Introduction to Verilog

- **Introduction to CAD Tools**
- To design a logic circuit, a number of CAD tools are needed.

- Usually packaged together into a CAD system, which typically includes tools for the following tasks: design entry, synthesis and optimization, simulation, and physical design.

Design Entry

- The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure.

- The first stage of this process involves entering into the CAD system a **description of the circuit** being designed. This stage is called design entry.

- For design entry we are **writing source code** in a **hardware description language**.

Hardware Description Languages

- A hardware description language (HDL) is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer.

- Two HDLs are IEEE standards: Verilog HDL and VHDL.

  Verilog:

  Weakly typed.
  Allows more flexibility but can cause errors if not careful.
  Example: You can assign an integer to a signal without explicit conversion.

  VHDL:

  Strongly typed.
  Forces strict data type definitions and conversions.
  Good for safety-critical systems (aerospace, defense).

Why to use Verilog

- Supported by most companies that offer digital hardware technology.

- Verilog provides **design portability**. A circuit specified in Verilog can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the Verilog specification.

- Both small and large logic circuit designs can be efficiently represented in Verilog code.

Representation of Digital Circuits in Verilog

i) Structural representation- A larger circuit is defined by writing code that connects simple circuit elements together.

Using Verilog constructs, describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the *structural* representation of logic circuits.

ii) **Behavioral representation-** Describing a circuit by using logic expressions and programming constructs that define the behavior of the circuit but not its actual structure in terms of gates.

Using logic expressions and Verilog programming constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates, describe a circuit more abstractly.

Structural Specification of Logic Circuits

- Verilog includes a set of gate-level primitives that correspond to commonly-used logic gates.
- A gate is represented by indicating its functional name, output, and inputs.
- For example,
- A two-input AND gate, with inputs x1 and x2 and output y, is denoted as

  and (y, x1, x2);
- A four-input OR gate is specified as

  or (y, x1, x2, x3, x4);
- The NOT gate given by **not** (y, x); implements y = x'.

The available Verilog gate-level primitives are

Name	Description	Usage
and	$f = (a \cdot b \cdot \ldots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \ldots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)

Verilog Module

- It is a circuit or sub circuit described with Verilog code.
- The module has a name, ***module_name***, which can be any valid identifier, followed by a list of ports.
- The term port refers to an input or output connection in an electrical circuit. The ports can be of type **input**, **output**, or **inout** (bidirectional), and can be either scalar or vector

The General Form of a Module

module module name [(port name{, port name})];
[**parameter** declarations]
[**input** declarations]
[**output** declarations]
[**inout** declarations]
endmodule

Identifier Names

- Identifiers are the names of variables and other elements in Verilog code.
- The rules for specifying identifiers are simple: any letter or digit may be used, as well as the _ underscore and $ characters.
- An identifier must not begin with a digit and it should not be a Verilog keyword.

- Examples of legal identifiers are f, x1, x, y, and Byte.
- Some examples of illegal names are 1x, +y, x*y, and 258
- Verilog is case sensitive, hence k is not the same as K, and BYTE is not the same as Byte.

Verilog Operators

- Verilog operators are useful for synthesizing logic circuits.

Documentation in Verilog Code

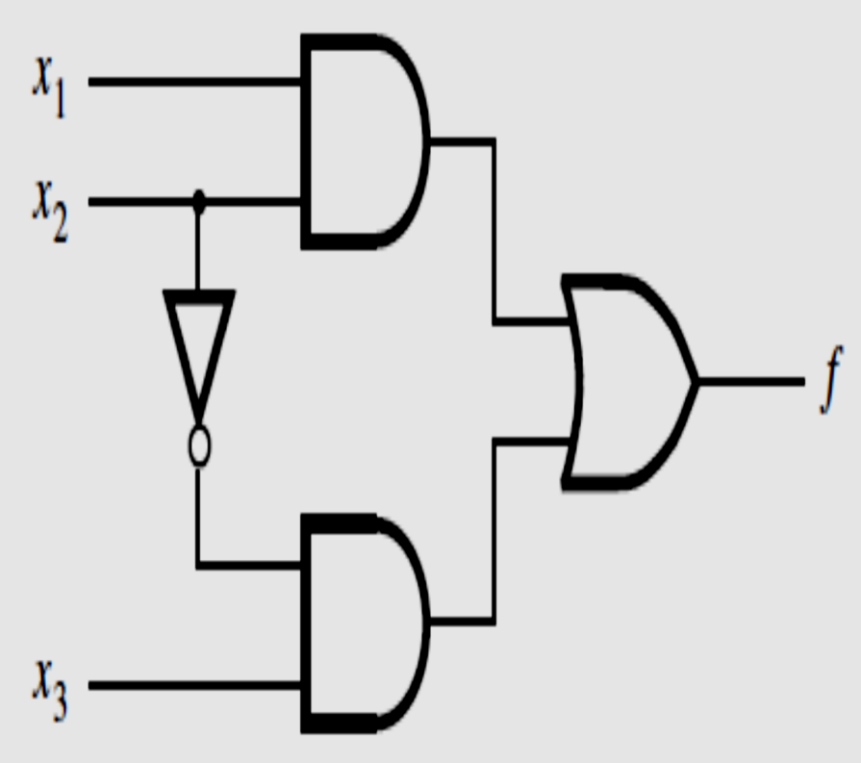
- Documentation can be included in Verilog code by writing a comment.
- A short comment begins with the double slash, //, and continues to the end of the line.
- A long comment can span multiple lines and is contained inside the delimiters /* and */.

White Space

- White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler.
- Multiple statements can be written on a single line.
- Placing each statement on a separate line and using indentation within blocks of code, such as an **if-else** statement are good ways to increase the readability of code.

Gate level primitives(structural)

- Simple logic function



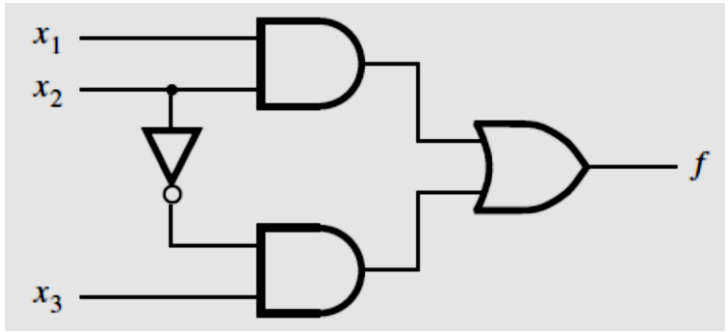
- Using gate level primitives

```
module example2(x1,x2,x3,f);
        input x1,x2,x3;
        output f;
        and (g,x1,x2);
        not (k,x2);
        and (h,k,x3);
        or (f,g,h);
endmodule
```

Behavioral Specification of Logic Circuits

- Gate level primitives can be tedious when large circuits have to be designed.
- To use more abstract expressions and programming constructs to describe the behavior of a digital circuit.
- To define the circuit using logic expressions. The AND and OR operations are indicated by the “&” and “|” signs, respectively.
- The **assign** keyword provides a continuous assignment for the output signal.
- Whenever any signal on the right-hand side changes its state, the value of output will be re-evaluated.

Behavioral(Continuous assignment) Specification of Logic Circuits




```
module example2 (x1, x2, x3, f);
        input x1, x2, x3;
        output f;
    assign f = (x1 & x2) | ( ~x2 & x3);
endmodule
```

NO SEMICOLON

Running a sample Verilog code

1. Create a directory with section followed by roll number (to be unique); e.g. CCE B21
2. Open any text editor in Ubuntu (say, gedit) and type the source code.

```
module example2(x1, x2, x3, f);
input x1, x2, x3;
output f;
and (g, x1, x2);
not (k, x2);
and (h, k, x3);
or (f, g, h);
endmodule
```

3. Save the source code with file name same as module name but with '**.v**' extension in the required directory.

4. Type the following testbench code.

```
`timescale 1ns/1ns
`include "example2.v"    //Name of the Verilog file

module example2_tb();
reg x1, x2, x3;          //Input
wire f;                  //Output

example2 ex2(x1, x2, x3, f);   //Instantiation of the module
initial
begin

        $dumpfile("example2_tb.vcd");
        $dumpvars(0, example2_tb);
```

$dumpvars(0, example2_tb); means: record all signals in module example2_tb and its submodules (like example2).

$dumpfile = name of file to save waveforms.

$dumpvars = what signals to save.
.vcd = Value Change Dump format.

The first argument (0) = dump all hierarchy levels under example2_tb.
If you put 1, it would only dump top-level signals, not inside submodules.

```
        x1=1'b0; x2=1'b0; x3=1'b0;
        #20;
        x1=1'b0; x2=1'b0; x3=1'b1;
        #20;
        x1=1'b0; x2=1'b1; x3=1'b0;
        #20;
        x1=1'b0; x2=1'b1; x3=1'b1;
        #20;
        x1=1'b1; x2=1'b0; x3=1'b0;
        #20;
        x1=1'b1; x2=1'b0; x3=1'b1;
        #20;
        x1=1'b1; x2=1'b1; x3=1'b0;
        #20;
        x1=1'b1; x2=1'b1; x3=1'b1;
        #20;

        $display("Test complete");
end

endmodule
```

5. Save the source code with file name same as module name but with '.v' extension in the same directory.

6. Go to the terminal. Type the following commands for compilation.

- **iverilog -o example2_tb.vvp example2_tb.v**

On successful execution, file example2_tb.vvp is created.

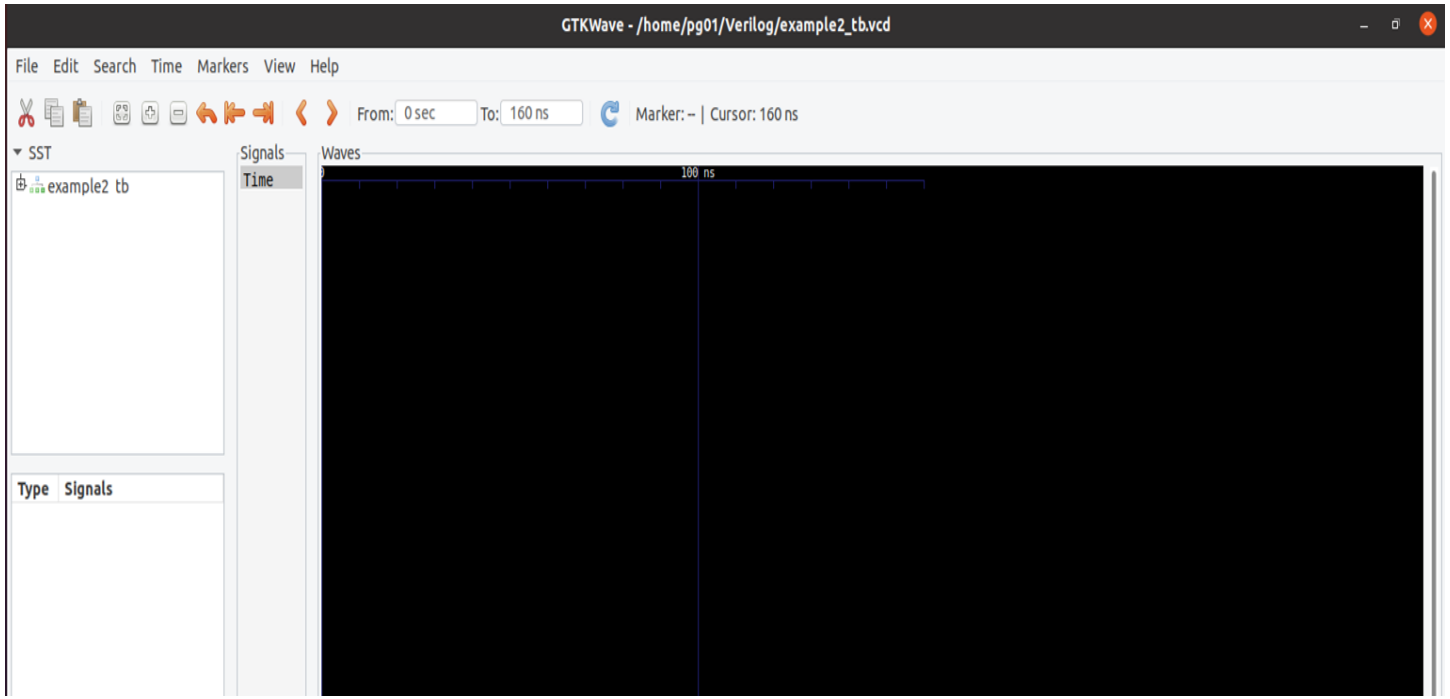
- **vvp example2_tb.vvp**

On successful execution, file example2_tb.vcd is created and the following messages are displayed in the terminal.

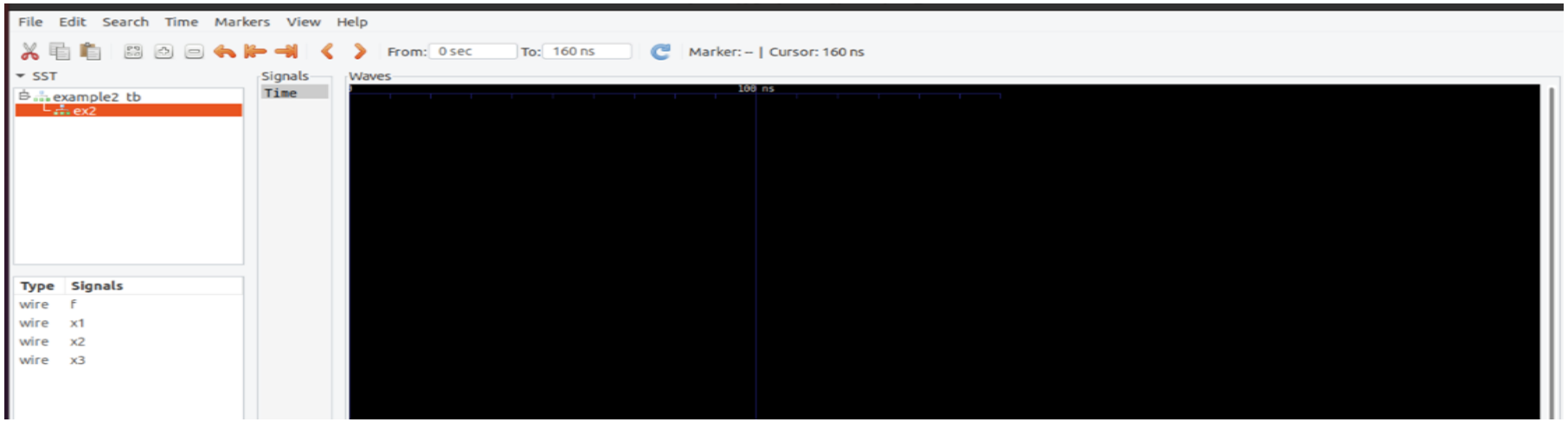
VCD info: dumpfile example2_tb.vcd opened for output.

Test complete

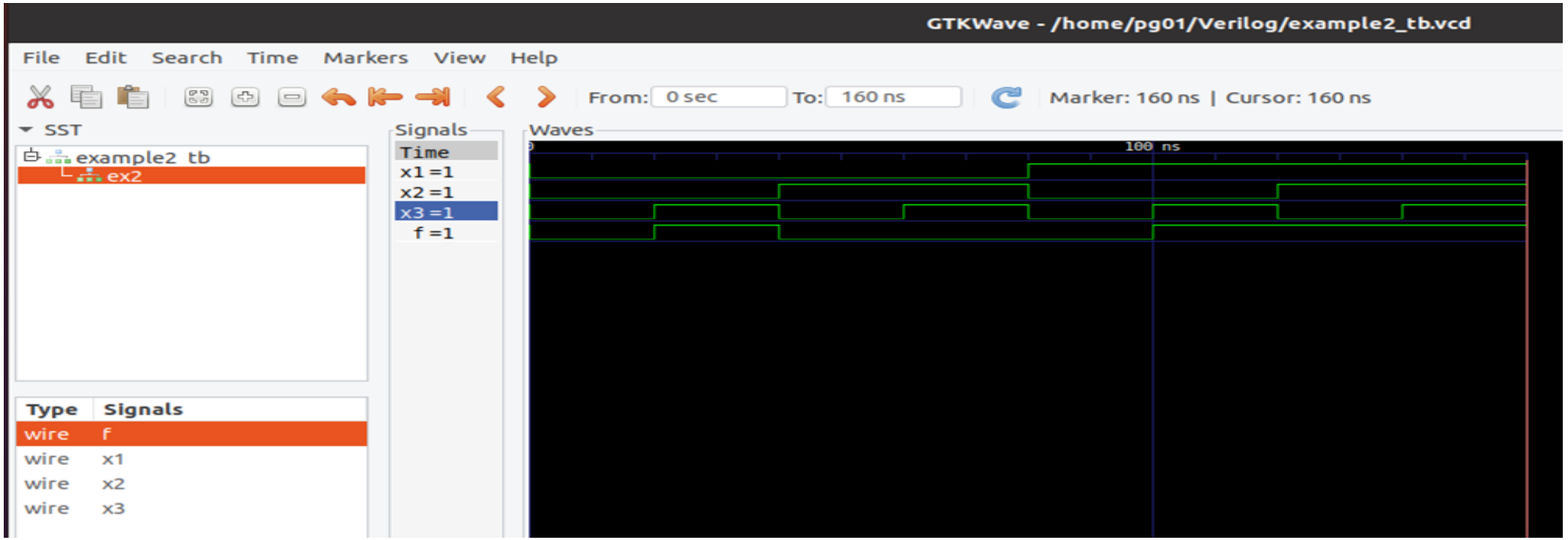
7. Open gtkwave by double clicking the example2_tb.vcd file available in your directory or type gtkwave in the terminal and open the example2_tb.vcd file in gtkwave. A window appears as shown below.


GTKWave - /home/pg01/Verilog/example2_tb.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 160 ns
Marker: -- | Cursor: 160 ns
SST
example2_tb
Type Signals
Signals
Time
Waves
100 ns

8. Expand example2_tb that appears in the left pane. The screen appears as shown below. It can be observed that the input and output variables appear in the bottom left pane.



9. Drag and drop the variables that appears in the Signals tab of the bottom left pane into Signals column that appears in the middle of the window. The output appears as shown below. Cross-verify the output with the truth table.



Example2:

It defines a circuit that has four input signals, $x1$, $x2$, $x3$, and $x4$, and three output signals, f, g, and h. It implements the

logic functions

$g = x1x3 + x2x4$

$h = (x1 + x3')(x2' + x4)$

$f = g + h$

```
module example2 (x1, x2, x3, x4, f, g, h);
input x1, x2, x3, x4;
output f, g, h;
and (z1, x1, x3);
and (z2, x2, x4);
or (g, z1, z2);
or (z3, x1, ~x3);
or (z4, ~x2, x4)
and (h, z3, z4);
or (f, g, h);
endmodule
```

Instead of using explicit NOT gates to define *x*2 and *x*3, we have used the Verilog operator "~" (tilde character on the keyboard) to denote complementation. Thus, *x*2 is indicated as ~x2 in the code.

The AND and OR operations are indicated by the "&" and "|" Verilog operators, respectively. The *assign* keyword provides a *continuous assignment* for the signal *f*.
Behavioral code for example 2 is as follows:

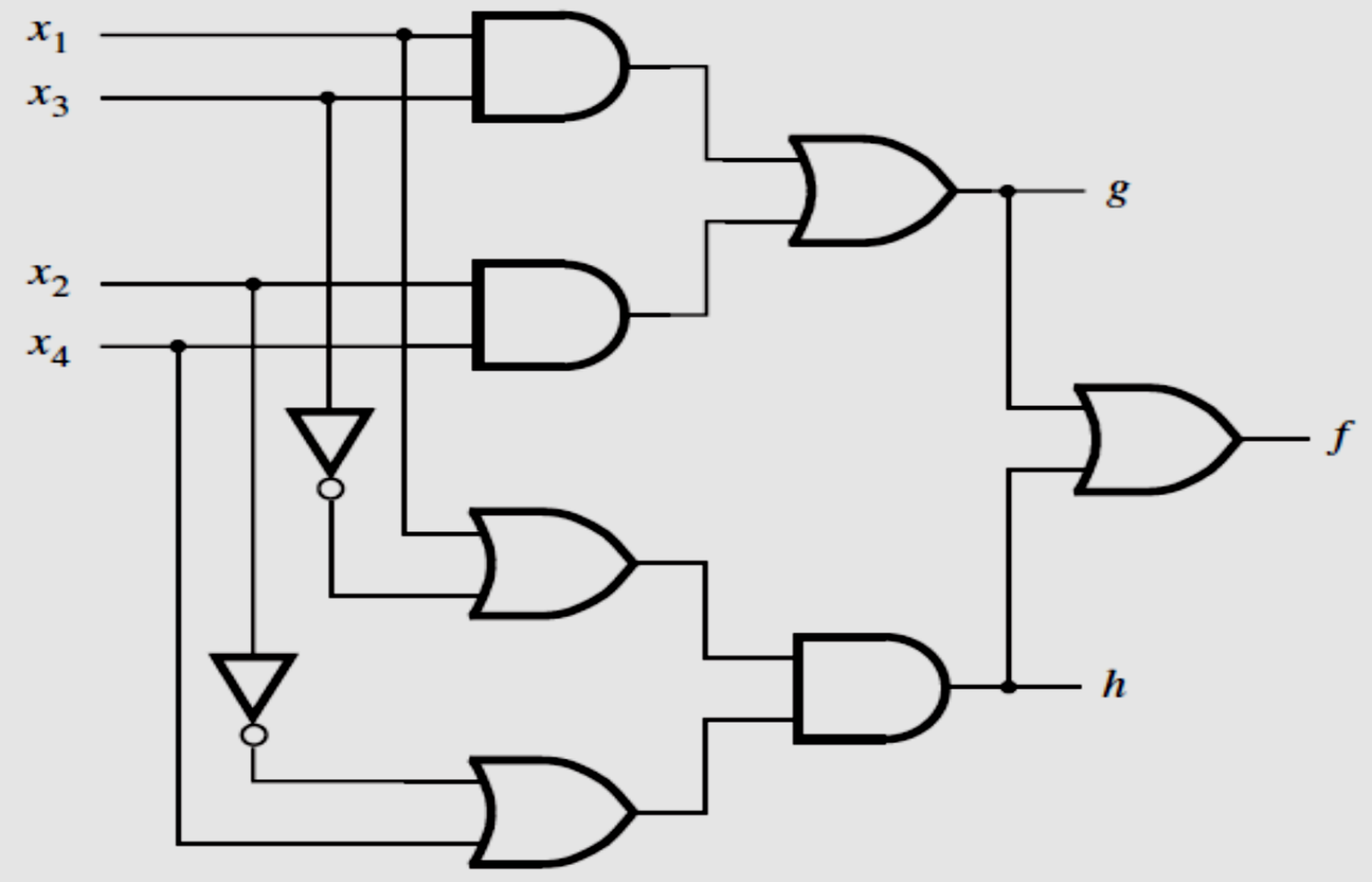
```
module example2 (x1, x2, x3, x4, f, g, h);
input x1, x2, x3, x4;
output f, g, h;
assign g = (x1 & x3) | (x2 & x4);
assign h = (x1 | x3) & ( x2 | x4);
assign f = g | h;
endmodule
```

Operator type	Operator Symbols	Operation Performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
Arithmetic	+	Addition	2
	-	Subtraction	2
	-	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Lesser than	2
	>=	Greater than or equal to	2
	<=	Lesser than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

Write the Verilog code to implement the circuit in the following figure

1.Using gate level primitives

2.Using continuous assignment



Write the Verilog code to implement the circuit in the following figure

1.Using gate level primitives

2.Using continuous assignment