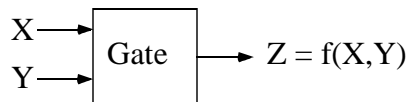


Gates

- Gate: A simple electronic circuit (a system) that realizes a logical operation.
 - The direction of information flow is from the input terminals to the output terminal.
 - The number of input and output terminals is finite and they carry binary-valued signals (i.e., 0 and 1).
 - The transformation of input signals to output signals can be modeled as a logical operation.
- Reading assignment
 - Sections 2.1—2.5 from the text



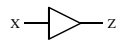
□ Truth Tables

- Since there is a finite number of input signal combinations, we can represent the behavior of a gate by simply listing all of its possible input configurations and the corresponding output signal. Such a list is called a *truth table*.
- For example, the following gate could have the behavior given by the following truth tables.



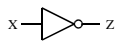
- The use of the symbols L and H usually correlates with the high and low voltages.

□ Some standard gates and their symbols and truth tables:



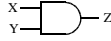
| X | Z |
|---|---|
| 0 | 0 |
| 1 | 1 |

a. Buffer



| X | Z |
|---|---|
| 0 | 1 |
| 1 | 0 |

b. Inverter



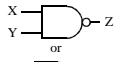
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

c. AND Gate



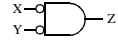
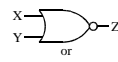
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

d. OR Gate



| X | Y | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

e. NAND Gate



| X | Y | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

f. NOR Gate



| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

g. Exclusive OR Gate



| X | Y | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

h. Equivalence Gate

■ Gates with more than 3 inputs:

- ◆ **AND gates:** The output is **1** if and only if ... ?
- ◆ **OR gates:** The output is **1** if and only if ... ?
- ◆ **NAND gates:** The output is **0** if and only if ... ?
- ◆ **NOR gates:** The output is **0** if and only if ... ?
- ◆ **EXCLUSIVE OR gates:** The output is **1** if and only if ... ?
- ◆ **EQUIVALENCE gates:** The output is **1** if and only if ... ?

□ Logical Expressions

- We can also represent the behavior of gates with a logical expressions constructed from variables and logical operations symbols.
- The following table gives the most common ones.

| Connective | Example | Meaning |
|--------------|----------------------|--|
| NEGATION | $C = A'$ | C is 1 iff A is 0. |
| AND | $C = A \cdot B$ | C is 1 iff A is 1 and B is 1. |
| OR | $C = A + B$ | C is 1 iff A is 1 or B is 1. |
| EXCLUSIVE OR | $C = A \oplus B$ | C is 1 iff A or B is 1, both not both. |
| NAND | $C = A \uparrow B$ | C is 1 iff It is not the case that A and B are both 1. |
| NOR | $C = A \downarrow B$ | C is 1 iff It is not the case that either A or B is 1. |
| EQUIVALENCE | $C = A \equiv B$ | C is 1 iff Both A and B are 1 or both A and B are 0. |

- Comments on the logical symbols
 - ◆ The NAND and NOR symbols are not very useful.
 - ◆ There are several different symbols that have been used for the logical connectives.

AND: \cdot , $\&$, \wedge

OR: $+$, $|$, \vee

NOT: $'$, $\bar{}$, \sim , \neg

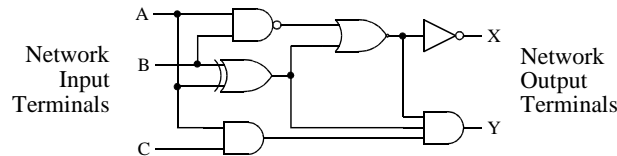
□ Exercise:

Determine how many different two-input gates there can be?
How many three-input gates?

| N | 2^{2^n} |
|---|-----------------------|
| 2 | 16 |
| 3 | 256 |
| 4 | 65,536 |
| 5 | 4,294,967,396 |
| 6 | 1.84×10^{19} |

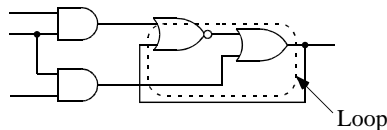
Gate Networks

- A *gate network* is a finite collection of interconnected gates, network input terminals, and network output terminals with the following restrictions:
 - No gate output terminal or network input terminal is connected to another gate output terminal or network input terminal.
 - Every network output terminal or gate input terminal is wired (via one or more wires) to a constant value, a network input terminal, or a gate output terminal.
- Example



Types of networks

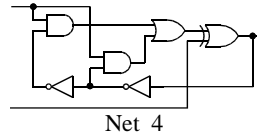
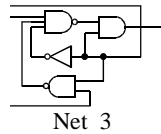
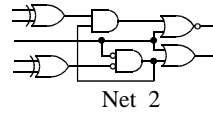
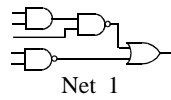
- A *combinational gate network* is one in which the values of the signals present on its input terminals uniquely determine the signal values at its output terminals.
- A gate network that is not combinational is called a *sequential gate network*.
- A *loop* in a gate network is a path that starts at a gate terminal, passes along wires and through gates, does not pass any wire or gate more than once, and terminates back at the starting gate terminal.



- ◆ Networks without loops are combinational.
 - We call a gate network without loops a *logic network*, since we can describe its behavior with a logical expression.
- ◆ Sequential networks have loops.
- ◆ Combinational networks may have loops.

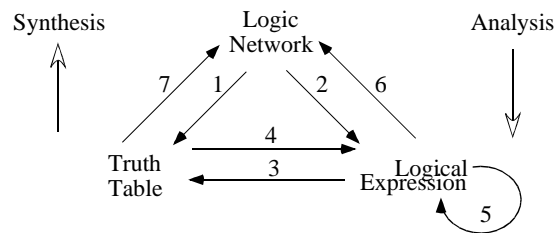
□ Exercise:

Which of the following networks are combinational and which are sequential?



Analysis & Synthesis of Logic Networks

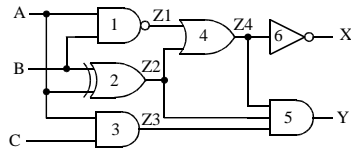
□ Overview



1. For a given logic network, find a truth table that describes its behavior.
2. For a given logic network, find a set of logical expressions that describes its behavior.
3. Transform a logical expression into the equivalent truth table representation.
4. Transform a truth table into an equivalent logical expression representation.
5. Transform a logical expression into an equivalent (and possibly simpler) logical expression.
6. Design a logic network to have the behavior specified by a given set of logical expressions.
7. Design a logic network to have the behavior specified by a given truth table.

□ Analysis of Logic Networks

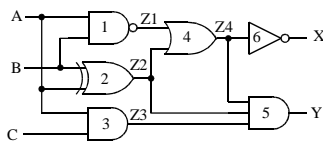
■ Logic Network



■ Truth Table Behavioral Description

| A | B | C | Z1 | Z2 | Z3 | Z4 | X | Y |
|---|---|---|----|----|----|----|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

■ Logical Expression Behavioral Description



$$X = Z4'$$

$$X = (Z1 + Z2)'$$

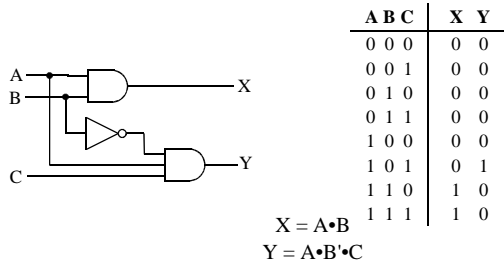
$$X = ((A \cdot B)' + (A \oplus B))'$$

$$Y = Z2 \cdot Z3 \cdot Z4$$

$$Y = (A \oplus B) \cdot (A \cdot C) \cdot (Z1 + Z2)$$

$$Y = (A \oplus B) \cdot (A \cdot C) \cdot ((A \cdot B)' + (A \oplus B))$$

■ Example



- The networks in these two examples (slides 2.19 and 2.20) are equivalent because they have the same truth table. The logical expressions for X and Y are also equivalent, but very different structurally.
- Graph levelization for analysis

■ Notation: We can represent a truth table by simply listing the indices of the rows that have value 1 or listing those that have value 0.

◆ Example

| A | B | C | X | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

We will revisit the Σ and Π notations when we look at sum-of-product and product-of-sum representations

- List of 1's:

$$X = \Sigma_{A,B,C} (6, 7) \quad Y = \Sigma_{A,B,C} (5)$$

- List of 0's:

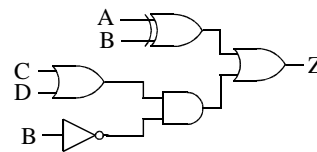
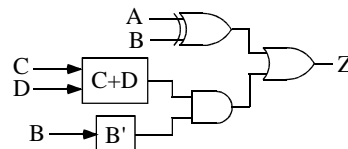
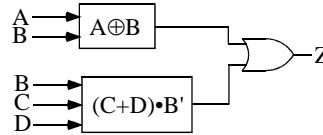
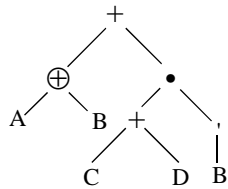
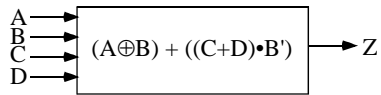
$$X = \Pi_{A,B,C} (0, 1, 2, 3, 4, 5) \quad Y = \Pi_{A,B,C} (0, 1, 2, 3, 4, 6, 7)$$

- ◆ The value of this notation is that it is a more compact way of specifying a logical function than writing the truth table. It is useful for specifying a function to be designed (e.g., in homework problems).

□ Synthesis of Logic Networks

■ Example

$$Z = (A \oplus B) + ((C + D) \cdot B')$$

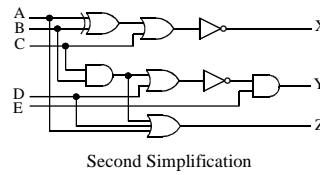
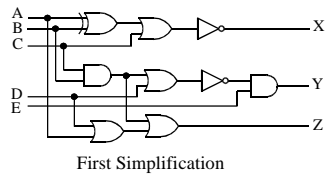
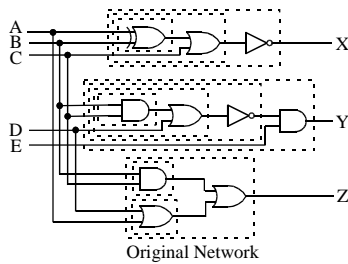


■ Example

$$X = ((A \oplus B) + C)'$$

$$Y = (B \cdot C + D)' \cdot E$$

$$Z = (A + D) + (B \cdot C)$$

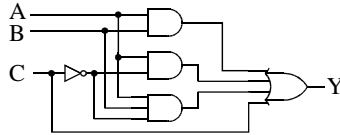


■ Two level gate networks and logical expressions

◆ Sum-of-Products (SOP) logical expressions

$$A \cdot B + A \cdot C' + B \cdot C' \cdot A + C$$

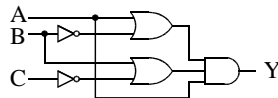
◆ Two level AND-OR Networks



◆ Product-of-Sums (POS) logical expressions

$$(A+B') \cdot (B+C') \cdot A$$

◆ Two level OR-AND Networks

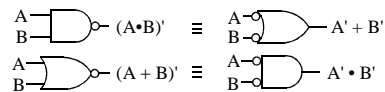


NAND/NOR Networks

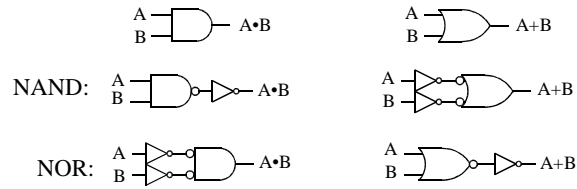
□ This topic deals with designing combinational logic networks using only NAND or NOR gates.

■ Approach: First design the network with AND, OR and NOT gates. Then Transform it to an equivalent network of NAND or NOR gates.

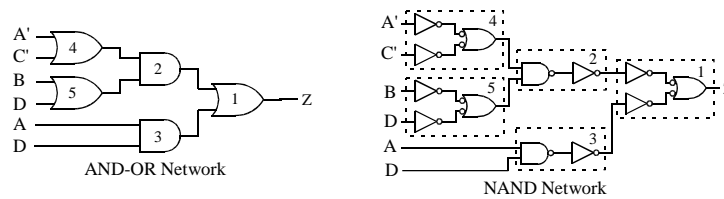
◆ The following alternative NAND and NOR symbols are useful for this.



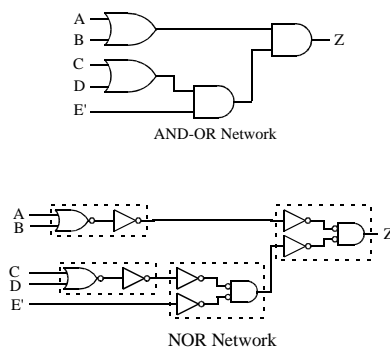
- One approach is to replace AND and OR gates by the following equivalent NAND or NOR circuits:



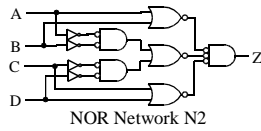
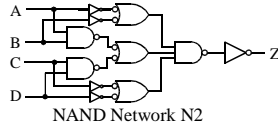
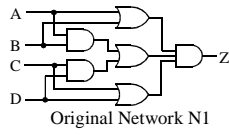
- NAND Network Example:



- NOR Network Example

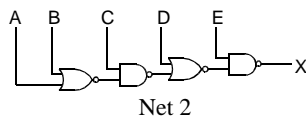
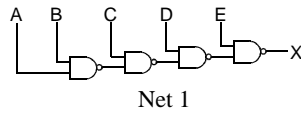


□ Example of Transformation to a NAND and a NOR Network:



□ Exercise:

- Derive logical expressions, using only the logical operations \cdot , $+$ and $'$, for the following two networks.



An Introduction to Verilog

- Verilog is a programming language that was developed for describing the behavior and structure of digital systems. Languages such as this are called Hardware Description Languages or HDL's.
 - Verilog is an IEEE standard and widely used today.
 - Verilog is probably most useful for describing systems at a high-level of abstraction, before the implementation details of the systems are developed.
 - ◆ While HDL's have been around for 30 years, it has only been with the advent of large integrated circuits that they have become popular with digital designers.
 - Why?

□ Verilog Uses

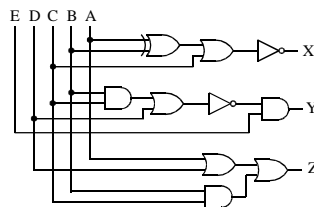
- Verilog can be viewed as a simulation modeling language
 - ◆ Enables digital designers to simulate and verify the behavior of their systems before they design a detailed implementation.
 - Verification of functional behavior
 - Timing analysis
 - ◆ Verilog can also be used to specify test patterns for testing Verilog simulation models. Verilog programs used for generating test patterns are called *test benches* or *test fixtures*.
- Verilog is also used to specify the input to synthesis tools that produce implementations automatically.
 - ◆ When this is possible, the designer need not use the classical design techniques.

□ Verilog Modules

- An elementary Verilog program is called a *module*.
 - ◆ A module corresponds to a digital circuit.
 - ◆ Modules have input and output *ports* that correspond to the input and output terminals of a digital circuit.
 - ◆ The ports and variables used to represent internal signals are declared at the beginning of the program.
 - ◆ Modules have other statements used to define how it transforms the input signals to output signals.
- Verilog modules can be used to specify the structure or the behavior of a digital circuit.
 - ◆ Structural modules consist of a list of component modules (defined elsewhere) and a list of wires used to interconnect the modules.
 - ◆ Behavior modules specify the output signals as functions of the input signals. They need not give any indication of the structure of the circuit.

□ Example of a behavioral Verilog module

- Circuit:



- Verilog module:

```
module circuit1b (A, B, C, D, E, X, Y, Z);  
  input A, B, C, D, E;  
  output X, Y, Z;  
  assign X = ~(A ^ B) | C;  
  assign Y = ~(B & C) | D) & E;  
  assign Z = A | D | (B & C);  
endmodule
```

■ Comments on the module

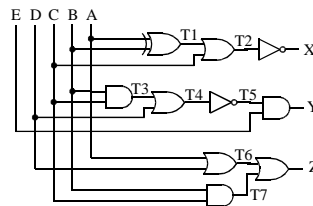
- ◆ Case matters in Verilog statements.
 - The symbol string “Out”, “out”, and “OUT” represent three different variables.
- ◆ Individual signals (e.g., A, B, ... in the previous example) can take any of the following four values:
 - 0 = logic value 0
 - 1 = logic value 1
 - z = tri-state (high impedance)
 - x = unknown value
- ◆ The unknown value is used by simulators to indicate that they do not know how to determine a signals value (e.g., the user has not specified a value for an input signal). The tri-state value means that no signal is assigned to the variable.
- ◆ The unknown value x can also be used to specify a don't care condition to the synthesis tools.
- ◆ Note that ports must be listed in the module statement (first line) and their direction (input or output) declared in the following statements.

- ◆ The three **assign** statements are independent and can execute in any order or concurrently.
 - The right side of an **assign** statement is evaluated and its resulting value assigned to the signal on the left side whenever one of the signals used in the right side changes value.
 - ◆ This type of assignment is also called a *continuous assignment*.
 - This method of interpreting the execution of **assign** statements is quite different from that use in conventional programming languages in the following ways:
 - ◆ Two or more assignments can execute simultaneously. This is necessary to represent the timing characteristics of hardware systems.
 - ◆ An assignment executes whenever it is ready (i.e., has new data for the variables on its right side)
 - ◆ There is no concept of "locus of control" or "program counter" that determines the next instruction to execute. Therefore, the order the assignments are written does not matter
 - This method of assignment statement execution is sometimes called *non-procedural or data-driven* execution, where conventional programming languages are said to be *procedural*.

□ Behavioral vs. Structural Verilog Descriptions

- The previous example is a behavioral description because it specifies the logical values of the circuit's outputs as logical equations with no reference to how the gates in a possible implementation might be interconnected.
- It is also possible to specify a structural description in Verilog that specifies explicitly how a set of smaller components (e.g., gates) are interconnected to form a larger system as shown on the following slide.
 - ◆ Note that the internal connections from gate outputs to gate inputs are declared to be of type **wire**.
 - These wire declarations could be omitted as long as the component modules are simple gates. Verilog will assume that any gate output signal that is not declared is of type **wire**.
 - ◆ Each gate is specified by its name (e.g., **and**, **or**, ...) and a list of ports or wires connected to its terminals. All gates have a single output and it is always listed first in this list.

```
module circuit1s (A, B, C, D, E, X, Y, Z);  
  input A, B, C, D, E;  
  output X, Y, Z;  
  wire T1, T2, T3, T4, T5, T6, T7;  
  
  xor(T1, A, B);  
  or(T2, T1, C);  
  not(X, T2);  
  and(T3, B, C);  
  or(T4, T3, D);  
  not(T5, T4);  
  and(Y, T5, E);  
  or(T6, A, D);  
  and(T7, B, C);  
  or(Z, T6, T7);  
  
endmodule
```



□ Always Blocks

- A procedural block is a construct that contains statements that are executed procedurally (i.e., in the order they are written).

```
always @(sensitivity_list)
begin
    procedural statements
end
```

- The sensitivity list is a list of signals separated by **or**.
 - ◆ When any one of the signals in the sensitivity list changes value, the always block wakes up, executes its procedural statements, and then goes back to sleep.
- The always block acts like a generalized assign statement where the action that takes place can be specified by sequential code.

■ Example

```
module always_example(x, y, z);
    input x, y;
    output z;
    reg z, s;

    always @(x or y)
    begin
        s = x ^ y;
        z = x & s;
    end
endmodule
```

- Variables declared as type **reg** hold their value until they are assigned a new value. It is said that the assigned value is *registered* in the variable
- Whenever x or y changes value, the always block is executed as follows:
 - ◆ First, the statement $s = x \wedge y$ executes and registers a new value in s
 - ◆ Next, the statement $z = x \& s$ executes using the new value of s that it received when the first statement was executed.
 - ◆ Then the block stops executing and waits for either x or y to change again.
- Variables on the left side of a procedural statement must be declared as type **reg**.
- The main advantage of using always blocks to represent combinational circuits is that you can use control statements such as “if then else” as illustrated by the following example

```

module mux1 (x1, x2, s, y);
  input x1, x2, s;
  output y;
  reg y;

  always @(x1 or x2 or s)
    if (s == 1)
      y = x1;
    else
      y = x2;
endmodule

```

```

module mux2 (x1, x2, s, y);
  input x1, x2, s;
  output y;

  assign y = (s & x1) | (~s & x2);
endmodule

```

- ◆ Modules mux1 and mux2 do exactly the same thing.
- ◆ The control statements used in always blocks will be covered in more detail in later chapters.

- Exercise: Explain how the behaviors of the following two modules differ.

```

module always_example(x, y, z, f);
  input x, y, z;
  output f;
  reg f, s;

  always @(x or y or z)
  begin
    s = x ^ y;
    f = z & s;
  end

endmodule

```

```

module assign_example (x, y, z, f);
  input x, y, z;
  output f;
  wire s;

  assign s = x ^ y;
  assign f = z & s;

endmodule

```

- How would the behavior of each of these modules change if the order of the two assignment statements is reversed?

- Conditions for combinational behavior of always blocks

- ◆ The following conditions are necessary for an always block to represent combinational logic (as opposed to sequential logic)
 - All reg, wire and input signals that appear on the right side of an assignment statement within the always block must appear in the sensitivity list
 - ◆ We call a sensitivity list that satisfies this condition a *complete sensitivity list*.
 - All signals in the sensitivity list must appear without edge specifiers
 - ◆ Edge specifiers indicate that a signal is asserted by a change in value as opposed to its level (e.g., a rising edge or falling edge). They are introduced and utilized in later chapters on sequential circuits.
 - All output signals must be assigned a value every time the always block executes
- ◆ These conditions guarantee that the input signals uniquely determine the output signals, which is the very definition of a combinational circuit.
- ◆ Some Verilog synthesizers (Including the one in the Xilinx ISE) assume that if you omit one or more of the signals in the sensitivity list, that you really meant to put them in, so it does it for you. However, it will issue a warning message in this case.
- ◆ A sensitivity list of the form @(*) is shorthand for a complete list.
 - It is recommended that you use this notation for combinational always blocks.

■ Example (Incomplete sensitivity list)

```
module example1(a, b, c, f);  
  input a, b, c;  
  output f;  
  reg f;  
  
  always @(a, b)  
    if (a==1)  
      f = b;  
    else  
      f = c;  
endmodule
```

◆ The input c is missing from the sensitivity list

- According to the semantics of Verilog, this represents a sequential circuit. Why?
- Xilinx ISE will assume you meant to put c in the sensitivity list and add it for you, so a circuit it synthesizes from this module will be combinational.
- It is better to always use complete sensitivity lists. The easiest way to do this is to replace @(a,b) by @(*).

■ Example (Missing output signal in a control path)

- ◆ A control path is a sequence of operations (possibly null) that can be performed by an always block. It corresponds to a path from the input node to the output node of a flow chart of the block.

```
module example2(a, b, c, f);  
  input a, b, c;  
  output f;  
  reg f;  
  
  always @(*)  
    if (a==1)  
      f = b&c;  
endmodule
```

- ◆ If a is 0 and b or c changes value, the always block executes but does not execute an assignment to f, so f is not uniquely determined by the static values of a, b, and c.
- ◆ Xilinx ISE will synthesize this module as a sequential circuit.