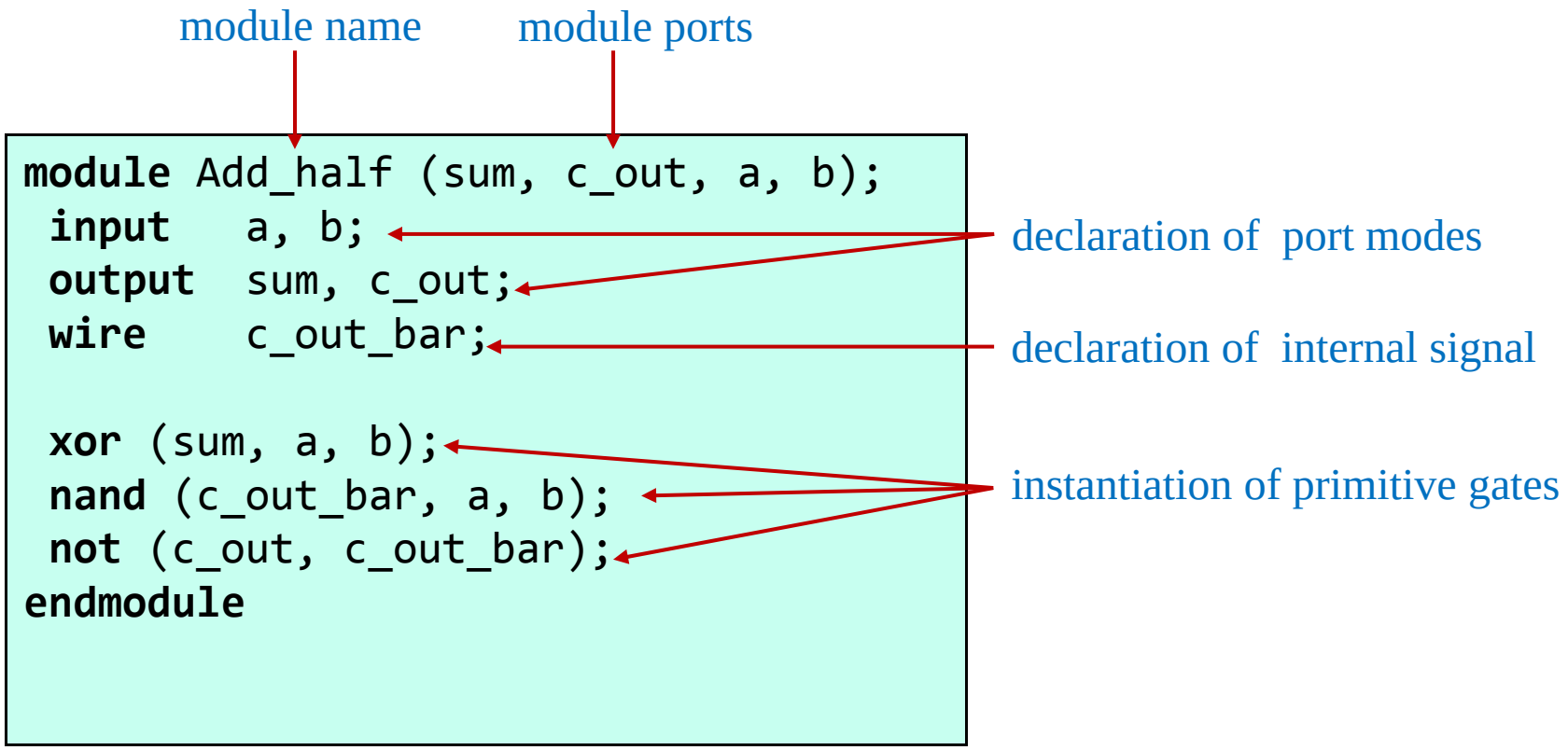
```
module add_1 (. . .);
  . . .
  . . .
endmodule
```

Module Ports

- Module ports describe the input and output terminals of a module
  - Listed in parentheses after the module name
  - Declared to be **input, output,** or **inout** (bi-directional)

```
module add_1 (sum, a, b, c_in, c_out);
output sum;
input a;
input b;
input  c_in;
output c_out;

// output sum, c_out;
// input  a, b, c_in;
 . . .
 . . .
endmodule
```

Structural Level Description

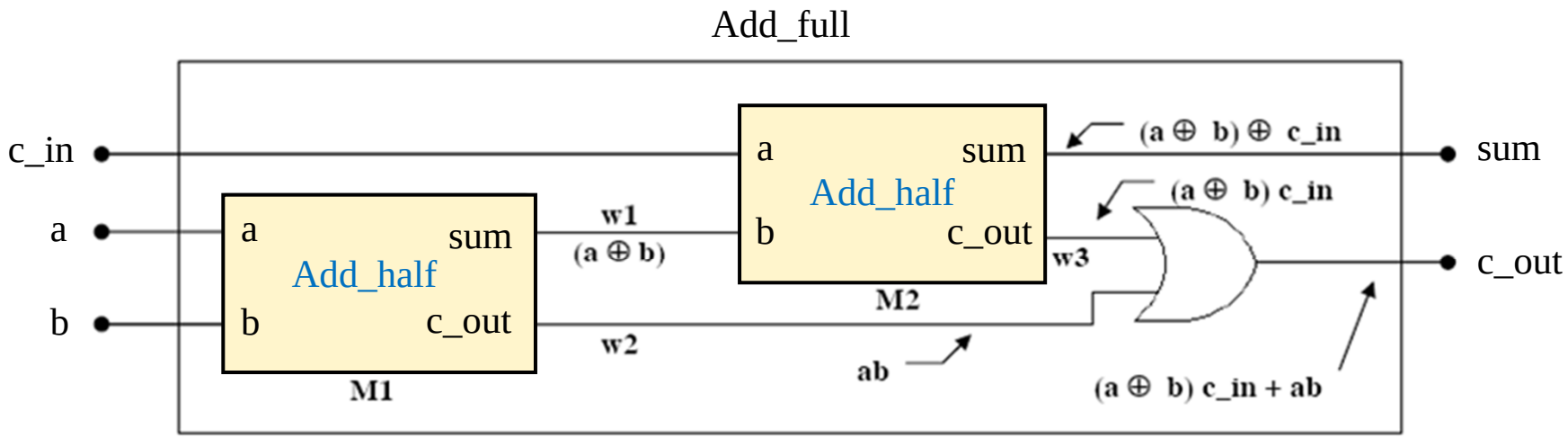


Note: All bold-faced items are Verilog keywords

Predefined Primitives

- Combinational logic gates
  - **and, nand**
  - **or, nor**
  - **xor, xnor**
  - **buf, not**

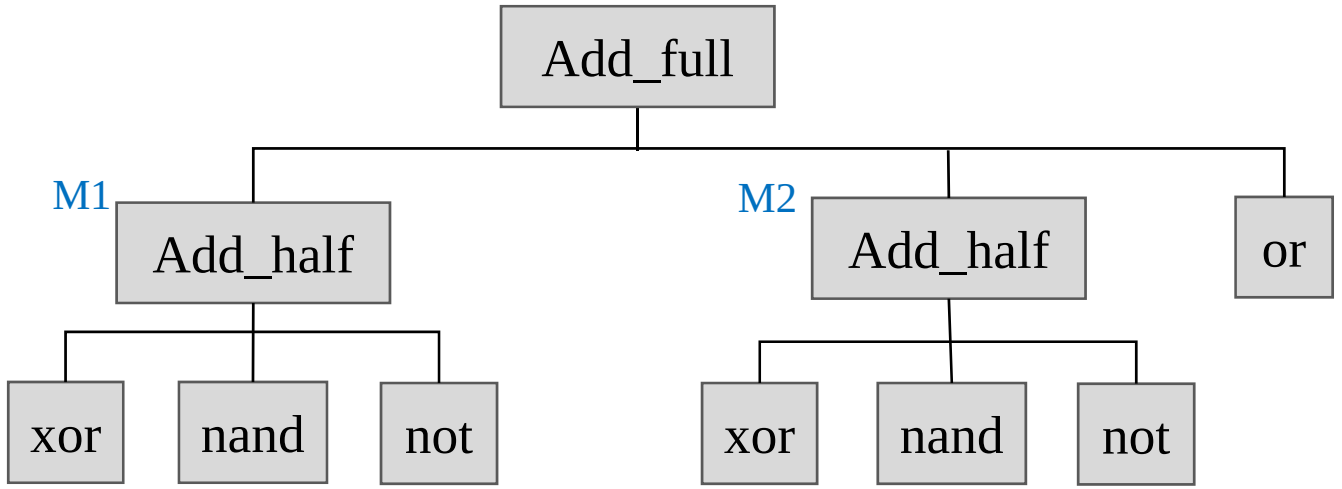
Full Adder




```
module Add_full (sum, c_out, a, b, c_in); //parent module
 input  a, b, c_in;
 output c_out, sum;
 wire w1, w2, w2;

 Add_half M1 (w1, w2, a, b);                //child module
 Add_half M2 (sum, w3, w1, c_in);           //child module
 or (c_out, w2, w3);                        //primitive instantiation
endmodule
```

module instance name

Hierarchical Description



Nested module instantiation is the mechanism for hierarchical decomposition of a design

RTL Level Description

- An RTL model provides sufficient architectural details that a synthesis tool can construct the circuit

```
module add_1 (sum, a, b, c_in, c_out);
output sum, c_out;
input  a, b, c_in;

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

endmodule
```

The Behavioral Level

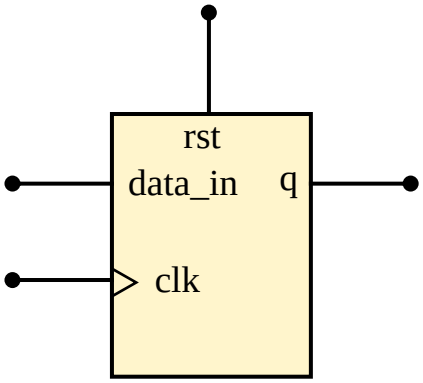
- Describes the behavior of a design without implying any specific internal architecture
  - High level constructs, such as **@, case, if, repeat, wait, while**, etc
  - Testbench
  - Limited support by synthesis tools
- The difference between a behavioral model and an RTL model and an RTL model is not always clear
  - whether synthesizable or not

Behavioral Level Description

```
module Add_half (sum, c_out, a, b);
input  a, b;
output sum, c_out;
reg    sum, c_out;

always@(a or b)
  begin
    sum = a ^ b;        //exclusive or
    c_out = a & b;      //and
  end
endmodule
```

procedural statements

Behavioral Description of D F/F



block diagram

```
module Flip_flop (q, data_in, clk, rst);
  input  data_in, clk, rst;
  output q;
  reg q;

always @ (posedge clk)
  begin
    if(rst == 1) q=0;
    else q = data_in;
  end
endmodule
```

declaration of synchronous behavior

Variables

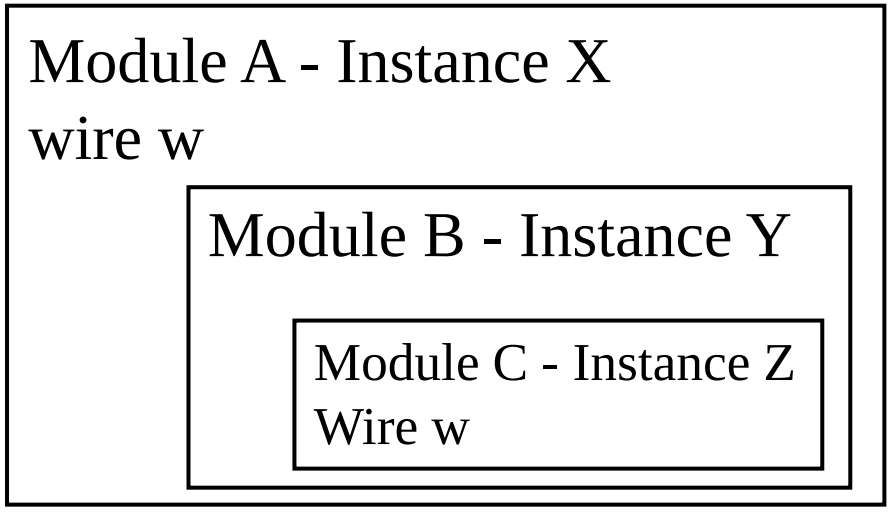
- Nets “wire”: structural connectivity
- Registers “reg”: abstraction of storage (may or may not be physical storage)
- Both nets and registers are informally called signals, and may be either scalar or vector

Logic Values

- Verilog signal values
  - 0: logical 0, or a FALSE condition
  - 1: logical 1, or a TRUE condition
  - x: an unknown value
  - z: a high impedance condition

Hierarchical De-Referencing

- To reference a variable defined inside an instantiated module
- Supported by a variable's hierarchical path name
  - X.w
  - X.Y.Z.w



if…else

- **else** is paired with nearest **if** when ambiguous (use **begin**…**end** in nesting to clarify)

(a)
```
if (a < b) c = d + 1;
```

(b)
```
if (a < b);
```

(c)
```
if (k == 1)
      begin : A_block
       sum_out = sum_reg;
       c_out = c_reg;
      end
```

(d)
```
if (a < b)
      sum = sum + 1;
else
      sum = sum + 2;
```

Example

```
module mux4_PCA (a, b, c, d, select, y_out);
  input       a, b, c, d;
  input [1:0] select;
  output      y_out;
  reg         y_out;

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

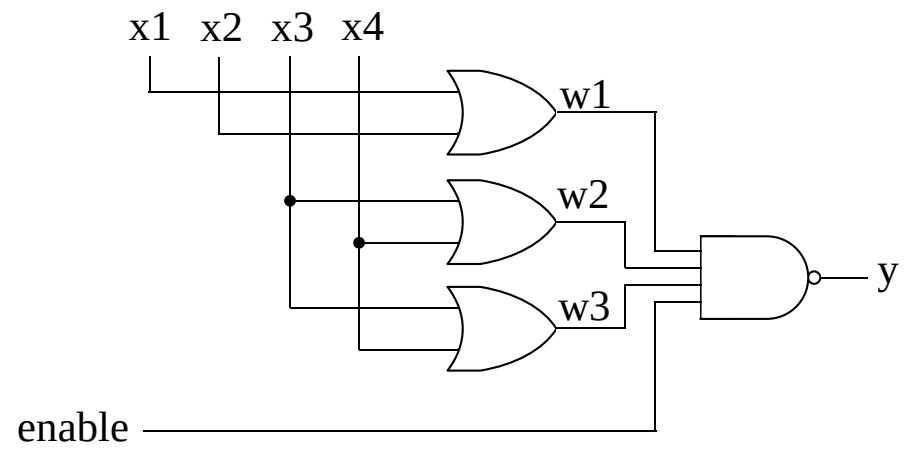
case

- Require complete bitwise match
- Example:

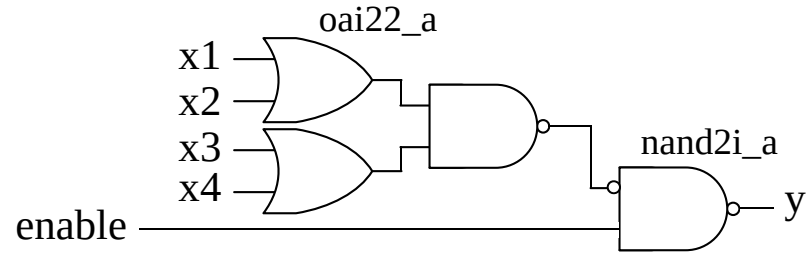
```
…
 always @ (a or b or c or d or
select)
  begin
    case (select)
      0: y = a;
      1: y = b;
      2: y = c;
      3: y = d;
      default: y = 1'bx;
    endcase
  end
…
```

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
```



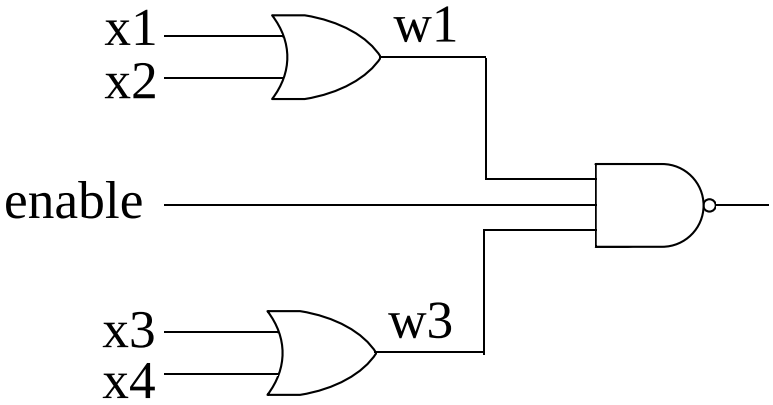
(a) Pre-optimized circuit

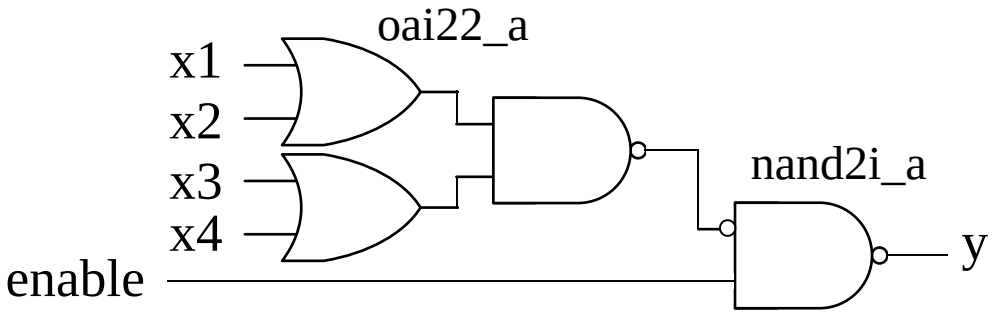


(b) Circuit synthesized from the circuit in (a)

Continuous Assignment

```
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
```

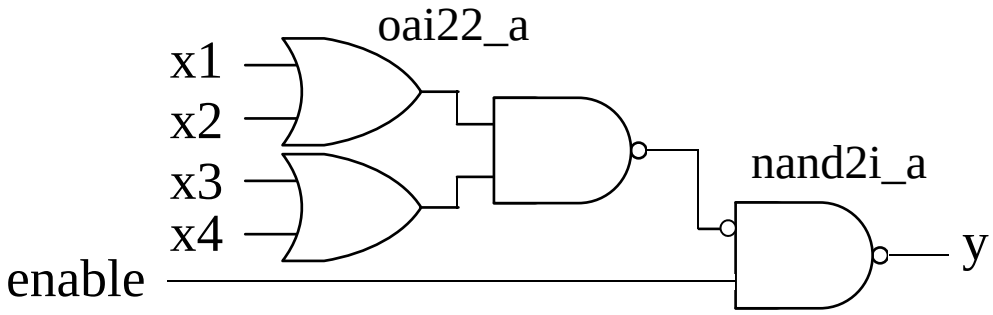




Synthesis result

Behavioral Description

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

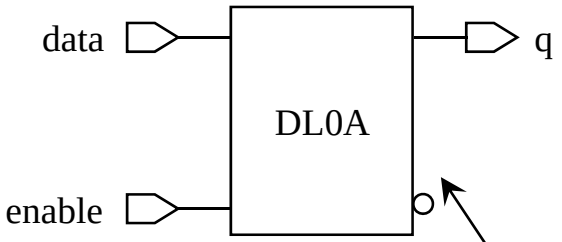


Circuit synthesized from a behavioral description

Avoid Latch Inference (1/2)

- When if or case statements are used without specifying outputs in all possible conditions, a latch will be created

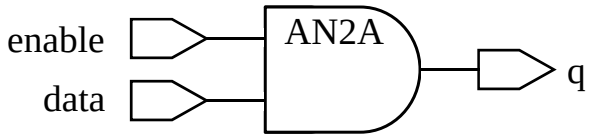
```
always@(enable or data)
if(enable)
   q = data;
```



Latch inferred

```
always@(enable or data)
begin
   if(enable)
      q = data;
   else
      q = 0;
end
```

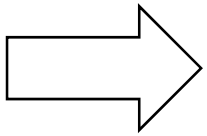
Specify all output values



Avoid Latch Inference (2/2)

```
always@(a or b or c)
case(a)
   2'b11 : e = b;
   2'b10 : e = ~c;
endcase
```

No latch



```
always@(a or b or c)
case(a)
   2'b11 : e = b;
   2'b10 : e = ~c;
   default : e = 0;
endcase
```

Add a default statement

result in a latch

Code Converter (1/3)

- A code converter transforms one representation of data to another
- Ex: A BCD to excess-3 code converter
  - BCD: Binary Coded Decimal
  - Excess-3 code: the decimal digit plus 3

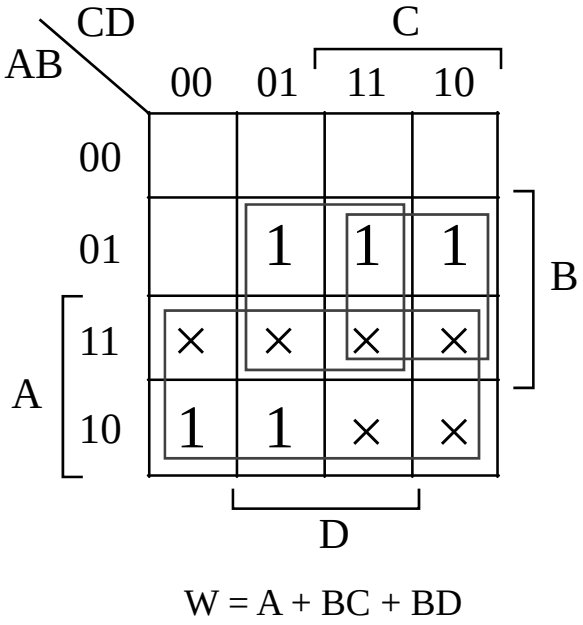
Truth Table for Code Converter Example

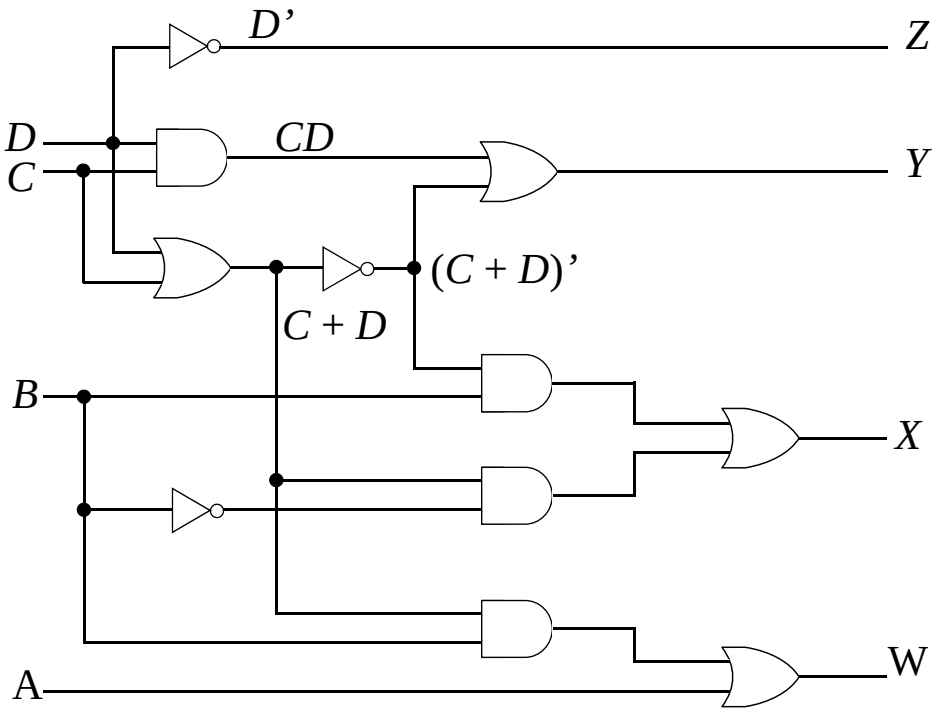
Input BCD				Output Excess-3			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Code Converter (2/3)

- Equations: (share terms to minimize cost)

$$W = A + BC + BD = A + B(C + D)$$
$$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D} = \overline{B}(C + D) + B\overline{C}\,\overline{D}$$
$$Y = CD + \overline{C}\,\overline{D} = \overline{C \oplus D}$$
$$Z = \overline{D}$$





Code Converter (3/3)

- **RTL style**

```
assign W = A|(B&(C|D));
assign X = ~B&(C|D)|(B&~C&~D);
assign Y = ~(C^D);
assign Z = ~D;
```

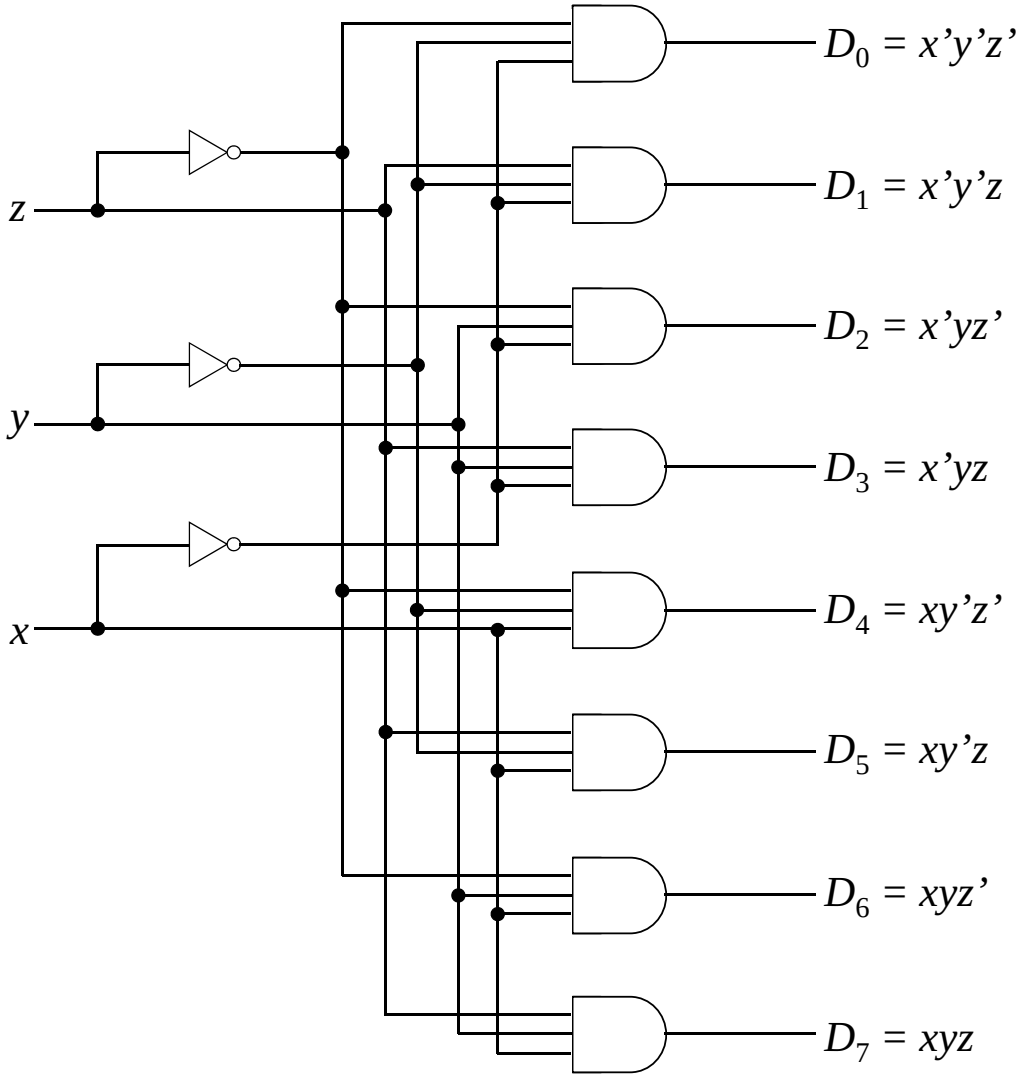
- **Behavior style**

```
assign ROM_in = {A, B, C, D};
assign {W, X, Y, Z} = ROM_out;
always @(ROM_in) begin
  case (ROM_in)
   4‘b0000: ROM_out = 4‘b0011;
   4‘b0001: ROM_out = 4‘b0100;
   …
   4‘b1001: ROM_out = 4‘b1100;
   default: ROM_out = 4‘b0000;
  endcase
end
```

Decoder (1/2)

- A decoder is to generate the 2^n (or fewer) minterms of n input variables
- Ex: a 3-to-8 line decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Decoder (2/2)

- **Behavior style 1**

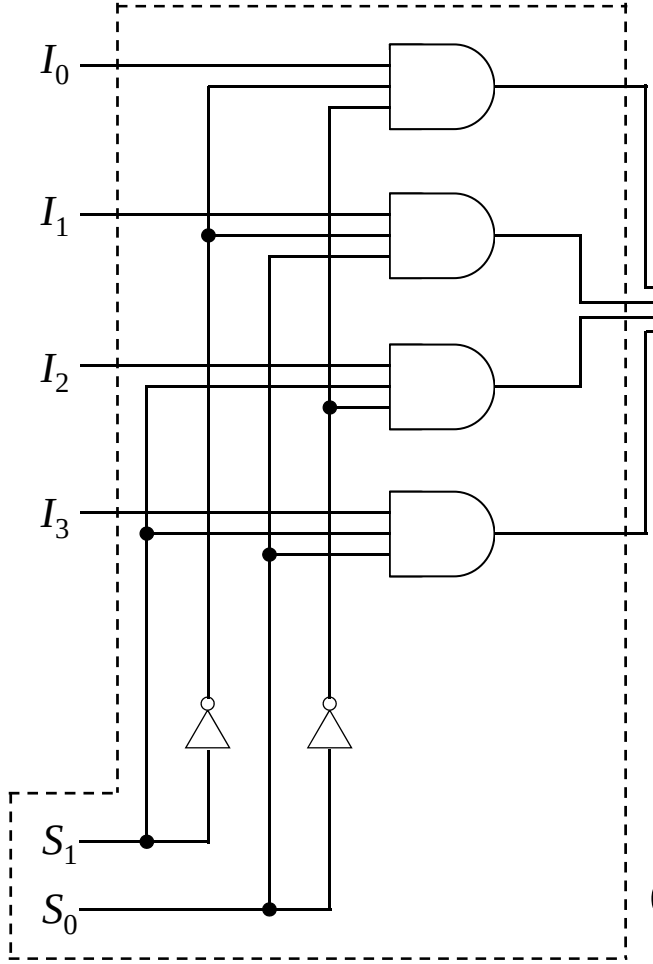
```
input x, y, z;
output [7:0] D;
reg [7:0] D;
always @(x or y or z) begin
  case ({x, y, z})
   3‘b000: D = 8‘b00000001;
   3‘b001: D = 8‘b00000010;
   …
   3‘b111: D = 8‘b10000000;
   default: D = 8‘b0;
  endcase
end
```

- **Behavior style 2**

```
input x, y, z;
output [7:0] D;
wire [2:0] addr;
reg [7:0] D;
assign addr = {x, y, z};
always @(addr) begin
   D = 8‘b0;
   D[addr] = 1;
end
```

Multiplexer (1/2)

- A multiplexer uses n selection bits to choose binary info. from a maximum of 2^n unique input lines



(a) Logic diagram

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

(b) Function table

decoder !!

Multiplexer (2/2)

- **Behavior style 1**

```
input [1:0] S;
input [3:0] I;
output Y;
reg Y;
always @(S or I) begin
  case (S)
  2‘b00: Y = I[0];
  2‘b01: Y = I[1];
  2‘b10: Y = I[2];
  2‘b11: Y = I[3];
  default: Y = 0;
  endcase
end
```

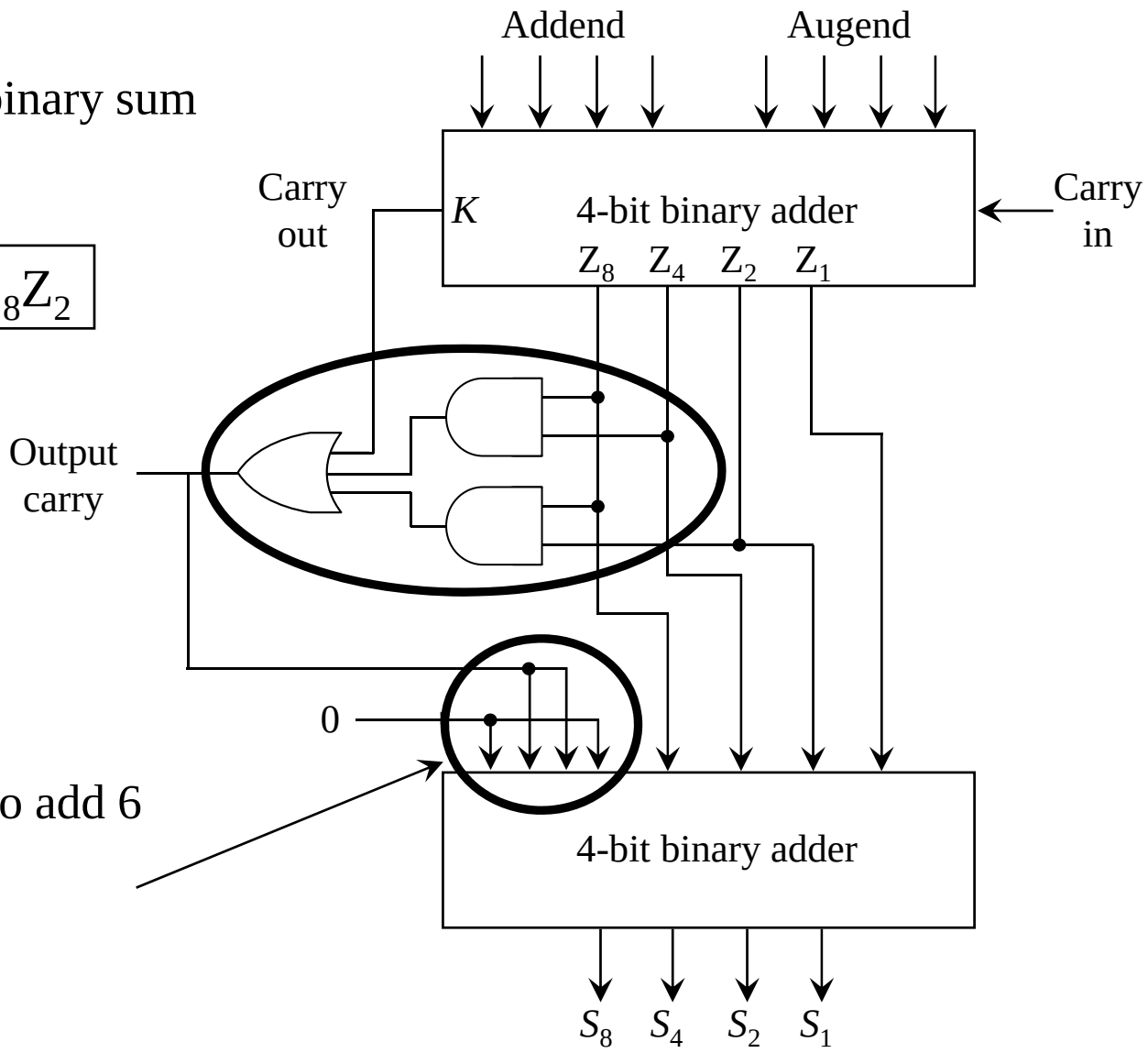
- **Behavior style 2**

```
input [1:0] S;
input [3:0] I;
output Y;
reg Y;
always @(S or I)
  begin
  Y = I[S];
end
```

BCD Adder (1/2)

C detects whether the binary sum is greater than 9 (1001)

$$C = K + Z_8Z_4 + Z_8Z_2$$



If C=1, it is necessary to add 6 (0110) to binary sum

BCD Adder (2/2)

```
module bcd_add (S, Cout, A, B, Cin);
   input [3:0] A, B;
   input Cin;
   output [3:0] S;
   output Cout;
   reg [3:0] S;
   reg Cout;

   always @(A or B or Cin) begin
      {Cout, S} = A + B + Cin;
      if (Cout != 0 || S > 9) begin
       S = S + 6;
       Cout = 1;
      end
   end
endmodule
```

Latch Inference

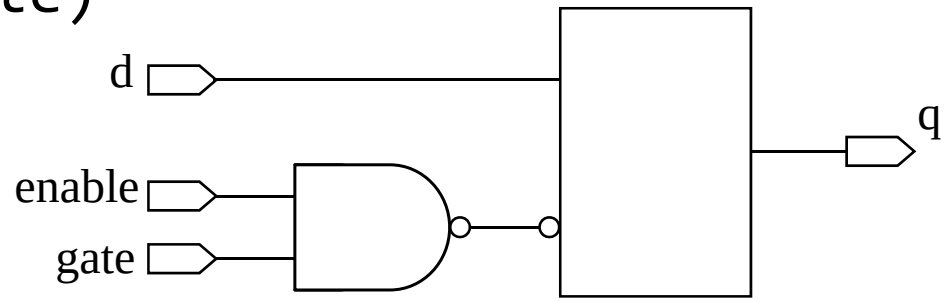
- **Incompletely specified wire in the synchronous section**
- **D latch**

```
always @(enable or data)
    if (enable)
     q = data;
```

More Latches

- **D latch with gated enable**

```
always @(enable or d or gate)
    if (enable & gate)
      q = d;
```



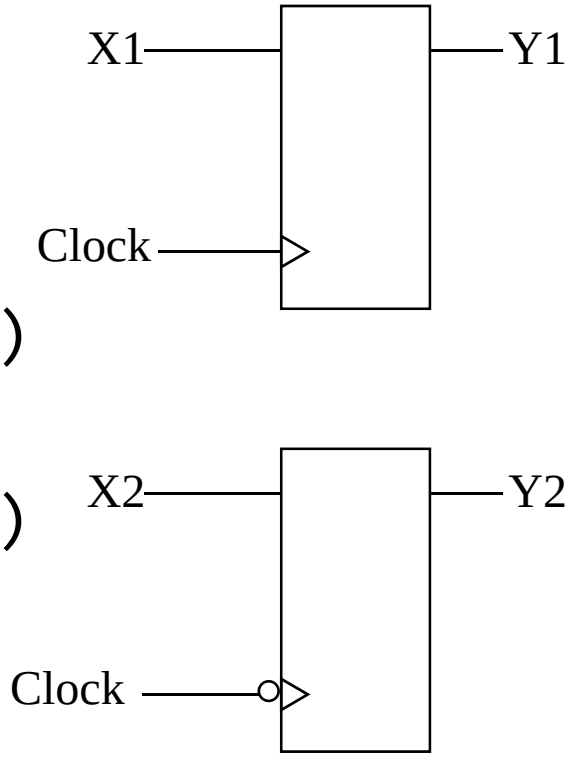
- **D latch with asynchronous reset**

```
always @(reset or data or enable)
    if (reset)
      q = 1'b0;
    else if (enable)
      q = data;
```

Flip-Flop Inference

- **Wire (port) assigned in the synchronous section**

```
module FF_PN (Clock, X1, X2, Y1, Y2);
    input Clock;
    input X1, X2;
    output Y1, Y2;
    reg Y1, Y2;
    always @(posedge Clock)
        Y1 = X1;
    always @(negedge Clock)
        Y2 = X2;
endmodule
```



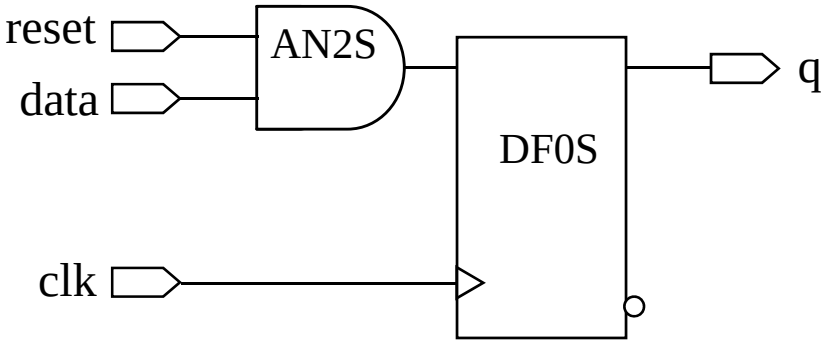
Modeling Reset

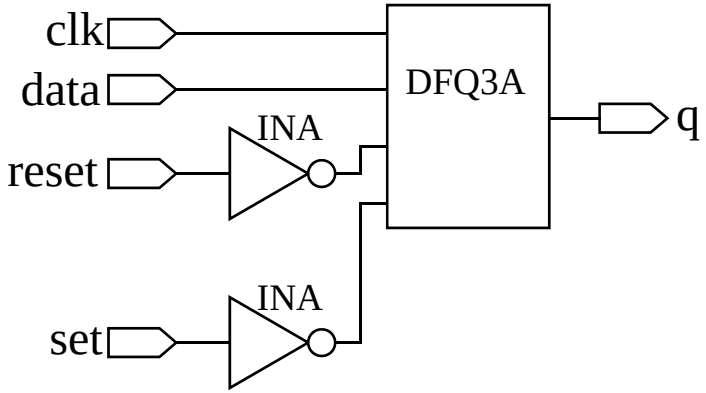
Synchronous reset

```
always@(posedge clk)
 if(!reset) q=1'b0;
 else q=data;
```

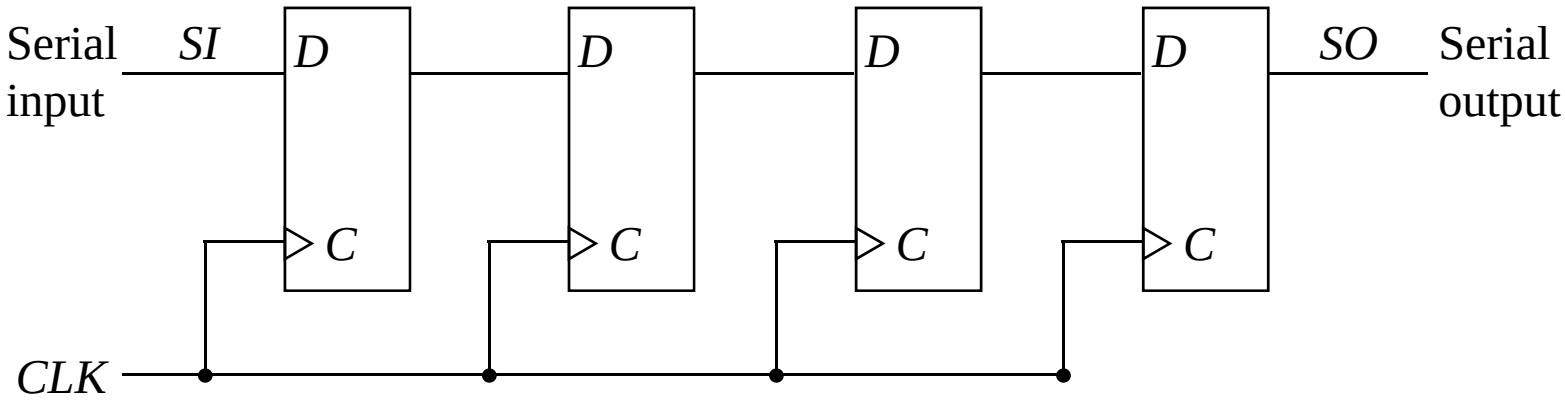
Asynchronous reset

```
always@(posedge clk or negedge reset)
 if(!reset) q=1'b0;
 else q=data;
```

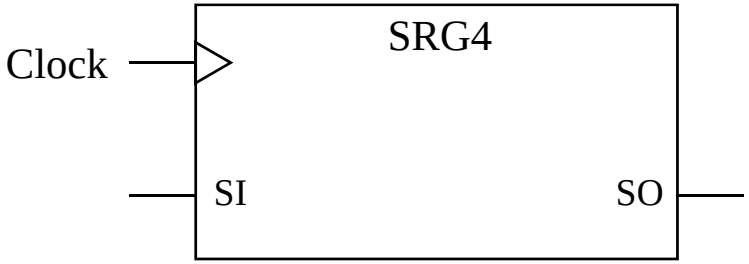




The Simplest Shift Register



4-Bit Shift Register



Symbol

Procedural Assignment

- Assign value to registers
- Blocking procedural assignment
  - Use “=”
  - An assignment is completed before the next assignment starts
- Non-blocking procedural assignment
  - Use “<=”
  - Assignments are executed in parallel

Examples

```
a = 1;
b = 0;
…
a <= b;          // use b=0
b <= a;          // use a=1
```

```
a = 1;
b = 0;
…
b <= a;          // use a=1
a <= b;          // use b=0
```

```
a = 1;
b = 0;
…
a = b;           // use b=0
b = a;           // use a=0
```

```
a = 1;
b = 0;
…
b = a;           // use a=1
a = b;           // use b=1
```

HDL Modeling for Shift Reg.

- **Blocking assignment**

```
assign SO = D;
always @(posedge CLK) begin
 if (reset) begin
      A=0; B=0; C=0; D=0;
 end
 else if (shift) begin
      D = C;
      C = B;
      B = A;
      A = SI;
 end
end
```

order dependent

- **Non-blocking assignment**

```
assign SO = D;
always @(posedge CLK)begin
 if (reset) begin
      A<=0; B<=0; C<=0; D<=0;
 end
 else if (shift) begin
      A <= SI;
      C <= B;
      D <= C;
      B <= A;
 end
end
```

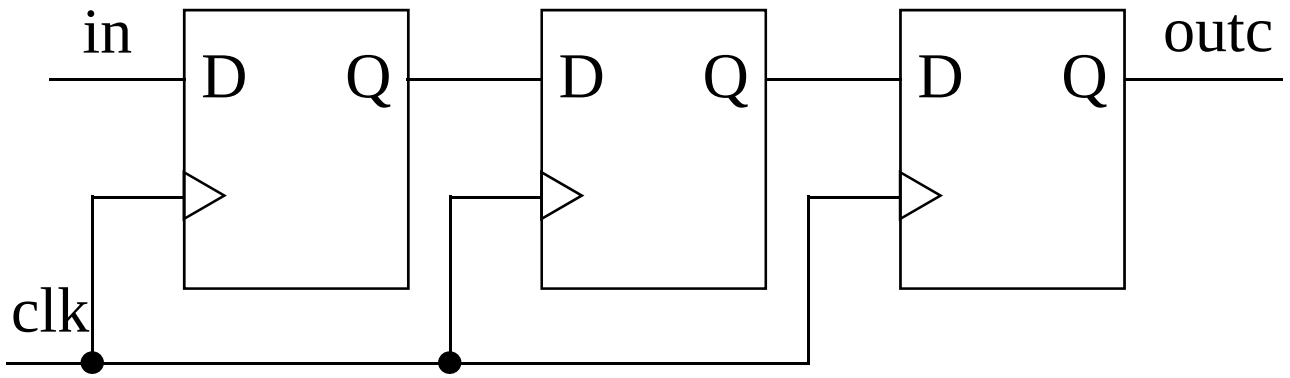
can be any order

Non-Blocking Assignment

```
always@(posedge clk)
begin
   outa<=in;
   outb<=outa;
   outc<=outb;
end
```

=

```
always@(posedge clk)
   outa=in;
always@(posedge clk)
   outb=outa;
always@(posedge clk)
   outc=outb;
```



References

- T.-C. Wang, course slides of “HDL and Synthesis”, Dept. of CS, NTHU.
- C.-N. Liu, course slides of “Digital System Design”, Dept. of EE. NCU.