Review on Digital Design (Combinational Circuits)

University of South Carolina

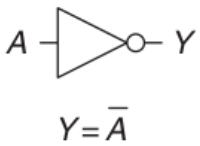
Introduction to Computer Architecture
Fall, 2024
Mehdi Yaghouti

Molinaroli College of Engineering and Computing
UNIVERSITY OF SOUTH CAROLINA

Fundamental Logic Gates

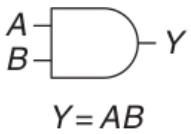
- **Fundamental**: AND, OR, and NOT are the core building blocks of digital logic.
- **Universal**: They can implement all possible logical operations.

NOT



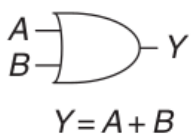
A	Y
0	1
1	0

AND



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR



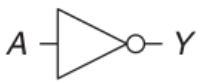
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Boolean Algebra

- The set of logical axioms/rules governing the manipulation binary variables.

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

NOT



$Y = \overline{A}$

A	Y
0	1
1	0

AND

$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

$Y = A + B$

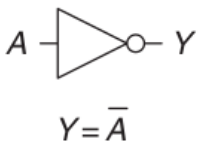
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Manipulation Rules

- Primitive rules

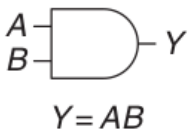
	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

NOT



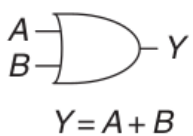
A	Y
0	1
1	0

AND



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

- Associativity

$$(B \bullet C) \bullet D = B \bullet (C \bullet D) \qquad (B + C) + D = B + (C + D)$$

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

- Associativity

$$(B \bullet C) \bullet D = B \bullet (C \bullet D) \qquad (B + C) + D = B + (C + D)$$

- Distributivity

$$(B \bullet C) + (B \bullet D) = B \bullet (C + D) \qquad (B + C) \bullet (B + D) = B + (C \bullet D)$$

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

- Associativity

$$(B \bullet C) \bullet D = B \bullet (C \bullet D) \qquad (B + C) + D = B + (C + D)$$

- Distributivity

$$(B \bullet C) + (B \bullet D) = B \bullet (C + D) \qquad (B + C) \bullet (B + D) = B + (C \bullet D)$$

- Covering

$$B \bullet (B + C) = B \qquad B + (B \bullet C) = B$$

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

- Associativity

$$(B \bullet C) \bullet D = B \bullet (C \bullet D) \qquad (B + C) + D = B + (C + D)$$

- Distributivity

$$(B \bullet C) + (B \bullet D) = B \bullet (C + D) \qquad (B + C) \bullet (B + D) = B + (C \bullet D)$$

- Covering

$$B \bullet (B + C) = B \qquad B + (B \bullet C) = B$$

- Combining

$$(B \bullet C) + (B \bullet \overline{C}) = B \qquad (B + C) \bullet (B + \overline{C}) = B$$

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

- Associativity

$$(B \bullet C) \bullet D = B \bullet (C \bullet D) \qquad (B + C) + D = B + (C + D)$$

- Distributivity

$$(B \bullet C) + (B \bullet D) = B \bullet (C + D) \qquad (B + C) \bullet (B + D) = B + (C \bullet D)$$

- Covering

$$B \bullet (B + C) = B \qquad B + (B \bullet C) = B$$

- Combining

$$(B \bullet C) + (B \bullet \overline{C}) = B \qquad (B + C) \bullet (B + \overline{C}) = B$$

- Consensus

$$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) = (B \bullet C) + (\overline{B} \bullet D)$$
$$(B + C) \bullet (\overline{B} + D) \bullet (C + D) = (B + C) \bullet (\overline{B} \bullet D)$$

Manipulation Rules

- Commutativity

$$B \bullet C = C \bullet B \qquad B + C = C + B$$

- Associativity

$$(B \bullet C) \bullet D = B \bullet (C \bullet D) \qquad (B + C) + D = B + (C + D)$$

- Distributivity

$$(B \bullet C) + (B \bullet D) = B \bullet (C + D) \qquad (B + C) \bullet (B + D) = B + (C \bullet D)$$

- Covering

$$B \bullet (B + C) = B \qquad B + (B \bullet C) = B$$

- Combining

$$(B \bullet C) + (B \bullet \overline{C}) = B \qquad (B + C) \bullet (B + \overline{C}) = B$$

- Consensus

$$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) = (B \bullet C) + (\overline{B} \bullet D)$$
$$(B + C) \bullet (\overline{B} + D) \bullet (C + D) = (B + C) \bullet (\overline{B} \bullet D)$$

- De Morgan's Theorem

$$\overline{B_0 \bullet B_1 \bullet B_2 \bullet \cdots} = \overline{B_0} + \overline{B_1} + \overline{B_2} + \cdots$$
$$\overline{B_0 + B_1 + B_2 + \cdots} = \overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \bullet \cdots$$

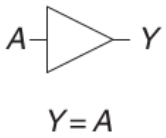
Manipulation Rules

- Theorems

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= (B \bullet C) + (\overline{B} \bullet D)$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B}_0 + \overline{B}_1 + \overline{B}_2 \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B}_0 \bullet \overline{B}_1 \bullet \overline{B}_2 \ldots)$	De Morgan's Theorem

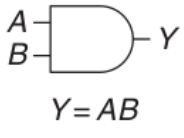
Common Logic Gates

BUF



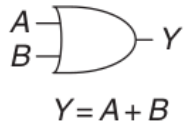
A	Y
0	0
1	1

AND



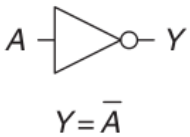
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR



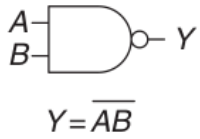
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT



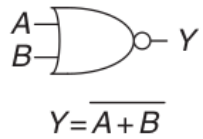
A	Y
0	1
1	0

NAND



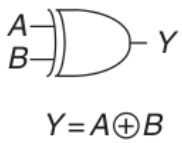
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR



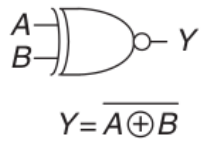
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XOR



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

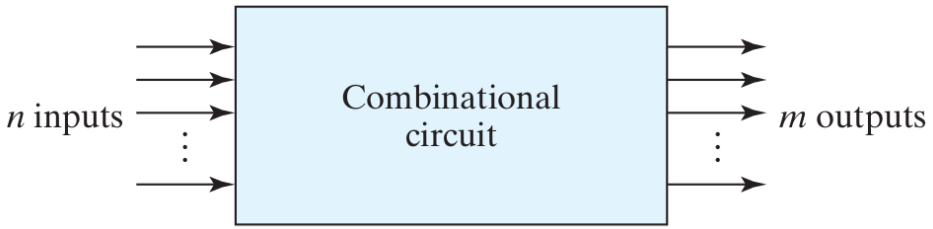
XNOR



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

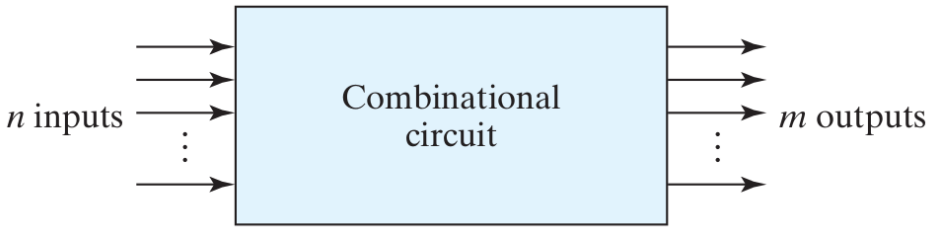
Digital Circuits

- Combinational Circuits
  - Input terminals
  - Output terminals
  - Memoryless
  - No Cyclic path
  - Functional specification
  - Timing specification

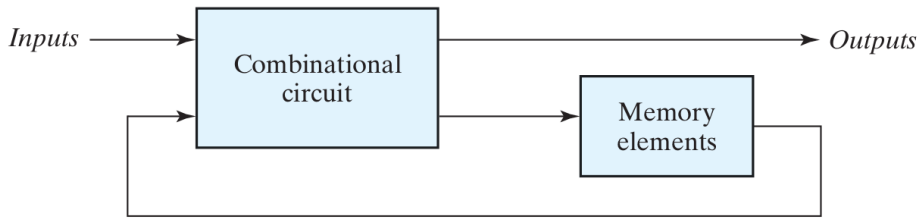


Digital Circuits

- Combinational Circuits
  - Input terminals
  - Output terminals
  - Memoryless
  - No Cyclic path
  - Functional specification
  - Timing specification

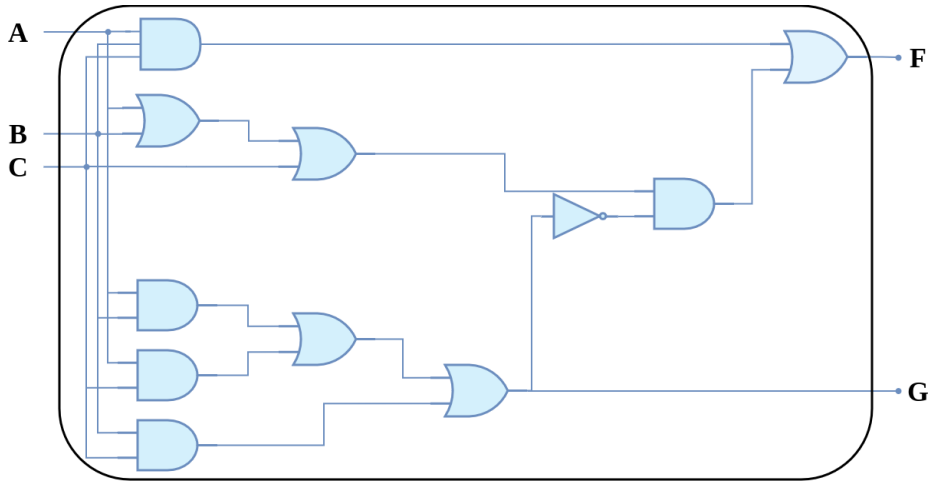


- Sequential Circuits
  - Input terminals
  - Output terminals
  - Has Memory
  - Synchronous/Asynchronous
  - Functional specification
  - Timing specification



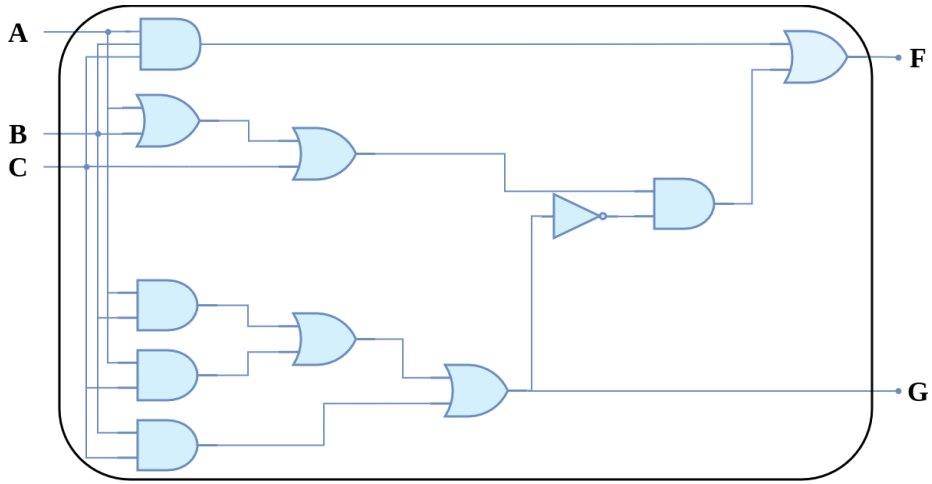
Combinational Circuits

- Functional Specification
  - Circuit Diagram



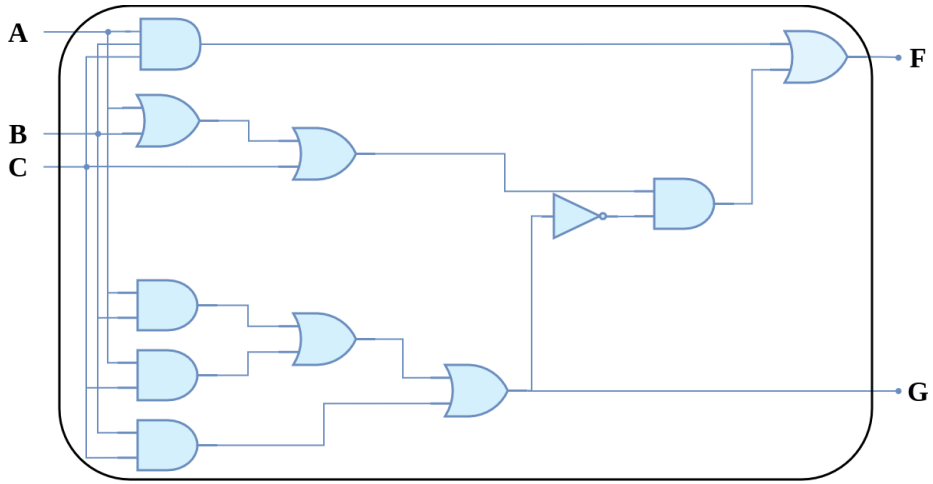
Combinational Circuits

- Functional Specification
  - Circuit Diagram
  - Boolean Equation



Combinational Circuits

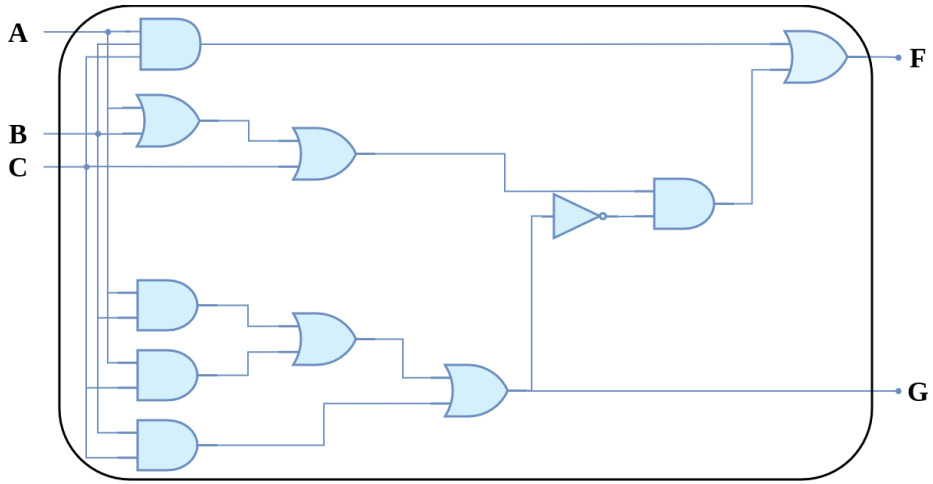
- Functional Specification
  - Circuit Diagram
  - Boolean Equation



$$F = ABC + (A + B + C)(\overline{AB + AC + BC})$$
$$G = (AB + AC + BC)$$

Combinational Circuits

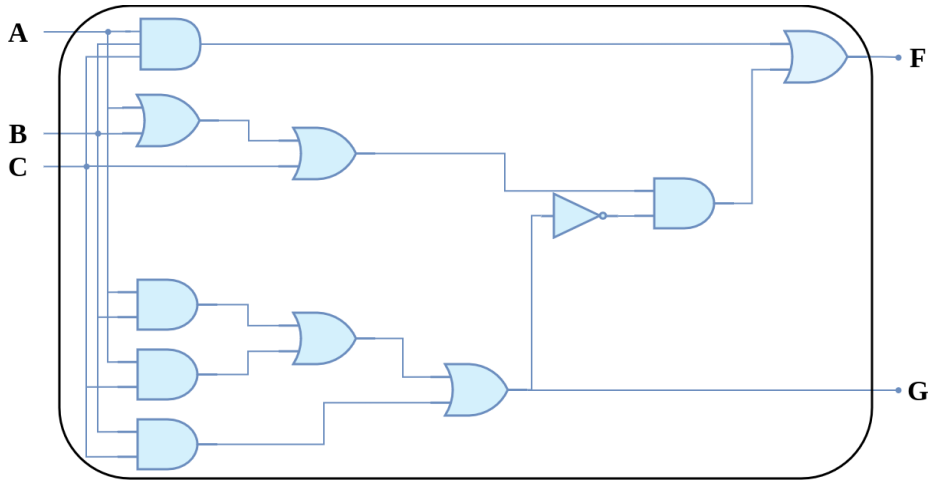
- Functional Specification
  - Circuit Diagram
  - Boolean Equation
  - Truth Table



$$F = ABC + (A + B + C)(\overline{AB + AC + BC})$$
$$G = (AB + AC + BC)$$

Combinational Circuits

- Functional Specification
  - Circuit Diagram
  - Boolean Equation
  - Truth Table

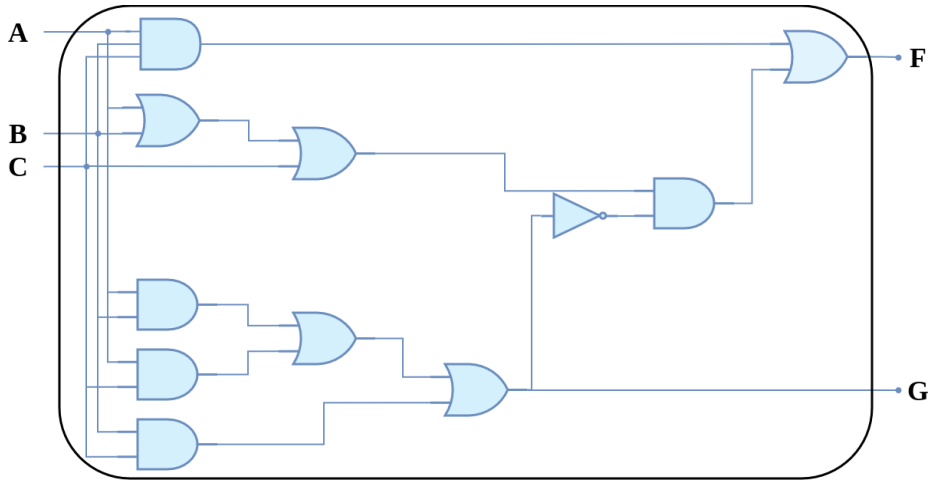


$$F = ABC + (A + B + C)(\overline{AB + AC + BC})$$
$$G = (AB + AC + BC)$$

A	B	C	G	F
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

Combinational Circuits

- Functional Specification
  - Circuit Diagram
  - Boolean Equation
  - Truth Table



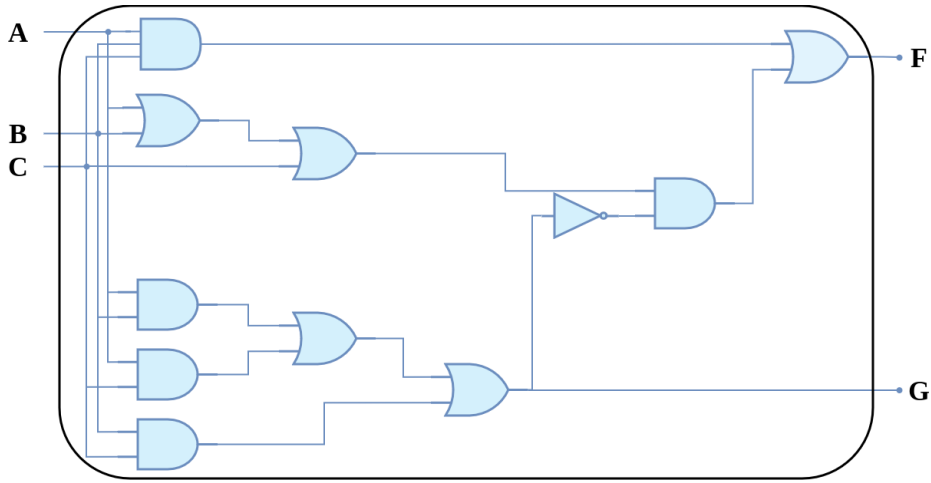
$$F = ABC + (A + B + C)(\overline{AB + AC + BC})$$
$$G = (AB + AC + BC)$$

A	B	C	G	F
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

$$F = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$
$$G = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Combinational Circuits

- Functional Specification
  - Circuit Diagram
  - Truth Table
  - Boolean Equation
  - Hardware Description Language



$$F = ABC + (A + B + C)(\overline{AB + AC + BC})$$
$$G = (AB + AC + BC)$$

A	B	C	G	F
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

$$F = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$
$$G = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Logic Minimization / K-Maps

- Boolean Algebra as simplification rules

$$G = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Logic Minimization / K-Maps

- Boolean Algebra as simplification rules

$$
\begin{aligned}
G &= \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \\
&= \left(\bar{A}BC + ABC\right) + \left(A\bar{B}C + ABC\right) + \left(AB\bar{C} + ABC\right)
\end{aligned}
$$

Logic Minimization / K-Maps

- Boolean Algebra as simplification rules

$$
\begin{aligned}
G &= \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \\
&= \left(\bar{A}BC + ABC\right) + \left(A\bar{B}C + ABC\right) + \left(AB\bar{C} + ABC\right) \\
&= \left(\bar{A} + A\right)BC + A\left(\bar{B} + B\right)C + AB\left(\bar{C} + C\right)
\end{aligned}
$$

Logic Minimization / K-Maps

- Boolean Algebra as simplification rules

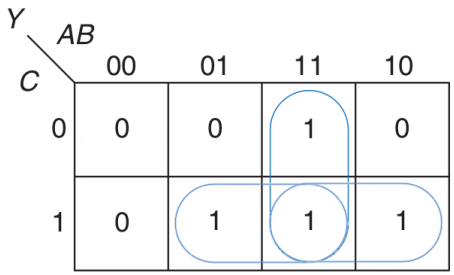
$$
\begin{aligned}
G &= \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \\
&= \left(\bar{A}BC + ABC\right) + \left(A\bar{B}C + ABC\right) + \left(AB\bar{C} + ABC\right) \\
&= \left(\bar{A} + A\right)BC + A\left(\bar{B} + B\right)C + AB\left(\bar{C} + C\right) \\
&= BC + AC + AB
\end{aligned}
$$

Logic Minimization / K-Maps

- Boolean Algebra as simplification rules

$$
\begin{aligned}
G &= \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \\
&= \left(\bar{A}BC + ABC\right) + \left(A\bar{B}C + ABC\right) + \left(AB\bar{C} + ABC\right) \\
&= \left(\bar{A} + A\right)BC + A\left(\bar{B} + B\right)C + AB\left(\bar{C} + C\right) \\
&= BC + AC + AB
\end{aligned}
$$

A	B	C	G
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Hardware Description Languages

- Schematic-level circuit design is labor-intensive and error-prone
- Manual simplification of truth tables and FSMs is cumbersome
- Started from 1990s, designers shifted to higher abstraction levels
- CAD tools now optimize gates from logical functions
- HDLs are used for specifications
- Leading HDLs: Verilog and VHDL
- SystemVerilog extends Verilog with advanced features while maintaining backward compatibility
- Two main phases:
  - **Simulation**
    - Testing the module in a software environment to verify correct behavior
  - **Synthesis**
    - Converting the high-level design into a gate-level representation
    - Synthesis varies by the target platform
- **The Best way to learn HDL is by examples**

SystemVerilog

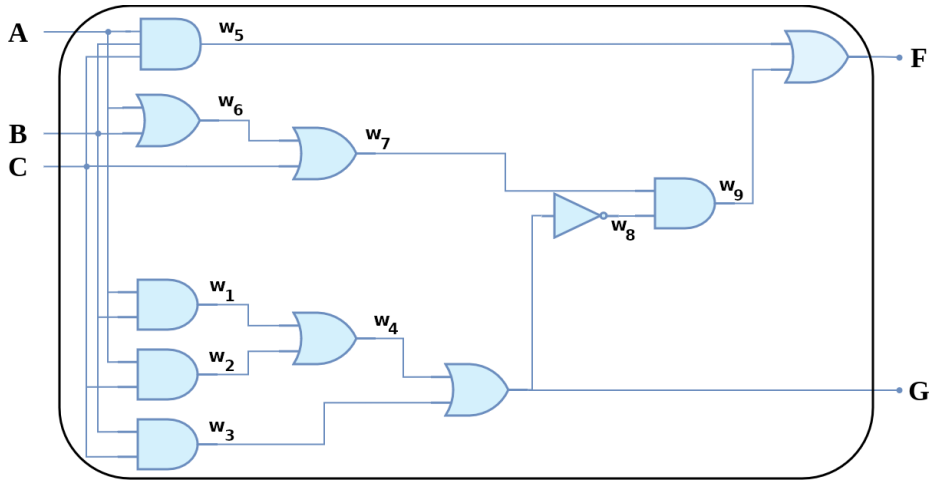
- Combinational circuit as an input/output module

```
module comb_circuit (    input  logic  a, b, c,
                         output logic  f, g    );




endmodule
```

SystemVerilog

- Declaring internal wires

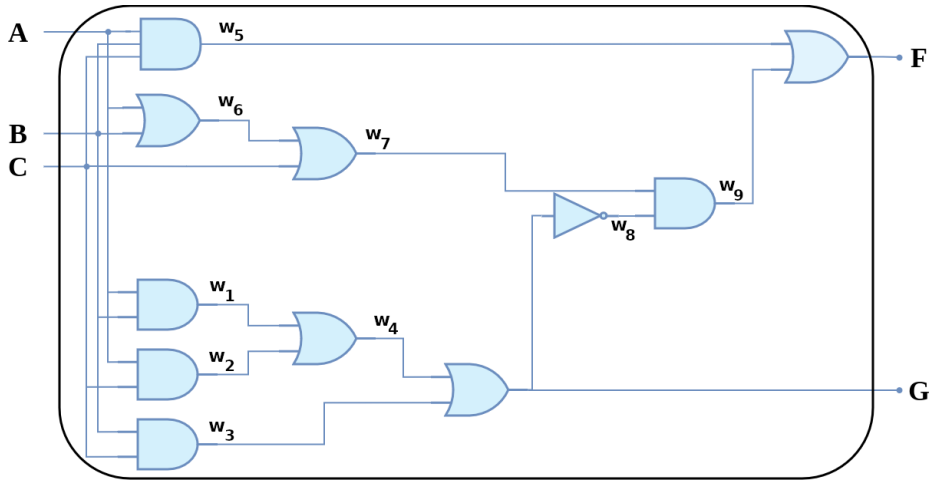
```
module comb_circuit (     input  logic a, b, c,
                          output logic  f, g  );

logic w1, w2, w3, w4;
logic w5, w6, w7, w8, w9;



endmodule
```



SystemVerilog

- Logical operation in SystemVerilog

```
module comb_circuit (    input  logic a, b, c,
                         output logic  f, g  );
  logic w1, w2, w3, w4;

  assign w1 = a & b;      // AND operation
  assign w2 = a & c;      // AND operation
  assign w3 = b & c;      // AND operation
  assign w4 = w1 | w2;    // OR operation
  assign g = w4 | w3;     // OR operation

  logic w5, w6, w7, w8, w9;

  assign w5 = a & b & c; // AND operation
  assign w6 = a | b;      // OR operation
  assign w7 = w6 | c;     // OR operation
  assign w8 = ~g;         // NOT operation
  assign w9 = w7 & w8;    // AND operation
  assign f  = w5 | w9;    // OR operation
endmodule
```



SystemVerilog Operators

- Operations precedence in SystemVerilog

```
module comb_circuit (    input  logic a, b, c,
                         output logic  f, g  );

    assign g =  (a & b) | (a & c) | ( b & c);
    assign f = ( a & b & c ) | (( a | b | c) & ~g);

endmodule
```

Op	Meaning
~	NOT
*, /, %	MUL, DIV, MOD
+, -	PLUS, MINUS
<<, >>	Logical Left/Right Shift
<<<, >>>	Arithmetic Left/Right Shift
<, <=, >, >=	Relative Comparison
==, !=	Equality Comparison
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	Conditional

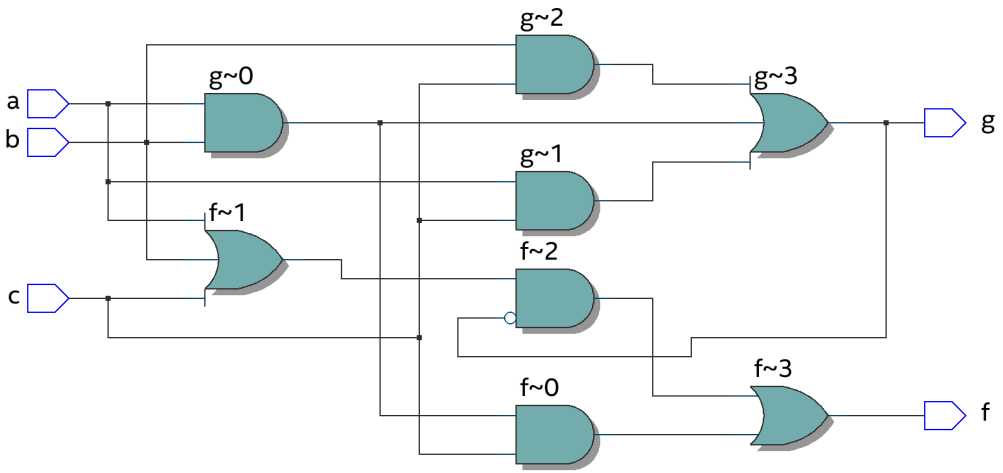
- What would happen if we change the order of the two assign lines?

SystemVerilog

- SystemVerilog code

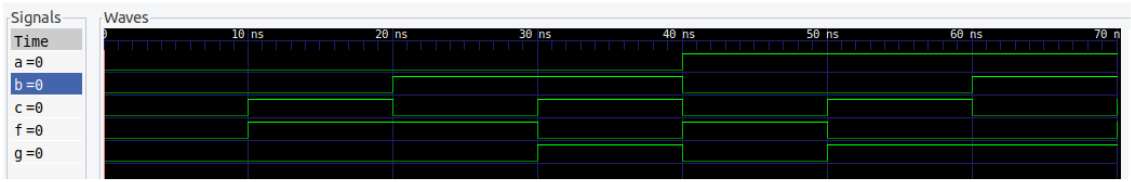
```
module comb_circuit (    input  logic a, b, c,
                         output logic  f, g  );

    assign g =  (a & b) | (a & c) | ( b & c);
    assign f = ( a & b & c ) | (( a | b | c) & ~g);

endmodule
```

- Synthesized circuit



SystemVerilog (Testbench)

- Testbench

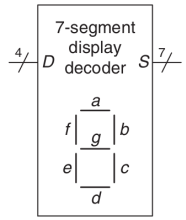


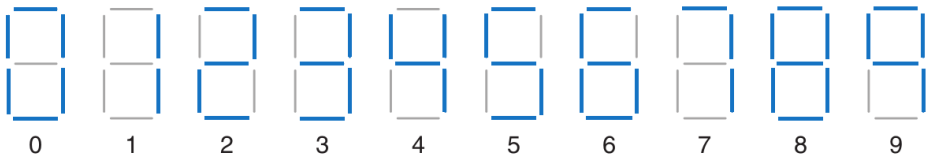

```
`timescale 1ns/1ps
module comb_circuit_tb;

    // Inputs
    logic a, b, c;

    // Outputs
    logic f, g;

    // Instantiate the Unit Under Test (DUT)
    comb_circuit dut (.a(a), .b(b), .c(c), .f(f), .g(g) );

    // Test sequence
    initial begin
        // Setup waveform dumping
        $dumpfile("comb_circuit.vcd");   // Name of the VCD file
        $dumpvars(0, comb_circuit_tb);   // Dump all variables in the testbench
        // Display the header
        $display("A B C | F G");
        $display("------------");
        // Apply test vectors using a loop
        for (int i = 0; i < 8; i++) begin
            {a, b, c} = i;
            #10 $display("%b %b %b | %b %b", a, b, c, f, g);
        end
        // End simulation
        $stop;
    end
endmodule
```

```
A B C | F G
------------
0 0 0 | 0 0
0 0 1 | 1 0
0 1 0 | 1 0
0 1 1 | 0 1
1 0 0 | 1 0
1 0 1 | 0 1
1 1 0 | 0 1
1 1 1 | 1 1
```

Seven-Segment Decoder

- 4-inputs 7-outputs
- Each segment is only a function of 4-inputs

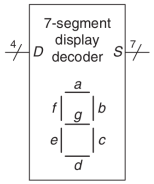




$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

Seven-Segment Decoder

- 4-inputs 7-outputs
- Each segment is only a function of 4-inputs

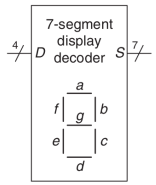


$$S_a = \overline{D_3}\,\overline{D_2}\,\overline{D_1}\,\overline{D_0} + \overline{D_3}\,\overline{D_2}\,D_1\,\overline{D_0} + \overline{D_3}\,\overline{D_2}\,D_1\,D_0 \overline{D_3}\,D_2\,\overline{D_1}\,D_0$$
$$+ \overline{D_3}\,D_2\,D_1\,\overline{D_0} + \overline{D_3}\,D_2\,D_1\,D_0 + D_3\,\overline{D_2}\,\overline{D_1}\,\overline{D_0} + D_3\,\overline{D_2}\,\overline{D_1}\,D_0$$

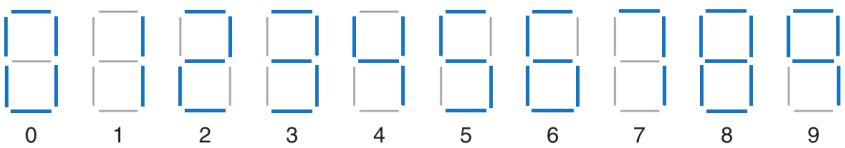
$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

Seven-Segment Decoder

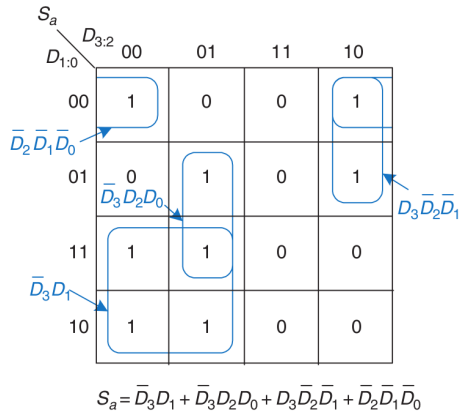
- 4-inputs 7-outputs
- Each segment is only a function of 4-inputs



$$S_a = \overline{D_3}\,\overline{D_2}\,\overline{D_1}\,\overline{D_0} + \overline{D_3}\,\overline{D_2}\,D_1\,\overline{D_0} + \overline{D_3}\,\overline{D_2}\,D_1\,D_0\overline{D_3}\,D_2\,\overline{D_1}\,D_0$$
$$+ \overline{D_3}\,D_2\,D_1\,\overline{D_0} + \overline{D_3}\,D_2\,D_1\,D_0 + D_3\,\overline{D_2}\,\overline{D_1}\,\overline{D_0} + D_3\,\overline{D_2}\,\overline{D_1}\,D_0$$



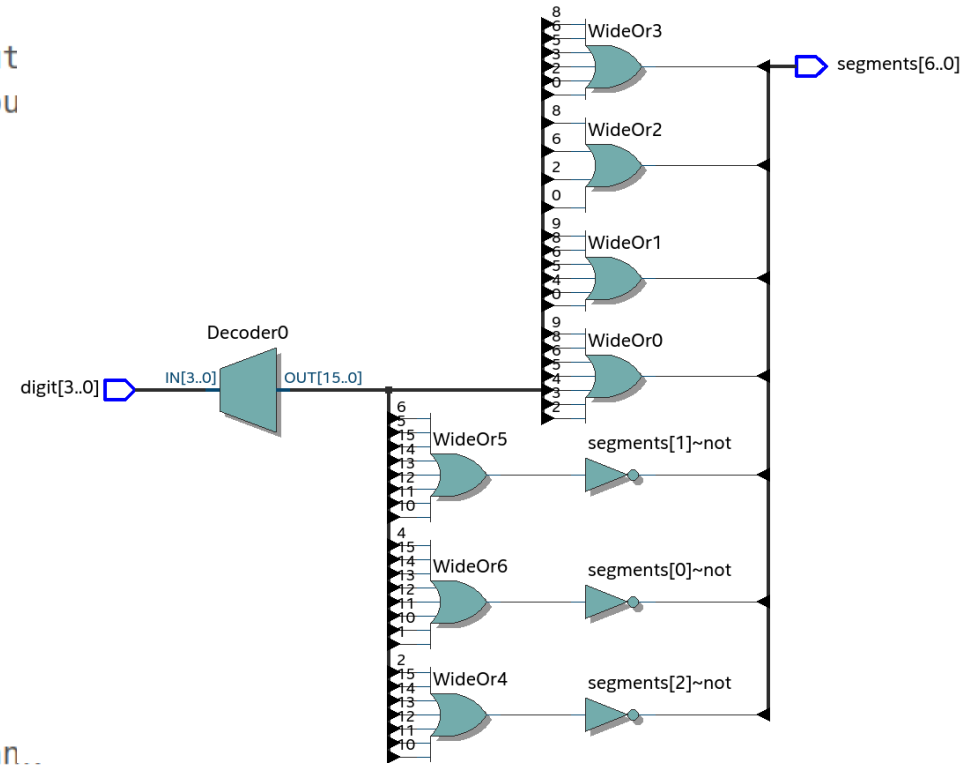
$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0



SystemVerilog

- Bundle notation $[n:0]$
- *always_comb* and *case* statement

```
module seven_segments_decoder (
    input  logic [3:0] digit,     // 4-bit input
    output logic [6:0] segments   // 7-bit outpu
);

    always_comb begin
        case (digit)
            4'b0000: segments = 7'b0111111; // 0
            4'b0001: segments = 7'b0000110; // 1
            4'b0010: segments = 7'b1011011; // 2
            4'b0011: segments = 7'b1001111; // 3
            4'b0100: segments = 7'b1100110; // 4
            4'b0101: segments = 7'b1101101; // 5
            4'b0110: segments = 7'b1111101; // 6
            4'b0111: segments = 7'b0000111; // 7
            4'b1000: segments = 7'b1111111; // 8
            4'b1001: segments = 7'b1100111; // 9
            default: segments = 7'b0000000; // Blan
        endcase
    end

endmodule
```

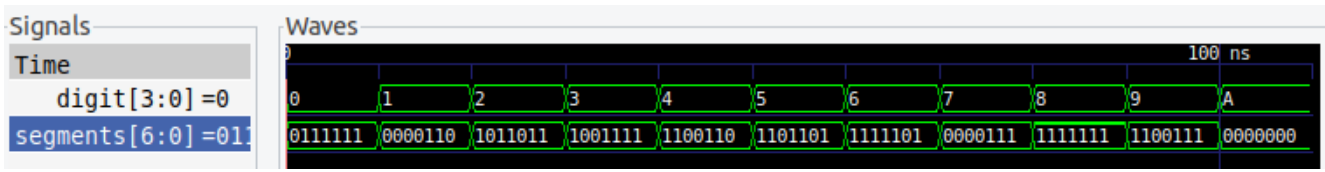


SystemVerilog

- Testbench

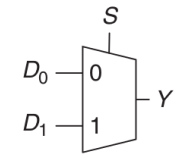
```
`timescale 1ns/1ps
module seven_segments_decoder_tb;
    logic [3:0] digit;      // 4-bit input
    logic [6:0] segments;   // 7-segment output

    seven_segments_decoder dut (
        .digit(digit),
        .segments(segments)
    );
    initial begin
        $dumpfile("seven_segment_decoder.vcd");
        $dumpvars(0, seven_segments_decoder_tb);

        $display("Time  | Digit | Segments");
        $display("------------------------");

        for (int i = 0; i < 10; i++) begin
            digit = i;
            #10;
            $display("%0dns |   %0d   | %b", $time, digit, segments);
        end
        digit = 4'b1010;  // Invalid input (A)
        #10;
        $display("%0dns |   %0d   | %b", $time, digit, segments);
        $stop;
    end
endmodule
```

Time	Digit	Segments
10ns	0	0111111
20ns	1	0000110
30ns	2	1011011
40ns	3	1001111
50ns	4	1100110
60ns	5	1101101
70ns	6	1111101
80ns	7	0000111
90ns	8	1111111
100ns	9	1100111
110ns	10	0000000

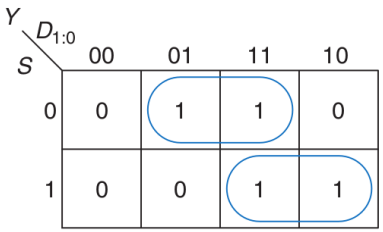


Multiplexer

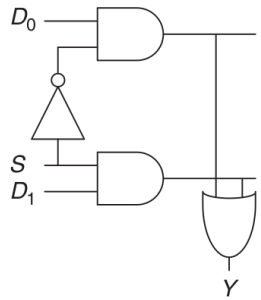
- A $N:1$ Mux chooses 1 out of N inputs
- 2:1 Mux as a combinational circuit



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

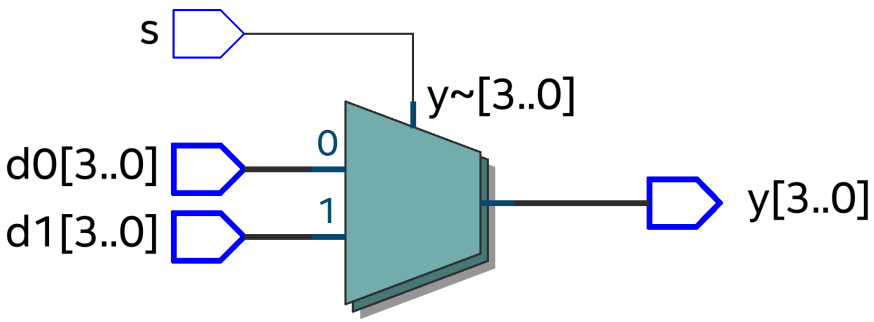


$Y = D_0\bar{S} + D_1S$



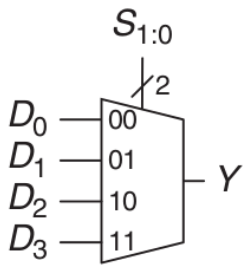
- A $2:1$ Mux can be easily implemented using *ternary operator*

```
module mux2(input logic [3:0] d0, d1,
            input logic s,
            output logic [3:0] y);

    assign y = s ? d1 : d0;
endmodule
```

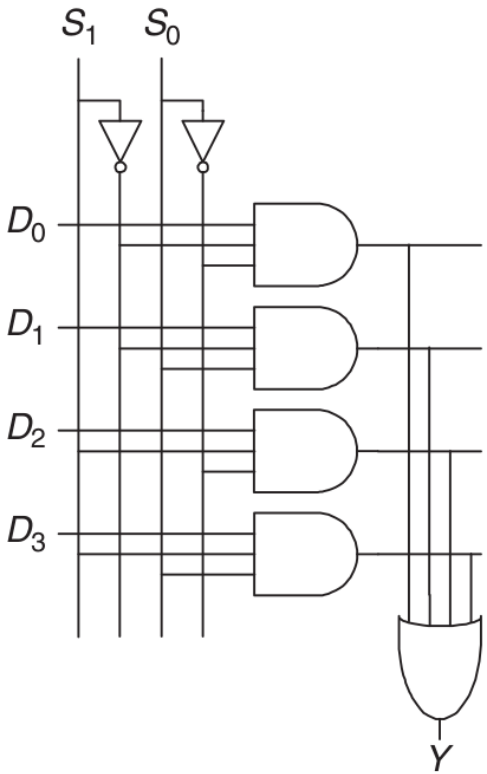


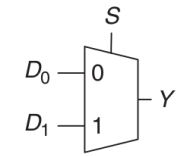
Multiplexer

- A $N:1$ Mux chooses 1 out of N inputs
- $2:1$ Mux as a combinational circuit
- $4:1$ Mux implemented based on description

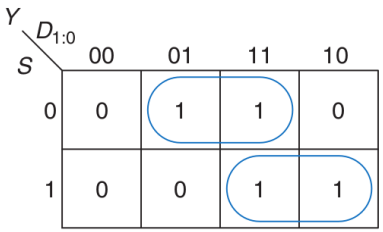


S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

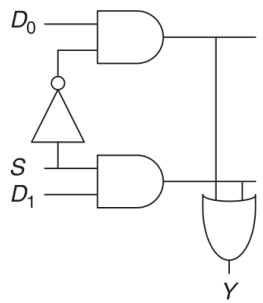




S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

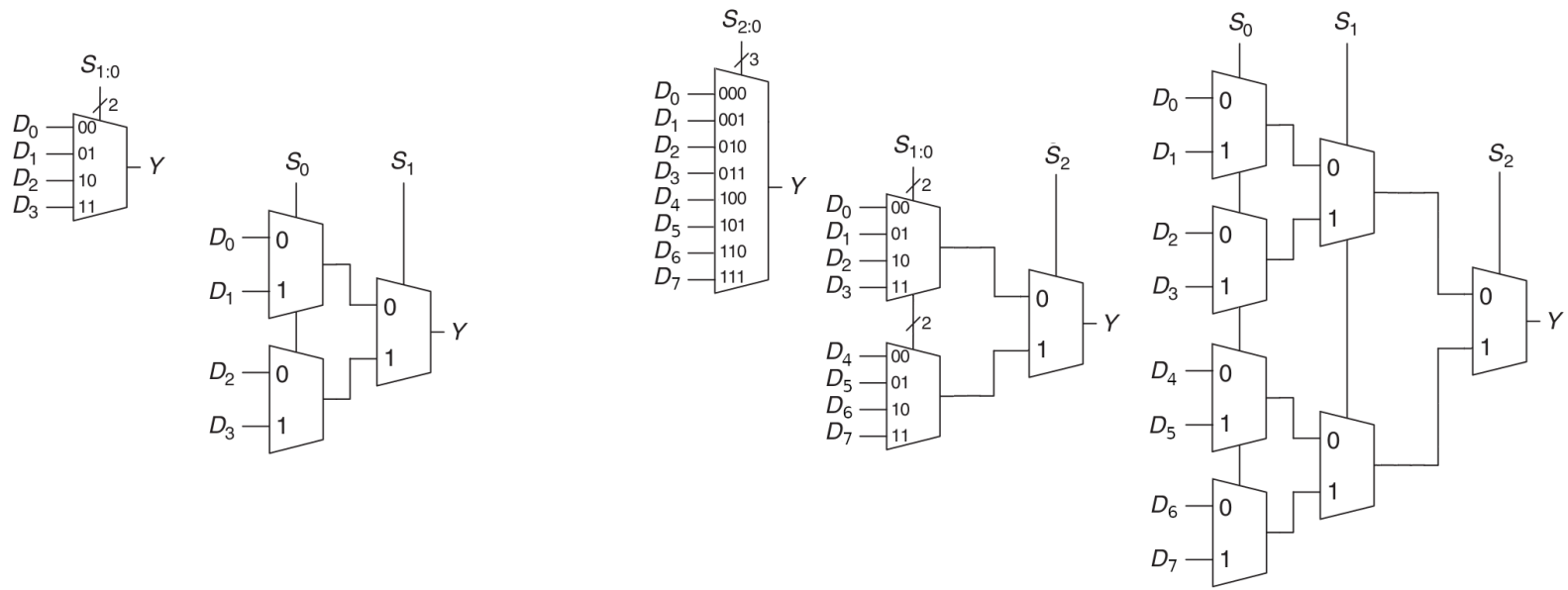


$Y = D_0\bar{S} + D_1 S$



Multiplexer

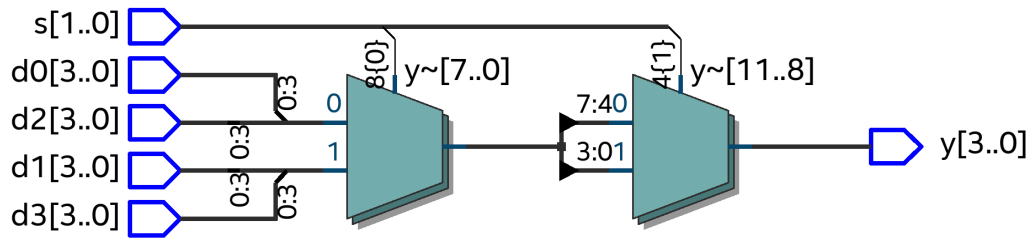
- A $2^n : 1$ Mux can be implemented as a binary tree
- n cascaded layer is needed



SystemVerilog

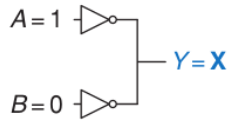
- *ternary* operators can be combined to implement larger Mux

```
module mux4(input logic [3:0] d0, d1, d2, d3,
            input logic [1:0] s,
            output logic [3:0] y);
  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```



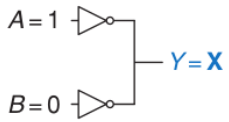
Tristate Buffer

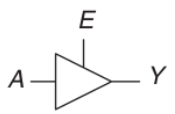
- X designates an illegal or unknown logic value



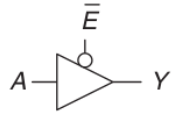
Tristate Buffer

- X designates an illegal or unknown logic value
- Z designates a high impedance or floating terminal
- X and Z are neither 0 nor 1 logical level
- Tristate buffer possible output states: 1, 0, and Z





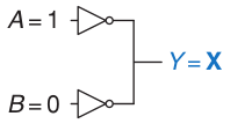
E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

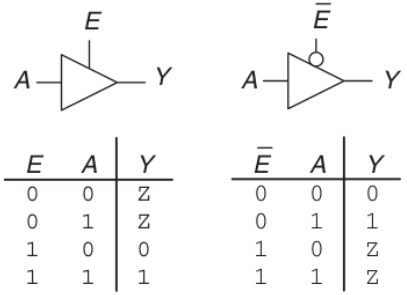


$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Tristate Buffer

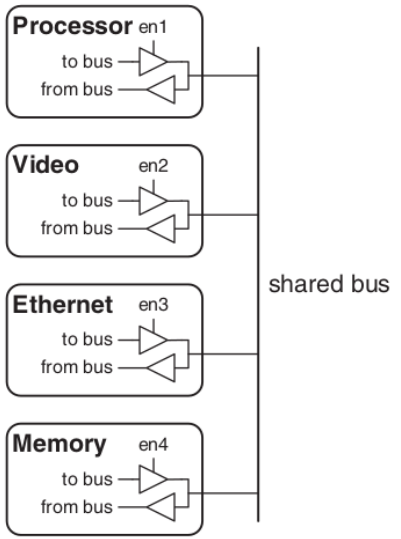
- X designates an illegal or unknown logic value
- Z designates a high impedance or floating terminal
- X and Z are neither 0 nor 1 logical level
- Tristate buffer possible output states: 1, 0, and Z
- They are commonly used for bus connections





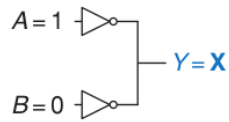
E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

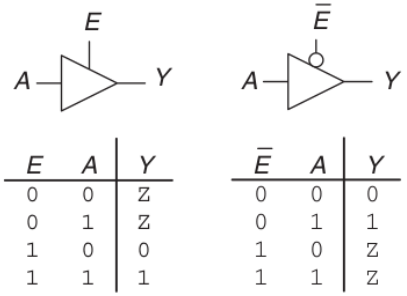
Ē	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z



Tristate Buffer

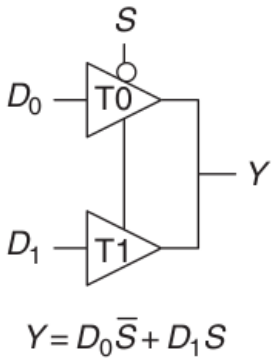
- X designates an illegal or unknown logic value
- Z designates a high impedance or floating terminal
- X and Z are neither 0 nor 1 logical level
- Tristate buffer possible output states: 1, 0, and Z
- They are commonly used for bus connections
- Mux implementation with Tristate buffers

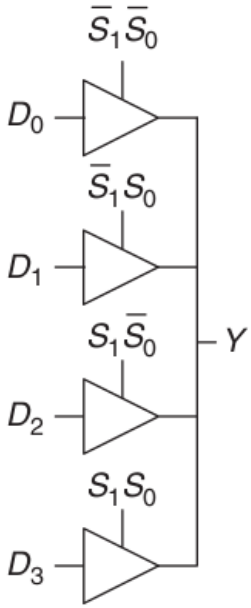


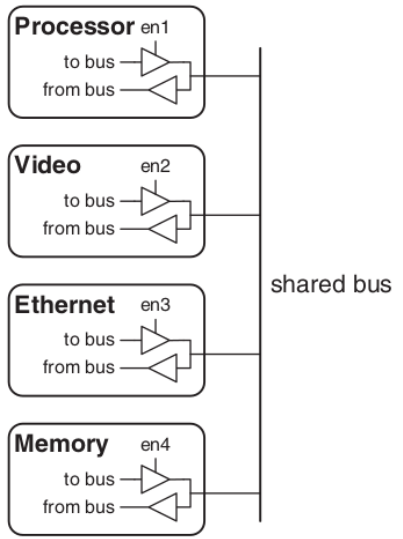


E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Ē	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z



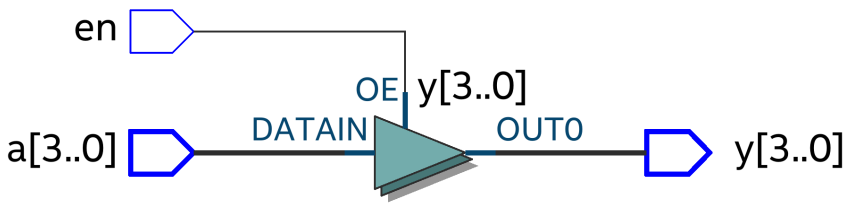




Tristate Buffer

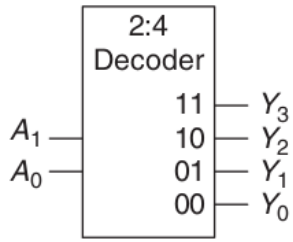
- Tristate variables must be declared as *tri* rather than *logic*

```
module tristate(input logic [3:0] a,
                input logic en,
                output tri [3:0] y);

    assign y = en ? a : 4'bz;

endmodule
```

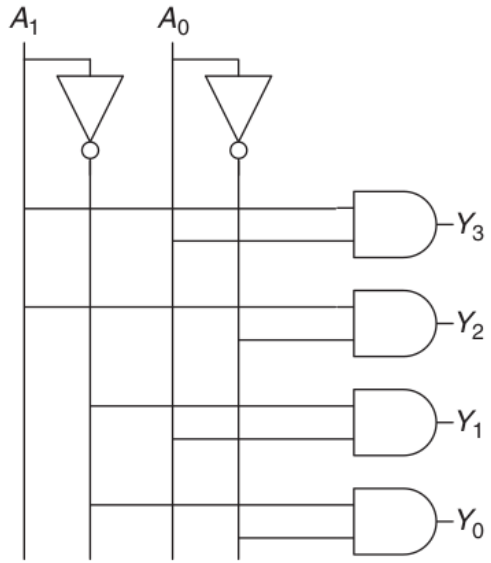


Decoder

- Decodes a N-bits combination into a one-hot output
- A $N : 2^N$ decoder has N inputs and 2^N outputs

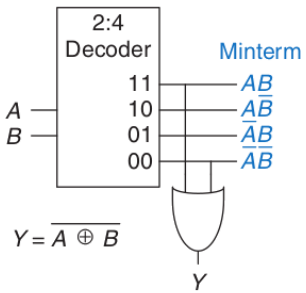


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



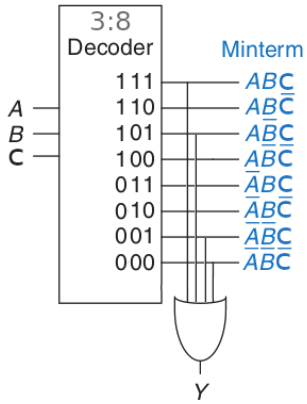
Decoder

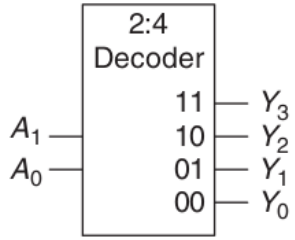
- Decodes a N-bits combination into a one-hot output
- A $N : 2^N$ decoder has N inputs and 2^N outputs
- Sum-of-products implemented as a decoder combined with an OR gate



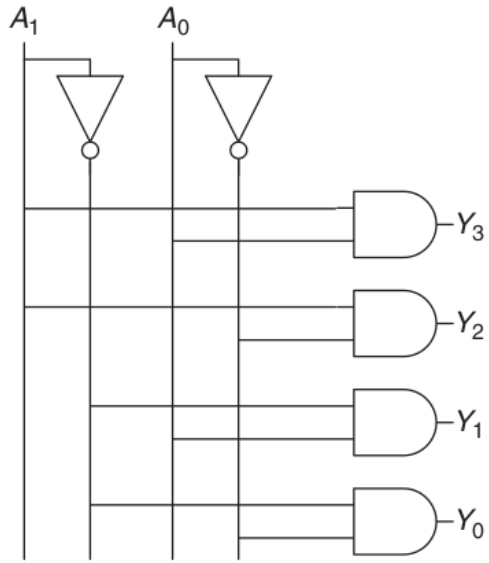
A	B	C	
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

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



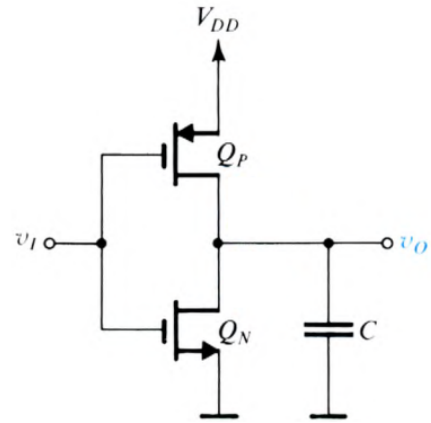


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

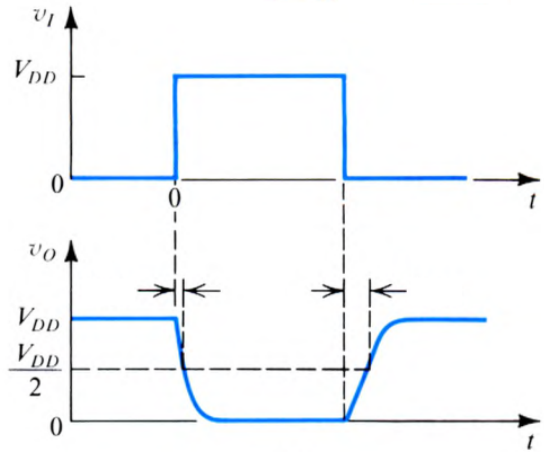


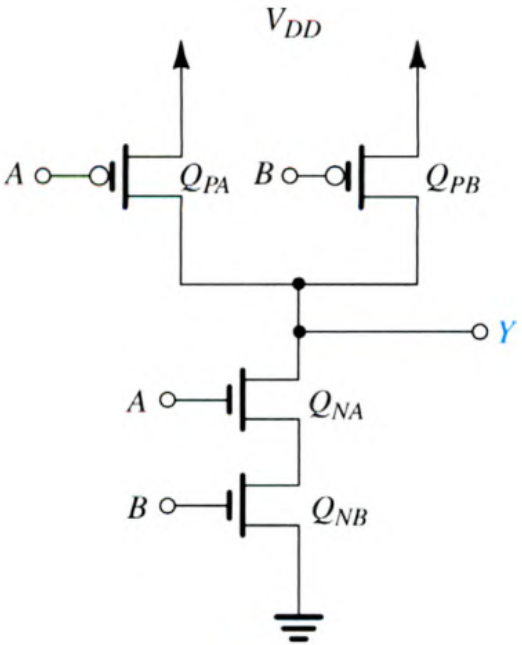
Under the hood

- Logic gates are made of transistors
- CMOS is the common technology
- NAND, NOR and NOT gates
- Transistors have intrinsic delay and non-perfections

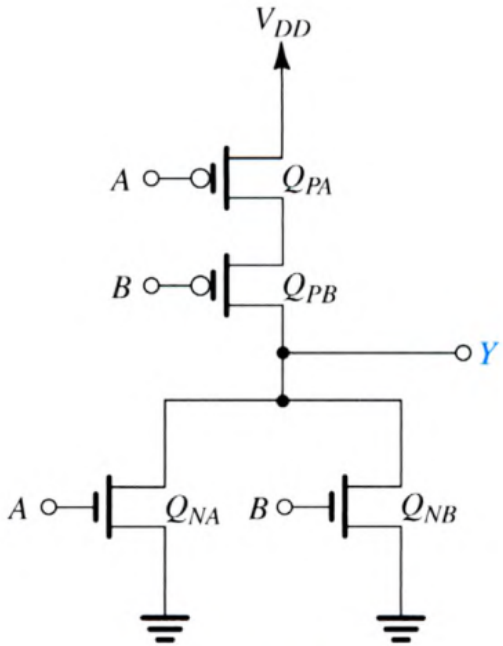


$Y = \overline{A}$





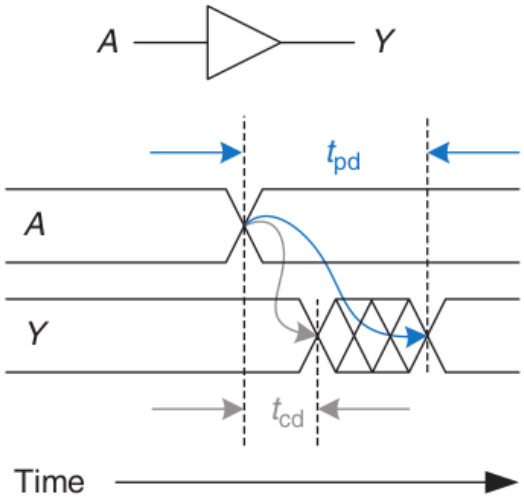
$Y = \overline{AB}$



$Y = \overline{A + B}$

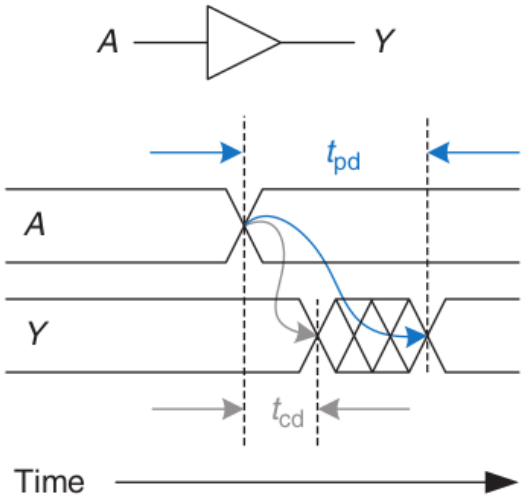
Timing Specification

- Propagation and Contamination delays
- We consider t_{pd} and t_{cd} given for each element



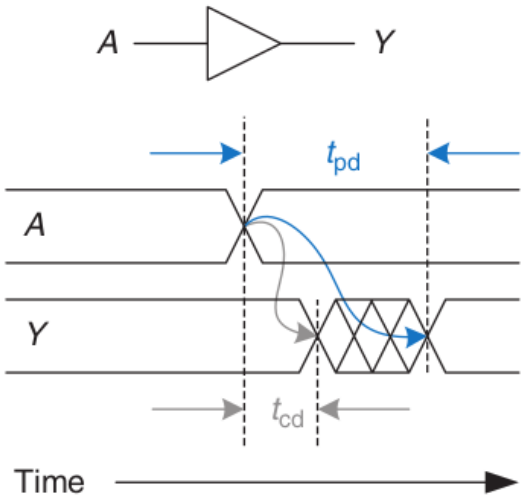
Timing Specification

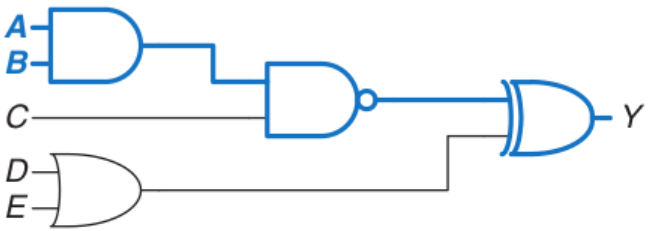
- Propagation and Contamination delays
- We consider t_{pd} and t_{cd} given for each element
- Different paths, different delays
- Critical path limits the speed



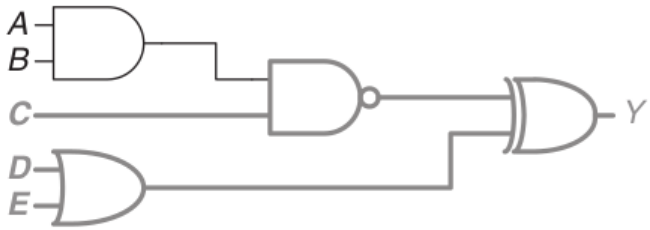
Timing Specification

- Propagation and Contamination delays
- We consider t_{pd} and t_{cd} given for each element
- Different paths, different delays
- Critical path limits the speed
- Assuming $t_{pd} = 100ps$ and $t_{cd} = 60ps$ for each gate:



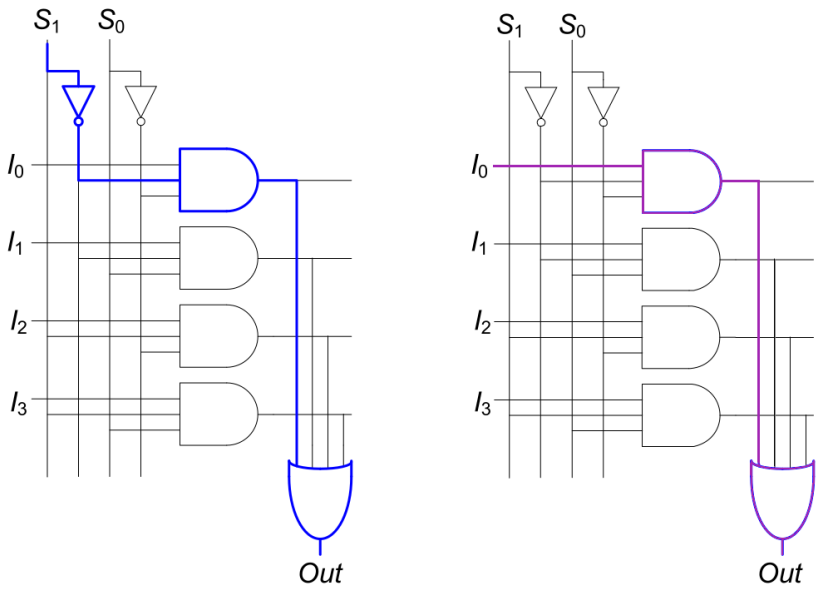


$t_{pd} = 3 \times (100) = 300ps$



$t_{cd} = 2 \times (60) = 120ps$

Example: MUX Timing Specification



Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

$t_{pd\_sy} = t_{pINV} + t_{pAND3} + t_{pOR4}$

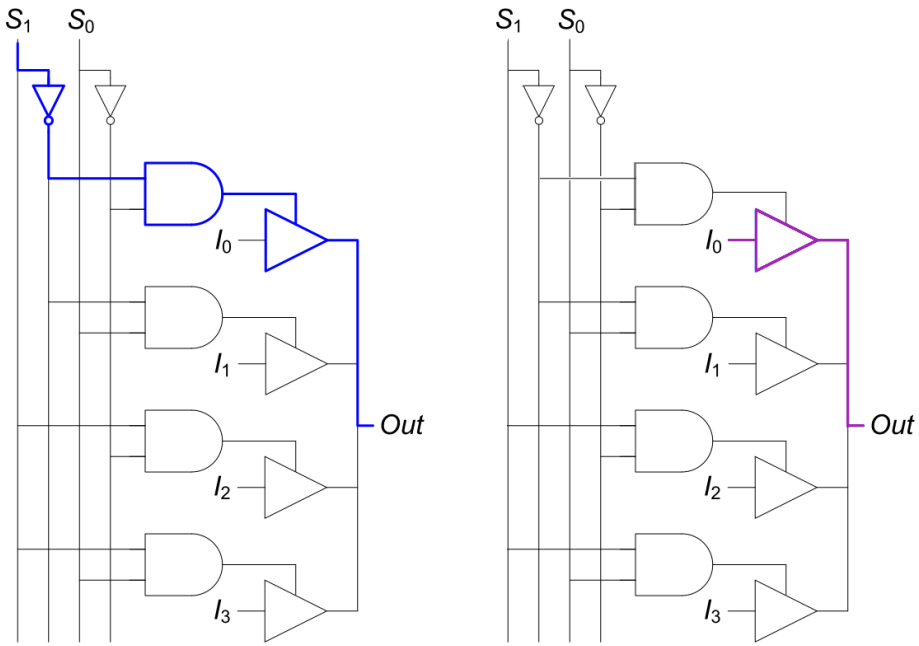
$= 30 \text{ ps} + 80 \text{ ps} + 90 \text{ ps}$

$= \mathbf{200 \text{ ps}}$

$t_{pd\_dy} = t_{pAND3} + t_{pOR4}$

$= \mathbf{170 \text{ ps}}$

Example: MUX Timing Specification

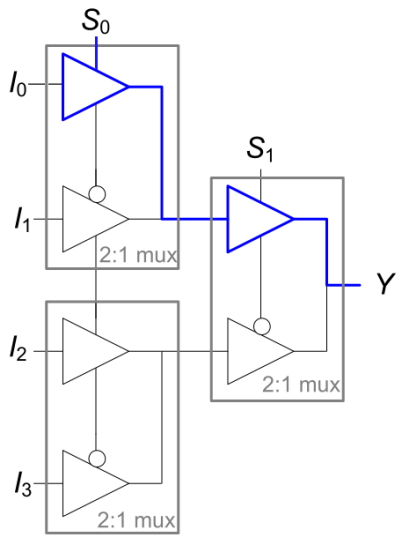


t_{pd_sy} = t_{pINV} + t_{pAND2} + t_{pTRI_SY}
= 30 ps + 60 ps + 35 ps
= **125 ps**

t_{pd_dy} = t_{pTRI_AY}
= **50 ps**

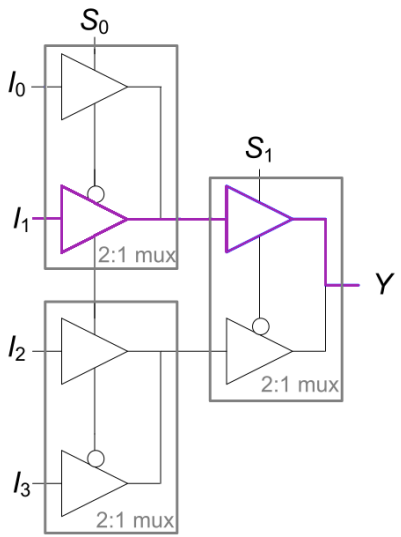
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

Example: MUX Timing Specification



$t_{pd\_s0y} = t_{pTRI\_SY} + t_{pTRI\_AY}$

= 85 ns

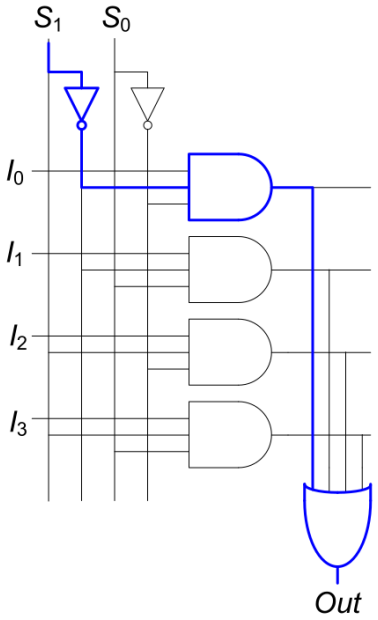


$t_{pd\_dy} = 2\ t_{pTRI\_AY}$

= 100 ns

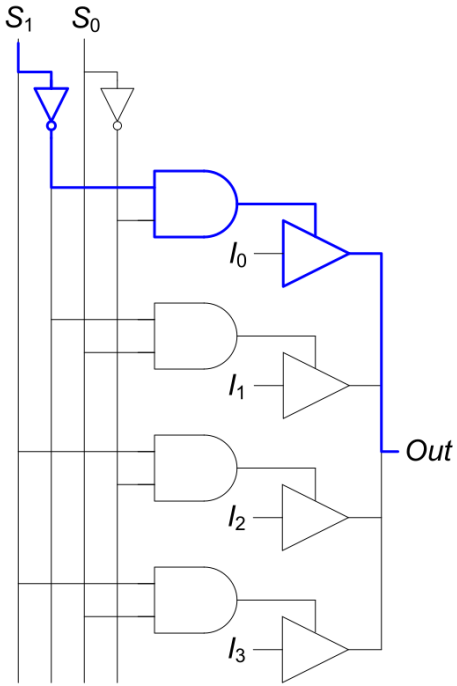
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

Example: MUX Timing Specification



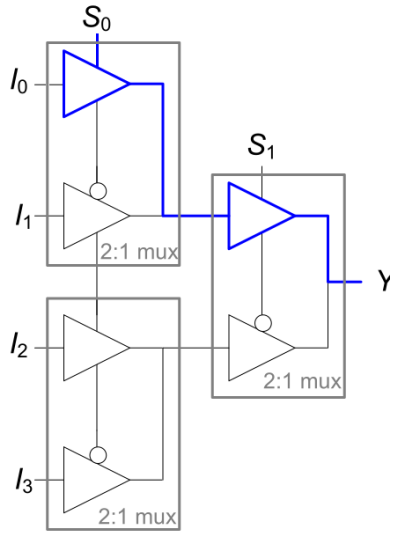
t_{pd_sy} = t_{pINV} + t_{pAND3} + t_{pOR4}
= 30 ps + 80 ps + 90 ps
= **200 ps**

t_{pd_dy} = t_{pAND3} + t_{pOR4}
= **170 ps**



t_{pd_sy} = t_{pINV} + t_{pAND2} + t_{pTRI_SY}
= 30 ps + 60 ps + 35 ps
= **125 ps**

t_{pd_dy} = t_{pTRI_AY}
= **50 ps**



t_{pd_s0y} = t_{pTRI_SY} + t_{pTRI_AY}
= **85 ns**

t_{pd_dy} = 2 t_{pTRI_AY}
= **100 ns**

Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35