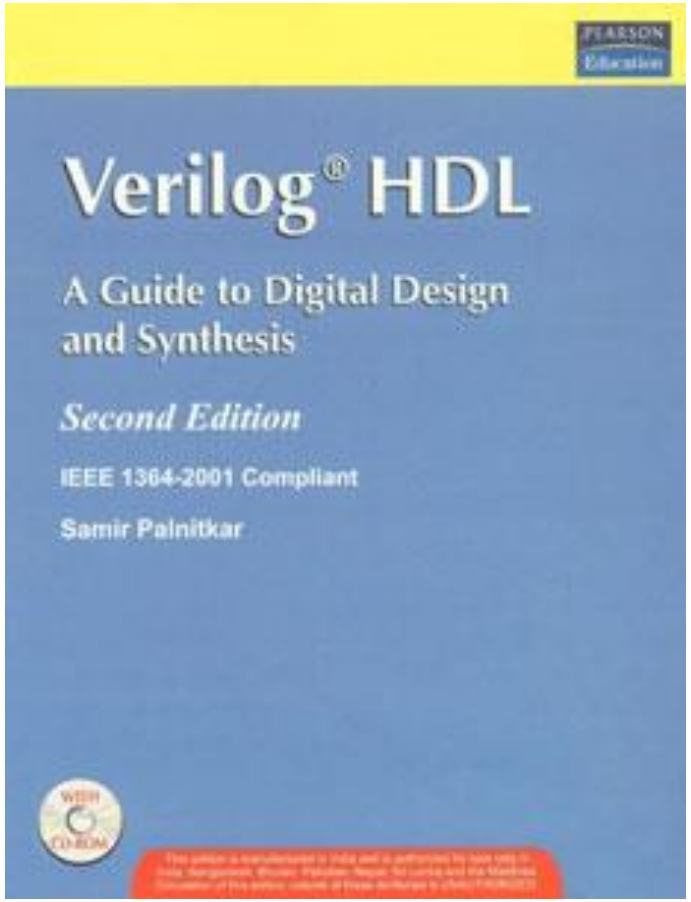
Verilog HDL

OVERVIEW OF DIGITAL DESIGN WITH VERILOG HDL

Reference

Verilog HDL: A Guide to Digital Design and Synthesis, 2e, Samir Palnitkar

Chapters 1



Evolution of Computer-Aided Digital Design

Small circuits with just hundreds of transistors :

- Design the layout on paper or by hand on a graphics computer terminal.
- Test on a breadboard.

But a modern processors has millions of transistors (eg : Intel i7 has 731,000,000 transistors) :

- Computer aided techniques for design and verification is required.

Emergence of HDL

- Programming languages like C describe computer programs which are sequential in nature.
- But digital circuits involves concurrency -> general programming languages doesn’t suit.
- Languages that describe digital circuits called Hardware Description Languages (HDL) came into existence.

Integrated Circuit Design Processes

Formal and precise description of a complex circuit in an abstract level

Automated analysis and simulation

Automated synthesis into a netlist (specification of electronic component and how they are connected)

Automated placing of electronic components and routing of wires to be sent for fabrication

HDL

Hardware Description Languages (HDL)

Example HDL :

- **Verilog HDL**
- **VHDL**

} Widely used

- AHDL
- AHPL
- Bluespec

Verilog HDL - History

- Verilog HDL invented by Philip Moorby in 1983 at Gateway Design Automation.
- Verilog- based synthesis tool introduced by Synopsys in 1987
- Gateway Design Automation bought by Cadence in 1989
- Verilog placed in public domain to compete with VHDL
  - Open Verilog International (OVI) IEEE 1364 -1995 and
  - revised version IEEE 1364 -2001
  - revised version IEEE 1364 -2005

Verilog HDL

- Easy to use : similar to syntax of C programming language
- Mixed level modelling
  - Behavioral - Algorithmic (like high level language)
  - Data-flow - Register transfer (synthesizable)
  - Gate-level - Structural (AND, OR ……)
- Single language for design and simulation
- Built-in primitives, logic functions and data types
- User-defined primitives
- Built-in High-level programming constructs

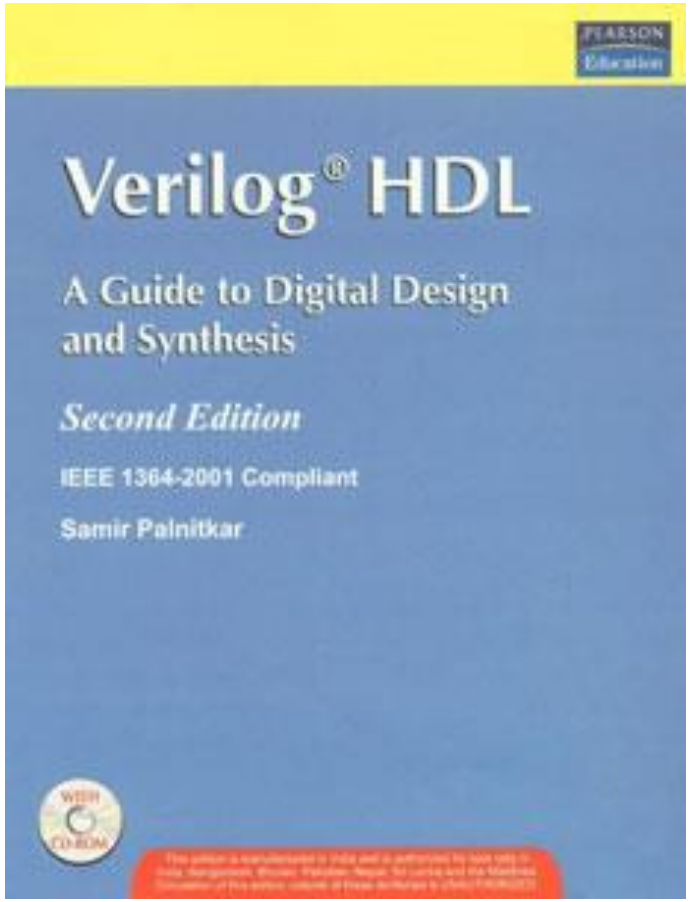
Verilog HDL

BASIC CONCEPTS

Reference

Verilog HDL: A Guide to Digital Design and Synthesis, 2e, Samir Palnitkar

Chapters 3



Outline

- **Lexical conventions**
- Data types
- System tasks and compiler directives

Lexical Conventions

- The basic lexical conventions used by Verilog HDL are similar to those in the C.
- Verilog contains a stream of tokens.
- Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords.
- Verilog HDL is case-sensitive.
- All keywords are in lowercase.

Whitespace

- Blank spaces, tabs and newlines comprise the whitespace.
- Whitespace is ignored by Verilog except when it separates tokens.
- Whitespace is not ignored in strings.

Comments

- Comments can be inserted in the code for readability and documentation.
- A one-line comment starts with "//".
- A multiple-line comment starts with "/*" and ends with "*/".
- Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment
/* This is a multiple line
comment */
/* This is /* an illegal */ comment */
/* This is //a legal comment */
```

Operators

- Operators are of three types: unary, binary, and ternary.
- Unary operators : precede the operand.
- Binary operators : appear between two operands.
- Ternary operators : have two separate operators that separate three operands.

```
a = ~ b; /* ~ is a unary operator. b is the operand */
a = b && c; /* && is a binary operator. b and c are
operands */
a = b ? c : d; /* ?: is a ternary operator. b, c and d
are operands */
```

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Logical Equality Operators	==, !=
Case Equality Operators	===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^(xor), ~^(xnor)
Unary Reduction Operators	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	? :
Concatenation Operator	{ }
Replication Operator	{ { } }

Number Specification

Sized numbers : <size> '<base format> <number>

<size> : Number of bits in a number in decimal

'<base format> : decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O)

<number> : digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a (or A), b or (B), c (or C), d (or D), e (or E), f (or F).

```
4'b1111 // This is a 4-bit binary number
12'habc // This is a 12-bit hexadecimal number
16'd255 // This is a 16-bit decimal number
```

Number Specification

Unsized numbers : '<base format> <number>

Numbers written without <size> have default number of bits that is simulator and machine specific (must be at least 32)

Numbers written without <base format> are decimal numbers by default.

```
'hc3 // This is a 32-bit hexadecimal number
'o21 // This is a 32-bit octal number
23456 // This is a 32-bit decimal number by default
```

Strings

- A string is a sequence of characters that are enclosed by double quotes.
- Strings are treated as a sequence of one-byte ASCII values.

```
"Hello Verilog World" // is a string
"a / b" // is a string
```

Identifiers and Keywords

Keywords :

- Keywords are special identifiers reserved to define the language constructs.
- Keywords are in lowercase.
- Eg : and, if, for, begin, end …. (See Appendix C of the book for all keywords)

Identifiers and Keywords

Identifiers :

- Identifiers are names given to user provided objects in descriptions
- Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign ($).
- Identifiers are case sensitive.
- Identifiers start with an alphabetic character or an underscore.

Identifiers and Keywords

```
reg value; // reg is a keyword; value is an identifier
input clk; // input is a keyword, clk is an identifier
```

Outline

- Lexical conventions
- **Data types**
- System tasks and compiler directives

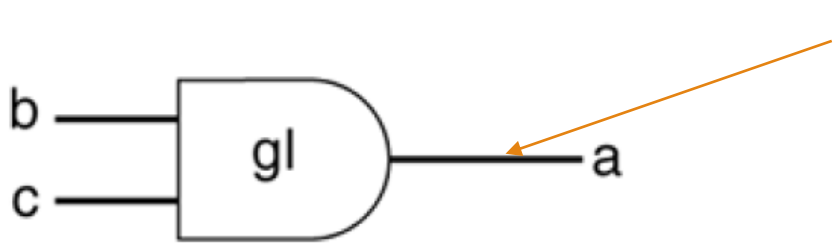
Value Set

Verilog supports four values to model the functionality of real hardware.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Nets (wires)

- Nets represent connections between hardware elements.
- Nets are declared with the keyword **wire**.
- Nets get the output value of their drivers. If a net has no driver, it gets the value z (default value).



Net **a** is driven by the output of the gate **gl**.

```
wire a; // Declare net a for the above circuit
wire b,c; // Declare two wires b,c for the above circuit
wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.
```

Registers

- Registers represent data storage elements.
- Registers retain value until another value is placed onto them.
- A variable that can hold a value.
- Declared by keyword **reg**.

```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
  reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
  #100 reset = 1'b0; // after 100 time units reset is deasserted.
end
```

Don’t confuse the term registers in Verilog with hardware registers built from flipflops in real circuits.

Vectors

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

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
```

Vector part select

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
```

It is possible to address bits or parts of vectors declared above

```
busA[7] // bit # 7 of vector busA
bus[2:0] // Three least significant bits of vector bus,
```

Integer, Real, and Time (Register Data Types)

Integer

- The default width for an integer is the host-machine word size, but is at least 32 bits.
- Integers can store signed values as well while variables declared as reg could only store unsigned.

```
integer counter; // general purpose variable used as a counter.
initial
counter = -1; // A negative one is stored in the counter
```

Integer, Real, and Time (Register Data Types)

Real

- Specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 10^6)

```
real delta; // Define a real variable called delta
initial
begin
       delta = 4e10; // delta is assigned in scientific notation
       delta = 2.13; // delta is assigned a value 2.13
end
integer i; // Define an integer i
initial
i = delta; // i gets the value 2 (rounded value of 2.13)
```

Integer, Real, and Time (Register Data Types)

Time

- Verilog simulation is done with respect to simulation time.
- Time register datatype is used in Verilog to store simulation time.
- Simulation time is measured in terms of simulation seconds.

```
time save_sim_time; /* Define a time variable save_sim_time */
initial
  save_sim_time = $time; /* Save the current simulation time */
```

Strings

- Strings can be stored in reg.
- Each character in the string takes up 8 bits (1 byte).
- If **register width > string size** : bits to the left of the string is filled with zeros.
- If **register width < string size** : leftmost bits in the string are truncated.
- Special characters such as \n, \t are allowed.

```
reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide
initial
    string_value = "Hello Verilog World"; // String can be stored
// in variable
```

Outline

- Lexical conventions
- Data types
- **System tasks and compiler directives**

System Tasks

- All system tasks appear in the form **$<keyword>**.
- Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks.
- Some examples are:

$bitstoreal	$countdrivers	$display	$fclose
$fdisplay	$fmonitor	$fopen	$fstrobe
$fwrite	$finish	$getpattern	$history
$incsave	$input	$itor	$key
$list	$log	$monitor	$monitoroff
$monitoron	$nokey	$time	

System Task : Displaying Information

- $display is the main system task for displaying values of variables or strings or expressions.
- The format of $display is very similar to printf in C.
- A $display inserts a newline at the end of the string by default.
- A $display without any arguments produces a newline.

- Usage: $display(p1, p2, p3,....., pn);

p1, p2, p3,..., pn can be quoted strings or variables or expressions

$display : Format Specification

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

$display : Examples

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World


//Display value of current simulation time 230
$display($time);
-- 230
```

$display : Examples

```
// Display value of 41-bit virtual address 1fe0000001c at time 200
reg [40:0] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c


//Display value of port_id 5 in binary
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
-- ID of the port is 00101
```

Compiler Directives

- All compiler directives are defined by using the **'<keyword>** construct.
- We will focus on two important directives only; 'define and 'include.

'define

- similar to the #define construct in C.

```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32
```

Compiler Directives

' include

- Similarly to the #include in C.

```
// Include the file header.v, which contains declarations in the
// main verilog file design.v.
'include header.v
...
...
<Verilog code in file design.v>
...
...
```

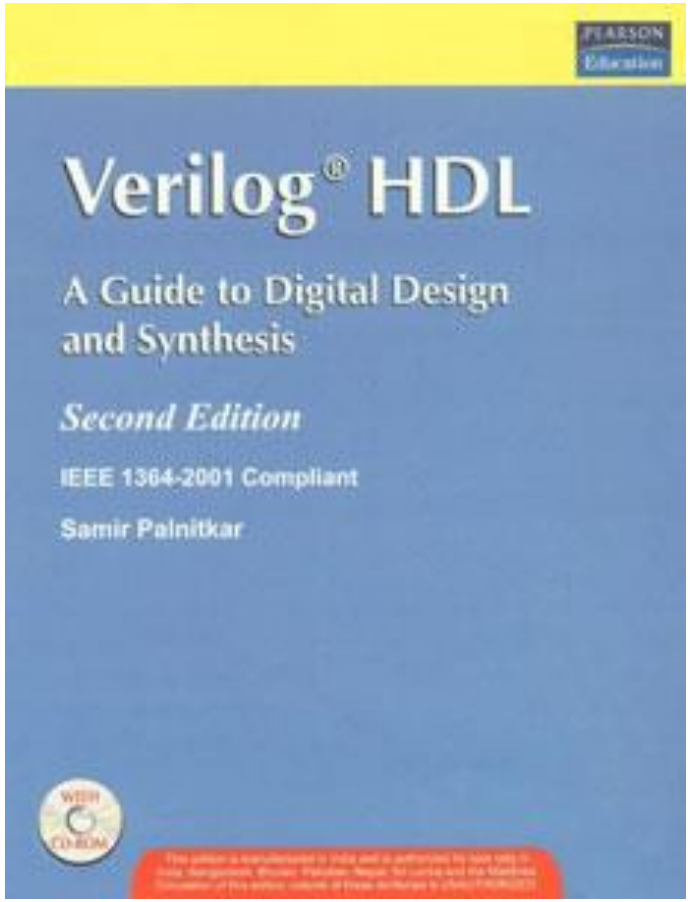
Verilog HDL

HIERARCHICAL MODELING CONCEPTS

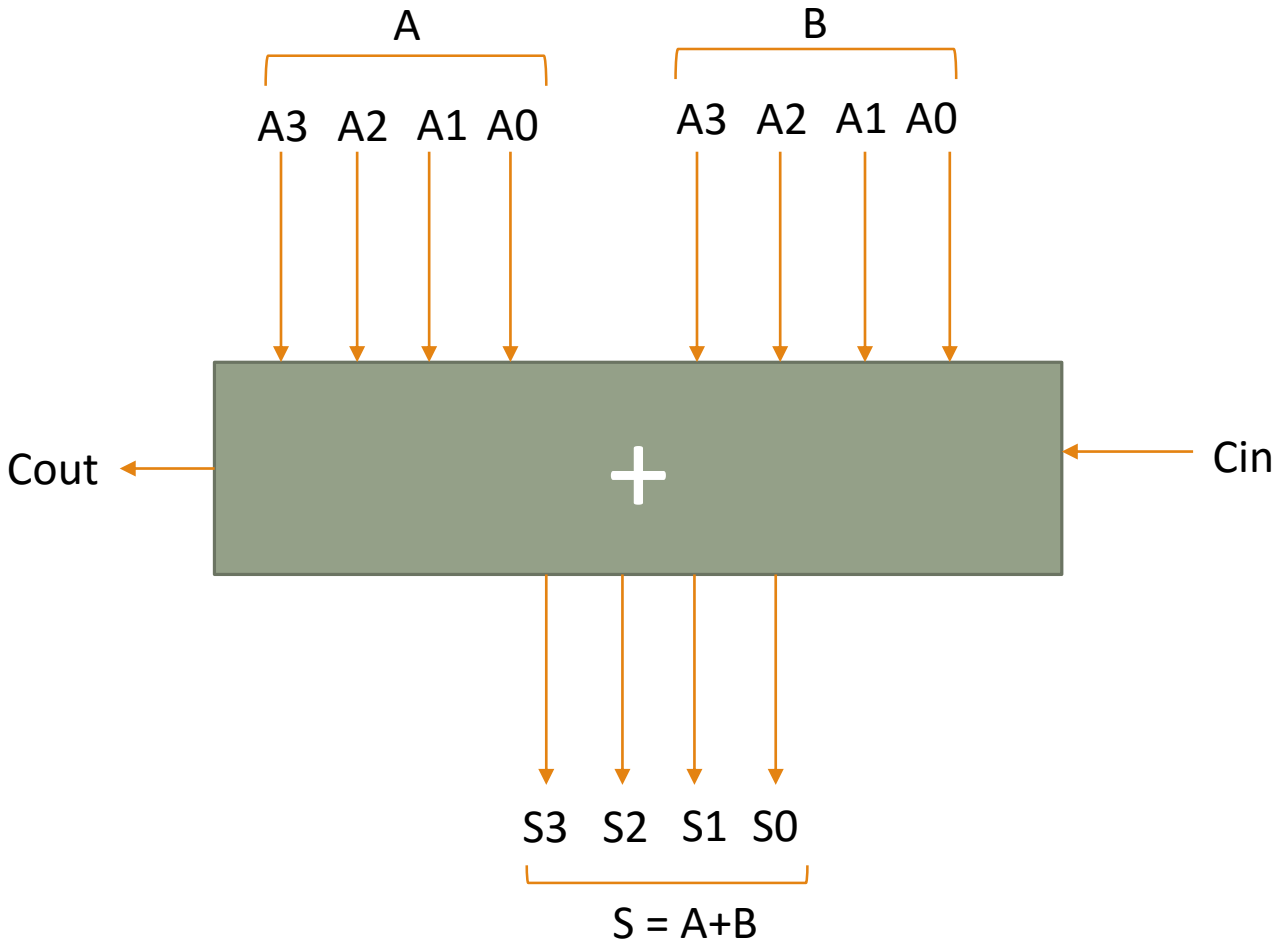
Reference

Verilog HDL: A Guide to Digital Design and Synthesis, 2e, Samir Palnitkar

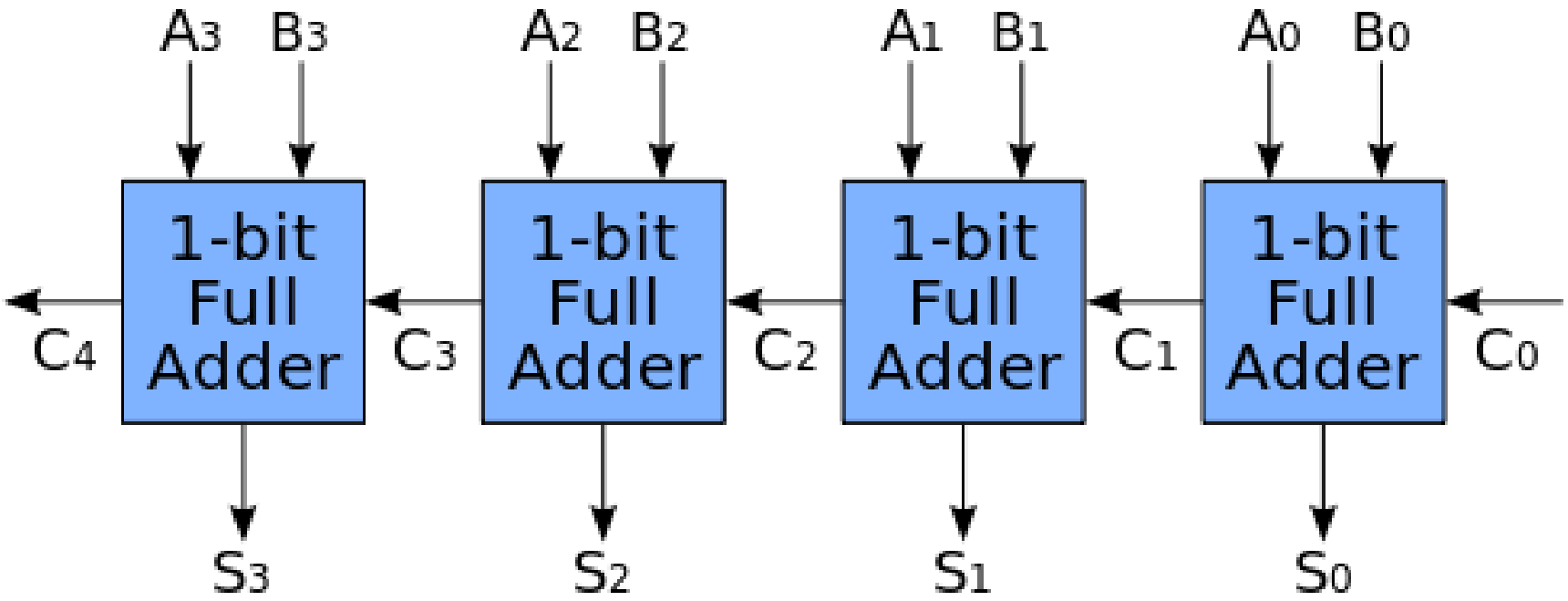
Chapters 2



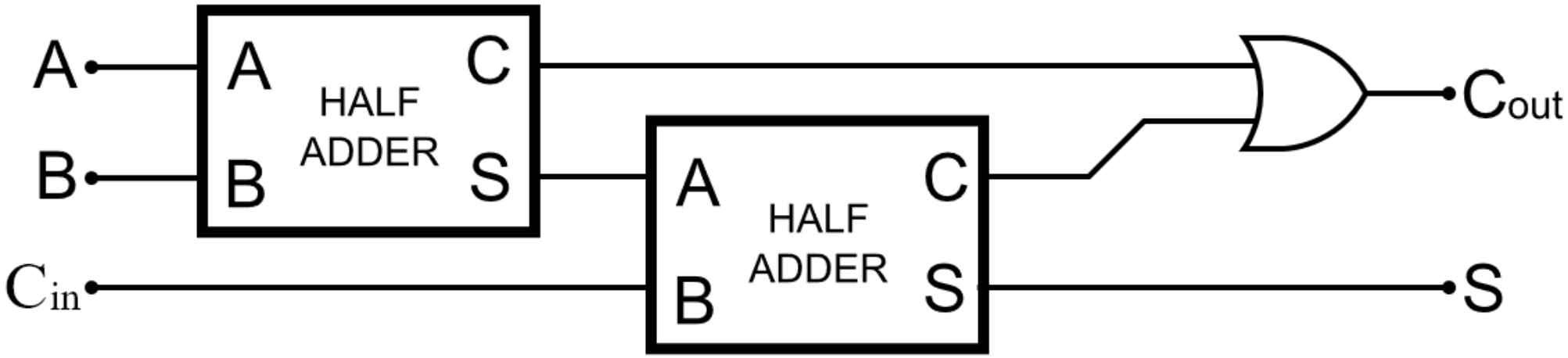
4-bit ripple carry adder : top level view



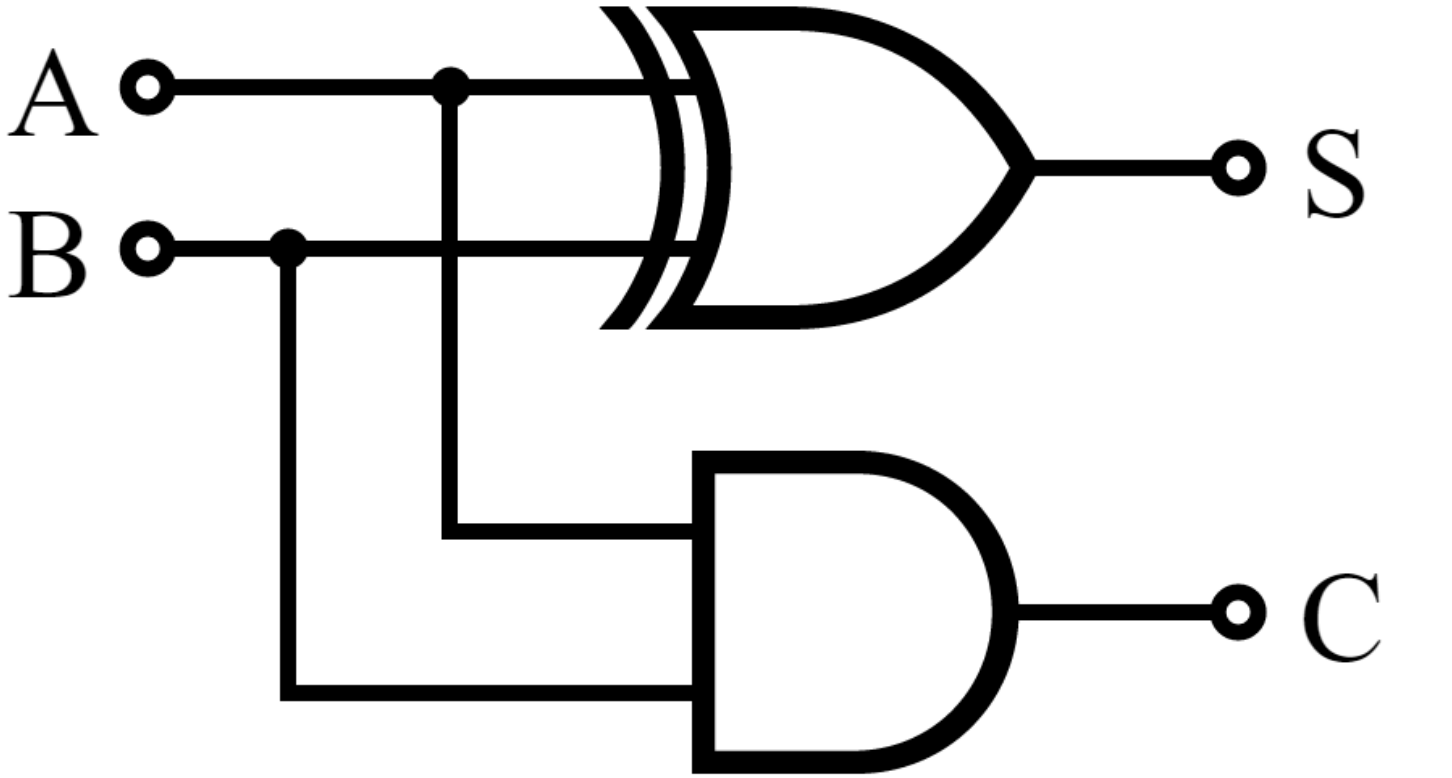
4-bit adder implemented using full adders



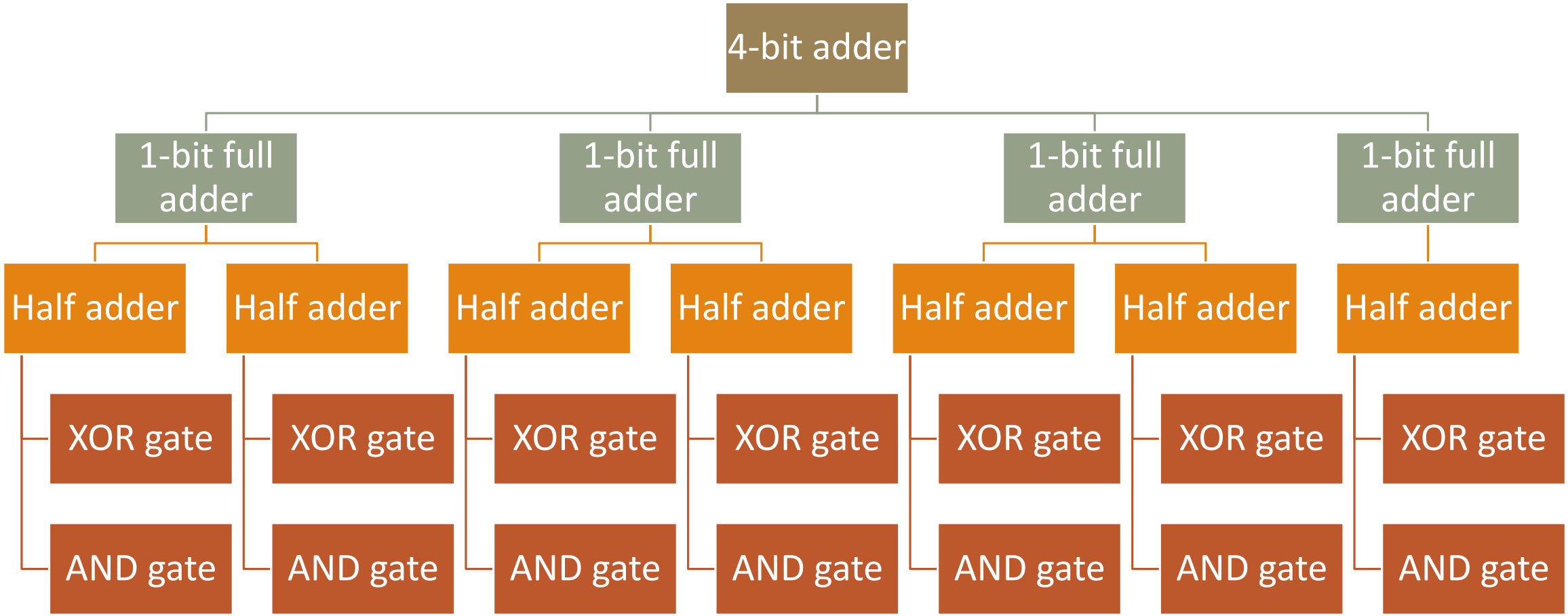
Full adder built using half adders



Half adder implemented using logic gates



Design Hierarchy of the 4-bit adder



Design Methodologies

There are two basic types of digital design methodologies:

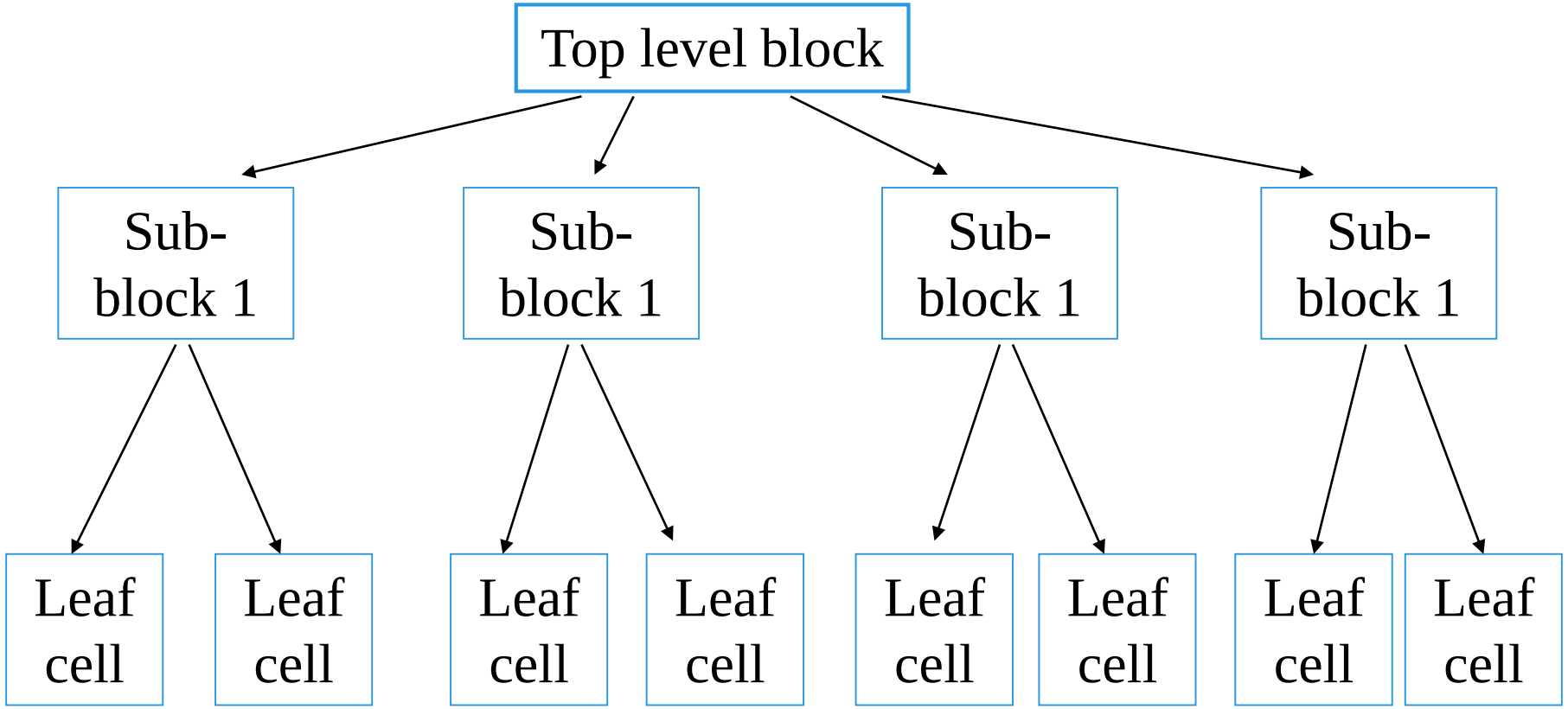
- Top-down design methodology

  we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.

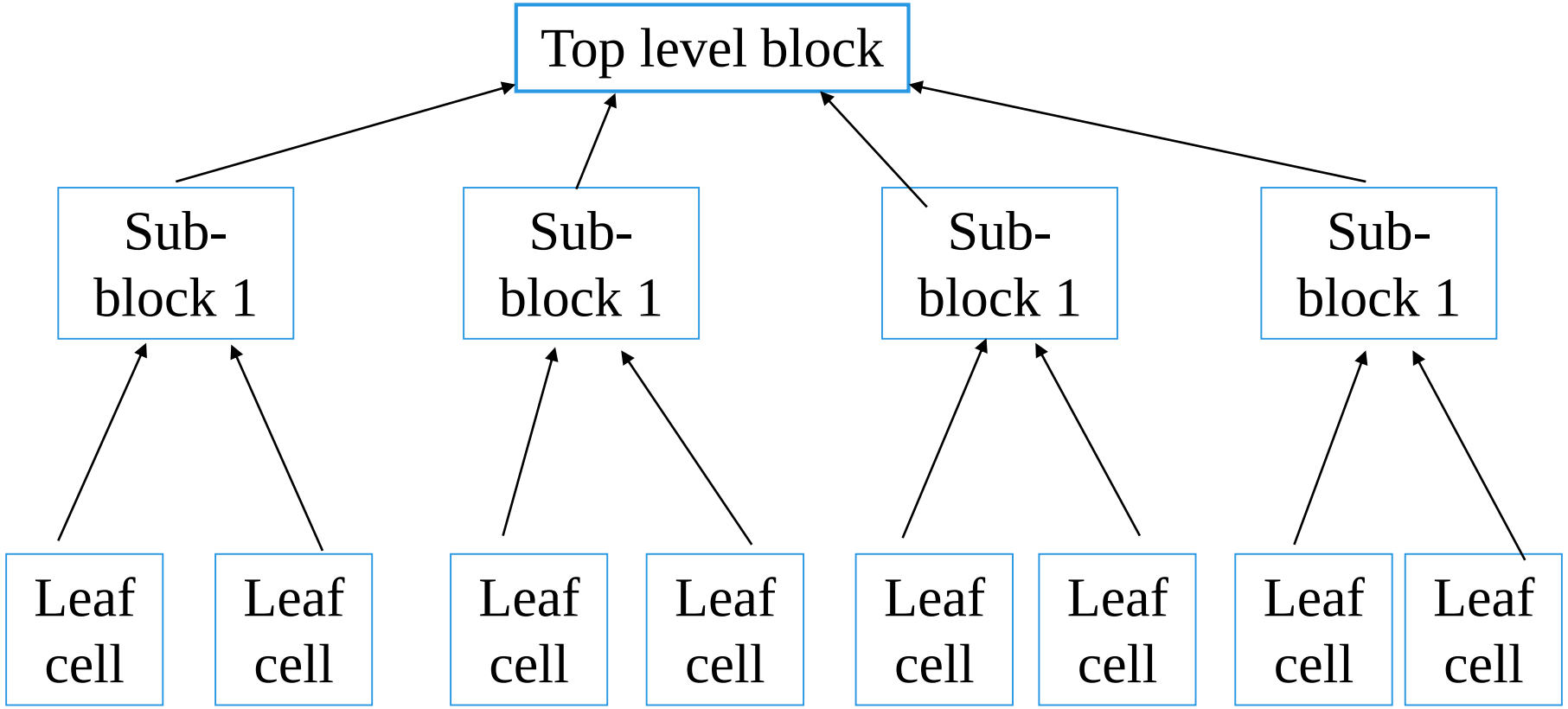
- Bottom-up design methodology

  we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design

Top-down design methodology



Bottom-up design methodology



Hybrid method

Typically, a combination of top-down and bottom-up flows is used.

- Design architects -> define specifications of the top-level block.
- Logic designers -> break up the functionality of the top level block into blocks and sub-blocks.

At the same time,

- circuit designers -> design optimized circuits for leaf-level cells -> build higher-level cells by using these leaf cells

The flow meets at an intermediate point!

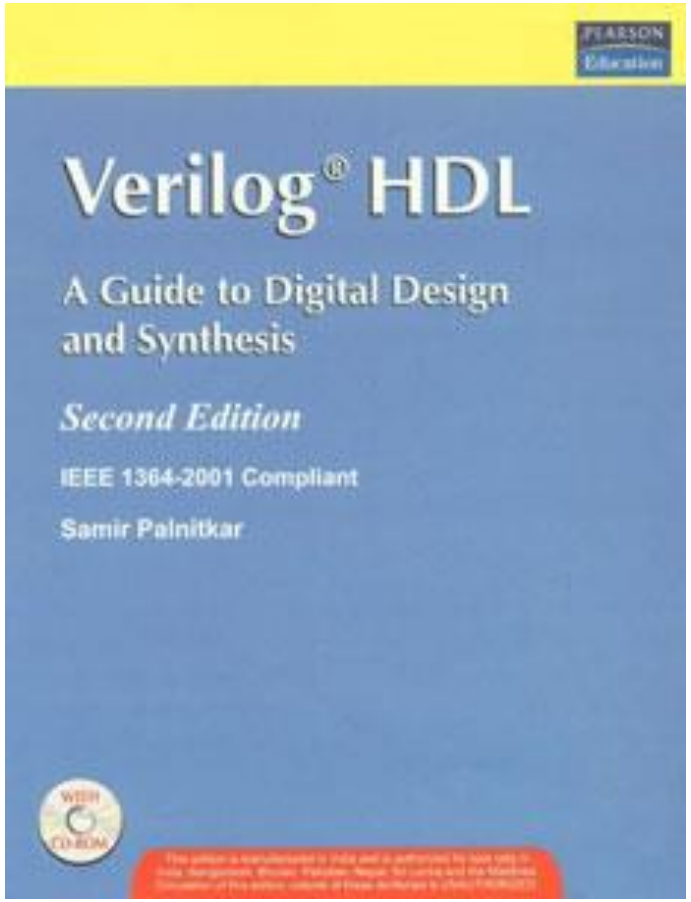
Verilog HDL

MODULES AND PORTS

Reference

Verilog HDL: A Guide to Digital Design and Synthesis, 2e, Samir Palnitkar

Chapters 4



Modules

- Hierarchical modeling concepts in Verilog :
  - facilitated by the concept of a module.
- A module is the basic building block in Verilog.
- A module is an element or a collection of lower-level design blocks.

Modules

- Typically, elements are grouped into modules to provide common functionality that is used at many places in the design.
- Essential components of any module: Keywords *module* and *endmodule* and module name
- Optional components of a module: Port list, port declarations, data type declarations, instantiation of lower modules and many more

Defining a module in Verilog

```
module <module_name> (<module_terminal_list>);
...
<module internals>
...
...
endmodule
```

module_terminal_list describes the input and output terminals of the module.

Example : Verilog module for adder

```
module RippleCarryAdder4(a,b,cin,s,cout);
  .
  .
  <functionality of the adder>
  .
  .
endmodule
```

Instances

- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters, and I/O interface.
- The process of creating objects from a module template is called instantiation
- The objects are called instances.

Example : 4-bit adder

```
// Define the top-level module called RippleCarryAdder4.
//It instantiates 4 FullAdders.
module RippleCarryAdder4(a,b,cin,s,cout);

    input [3:0] a, b;    input cin; // I/O signals will be explained later
    output [3:0] s;      output cout;

    wire cout1, cout2, cout3;  //wires for connecting instances

    //Four instances of the module FullAdder are created.
    //Each has a unique name.
    //Each instance is passed a set of signals.
    //Notice, that each instance is a copy of
    // the module FullAdder.
    FullAdder fa0(a[0],b[0],cin,s[0],cout1);
    FullAdder fa1(a[1],b[1],cout1,s[1],cout2);
    FullAdder fa2(a[2],b[2],cout2,s[2],cout3);
    FullAdder fa3(a[3],b[3],cout3,s[3],cout);

endmodule
```

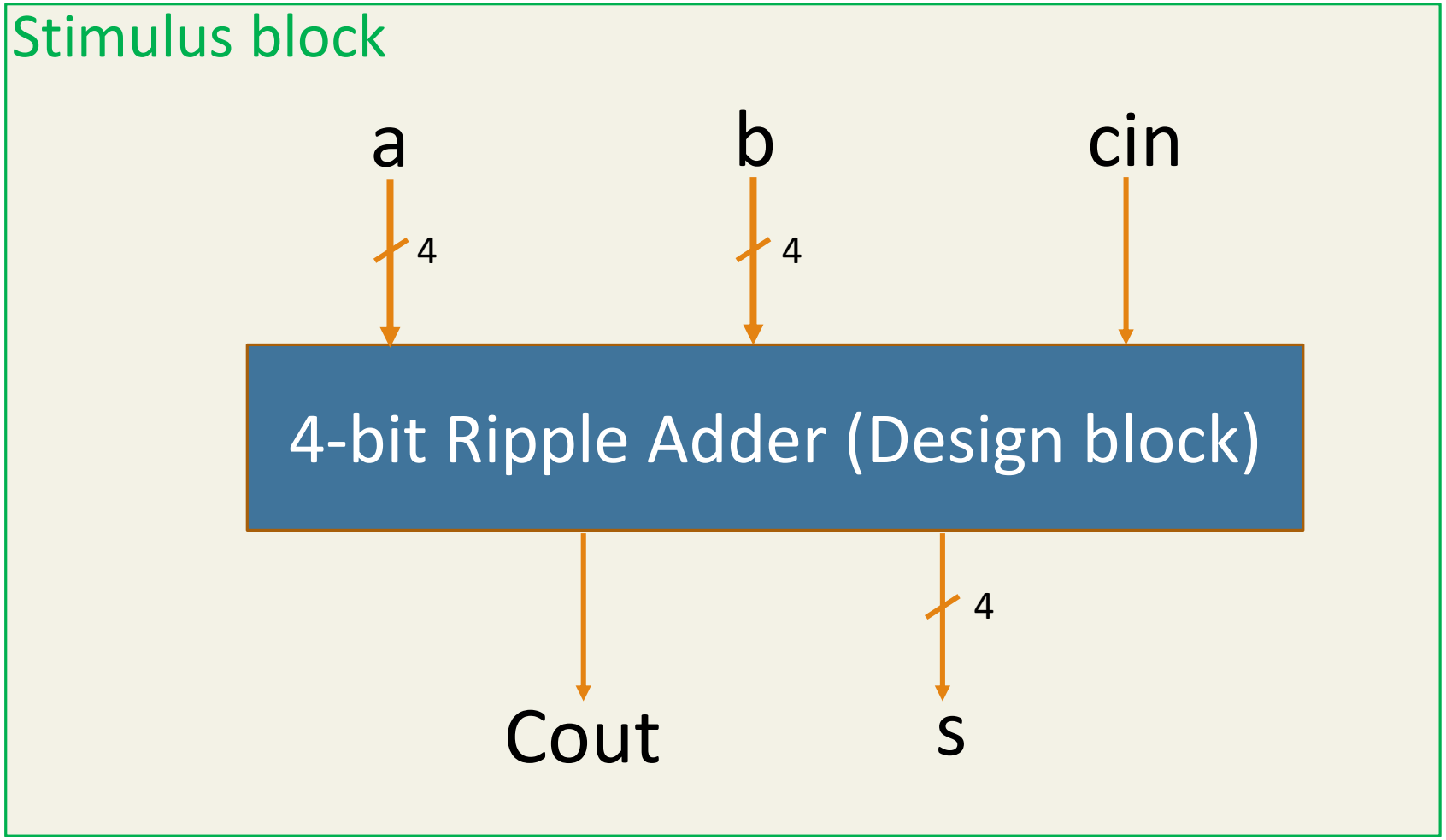
Assume that module **FullAdder** is defined elsewhere in the design as :

```
module FullAdder(a,b,cin,s,cout);
    input a,b,cin; output s,cout;
    assign {cout,s} = a + b + cin;
endmodule
```

Components of a Simulation

- Once a design block is completed, it must be tested.
- The functionality of **the design block** can be tested by applying stimulus and checking results.
- We call such a block the **stimulus block**.
- The stimulus block can be written in Verilog.
- The stimulus block is also commonly called a **test bench**.

Stimulus Block Instantiates Design Block



Example testbed (Stimulus block)

```
module stimulus;
    reg [3:0] a, b;
    wire [3:0] s;
    wire cout;

    // instantiate the design block
    RippleCarryAdder4 rca(a,b,1'b0,s,cout);
    initial
    begin
        a <= 4'b1100; //assign value for a
        b <= 4'b0010; //assign value for b
        #1 //wait for 1 simulation time unit
        $display("%d + %d = %d, %d", a,b,s,cout); //print the result
    end
endmodule
```

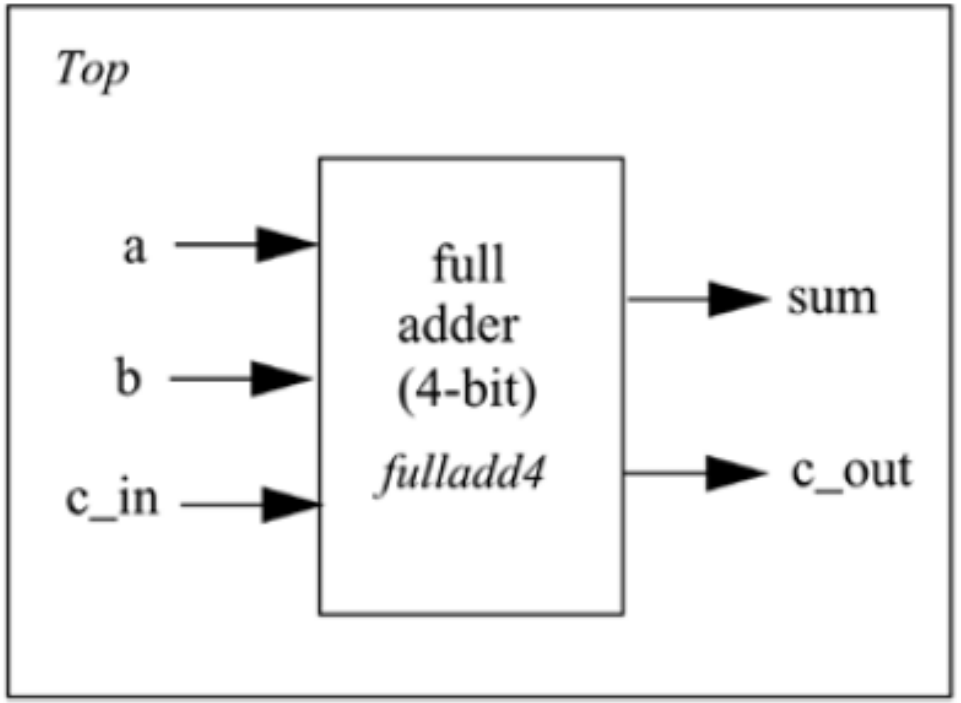
Ports / terminals

- Ports provide the interface by which a module can communicate with its environment.
  - For example, the input/output pins of an IC chip are its ports.
- The environment can interact with the module only through its ports.
- The internals of the module are not visible to the environment.
  - Provides flexibility to the designer : The internals of the module can be changed without affecting the environment as long as the interface is not modified.

List of Ports

- A module definition contains an optional list of ports.
- If the module does not exchange any signals with the environment, there are no ports in the list.

Example : I/O Ports for Top and Full Adder



- module *Top* is a top-level module
  - no communication with the environment
  - no ports
- The module *fulladd4* is instantiated below *Top*
  - Does addition for its environments and hence does communication
  - has ports

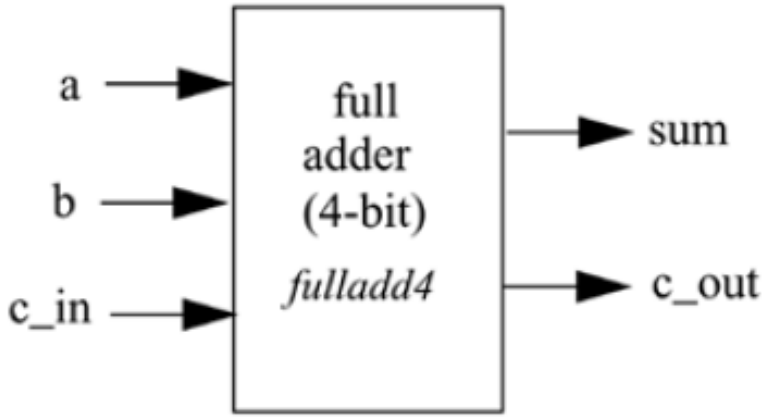
```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```

Port declaration

- All ports in the list of ports must be declared in the module.

Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

Port declaration : an example

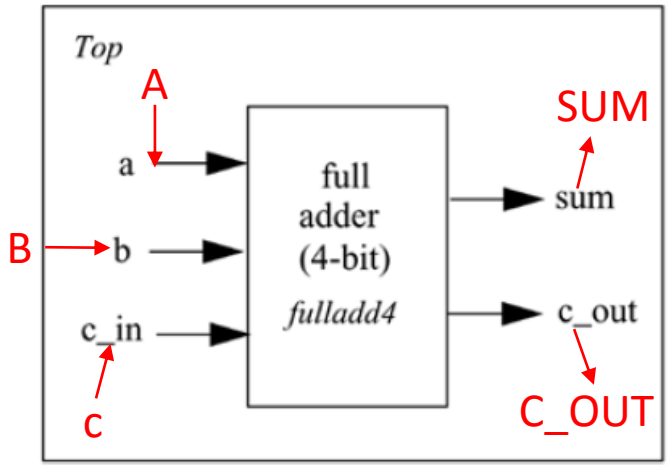



```
module fulladd4(sum, c_out, a, b, c_in);

   //Begin port declarations section
   output[3:0] sum;
   output c_cout;
   input [3:0] a, b;
   input c_in;
   //End port declarations section

   ... <module internals> ...

endmodule
```

- Note that all port declarations are implicitly declared as wire in Verilog.

Connecting ports to external signals

- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.




```
module Top; //Declare connection variables
        reg [3:0] A,B; reg C_IN;
        wire [3:0] SUM; wire C_OUT;
        //Instantiate fulladd4, call it fa_ordered.
        //Signals are connected to ports in order (by positior
        fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
        ... <stimulus> ...
endmodule

module fulladd4(sum, c_out, a, b, c_in);
    output[3:0] sum; output c_cout;
    input [3:0] a, b; input c_in;
    ... <module internals> ...
endmodule
```

Verilog HDL

ICARUS VERILOG

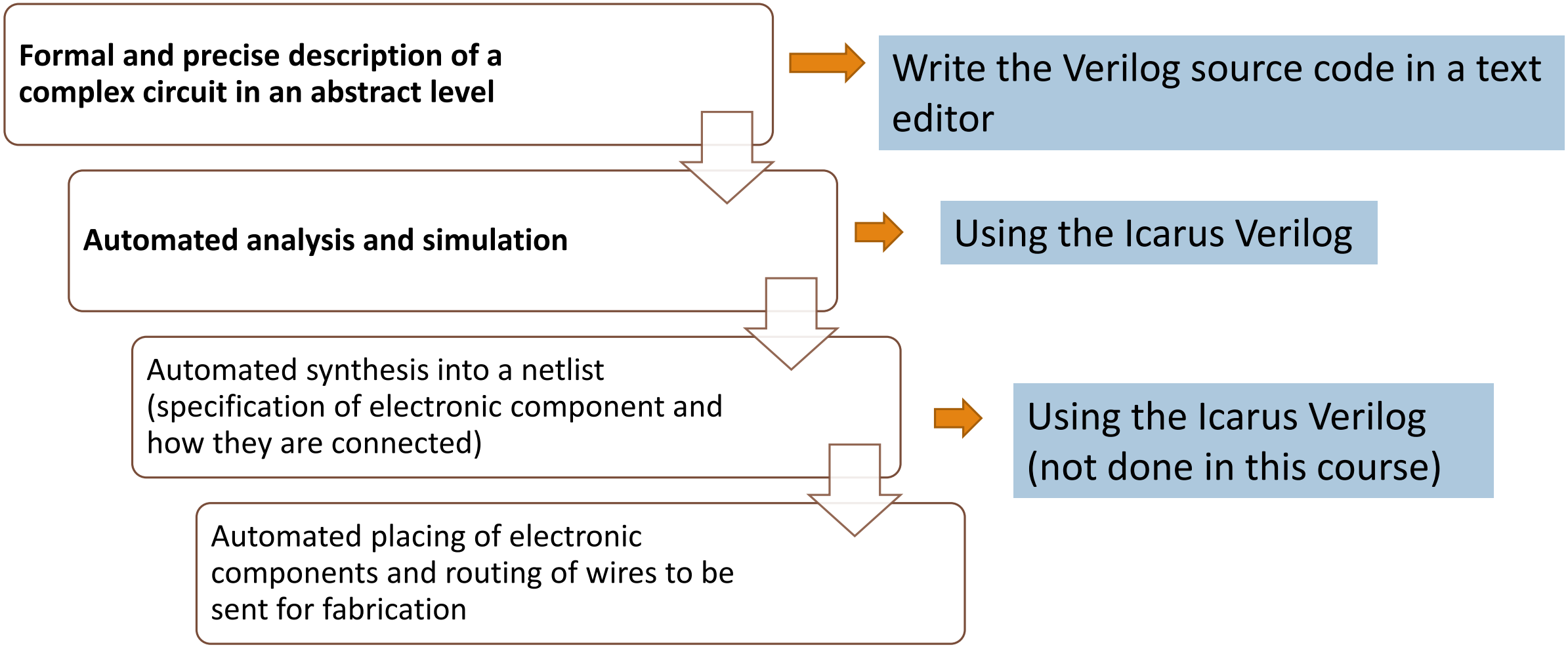
Icarus Verilog

- A Verilog simulation and synthesis tool.
- Operates as a compiler that compiles source code written in Verilog into some target format.
- For simulation, the compiler can generate an intermediate form called vvp assembly.
- For synthesis, the compiler generates netlists.

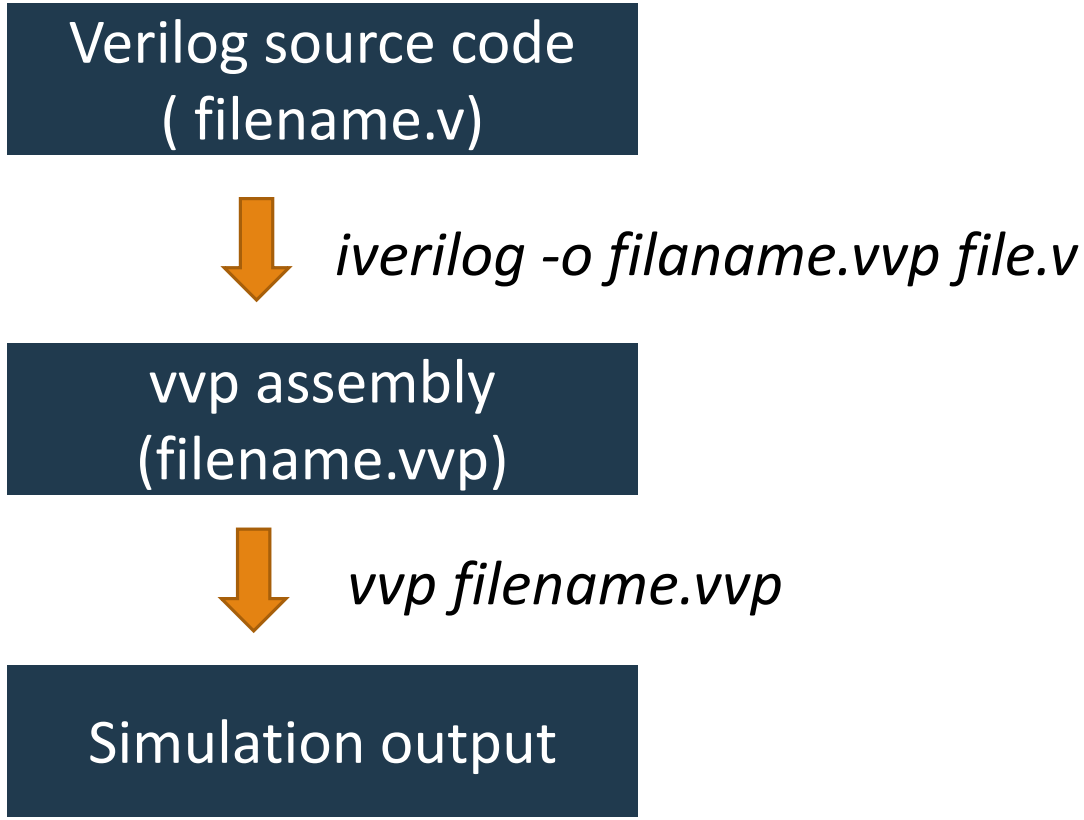
Icarus Verilog

More info @ http://iverilog.icarus.com

How we do?


Formal and precise description of a complex circuit in an abstract level
Write the Verilog source code in a text editor
Automated analysis and simulation
Using the Icarus Verilog
Automated synthesis into a netlist (specification of electronic component and how they are connected)
Using the Icarus Verilog (not done in this course)
Automated placing of electronic components and routing of wires to be sent for fabrication

Using Icarus Verilog for Simulation



Installing Icarus Verilog

In Ubuntu based Linux :

- sudo apt-get install Verilog

For Windows :

- Install the setup at http://bleyer.org/icarus/
- Set the path environmental variable if not automatically set.

A full guide for all operating systems :

- http://iverilog.wikia.com/wiki/Installation_Guide

Lets do some examples ……

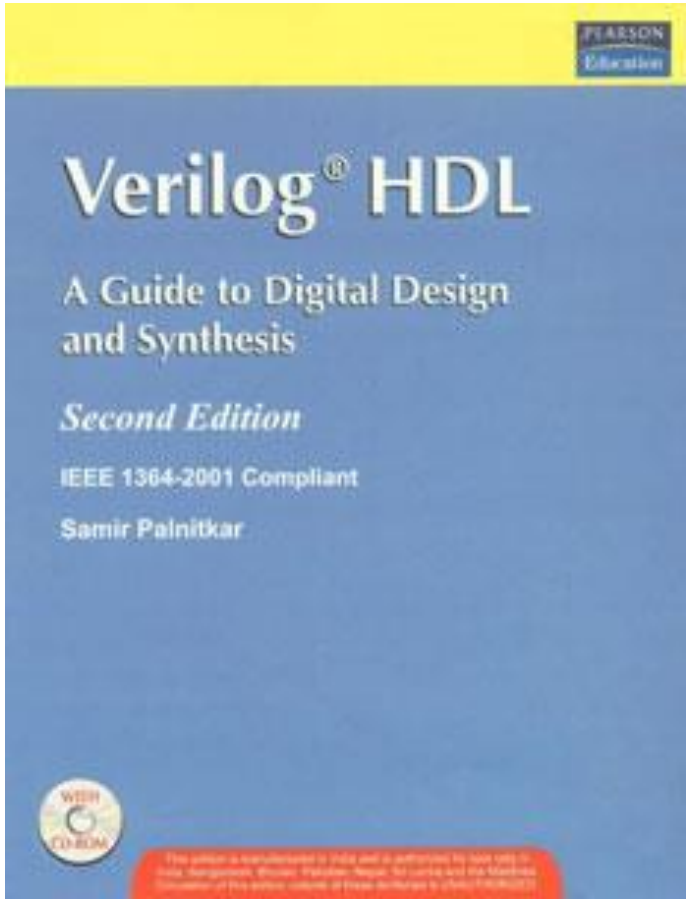
Verilog HDL

GATE-LEVEL MODELING

Reference

Verilog HDL: A Guide to Digital Design and Synthesis, 2e, Samir Palnitkar

Chapters 5

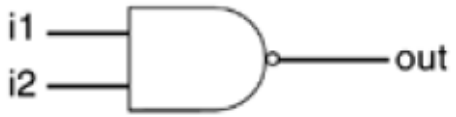


Basic logic gates in Verilog

- Verilog supports basic logic gates as predefined primitives.
- These primitives are instantiated like modules.
- But as they are predefined, a module definition is not needed.

Basic gates

and



nand

or

nor

xor

xnor

Instantiating gates

```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// More than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name
and (OUT, IN1, IN2); // legal gate instantiation
```

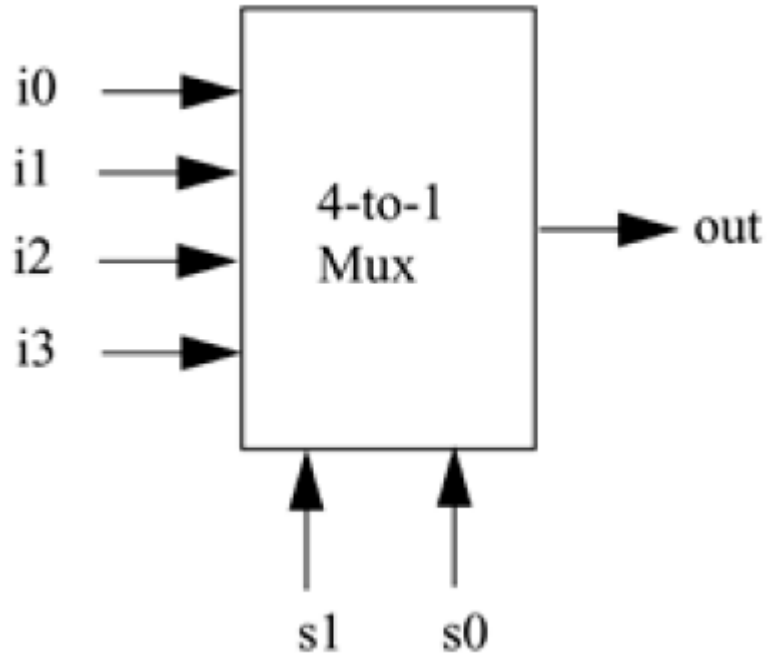
Instantiating gates

- Instance name does not need to be specified for primitives.
- This lets the designer instantiate hundreds of gates without giving them a name.
- More than two inputs can be specified in a gate instantiation.
- Gates with more than two inputs are instantiated by simply adding more input ports in the gate instantiation.

Array of Instances

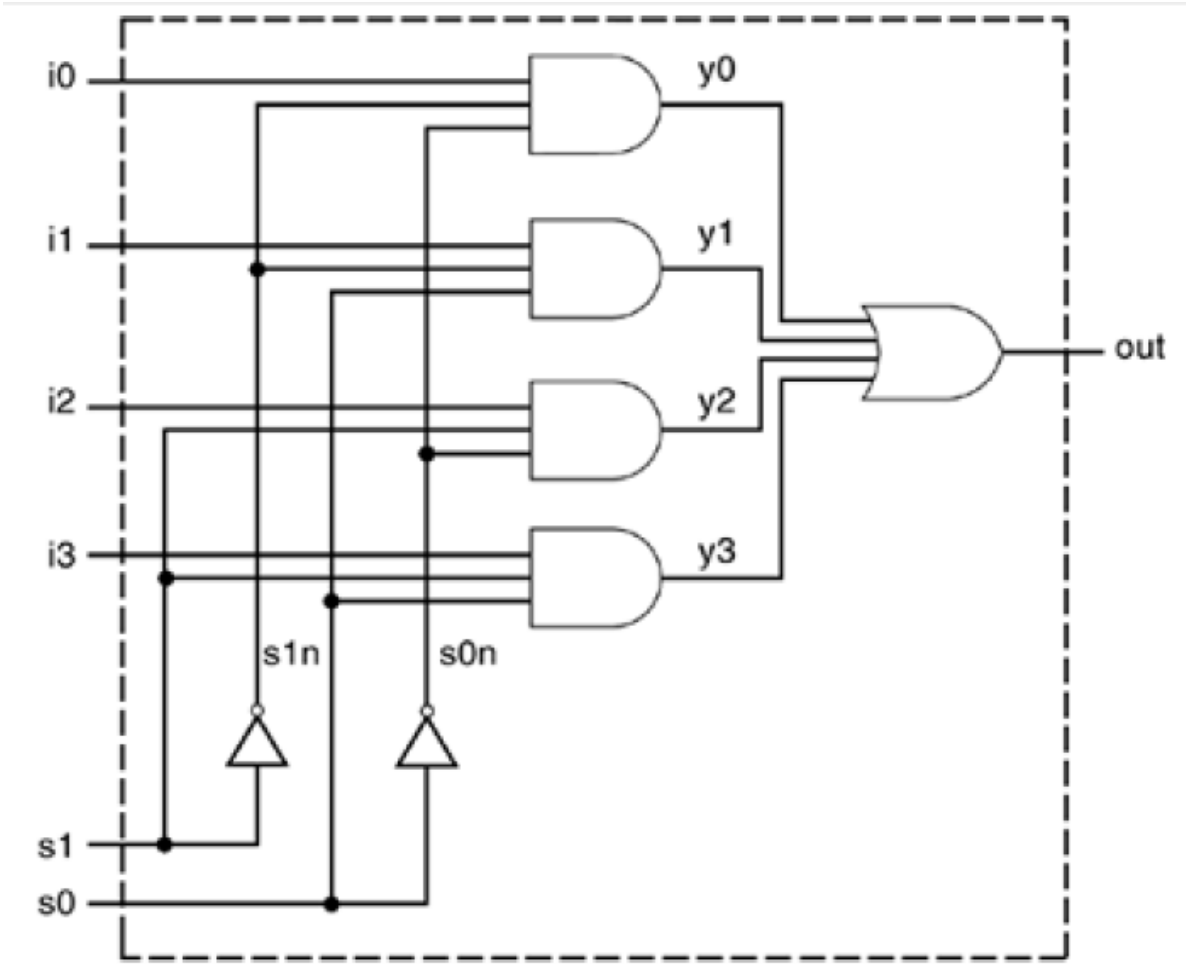
```
wire [7:0] OUT, IN1, IN2;

// basic gate instantiations.
nand n_gate[7:0](OUT, IN1, IN2);

// This is equivalent to the following
8 instantiations
nand n_gate0(OUT[0], IN1[0], IN2[0]);
nand n_gate1(OUT[1], IN1[1], IN2[1]);
nand n_gate2(OUT[2], IN1[2], IN2[2]);
nand n_gate3(OUT[3], IN1[3], IN2[3]);
nand n_gate4(OUT[4], IN1[4], IN2[4]);
nand n_gate5(OUT[5], IN1[5], IN2[5]);
nand n_gate6(OUT[6], IN1[6], IN2[6]);
nand n_gate7(OUT[7], IN1[7], IN2[7]);
```

Example : Gate level multiplexer

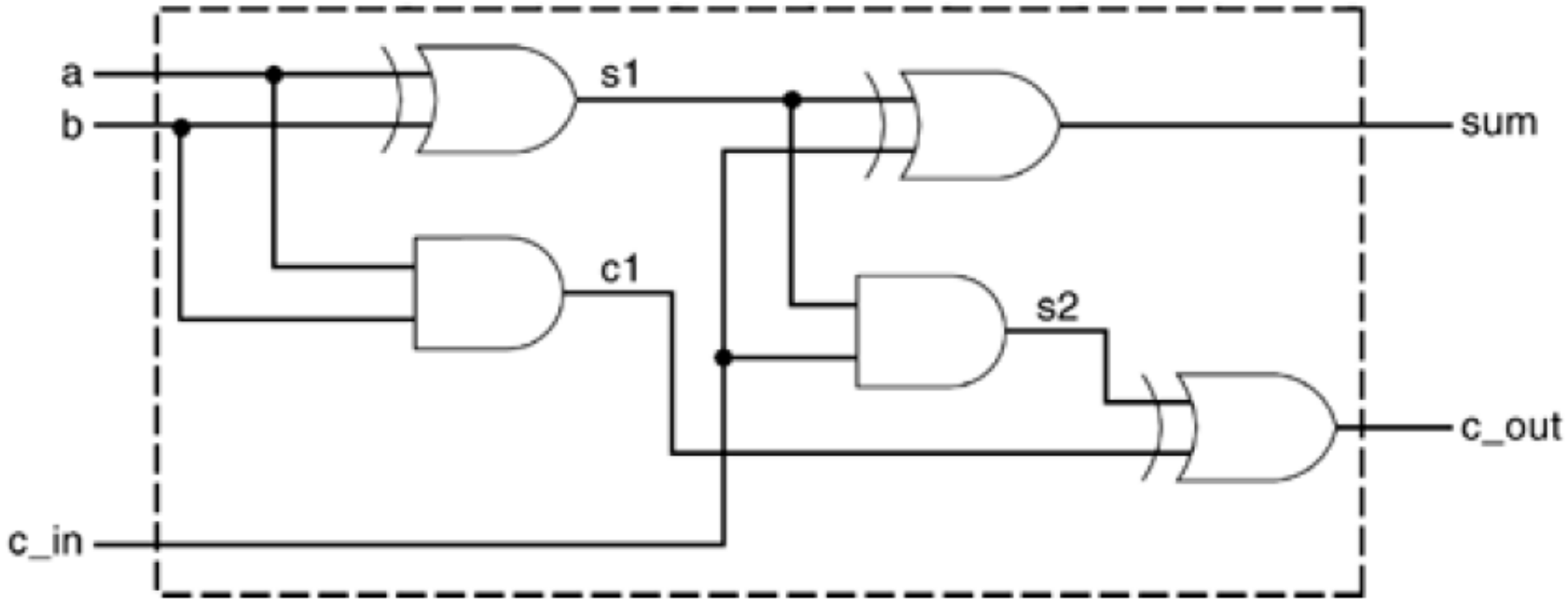


s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Example : Gate level multiplexer

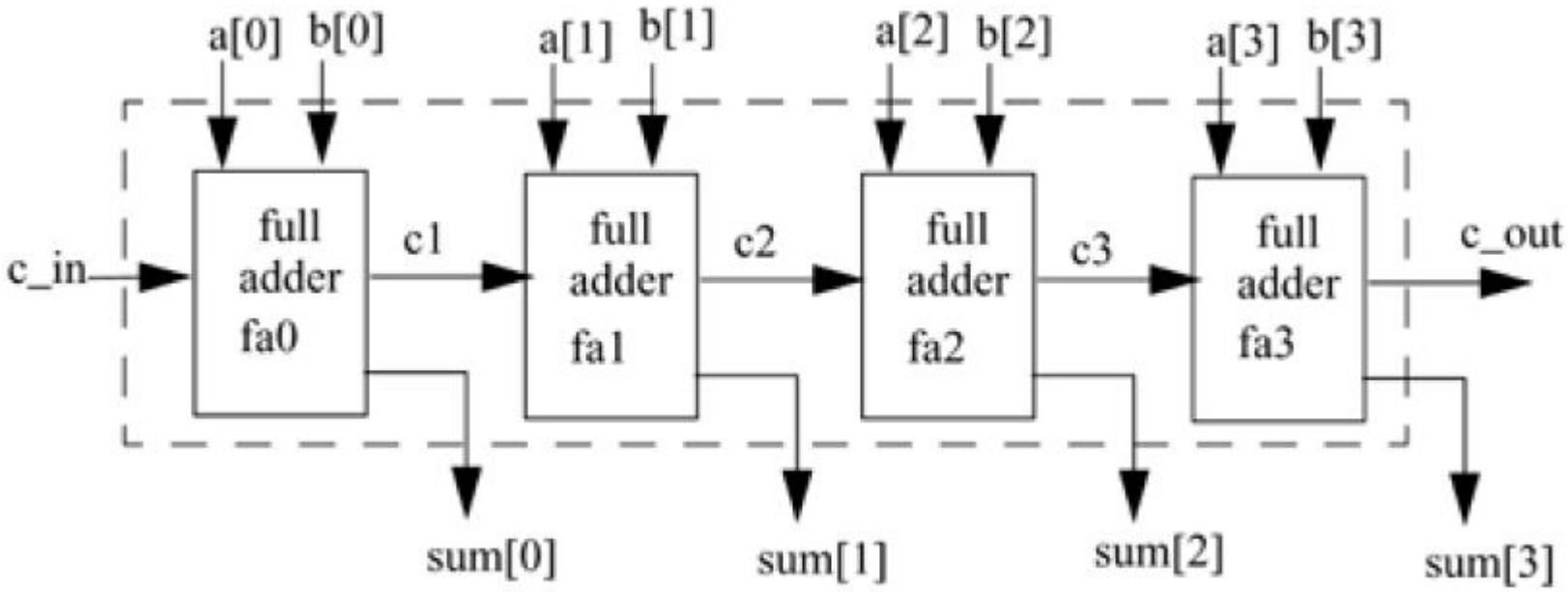


Example : 4-bit adder



1 bit full adder

Example : 4-bit adder



4-bit adder

More examples ……