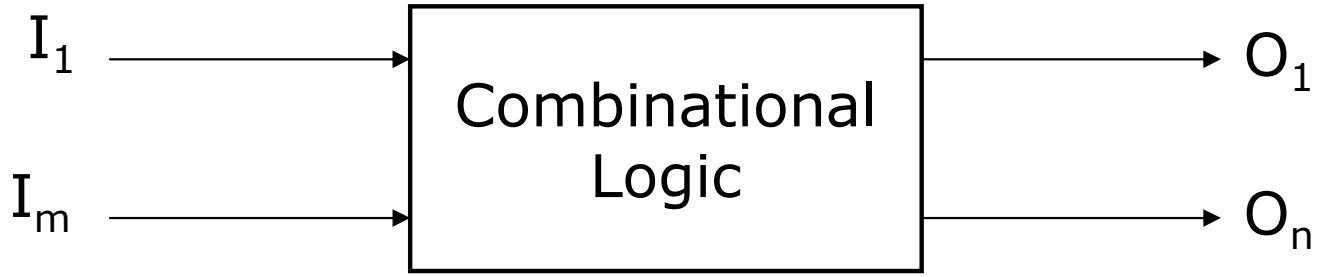
CENG-232
Logic Design

Lecture 3
Combinational Logic Design

Spring 2015 - Uluç Saranlı
saranli@ceng.metu.edu.tr

Combinational Logic

- One or more digital signal inputs
- One or more digital signal outputs
- Outputs are only functions of current input values (ideal) plus logic propagation delays



$$O_1(t+\Delta t) = F_1(I_1(t),\ldots I_m(t))$$
$$\vdots$$
$$O_n(t+\Delta t) = F_n(I_1(t),\ldots I_m(t))$$

Combinational Logic (Cont'd.)

- Combinational logic has no memory!
  - Outputs are only function of current input combination
  - Nothing is known about past events
  - Repeating a sequence of inputs always gives the same output sequence
- Sequential logic (covered later) does have memory
  - Repeating a sequence of inputs can result in an entirely different output sequence

Design Hierarchy

- Large systems are usually too complex to design as a single entity
  - i.e., a 16 bit binary adder has 32 inputs and therefore there are 232 rows in the truth table!
- System is usually partitioned into smaller parts which are further partitioned …
- This defines a hierarchy of design from complex to simple;
  - “top to bottom”

Design Methodologies

- There are two basic approaches to system design
  - Top-Down: start at the top system level and decompose into ever simpler subsystems and components
  - Bottoms-Up: start with known low-level building blocks and put them together into increasing complex functions

- Ideally either should work; however, in practice neither method does

Concurrent Design

- The practical approach is to combine the two
  - Basic top-down to provide proper decomposition and validation
  - BUT as you decompose functions, be aware of:
    - already existing and available components
    - component to component interface characteristics
    - reality - cost, size, weight, power, etc.

- If done properly, you end up with a low-cost practical solution that works!

Rapid Prototyping and CAD

- Design verification is much more difficult with VLSI ASICs that with SSI designs
  - Lots more signals and less accessibility
- Rapid prototyping assumes we can built many different versions and see which ones work
  - Programmable logic is vital to this approach
  - Good development tools are also essential
- Hardware description languages are the way we quickly specify and change our designs

Hardware Description Languages (HDLs)

- Two main HDLs in use today
  - VHDL
  - Verilog
- Both are IEEE standards
- Both allow us to specify logic designs as textual descriptions
  - BE AWARE - both look like a software procedure but are describing HARDWARE!
    - Concurrency!!!
- We will use Verilog

Logic Synthesis

- Logic synthesis translates the HDL to our hardware implementation
  - 1st phase translates HDL to a generic, ideal logic description
    - logic expressions generated and minimized
    - allows us to verify functional operation
  - 2nd phase targets the design to the final physical device
    - complexity, speed, delays, power must be addressed
    - we can now simulate physical operation of device

Digital Logic Implementation

- Circuit Properties: logic representation, size, weight, power, package, temperature, COST
- Levels of Integration: ranges from small scale ICs with a few basic gates per package to VLSI devices containing millions of gates per package
- Circuit Technology: TTL, ECL, CMOS, GaAs, SiGe, etc.

Technology Parameters

- Fan-in
- Fan-out
- Noise margin
- Propagation delay
- Power dissipation

Fan-In

- For logic gates, it's the max number of inputs to a specific gate
  - Defined by gate design; usually limited to a max of 4 or 5
  - E.G. 74LS08 is a Quad, 2-input AND device
    - 4 AND gates in package, each AND has 2 inputs

- Primary impact is when you have more variables than your gate has input
  - Cascade gates, transform function, etc.

Fan-Out

- Fan-out is usually defined as the max number of “standard” logic gate inputs that can be connected to a logic gate output
  - Specifies the “drive” capability of an output
- If an output is overloaded, other characteristics such as noise margin, rise & fall times are degraded
- Different logic types are affected by “overload” in different ways

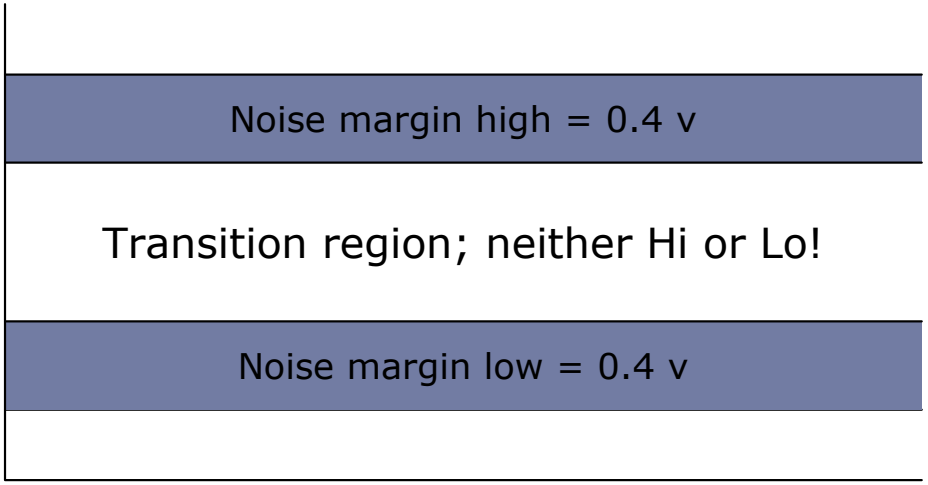
Noise Margin

- Noise Margin defines how much noise can be induced onto a logic signal and still be correctly recognized as a high or low level
  - Difference between output high or low level and input level that will recognized as high or low

TTL Logic Levels

V_{oh} = 2.8 v
V_{ih} = 2.4 v

V_{il} = 0.8 v
V_{ol} = 0.4 v



Propagation Delay

- Real devices do not have zero delay!
- Propagation delays are measured from input change to output change (tPD)
  - Usually referenced to 50% point on transition
- Gates usually have different delays for the output low to high (tLH) and high to low (tHL)
- Best to design using the max of the two
- Not all input changes show up at the output
  - Gate may not respond to a narrow pulse

Power Dissipation

- The quantity of electrical power that is dissipated by the device as heat
  - Devices have temperature operating range that device cannot exceed
- Power dissipation is mainly static or dynamic depending on the logic type
  - TTL/ECL dissipation is mainly static and therefore independent of signal rate of change
  - CMOS dissipation is mainly dynamic and increases linearly with increasing signal freq.

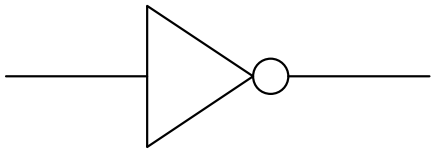
Signal Active States and Bubbles

- Primarily applies to control signals; used to denote when a condition is active or enabled
  - Active State - signal state (0 or 1) that indicates the assertion of some condition or action
    - Also called the excitation state
    - A signal is asserted when it is in the active state
    - A signal is negated when it is in the inactive state
    - Active-1 (active high) is when active state is logic 1
    - Active-0 (active low) is when active state is logic 0
  - Symbol pins without bubbles denote active-1
  - Symbol pins with bubbles denote active-0

Active States and Bubbles (Cont'd.)

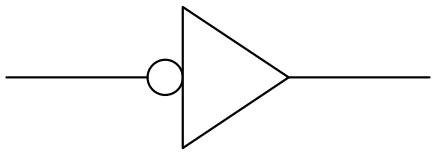
- Inverter (NOT) has two different forms

Input asserted
active-1



Output asserted
active-0

Input asserted
active-0



Output asserted
active-1

Alternative Symbols

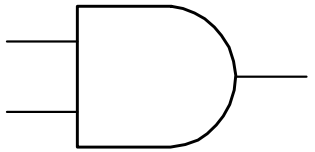
- The NOT example can be extended to all logic gates
  - Each logic gate has two equivalent symbols
    - The one we've seen so far for active-1 inputs
    - The alternate for active-0 inputs
  - In each case the gate operates the same
    - The only difference is how we interpret the values

AND Gate Alternate Symbol

X	Y	X·Y
0	0	0
0	1	0
1	0	0
1	1	1

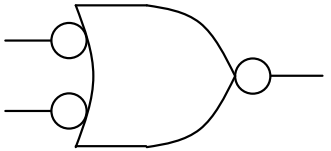
Active-1

X	Y	X·Y
F	F	F
F	T	F
T	F	F
T	T	T



Active-0

X	Y	X+Y
T	T	T
T	F	T
F	T	T
F	F	F

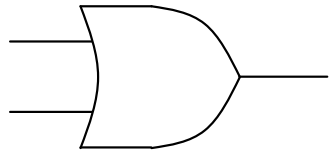


OR Gate Alternate Symbol

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

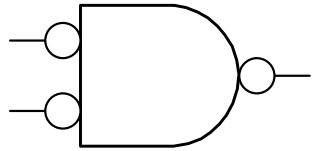
Active-1

X	Y	X+Y
F	F	F
F	T	T
T	F	T
T	T	T

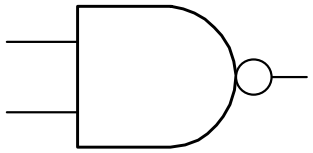
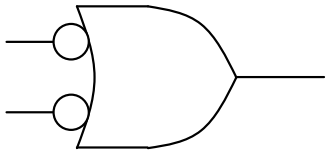
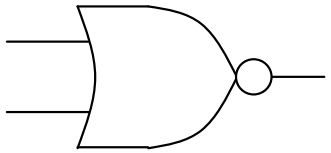
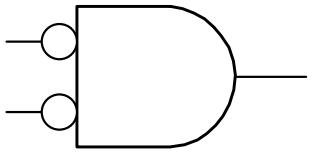
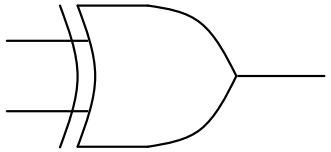
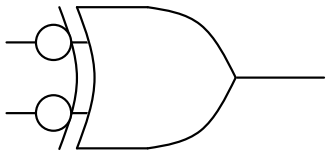
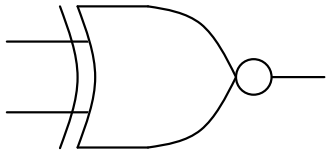
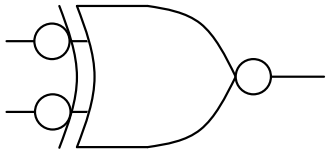


Active-0

X	Y	X·Y
T	T	T
T	F	F
F	T	F
F	F	F



Other Gate Alternative Symbols

	Active-1	Active-0
NAND		
NOR		
XOR		
XNOR		

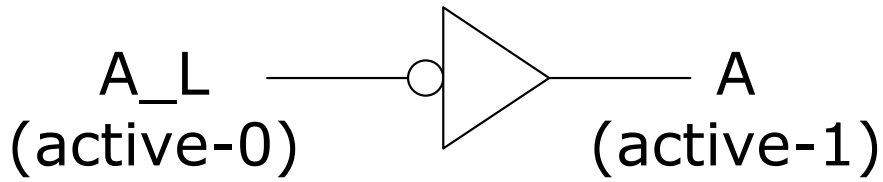
Signal Naming Conventions

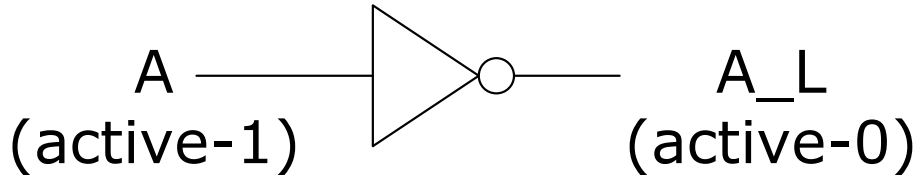
- The problem is how to distinguish between active-1 and active-0 signals.
  - Barring a signal name to designate active-0 is not recommended
    - Is A active-0 or NOT A?? $\overline{A}$
  - Use suffix of '_0'; (i.e A_0) after signal name
  - Use suffix of '_LO' or '_L'
  - Use suffix of '_BAR'

- No matter what you use, BE CONSISTENT!

Signal Naming (Cont'd.)

- Active-0 signal naming and symbol bubbles require some thought to interpret properly

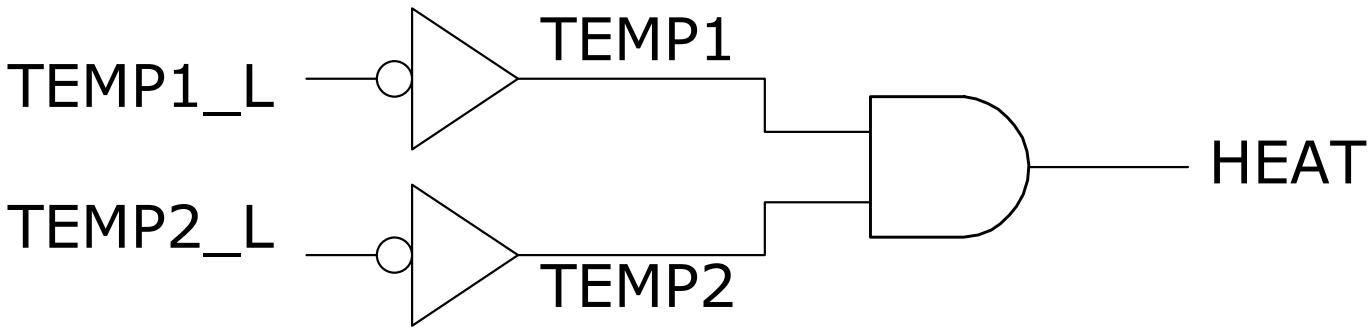


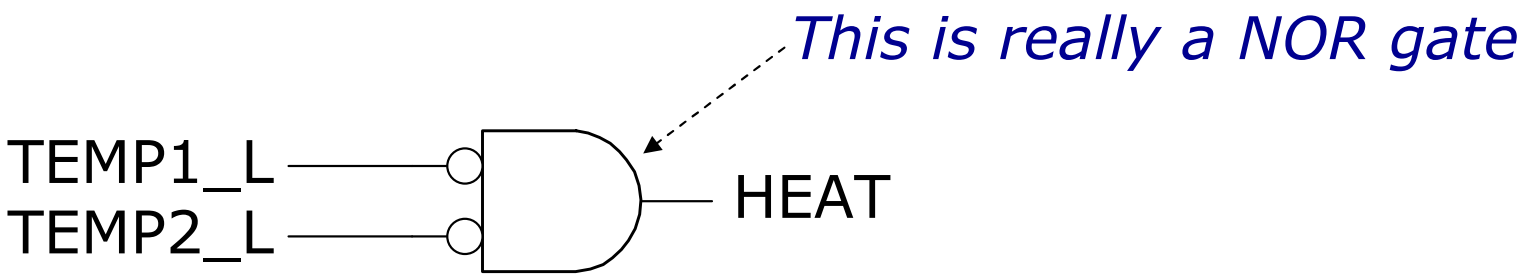


Naming and Alternate Symbols

- Proper active-0 signal naming and usage of alternate symbols can clarify the circuit intent

$$HEAT = TEMP1\_L \cdot TEMP2\_L$$





Design Methodology

- We start with some form of a problem statement
  - Usually just text; ambiguous, poorly stated
- We must produce a design representation
  - S.A. expressions, minimized
- The primary problem we have, is first to concisely define the true problem we are to solve
  - Define the “system” requirements

Design Methodology (Cont'd.)

- Step 1 - Break down the problem statement
  - Identify system inputs and outputs
  - Extract the stated input-output relationship(s)
  - State the above as system (black-box) level requirements. BE PRECISE!

- Step 2 - Perform initial system definition
  - Define interface variables and representation
    - If representation is not defined in problem statement, make preliminary assignment
  - Restate I/O relationship as algorithm, equation, simulation, etc.

Design Methodology (Cont'd.)

- Step 3 - Translate relationships to logic representation
  - Construct truth table, generate S.A. expressions
  - This is the formal statement of the I/O relationship(s)

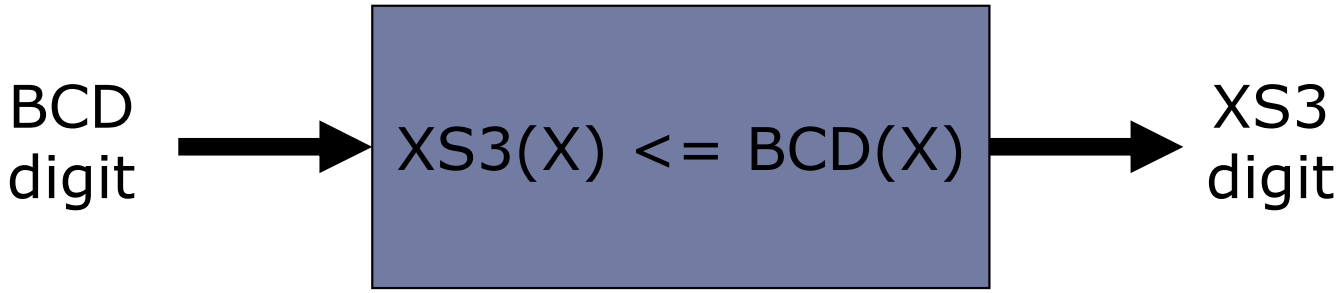
- Step 4 - Generate minimal set of logic expressions
  - Rapid prototyping development tools
  - Karnaugh maps, Quine-McCluskey, etc.

Design Methodology (Cont'd.)

- Step 5 - Implement and verify the design
  - Rapid prototyping (programmable logic) - target device and simulate timing behavior
  - Otherwise, draw schematic
    - Lots of ways to actually implement the equations
    - Must know what logic family you are to use.
  - Acid test is to build the circuit and test it
    - Must operate in real-world environment
    - Noise, temperature, other factors may cause problems

BCD to XS3 Example

- Initial Statement: "Design a circuit to convert BCD to XS3."
  - 1) We need to restate and translate this to specific requirements
    - R1: The circuit shall input one BCD digit
    - R2: The circuit shall output one XS3 digit
    - R3: The XS3 output shall be the equivalent decimal value as the BCD input value



BCD to XS3 Example (Cont'd.)

- 2) Now we need to define the interfaces in detail
  - We know that the input is one decimal digit in BCD representation, i.e. 4 bits, BCD := {b3, b2, b1, b0}
  - The output is one XS3 decimal digit, which is also 4 bits, i.e. XS3 := {x3, x2, x1, x0}
  - Usually, well known representations don't need explicit definition; when in doubt, DEFINE IT!

- 3) Now we use a truth table to define the logical input/output relationship
  - Only 10 of 16 possible input combinations are valid
  - We'll assume last 6 won't occur; i.e. are don't cares

BCD to XS3 Example (Cont'd.)

b_3	b_2	b_1	b_0	x_3	x_2	x_1	x_0
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	-	-	-	-
1	0	1	1	-	-	-	-
1	1	0	0	-	-	-	-
1	1	0	1	-	-	-	-
1	1	1	0	-	-	-	-
1	1	1	1	-	-	-	-

Note: Don't cares can work to our advantage during minimization; we can assign either 0 or 1 as needed.

BCD to XS3 Example (Cont'd.)

- 4) Now we can generate the logical expressions for the outputs (canonical SofP form)

$$x_3 = \overline{b_3}b_2\overline{b_1}b_0 + \overline{b_3}b_2b_1\overline{b_0} + \overline{b_3}b_2b_1b_0 + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}b_0$$

$$x_2 = \overline{b_3}\,\overline{b_2}\,\overline{b_1}b_0 + \overline{b_3}\,\overline{b_2}b_1\overline{b_0} + \overline{b_3}\,\overline{b_2}b_1b_0 + \overline{b_3}b_2\overline{b_1}\,\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}b_0$$

$$x_1 = \overline{b_3}\,\overline{b_2}\,\overline{b_1}\,\overline{b_0} + \overline{b_3}\,\overline{b_2}b_1b_0 + \overline{b_3}b_2\overline{b_1}\,\overline{b_0} + \overline{b_3}b_2b_1b_0 + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0}$$

$$x_0 = \overline{b_3}\,\overline{b_2}\,\overline{b_1}\,\overline{b_0} + \overline{b_3}\,\overline{b_2}b_1\overline{b_0} + \overline{b_3}b_2\overline{b_1}\,\overline{b_0} + \overline{b_3}b_2b_1\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0}$$

BCD to XS3 Example (Cont'd.)

- The minimized equations are as follows:

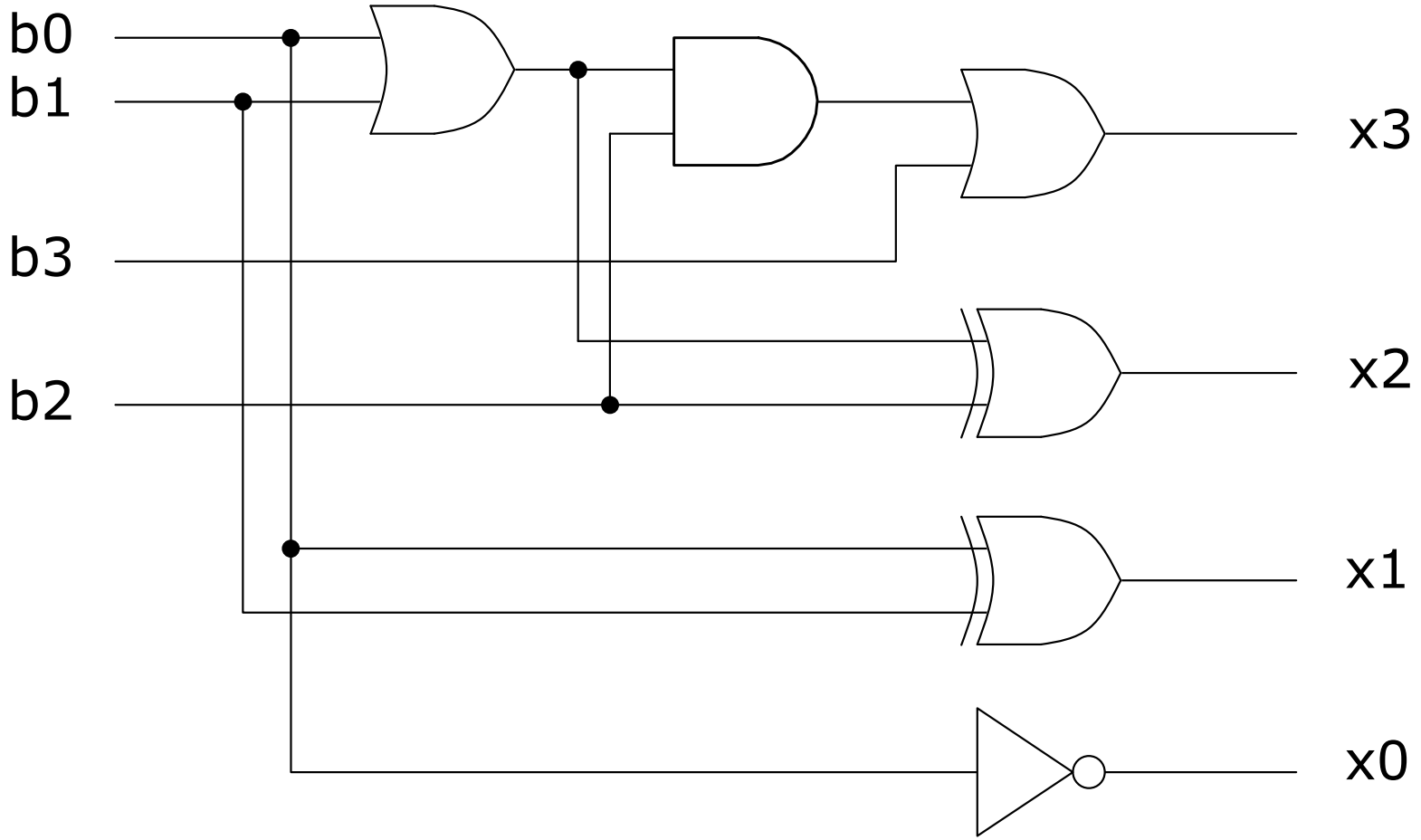
$$x_3 = b_3 + b_2 b_1 + b_2 b_0$$

$$x_2 = \overline{b_2} b_1 + \overline{b_2} b_0 + b_2 \overline{b_1 b_0}$$

$$x_1 = \overline{b_1 b_0} + b_1 b_0 = \overline{b_1 \oplus b_0}$$

$$x_0 = \overline{b_0}$$

BCD to XS3 Example (Cont'd.)



Technology Mapping

- Translation of out “ideal” circuit design to actual hardware must account for the implementation method
  - ASICs: full custom, standard cell, or gate arrays
  - Programmable logic: FPGAs or PLDs
  - Gate types, input configurations available
- Vendors supply you with a set of logic gate design patterns known as a cell library
  - Defines implementation rules as well as gate types
- CAD tools then use the provided library to map the ideal design to the physical implementation
  - Translates design to preferred logic types
  - Checks for problems, fan-out, propagation times exceeded

Verification

- Ensuring that the final device actually works is mandatory and can also be hard to do
  - Must start with good requirements (validation)
    - It's really bad to find out your design meets the stated requirements but it's not what the customer wanted
  - Done at different stages of the development
    - Simulation used during the design capture and implementation mapping phase
    - Functional and parametric testing after device fabrication