



Future Ready

Introduction to Digital Design in Verilog

Prepared by

Beig Rajibul Hasan

Lecturer, CSE, BRAC University

Contact: rajib.hasan@bracu.ac.bd

Outline

1

Introduction to CAD tools and design flow of a CAD system

2

Emergence of HDL and basic structures of Verilog

3

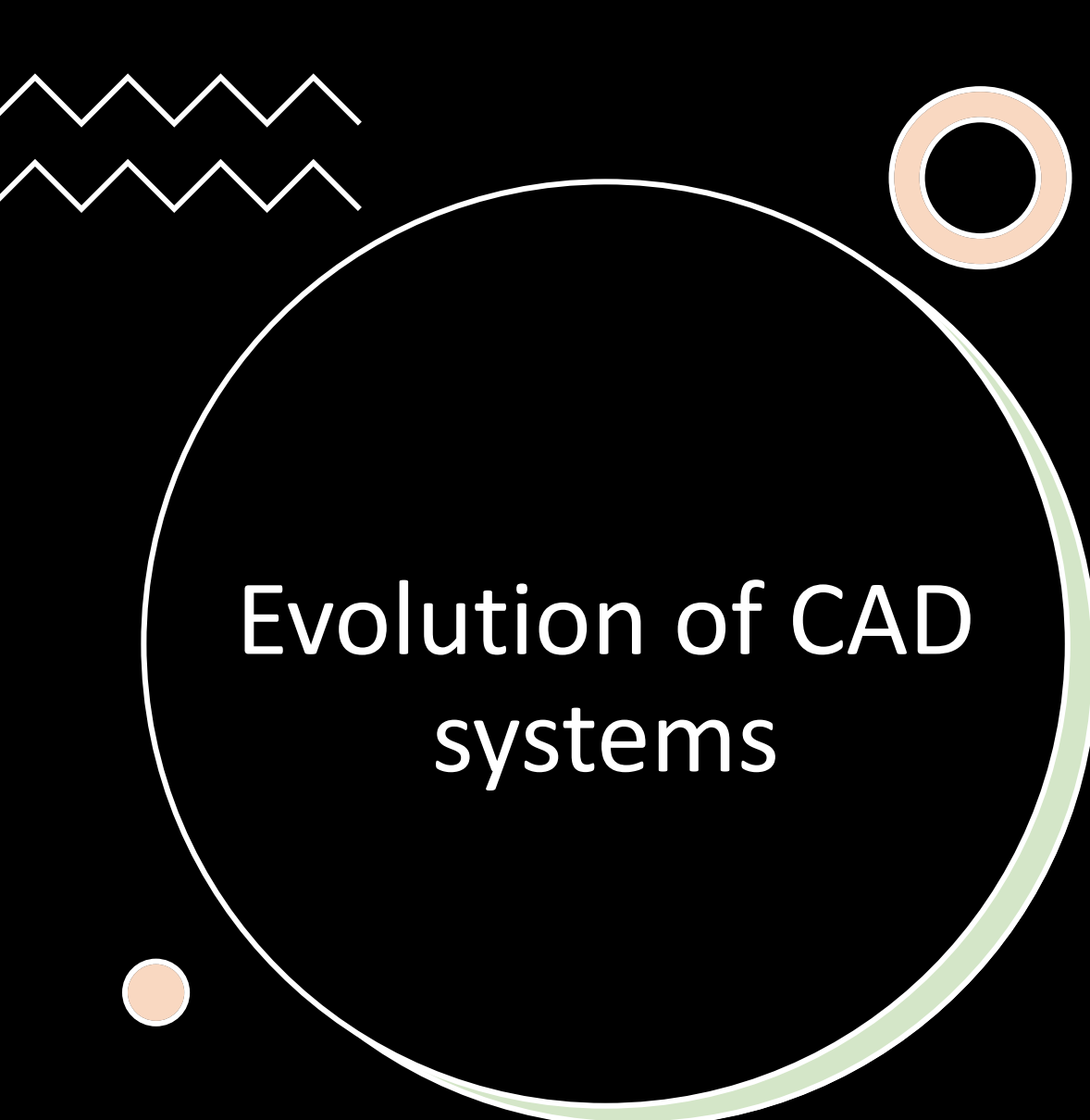
Digital circuit representations in Verilog

4

Verilog operators and subcircuits in Verilog

5

Advanced Verilog datatypes



Evolution of CAD systems

- Modern Integrated Circuits (IC) contain millions of transistors in a single chip. Placing so many transistors on the chip and finally *verifying the functionality of the chip manually* can be really *frustrating* for the chip architects.
- As the complexity of the *ICs grows every year proportionally with the shrinkage of the transistor sizes*, computer-aided techniques become crucial for the design and verification of the digital ICs.
- A general-purpose CAD system includes tools for different tasks such as: *design entry, logic synthesis, simulation and physical design*





Introduction to CAD Tools

▪ Design Entry

- This is the first step of any design process which involves entering into the CAD system a description of the circuit being designed either by **using schematic or writing source code**.

▪ Logic Synthesis

- Synthesis is the process of **generating a logic circuit** from an initial specification provided in the **Design Entry** stage. The output is a set of logic expressions that describe the logic functions needed to realize the circuit.



Introduction to CAD Tools



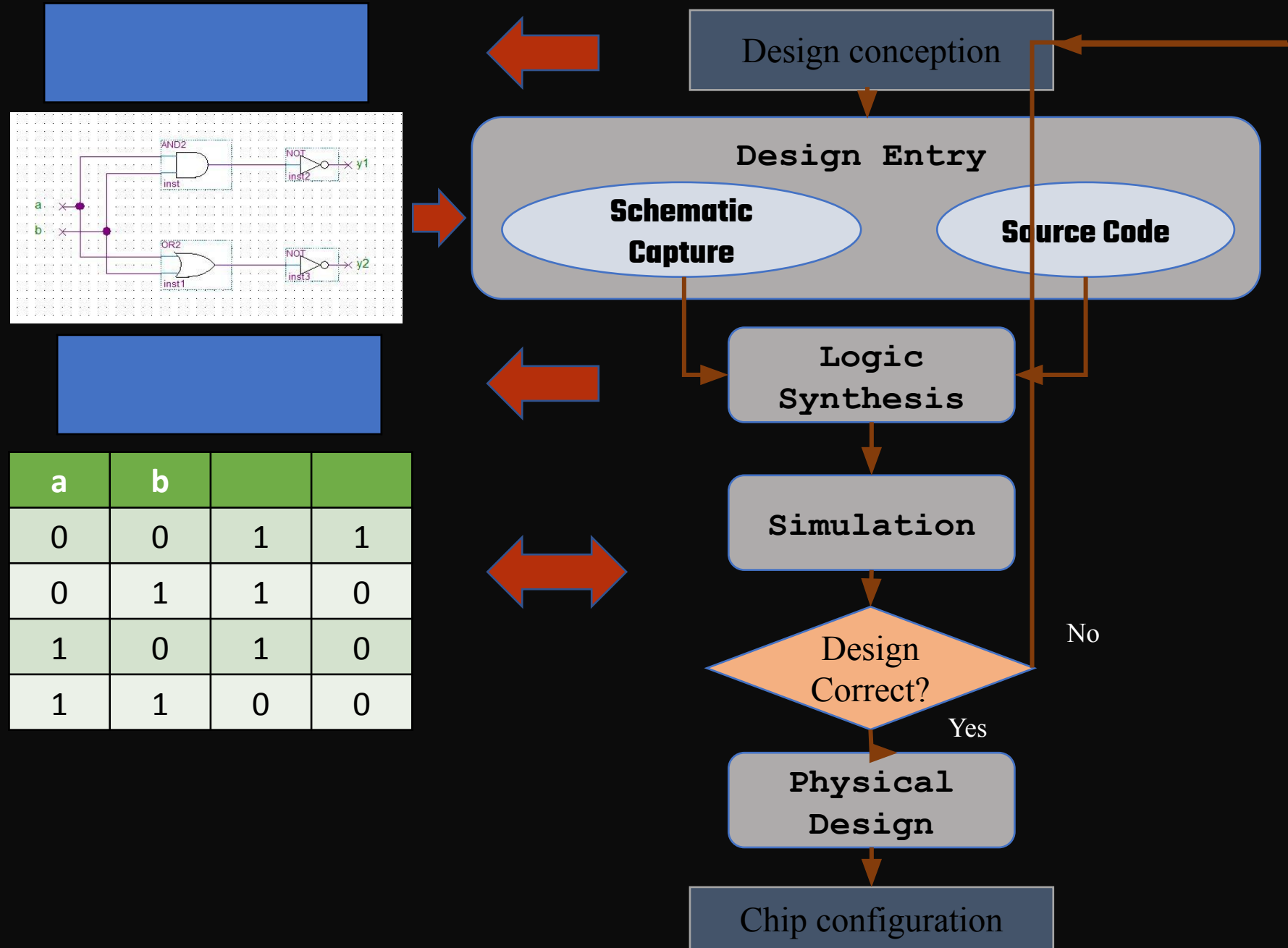
▪ Simulation

- A circuit represented in the form of **logic expressions can be simulated to verify** that it will function as expected. The tool that performs this task is called a functional simulator.

▪ Physical Design

- The physical design tools **map a circuit specified in the form of logic expressions** into a realization that makes use of the resources available on the target chip.

Design Flowchart of a Typical CAD System



CAD tools providers



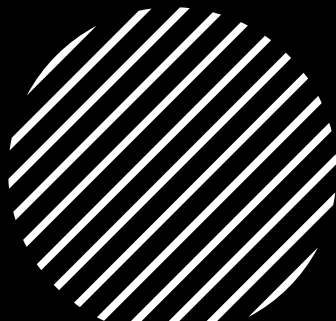
Emergence of HDLs

- **Hardware Description Languages (HDLs)** allow the designers to model the concurrency of the processes found in the hardware elements.
- Designs can be described **at a very abstract level** by using HDLs.
- The RTL description of the design derived from the **HDL source code can be converted into any fabrication technology** using the logic synthesizer tools.
- By describing the design in HDLs, **functional verification can be performed** very easily.



Verilog HDL

- Verilog HDL is a general-purpose **hardware description language which can describe the digital circuits** with C-like syntax.
- Most popular logic synthesis tools support Verilog HDL.
- Digital circuits can be described at the RTL of abstraction which ensures design portability.

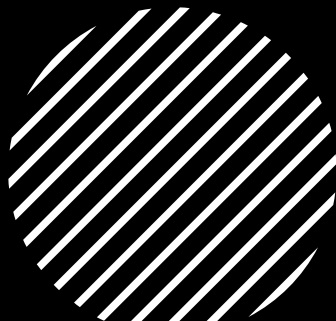




Basic building block of Verilog

- A module is the basic building block of Verilog. A **module consists of port declaration** and Verilog codes to perform the desired functionality.
- Ports are means for the Verilog module to communicate with other modules or interfaces. **Ports can be of 3 types, such as: *input*, *output*, *inout*.**
- A typical Verilog module declaration:

```
module <name> (<ports_list>);  
...  
// Verilog Codes //  
...  
endmodule
```

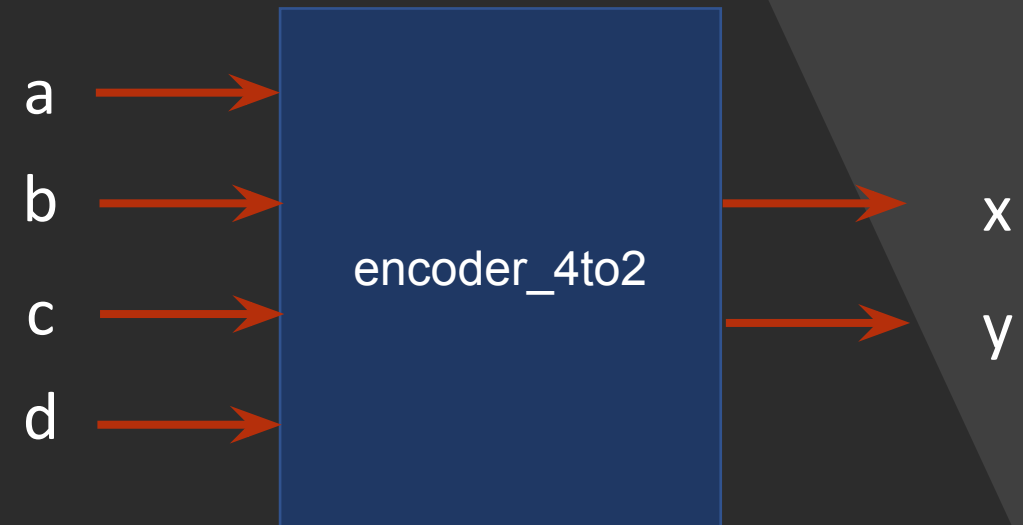


Verilog module and ports

Verilog module declaration

```
module encoder_4to2 (x, y, a, b, c, d);  
input a, b, c, d;  
output x, y;  
...  
// Verilog Code //  
...  
endmodule
```

Logic Synthesis



Basic Syntax and Lexical Conventions

- Documentation in Verilog code

- Documentation can be included in Verilog code by writing comments. A **short comment begins with a double slash (//)**. A **long comment spans multiple lines and is contained inside /* and */**.

```
module fulladd(s, cout, a, b, cin);  
// A full adder verilog module  
/*  
This module takes three inputs a, b, cin and adds them.  
The sum of the inputs are stored in s, the carryout is stored in cout.  
*/  
input a, b, c;  
output s, cout;  
// Verilog Code //  
endmodule
```

Basic Syntax and Lexical Conventions

- White space

- White space characters such as *SPACE* and *TAB* are ignored by the Verilog compiler. Although multiple statements can be written in a single line, placing each statement in a single line and using *indentation* within blocks of code are good ways to increase readability of the code.

- Number specification

- There are two types of number specifications found in verilog, *sized* and *unsized*.

- *sized* numbers are represented as: *<size> ' <base format> <number>*. Supported formats are:

Base format	Illustration
d	decimal
b	binary
h	hexadecimal
o	octal

Basic Syntax and Lexical Conventions

- If a number is specified *without a base format*, it is treated as a *decimal number* by the Verilog compiler.
- *unsized* numbers are specified without a size specification. *unsized* numbers are assigned a *specific number of bits which is simulator and machine-specific (at least 32 bits)*.

<i>sized</i> number representation	<i>unsized</i> number representation
5'b10001 // This is a 5-bit <i>binary</i> number	1254 // This is a 32-bit <i>decimal</i> number by default
4'hff01 // This is a 4-bit <i>hexadecimal</i> number	`h21ff // This is a 32-bit <i>hexadecimal</i> number
3'o123 // This is a 3-bit <i>octal</i> number	`o345 // This is a 32-bit <i>octal</i> number
2'd10 // This is a 2-bit <i>decimal</i> number	`b1100 // This is a 32-bit <i>binary</i> number

- Value Set

- Each individual signal/variable in Verilog can be assigned one of 4 values:

Value level	Condition in Hardware
0	logic 0 / False
1	logic 1 / True
x	undefined
z	high impedance



Basic Syntax and Lexical Conventions

- Nets

- Nets represent connections between hardware elements. Nets are continuously driven by the outputs of the devices they are connected to.

- Nets are declared with the keyword *wire*. A net is assigned the value *z* by default.

- Registers

- In verilog **register means a variable that can hold a value**. Unlike net, a register doesn't need a driver.

- Registers are declared with the keyword *reg*. The default value of a *reg* data type is *x*.

```
wire a, b; // wire declaration
reg clock; // register declaration
```

Basic Syntax and Lexical Conventions

- Vectors

- *wire* or *reg* type data types can also be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).

- The multibit nets/registers or vectors in general can be declared in Verilog using the syntax:

<data type> <MSB bit index : LSB bit index> <name>

```
module example(a, b);  
input b;  
output a;  
wire [7:0] in1, in2; // 8-bit wire type  
variables  
reg [0:31] base_clk; // 32-bit reg type  
variable  
assign a = in1[7] * in2[2];  
endmodule
```

Basic Syntax and Lexical Conventions

- Memories

- A **memory** is a **two-dimensional array of bits**. Verilog allows such a structure to be declared as a variable that is an array of vectors.

- Parameters

- A *parameter* associates an identifier name with **constants**.

```
reg [7:0] mem [15:0];  
// mem is defined as an array containing 16  
elements each being 8 bits wide
```

mem[15]	mem[14]	mem[1]	mem[0]
---------	---------	-------	--------	--------

```
reg [7:0] mem [7:0] [15:0];  
// mem is defined as a two-dimensional array.
```

mem[7][15]	mem[7][14]	...	mem[7][1]	mem[7][0]
mem[6][15]	mem[6][14]	...	mem[6][1]	mem[6][0]
...
mem[1][15]	mem[1][14]	...	mem[1][1]	mem[1][0]
mem[0][15]	mem[0][14]	...	mem[0][1]	mem[0][0]

```
parameter n = 5;
```

Basic Syntax and Lexical Conventions

Identifier Names

- Identifiers are the **names of variables** and other elements in Verilog code.
- Valid identifier can include any letter and digit as well as “_” and “\$” characters. There are two restrictions too , an identifier must not begin with a digit and it should not be a Verilog keyword. Furthermore, Verilog is case sensitive.

Identifier name	Validity
x1	Valid
x_y	Valid
1x	Invalid
+y	Invalid
x*y	Invalid
258	Invalid
ex_\$1	Valid

Verilog Representations of Digital Circuits

Verilog allows designers to describe a digital circuit in several ways. Among them two fundamental representations are: *structural representation* and *behavioral representation*.

- Structural representation

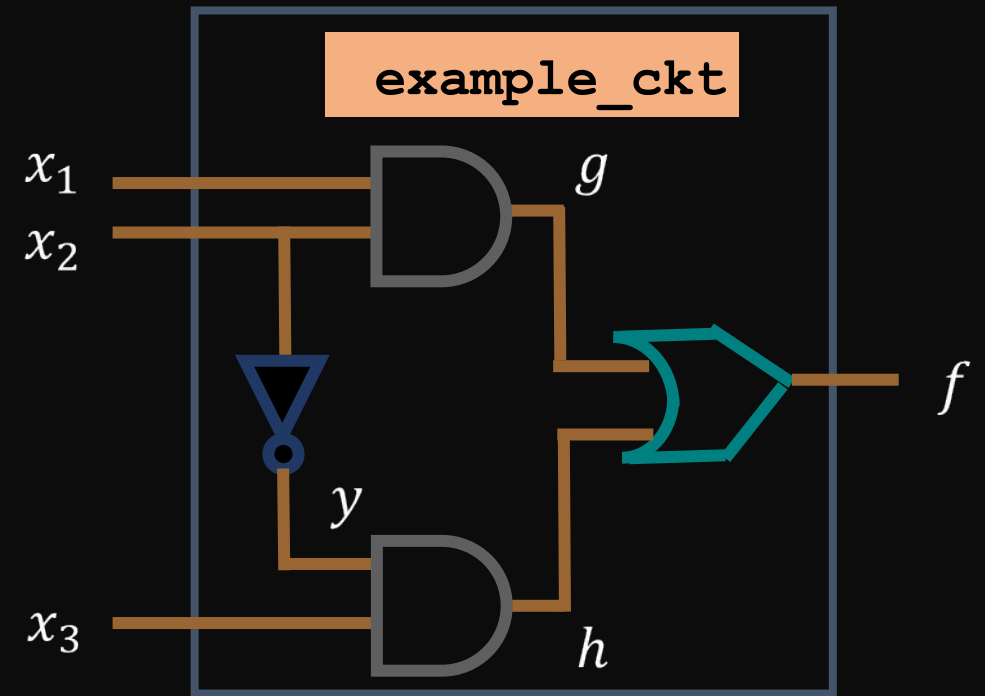
- Structural representation is to use Verilog's **gate-level primitives to describe the digital circuit**. Various gate-level primitives are included in Verilog such as: *and* gate, *or* gate, *not* gate, *nand* gate, *nor* gate etc.

- Behavioral representation

- Using gate-level primitives can be tedious while designing larger circuits. Instead, the designers use **more abstract expressions and programming constructs** to describe the circuit. This is called the behavioral representation of the digital circuit.

Structural representation

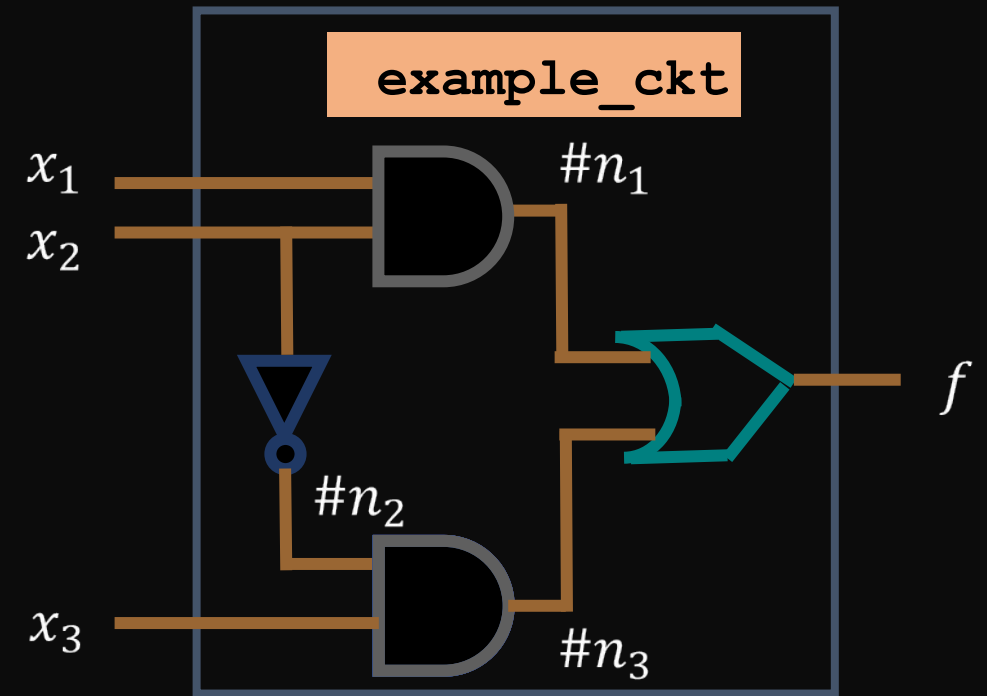
```
module example_ckt(f, x1, x2,
x3);
    input x1, x2, x3;
    output f;
    and(g, x1, x2);
    not(y, x2);
    and(h, y, x3);
    or(f, g, h);
endmodule
```



x1	x2	x3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Behavioral representation

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



x1	x2	x3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



Verilog Operators

Verilog supports a large number of operators for carrying out different types of operations. Verilog operators can be broadly classified into the following categories:

- ☐ Bitwise operators
- ☐ Logical operators
- ☐ Arithmetic operators
- ☐ Relational operators
- ☐ Shift operators
- ☐ Concatenate operator
- ☐ Replication operator
- ☐ Condition operator



Verilog Operators



Bitwise operators

Operator	Operation
$\sim A$	This will produce 1's complement of A
$-A$	This will produce 2's complement of A
$A \& B$	Bitwise AND
$A B$	Bitwise OR
$A \wedge B$	Bitwise XOR
$A \wedge \sim B / A \sim \wedge B$	Bitwise XNOR



Verilog Operators

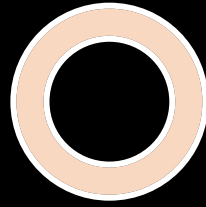


Logical operators

These operators work on single or multi bit operands but generate 1 bit result i.e., *True* or *False*.

Operator	Operation
!A	NOT A(!A) produces “1(True)” only if all bits of A are 0 else !A gives “0(False)”
A&&B	The result of A&&B is “1(True) if both A and B are nonzero
A B	A B gives “1(True)” unless both A and B are zero.

Verilog Operators

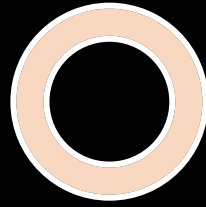


Arithmetic operators

Operator	Operation
A+B	Addition of two single or multibit numbers
A-B	Subtraction of two single or multibit numbers
A*B	Multiplication of two single or multibit numbers
A/B	Division of two single or multibit numbers
A%B	This returns the remainder of the integer division A/B



Verilog Operators



Relational operators

Operator	Operation
$A == B$	1 (<i>True</i>) if A is equal to B, 0 (<i>False</i>) otherwise
$A != B$	1 (<i>True</i>) if A is not equal to B, 0 (<i>False</i>) otherwise
$A > B$	1 (<i>True</i>) if A is greater than B, 0 (<i>False</i>) otherwise
$A < B$	1 (<i>True</i>) if A is less than B, 0 (<i>False</i>) otherwise

Verilog Operators

Operator type	Symbol	Operation performed
Shift	>>	Shift right logical (division by 2)
	<<	Shift left logical (multiplication by 2)
Concatenate	{}	Join bits
Replicate	{n{m}}	Duplicate m, n times
Condition	D = A ? B : C	D is equal to B if A is True, otherwise D is equal to C

```
wire [7:0] d;  
wire [31:0] e;  
wire [31:0] f;  
assign f = {d, e[23:0]}; // Concatenation + Slicing  
assign f = { 32{d[5]} }; // Replication + Indexing
```

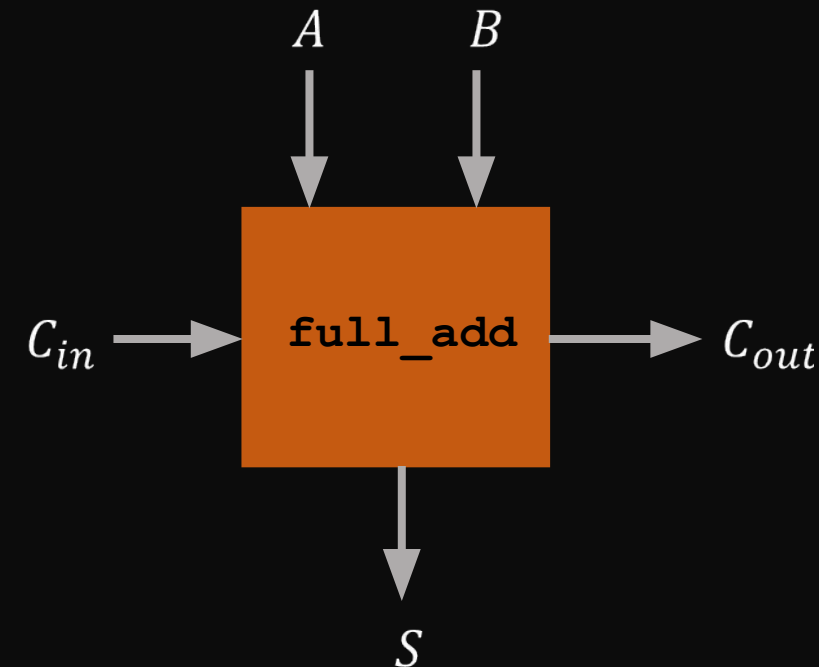
Concurrent Statements

In any HDL, concurrent statement means the code may include a number of statements and each represent a part of the circuit.

What concurrent means:

Concurrent is used because the statements are considered in parallel and the ordering of statements in the code doesn't matter. Most frequently used concurrent statements in Verilog are the continuous assignments.

```
module full_add(S, Cout, A, B, Cin);  
// This module implements a 1-bit full adder  
input A, B, Cin;  
output S, Cout;  
  
assign S = A ^ B ^ Cin;  
assign Cout = (A & B) | (Cin & (A ^ B));  
endmodule
```



Procedural Statements

In addition to the concurrent statements Verilog also provides procedural statements. As discussed, concurrent statements are executed in parallel, **procedural statements are evaluated in the order they appear in the code**. Verilog syntax requires that procedural statements to be contained inside an **always** block

Always Block:

Concurrent is used because the statements are considered in parallel and the ordering of statements in the code doesn't matter. Most frequently used concurrent statements in Verilog are the continuous assignments.

- An **always** block is a construct that contains one or more procedural statements.
- When multiple statements are included in an always block . The **begin** and **end** keywords must be used.
- **Output variables within the always block must be variable of reg type.**

```
always @ (sensitivity_list)
[begin]
    [procedural assignment statements]
    [if-else statements]
    [case statements]
    [while, repeat and for loops]
[end]
```

Procedural Statements

The If-else statement:

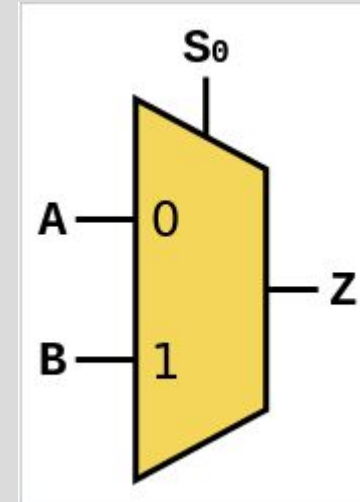
- If expression1 is *True* then the first statement is evaluated.
- If not , then the compiler will consider other expressions.
- The **else if** and **else** clauses are optional.
- When multiple statements are involved, they have to be included inside a **begin-end block**

```
always @(*)  
    if(expression1)  
        begin  
            statement;  
        end  
    else if(expression2)  
        begin  
            statement;  
        end  
    else  
        begin  
            statement  
        end  
end
```

Procedural Statements

The If-else statement:

```
module mux2to1(A, B, S, Z);  
  
    input A, B, S;  
    output reg Z;  
  
    always @(A, B, S)  
    begin  
        if(S == 0)  
            Z = A;  
        else  
            Z = B;  
        end  
    end  
endmodule
```



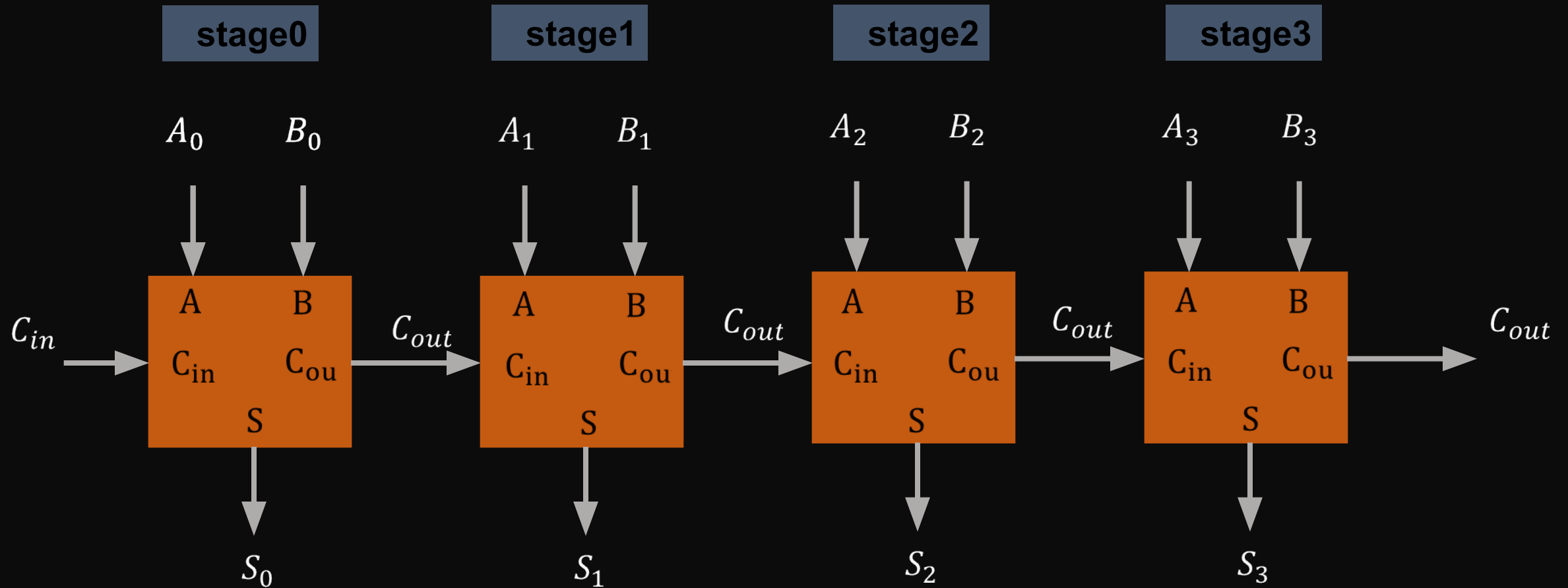
Subcircuits in Verilog

- A Verilog module can be included as a subcircuit in another module.
- Both module must be defined in the same file or else Verilog compiler must be told where each module is located.
- The general form of module instantiation expression is

<module_name> <instance_name> (<port_name[expressions]>)

- In the above definition
 - *module_name* is the name of the module of the child circuit that is to be included in the parent circuit.
 - *instance_name* can be any legal Verilog identifiers.
 - *port_name* basically is the list of ports that specify the connections that will be passed to the subcircuit.

4-bit Ripple Carry Adder



4-bit Ripple Carry Adder

```
module fulladd4(S0, S1, S2, S3, Cout, A0, A1, A2, A3, B0, B1, B2, B3, Cin);  
    input A0, A1, A2, A3, B0, B1, B2, B3, Cin;  
    output S0, S1, S2, S3, Cout;  
    wire Cout0, Cout1, Cout2;  
    fulladd stage0 (S0, Cout0, A0, B0, Cin);  
    fulladd stage1 (S1, Cout1, A1, B1, Cout0);  
    fulladd stage2 (S2, Cout2, A2, B2, Cout1);  
    fulladd stage3 (S3, Cout, A3, B3, Cout2);  
endmodule  
  
module fulladd(S, Cout, A, B, Cin);  
    // This module implements a 1-bit full adder  
    input A, B, Cin;  
    output S, Cout;  
  
    assign S = A ^ B ^ Cin;  
    assign Cout = (A & B) | (Cin & (A ^ B));  
endmodule
```

wire vs reg

When to use which?

- If a signal needs to be assigned **inside an *always* block**, it must be declared as a *reg*.
- If a signal is assigned using a continuous assignment statement, it must be declared as a wire.
- By default, module input and output ports are wires; if any output ports are assigned in an *always* block, they must be explicitly declared as *reg*:

output reg <signal name>

How to know if a net represents a register or a wire?

- A *wire* net always represents a combinational link
- A *reg* net represents a wire if it is assigned in an ***always @ (*) block***
- A *reg* net represents a register if it is assigned in an ***always @ (posedge/negedge clock) block***

Additional Verilog Datatypes

Verilog supports 3 additional *register* datatypes such as: integer, real and time.

- **Integer datatypes** are general-purpose register datatypes which are broadly used as loop variables (*iterators*). The default size of an integer is 32 bits. Integer type variables are declared with the Verilog keyword **integer**. Unlike **reg** which can only store unsigned values, **integer** can store signed quantities as well.
- **Real number** constants and real register datatypes are declared with the keyword **real**. They can be specified in **decimal notation** or in **scientific notation**. A real variable is declared with the keyword **real**. When a real value is assigned to an integer, the real number is rounded off to the nearest integer.
- Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A **time variable** is declared with the keyword **time**. The default size of a time variable is 64 bits. The system function **\$time** is invoked to get the current simulation time.

Additional Verilog Datatypes

```
// Syntax for integer, real and time register datatypes
real delta, alpha; // Defines two real variables called 'delta' & 'alpha'
initial
/* initial block represents procedural statements in Verilog. The lines
written inside this block are executed only once, right at the beginning
of the simulation type. */
begin
    delta = 4e10;
    alpha = 3.14;
end

integer i; // Defines an integer 'i'
initial
    i = alpha; // i gets the value 3 (rounded value of 3.14)

time sim_time; // Defines a time variable named 'sim_time'
initial
    sim_time = $time; // Saves the current simulation time
```

Text & References

#	Title	Author(s)	Edition
1.	CMOS VLSI Design	N.H.E. Weste, D. Harris & A. Banerjee	4 th ed.
2.	Fundamentals of Digital Logic with Verilog Design	Stephen Brown & Zvonko Vranesic	2 nd /3 rd ed.
3.	Verilog HDL (A guide to Digital Design and Synthesis)	Samir Palnitkar	1 st ed.

Thank You