LECTURE

Hardware Description Language

REFs

Introduction to Microelectronics to Nanoelectronics

Design and Technology

Manoj Kumar Majumde

Electronic Circuits

Fundamentals and applications

Mike Tooley

LECTURE OUTLINE

Introduction

- ✓ To describe the logical functions of Modern VLSI-based systems can be extremely complex and very demanding task as conventional techniques (*such as truth tables, logic diagrams and Boolean algebra*) quickly become cumbersome and error prone.
- ✓ In the design process there's also a need to take into account internal propagation delays and other limitations of real-world logic devices.
- ✓ Hardware description language (HDL) seeks to remedy this problem by providing a formal and rigorous description of a digital logic circuit.
- ✓ HDL provides a means of describing the operation and logical function of a circuit as well as its architecture and behavior.

- ✓ By forming an HDL description of a circuit we are able to exhaustively test and verify the circuit without having to manufacture any physical hardware.
- ✓ Not only does HDL use standard text-based constructs to describe the operation of logic elements, but it can also take into account the timing (*and more particularly the time delays*) associated with real logic devices.

VERILOG HDL

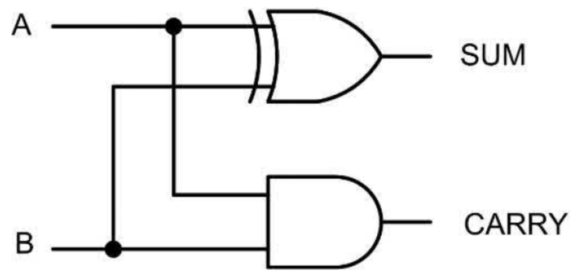
- ✓ Verilog HDL is a hardware description language that is utilized in designing of digital systems. The designing may occur at different levels starting from the switch to the top behavioural level
- ✓ It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip–flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology.

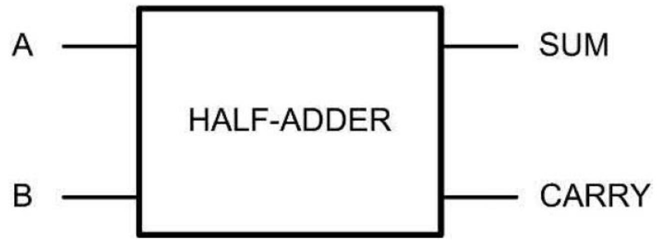
ENTITIES AND ENTITY DECLARATIONS

- ✓ In a complex logic system where many components and logical sub-systems can be present it is essential for each individual component to be seen (*and can thus be interacted with*) by all of the other components present.
- ✓ Thus a primary requirement of any logic system description language is a construct that provides us with a rigorous definitior of the component's interface.
- ✓ In VHDL this is referred to as an **entity**. A VHDL entity must be declared before it can be used. This declaration must begin with a unique name for the entity and then must specify the entity's input and output ports.

Example

Consider the basic half-adder below





It can be represented as

```
ENTITY half_adder IS
        PORT( x, y: IN bit;
              sum, carry: OUT bit);
END half-adder;
```

The code fragment provides us with the following information:

1. *The name of the entity is* ***half_adder****.*

2. *The entity has two inputs named* ***x*** *and* ***y****.*

3. *The entity has two outputs named* ***sum*** *and* ***carry****.*

4. *The inputs and outputs are all bit (binary digit)* ***type***

PORT STATEMENT

- ✓ The PORT statement specifies the interface to the entity in terms o both the direction of data flow and the type of signal.
- ✓ Notice also that, at this stage, we have not provided any clues as to the behaviour or the entity. We need to do this separately with some further code statements.
- ✓ Note that VHDL allows for both concurrent and sequential signal assignment.

COMPILER DIRECTIVES

A compiler directive provides certain information that remains through the compilation process until other compiler directives are specifed. The syntax of compiler directives starts with a backquote (`) character. The main different compiler directives in Verilog are listed here:

defne, `undef

`timescale

`resetall

`include

```
`define array_size 40
 ...............
reg [array_size-1:0] y;
        ................
`undef array_size
```

OPERATORS

There are several different operators available in Verilog as listed below:

Arithmetic operator

Equality operators

Relational operators

Logical operators

Bitwise operators

Arithmetic Operator

The different arithmetic operators are listed below:

• +(addition) • -(Subtraction) • *(Multiplication) /(divide)

• %(modulus) • **(power)

MODULE AND TEST BENCH DEFINITIONS

Verilog consists of a module and a test bench. A module is used to describe the design and a test bench is used to provide the stimulus to test the design described in the module.

MODULE

A module block starts with the keyword 'module' and ends with the keyword 'endmodule'.

In the module definition, all the input and output terminals of modules are declared.

```
module module_name (Output1, Output2, …….Outp tn,
    ….inputn);
output variable_1;
input variable_1, variable_2,……………variable_n;
begin
    statement_1;

    statement_2;
    ……………
    …………….
    Statement_n;
end
endmodule
```

Example:

Write a module for full adder design.

```
Module fulladder1 (Sout, Cout, a, b, cin);
Output Sout, Cout;
Input a, b, cin;
begin
    fulladder design description
end
endmodule
```

TEST BENCH

A test bench contains a Verilog program that is used to generate test patterns that are used to test the main module. Test bench not only generates the test patterns but also applies those to design. From this, the module performance can be analyzed and tested. The syntax of the test bench is as follows:

module *test_bench*;
reg variable_1, variable_2, …….variable_n;
wire variable_1, variable_2, …….variable_n;
instantiation of module;
test patterns;
endmodule

Example:

Write a test bench for full adder design.

```
module fulladder_test;
reg in1, in2, in3;
wire out1, out2;
fulladder1 g1 (out1, out2, in1, in2, in3);
initial
begin
in1=0;
in2=0;
in3=0;
#50 in1=0;
#50 in2=1;
#50 in3=0;
end
endmodule
```

GATE-LEVEL MODELING

Single And Multiple Input Gates

The available single and multi-input gates in Verilog are as listed.

These built-in logic gate primitives will have only one output and multiple inputs and their syntax for declaration is shown below

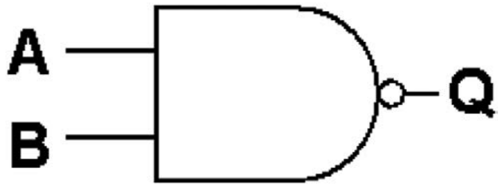
List of Built-in Gates and Significance

S.No	Keyword	Significance
1	not	Signifies inverter
2	and	Signifies and gate
3	nand	Signifies nand gate
4	Or	Signifies or gate
5	nor	Signifies nor gate
6	xor	Signifies xor gate
7	xnor	Signifies xnor gate

multiple-input gate keyword *instance_name (output, Input1, Input2, ...Inputn);*

Example:

Declare the two-input NAND gate in Verilog,



```
nand g1(Q, A, B)
```

to declare NAND gate, use nand, a keyword with an instance name *g1*. The frst terminal declared as output *Q* is followed by the 2-inputs *A* and *B*.

GATE DELAYS

Gate delay is the propagation delay from the input to the output of any gate. This can be specified with the gate instantiation, and the syntax gate instantiation with gate delay is shown below:

gate_type *gate_delay instance_name* (terminal list);

If the gate delay is not mentioned, it means the propagation delay of the gate is zero. The gate delay consists of three values rise delay, fall delay, and turn_off delay. All the time delay units are specified by using `timescale directive in Verilog HDL.

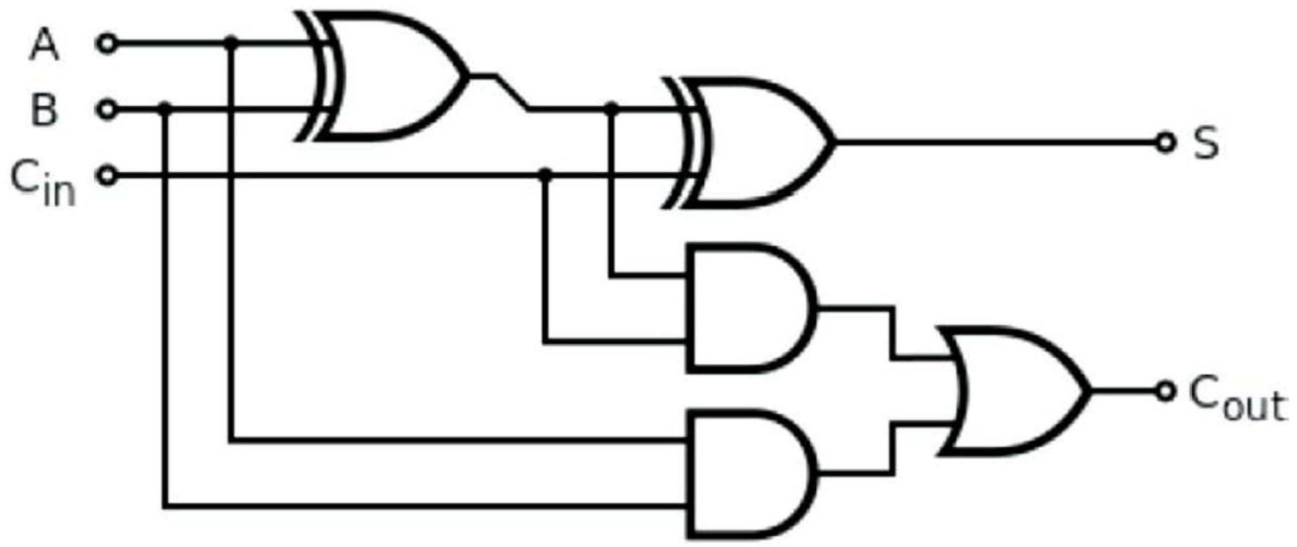
Examples

```
nor #7 (y, a, b);

nand #(8, 10) (y, a, b);
```

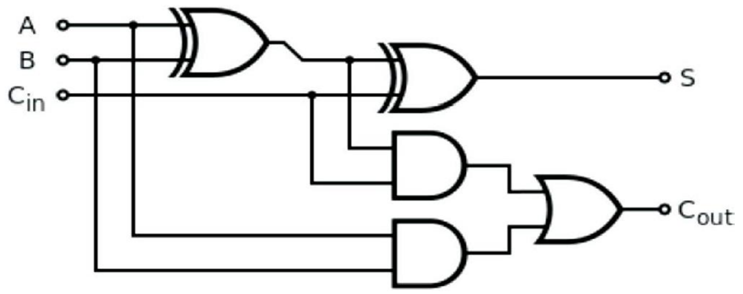
EXAMPLE

Consider a gate-level design of full adder, as shown below,



Write a Verilog program for it

Solution



```
module fulladder1 (S, Cout, A, B, Cin);
output S, Cout;
input A, B, Cin;
wire t1, t2, t3;
xor g1 (t1, A, B);
xor g2 (S, Cin, t1);
and g3 (t2, t1, Cin);
and g4 (t3, A, B);
or g5 (Cout, t2, t3);
endmodule
```

BEHAVIORAL MODELING

Other than the names that we chose for our entities, the declaration provides no clues as to the underlying logic function. In order to do this, we need some additional code to specify the **behaviour** of our entity.

initial and always statements.

These statements in the Verilog program execute concurrently, which means there is no order for execution of these statements. Initial statement generally executes only once in the program. The syntax of the initial block is as follows

```
initial
    begin
procedural_statements;
    end
```

Execution of procedural statements inside the initial happens only once. The initial block is generally used to initialize the variables since it is executed only once.

Example:

Write an initial statement for the SR flip-flop.

```
initial
begin
Q= 0;//the intial state is assumed to be 00 for SR flip-flop and declared
Qb=1;
end
```

ALWAYS STATEMENT

Unlike the initial statement, the always statement executes repeatedly. The always statement is very important in behavioral-level modeling and is used for higher-level modeling in Verilog.

The syntax of the always statement is as follows:

```
always @ (senstivitity list)
begin
procedural_assignment;
end
```

Always block starts with the always statement, and it always executes if any variable in the sensitivity list changes. Here, procedural assignment statements execute repeatedly based on the sensitivity list and execute sequentially.

Example:

Write always statement for clock generation with a period of 10 time units

```
always
begin
#10 clk = ~clk;
end
```

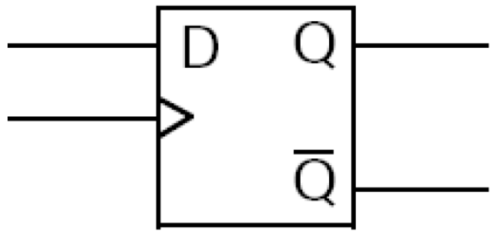
Example:

Write always statement with an event control

```
always @ (enable)
begin
a = ~b;
end
```

Example:

Write a Verilog program for D flip-flop



Consider a D fip-fop with an input D, output q, and clock signal clk, and assume that fip-fop is negative edge triggered.

```
module Dflipflop(clk, D, q, qb);
input clk, D;
output reg q, qb;
always @(negedge clk)
begin
D=q;
qb=~q;
end
endmodule
```

In the behavioral level, Verilog modeling all the outputs are declared as registers. Here, as D fip-fop is negative edge triggered, the always block is repeated depending on the negative edge event of the clock signal and it is represented as ‘negedge clk’.

FURTHER READING

Procedural Assignments

Conditional statements

Tasks and functions