Digital Logic Circuit (SE273 – Fall 2020)

Lecture 4: Combinational Logic Design

Jaesok Yu, Ph.D. (jaesok.yu@dgist.ac.kr)

Assistant Professor

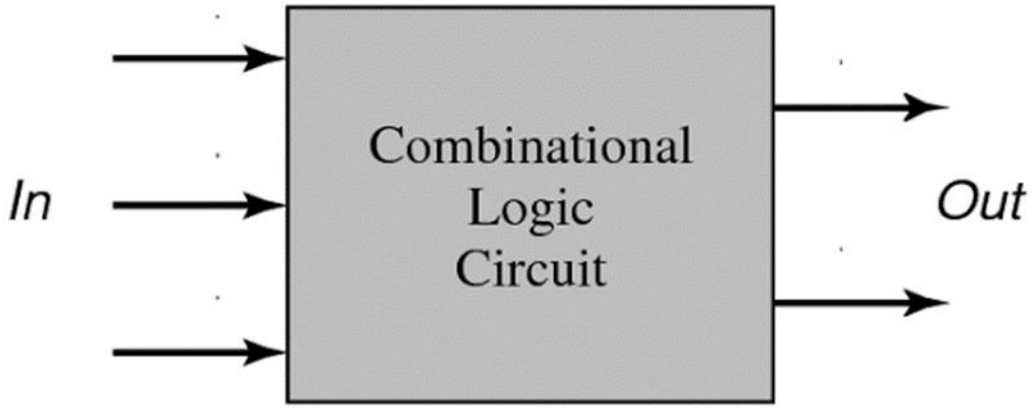
Department of Robotics Engineering, DGIST

Goal

- Learn to design combinational circuits
  - Specification, formulation, optimization
  - Technology mapping, verification
- Implement a number of important functions
  - The fundamental and reusable circuits (= functional blocks)
  - Implement functions of a single variable (decoders, encoders, multiplexers,…)

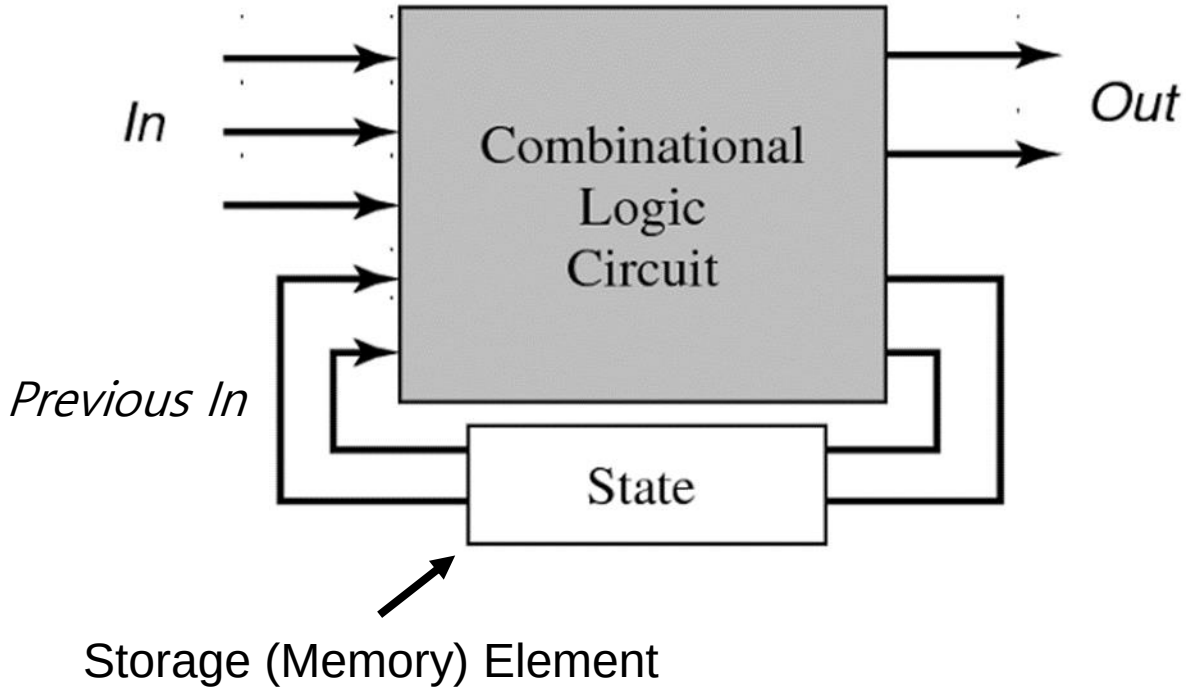
Combinational Logic vs Sequential Logic

Combinational Logic



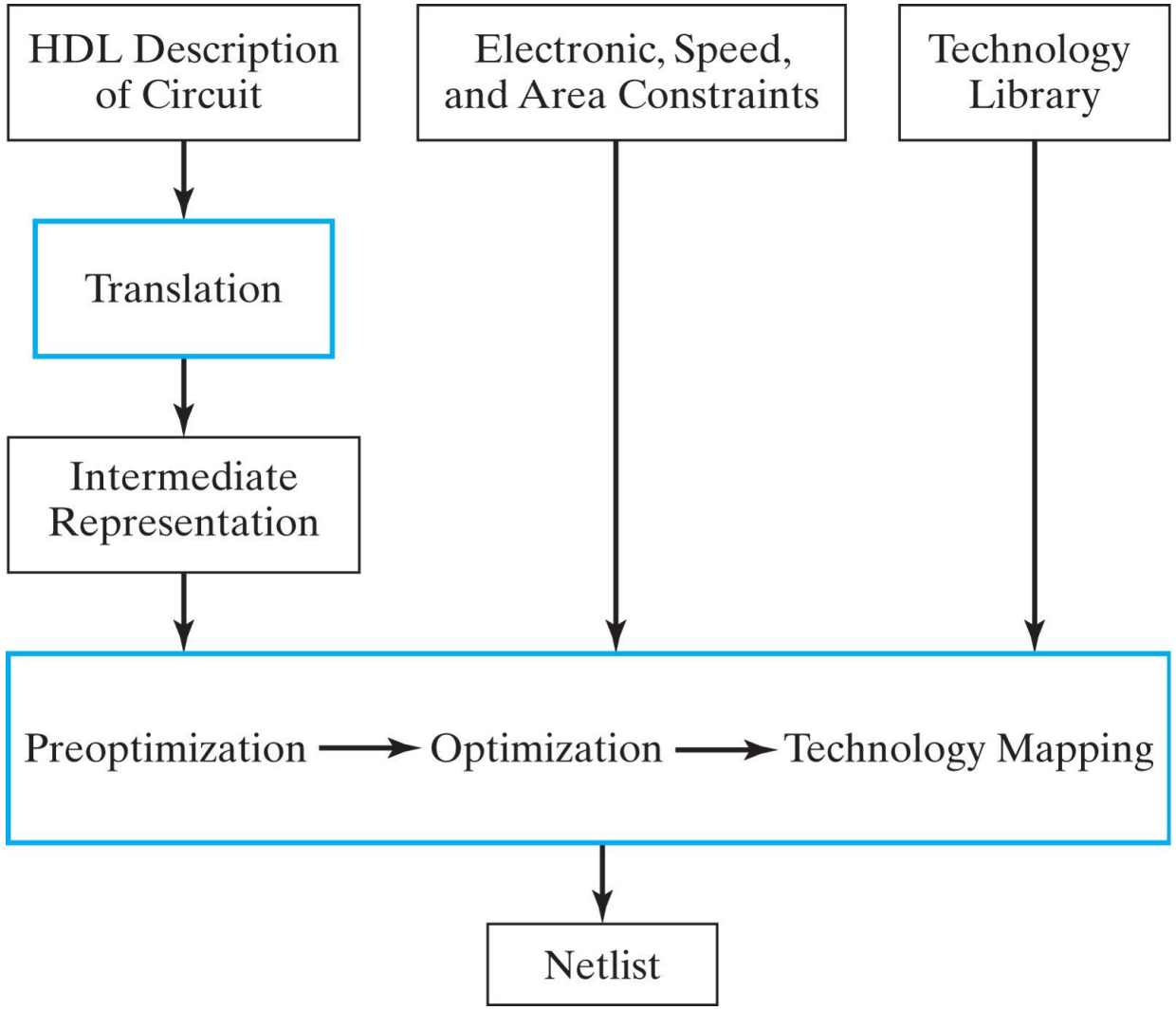
$$Out = f(In)$$

Sequential Logic



$$Out = f(In, Previous\ In)$$

Design Procedure



- **Specification**: write a specification for the circuit
- **Translation (Formulation)**: derive the truth table or initial Boolean equations
- **Optimization**: apply two-level and/or multi-level optimization
- **Technology Mapping**: transform the logic diagram to a new diagram using the available technology
- **Verification**: verify the correctness of the final design

Design Example: BCD-to-Excess3 Code Converter

- **Specification**
  - The excess-3 code for a decimal digit ‘d’ is the binary combination equal to ‘d+3’
  - The excess-3 code has desirable properties with respect to implementing decimal subtraction

Truth Table for Code-Converter Example

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Excess3 Code?

- **Self-complementary property**

Truth Table for Code-Converter Example

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

<taking 1's complement>

W X Y Z: **1 0 0 1 (6)**

<9's complement>

9 – 3
= 9 + 6 (9's complement of 3)
= 15 (1 is end-around carry, add again)
> 5+1 = 6

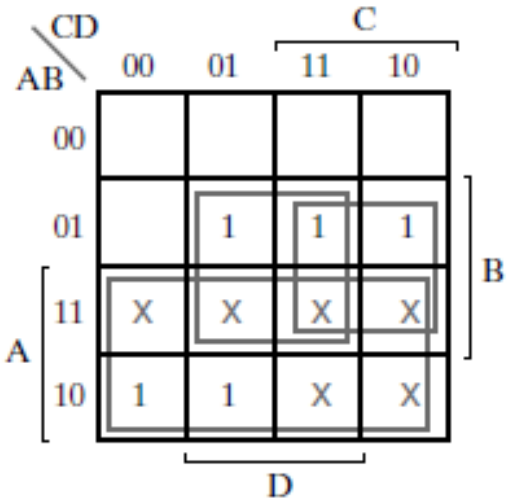
Design Example: BCD-to-Excess3 Code Converter

- **Formulation**
  - The excess-3 code is obtained from a BCD code by adding ‘0011’
  - We can build a truth table accordingly
  - The six combinations from 1010 through 1111 are not listed as they are never used in BCD code
  - We can treat them as **don’t-care conditions**

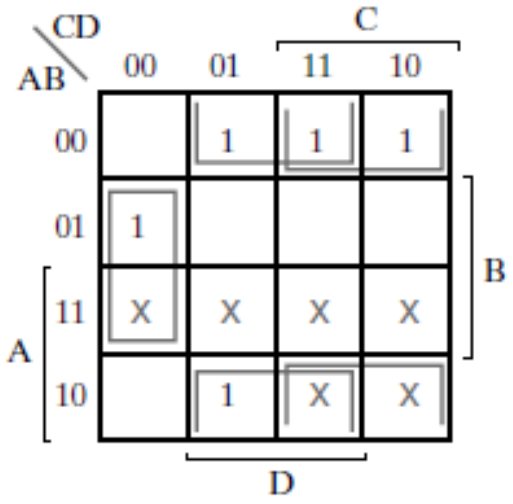
BCD(8421)				Excess-3			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Design Example: BCD-to-Excess3 Code Converter

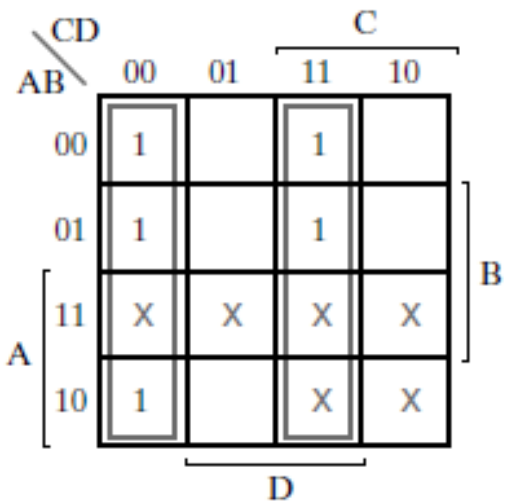
- **Optimization**
  - We may use the K-maps for the initial optimization of the four output functions
  - Don't-care conditions (X's) are useful to further reduce # of literals/terms
  - Each map represents one of the outputs of the circuit



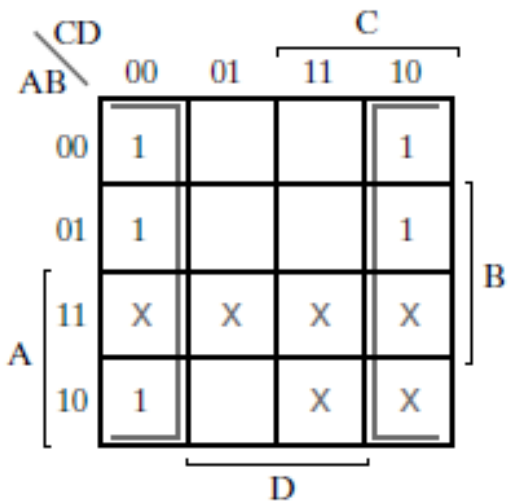
$W = A + BC + BD$



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



$Y = CD + \overline{CD}$



$Z = \overline{D}$

Design Example: BCD-to-Excess3 Code Converter

- Now, we can draw two-level AND-OR logic diagram
  - Directly from the Boolean expressions derived from the maps
  - We may apply multiple-level optimization as a 2nd optimization step (ex: reduce the gate input cost)
  - For the additional optimization, we consider sharing subexpressions

$$T_1 = C + D$$

$$W = A + BC + BD = A + BT_1$$

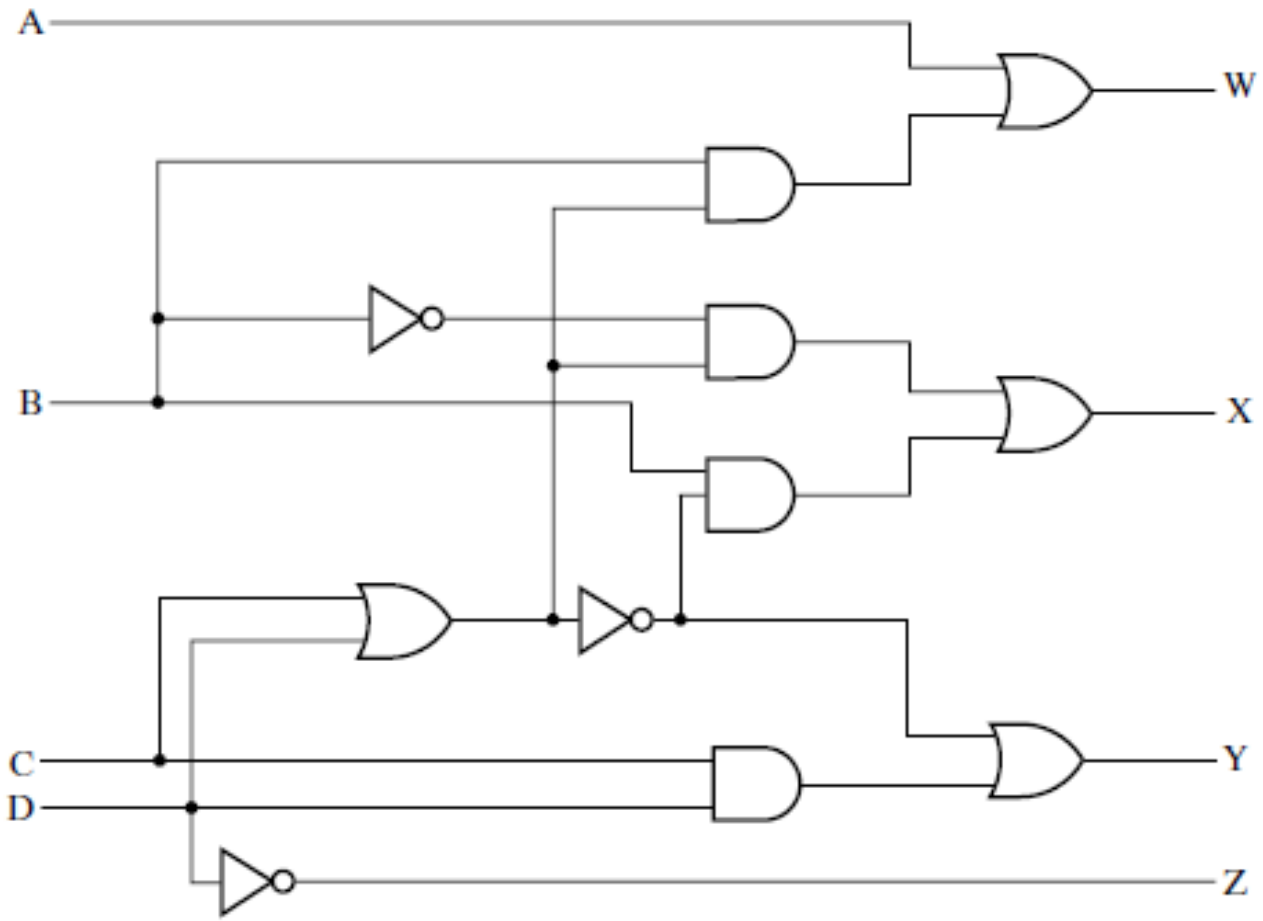
$$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D} = \overline{B}T_1 + B\overline{T}_1$$

$$Y = CD + \overline{T_1}$$

$$Z = \overline{D}$$

Design Example: BCD-to-Excess3 Code Converter

- Then, the logic diagram becomes



$$T_1 = C + D$$

$$W = A + BC + BD = A + BT_1$$

$$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D} = \overline{B}T_1 + B\overline{T}_1$$

$$Y = CD + \overline{T}_1$$

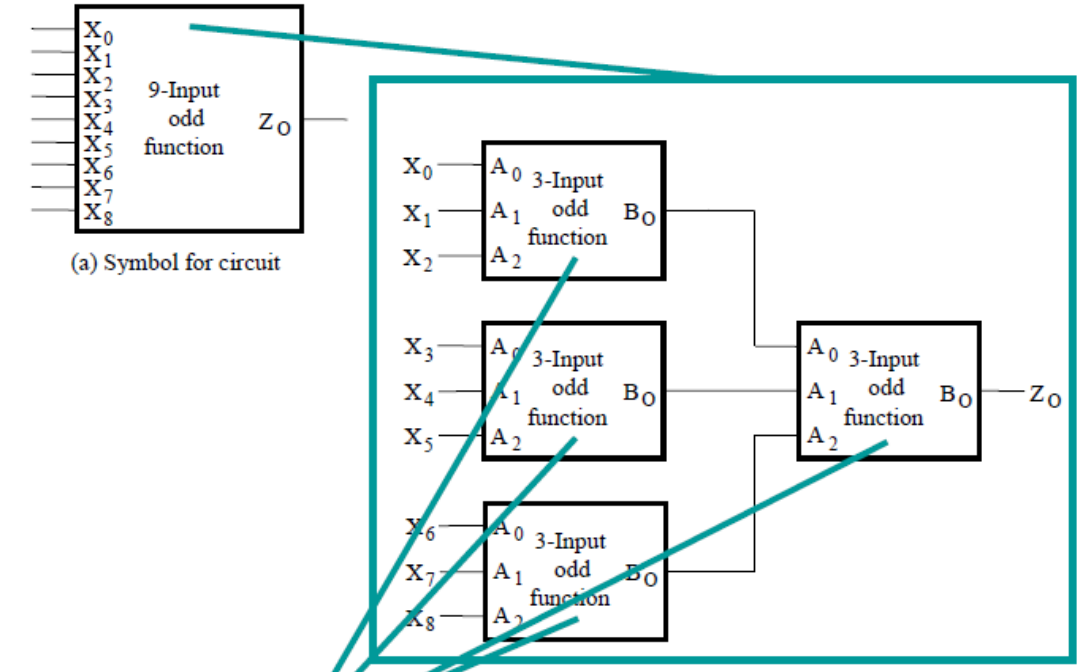
$$Z = \overline{D}$$

Circuit netlists

Hierarchical Design

- Hierarchical design approach is generally used in designing a digital system (divide-and-conquer)
  - We may deal with circuit complexity as we broken down into pieces (called functional blocks)
  - If a block is still to large, it can be broken into smaller blocks
  - This process can be repeated as necessary

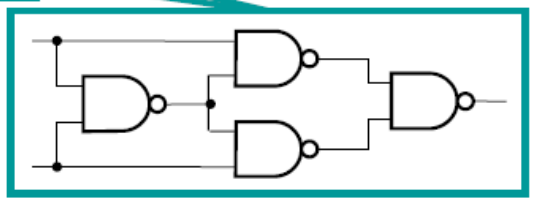
Hierarchical Design Procedure



(a) Symbol for circuit

(b) Circuit as interconnected 3-input odd function blocks

(c) 3-input odd function circuit as interconnected exclusive-OR blocks



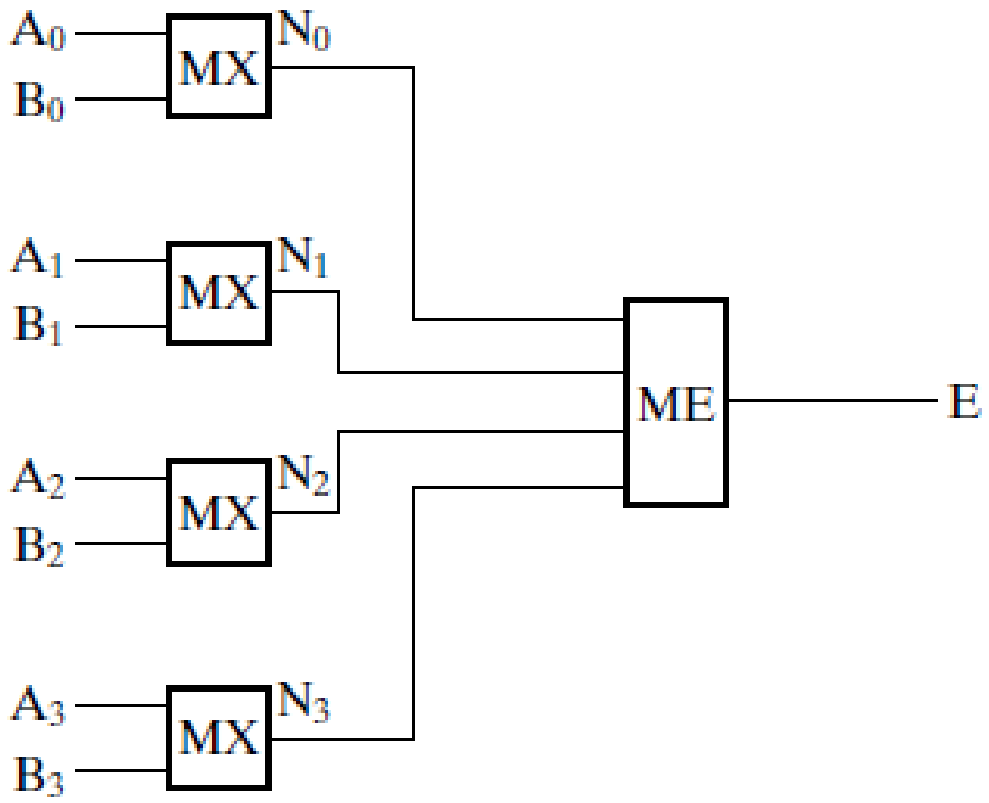
(d) Exclusive-OR block as interconnected NANDs

Hierarchical Design Example: 4-bit Equality Comparator

- **Specification**
  - An equality comparator is a circuit that compares two binary vectors to determine whether they are equal or not
  - Inputs: two vectors A[3:0] and B[3:0]
  - Output: a single-bit variable E (E=1 if vectors A and B are equal)

Hierarchical Design Example: 4-bit Equality Comparator

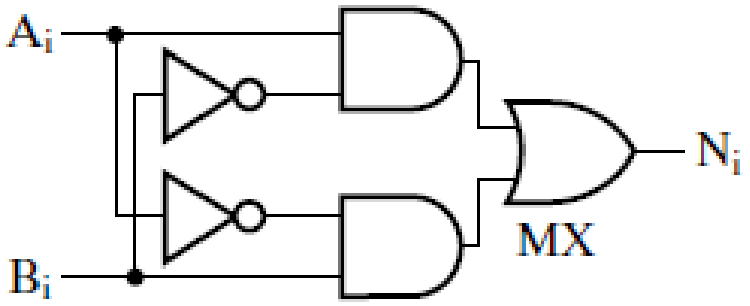
- **Formulation**
  - We bypass the use of a truth table due to its size!!
  - Step1: To compare A and B, the bit values in each of the respective positions, 3→0, of A and B must be equal
  - Step2: If all of bit positions of A and B contain equal values, then E = 1
  - Intuitively, we developed a simple 2-level hierarchy



Hierarchical Design Example: 4-bit Equality Comparator

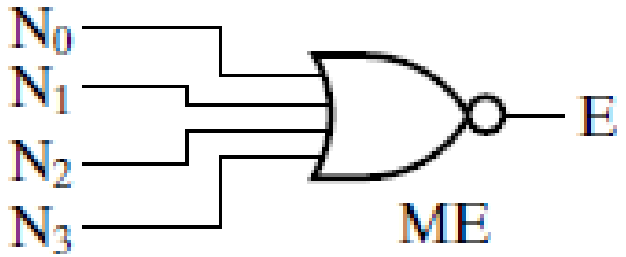
- **Optimization**
  - For bit position *i*, we define the circuit output N_i to be 0 if A_i and B_i have the same values
  - And $N_i = 1$ if A_i and B_i have different values
  - Then, MX circuit can be described by

$$N_i = \overline{A}_i B_i + A_i \overline{B}_i$$



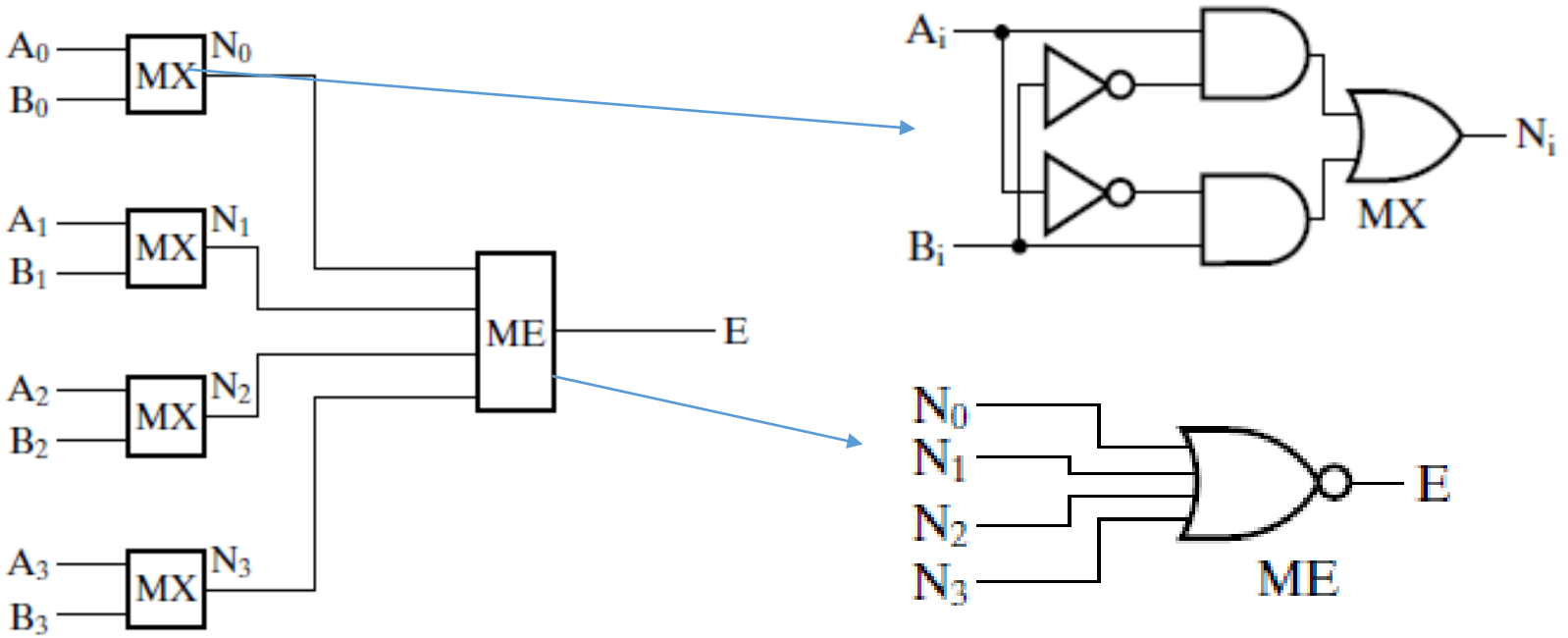
  - Now, we employ four copies of this circuit
  - Output E can be computed by the relation:

$$E = \overline{N_0 + N_1 + N_2 + N_3}$$



Hierarchical Design Example: 4-bit Equality Comparator

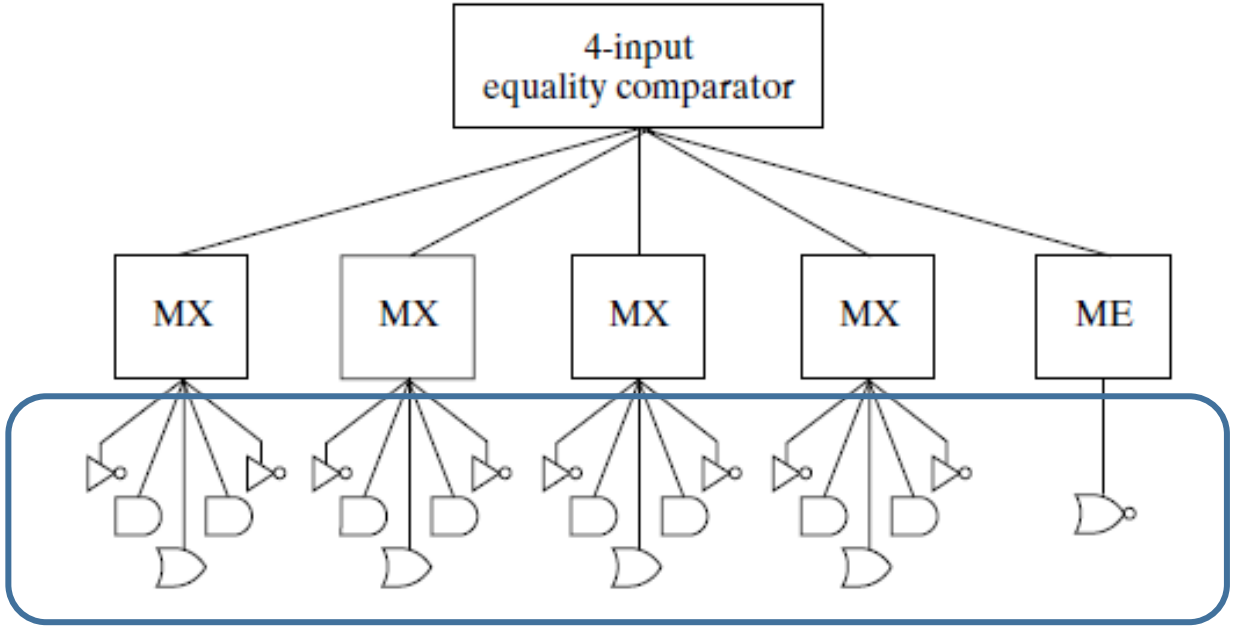
- Both circuits given are optimum two-level circuits
  - Two circuit diagrams plus the block diagram represent the hierarchical design of a 4-bit equality comparator



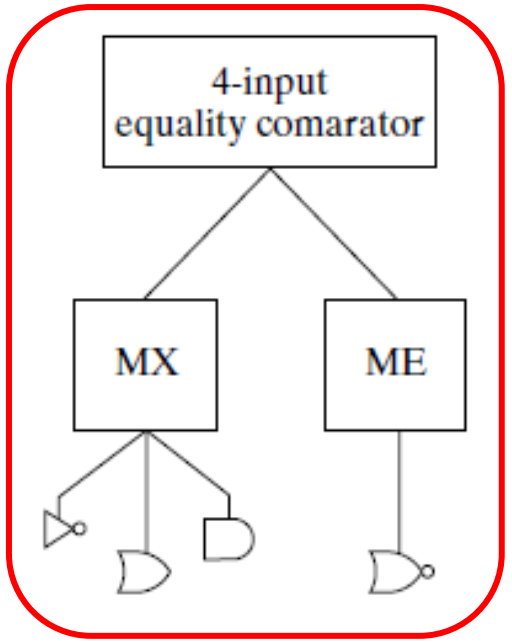
11 Symbols are required to represent the circuit

Hierarchical Design Example: 4-bit Equality Comparator

- We may represent the circuit without interconnections
  - By simply starting with the top block
  - Then, connect blocks or primitives from which the block is constructed
  - The hierarchy can be easily understood



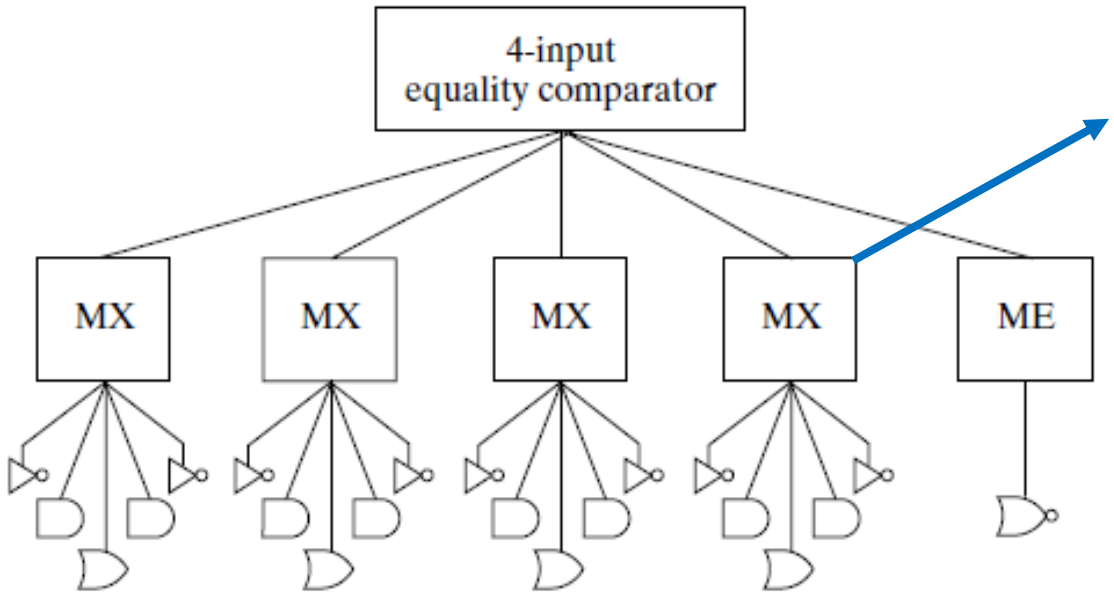
How many gates are needed?

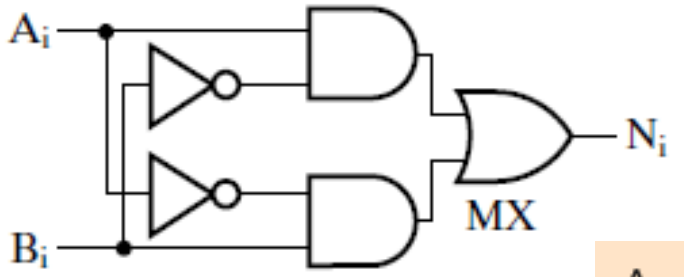


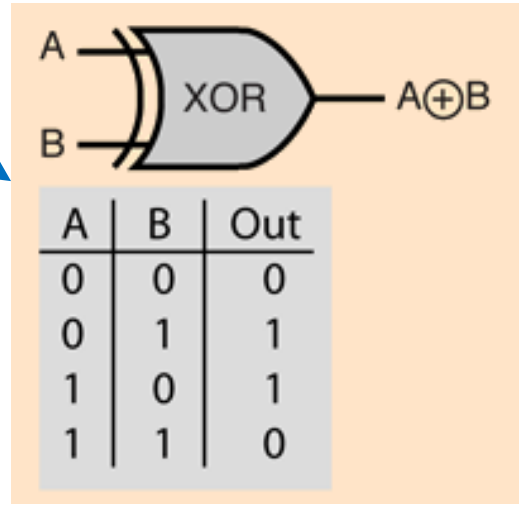
What is the purpose of this diagram?

Primitive Blocks

- At leaves of the hierarchy tree, there are logic gates
  - These gates are commonly called primitive blocks
  - There can be more complex blocks, other than simple gates, as predefined blocks (MX blocks can be considered as predefined XOR gates)



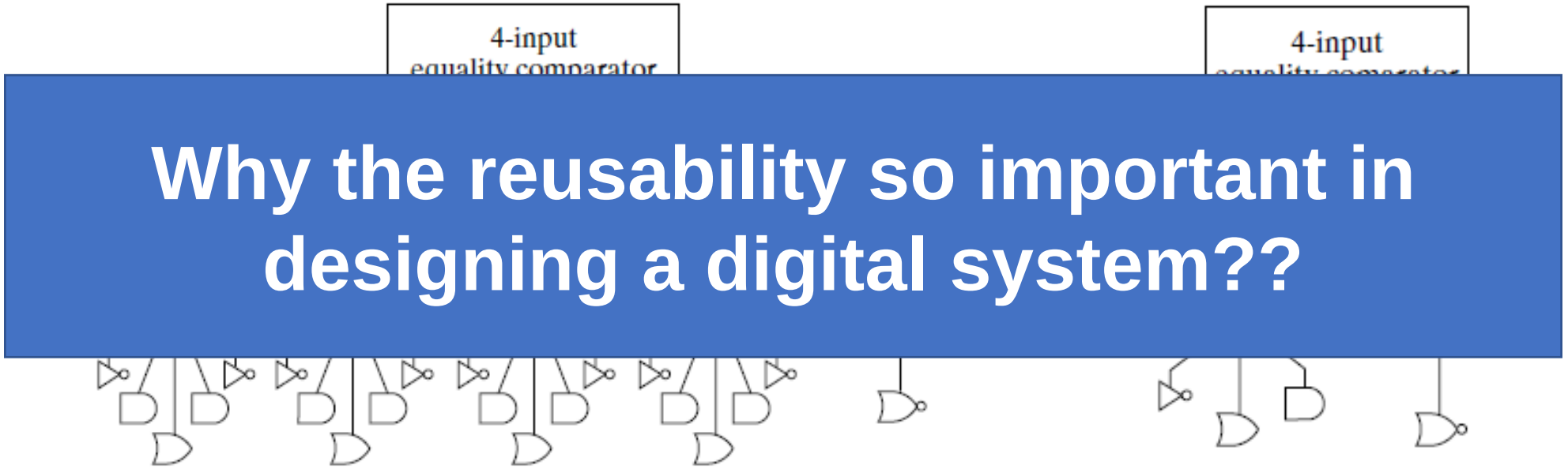




A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

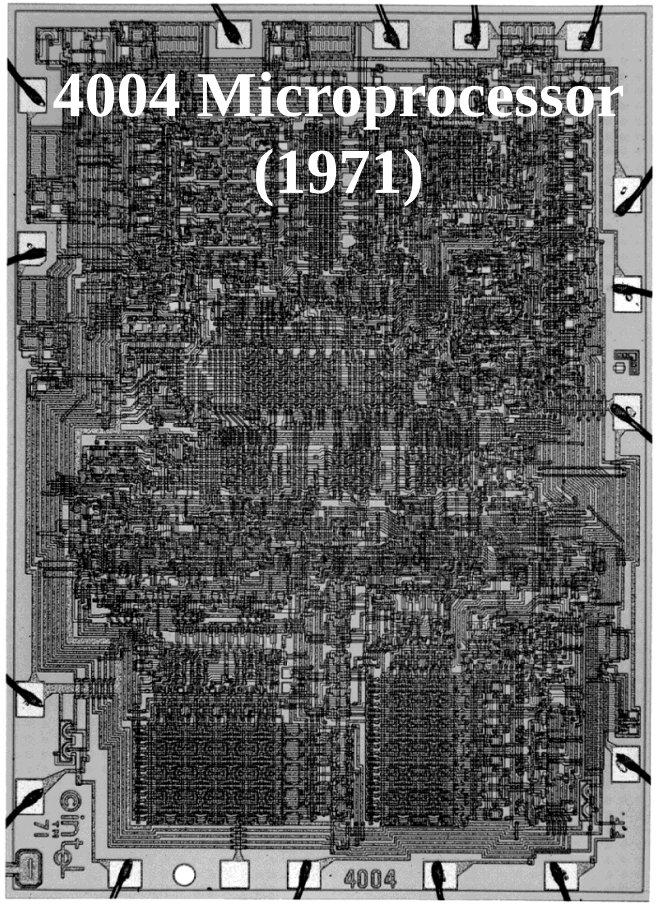
Reusability of Blocks

- Reusability is another important property of hierarchical design
  - There are four copies of 2-input MX blocks
  - On the right, there is only one copy of the 2-input MX block
  - That means a circuit designer need to implement one MX block and use this design four times

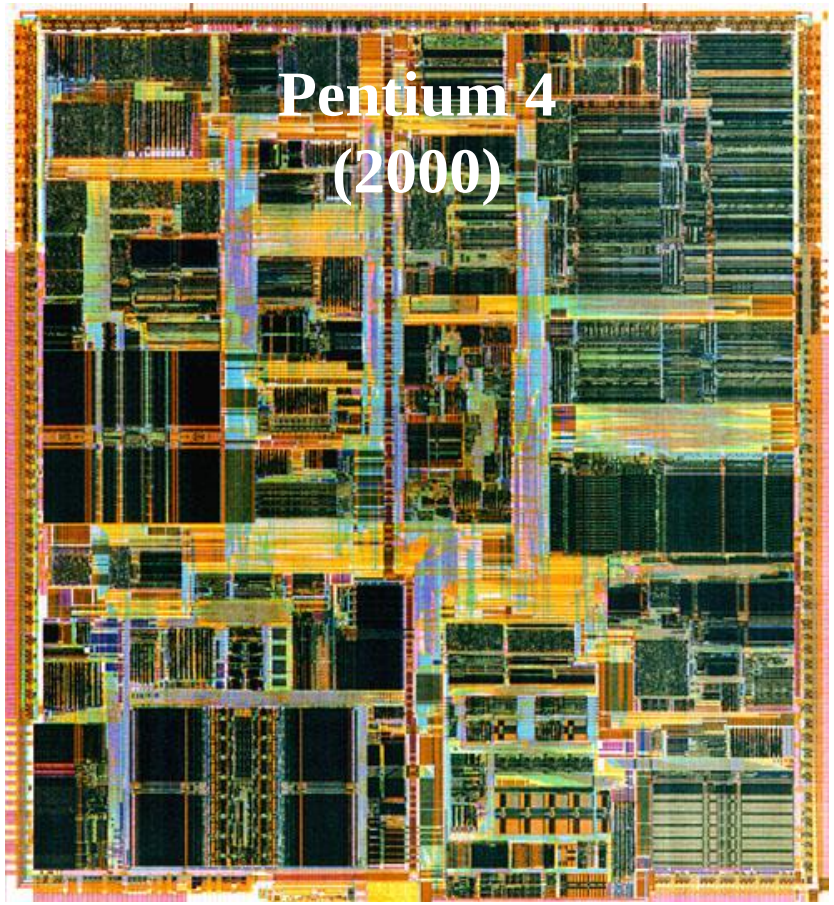


Why the reusability so important in designing a digital system??

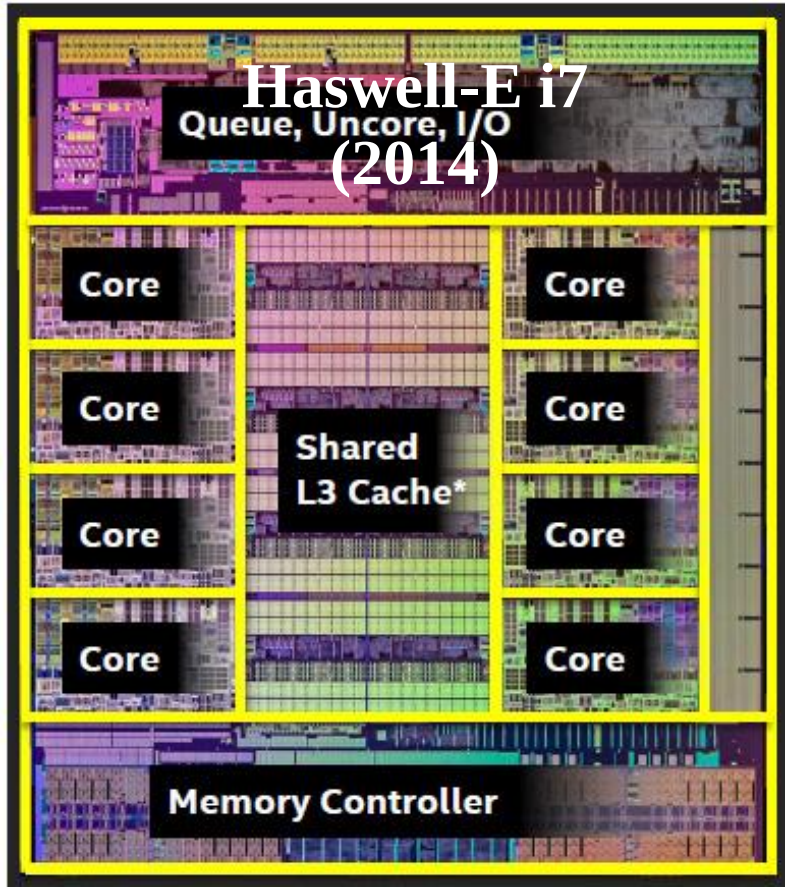
Technology – Transistors (Intel Processors)



2,250 transistors in 12mm^2
10,000 nm, 1MHz operation

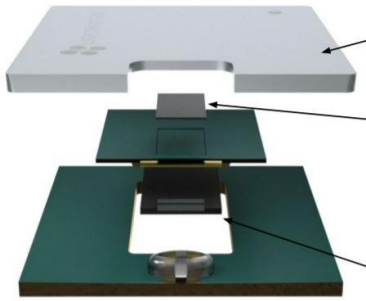


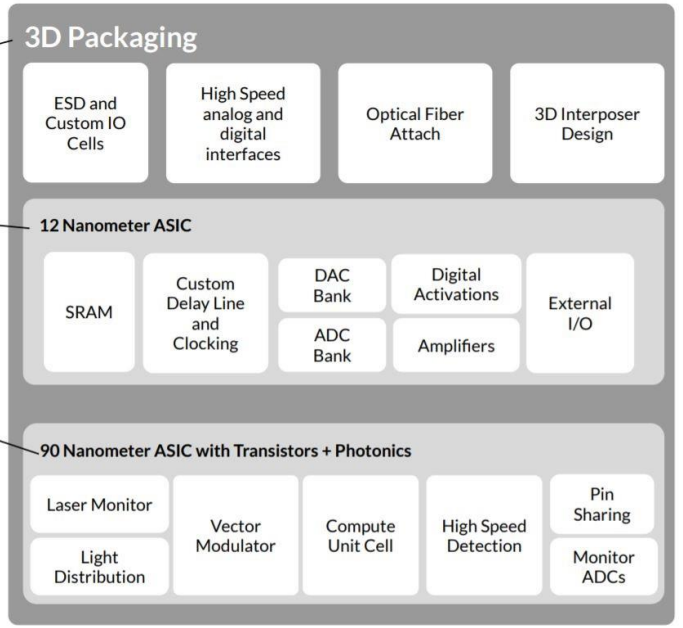
42 million transistors in 217mm^2
180 nm, 1.3GHz operation

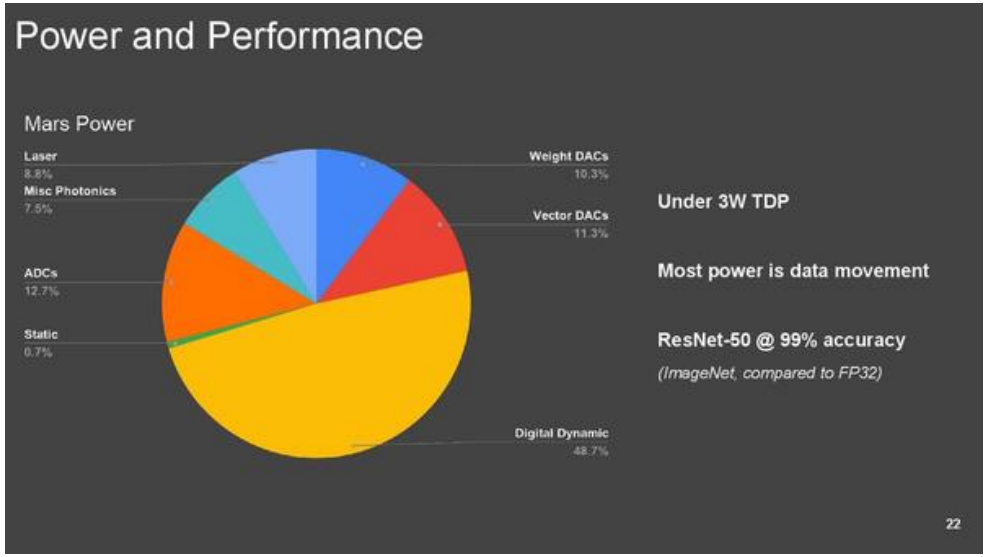


2.6 billion transistors in 355mm^2
22 nm, 3GHz operation

Future Technology: Light? Quantum?







Technology Mapping

- Mapping AND, OR, NOT gates **to NAND / NOR gate cells** is critical in designing a digital circuit (why??)
  - A NAND technology consists of a collection of cells, each of which includes a NAND gate with a fixed # of inputs
  - Assume we have four cell types, based on # of inputs (n = 1, 2, 3, 4)
  - INV, 2NAND, 3NAND, and 4NAND, respectively

Technology Mapping

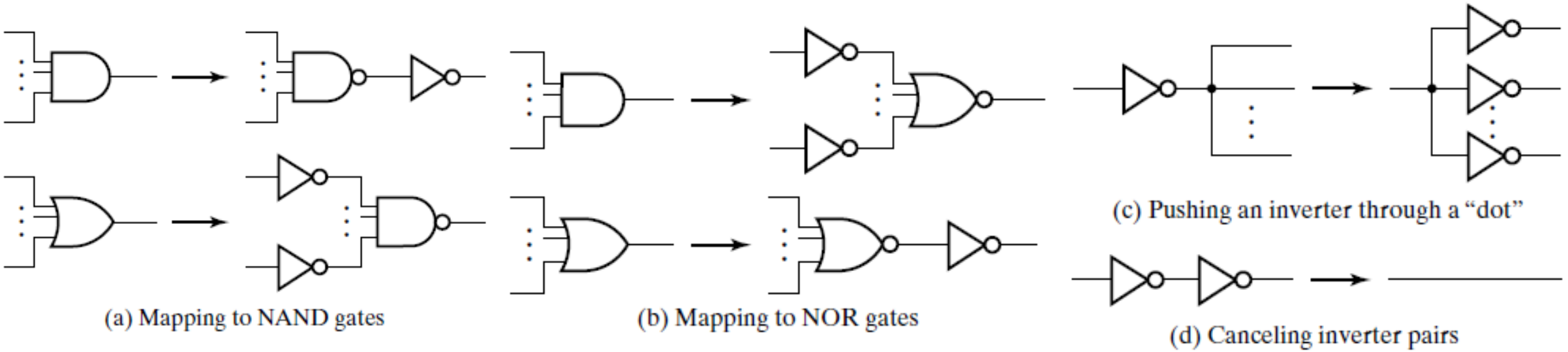
- We begin with the optimized logic diagram of the circuit consisting of AND and OR gates (+ inverters)
  - Then, the function is converted to NAND logic by converting the logic diagram to NAND gates and INVs
  - Same applies for NOR gate cells

Technology Mapping

- Procedure for technology mapping
  - Objective: obtain a circuit diagram with the fewest INVs

1) Replace each AND and OR gate with NAND (NOR) gate and inverter equivalent circuits: (a) and (b)

2) Cancel all inverter pairs



(a) Mapping to NAND gates

(b) Mapping to NOR gates

(c) Pushing an inverter through a "dot"

(d) Canceling inverter pairs

Implementation with NAND Gates

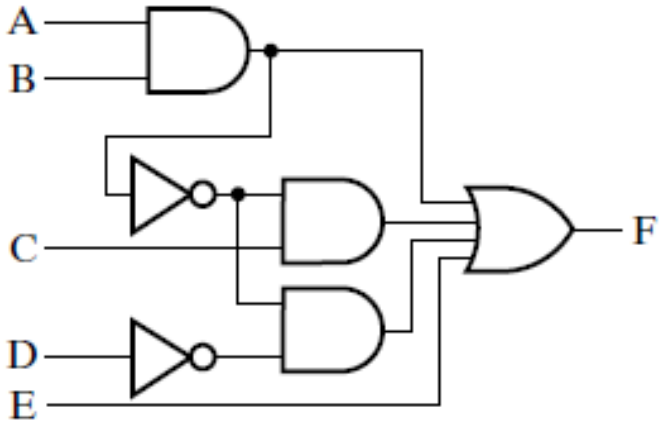
- Implement the following optimized function with NAND gates:

$$F = AB + \overline{(AB)}C + (\overline{AB})\overline{D} + E$$

Implementation with NAND Gates

- Implement the following optimized function with NAND gates:

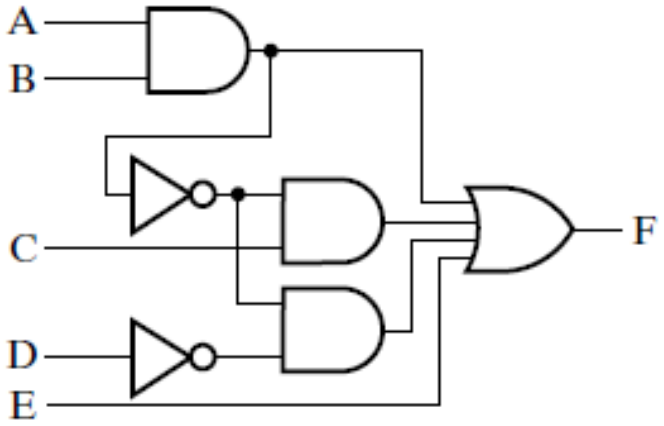
$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$

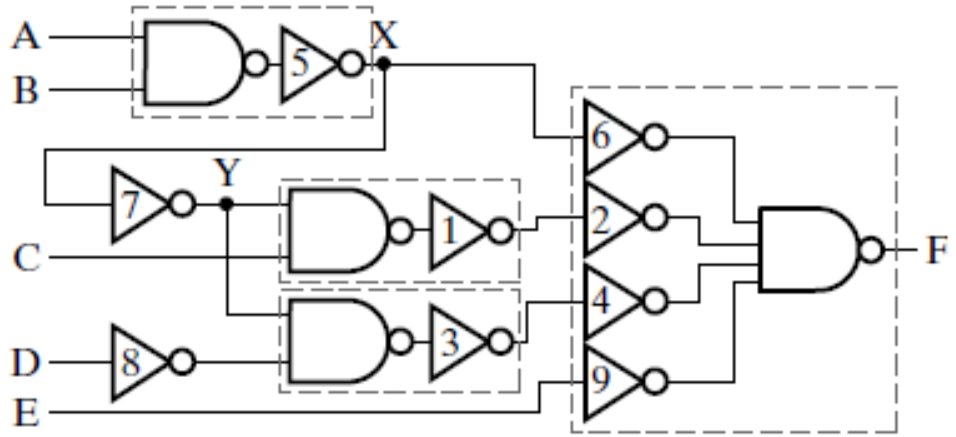


Implementation with NAND Gates

- Implement the following optimized function with NAND gates:

$$F = AB + \overline{(AB)}C + (\overline{AB})\overline{D} + E$$





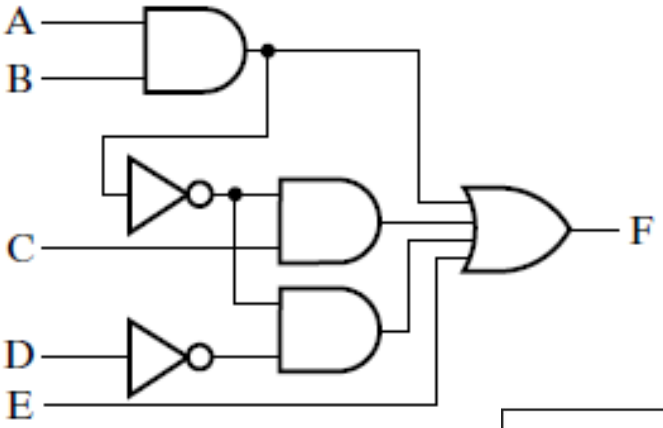
Step 1 is applied!!

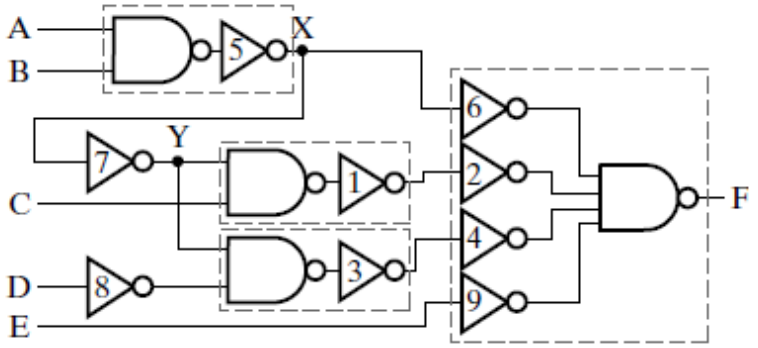
1) Replace each AND and OR gate with NAND (NOR) gate and inverter equivalent circuits

Implementation with NAND Gates

- Implement the following optimized function with NAND gates:

$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$

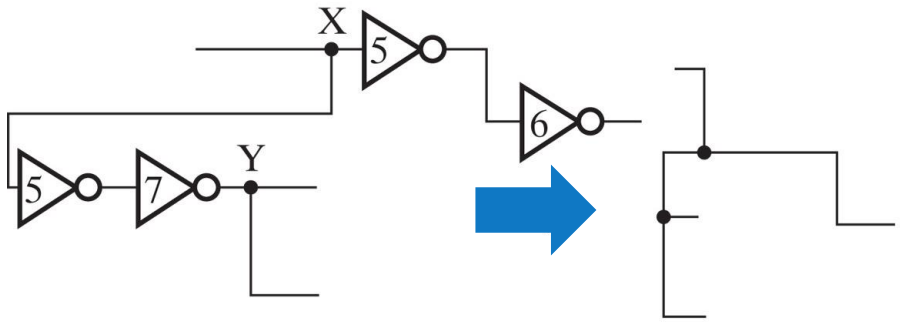




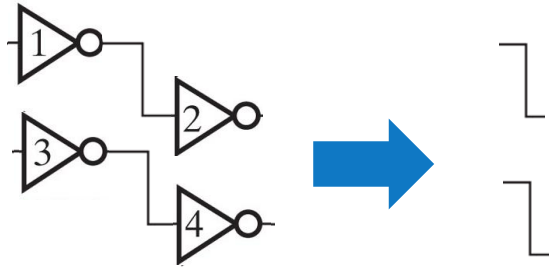
Step 2 is applied!!

2) Cancel all Inverter pairs

Node 1



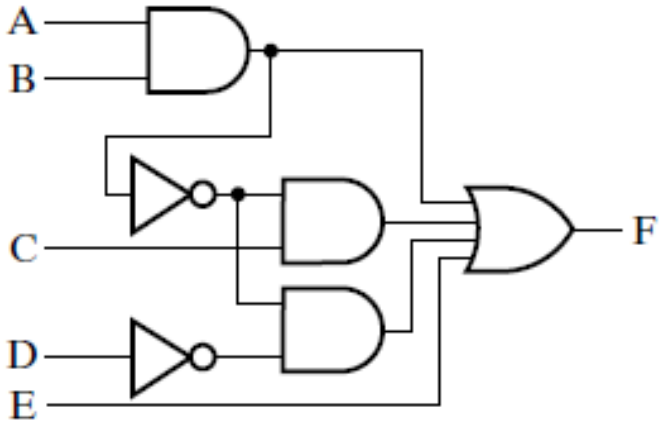
Node 2,3

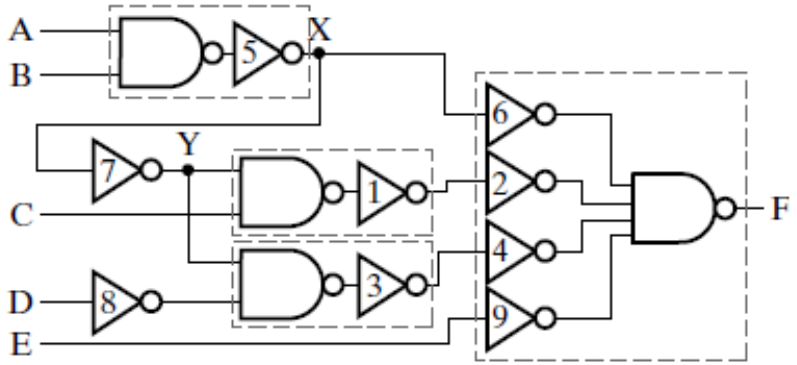


Implementation with NAND Gates

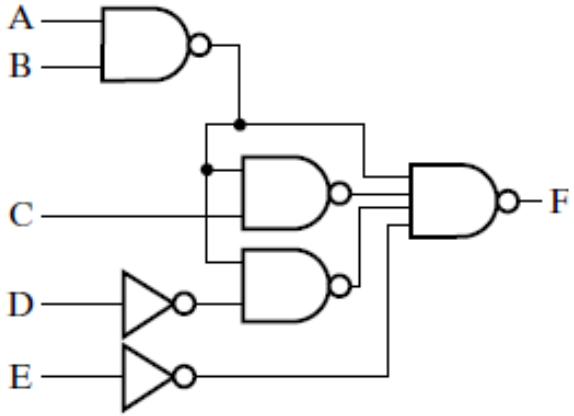
- Implement the following optimized function with NAND gates:

$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$





Step 1

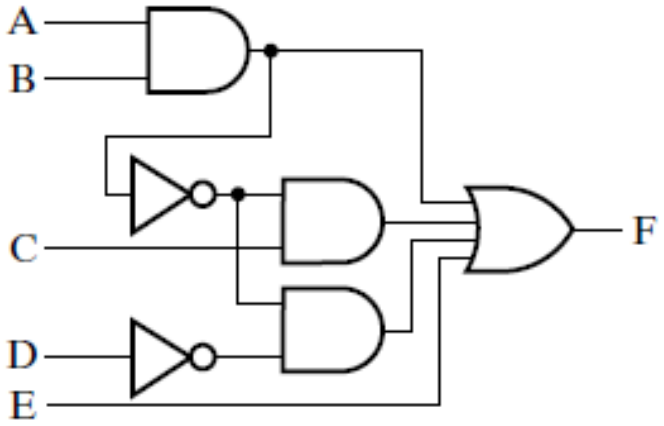


Step 2

Implementation with NOR Gates

- Implement the following optimized function with NOR gates:

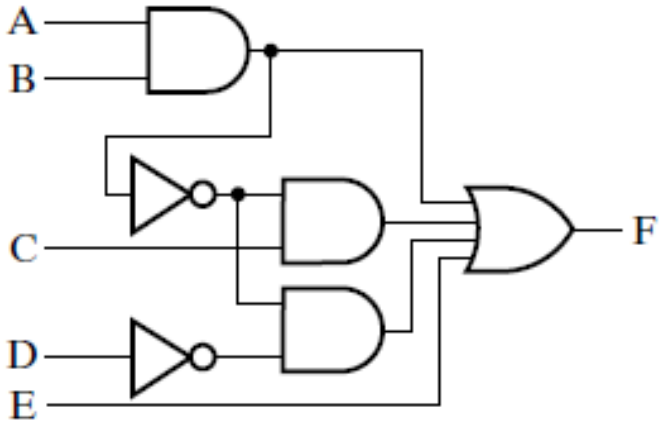
$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$

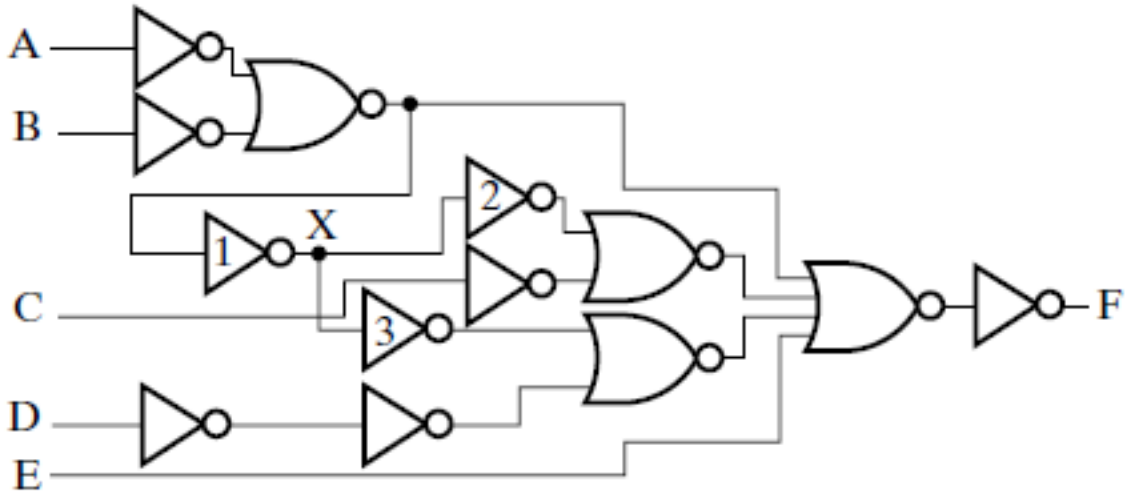


Implementation with NOR Gates

- Implement the following optimized function with NOR gates:

$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$



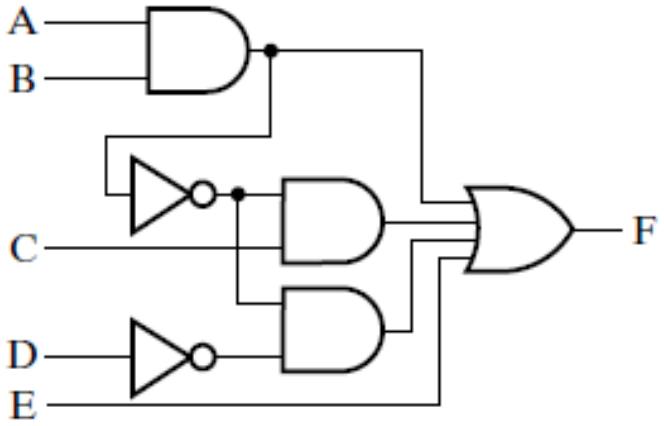


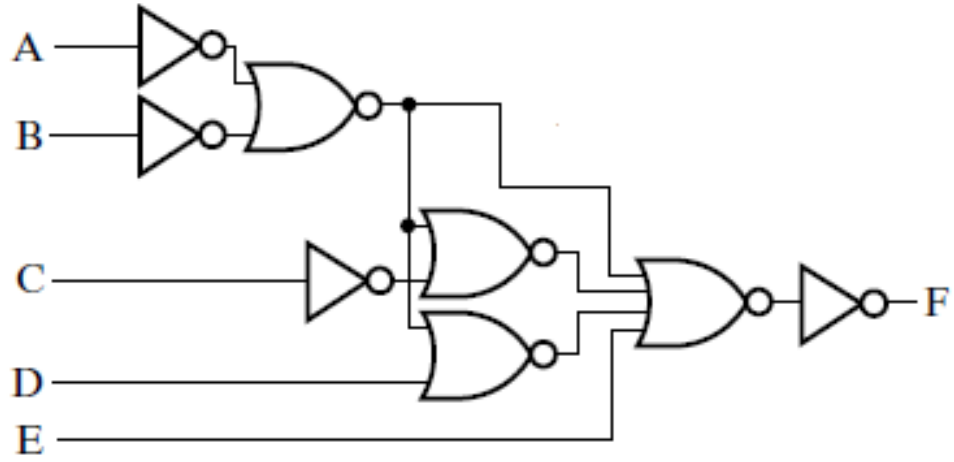
Step 1 is applied!!
(Replace to NOR gate)

Implementation with NOR Gates

- Implement the following optimized function with NOR gates:

$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$



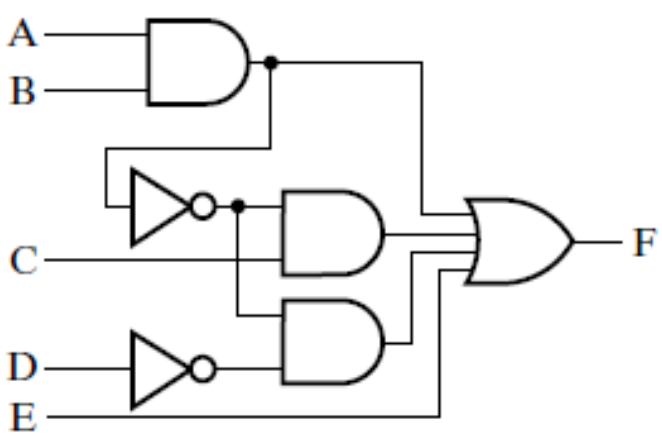


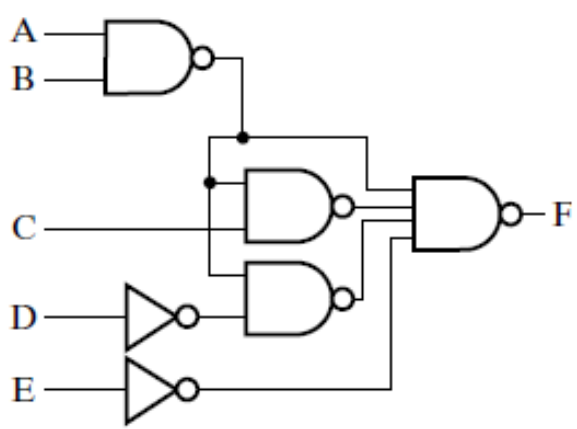
Step 2 is applied!!
(Cancel all inverters)

Comparison Btw NAND and NOR Implementations

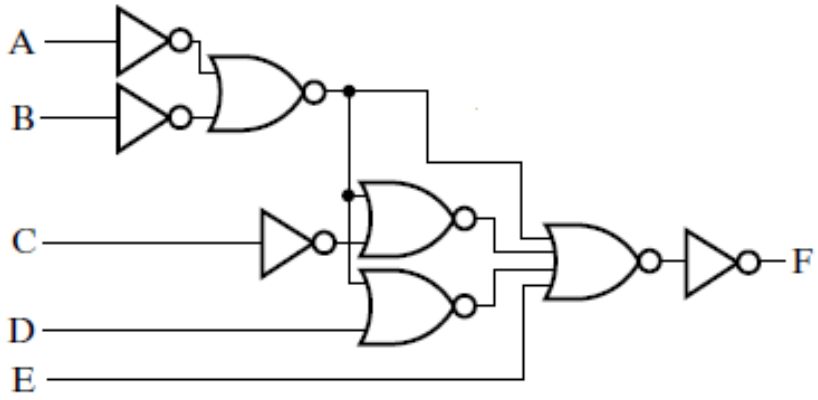
- Which implementation seems to be better and why?

$$F = AB + \overline{(AB)}C + \overline{(AB)}\overline{D} + E$$





NAND



NOR

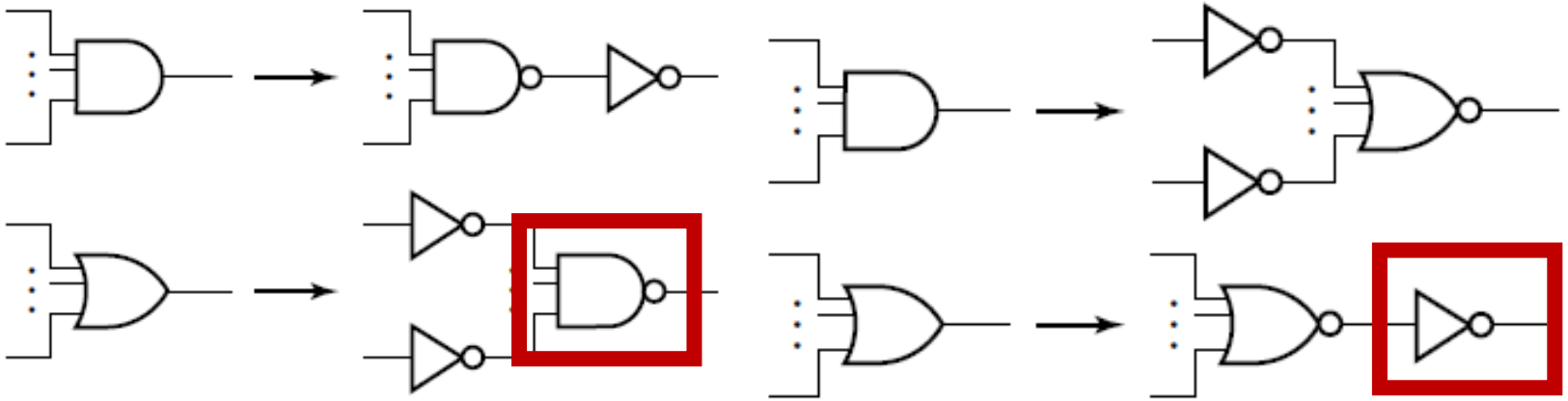
Comparison Btw NAND and NOR Implementations

- Which implementation seems to be better and why?
  - Gate-input cost:
  - Depth of logic gates in series:

- The result of a technology mapping is clearly influenced by the initial circuit or equation forms prior to mapping
  - Mapping to NANDs for a circuit with an OR gate at the output produces a NAND gate at the output (natural)
  - For NOR counterpart, it produces an inverter driven by a NOR gate at the output

Comparison Btw NAND and NOR Implementations

- Which imp
  - Gate-inp
  - Depth o



(a) Mapping to NAND gates (b) Mapping to NOR gates

- The result of a technology mapping is clearly influenced by the initial circuit or equation forms prior to mapping
  - Mapping to NANDs for a circuit with an OR gate at the output produces a NAND gate at the output (natural)
  - For NOR counterpart, it produces an inverter driven by a NOR gate at the output

Verification

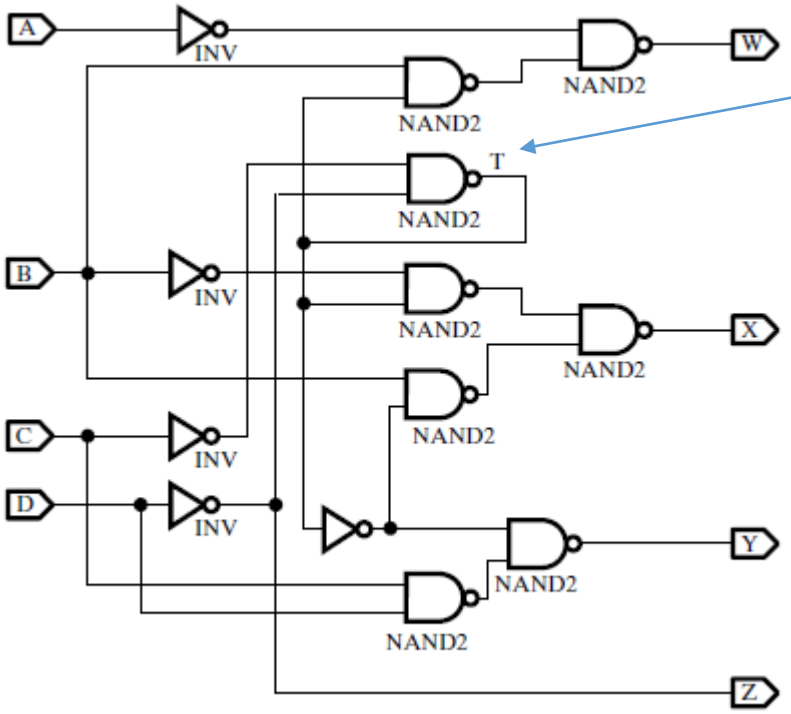
- Determine whether or not a given circuit implements its specified function
  - Manual logic analysis
  - Computer simulation-based logic analysis
- If the circuit does not meet its specification, then it is incorrect
- Verification plays a vital role in preventing incorrect circuit designs from being manufactured

Manual Logic Analysis

- It consists of finding Boolean equations for the circuit outputs, then finding the truth table for the circuit
  - It is often convenient to break up the circuit into subcircuit **by defining intermediate variables** at selected points
  - ***Fan-out points***: at which a gate output drives two or more gate inputs

Manual Verification of BCD-to-Excess3 Code Converter

- Objective is to complete the truth table from the circuit implementation
  - The intermediate point T is selected to simplify the analysis



NAND-mapped circuit implementation

$$T = \overline{\overline{C}\,\overline{D}} = C + D$$

$$W = \overline{\overline{A}\,\overline{(T \cdot B)}} = A + BT$$

$$X = \overline{\overline{(\overline{B}T)}\,\overline{(B\overline{C}\,\overline{D})}} = \overline{B} \cdot T + B\overline{T}$$

$$Y = CD + \overline{T}$$

$$Z = \overline{D}$$

$$W = A + B(C + D) = A + BC + BD$$

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

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

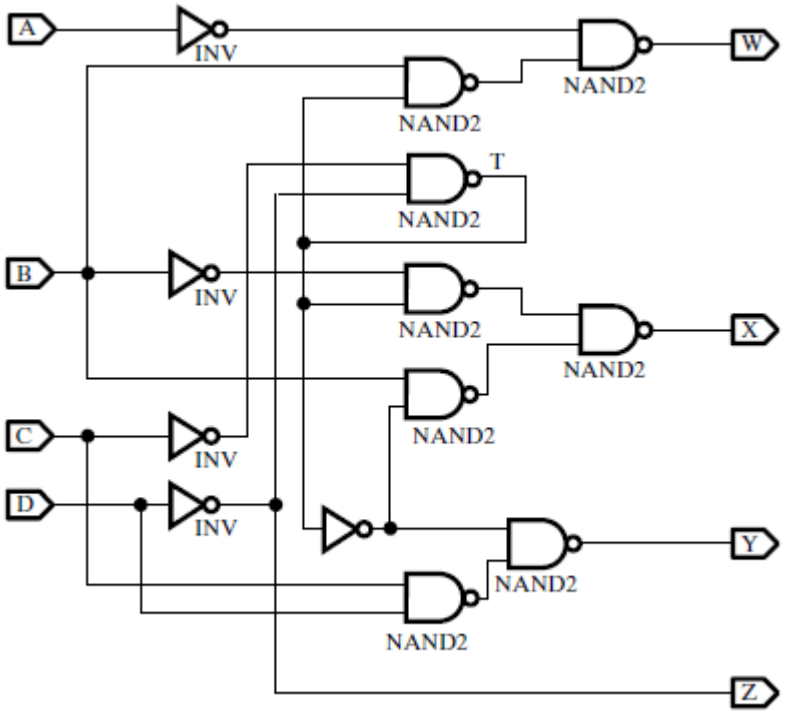
Fill the remaining product terms!

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

Truth table to be completed

Manual Verification of BCD-to-Excess3 Code Converter

- Objective is to complete the truth table from the circuit implementation
  - The intermediate point T is selected to simplify the analysis



NAND-mapped circuit implementation

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

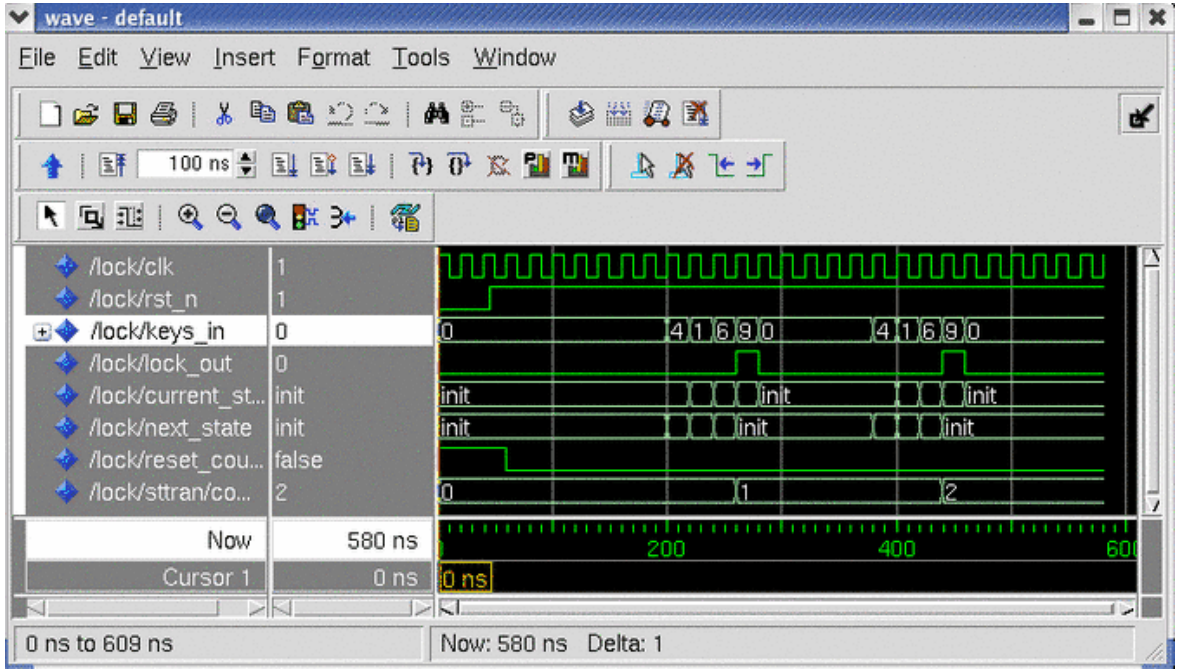
Truth table to be completed

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

Original truth table

Simulation-Based Verification

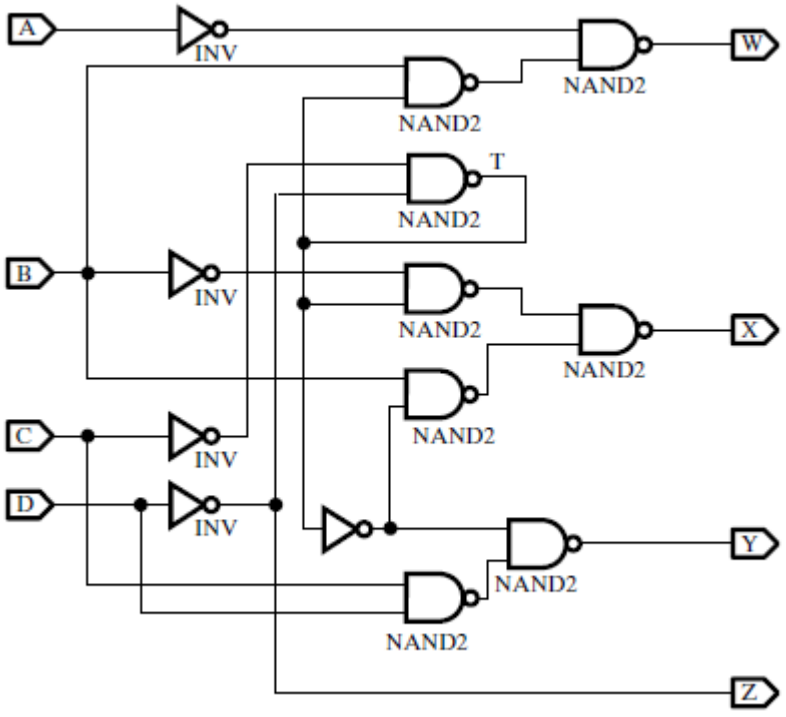
- You can use computer simulation for verification
  - A computer permits verification with significantly larger number of variables
  - Greatly reduces the tedious analysis effort
- Simulation is performed on provided input values
  - You need to apply (almost) all possible input combinations



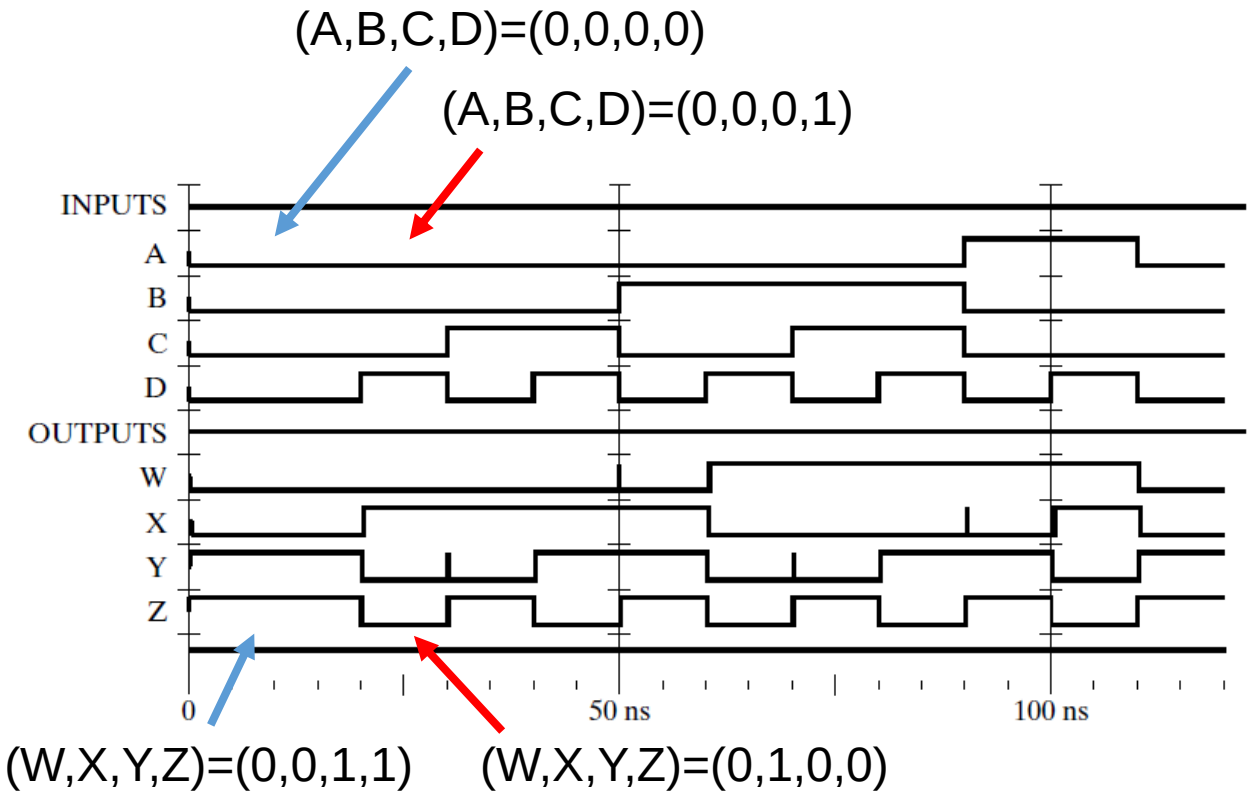
ModelSim®

Simulation-Based Verification

- In addition to entering the circuit schematic, input combinations are given as a waveform



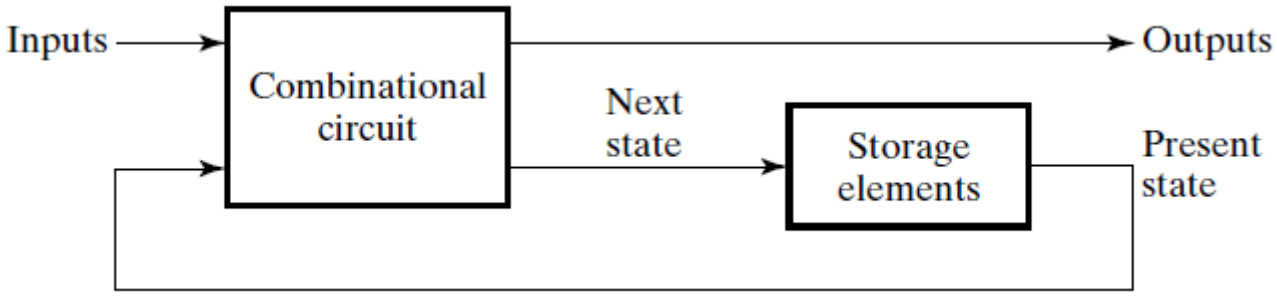
NAND-mapped circuit implementation



Simulation results of computer-based verification

Combinational Functional Blocks

- **Functional blocks**: a pre-defined combinational circuit that is specific to a certain combinational function
- Normally, a large-scale integrated circuits are almost always sequential
  - It is a combination of a combinational circuit and storage elements
  - To ease the design process, some frequently-used blocks are pre-designed



Block diagram of a sequential circuit

Rudimentary Logic Functions

- What are the most basic functions?
  - Value-fixing, transferring, inverting, and enabling

Functions of One Variable

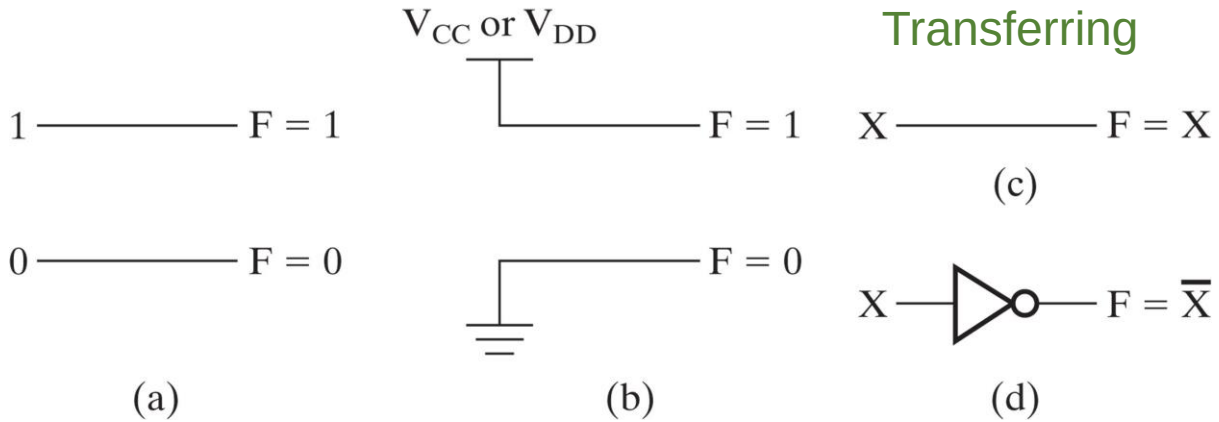
X	F = 0	F = X	F = $\overline{X}$	F = 1
0	0	0	1	1
1	0	1	0	1

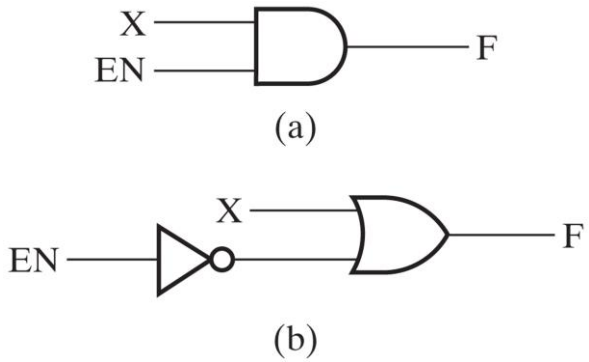
Value-fixing

Transferring

Inverting

Value-fixing

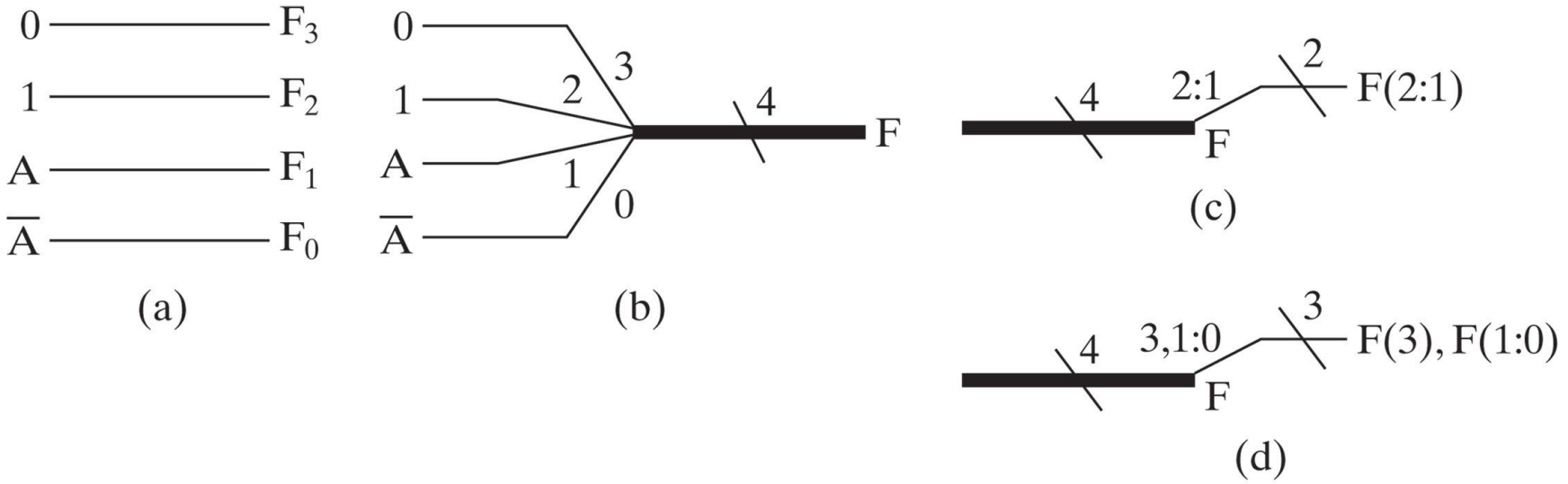




Enabling

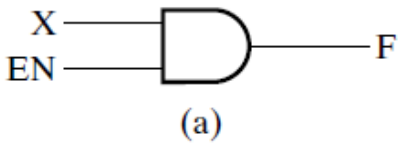
Multi-Bit Functions

- The functions defined so far can be applied to multiple bits on a bitwise basis
  - We can think of multiple-bit functions as vectors of single-bit functions
  - With F = (0,1,A,$\overline{A}$), it becomes (0,1,0,1) with A = 0
  - For multiple-bit wires, we use a single line with greater thickness with a slash, representing the bit-width

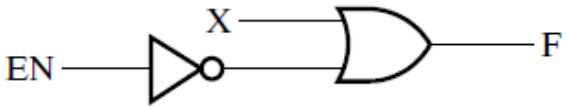


Enabling

- The enabling permits an input signal to pass through to an output
  - With EN=1, input X reaches the output
  - With EN=0, output is fixed at 0
- When do we need enabling signal?
  - One example is a three-state buffer

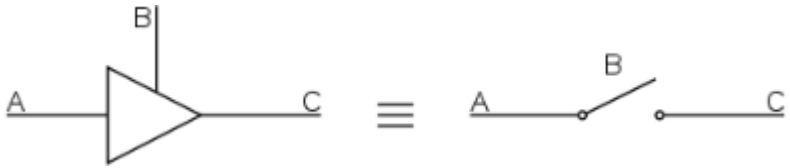


(a)



(b)

INPUT		OUTPUT
A	B	C
0	1	0
1		1
X	0	Z (high impedance)

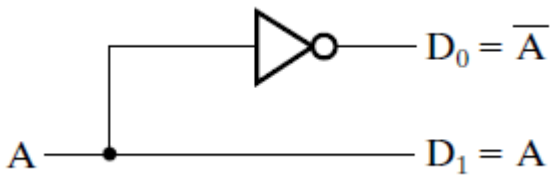


Tri-state buffer

Decoding

- In digital computers, discrete quantities of information are repre-sented by binary codes
  - An n-bit binary code is capable of representing 2^n distinct elements
  - **Decoding**: conversion of an input code to an m-bit code with $n \leq m \leq 2^n$
  - Decoding is performed by a circuit called *decoder*
- We design functional blocks, called n-to-m line decoders, which are the most important building blocks in a digital system

A	D_0	D_1
0	1	0
1	0	1

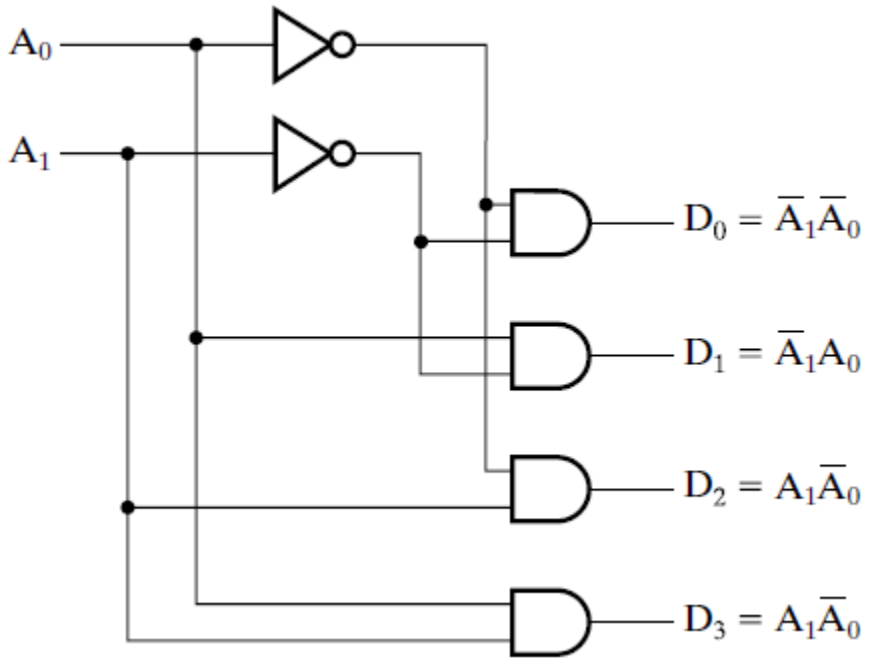


1-to-2 line decoder

Decoder Design Example

- A 2-to-4 line decoder
  - Output D_i is equal to 1 whenever the two input values on A_1 and A_0 are the binary code for the number '*i*'
  - There are four possible minterms in total

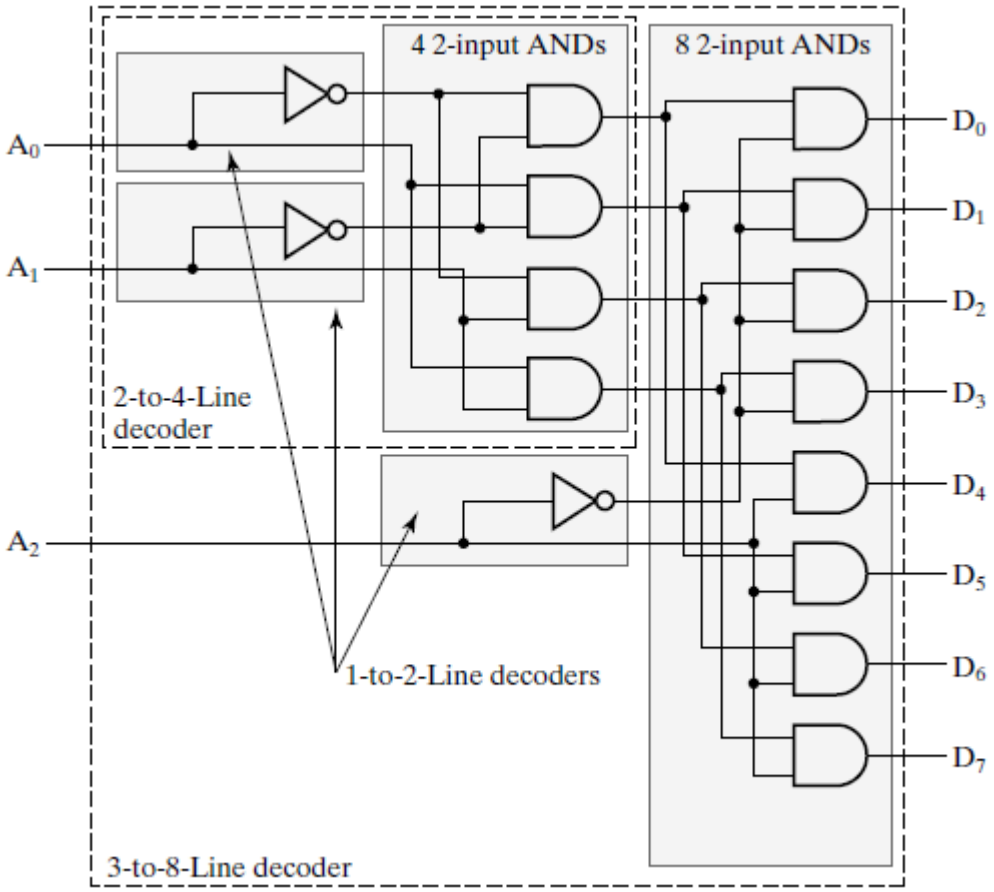
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Larger Decoders

- Larger decoders require a high gate-input cost (why?)
  - We use a hierarchical design approach
  - The resulting decoder will have the same or a lower gate-input cost than the one constructed by simply enlarging each AND gate

Can you design a 6-to-64 line decoder?

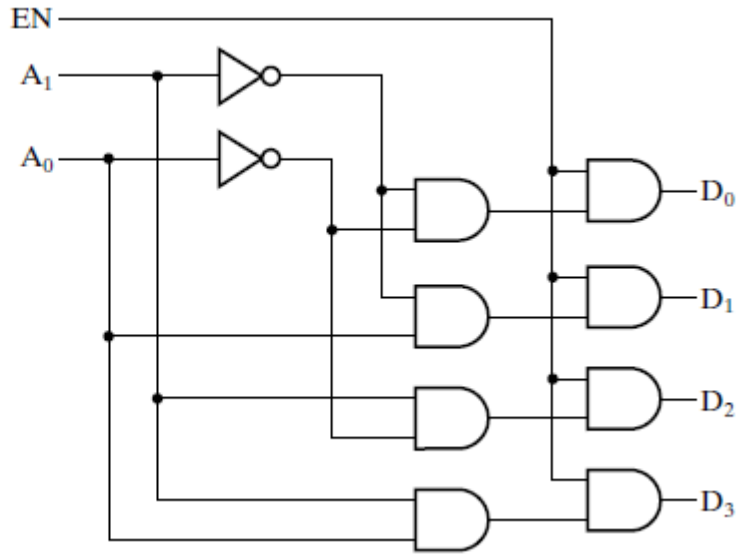


Decoding and Enabling Combinations

- The n-to-m line decoding with enabling can be implemented by attaching ‘m’ enabling circuits to the outputs

With EN=0, it resets all D_i signals

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



What about for larger decoders?

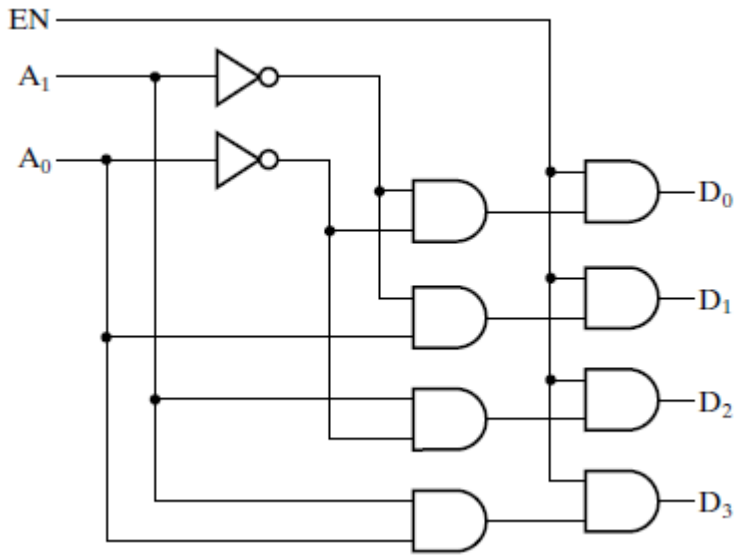
Demultiplexer (DEMUX)

- How can we transmit the information from a single line to one of 2^n possible output lines?
  - Such distribution is done by a circuit, called demultiplexer
  - An 1-to-4 line demultiplexer can be implemented by a 2-to-4 line decoder

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

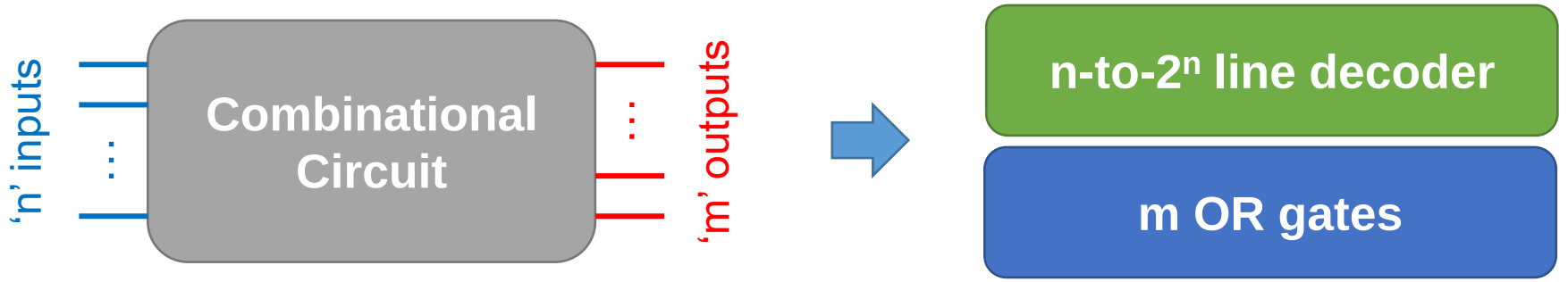
Data

Selection variable



Decoder-Based Combinational Circuits

- Any Boolean function can be expressed as a sum of minterms
  - One can use a decoder to generate the minterms and combine them with OR gate to form a sum-of-minterms implementation
  - A Boolean function can be expressed as a sum-of-minterms from the truth table or K-map



1-bit Binary Adder with Decoder and OR-Gate Implementation

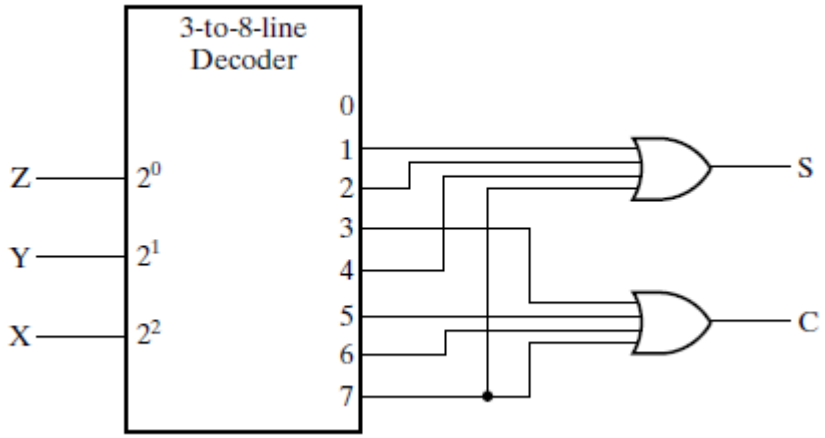
- Let's consider binary addition
  - The sum bit S and the carry bit C can be derived from three input bits
  - X and Y: two bits being added
  - Z: incoming carry

Truth Table for 1-Bit Binary Adder

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

$$S(X,Y,Z) = \Sigma m(1,2,4,7)$$

$$C(X,Y,Z) = \Sigma m(3,5,6,7)$$



Minterm 0 is not used

What happens with a long list of minterms?

Encoding

- Inverse operation of a decoder
  - An encoder has '2^n' (or fewer) input lines and 'n' output lines
  - The output generates a binary code for the input value
  - Example: octal-to-binary encoder

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The remaining 56 rows of binary combinations are considered don't cares

Octal-to-Binary Encoder

- A_i = 1 for the columns in which D_j = 1 only if subscript '*j*' has a binary representation with a 1 in the i^{th} position
  - Only one input becomes active at any given time
  - Sometimes, two inputs may be active simultaneously which results in incorrect output (ambiguity)

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

Four-input OR gates

Octal-to-Binary Encoder

- To resolve such ambiguity, some encoder circuits must establish an input priority
  - Ensure only one input to be encoded
  - Ex: put a higher priority for inputs with higher subscript numbers

Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The output will be still 110,
because D_6 has higher priority than D_3

Priority Encoder

- A combinational circuit that implements a priority function
  - If two (or more) inputs are equal to 1, the input having the highest priority takes precedence
  - The truth table for a four-input priority encoder is as follows
  - No conflict of representing a minterm should appear when using ‘X’

Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

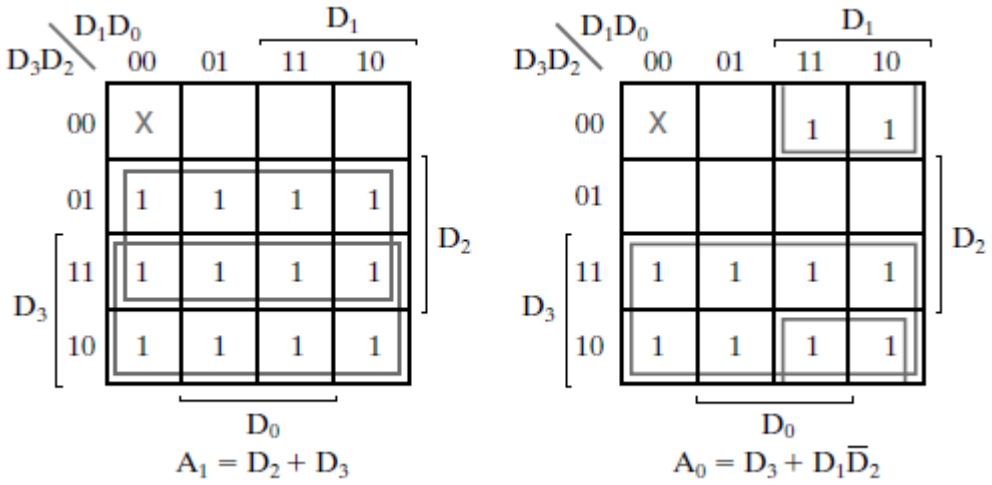
Valid bit

Priority Encoder

- A combinational circuit that implements a priority function
  - Consider the case when input D_3 has the highest priority
  - If D_3=1, then output becomes ’11 (binary 3)’
  - The output is 10 if D_2=1, provided that D_3=0

Truth Table of Priority Encoder

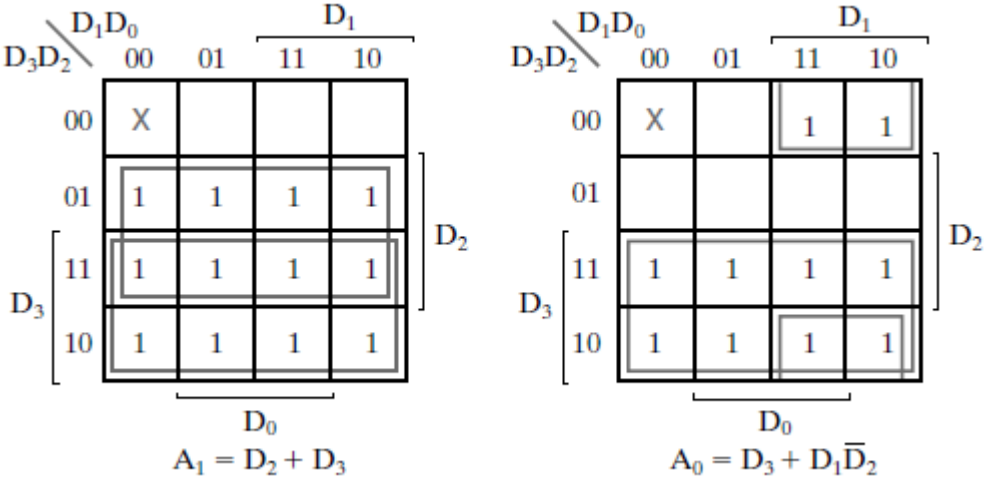
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1



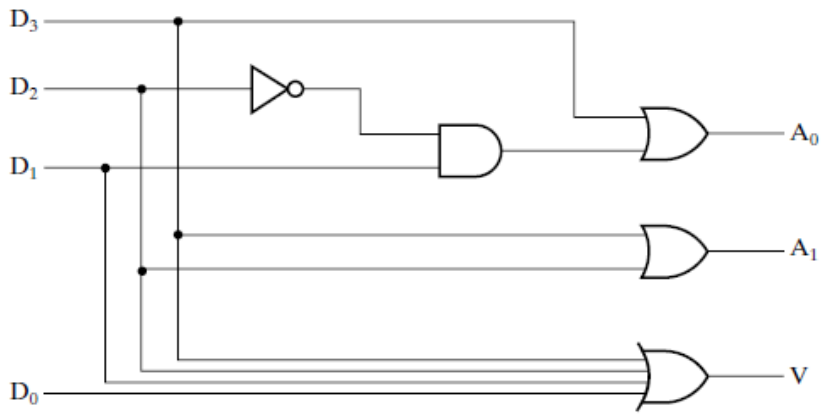
Simplification using map reduction

Priority Encoder

- A combinational circuit that implements a priority function
  - Consider the case when input D_3 has the highest priority
  - If D_3=1, then output becomes ’11 (binary 3)’
  - The output is 10 if D_2=1, provided that D_3=0



Simplification using map reduction



Logic diagram of a 4-input priority encoder

Selecting

- Selection of information to be used in a computer is a very important function
  - Selection circuits have a set of inputs from which selections are made, a single output, and a set of control lines
  - **Multiplexers** are widely used as selection circuits

Multiplexer (MUX)

- A multiplexer selects binary information from one of many input lines and directs it to a single output line
  - The selection is governed by ‘selection inputs’
  - Normally, there are 2^n input lines with ‘n’ selection inputs

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

Truth Table for 2-to-1-Line Multiplexer

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

4-to-1 Line Multiplexer

- The output Y depends on four inputs I_0, I_1, I_2, and I_3 and two select inputs S_1 and S_0
  - We can form a condensed truth table by putting information variables in the output column
  - Simply, it can be implemented by using two inverters, four 3-input AND gates, and a 4-input OR gate (gate-input cost: 18)

Condensed Truth Table for 4-to-1-Line Multiplexer

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

$$Y = \overline{S}_1\overline{S}_0\, I_0 + \overline{S}_1 S_0\, I_1 + S_1\overline{S}_0\, I_2 + S_1 S_0 I_3$$

Any different implementations?

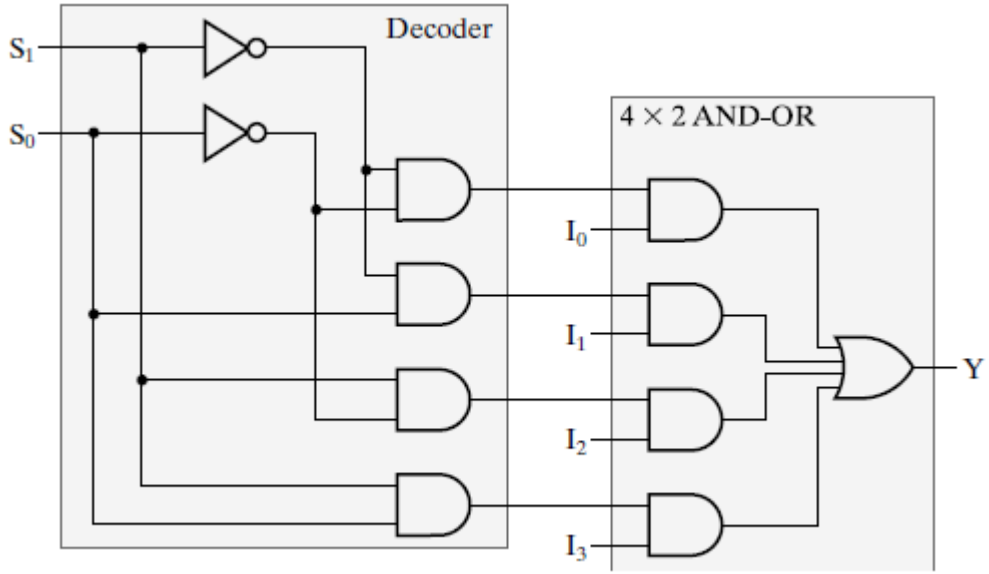
4-to-1 Line Multiplexer

- A different implementation is obtained by factoring the AND terms
  - Its implementation is combining a 2-to-4 line decoder, four AND gates (enabling circuit), and a 4-input OR gate
  - The resulting circuit has a gate-input cost of 22 (more costly!!)

Condensed Truth Table for 4-to-1-Line Multiplexer

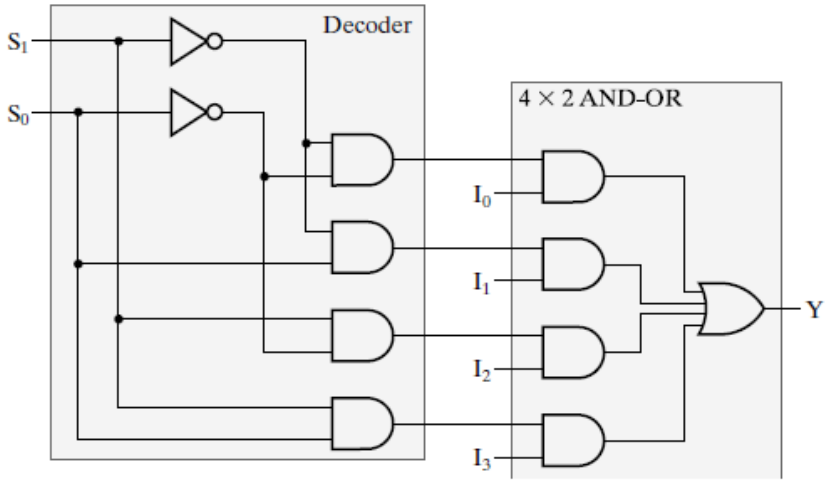
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

$$Y = (\bar{S}_1 \bar{S}_0) I_0 + (\bar{S}_1 S_0) I_1 + (S_1 \bar{S}_0) I_2 + (S_1 S_0) I_3$$

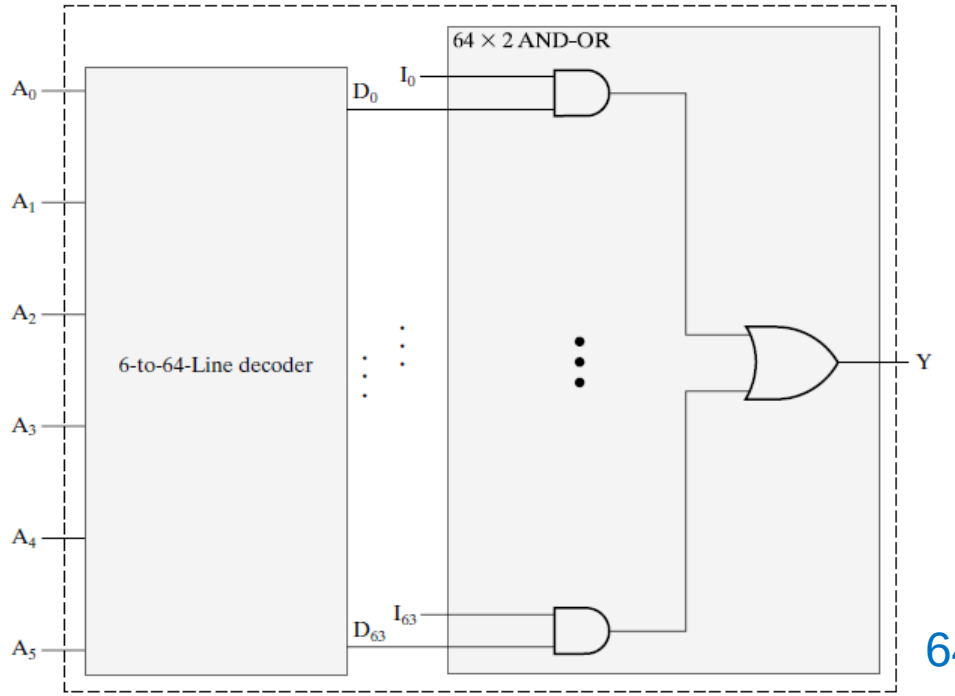


64-to-1 Multiplexer

- A multiplexer design can be easily expanded to 64-to-1 MUX
  - (6-to-64 decoder) + (64 x 2 AND-OR gate)
  - This structure has a gate-input cost of 183 + 128 + 64 = **374**
  - If this implementation is replaced by inverters and 7-input AND gates, the gate-input cost becomes 6 + 448 + 64 = **518**!!



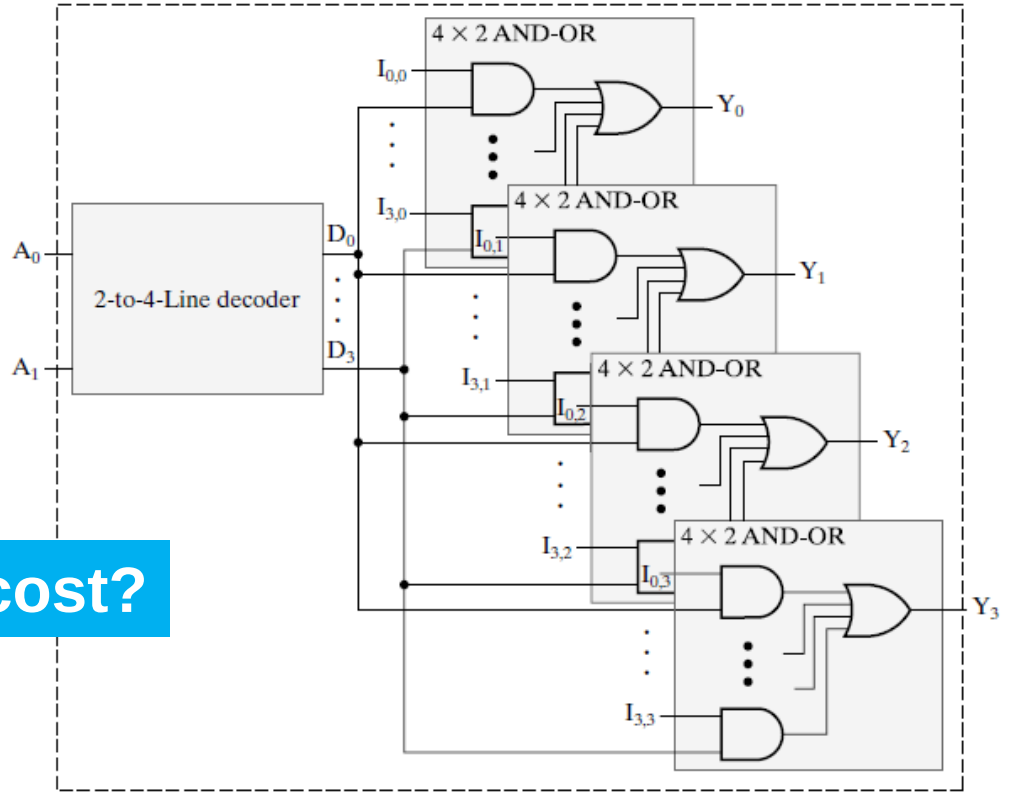
4-to-1 MUX



64-to-1 MUX

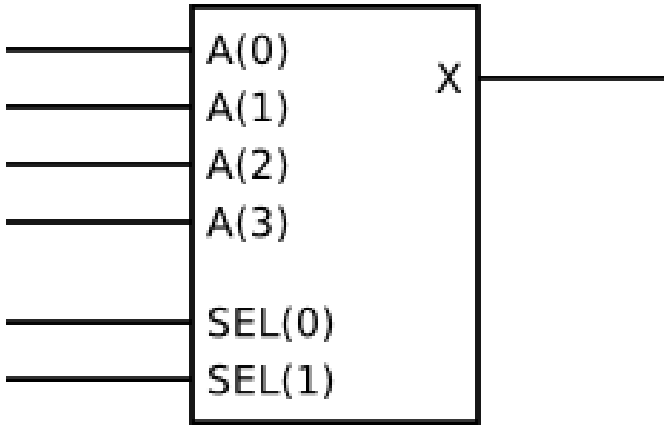
4-to-1 Quad Multiplexer

- It has two selection bits and each information input is replaced by a vector of four inputs
  - Since the information inputs are a vector, the output Y becomes a four-element vector



Gate-input cost?

VHDL example – 4 to 1 Multiplexer




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_4to1_top is
    Port ( SEL : in  STD_LOGIC_VECTOR (1 downto 0);     -- select input
           A   : in  STD_LOGIC_VECTOR (3 downto 0);     -- inputs
           X   : out STD_LOGIC);                        -- output
end mux_4to1_top;

architecture Behavioral of mux_4to1_top is
begin
with SEL select
    X <= A(0) when "00",
         A(1) when "01",
         A(2) when "10",
         A(3) when "11",
         '0'  when others;
end Behavioral;
```

Design Constraints

- Fan-out
  - The number of gate inputs driven by the output of another single logic gate.
- Power Dissipation (Primary design constraint in the last 15 years!)
  - As a result of applied voltage and currents flowing through the logics, some power will be dissipated in it in the form of heat.
- Propagation Delay
  - Propagation delay is the length of time taken for a signal to reach its destination
    - Wires have an approximate propagation delay of 1 ns for every 6 inches (15 cm) of length. Logic gates can have propagation delays ranging from more than 10 ns down to the picosecond range, depending on the technology being used.
- Noise Margin
  - the noise margin is the amount by which the signal exceeds the threshold for a proper '0' or '1'.

Summary

- There are five-step design procedure in designing a digital circuit
  - Specification, formulation, optimization, technology mapping, verification
  - These steps apply to both **manual** and **computer-aided** design
- We have dealt with functional blocks, combinational circuits that are frequently used to design larger circuits
  - Decoders
  - Encoders
  - Multiplexers