Introduction to Verilog

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- Hardware Description Language
- Logic Simulation versus Synthesis
- Verilog Module
- Gate-Level Description and Gate Delays
- Module Instantiation
- Continuous Assignment
- Writing a Simple Test Bench

Hardware Description Language

- Describes the hardware of digital systems in a textual form
- Describes the hardware structures and behavior
- Can represent logic diagrams, expressions, and complex circuits
- NOT a software programming language
- Two standard hardware description languages (HDLs)

1. **Verilog** (will be studied in this course)
2. **VHDL** (harder to learn than Verilog)

Verilog = "Verifying Logic"

- Invented as a simulation language in 1984 by Phil Moorby
- Opened to public in 1990 by Cadence Design Systems
- Became an IEEE standard in 1995 (Verilog-95)
- Revised and upgraded in 2001 (Verilog-2001)
- Revised also in 2005 (Verilog-2005)
- Verilog allows designers to describe hardware at different levels
  - Can describe anything from a single gate to a full computer system
- Verilog is supported by the majority of electronic design tools
- Verilog can be used for logic simulation and synthesis

Logic Simulation

- Logic simulator interprets the Verilog (HDL) description
- Produces **timing diagrams**
- Predicts how the hardware will behave before it is fabricated
- Simulation allows the detection of functional errors in a design
  - Without having to physically implement the circuit
- Errors detected during the simulation can be corrected
  - By modifying the appropriate statements in the Verilog description
- Simulating and verifying a design requires a **test bench**
- The test bench is also written in Verilog

Logic Synthesis

- Logic synthesis is similar to translating a program
- However, the output of logic synthesis is a digital circuit
- A digital circuit modeled in Verilog can be translated into a list of components and their interconnections, called **netlist**
- Synthesis can be used to fabricate an integrated circuit
- Synthesis can also target a Field Programmable Gate Array
  - An FPGA chip can be configured to implement a digital circuit
  - The digital circuit can also be modified by reconfiguring the FPGA
- Logic simulation and synthesis are automated
  - Using special software, called Electronic Design Automation (EDA) tools

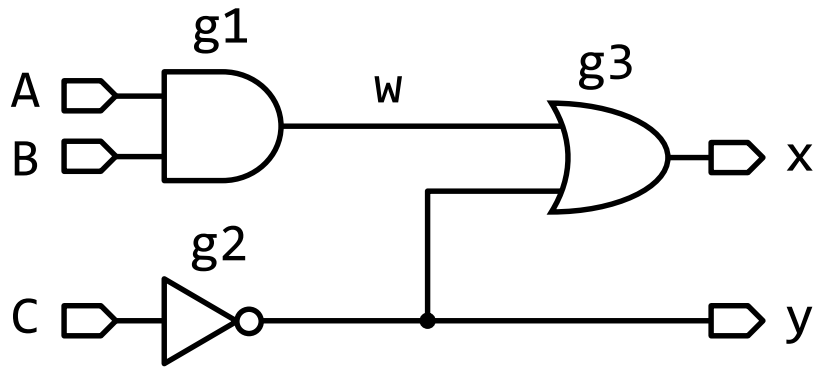
Verilog Module

- A digital circuit is described in Verilog as a set of modules
- A module is the design entity in Verilog
- A module is declared using the **module** keyword
- A module is terminated using the **endmodule** keyword
- A module has a name and a list of **input** and **output** ports
- A module is described by a group of statements
- The statements can describe the module structure or behavior

Example of a Module in Verilog

```
// Description of a simple circuit
module simple_circuit(input A, B, C, output x, y);
  wire w;
  and g1(w, A, B);
  not g2(y, C);
  or  g3(x, w, y);
endmodule
```

Order is not important



The **input** keyword defines the input ports: A, B, C

The **output** keyword defines the output ports: x, y

The **wire** keyword defines an internal connection: w

The structure of `simple_circuit` is defined by three gates: **and**, **not**, **or**

Each gate has an optional name, followed by the gate output then inputs

Verilog Syntax

- **Keywords:** have special meaning in Verilog

  Many keywords: **`module`**, **`input`**, **`output`**, **`wire`**, **`and`**, **`or`**, etc.

  Keywords cannot be used as identifiers

- **Identifiers:** are user-defined names for modules, ports, etc.

  Verilog is **case-sensitive**: `A` and `a` are different names

- **Comments:** can be specified in two ways (similar to C)
  - Single-line comments begin with **`//`** and terminate at end of line
  - Multi-line comments are enclosed between **`/*`** and **`*/`**

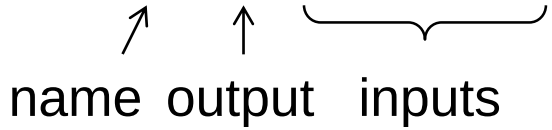
- **White space:** space, tab, newline can be used freely in Verilog

- **Operators:** operate on variables (similar to C: **`~ & | ^ + -`** etc.)

Basic Gates

- Basic gates: **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf**
- Verilog define these gates as **keywords**
- Each gate has an optional name
- Each gate has an output (listed first) and one or more inputs
- The **not** and **buf** gates can have only one input
- Examples:

```
and g1(x,a,b);      // 2-input and gate named g1
or  g2(y,a,b,c);    // 3-input or  gate named g2
nor g3(z,a,b,c,d);  // 4-input nor gate named g3
```



Modeling a Half Adder

A half adder adds two bits: **a** and **b**

Two output bits:

1. Carry bit: **cout = a · b**
2. Sum bit: **sum = a ⊕ b**

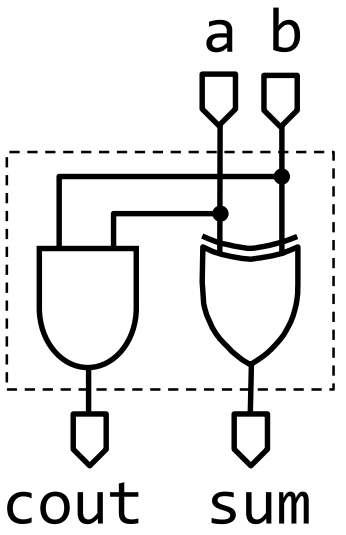
Truth Table

a	b	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

```
module Half_Adder(a, b, cout, sum);
   input a, b;
   output sum, cout;
   and (cout, a, b);
   xor (sum, a, b);
endmodule
```

Verilog-95 Syntax

Half_Adder



Full Adder

- Full adder adds 3 bits: **a**, **b**, and **c**
- Two output bits:
  1. Carry bit: **cout**
  2. Sum bit: **sum**
- Sum bit is 1 if the number of 1's in the input is odd (odd function)

  sum = (a ⊕ b) ⊕ c
- Carry bit is 1 if the number of 1's in the input is 2 or 3

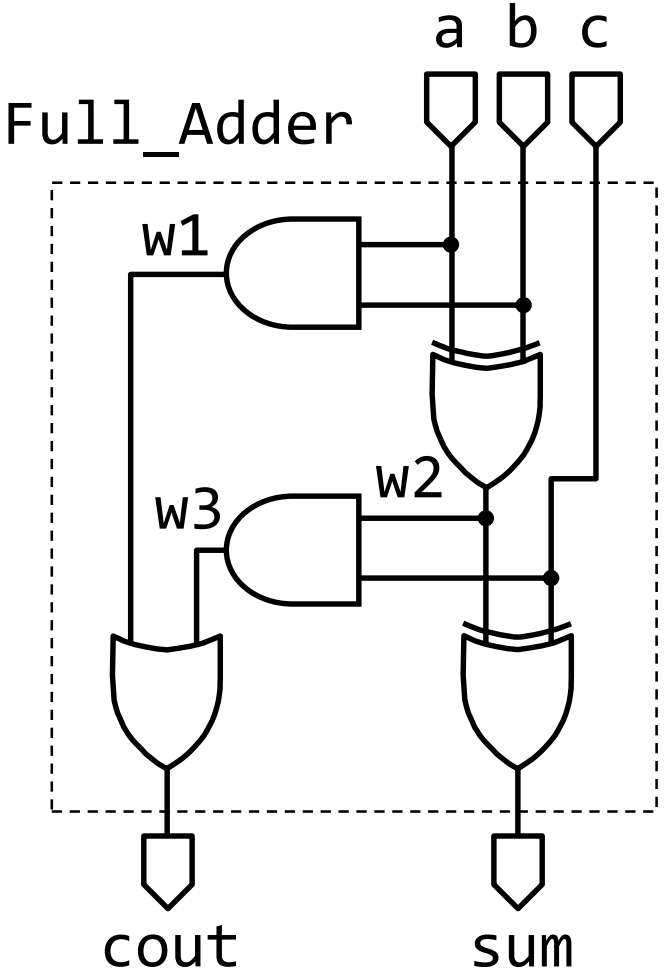
  cout = a·b + (a ⊕ b)·c

Truth Table

a	b	c	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder Module

```
module Full_Adder(input a, b, c, output cout, sum);

  wire w1, w2, w3;

  and (w1, a, b);

  xor (w2, a, b);

  and (w3, w2, c);

  xor (sum, w2, c);

  or  (cout, w1, w3)

endmodule
```



Gate Delays

- When simulating Verilog modules, it is sometime necessary to specify the delay of gates using the **#** symbol
- The **`timescale** directive specifies the time unit and precision

  timescale is also used as a simulator option

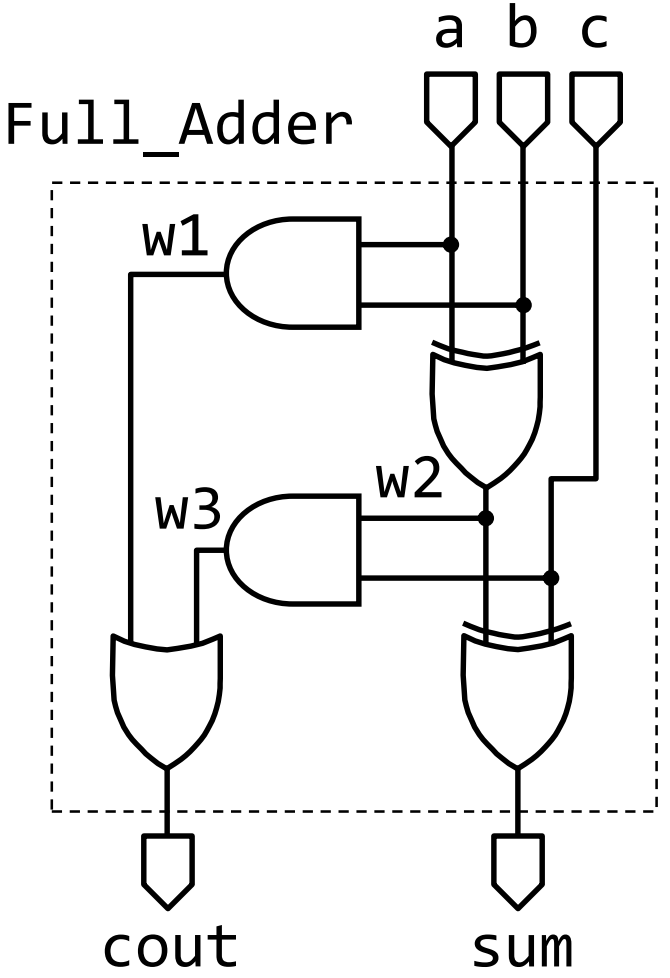
Time unit = 1ns = 10^{-9} sec

Precision = 100ps = 0.1ns

```
`timescale 1ns/100ps
module Half_Adder(input a, b, output cout, sum);
  and #2 (cout, a, b);      // gate delay = 2ns
  xor #3 (sum, a, b);       // gate delay = 3ns
endmodule
```

Full Adder Module with Gate Delay

```
module Full_Adder(input a, b, c, output cout, sum);

  wire w1, w2, w3;

  and #2 (w1, a, b);

  xor #3 (w2, a, b);

  and #2 (w3, w2, c);

  xor #3 (sum, w2, c);

  or  #2 (cout, w1, w3)

endmodule
```



Continuous Assignment

- The **assign** statement defines continuous assignment
- Syntax: **assign** *name* = *expression*;
- Assigns *expression* value to *name* (output port or wire)
- Examples:

```
assign x = a&b | c&~d;      // x = ab + cd'
assign y = (a|b) & ~c;      // y = (a+b)c'
assign z = ~(a|b|c);        // z = (a+b+c)'
assign sum = (a^b) ^ c;     // sum = (a ⊕ b) ⊕ c
```

- Verilog uses the bit operators: **~** (not), **&** (and), **|** (or), **^** (xor)
- Operator precedence: **(**parentheses**)**, **~** , **&** , **|** , **^**

Continuous Assignment with Delay

Syntax: **assign #*delay name* = *expression*;**

The optional **#*delay*** specifies the delay of the assignment

To have a delay similar to the gate implementation

```
module Full_Adder (input a, b, c, output cout, sum);
 assign #6 sum = (a^b)^c;              // delay = 6
 assign #7 cout = a&b | (a^b)&c;       // delay = 7
endmodule
```

The order of the **assign** statements does not matter

They are **sensitive** to inputs (a, b, c) that appear in the expressions

Any change in value of the input ports (a, b, c) will **re-evaluate** the outputs sum and cout of the **assign** statements

Test Bench

- In order to simulate a circuit, it is necessary to apply inputs to the circuit for the simulator to generate an output response
- A test bench is written to **verify the correctness** of a design
- A test bench is written as a **Verilog module with no ports**
- It **instantiates the module** that should be tested
- It **provides inputs** to the module that should be tested
- Test benches can be complex and lengthy, depending on the complexity of the design

Example of a Simple Test Bench

```
module Test_Full_Adder;   // No need for Ports
  reg  a, b, c;           // variable inputs
  wire sum, cout;         // wire outputs
  // Instantiate the module to be tested
  Full_Adder FA (a, b, c, cout, sum);
  initial begin           // initial block
    a=0; b=0; c=0;        // at t=0  time units
    #20 a=1; b=1;         // at t=20 time units
    #20 a=0; b=0; c=1;    // at t=40 time units
    #20 a=1; c=0;         // at t=60 time units
    #20 $finish;          // at t=80 finish simulation
  end                     // end of initial block
endmodule
```

Difference Between `wire` and `reg`

Verilog has two major data types

1. **Net data types:** are connections between parts of a design
2. **Variable data types:** can store data values

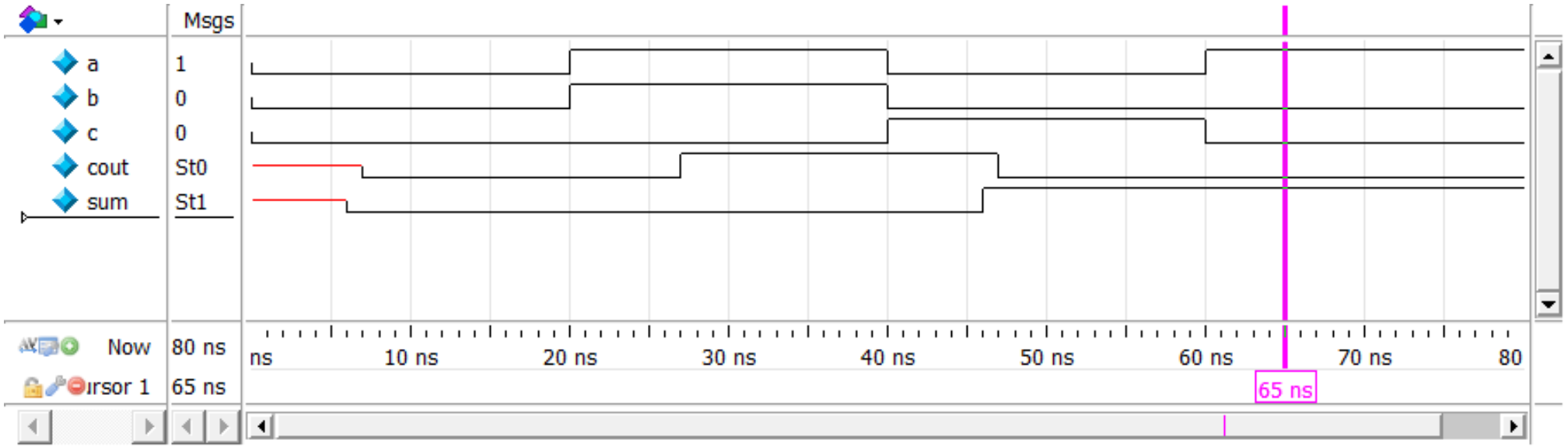
- The **`wire`** is a net data type
  - A wire cannot store a value
  - Its value is determined by its driver, such as a gate, a module output, or continuous assignment
- The **`reg`** is a variable data type
  - Can store a value from one assignment to the next
  - Used only in procedural blocks, such as the **`initial`** block

The initial Statement

- The **initial** statement is a procedural block of statements
- The body of the **initial** statement surrounded by **begin-end** is sequential, like a sequential block in a programming language
- Procedural assignments are used inside the **initial** block
- Procedural assignment statements are executed in sequence

  Syntax: **#***delay variable* = *expression*;
- Procedural assignment statements can be delayed
- The optional **#***delay* indicates that the *variable* (of **reg** type) should be updated after the time delay

Running the Simulator



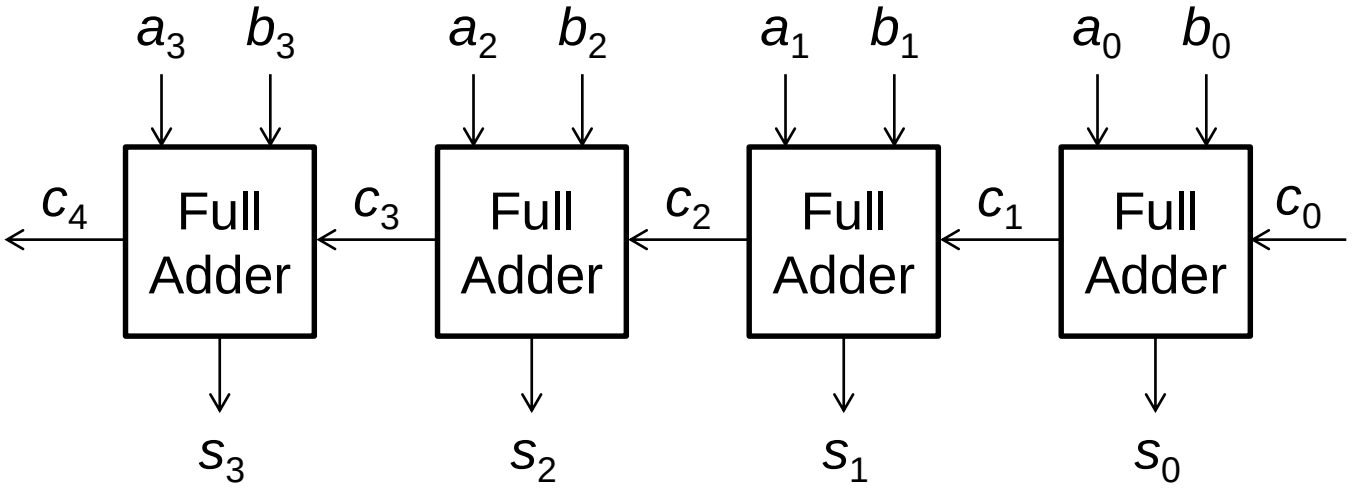
Examine the waveforms to verify the correctness of your design

At t = 0 ns, the values of **cout** and **sum** are unknown (shown in red)

The **cout** and **sum** signals are delayed by **7ns** and **6ns**, respectively

Modular Design: 4-bit Adder

- Uses **identical copies** of a full adder to build a large adder
- Simple to implement: the **cell** (iterative block) is a **full adder**
- Carry-out of cell *i* becomes carry-in to cell (*i* +1)
- Can be extended to add any number of bits



4-bit Adder using Module Instantiation

```
module Adder4 (input a0, a1, a2, a3, b0, b1, b2, b3, c0,
  output s0, s1, s2, s3, c4
);
  wire c1, c2, c3;  // Internal wires for the carries
 // Instantiate Four Full Adders: FA0, FA1, FA2, FA3
 // The ports are matched by position
  Full_Adder FA0 (a0, b0, c0, c1, s0);
  Full_Adder FA1 (a1, b1, c1, c2, s1);
  Full_Adder FA2 (a2, b2, c2, c3, s2);
  Full_Adder FA3 (a3, b3, c3, c4, s3);
 // Can also match the ports by name
 // Full Adder FA0 (.a(a0), .b(b0), .c(c0), .cout(c1), .sum(s0));
endmodule
```

Module Instantiation

- Module declarations are like **templates**
- Module instantiation is like creating an **object**
- Modules are instantiated inside other modules at different levels
- The top-level module does not require instantiation
- Module instantiation defines the **structure** of a digital design
- It produces **module instances** at different levels
- The ports of a module instance must match those declared
- The matching of the ports can be done **by name** or **by position**

Writing a Test Bench for the 4-bit Adder

```
module Adder4_TestBench;                    // No Ports
  reg  a0, a1, a2, a3;                      // variable inputs
  reg  b0, b1, b2, b3, cin;                 // variable inputs
  wire s0, s1, s2, s3, cout;                // wire outputs
  // Instantiate the module to be tested
  Adder4 Add4 (a0,a1,a2,a3, b0,b1,b2,b3, cin, s0,s1,s2,s3, cout);
  initial begin                             // initial block
    a0=0;a1=0;a2=0;a3=0;                    // at t=0
    b0=0;b1=0;b2=0;b3=0;cin=0;              // at t=0
    #100 a1=1;a3=1;b2=1;b3=1;               // at t=100
    #100 a0=1;a1=0;b1=1;b2=0;               // at t=200
    #100 a2=1;a3=0;cin=1;                   // at t=300
    #100 $finish;                           // at t=400 finish simulation
  end                                       // end of initial block
endmodule
```