HARDWARE DESCRIPTION LANGUAGE

HARDWARE DESCRIPTION LANGUAGE

- Manual methods for designing logic circuits are feasible only when the circuit is small.

- For anything else (i.e., a practical circuit), designers use computer-based design tools. Coupled with the correct-by-construction methodology, computer-based design tools leverage the creativity and the effort of a designer and reduce the risk of producing a flawed design.

- Prototype integrated circuits are too expensive and time consuming to build, so all modern design tools rely on a hardware description language to describe, design, and test a circuit in software before it is ever manufactured.

HARDWARE DESCRIPTION LANGUAGE

- A *hardware description language (HDL) is a computer-based language that describes* the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits.

- It can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit.

- For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

HARDWARE DESCRIPTION LANGUAGE

- As a *documentation language, an HDL is used to represent and document digital* systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers.

- The language content can be stored, retrieved, edited, and transmitted easily and processed by computer software in an efficient manner.

- HDLs are used in several major steps in the design flow of an integrated circuit: design entry, functional simulation or verification, logic synthesis, timing verification, and fault simulation.

Design entry

- *Design entry creates an HDL-based description of the functionality that is to be* implemented in hardware.

- Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a netlist of interconnected gates, or an abstract behavioral model.

- The HDL model may also represent a partition of a larger circuit into smaller interconnected and interacting functional units.

Logic simulation

- ***Logic simulation*** *displays the behavior of a digital system through the use of a computer.*
- A simulator interprets the HDL description and either produces readable output, such as a time-ordered sequence of input and output signal values, or displays waveforms of the signals.
- The simulation of a circuit predicts how the hardware will behave before it is actually fabricated. Simulation detects functional errors in a design without having to physically create and operate the circuit.
- Errors that are detected during a simulation can be corrected by modifying the appropriate HDL statements. The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a *test bench.*
- *Thus, to simulate a digital system, the design is first described in* an HDL and then verified by simulating the design and checking it with a test bench, which is also written in the HDL. An alternative and more complex approach relies on formal mathematical methods to prove that a circuit is functionally correct. We will focus exclusively on simulation.

Logic Synthesis

- *Logic synthesis is the process of deriving a list of physical components and their* interconnections (called a *netlist) from the model of a digital system described in an* HDL.

- The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list. Logic synthesis is similar to compiling a program in a conventional high-level language.

- The difference is that, instead of producing an object code, logic synthesis produces a database describing the elements and structure of a circuit. The database specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL.

- Logic synthesis is based on formal exact procedures that implement digital circuits and addresses that part of a digital design which can be automated with computer software. The design of today's large, complex circuits is made possible by logic synthesis software.

Timing Verification

- Timing verification confirms that the fabricated, integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit.

- Propagation delays ultimately limit the speed at which a circuit can operate. Timing verification checks each signal path to verify that it is not compromised by propagation delay.

- This step is done after logic synthesis specifies the actual devices that will compose a circuit and before the circuit is released for production.

Fault Simulation

- In VLSI circuit design, ***fault simulation*** *compares the behavior of an ideal circuit with* the behavior of a circuit that contains a process-induced flaw. Dust and other particulates in the atmosphere of the clean room can cause a circuit to be fabricated with a fault.

- A circuit with a fault will not exhibit the same functionality as a fault-free circuit. Fault simulation is used to identify input stimuli that can be used to reveal the difference between the faulty circuit and the fault-free circuit.

- These test patterns will be used to test fabricated devices to ensure that only good devices are shipped to the customer. Test generation and fault simulation may occur at different steps in the design process, but they are always done before production in order to avoid the disaster of producing a circuit whose internal logic cannot be tested.

VHDL and Verilog

- Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are two standard HDLs that are supported by the IEEE: **VHDL and Verilog.** VHDL is a Department of Defense–mandated language. (The *V in VHDL* stands for the first letter in VHSIC, an acronym for very high-speed integrated circuit.)

- Verilog began as a proprietary HDL of Cadence Design Systems, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard.

VHDL and Verilog

- Our emphasis will be on the modeling, verification, and synthesis (both manual and automated) of Verilog models of circuits having specified behavior.

- The Verilog HDL was initially approved as a standard HDL in 1995; revised and enhanced versions of the language were approved in 2001 and 2005.

- We will address only those features of Verilog, including the latest standard, that support our discussion of HDL-based design methodology for integrated circuits.

Module Declaration

- In particular, a Verilog model is composed of text using keywords, of which there are about 100. Keywords are predefined lowercase identifiers that define the language constructs.

- Examples of keywords are **module, endmodule, input, output, wire, and, or, and not.**

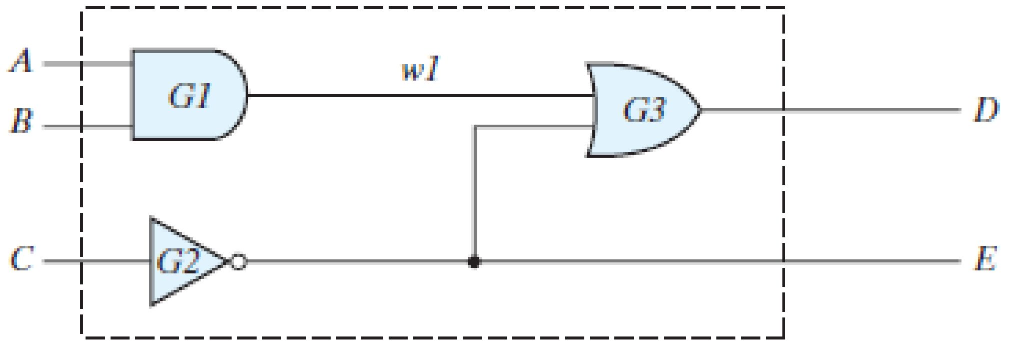
- Any text between two forward slashes (*//) and the end of the line is interpreted as a comment and will have no effect* on a simulation using the model. Multiline comments begin with / * and terminate with * /.

- Blank spaces are ignored, but they may not appear within the text of a keyword, a user-specified identifier, an operator, or the representation of a number.

- Verilog is case sensitive, which means that uppercase and lowercase letters are distinguishable (e.g., **not is not the same as NOT).**

- **The term *module refers to the text*** enclosed by the keyword pair **module . . . endmodule. A module is the fundamental** descriptive unit in the Verilog language. It is declared by the keyword **module and** must always be terminated by the keyword **endmodule.**

Circuit to demonstrate an HDL



```
// Verilog model of circuit of Figure. IEEE 1364–1995 Syntax
module Simple_Circuit (A, B, C, D, E);
output      D, E;
input       A, B, C;
wire        w1;
and         G1 (w1, A, B); // Optional gate instance name
not         G2 (E, C);
or          G3 (D, w1, E);
endmodule
```

Circuit to demonstrate an HDL

- The *port list of a module is the interface between the module and its environment.*

- In this example, the ports are the inputs and outputs of the circuit. the keywords **input and output specify which of the ports are** inputs and which are outputs.

- Internal connections are declared as wires. The circuit in this example has one internal connection, at terminal *w1 , and is declared with the keyword* **wire.**

Circuit to demonstrate an HDL

- The structure of the circuit is specified by a list of (predefined) *primitive* gates, each identified by a descriptive keyword (**and, not, or**). **The elements of the list** are referred to as *instantiations of a gate, each of which is referred to as a gate instance* .

- Each *gate instantiation consists of an optional name (such as G1, G2 , etc.) followed by* the gate output and inputs separated by commas and enclosed within parentheses.

- The output of a primitive gate is always listed first, followed by the inputs.

Circuit to demonstrate an HDL

- The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.
- The module description ends with the keyword **endmodule.**
- **Each statement must be terminated with a semicolon, but** there is no semicolon after **endmodule.**

Circuit to demonstrate an HDL

- It is important to understand the distinction between the terms *declaration and instantiation*. A Verilog module is declared. Its declaration specifies the input–output behavior of the hardware that it represents.

- Predefined primitives are not declared, because their definition is specified by the language and is not subject to change by the user.

- Primitives are used (i.e., instantiated), just as gates are used to populate a printed circuit board.

Boolean Expressions

- Boolean equations describing combinational logic are specified in Verilog with a continuous assignment statement consisting of the keyword **assign followed by a Boolean** expression.

Combinational Logic Modeled with Boolean Equations

$E = A + BC + B'D$

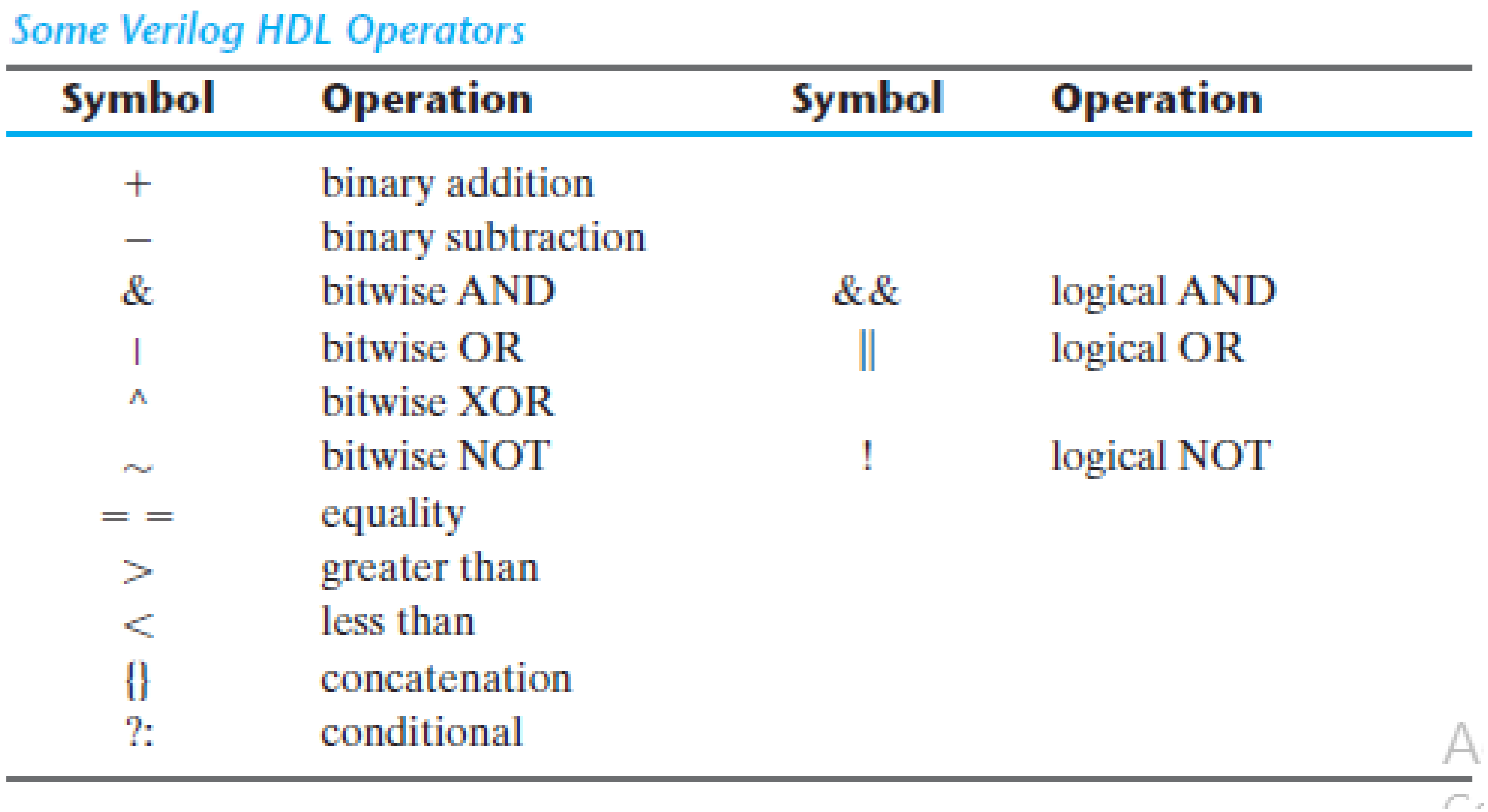
$F = B'C + BC'D'$

```
// Verilog model: Circuit with Boolean expressions
module Circuit_Boolean_CA (E, F, A, B, C, D);
output E, F;
input A, B, C, D;
assign E = A || (B && C) || ((!B) && D);
assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

Dataflow Modeling

- Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result.
- Verilog HDL provides about 30 different operators.
- It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each.

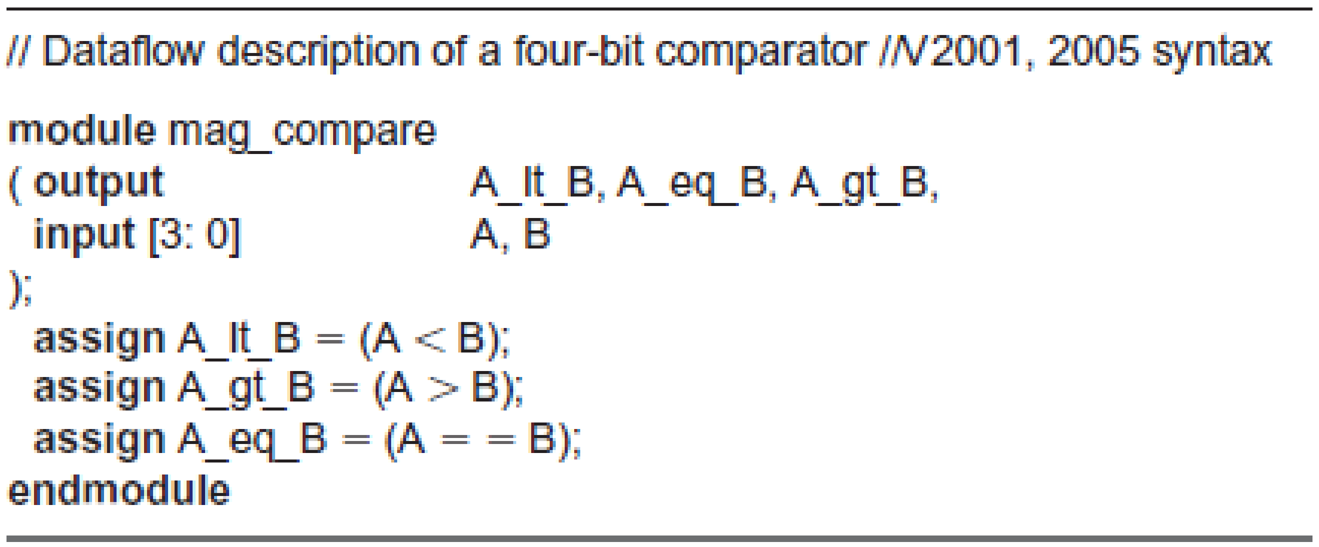
HDL Operators



Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Dataflow: Four-Bit Comparator



```
// Dataflow description of a four-bit comparator //V2001, 2005 syntax

module mag_compare
( output              A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]        A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A = = B);
endmodule
```

Behavioral Modeling

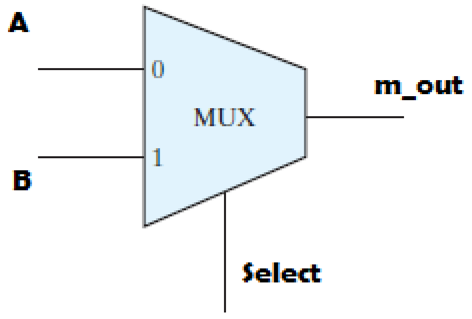
- Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.

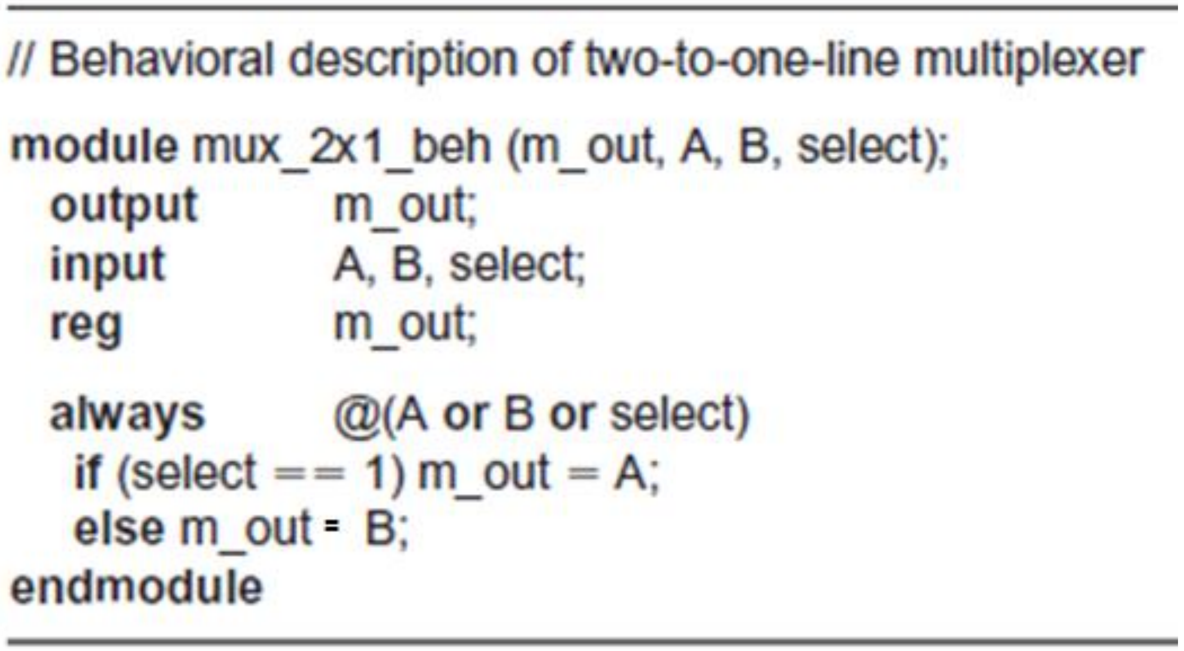
- Behavioral descriptions use the keyword **always , followed by an optional event control** expression and a list of procedural assignment statements.

- The event control expression specifies when the statements will execute. The target output of a procedural assignment statement must be of the **reg data type.**

- **Contrary to the wire data type,** whereby the target output of an assignment may be continuously updated, a **reg data** type retains its value until a new value is assigned.

Behavioral: Two-to-One Line Multiplexer





```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
  output        m_out;
  input         A, B, select;
  reg           m_out;

  always        @(A or B or select)
   if (select == 1) m_out = A;
   else m_out =  B;
endmodule
```

Gate-Level Modeling

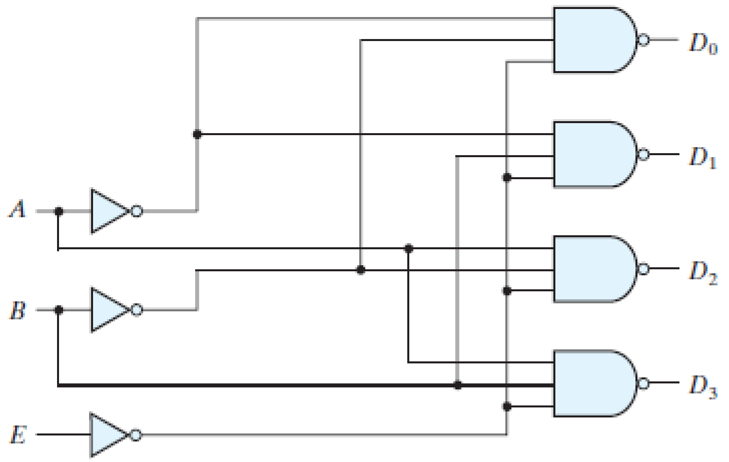
- In this type of representation, a circuit is specified by its logic gates and their interconnections. Gatelevel modeling provides a textual description of a schematic diagram.

- The Verilog HDL includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type.

- They are all declared with the lowercase keywords **and, nand, or, nor, xor, xnor, not, and buf .** Primitives such as **and** are ***n -input primitives.***

- ***They can have any number of scalar inputs*** (e.g., a three-input **and primitive). The buf and not** primitives are ***n -output primitives.*** A single input can drive multiple output lines distinguished by their identifiers.

Gate-Level Modeling



```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

module decoder_2x4_gates (D, A, B, enable);
 output        [0: 3]          D;
 input                         A, B;
 input                         enable;
 wire                          A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

Combinational Logic Modeled with Boolean Equations F=(A’C)AB+BCD’

- module Circuit_A (A, B, C, D, F);
- input A, B, C, D;
- output F;
- wire w, x, y, z, a, d;
- or (x, B, C, d);
- and (y, a ,C);
- and (w, z ,B);
- and (z, y, A);
- or (F, x, w);
- not (a, A);
- not (d, D);
- endmodule

- module Circuit_C (y1, y2, y3, a, b);
- output y1, y2, y3;
- input a, b;
- assign y1 = a || b;
- and (y2, a, b);
- assign y3 = a && b;
- endmodule