System I

Combinational Logic Design

Haifeng Liu

Zhejiang University

Overview

- Introduction to Verilog HDL
- About combinational logic circuits
- Some classic/basic designs
- Timing analysis

Overview

- Introduction to Verilog HDL
- About combinational logic circuits
- Some classic/basic designs
- Timing analysis

Introduction to Verilog HDL

- Background
- HDL-based design flow
- Verilog HDL

Background

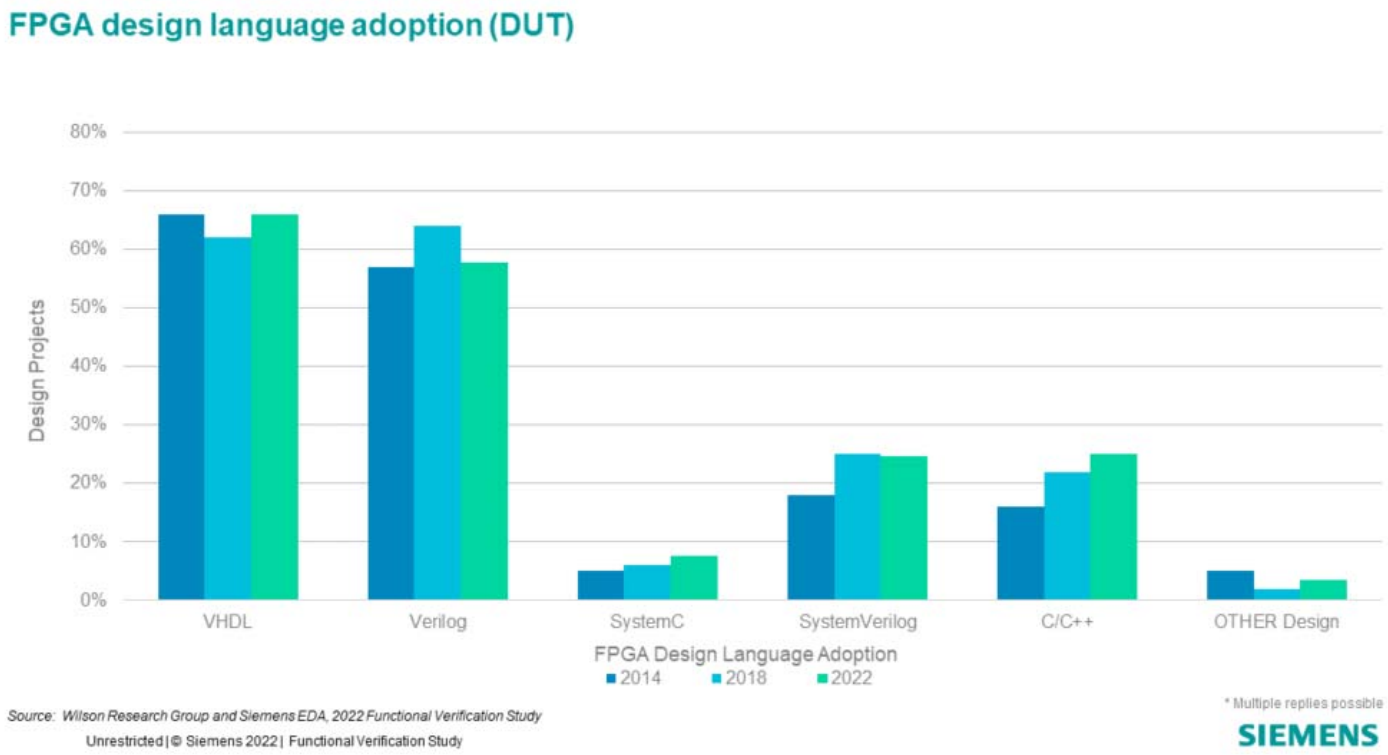
- Heterogeneous computing
  - CPU vs. GPU vs. FPGA vs. ASIC

	CPU	GPU	FPGA	ASIC
Compute Adaptability (to a variety of situations)	High	Medium	Low	None
Compute power	Medium	High	High	Medium
Latency	Medium	High	Low	Ultra low
Throughput	Low	High	High	High
Parallelism	Low	High	High	High
Power efficiency	Medium	Low	Medium	High

- Digital circuit design
  - FPGA vs. ASIC

What is HDL?

- Hardware description language is a language that uses formal methods to describe digital circuits and design digital logic systems.



HDL-based Design Flow

- Logic design with HDL
- Simulation
- Synthesis
- Physical Design
- Final step

Logic design with HDL

- Define a module

```
module top(                    Module definition start
    input a,                   Port definition
    input b,
    output c
);
assign c = a &b;               Assignment statement
endmodule                      Module definition end
```

Simulation

- Testbench

```
module sim_top();
reg a_in,b_in;
wire c_out;
top dut(
        .a(a_in),
        .b(b_in),
        .c(c_out)
);
```

```
initial begin
        a_in = 1'b0;
        b_in = 1'b0;
        #2
        b_in = 1'b1;
        #2
        a_in = 1'b1;
        #2
          $finish;
end

always @(*)begin
        $display("a=%d,b=%d,c=%d, a_in,b_in,c_out);
endmodule
```

Synthesis

- Parsing
- Multi-level synthesis
  - Non-synthesizable HDL code
- Technology mapping

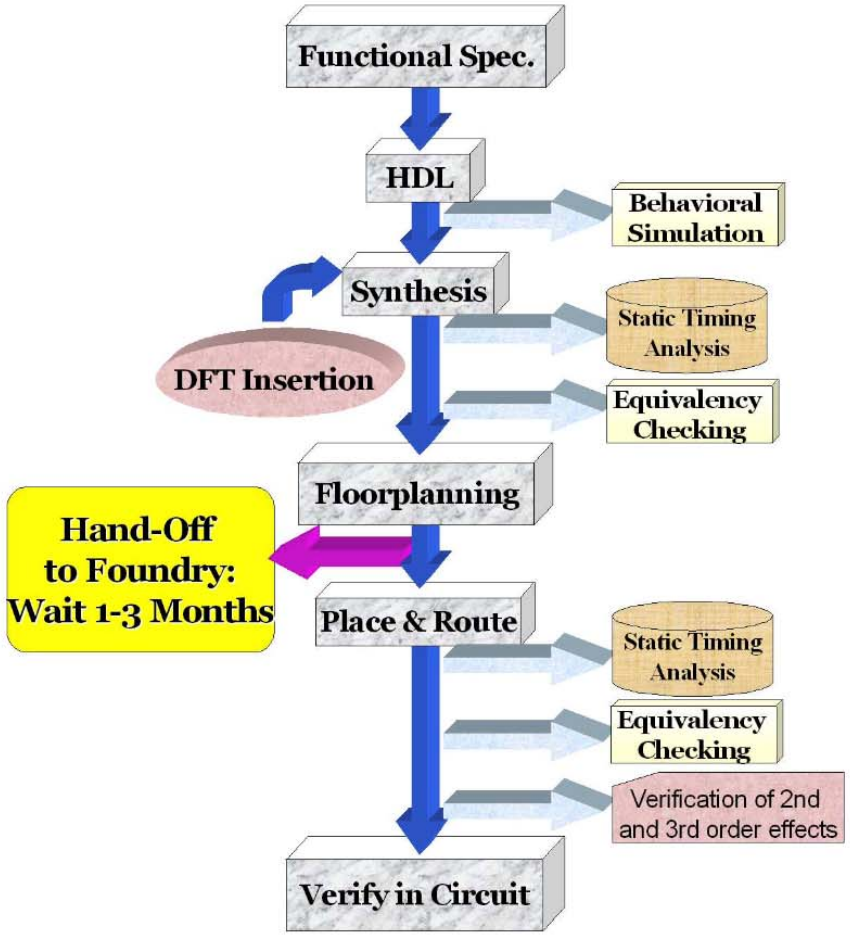
Physical Design

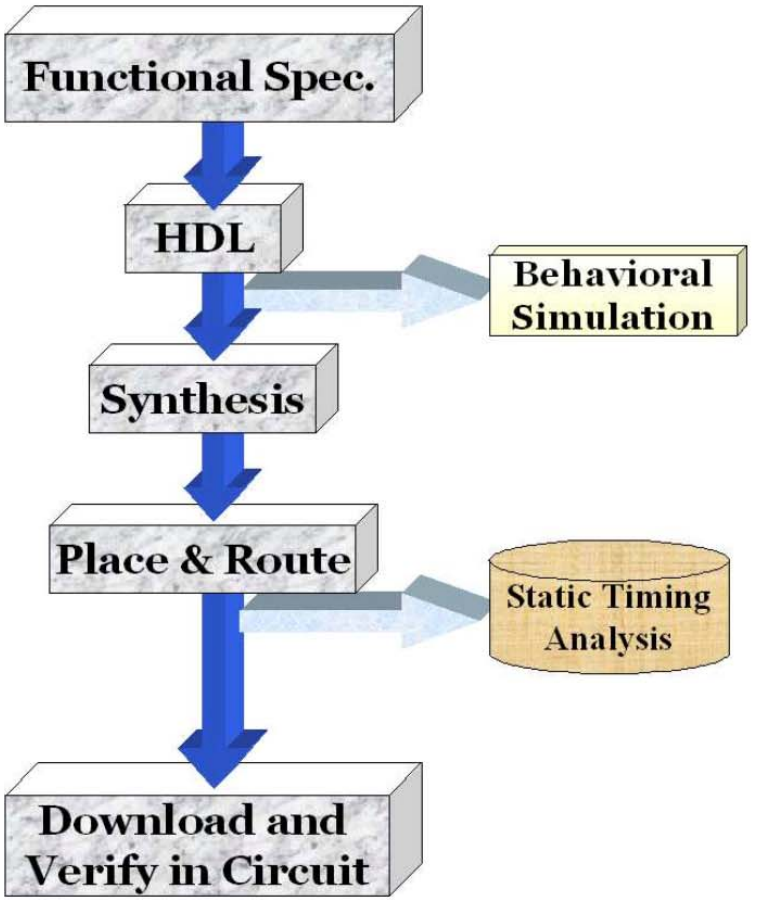
- Placement
- Routing
- Static timing analysis
- LVS and DRC
- Generating result
  - ASIC: layout file
  - FPGA: bitstream file

Final Step

- ASIC
  - Tape-out
- FPGA
  - Download bitstream file

Design Flow: ASIC vs. FPGA





https://www.xilinx.com/video/fpga/fpga-vs-asic-design-flow.html

Verilog HDL

- Lexical conventions
- Basic syntax
- Modeling methods

Lexical Conventions

- Very similar to C
  - Verilog is case-sensitive
  - All keywords are in lowercase
- A Verilog program is a string of tokens
  - Whitespace
  - Comments
  - Delimiters
  - Numbers
  - Strings
  - Identifiers
  - Keywords

Lexical Conventions (cont'd)

- Whitespace
  - Blank space (\b)
  - Tab (\t)
  - Newline (\n)
- Whitespace is ignored in Verilog except
  - In strings
  - When separating tokens
- Comments
  - Used for readability and documentation
  - Just like C:
    - // single line comment
    - /* multi-line comment */
- /* Nested comments /* like this */ may not be acceptable (depends on Verilog compiler) */

Lexical Conventions (cont'd)

- Number Specification
  - Sized numbers
  - Unsized numbers
  - Unknown and high-impedance values
  - Negative numbers

Lexical Conventions (cont'd)

- Sized numbers
  - General syntax: <size>'<base><number>
    - <size> number of bits (in decimal)
    - <number> is the number in radix <base>
    - <base> :
      - d or D for decimal (radix 10)
      - b or B for binary (radix 2)
      - o or O for octal (radix 8)
      - h or H for hexadecimal (radix 16)
    - Examples:
      - 4'b1111
      - 12'habc
      - 16'd255
- Unsized numbers
  - Default base is decimal
  - Default size is at least 32 (depends on Verilog compiler)
  - Examples
    - 23232
    - 'habc
    - 'o234

Lexical Conventions (cont'd)

- X or Z values
  - Unknown value: lowercase x
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - High-impedance value: lowercase z
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - Examples
    - 12'h13x, 6'hx, 32'bz
  - Extending the most-significant part
    - Applied when <size> is bigger than the specified value
      - Filled with x if the specified MSB is x
      - Filled with z if the specified MSB is z
      - Zero-extended otherwise
    - Examples:
      - 6'hx

Lexical Conventions (cont'd)

- Negative numbers
  - Put the sign before the <size>
  - Examples:
    - -6'd3
    - 4'd-2 // illegal
  - Two's complement is used to store the value
- Underscore character and question marks
  - Use '_' to improve readability
    - 12'b1111_0000_1010
    - Not allowed as the first character
  - '?' is the same as 'z' (only regarding numbers)
    - 4'b10?? // the same as 4'b10zz

Lexical Conventions (cont'd)

- Strings
  - As in C, use double-quotes
  - Examples:
    - "Hello world!"
    - "a / b"
    - "text\tcolumn1\bcolumn2\n"
- Identifiers and keywords
  - identifiers: alphanumeric characters, '_', and '$'
    - Should start with an alphabetic character or '_'
    - Only system tasks can start with '$'
  - Keywords: identifiers reserved by Verilog
  - Examples:
    - reg value;
    - input clk;

Lexical Conventions (cont'd)

- Escaped identifiers
  - Start with '\'
  - End with whitespace (space, tab, newline)
  - Can have any printable character between start and end
  - The '\' and whitespace are not part of the identifier
  - Examples:
    - \a+b-c // a+b-c is the identifier
    - **my_name** // **my_name** is the identifier
  - Used as name of modules

Basic Syntax: Module, Port and Instantiate

- Module and port

module model-name (port list);

<Port definition> - optional

Data type definition

Logic function description

endmodule

- Instantiate

model-name instance-identifier (port related list);

```
module top(
        input a,
        input b,
        output c
);
assign c = a & b;
endmodule
```

```
module sim_top();
reg a_in,b_in;
wire c_out;
top dut(
        .a(a_in),
        .b(b_in),
        .c(c_out)
);
initial begin
        a_in = 1'b0;
        b_in = 1'b0;
        #2
        b_in = 1'b1;
        #2
        a_in = 1'b1;
        #2
        $finish;
end
always @(*) begin
        $display("a=%d,b=%d,c=%d,
a_in,b_in,c_out);
end
endmodule
```

Basic Syntax: Data Type

- Net
  - Physical connection (wire type)
- Register
  - Storage element (reg type)
- Vectors and Arrays
  - Vector: multi-bit signal
    - E.g., wire[7:0] a; reg[31:0] rdata, wdata;
  - Array:
    - E.g., wire b[2:0] or reg[15:0] mem [1023:0]
- Parameter
  - Local scope (the current module)
  - Usually defined as constant

Basic Syntax: Data Type (cont'd)

- Module's port type
  - Input
    - Definition: wire
    - Instantiate: wire or reg
  - Output
    - Definition: wire or reg
    - Instantiate: wire

```
module Right (
        input in1,
        input wire in2,
        input [3:0] in3,
        output out1,
        output [3:0] out2,
        output reg [1:0] out3
);
…
endmodule


module Wrong (
        input reg in1,
        output out1 [2:0]
);
…
endmodule
```

Basic Syntax: Operator and Precedence

- Unary operators:
  - + -
- Bitwise operators
  - ~ & | ^ ^~
- Arithmetic operators
  - + - * / %
- Reduction operators
  - & ~& | |~ ^ ^~
- Logic operators
  - && || !
- Logic equality operators
  - == !=
- Relational operators
  - < <= > >=
- Bit Concatenation operators
  - { }
- Shift operators
  - >> <<
- Conditional operator
  - ?:

Operators	Operator Symbols	Precedence
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

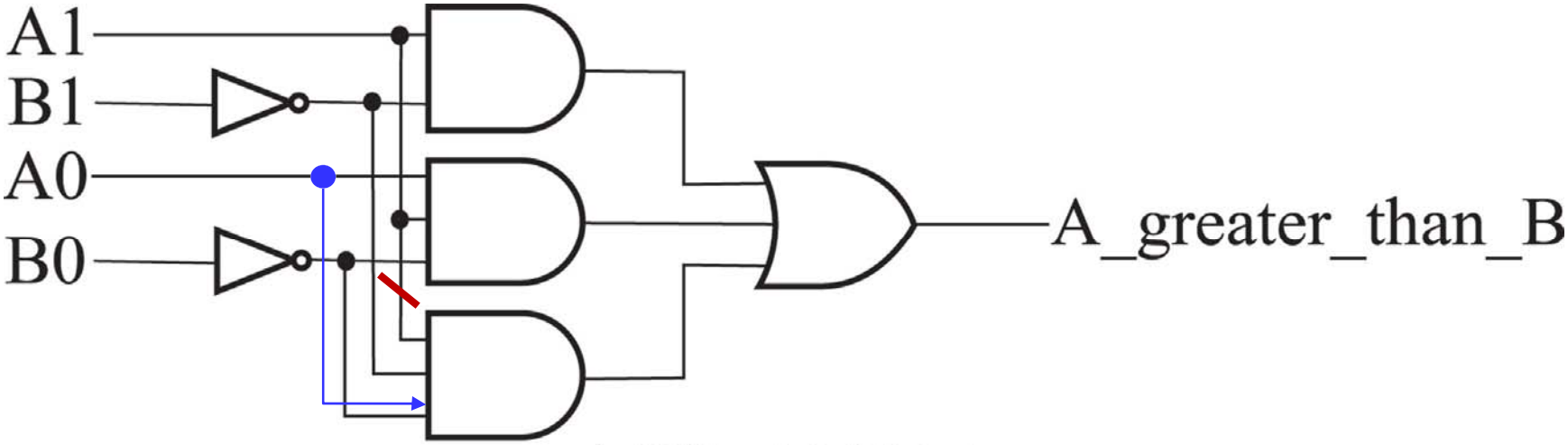
Modeling methods

- Structured modeling
  - Module-level
  - Gate-level
  - Switch-level
- Dataflow modeling
  - Suitable for modeling combinational logic circuits
  - Use continuous assignment statements: assign
- Behavioral modeling
  - Key elements: always

  always @(event signal list) procedure statement

Modeling methods: An Example

- Gate level schematic for a two-bit greater-than comparator circuit



Modeling methods: An Example (cont'd)

- Structural Verilog Description of Two-Bit Greater-Than Circuit

```
// Two-bit greater-than circuit: Verilog structural model                    //  1
// See Figure 2-27 for logic diagram                                         //  2
module comparator_greater_than_structural(A, B, A_greater_than_B);           //  3
 input [1:0] A, B;                                                          //  4
 output A_greater_than_B;                                                   //  5
 wire B0_n, B1_n, and0_out, and1_out, and2_out;                             //  6
  not                                                                       //  7
   inv0(B0_n,  B[0]),  inv1(B1_n,  B[1]);                                   //  8
  and                                                                       //  9
   and0(and0_out,  A[1],  B1_n),                                            // 10
   and1(and1_out,  A[1],  A[0],  B0_n),                                     // 11
   and2(and2_out,  A[0],  B1_n,  B0_n);                                     // 12
  or                                                                        // 13
   or0(A_greater_than_B,  and0_out,  and1_out,  and2_out);                  // 14
endmodule                                                                   // 15
```

Modeling methods: An Example (cont'd)

- Dataflow Verilog Description of Two-Bit Greater-Than Comparator

```
// Two-bit greater-than circuit: Dataflow model                    //  1
// See Figure 2-27 for logic diagram                               //  2
module comparator_greater_than_dataflow(A, B, A_greater_than_B);   //  3
 input [1:0] A, B;                                                 //  4
 output A_greater_than_B;                                          //  5
 wire B1_n, B0_n, and0_out, and1_out, and2_out;                    //  6
 assign B1_n = ~B[1];                                              //  7
 assign B0_n = ~B[0];                                              //  8
 assign and0_out = A[1] & B1_n;                                    //  9
 assign and1_out = A[1] & A[0] & B0_n;                             // 10
 assign and2_out = A[0] & B1_n & B0_n;                             // 11
 assign A_greater_than_B = and0_out | and1_out | and2_out;         // 12
endmodule                                                          // 13
```

Modeling methods: An Example (cont'd)

- Conditional Dataflow Verilog Description of Two-Bit Greater-Than Circuit

```
// Two-bit greater-than circuit: Conditional model                    // 1
// See Figure 2-27 for logic diagram                                  // 2
module comparator_greater_than_conditional2(A, B, A_greater_than_B);  // 3
 input [1:0] A, B;                                                    // 4
 output A_greater_than_B;                                             // 5
 assign A_greater_than_B = (A > B)? 1'b1 :                            // 6
      1'b0;                                                           // 7
endmodule                                                             // 8
```

Modeling methods: An Example (cont'd)

- Conditional Dataflow Verilog Description of Two-Bit Greater-Than Circuit Using Combinations

```
// Two-bit greater-than circuit: Conditional model                    //  1
// See Figure 2-27 for logic diagram                                  //  2
module comparator_greater_than_conditional(A, B, A_greater_than_B);   //  3
 input [1:0] A, B;                                                    //  4
 output A_greater_than_B;                                             //  5
 assign A_greater_than_B = (A == 2'b00)? 1'b0 :                       //  6
              (A == 2'b01)? ~(B[1]|B[0]):                             //  7
              (A == 2'b10)? ~B[1] :                                   //  8
              (A == 2'b11)? ~(B[1]&B[0]):                             //  9
              1'bx;                                                   // 10
 endmodule                                                            // 11
```

Modeling methods: An Example (cont'd)

- Behavioral Verilog Description of Two-Bit Greater-Than Circuit

```
// Two-bit greater-than circuit: Behavioral model                    // 1
// See Figure 2-27 for logic diagram                                 // 2
module comparator_greater_than_behavioral(A, B, A_greater_than_B);   // 3
 input [1:0] A, B;                                                   // 4
 output A_greater_than_B;                                            // 5
 assign A_greater_than_B = A > B;                                    // 6
endmodule                                                            // 7
```

Modeling methods: An Example (cont'd)

- Testbench for the Structural Model of the Two-Bit Greater-Than Comparator

```
// Testbench for Verilog two-bit greater-than comparator                    //  1
module comparator_testbench_verilog();                                      //  2
 reg [1:0] A, B;                                                            //  3
 wire struct_out;                                                           //  4
 comparator_greater_than_structural U1(A, B, struct_out);                   //  5
 initial                                                                    //  6
 begin                                                                      //  7
  A = 2'b10;                                                                //  8
  B = 2'b00;                                                                //  9
  #10;                                                                      // 10
  B = 2'b01;                                                                // 11
  #10;                                                                      // 12
  B = 2'b10;                                                                // 13
  #10;                                                                      // 14
  B = 2'b11;                                                                // 15
 end                                                                        // 16
endmodule                                                                   // 17
```

Overview

- Introduction to Verilog HDL
- About combinational logic circuits
- Some classic/basic designs
- Timing analysis

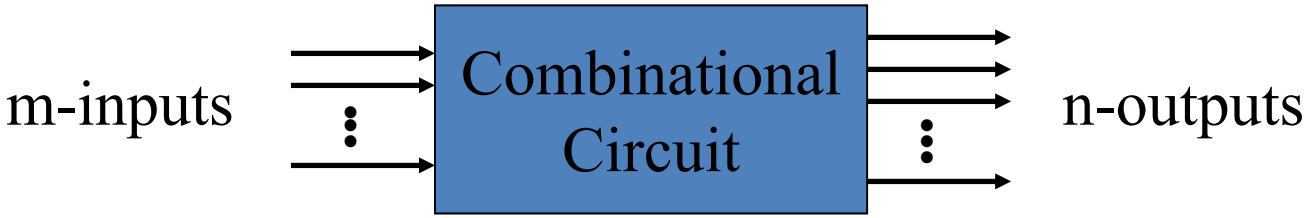
Digital Logic Circuit

- A “blackbox” with input(s) and output(s)
  - Input: 0/1
  - Output: 0/1
- A collection of interconnected digital components
  - Circuit elements
  - Nodes
    - Input, internal and output
- Two types
  - Combinational logic circuit
  - Sequential logic circuit

Combinational Logic Circuit

- A combinational logic circuit has:
  - A set of *m* Boolean inputs,
  - A set of *n* Boolean outputs, and
  - *n* switching functions, each mapping the 2^m input combinations to an output such that the current output depends only on the current input values

- A block diagram:

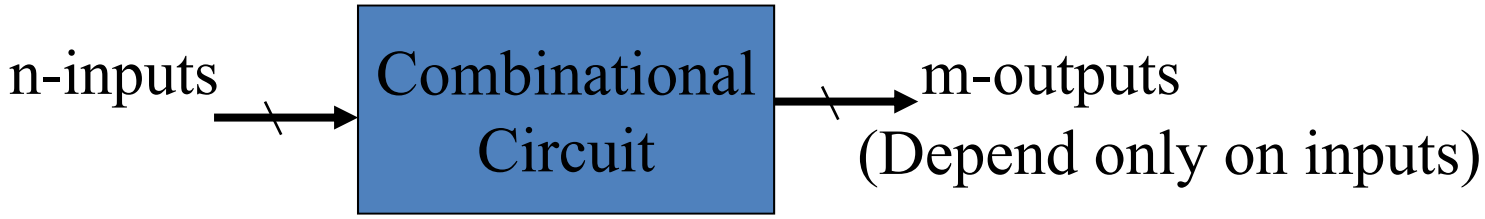


Combinational vs. Sequential Circuits

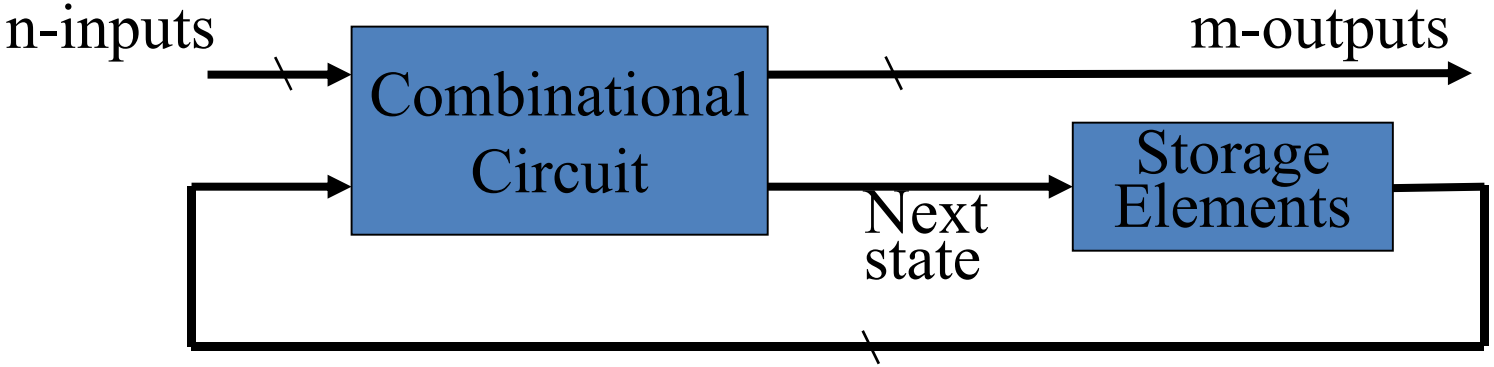
- Combinational circuits are memory-less
  - The output value depends ONLY on the current input values.

- Sequential circuits consist of combinational logic as well as memory elements (used to store certain circuit states)
  - Outputs depend on BOTH current input values and previous input values (kept in the storage elements).

Combinational vs. Sequential Circuits



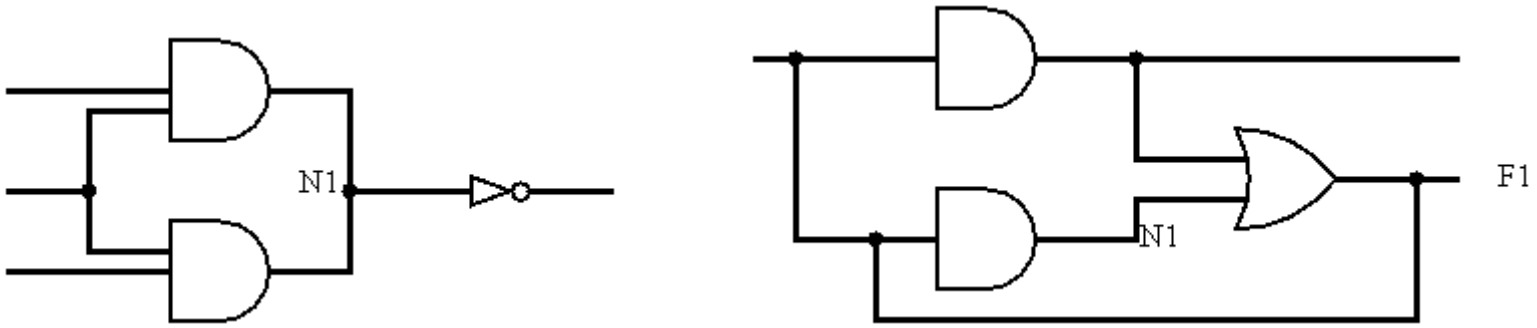
Combinational Circuit



Sequential Circuit

Characteristics of Combinational Logic Circuits

- Each element is a combinational logic circuit.
- A node can only be the output of one element.
- Loop is not allowed.



Design Choice Of Combinational Circuits

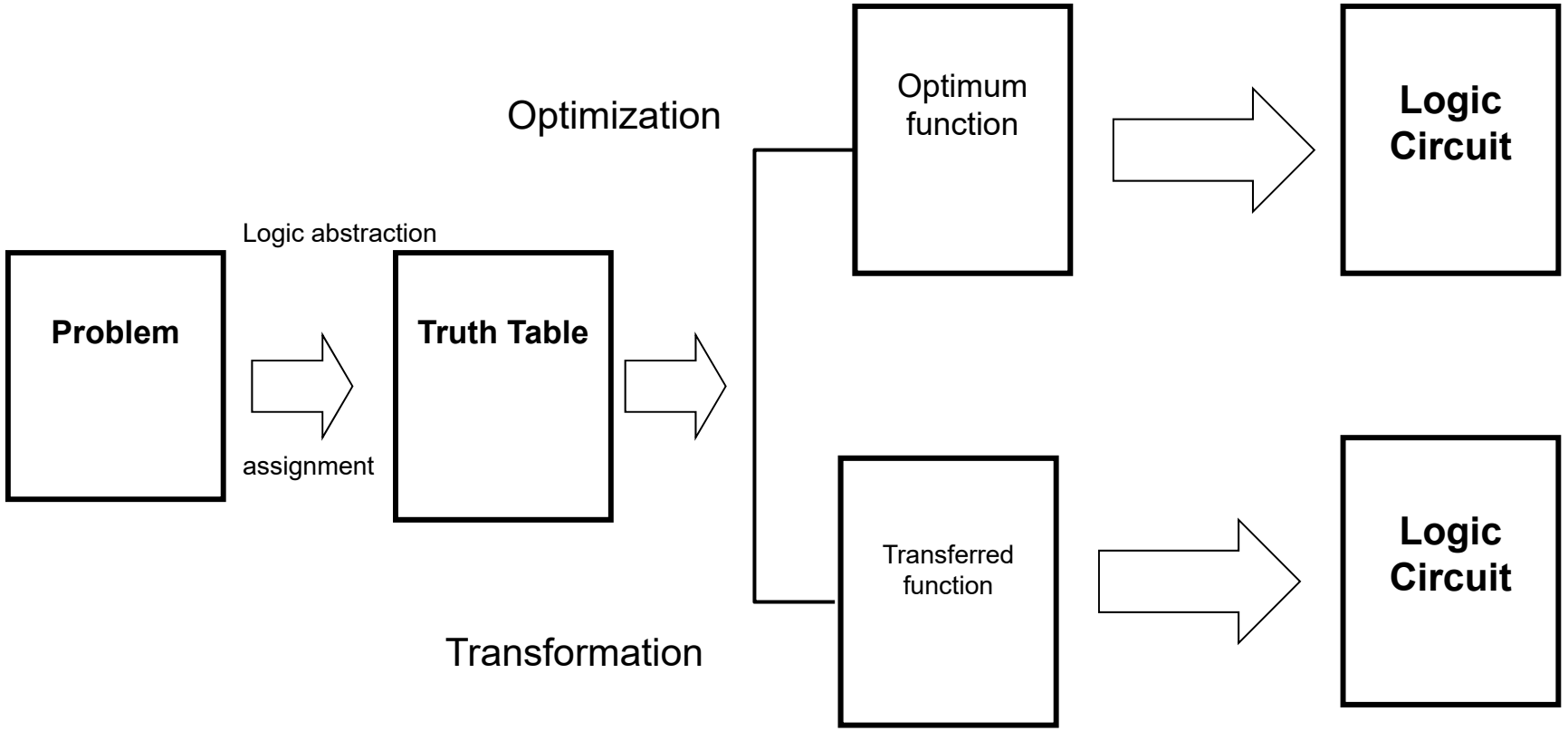
- 2-level vs. multi-level
  - Gate delay
  - Fan-in/fan-out
  - Trade-off between cost and speed
- Functional blocks

Design Procedure

1. Specification
   - Write a specification for the circuit if one is not already available (text, HDL)
2. Formulation
   - Derive a truth table or initial Boolean equations that define the required relationships between the inputs and outputs, if not in the specification
3. Optimization
   - Apply 2-level and multiple-level optimization
   - Draw a logic diagram or provide a netlist for the resulting circuit using ANDs, ORs, and inverters

Design Procedure

4. Technology Mapping
   - Map the logic diagram or netlist to the implementation technology selected
5. Verification
   - Verify the correctness of the final design


Optimization
Optimum function
Logic Circuit
Logic abstraction
Problem
Truth Table
assignment
Transferred function
Logic Circuit
Transformation

Design Example 1

Example: The only light in the room is controlled by three switches, and each switch can control the light separately. Please design the logic circuit for the light and switches with least gates.

Solutions: 1. Specification

Analysis result:

input signal: switches S_1,S_2,S_3
output signal: light F
"1" for switch closed and "0" for opened
"1" for light on and "0" for light off

Truth table

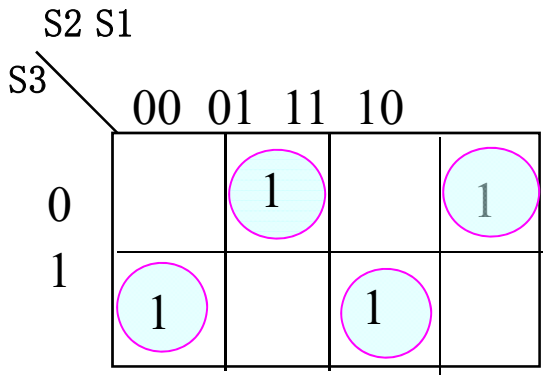
S_3	S_2	S_1	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

2. Formulation

$$F = \overline{S}_3\overline{S}_2S_1 + \overline{S}_3S_2\overline{S}_1 + S_3\overline{S}_2\overline{S}_1 + S_3S_2S_1$$

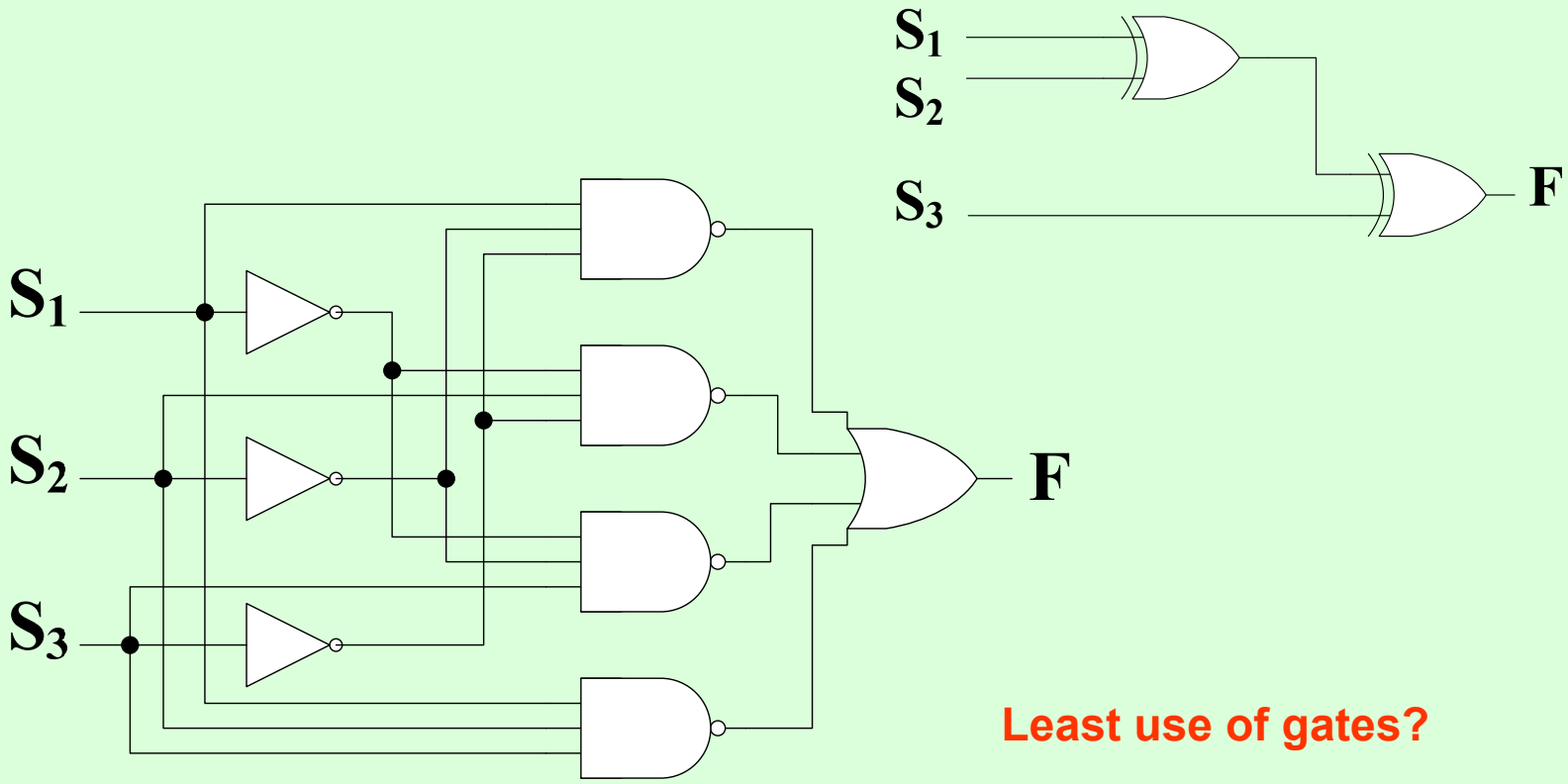
S_3	S_2	S_1	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

3. Optimization



Already optimized

4. Technology Mapping



Least use of gates?

$$F = \overline{S}_3\overline{S}_2S_1 + \overline{S}_3S_2\overline{S}_1 + S_3\overline{S}_2\overline{S}_1 + S_3S_2S_1$$

Verilog Programming

```
module lamp_control(s1, s2, s3, F );
   input  s1,s2,s3;
   output F;
   wire   s1,s2,s3,f;

   assign F= (~s3&~s2&s1) | (~s3&s2&~s1) | (s3&~s2&~s1) | (s3&s2&s1);

endmodule
```

Design Example 2

1. Specification
   - BCD to Excess-3 code converter
   - Transforms BCD code for the decimal digits to Excess-3 code for the decimal digits
   - BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively
   - Excess-3 code words for digits 0 through 9: 4-bit patterns consisting of 3 (binary 0011) added to each BCD code word
   - Implementation:
     - multiple-level circuit
     - NAND gates (including inverters)

Design Example 2 (continued)

2. Formulation

- Conversion of 4-bit codes can be most easily formulated by a truth table
- Variables
  - BCD: A,B,C,D
- Variables
  - Excess-3 W,X,Y,Z
- Don't Cares
  - BCD 1010 to 1111

Input BCD A B C D	Output Excess-3 WXYZ
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

Design Example 2 (continued)

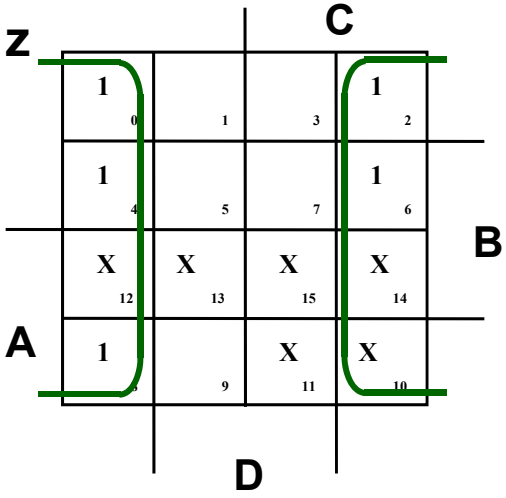
3. Optimization
   a. 2-level using K-maps

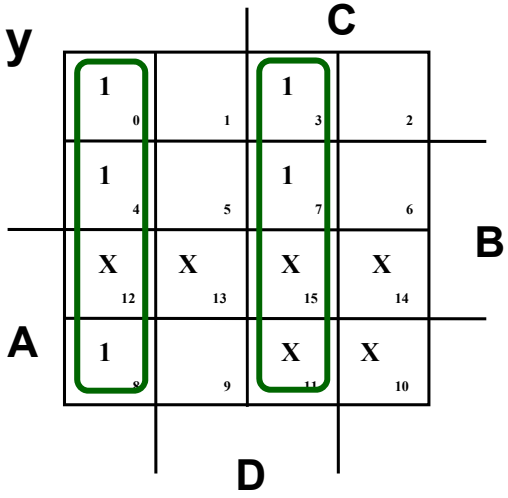
$$W = A + BC + BD$$

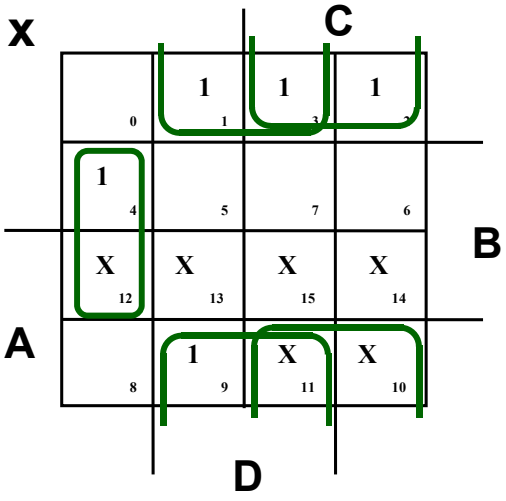
$$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D}$$

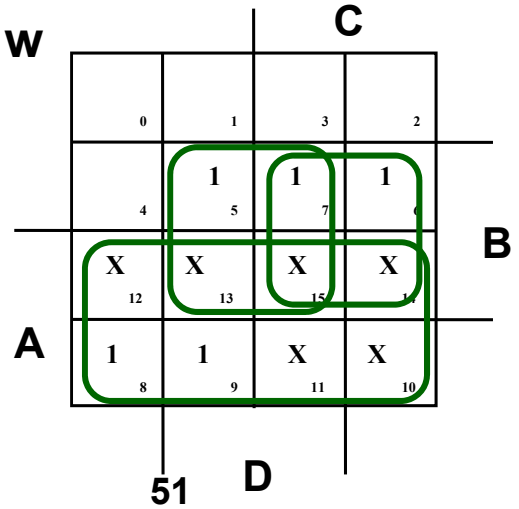
$$Y = CD + \overline{C}\,\overline{D}$$

$$Z = \overline{D}$$









Design Example 2 (continued)

3. Optimization (continued)

   b. Multiple-level using transformations

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

G = 7 + 10+ 6 + 0 = 23

- Perform extraction, finding factor: $T_1 = C + D$

$$T_1 = C + D$$
$$W = A + BT_1$$
$$X = \overline{B}T_1 + B\overline{C}\,\overline{D}$$
$$Y = CD + \overline{C}\,\overline{D}$$
$$Z = \overline{D}$$

G = 2+4+7+6+0=19

Design Example 2 (continued)

b. Multiple-level using transformations

$$T_1 = C + D$$
$$W = A + BT_1$$
$$X = \overline{B}T_1 + B\overline{C}\,\overline{D}$$
$$Y = CD + \overline{C}\,\overline{D}$$
$$Z = \overline{D}$$

G = 2+4+7+6+0=19

- An additional extraction not shown in the text since it uses a Boolean transformation:

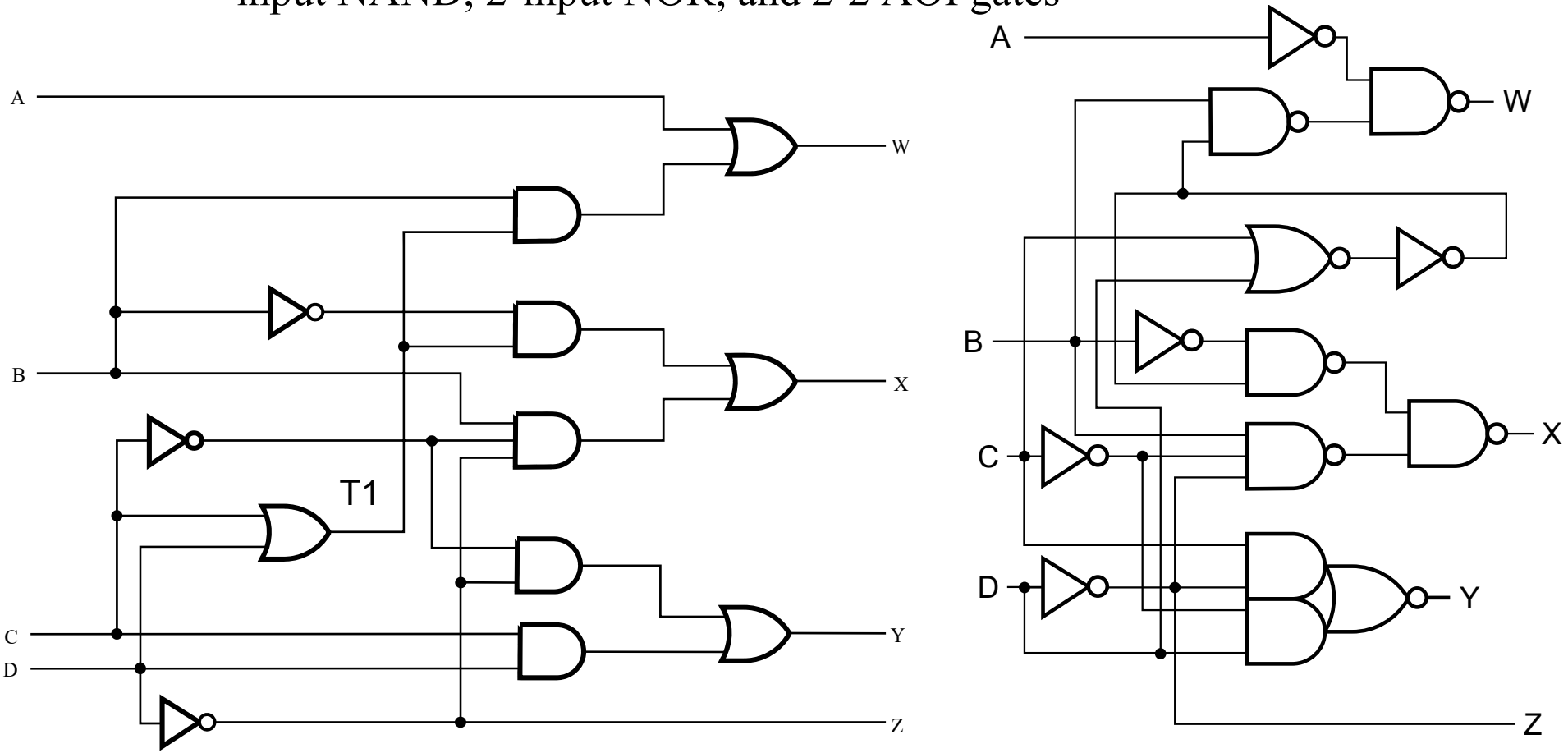
$$\overline{T_1} = \overline{C + D} = \overline{C}\,\overline{D}$$
$$W = A + BT_1$$
$$X = \overline{B}T_1 + B\overline{T_1}$$
$$Y = CD + \overline{T_1}$$
$$Z = \overline{D}$$

G = 2 +1 + 4 + 6 + 4 + 0 = 17!

Design Example 2 (continued)

4. Technology Mapping

- Mapping with a library containing inverters and 2-input NAND, 3-input NAND, 2-input NOR, and 2-2 AOI gates



5. Verilog Programming

```
module BCD_ Excess_3( A,B,C,D,W,X,Y,Z );
    input   A,B,C,D;
    output  W,X,Y,Z;
    wire A,B,C,D , W,X,Y,Z,T1;

    assign  T1 = C|D;
    assign  W = A|B&C|B&D ;
    assign  X = ~B&T1|B&~T1;
    assign  Y = C&D|~T1;
    assign  Z= ~D ;

endmodule
```

Overview

- Introduction to Verilog HDL
- About combinational logic circuits
- Some classic/basic designs
- Timing analysis

Some Classic/Basic Designs

- Functions and functional blocks
- Rudimentary logic functions
- Encoder and decoder
- Multiplexer and demultiplexer
- Half adder and full adder

Functions and Functional Blocks

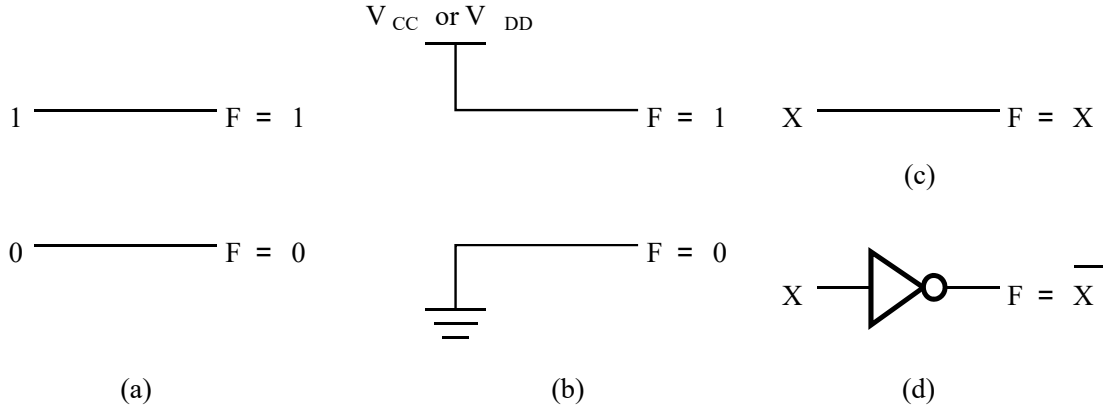
- The functions considered are those found to be very useful in design
- Corresponding to each function is a combinational circuit implementation called a *functional block*.
- In the past, many functional blocks were implemented as SSI, MSI, and LSI circuits.
- Today, they are often used as simply parts within a VLSI circuit.

Rudimentary Logic Functions

- Four elementary combinational logic functions
  - Value-Fixing: $F = 0$ or $F = 1$, *no Boolean operator*
  - Transferring: $F = X$, *no Boolean operator*
  - Inverting: $F = \overline{X}$, *involves one logic gate*
  - Enabling: $F = X \cdot EN$ *or* $F = X + \overline{EN}$, *involves one or two logic gates*
  - The first three are functions of a single variable X

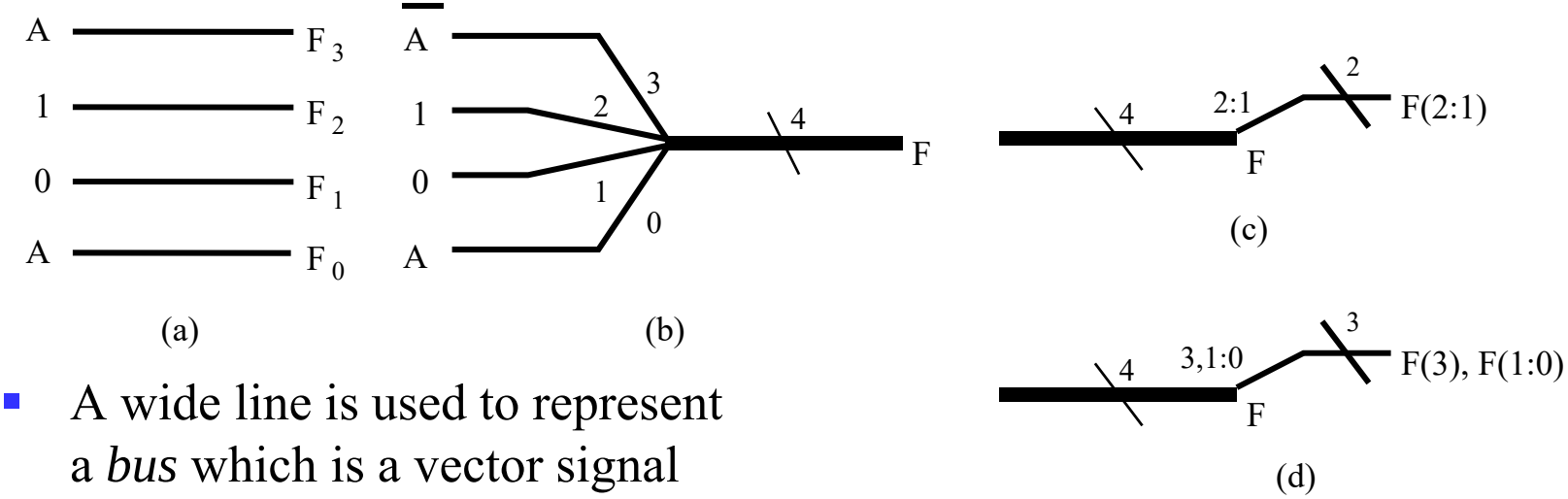
Table 4-1
Functions of one variable

X	F =0	F =X	F = $\overline{X}$	F=1
0	0	0	1	1
1	0	1	0	1



Multiple-bit Rudimentary Functions

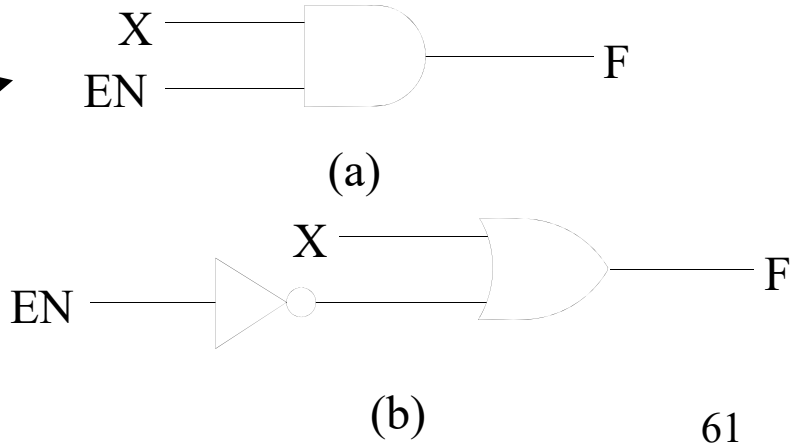
- Multi-bit Examples:



- A wide line is used to represent a *bus* which is a vector signal
- In (b) of the example, F = (F_3, F_2, F_1, F_0) is a bus.
- The bus can be split into individual bits as shown in (b)
- Sets of bits can be split from the bus as shown in (c) for bits 2 and 1 of F.
- The sets of bits need not be continuous as shown in (d) for bits 3, 1, and 0 of F.

Enabling Function

- *Enabling* permits an input signal to pass through to an output
- *Disabling* blocks an input signal from passing through to an output, replacing it with a fixed value
- The value on the output when it is disable can be Hi-Z (as for three-state buffers and transmission gates), 0 , or 1
- When disabled, 0 output
- When disabled, 1 output
- See Enabling App in text



Decoding

- Decoding - the conversion of an *n*-bit input code to an *m*-bit output code with $n \leq m \leq 2^n$ such that each valid code word produces a unique output code
- Circuits that perform decoding are called *decoders*
- Here, functional blocks for decoding are
  - called *n*-to-*m* line decoders, where $m \leq 2^n$, and
  - generate 2^n (or fewer) minterms for the *n* input variables

Classic Designs: Decoder

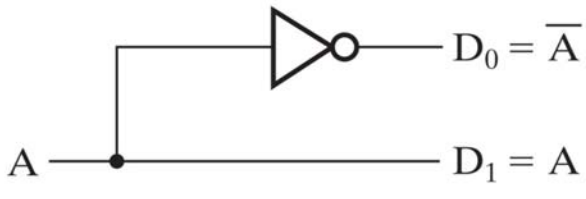
- Multiple inputs: address, binary code, ...
- Multiple outputs: one-hot, ...
- Example:
  - 3-bit binary input, 8-bit output.
  - When the input 3 bits are 000, then the first bit in output is 1, and others are 0.
  - When the input 3 bits are 001, then only the second bit in output is 1.
  - ...

Decoder Examples

- 1-to-2-Line Decoder

A	D_0	D_1
0	1	0
1	0	1

(a)

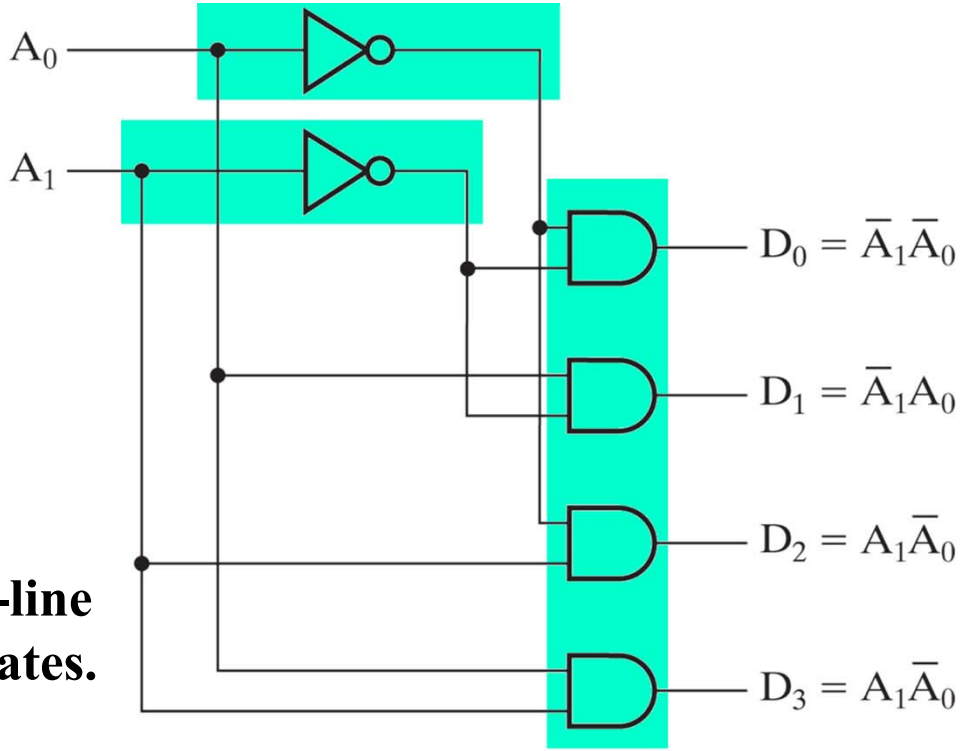


(b)

- 2-to-4-Line Decoder

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)

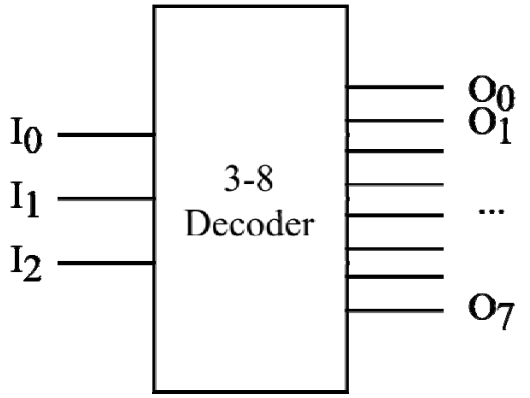


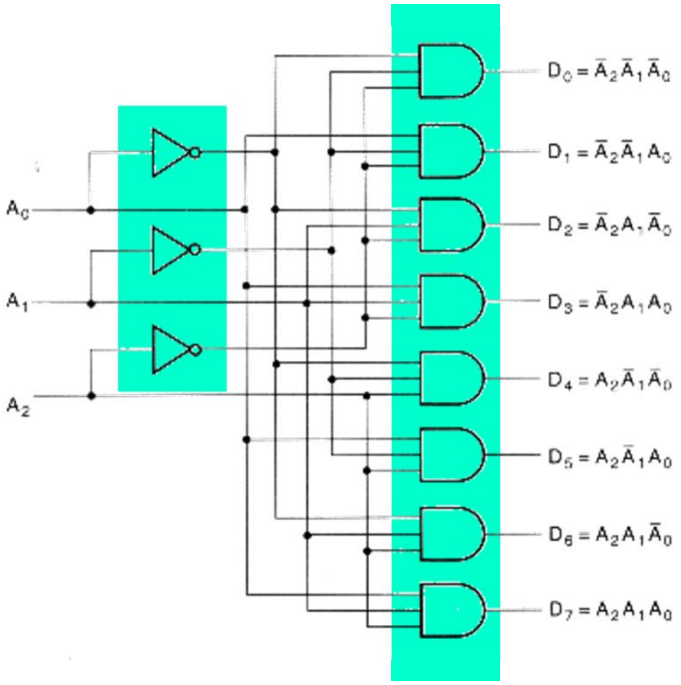
(b)

The 2-4-line made up of 2 1-to-2-line decoders and 4 (2-inputs) AND gates.

Decoder Examples (cont'd)

Input			Output							
A_2	A_1	A_0	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





The 3-8-line is made up of 3 1-to-2-line decoders and 8 (3-inputs) AND gates.

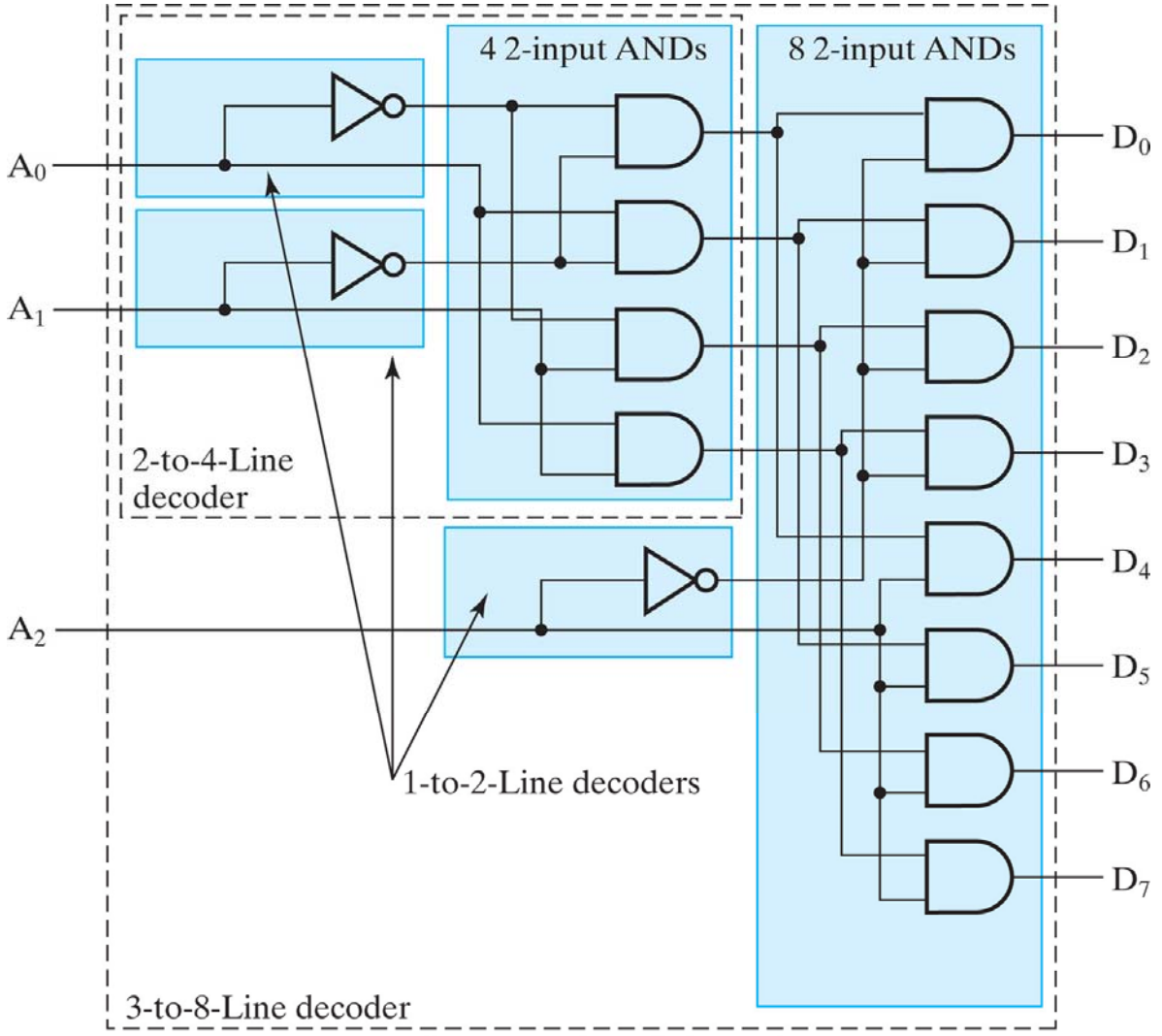
Decoder Expansion

- General procedure for any decoder with n inputs and 2^n outputs
  - Step 1: let k = n
  - Step 2: there are 2^k AND gates which will be driven by 2 decoders
    - If k is even, both decoders have $2^{k/2}$ outputs.
    - If k is odd, one decoder has $2^{(k+1)/2}$ outputs, while another one has $2^{(k-1)/2}$ outputs.
  - Step 3: for each decoder obtained from step 2, let the number of outputs becomes k, and repeat step 2 until k=1 (i.e., 1-to-2-line decoder).
- The output AND gates are driven by two decoders with their numbers of inputs either equal or differing by 1.
- These decoders are then designed using the same procedure until 1-to-2-line decoders are reached.
- The procedure can be modified to apply to decoders with the number of outputs $\neq 2^n$.

Decoder Expansion: 3-8 Decoder

- Number of output ANDs = 8
- Number of inputs to decoders driving output ANDs = 3
- Closest possible split to equal
  - 2-to-4-line decoder
  - 1-to-2-line decoder
- 2-to-4-line decoder
  - Number of output ANDs = 4
  - Number of inputs to decoders driving output ANDs = 2
  - Closest possible split to equal
    - Two 1-to-2-line decoders

Decoder Expansion: 3-8 Decoder (cont'd)

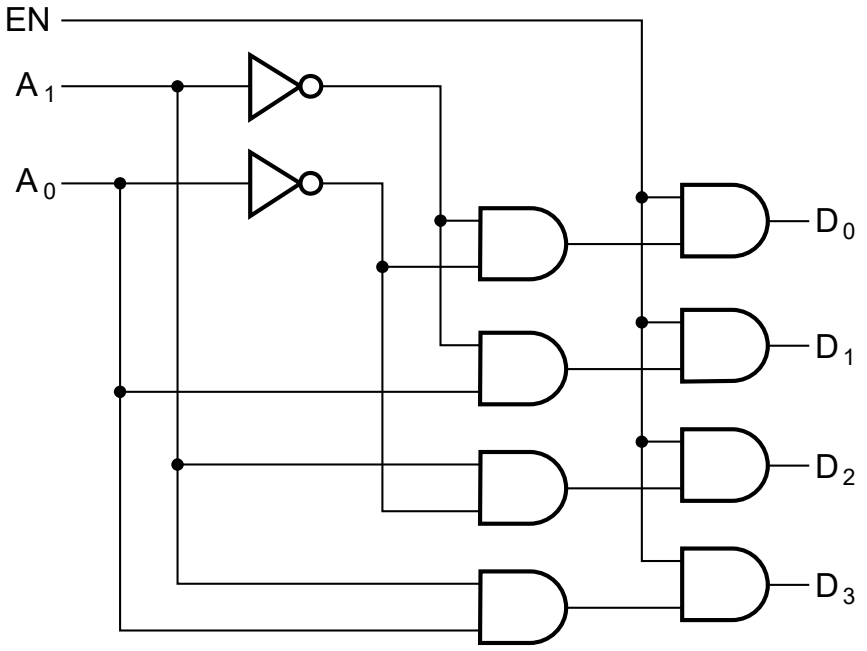


Decoder with Enable

- In general, attach *m*-enabling circuits to the outputs
- See truth table below for function
  - Note use of X's to denote both 0 and 1
  - Combination containing two X's represent four binary combinations
- Alternatively, can be viewed as distributing value of signal EN to 1 of 4 outputs
- In this case, called a *demultiplexer*

EN	A_1	A_0	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)

Combinational Logic Implementation - Decoder and OR Gates

- Implement m functions of n variables with:
  - Sum-of-minterms expressions
  - One n-to-2^n-line decoder
  - m OR gates, one for each output
- Approach 1:
  - Find the truth table for the functions
  - Make a connection to the corresponding OR from the corresponding decoder output wherever a 1 appears in the truth table
- Approach 2
  - Find the minterms for each output function
  - OR the minterms together

Decoder and OR Gates Example

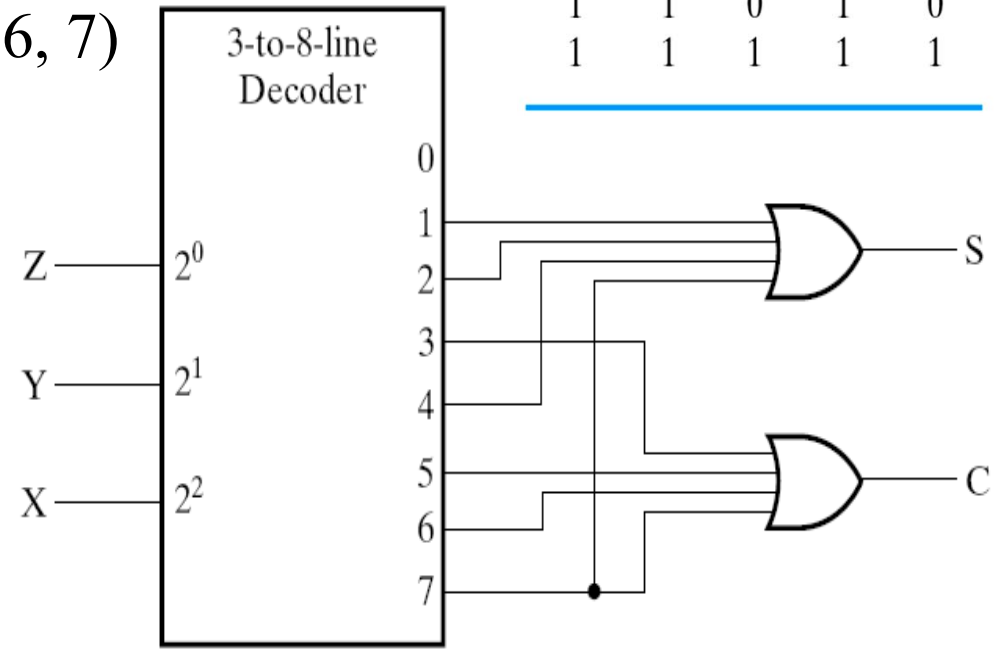
- Implement a binary Adder
- Finding sum of minterms expressions

$S(X, Y, Z) = \Sigma_m(1, 2, 4, 7)$
$C(X, Y, Z) = \Sigma_m(3, 5, 6, 7)$

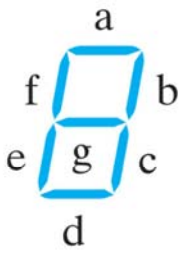
Find circuit

Truth Table

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

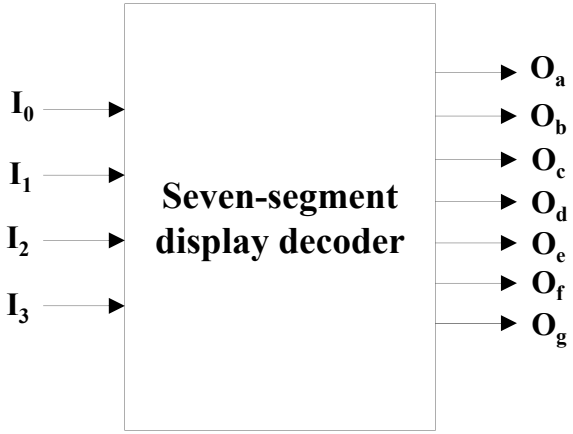


Seven-Segment Display Decoder



(a) Segment designation

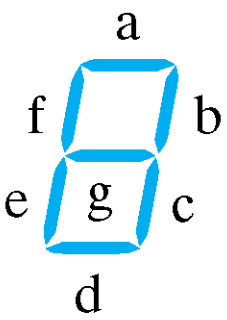
(b) Numeric designation for display



Input				Output						
I_0	I_1	I_2	I_3	O_a	O_b	O_c	O_d	O_e	O_f	O_g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

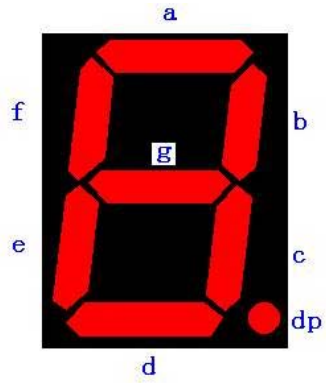
BCD-to-Segment Decoder

- Seven-Segment Displayer

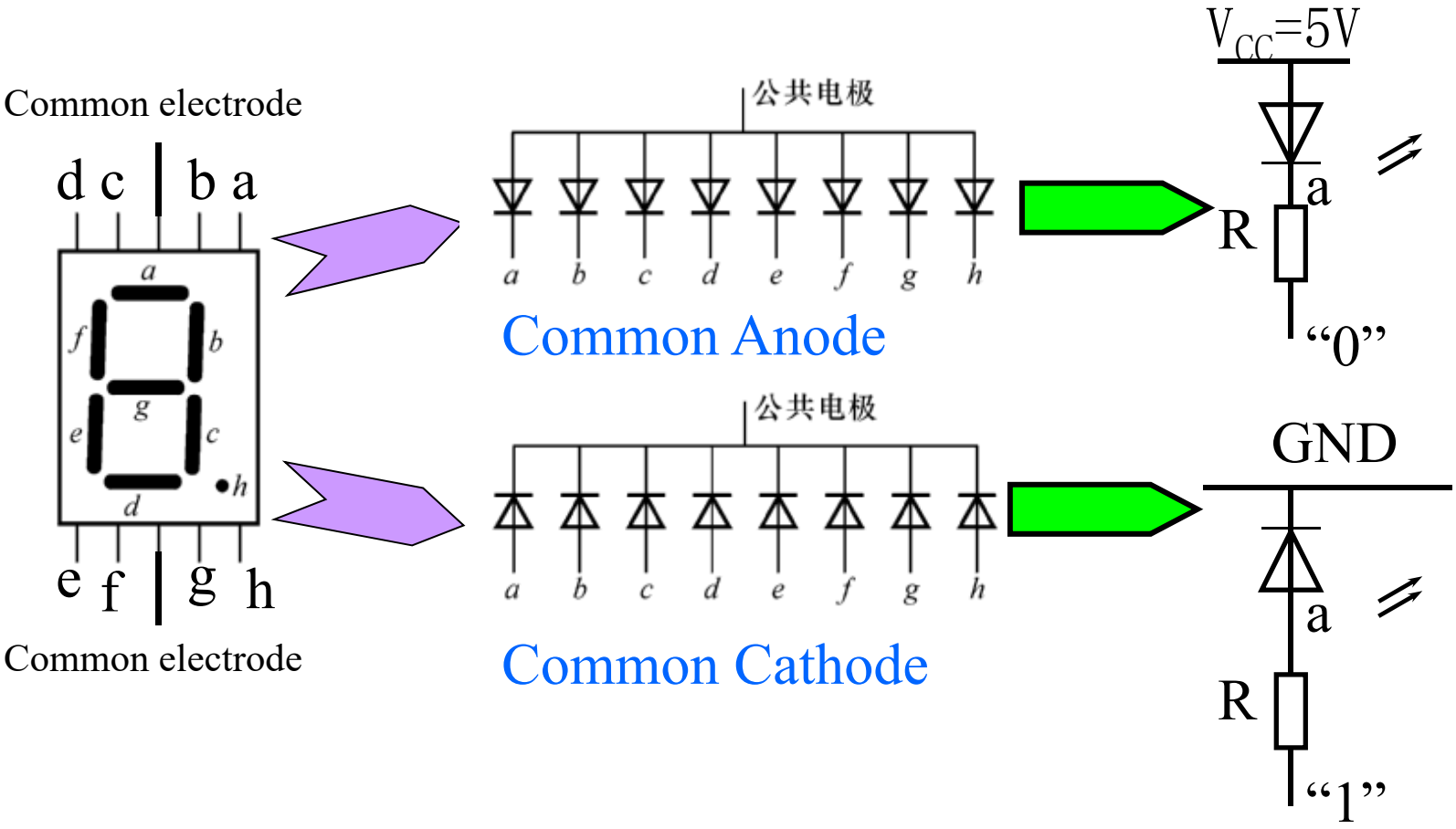


(a) Segment designation

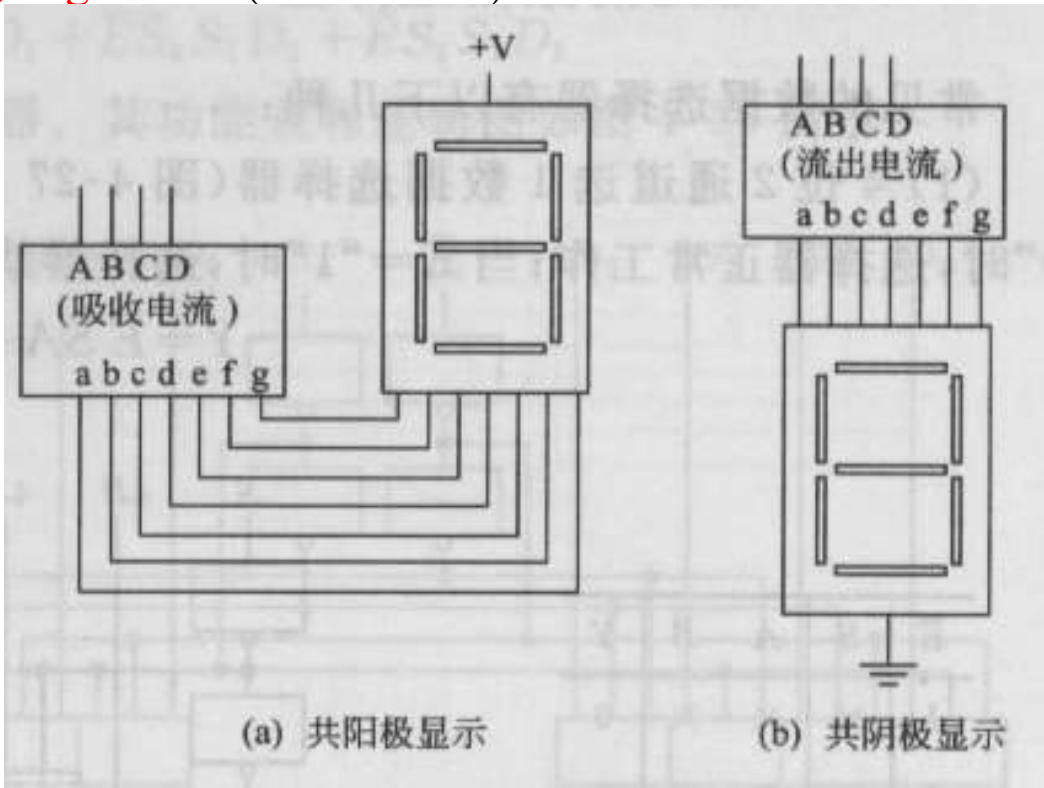
(b) Numeric designation for display



Seven Segment Displayer



- The Common Anode Display (CAD)
  - All the anode connections of the LED's are joined together to logic "1" and the individual segments are illuminated by connecting the individual Cathode terminals to a "LOW", logic "0" signal.
  - a~g = 0 on, a~g = 1 off (Active Low)



- The Common Cathode Display (CCD)
  - All the cathode connections of the LED's are joined together to logic "0" or ground.
  - The individual segments are illuminated by application of a "HIGH", logic "1" signal to the individual Anode terminals.
  - a~g = 1 on, a~g = 0 off (Active High)

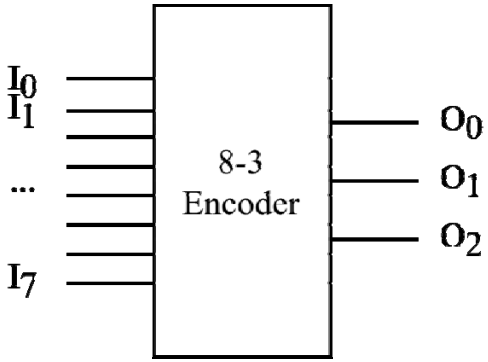
BCD-to-Seven Segment Decoder

- Truth Table for BCD-to-Seven-Segment Decoder

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

Classic Designs: Encoder

- 8-to-3-line encoder



$$O_0 = I_4 + I_5 + I_6 + I_7$$
$$O_1 = I_2 + I_3 + I_6 + I_7$$
$$O_2 = I_1 + I_3 + I_5 + I_7$$

OR gates!

Input								Output		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	O_0	O_1	O_2
1	0	0	0	0	0	0	0	0	0	0
0	**1**	0	0	0	0	0	0	0	0	1
0	0	**1**	0	0	0	0	0	0	1	0
0	0	0	**1**	0	0	0	0	0	1	1
0	0	0	0	**1**	0	0	0	1	0	0
0	0	0	0	0	**1**	0	0	1	0	1
0	0	0	0	0	0	**1**	0	1	1	0
0	0	0	0	0	0	0	**1**	1	1	1

However, there exists two issues that may make the encoder just designed does not work:

1. If more than one input value is 1 (e.g., I_3 & I_6 are both 1)
2. All inputs are 0 vs. $I_0 = 1$

Priority Encoder Example

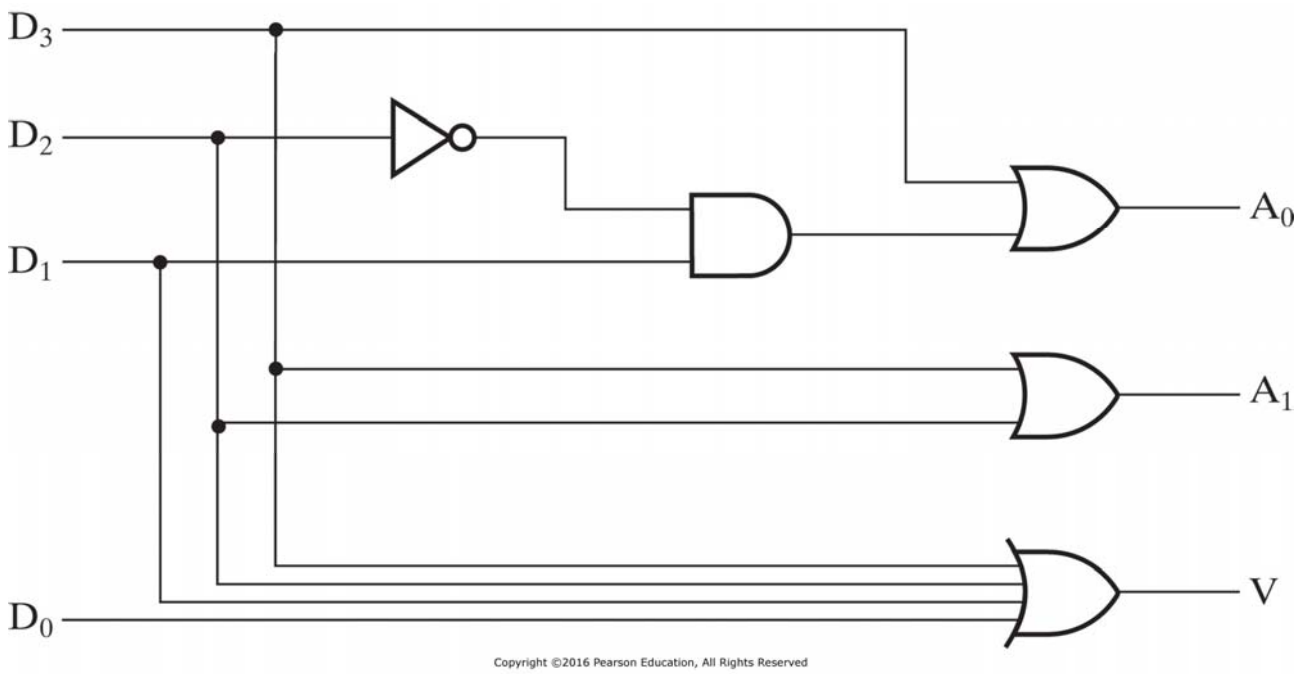
- Priority encoder with 4 inputs (D_3, D_2, D_1, D_0) - highest priority to most significant 1 present - Code outputs A1, A0 and V where V indicates at least one 1 present.

No. of Min-terms/Row	Inputs				Outputs		
	D3	D2	D1	D0	A1	A0	V
1	0	0	0	0	X	X	0
1	0	0	0	1	0	0	1
2	0	0	1	X	0	1	1
4	0	1	X	X	1	0	1
8	1	X	X	X	1	1	1

- Xs in input part of table represent 0 or 1; thus table entries correspond to product terms instead of minterms. The column on the left shows that all 16 minterms are present in the product terms in the table

Priority Encoder Example (cont'd)

- Could use a K-map to get equations, but can be read directly from table and manually optimized if careful:
  - $A_1 = ?$
  - $A_0 = ?$
  - $V = ?$

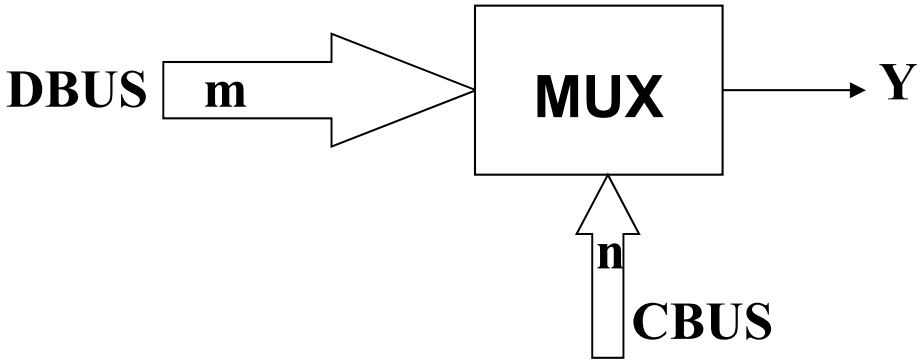


Classic Designs: Multiplexer

- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform selecting have:
  - A set of information inputs from which the selection is made
  - A single output
  - A set of control lines for making the selection
- Logic circuits that perform selecting are called *multiplexers*
- Selecting can also be done by three-state logic or transmission gates

Multiplexers

- A multiplexer selects information from an input line and directs the information to an output line
- A typical multiplexer has *n* control inputs (S_{n-1}, … S_0) called *selection inputs*, 2^n information inputs (I_{2^n-1}, … I_0), and one output Y
- A multiplexer can be designed to have *m* information inputs with $m < 2^n$ as well as *n* selection inputs

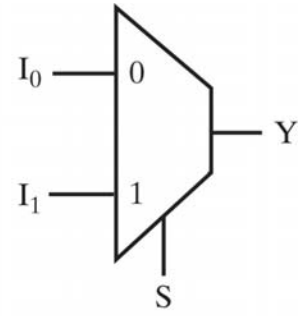


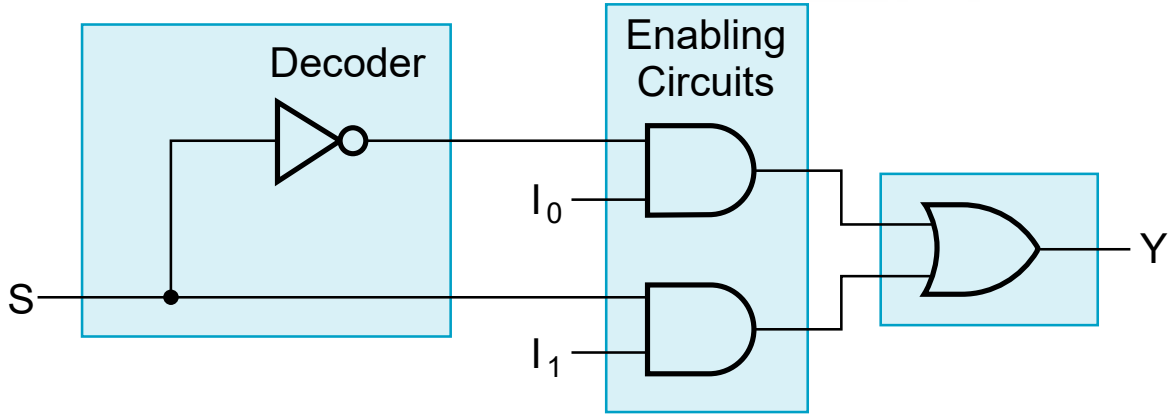
2-to-1-Line Multiplexer

- Since $2 = 2^1$, n = 1
- The single selection variable S has two values:
  - S = 0 selects input I_0
  - S = 1 selects input I_1
- The equation:

$$Y = \overline{S}I_0 + SI_1$$

- The circuit:



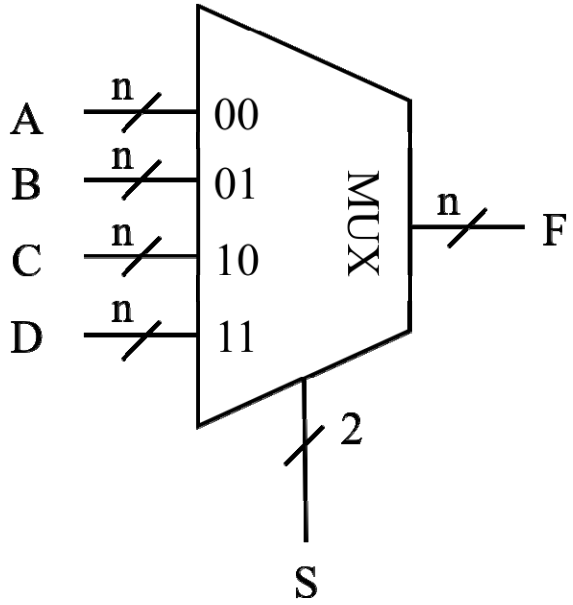


2-to-1-Line Multiplexer (cont'd)

- Note the regions of the multiplexer circuit shown:
  - 1-to-2-line Decoder
  - 2 Enabling circuits
  - 2-input OR gate
- To obtain a basis for multiplexer expansion, we combine the Enabling circuits and OR gate into a 2×2 AND-OR circuit:
  - 1-to-2-line Decoder
  - 2×2 AND-OR
- In general, for an 2^n-to-1-line multiplexer:
  - n-to-2^n-line Decoder
  - $2^n \times 2$ AND-OR

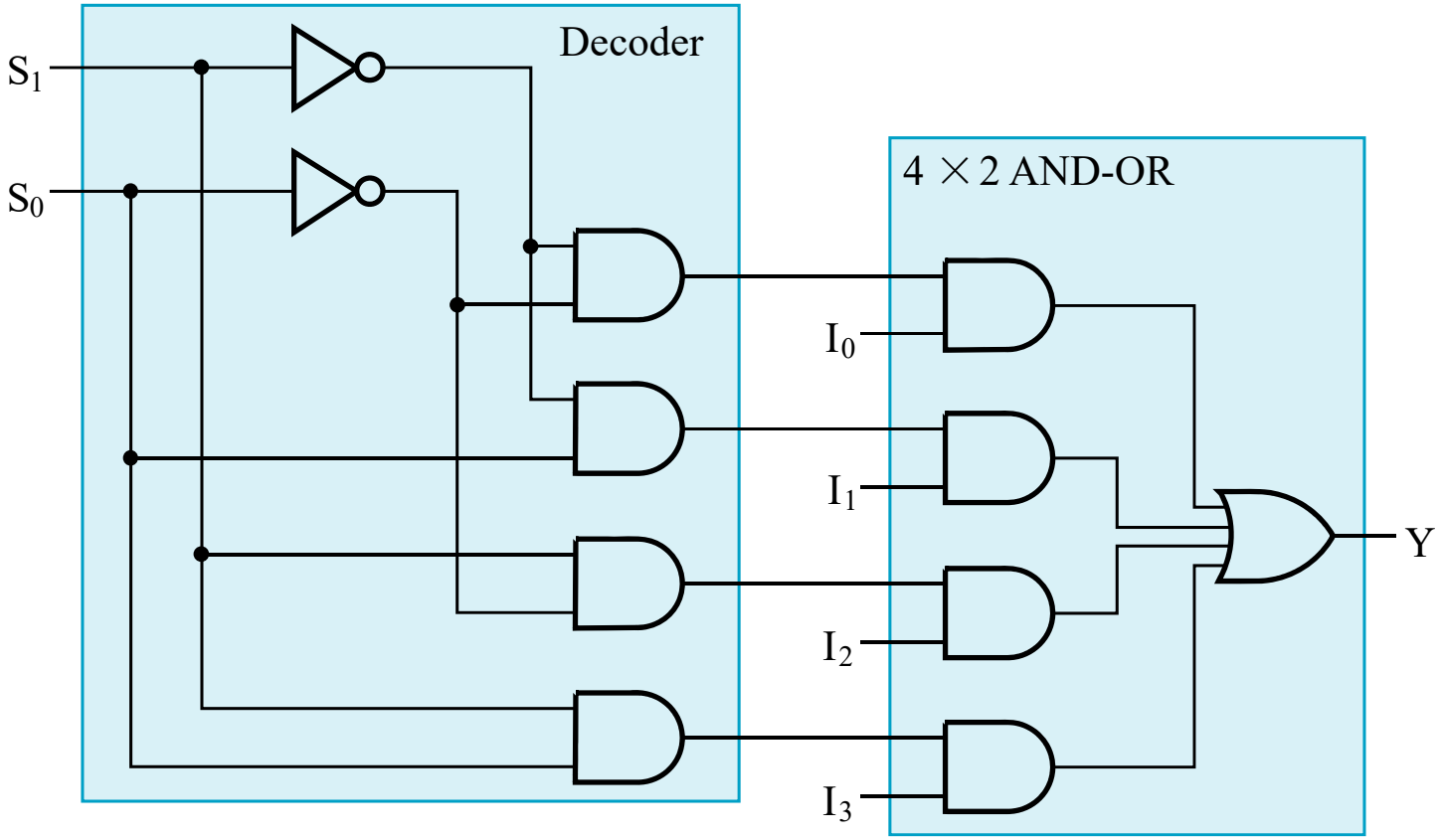
4-to-1-line Multiplexer

Input		Output
S_0	S_1	F
0	0	A
0	1	B
1	0	C
1	1	D



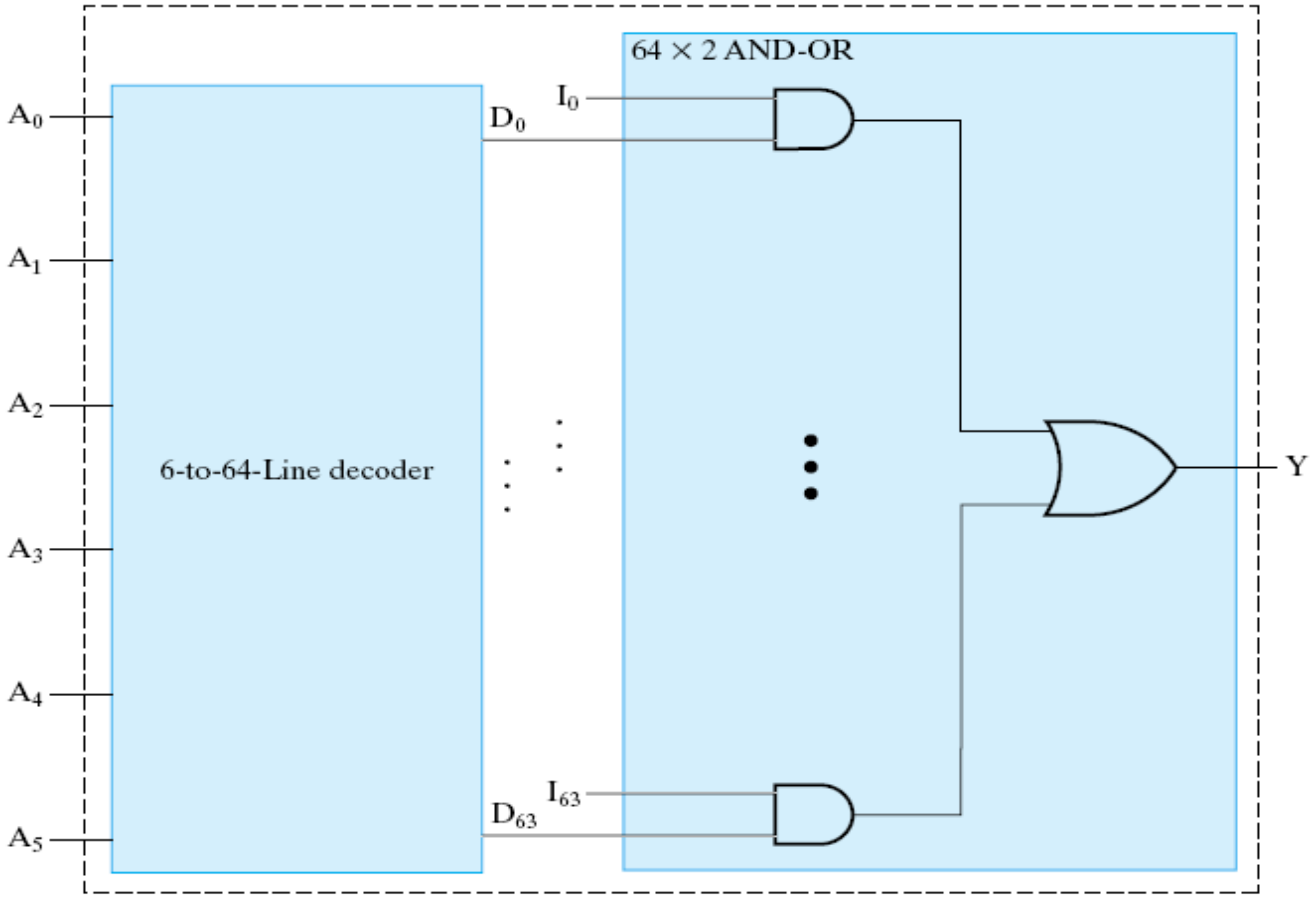
4-to-1-line Multiplexer (cont'd)

- 2-to-2^2-line decoder
- $2^2 \times$ 2 AND-OR



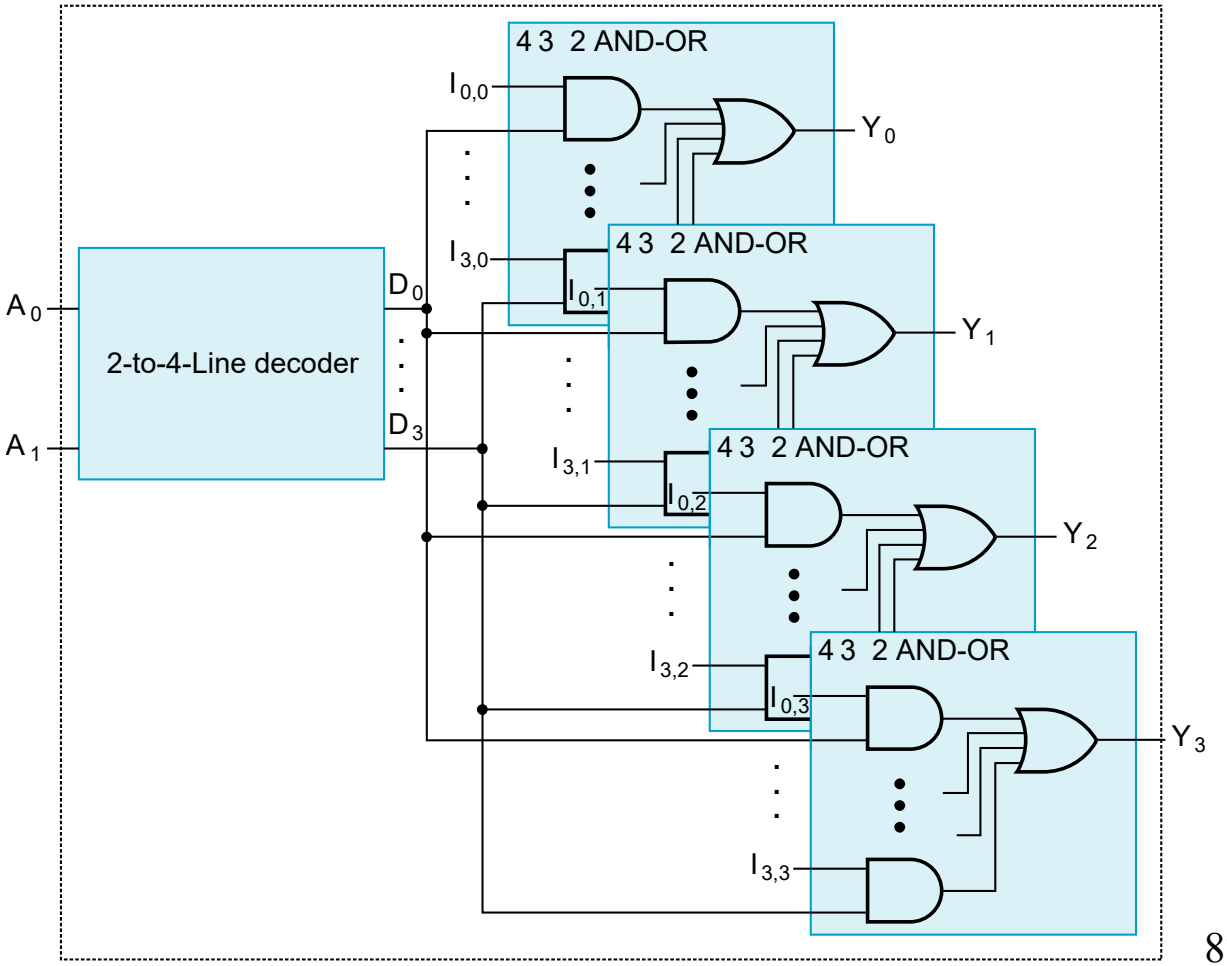
Example: 64-to-1-line Multiplexer

- 6-to-2^6-line decoder
- $2^6 \times 2$ AND-OR



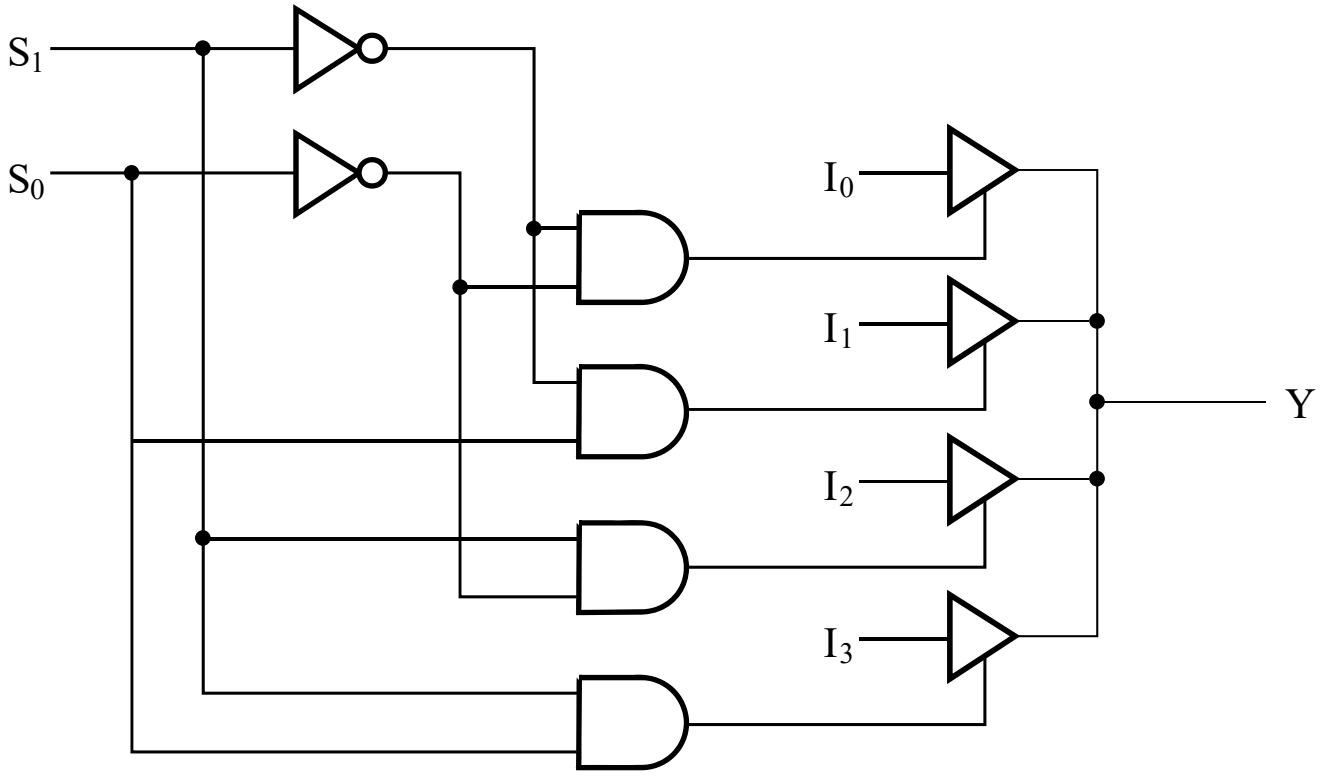
Multiplexer Width Expansion

- Select “vectors of bits” instead of “bit”
- Use multiple copies of $2^n \times 2$ AND-OR in parallel
- Example: 4-to-1-line quad multi-plexer



Other Selection Implementations

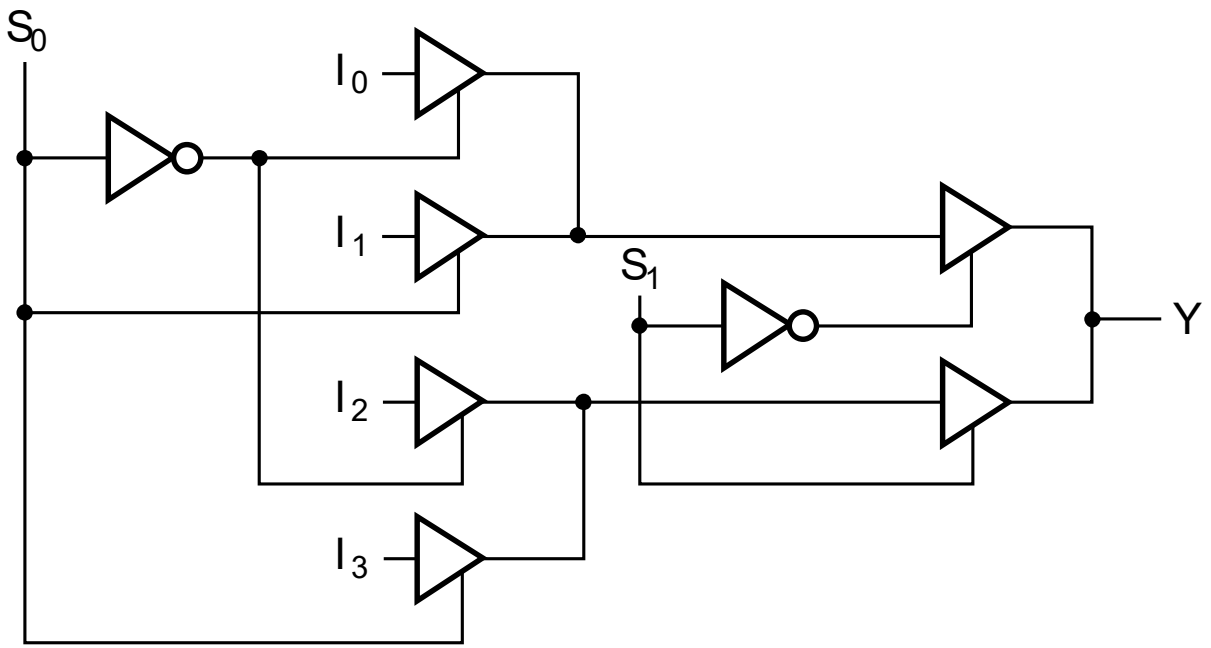
- Three-state logic in place of AND-OR



- Gate input cost = 18

Other Selection Implementations

- Distributing the decoding across the three-state drivers

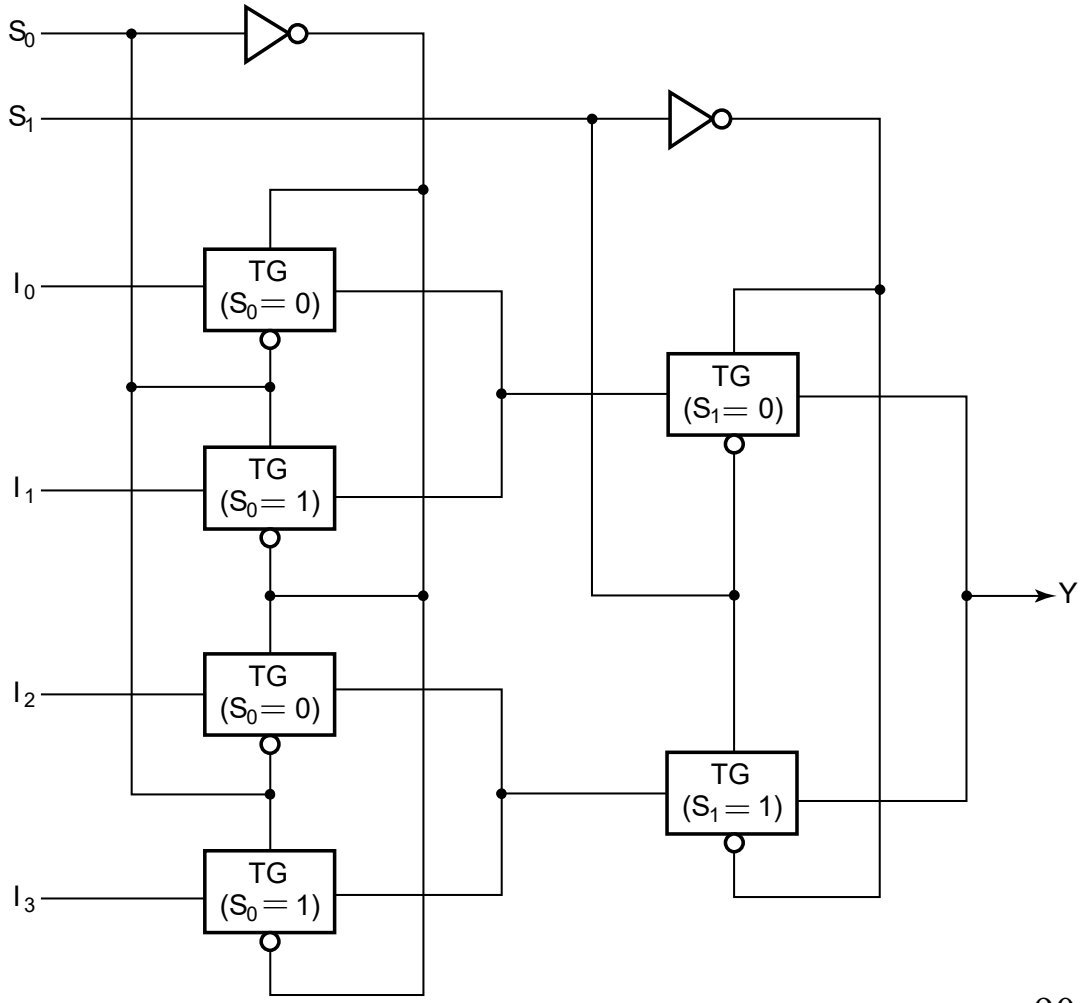


(b)

- Gate input cost = 14
- GN of AND-OR gate implementation:22
- GN of implementation with AND-OR gate replaced by three-state drivers: 18

Other Selection Implementations

- Transmission Gate Multiplexer
- Gate input cost = 8 compared to 14 for 3-state logic and 18 or 22 for gate logic



Combinational Logic Implementation - Multiplexer Approach 1

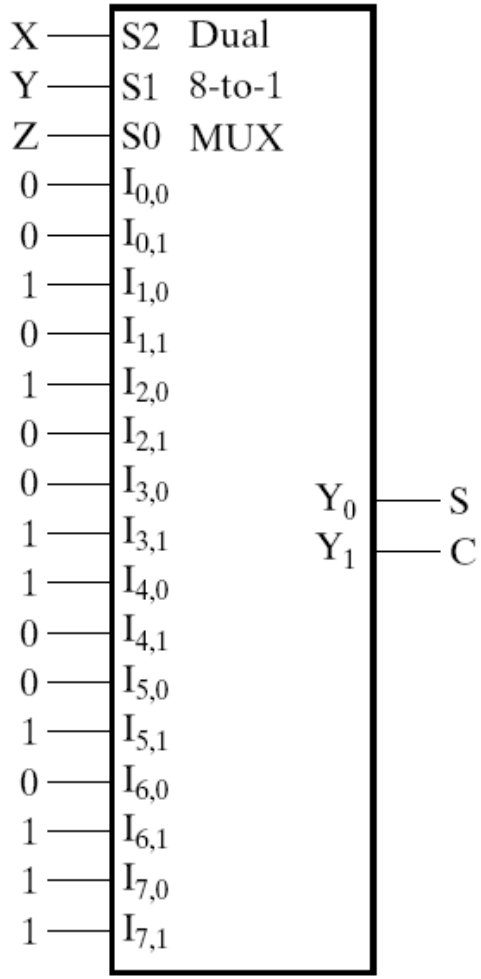
- Implement *m* functions of *n* variables with:
  - Sum-of-minterms expressions
  - An *m*-wide 2^n-to-1-line multiplexer
- Design:
  - Find the truth table for the functions.
  - In the order they appear in the truth table:
    - Apply the function input variables to the multiplexer inputs $S_{n-1}, \cdots, S_0$
    - Label the outputs of the multiplexer with the output variables
  - Value-fix the information inputs to the multiplexer using the values from the truth table (for don't cares, apply either 0 or 1)

One Bit Binary Adder By Multiplexer

- Implementing a 1-bit Binary Adder with a Dual 8-to-1-Line Multiplexer
  - Truth table

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

  - Circuit:

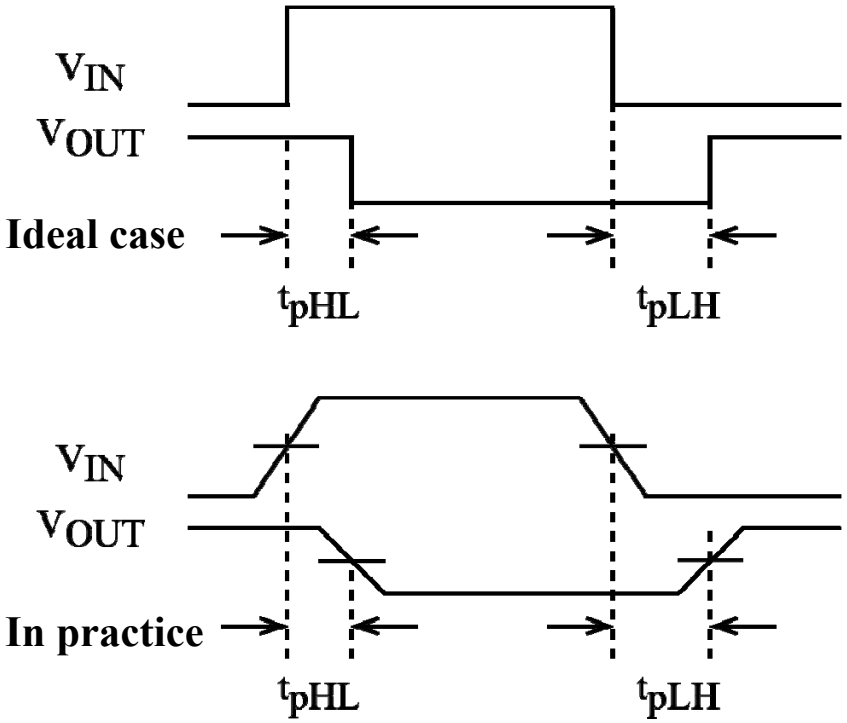


Overview

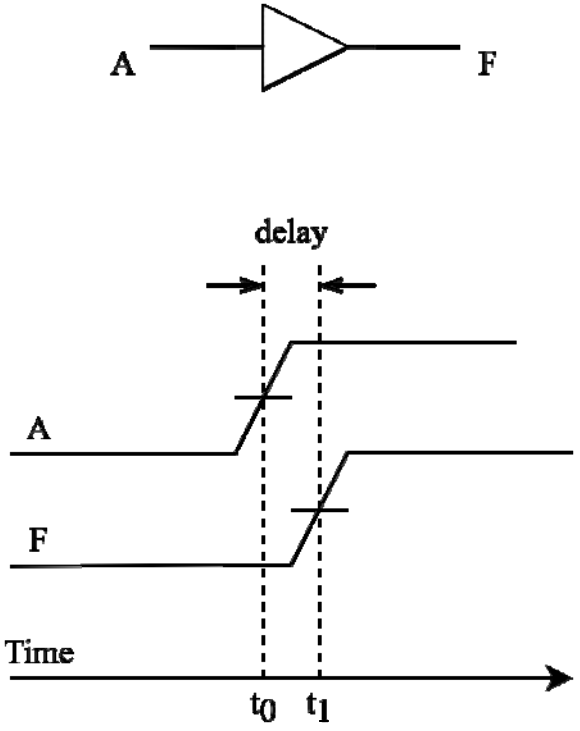
- Introduction to Verilog HDL
- About combinational logic circuits
- Some classic/basic designs
- Timing analysis

Timing Analysis

- Circuit delay



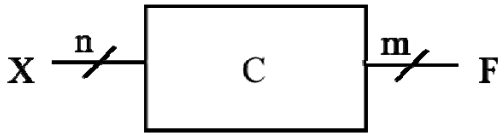
Delay of the invertor



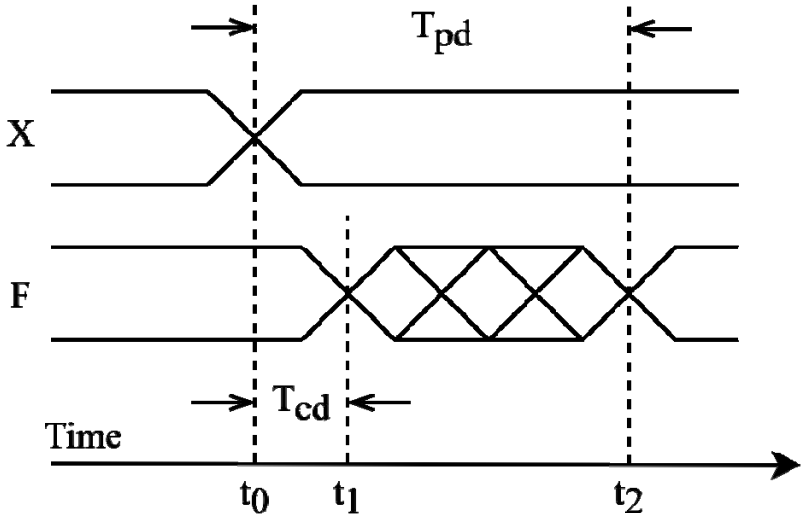
Rise-time delay of the transmission gate

Timing Analysis (cont'd)

- Propagation delay
  - T_{pd} = max delay from input to output
- Contamination delay
  - T_{cd} = min delay from input to output

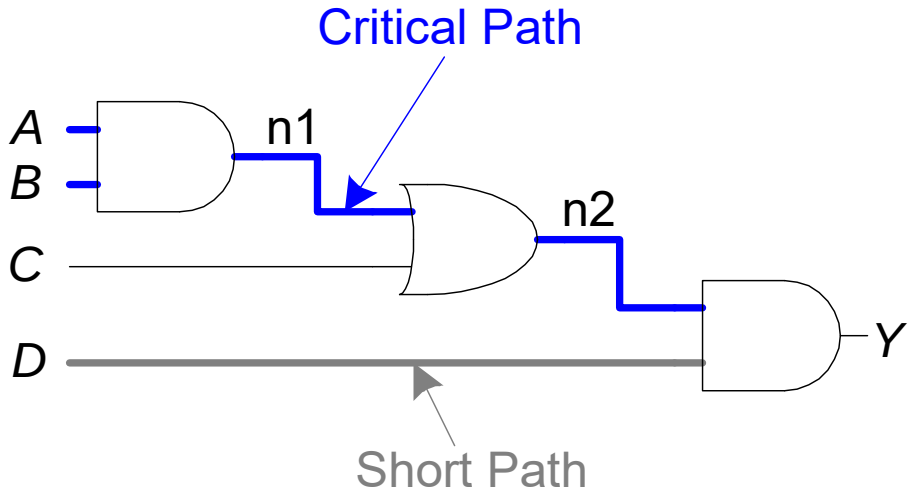


- Reasons why T_{pd} and T_{cd} may be different:
  - Different rising and falling delays
  - Multiple inputs and outputs, some of which are faster than others
  - Circuits slow down when hot and speed up when cold



Timing Analysis (cont'd)

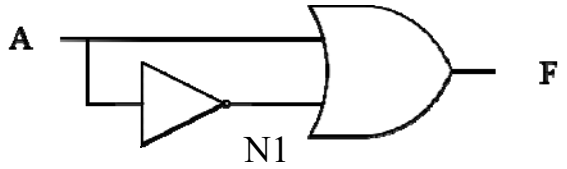
- The critical (longest) path
  - T_{pd} of the circuit = $\sum$ all T_{pd} of circuit elements along the critical path
- The shortest path
  - T_{cd} of the circuit = $\sum$ all T_{cd} of circuit elements along the shortest path



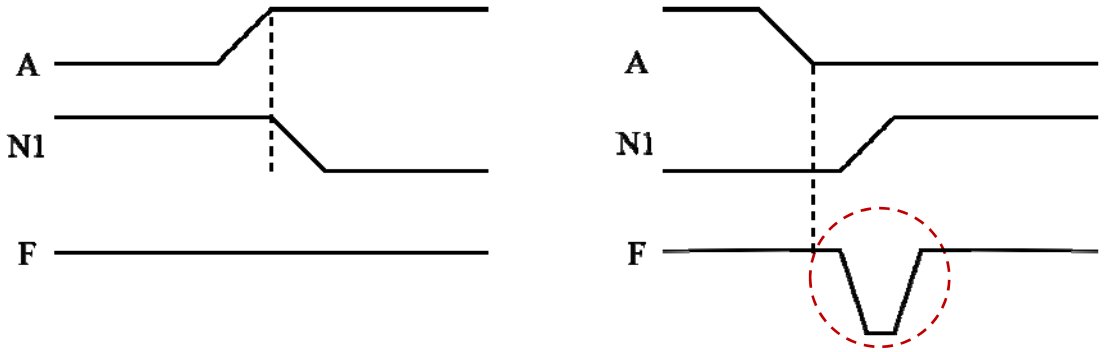
$$T_{pd} = 2T_{pd\_AND} + T_{pd\_OR} \qquad T_{cd} = T_{cd\_AND}$$

Timing Analysis (cont'd)

- Race hazard
  - Glitch: when a single input change causes multiple output changes



Race hazard in a circuit



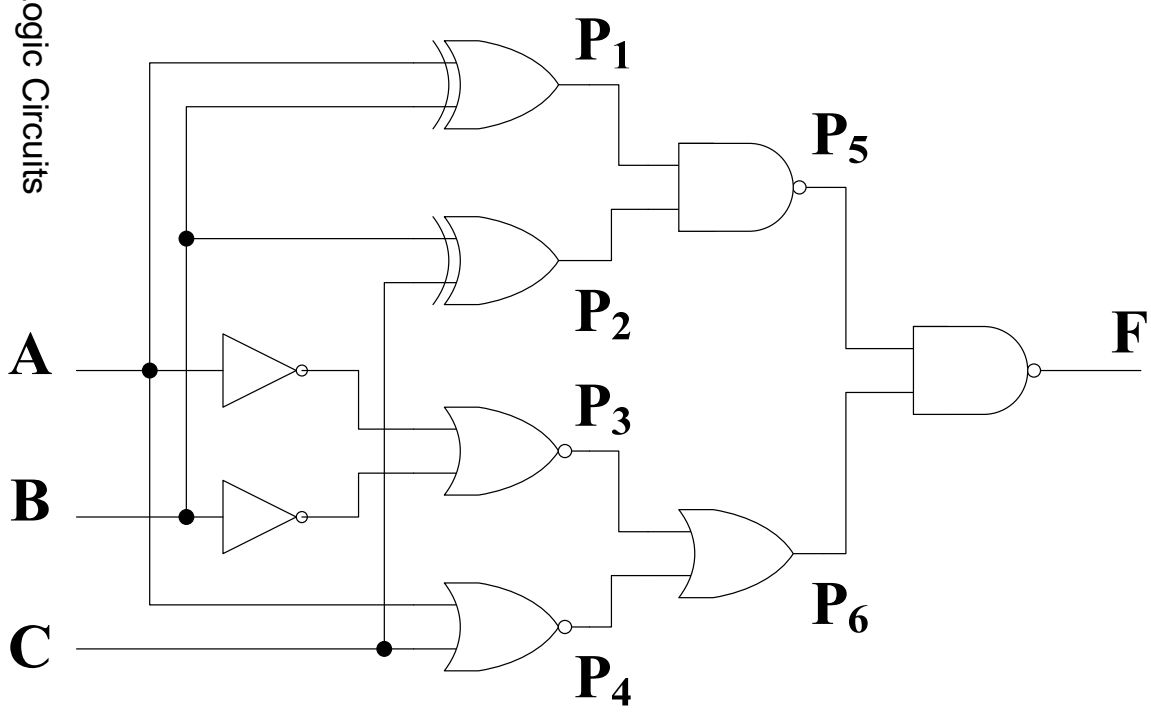
No glitch **Glitch**

Analysis of Logic Circuits

1. Write the Boolean function for the circuit
   - begin with the input signal
   - Define the relationship of each gate
   - optimization
2. Derive a truth table
   - Define the relationships between the inputs and outputs,
3. Functional Analysis
   - Define the function of each signal and the whole circuit
   - Draw the timing diagram of the circuit
4. Verification
   - Verify the correctness of the final design

Analyze the Function of the Logic Circuit

List the Boolean functions for all signals



$$P_1 = A \oplus B$$

$$P_2 = B \oplus C$$

$$P_3 = \overline{\overline{A} + \overline{B}} = AB$$

$$P_4 = \overline{A + C}$$

$$P_5 = \overline{P_1 P_2} = \overline{(A \oplus B) \cdot (B \oplus C)}$$

$$P_6 = P_3 + P_4 = AB + \overline{A + C}$$

$$F = \overline{P_5 \cdot P_6} = \overline{\overline{(A \oplus B) \cdot (B \oplus C)} \cdot (AB + \overline{A + C})}$$

Boolean Function Simplification

$$F = (A \oplus B)(B \oplus C) + \overline{AB + \overline{A + C}}$$

$$= (A\overline{B} + \overline{A}B)(B\overline{C} + \overline{B}C) + (\overline{A} + \overline{B})(A + C)$$

$$= A\overline{B}C + \overline{A}B\overline{C} + \overline{A}C + A\overline{B} + \overline{B}C$$

$$= \overline{A}B\overline{C} + \overline{A}C + A\overline{B} + \overline{B}C$$

$$= \overline{A}C + A\overline{B} + \overline{A}B\overline{C}$$

$$= \overline{A}(C + B\overline{C}) + A\overline{B}$$

$$= \overline{A}C + \overline{A}B + A\overline{B}$$

$$= \overline{A}C + (A \oplus B)$$

Derive the truth table

A	B	C	$A \oplus B$	$\overline{A}C$	F
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	1	0	0	0

Boolean Function Simpli

Derive the truth table

From the truth table, we can see that F=1 if A≠B or B<C.

$F = ($

$= (A$ $A+C)$

$= A\bar{B}$

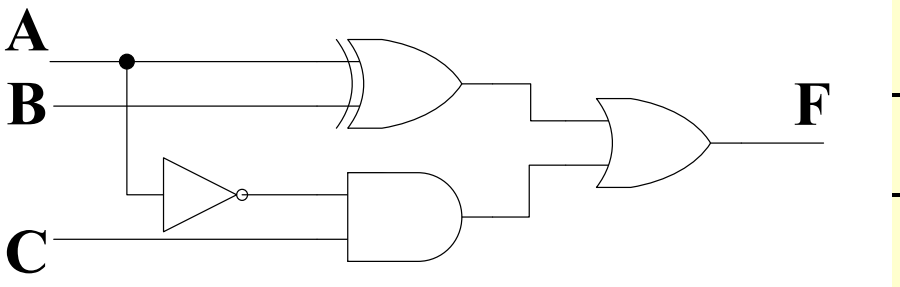
$= \bar{A}B\bar{C} + \bar{A}C + A\bar{B} + \bar{B}C$

$= AC + AB + A\bar{B}$

$= \bar{A}C + (A \oplus B)$

The original design is not the best. Use the following circuits instead.

A	B	C	$A \oplus B$	$\bar{A}C$	F
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	1	1
				0	1
				0	1
				0	0
1	1	1	0	0	0



Summary

- Introduction to Verilog HDL
- About combinational logic circuits
- Some classic/basic designs
- Timing analysis