BME2322 – Logic Design

The Instructors:

Dr. Görkem SERBES (C317)

gserbes@yildiz.edu.tr

https://avesis.yildiz.edu.tr/gserbes/

Lab Assistants:

Nihat AKKAN

nakkan@yildiz.edu.tr

https://avesis.yildiz.edu.tr/nakkan

LECTURE 6

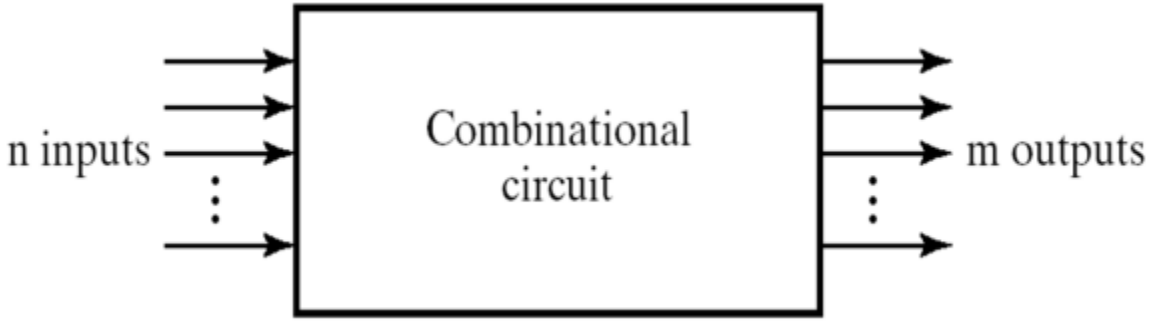
Combinational Circuit Design

- Up to now, we have learned all the prerequisite material:
  - Truth tables and Boolean expressions can be used to describe functions
  - Expressions can be converted into hardware circuits (Gates)
  - Boolean algebra and K-maps help simplify expressions and circuits
- Now, we can put all of these foundations to good use, to analyze and design some larger circuits

- Logic circuits for digital systems may be
  - Combinational or Sequential

- A combinational circuit consists of logic gates whose outputs at any time are determined by the current input values, i.e., it has no memory elements
- A sequential circuit consists of logic gates whose outputs at any time are determined by the current input values as well as the past input values, i.e., it has memory elements

Combinational Circuit Diagram



- Each input and output variable is a binary variable
- 2^n possible binary input combinations
- One possible binary value at the output for each input combination
- A truth table or m Boolean functions can be used to specify input-output relation

Design Procedure

1. Specification
   - Write a specification for the circuit if one is not already available
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
   - Apply hierarchical design if appropriate
3. Optimization
   - Apply 2-level and multiple-level optimization bu using Algebraic or K-map optimization
   - Draw a logic diagram or provide a netlist for the resulting circuit using ANDs, ORs, and inverters
4. Technology Mapping
   - Map the logic diagram or netlist to the implementation technology selected
5. Verification
   - Verify the correctness of the final design manually or using simulation

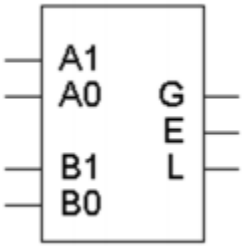
Example – Comparing 2-bit Numbers

Specification for 2-bit comparator

- Let's design a circuit that compares two 2-bit numbers, A and B. The circuit should have three outputs:
  - G ("Greater") should be 1 only when A > B
  - E ("Equal") should be 1 only when A = B
  - L ("Lesser") should be 1 only when A < B
- Make sure you understand the problem
  – Inputs A and B will be 00, 01, 10, or 11 (0, 1, 2 or 3 in decimal)
  – For any inputs A and B, exactly one of the three outputs will be 1

.

- Two 2-bit numbers means a total of four inputs
  – We should name each of them
  – Let's say the first number consists of digits A1 and A0 from left to right, and the second number is B1 and B0
- The problem specifies three outputs: G, E and L



Comparing 2-bit Numbers cont.

Formulation for 2-bit comparator

- To start with filling the truth table is a good way for problem solving. By this way, the relationship (>, =, <) between the inputs (A and B) can be explicitly shown
- A four-input function has a sixteen-row truth table
- It's usually clearest to put the truth table rows in binary numeric order; in this case, from 0000 to 1111 for A1, A0, B1 and B0
- Example: 01 < 10, so the sixth row of the truth table (corresponding to inputs A=01 and B=10) shows that output L=1, while G and E are both 0.

A1	A0	B1	B0	G	E	L
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1			
0	1	0	0			
0	1	0	1			
0	1	1	0	0	0	1
0	1	1	1			
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1			
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1			

Comparing 2-bit Numbers cont.

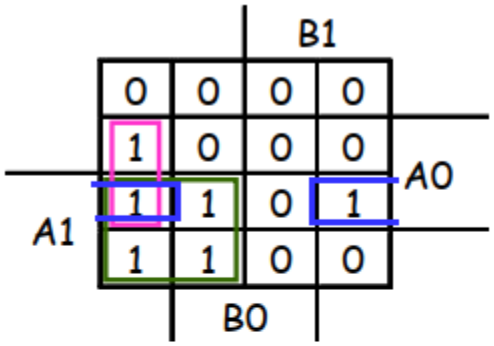
- The complete truth table

A1	A0	B1	B0	G	E	L
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
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	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

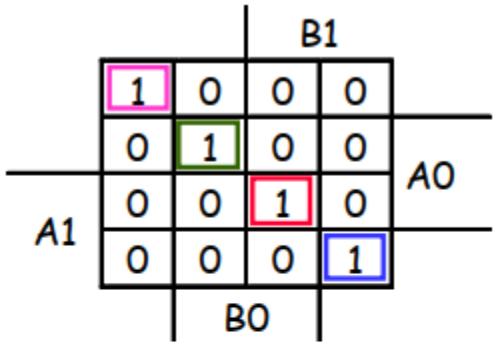
Comparing 2-bit Numbers cont.

Optimization for 2-bit comparator

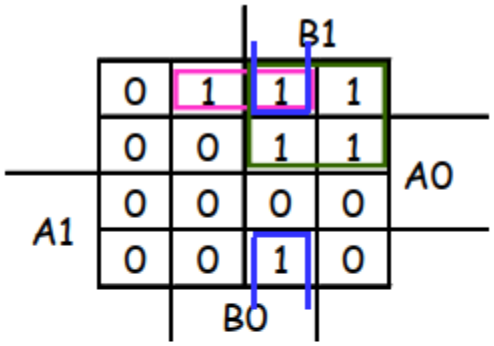
Let's use K-maps. There are three functions (each with the same inputs A1 A0 B1 B0), so we need three K-maps



G(A1,A0,B1,B0) =
A1 A0 B0' +
A0 B1' B0' +
A1 B1'



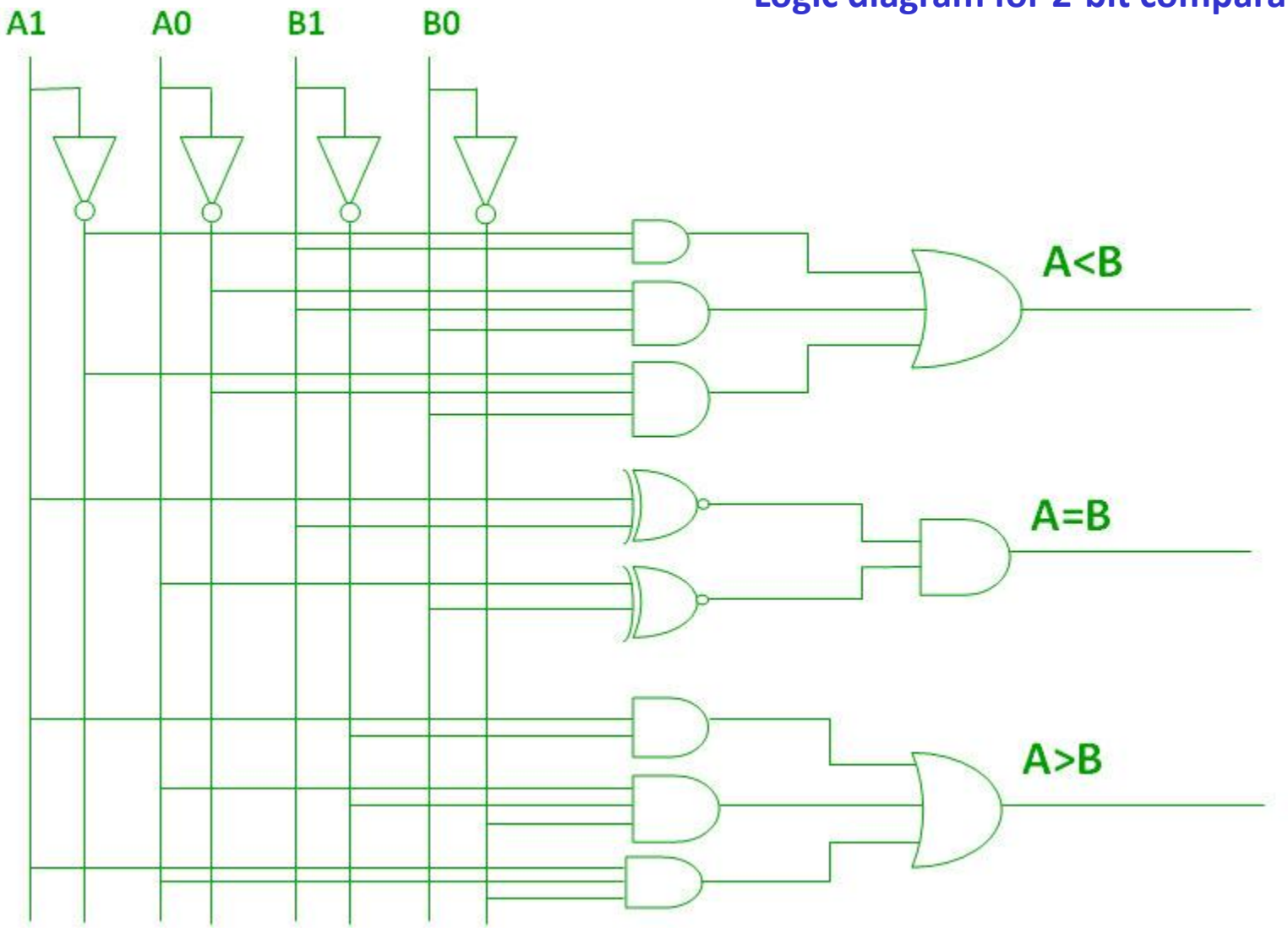
E(A1,A0,B1,B0) =
A1' A0' B1' B0' +
A1' A0 B1' B0 +
A1 A0 B1 B0 +
A1 A0' B1 B0'



L(A1,A0,B1,B0) =
A1' A0' B0 +
A0' B1 B0 +
A1' B1

Comparing 2-bit Numbers cont.

Logic diagram for 2-bit comparator



Technology Mapping

Mapping Procedures

- To NAND gates
- To NOR gates

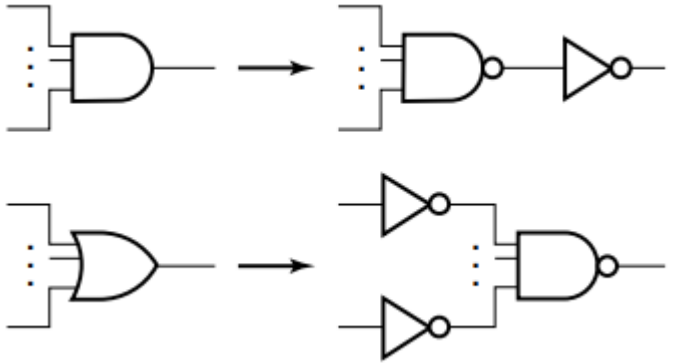
- Some implementation technology usually has a library with only one type of gate (such as 3-input NOR, or 3-input NAND)

- Technology mapping is a transformation of Boolean expressions into a logic schematic containing only given type(s) of gate(s)

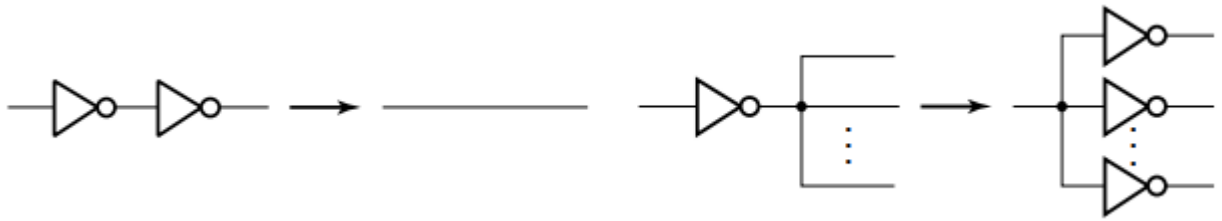
- The mapping is accomplished by:
  - Replacing AND and OR symbols,
  - Pushing inverters through circuit fan-out points,
  - Canceling inverter pairs

NAND Mapping Algorithm

1. Replace ANDs and ORs:

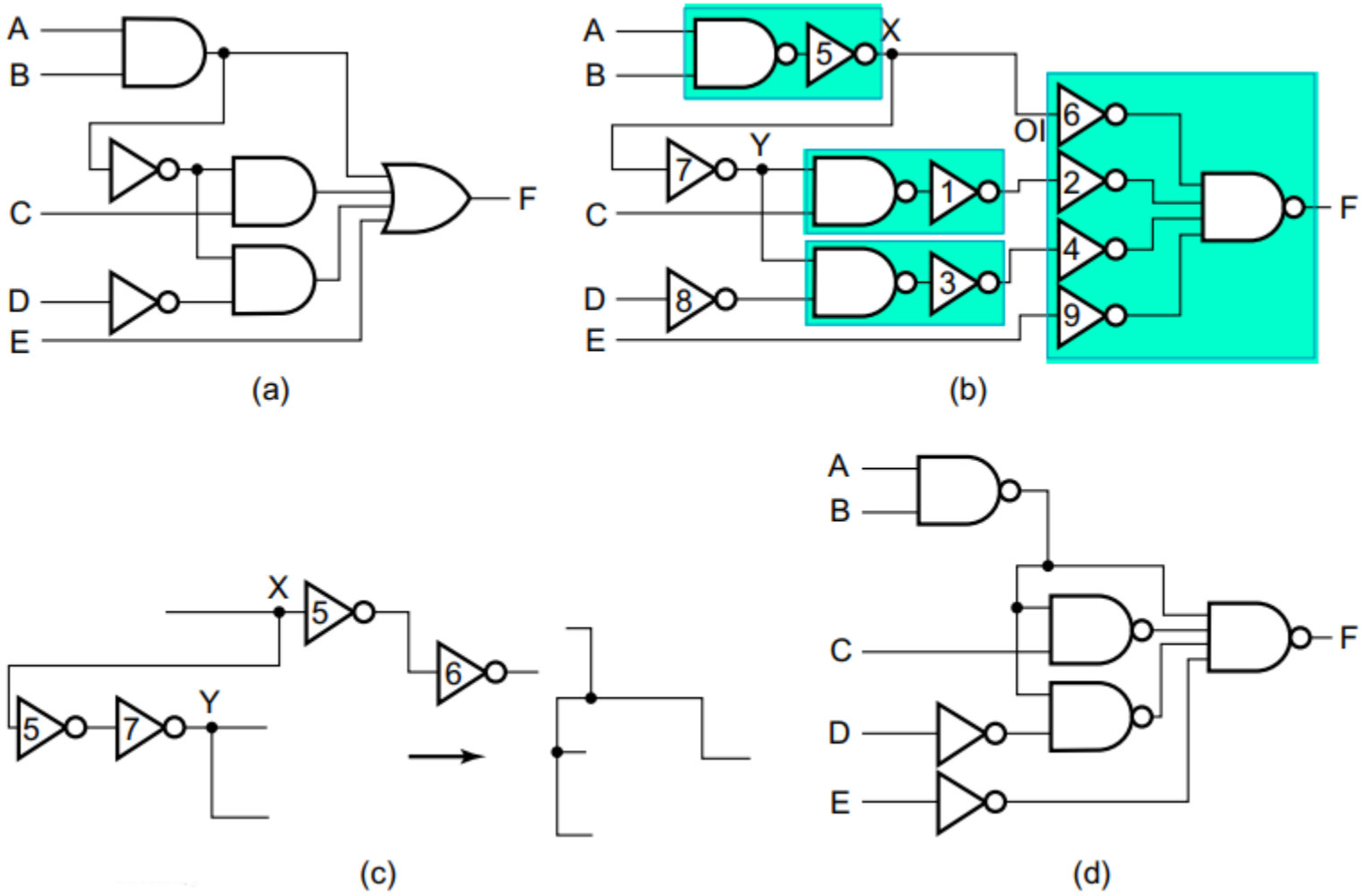


2. Repeat the following pair of actions until there is at most one inverter between :
   a. A circuit input or driving NAND gate output, and
   b. The attached NAND gate inputs.



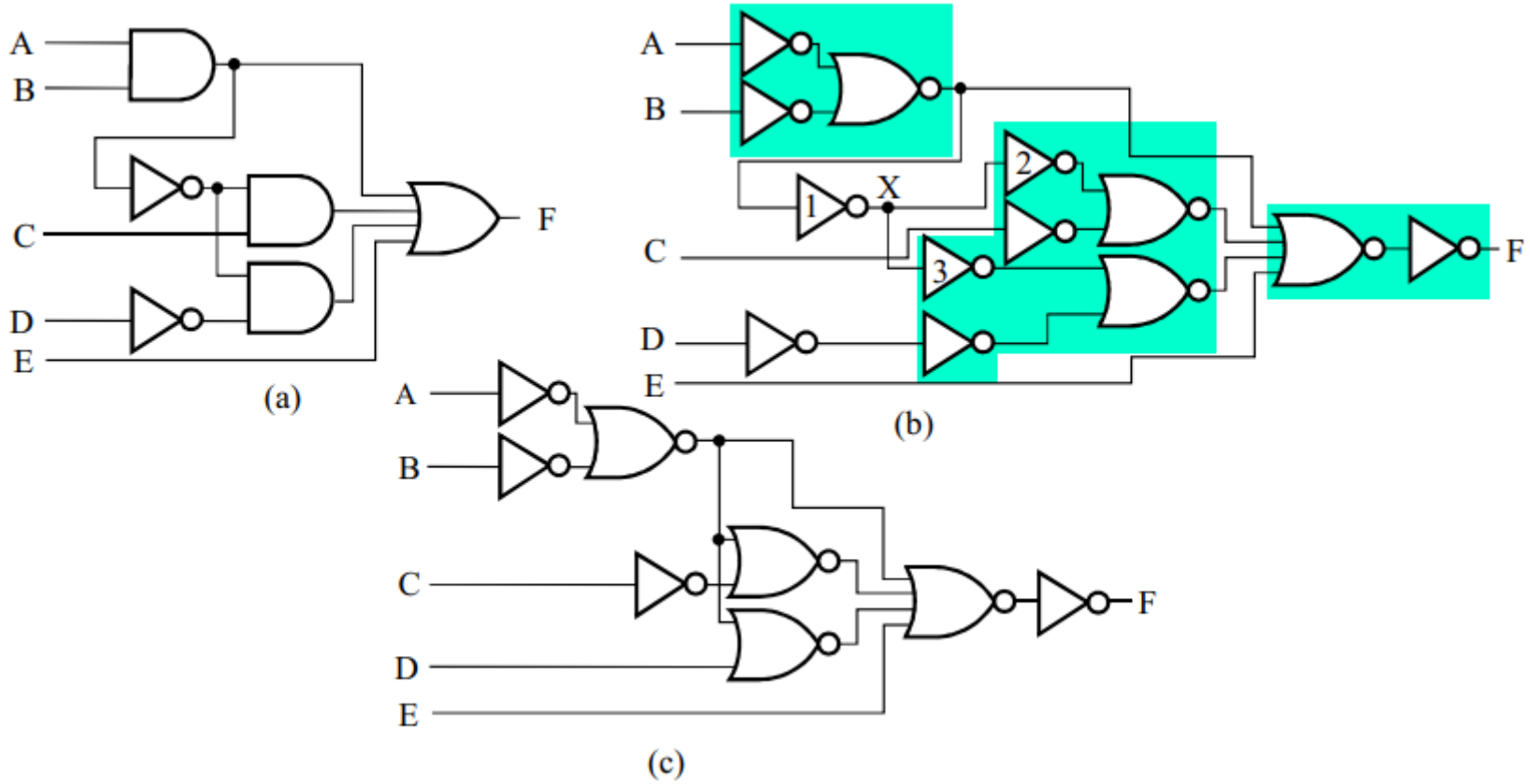
NAND Mapping Example

Example – F= AB + (AB)'C + (AB)'D' + E



NOR Mapping Example

Example – F= AB + (AB)'C + (AB)'D' + E



Verification

- Verification - show that the final circuit designed implements the original specification
- Simple specifications are:
  - truth tables
  - Boolean equations
  - HDL code

- Manual Logic Analysis
  - Find the truth table or Boolean equations for the final circuit
  - Compare the final circuit truth table with the specified truth table, or
  - Show that the Boolean equations for the final circuit are equal to the specified Boolean equations
- Simulation
  - Simulate the final circuit (or its netlist, possibly written as an HDL) and the specified truth table, equations, or HDL description using test input values that fully validate correctness.

Top-Down versus Bottom-Up

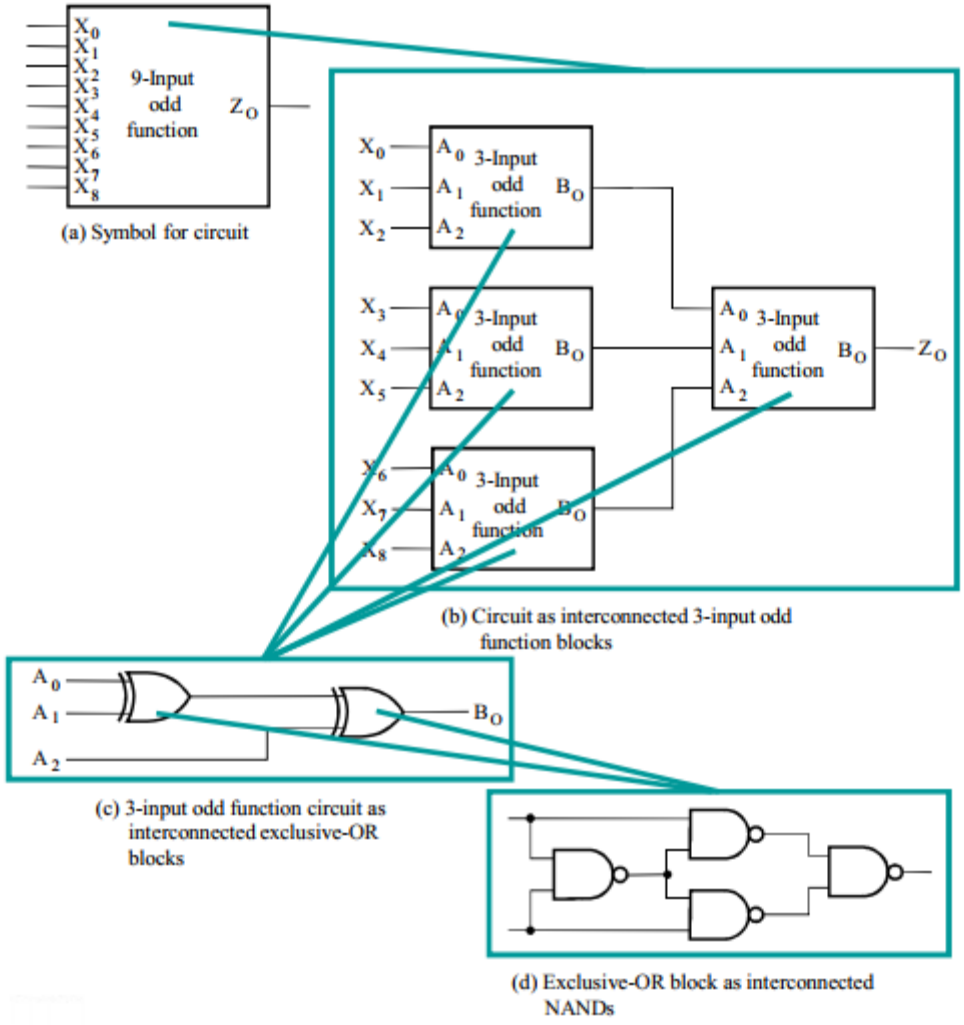
- A top-down design proceeds from an abstract, high-level specification to a more and more detailed design by decomposition and successive refinement
- A bottom-up design starts with detailed primitive blocks and combines them into larger and more complex functional blocks
- Design usually proceeds top-down to known building blocks ranging from complete CPUs to primitive logic gates or electronic components.
- Much of the material in this course is devoted to learning about combinational blocks used in top-down design.

Beginning Hierarchical Design

- A single very large-scale integrated (VLSI) processors circuit contains several tens of millions of gates!
- Imagine interconnecting these gates to form the processor
- No complex circuit can be designed simply by interconnecting gates one at a time
- Divide and Conquer approach is used to deal with the complexity

  - Break up the circuit into pieces (blocks)
  - Define the functions and the interfaces of each block such that the circuit formed by interconnecting the blocks obeys the original circuit specification
  - If a block is still too large and complex to be designed as a single entity, it can be broken into smaller blocks
  - Any block not decomposed is called a primitive block
  - The collection of all blocks including the decomposed ones is a hierarchy

Hierarchy for Parity Tree Example



Designing Complex Circuits

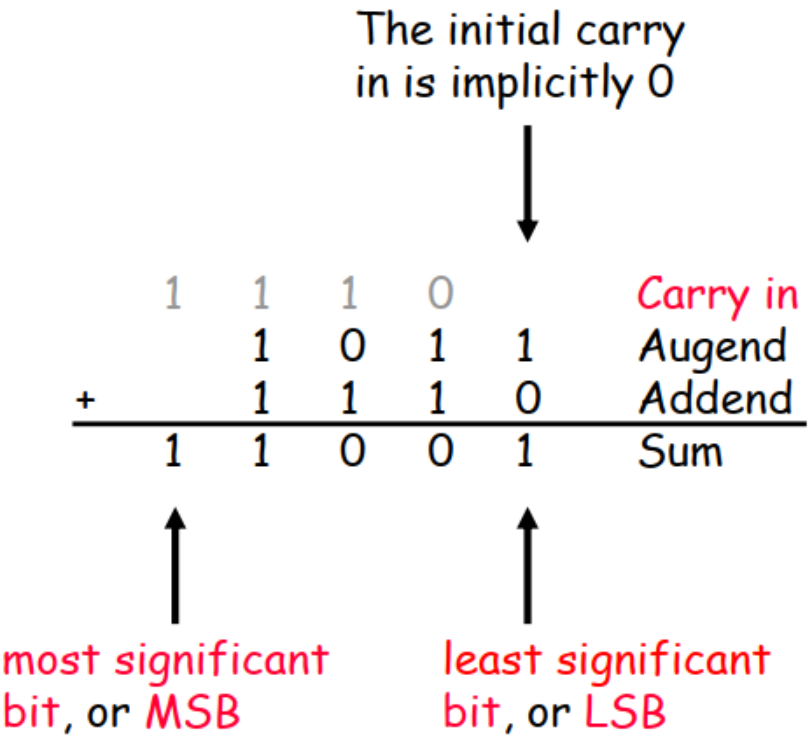
- Computer-Aided Design (CAD) tools
  - Schematic capture tools: Support the drawing of blocks and interconnections at all levels of the hierarchy
  - Libraries of graphic sysmbols
  - Logic Simulator
- Hardware Description Languages (HDLs)
  - VHDL and Verilog, both are the IEEE standard
  - VHDL: Very High Speed Integrated Circuits (VHSIC) HDL
  - Like programming languages, but tuned to describe hardware structures and behaviour
  - Alternative to schematics (structural description)
  - Behavioural description also possible
  - Logic synthesis: Register Transfer Level (RTL) of a system -> Netlist (structural description)

Levels of Integration

- Digital circuits are constructed with integrated circuits
- An integrated circuit (IC) is a silicon semiconductor crystal (informally a chip) containing the electronic comonents for the digital gates and storage elements
- Small-scale integrated (SSI): Primitive gates, # of gates < 10
- Medium-scale integrated (MSI): Elementary digital functions (4-bit addition), 10 < # of gates < 100
- Large-scale integrated (LSI): Small processors, small memories, programmable modules, 100 < # of gates < a few thousand
- Very large-scale integrated (VLSI): Complex microprocessors and digital signal processing chips, several thousand to tens of millions of gates

Arithmetic Functions - Adders

- You can add two binary numbers one column at a time starting from the right, just as you add two decimal numbers
- But remember that it's binary. For example, 1 + 1 = 10 and you have to carry!

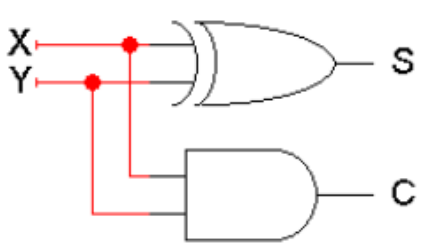


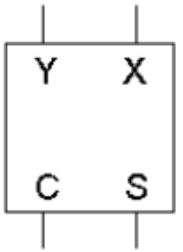
Adding Two Bits

- A hardware adder by copying the human addition algorithm
- Half adder: Adds two bits and produces a two-bit result: a sum (the right bit) and a carry out (the left bit)
- Here are truth tables, equations, circuit and block symbol

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

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10





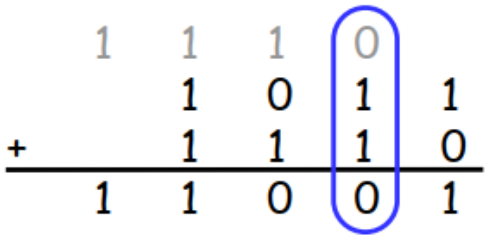
$C = XY$

$S = X'Y + XY' = X \oplus Y$

Be careful! Now we're using + for both arithmetic addition and the logical OR operation.

Adding Tree Bits – Full Adder

- what we really need to do is add three bits: the augend and addend, and the carry in from the right.



X	Y	C_{in}	C_{out}	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

0 + 0 + 0 = 00
0 + 0 + 0 = 01
0 + 1 + 0 = 01
0 + 1 + 1 = 10
1 + 0 + 0 = 01
1 + 0 + 1 = 10
1 + 1 + 0 = 10
1 + 1 + 1 = 11

Full Adder

- Full adder: Three bits of input, two-bit output consisting of a sum and a carry out
- Using Boolean algebra, we get the equations shown here

- XOR operations simplify the equations a bit
- We used algebra because you can't easily derive XORs from K-maps

X	Y	C_{in}	C_{out}	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

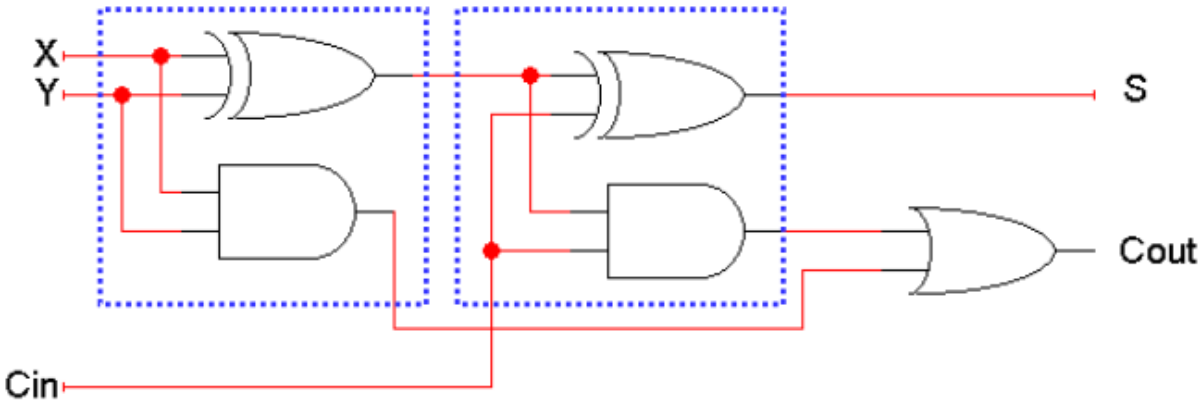
$$
\begin{aligned}
S &= \Sigma m(1,2,4,7) \\
&= X' Y' C_{in} + X' Y C_{in}' + X Y' C_{in}' + X Y C_{in} \\
&= X' (Y' C_{in} + Y C_{in}') + X (Y' C_{in}' + Y C_{in}) \\
&= X' (Y \oplus C_{in}) + X (Y \oplus C_{in})' \\
&= X \oplus Y \oplus C_{in}
\end{aligned}
$$

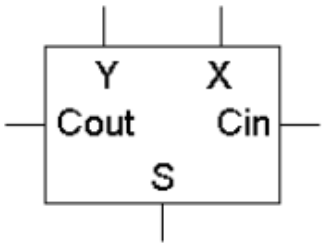
$$
\begin{aligned}
C_{out} &= \Sigma m(3,5,6,7) \\
&= X' Y C_{in} + X Y' C_{in} + X Y C_{in}' + X Y C_{in} \\
&= (X' Y + X Y') C_{in} + XY(C_{in}' + C_{in}) \\
&= (X \oplus Y) C_{in} + XY
\end{aligned}
$$

Full Adder Circuit

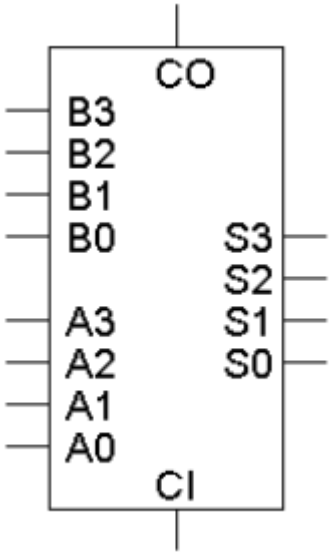
These things are called half adders and full adders because you can build a full adder by putting together two half adders!

$$S = X \oplus Y \oplus C_{in}$$
$$C_{out} = (X \oplus Y)\, C_{in} + XY$$

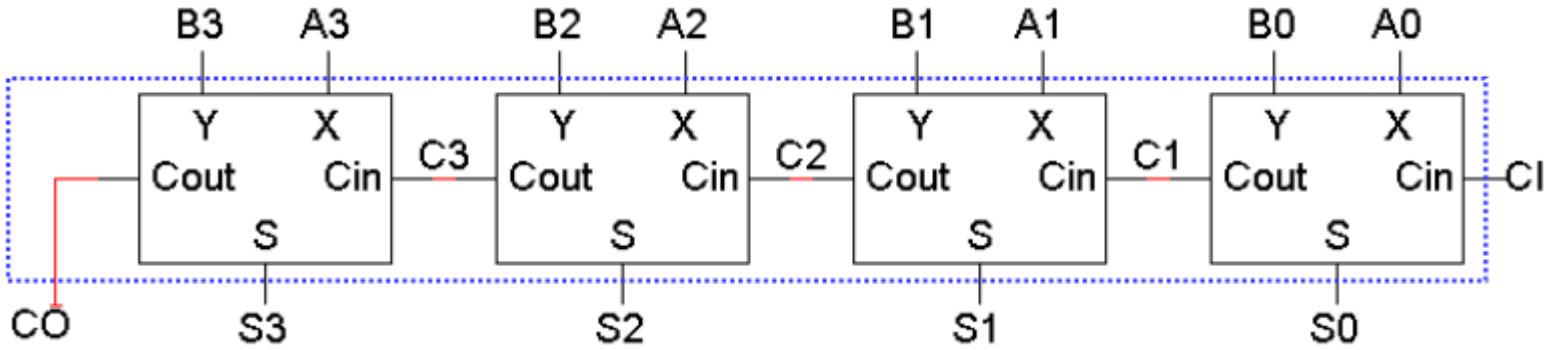




A 4-Bit Adder

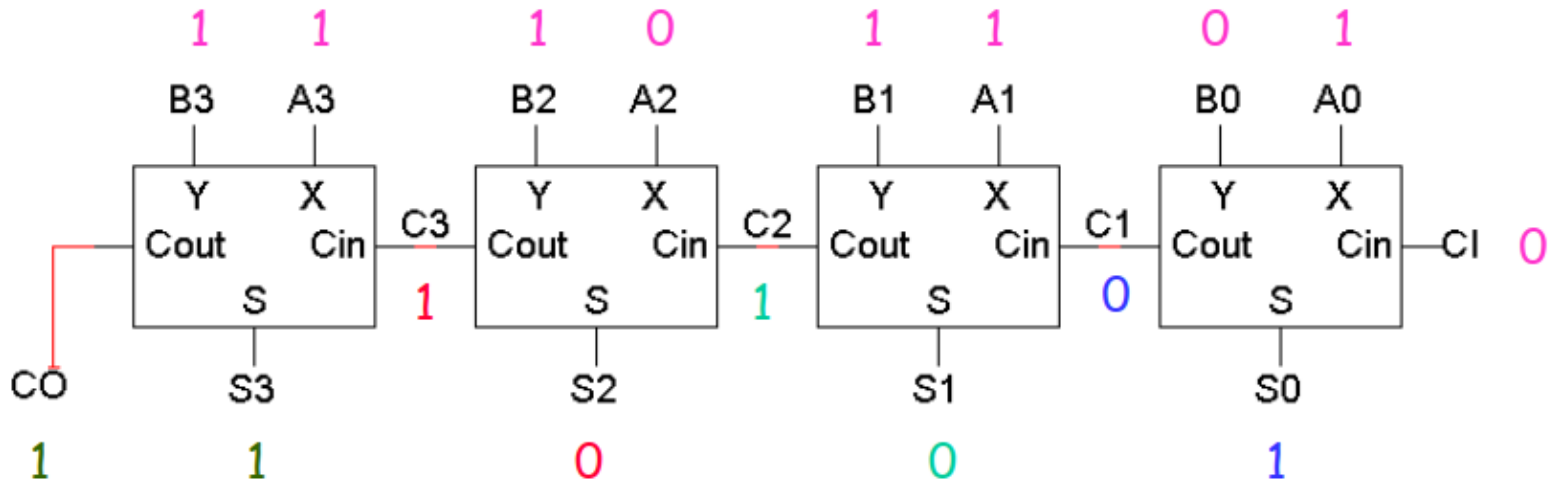


- Four full adders together make a 4-bit adder
- There are nine total inputs:
  – Two 4-bit numbers, A3 A2 A1 A0 and B3 B2 B1 B0
  – An initial carry in, CI
- The five outputs are:
  – A 4-bit sum, S3 S2 S1 S0
  – A carry out, CO
- Imagine designing a nine-input adder without this hierarchical structure, you'd have a 512-row truth table with five outputs!



An example of 4-bit addition

- Let's try our initial example: A=1011 (eleven), B=1110 (fourteen)



1. Fill in all the inputs, including CI=0
2. The circuit produces C1 and S0 (1 + 0 + 0 = 01)
3. Use C1 to find C2 and S1 (1 + 1 + 0 = 10)
4. Use C2 to compute C3 and S2 (0 + 1 + 1 = 10)
5. Use C3 to compute CO and S3 (1 + 1 + 1 = 11)

The final answer is 11001 (twenty-five)

In this case, note that the answer (11001) is *five* bits long, while the inputs were each only four bits (1011 and 1110). This is called overflow