

Digital System Design (EC- 20005)

Module-1

Basic VLSI System Design

Prof. (Dr.) Amitkumar V. Jha

School of Electronics Engineering (SOEE)

KIIT Deemed to be University



Disclaimer: The contents in this slide have been referred from many sources including [1] and [2] which I do not claim as my own. Some of the content has been modified for easier understanding of the students without any malafide intention. This slide is only for educational purpose strictly, and not for the commercial purpose.

[1] Morris Mano, and Michael D. Ciletti, "Digital Design", Fifth Edition, PHI, 2012.

[2] Samir Palnitkar, "Verilog HDL", Second Edition, Pearson Education, 2003.

Contents

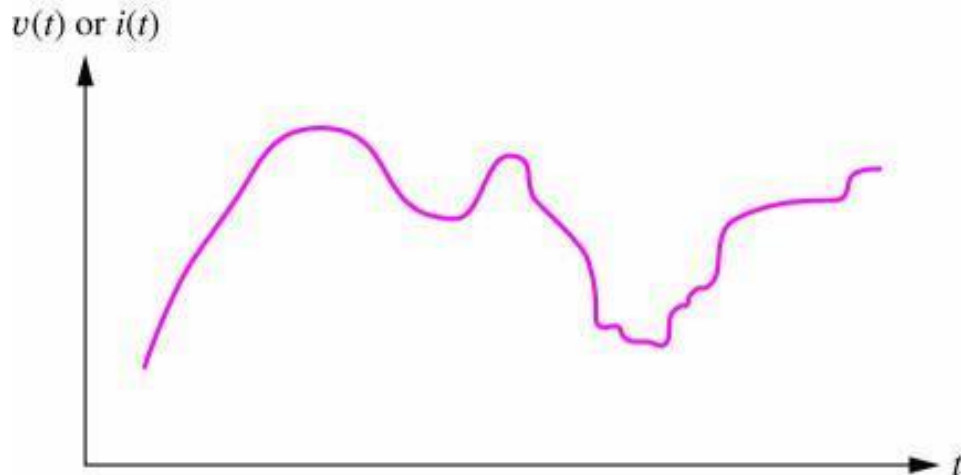
- Introduction to digital systems and VLSI design,
- Moore's Law
- VLSI Design flow, Design hierarchy
- Introduction to Verilog HDL and operators
- Modelling techniques (Gate-level, Data-flow, and Behavioral)
- Example: Half-adder implementation

Some Basic Details....

- There are basically two types of signals.
 - Analog Signal: Analog signals are continuous in time
 - Digital Signal: A signal that is discrete function of time, i.e. which is not a continuous signal, is known as a **digital signal**.

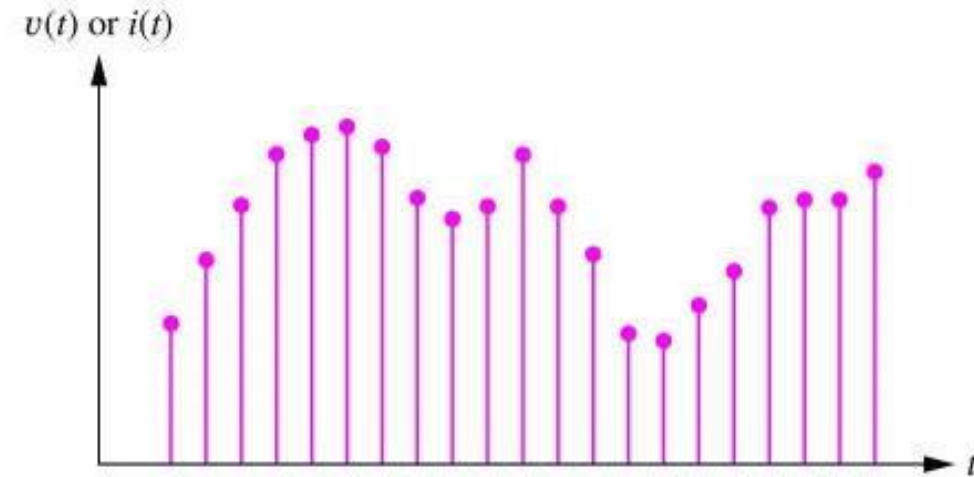
There too exists a mixed signal which has both digital and analog parts.

Analog and Digital Signals



(a)

- Analog signal is a continuous function of time.



(b)

- After digitization, the continuous analog signal becomes a set of discrete values, typically separated by fixed time intervals.

Analog Vs Digital Signals: Key Differences

Analog Signal	Digital Signal
An analog signal is typically represented by a sine wave function. There are many more representations for the analog signals also.	The typical representation of a signal is given by a square wave function.
Analog signals use a continuous range of values to represent the data and information.	Digital signals use discrete values (or discontinuous values), i.e. discrete 0 and 1, to represent the data and information.
The bandwidth of an analog signal is low.	The bandwidth of a digital signal is relatively high.
Due to more susceptibility to the noise, the accuracy of analog signals is less.	The digital signals have high accuracy because they are immune from the noise.
Analog signals use more power for data transmission.	Digital signals use less power than analog signals for conveying the same amount of information.
Analog signals are processed by analog circuits whose major components are resistors, capacitors, inductors, etc.	Digital circuits are required for processing of digital signals whose main circuit components are transistors, logic gates, ICs, etc.
The analog signals are used in land line phones, thermometer, electric fan, volume knob of a radio, etc.	The digital signals are used in computers, keyboards, digital watches, smartphones, etc.

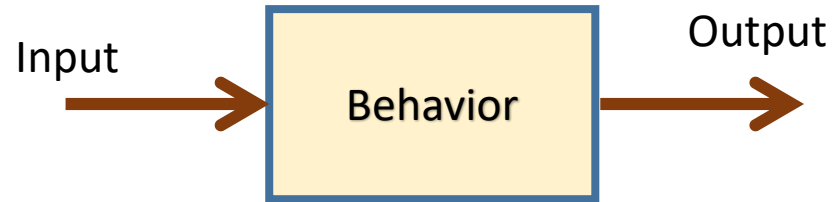
- As we know, digital circuits are required for processing of digital signals whose main circuit components are transistors, logic gates, ICs, etc.

Introduction to Digital System

- System- A set of related components work as a whole to achieve a goal

- A system contains

- Inputs
- Behavior
- Outputs



- Behavior is a function that translates inputs to outputs

Digital systems are designed to store, process, and communicate information in digital form. They are found in a wide range of applications, including process control, communication systems, digital instruments, and consumer products.

Digital Systems: Key Advantages

- Digital representation is very well suited for both numerical and non-numerical information processing
- Easy to design, particularly the automated design and fabrication
- Low cost
- Easy to duplicate similar circuits
- High noise immunity
- Easily controllable by computer
- Adjustable precision
- Complex digital ICs are manufactured with the advent of Microelectronics Technology

Digital Systems: Key Disadvantages

- The physical world is analog

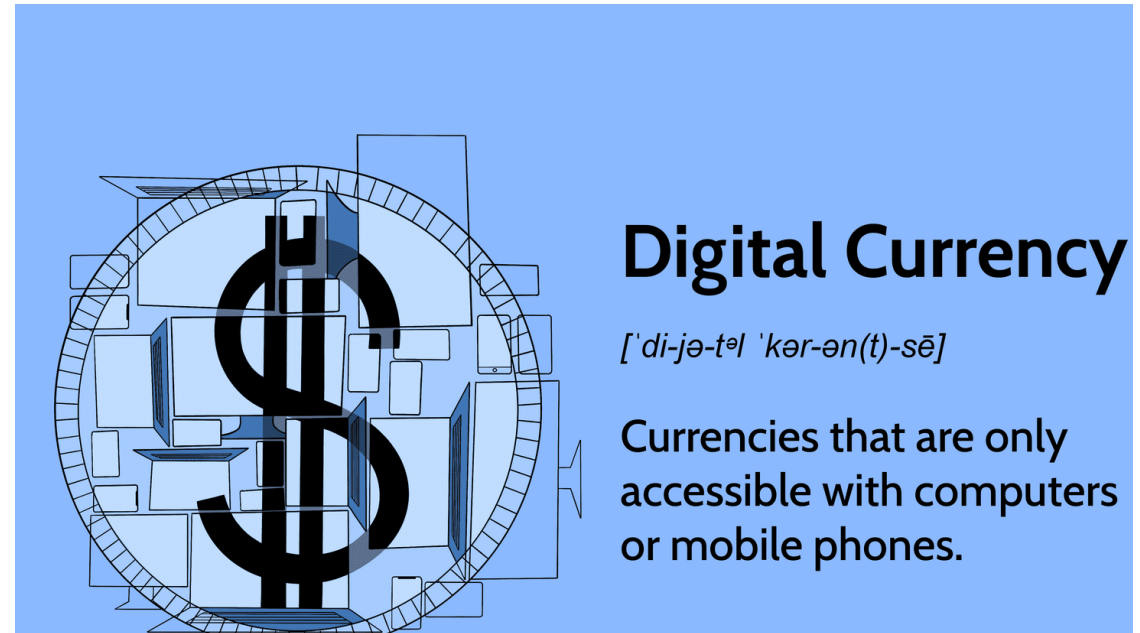
So, needs to convert digital to analog and vice versa to communicate with real world. This makes the digital system expensive and less precision

- Digital abstraction allows analog signals to be ignored and allow some discrete values to be used.

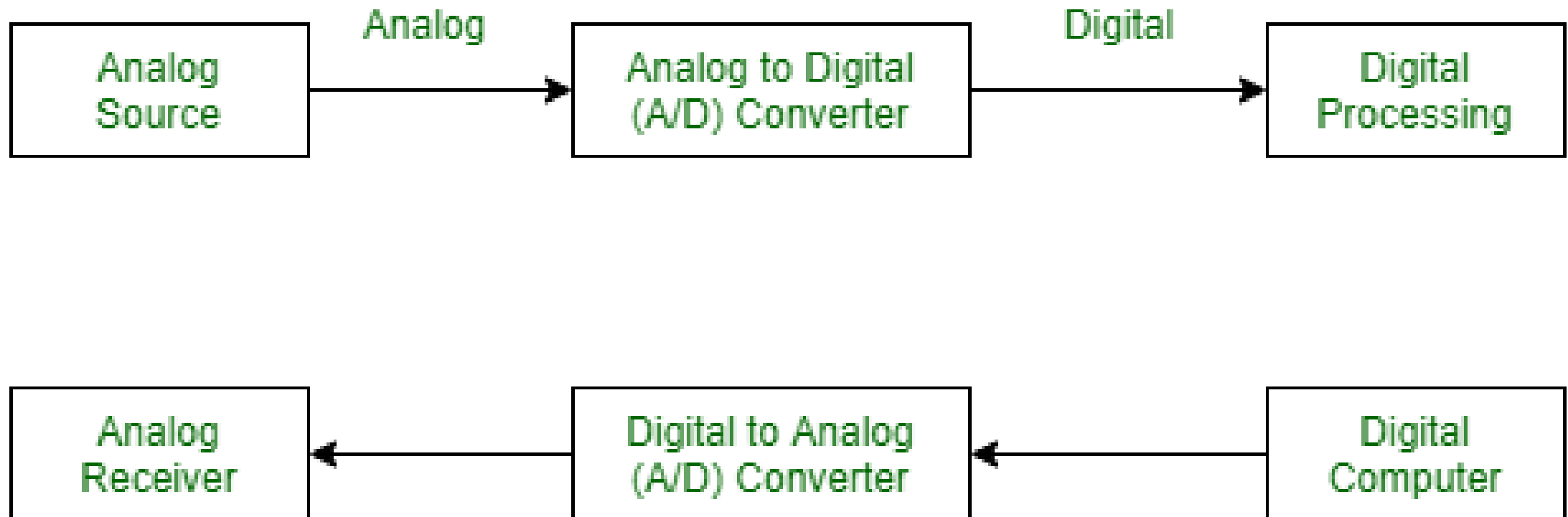
Example is Binary system; only two values are allowed- 1 and 0

- *1 means high value or logic “TRUE”*
- *0 means low value or logic “FALSE”*

Digital System Example



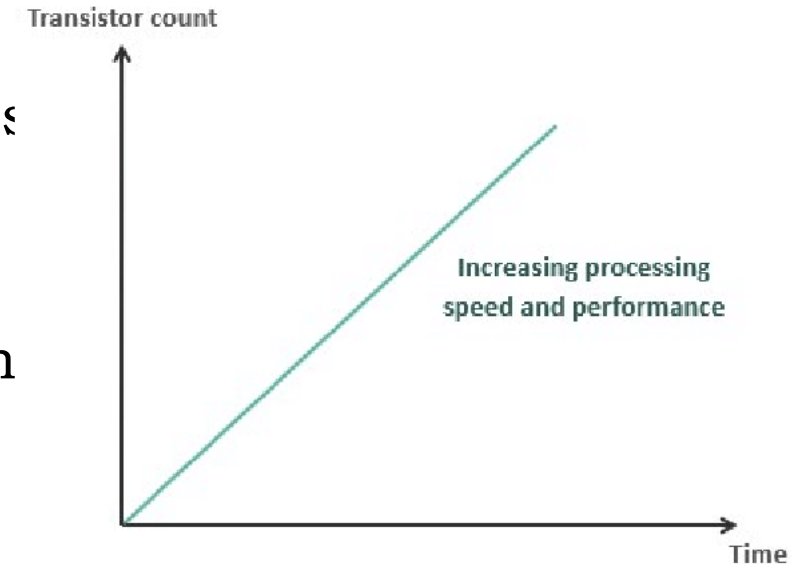
Digital systems: How Does it Work?



MOORE'S LAW

- In 1965, Gordon E. Moore, the co-founder of Intel, made this observation that became known as Moore's Law.¹
- Moore's Law states that the number of transistors on a microchip doubles about every two years, though the cost of computers is halved.
- Another tenet of Moore's Law says that the growth of microprocessors is exponential.

What is Moore's law?

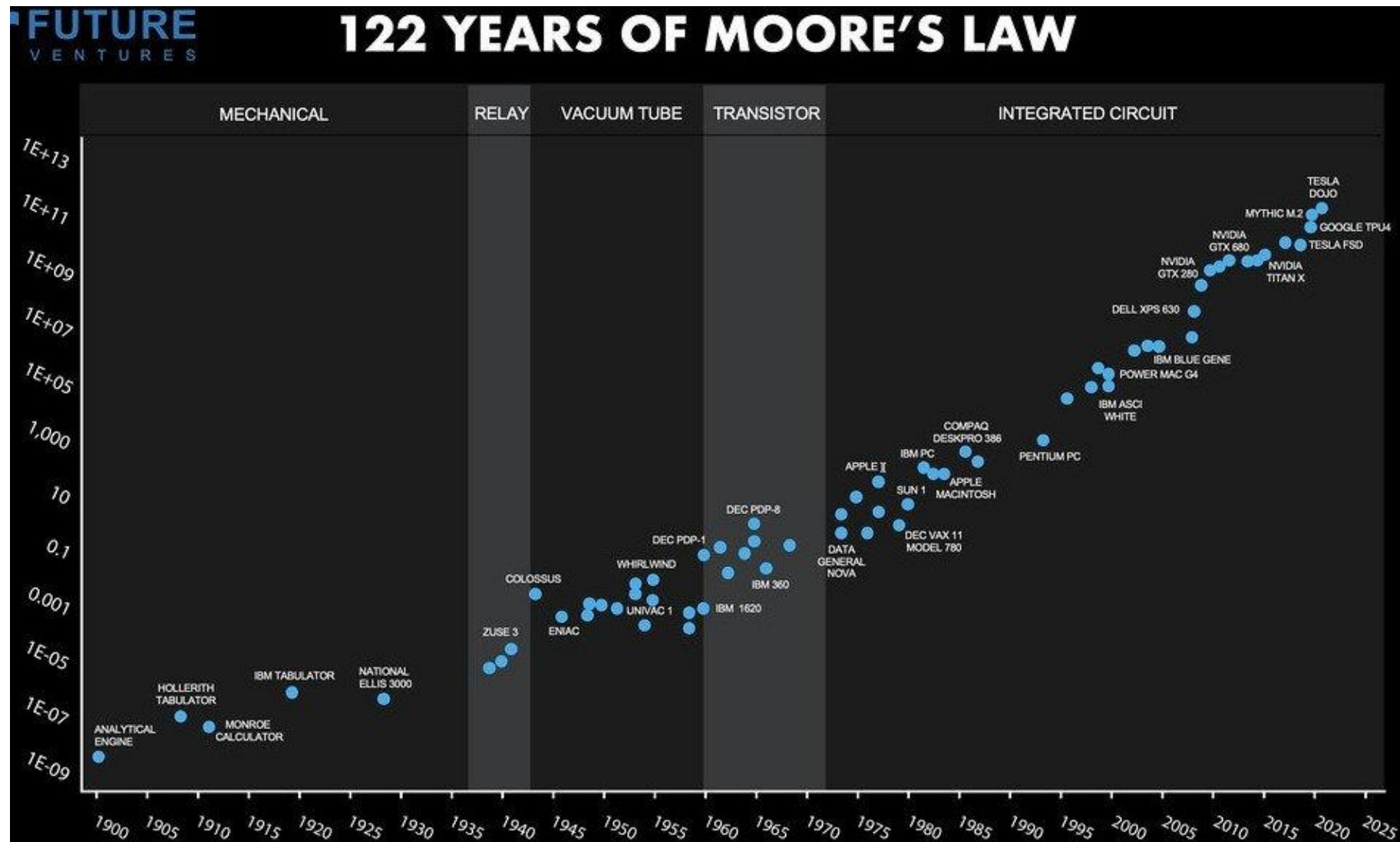


- Let's take the case of Intel Moore's law.

- In 1971, Intel introduced the Intel 4004 with a transistor count of 2250.
- And in 1974, the Intel 8080 processor came with the ability of 6,000 transistors.
- Two years later, Intel introduced the Intel 8085 processor with 6,500 transistors in 1976.
- In 1978, the Intel 8086 came with a transistor count of 29,000.
- Then, the Intel 8051 came in 1980 with 50,000 transistors, followed by the Intel 80186 with 55,000 transistors in 1982.
- Finally, in 1985, the Intel 80386 had a 275,000 transistor count.

This came to be known as the Intel Moore's Law. From the above data, it is evident that there have been increments in the transistor counts over the years with a period of two years.

Graph of Moore's Law



Nearly 60 Years Old and Still Strong

(Dr.) Amitkumar V. Jha, School of Electronics Engineering, KIIT-DU

Is Moore's Law Coming to an End?

- According to expert opinion, Moore's Law is estimated to end sometime in the 2020s.[5University of Arizona, Department of Physics. "Quantum Error Correction Codes," Page 1.](#)
- **What this means?**
- This means that computers are projected to reach their limits because transistors will be unable to operate within smaller circuits at increasingly higher temperatures. This is due to the fact that cooling the transistors will require more energy than the energy that passes through the transistor itself.

Specification of Digital Systems

- Specification of system is the description of its function and other characteristics required for it
 - Speed
 - Cost
 - Power
- Design can be improved at the expense of worsening one or both of the others
- These trade-offs exist at entry level in the system design every sub-piece and component
- A designer must make the trade-offs necessary to achieve the function within the constraints

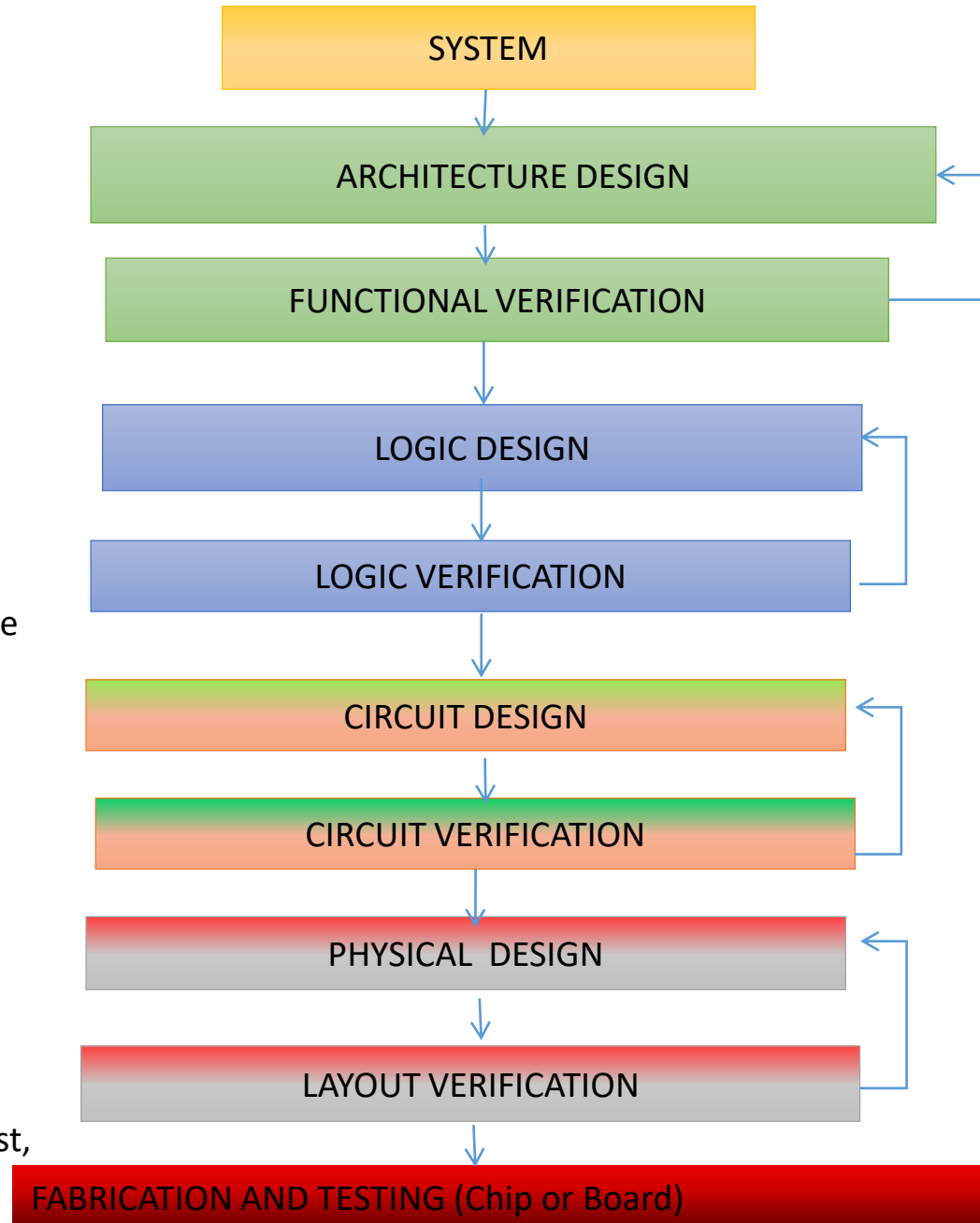
VLSI Design Flow

BEHAVIOURAL REPRESENTATION (Flow Graph, Pseudo Code)

LOGIC (GATE-LEVEL) REPRESENTATION (Gate Wirelist, Netlist)

CIRCUIT REPRESENTATION

LAYOUT REPRESENTATION (Transistor list, layout)



- The Figure provides a simplified view of the VLSI design flow, taking into account the various representations, or abstractions of design - behavioral, logic, circuit and mask layout.
- Note that the verification of design plays a very important role in every step during this process. The failure to properly verify a design in its early phases typically causes significant and expensive re-design at a later stage, which ultimately increases the time-to-market.
- Although top-down design flow provides an excellent design process control, in reality, there is no truly unidirectional top-down design flow.

Some of the classical techniques for reducing the complexity of IC design are: hierarchy, regularity, modularity and locality.

Steps of the VLSI Design Flow

1. Specification and Architecture

This is the initial stage where the requirements for the IC or SoC are defined, including the functionality, performance, power consumption, and area constraints.

2. Design Entry

This involves creating a high-level design representation of the IC or SoC using a hardware description language (HDL) such as Verilog or VHDL.

3. Functional Verification

This step involves verifying that the high-level design meets the specifications by simulating it using a hardware simulator.

4. **Synthesis**

In this step, the high-level design is translated into a gate-level netlist, which is a collection of logic gates and flip-flops that implement the design.

5. **Design Optimization**

The gate-level netlist is optimized for various design constraints such as power consumption, timing, and area.

6. **Physical Design**

This step involves placing the gates and routing the interconnections to meet the timing and area constraints.

7. Design Rule Check (DRC)

The physical design is checked against a set of design rules to ensure it is manufacturable.

8. Layout Verification

The physical design is verified using simulations to ensure that it meets the specifications.

9. Tape-out

Once the physical design is verified, the final design is sent to the fabrication facility for manufacturing.

10. Testing

After the IC or SoC is fabricated, it is tested to ensure that it meets the specifications.

Design Hierarchy

- The use of hierarchy, or “divide and conquer” technique involves dividing a module into sub- modules and then repeating this operation on the sub-modules until the complexity of the smaller parts becomes manageable.
- This approach is very similar to the software case where large programs are split into smaller and smaller sections until simple subroutines, with well-defined functions and interfaces, can be written.

Implementation of Digital System

- Implementation means how the system is constructed from smaller and simpler components called **modules**
- The modules can vary from simple gates to complex processors
- Digital system follows some hierarchical implementation

Hierarchical Implementation

Hierarchical implementation basically is of following three types:

1. Modular Design
2. Top-Down Design
3. Bottom-up Design

1. **Modular design**

- Divide and conquer
- Modules are designed and built separately and then assembled to form the system
- Simplifies implementation and debugging
- One of the major factors for cost-effectiveness of digital systems

Top-Down Design

- Starts at the top (root) and works down by successive refinement
- decomposes the system into subsystems and the subsystem into simpler and smaller subsystems and so on
- stops when subsystem can be realized by directly available module

Bottom-up Design

- Starts at the leaves and puts pieces together to build up the design
- subsystems are assembled to form a bigger subsystem
- stops when required functional specification is achieved

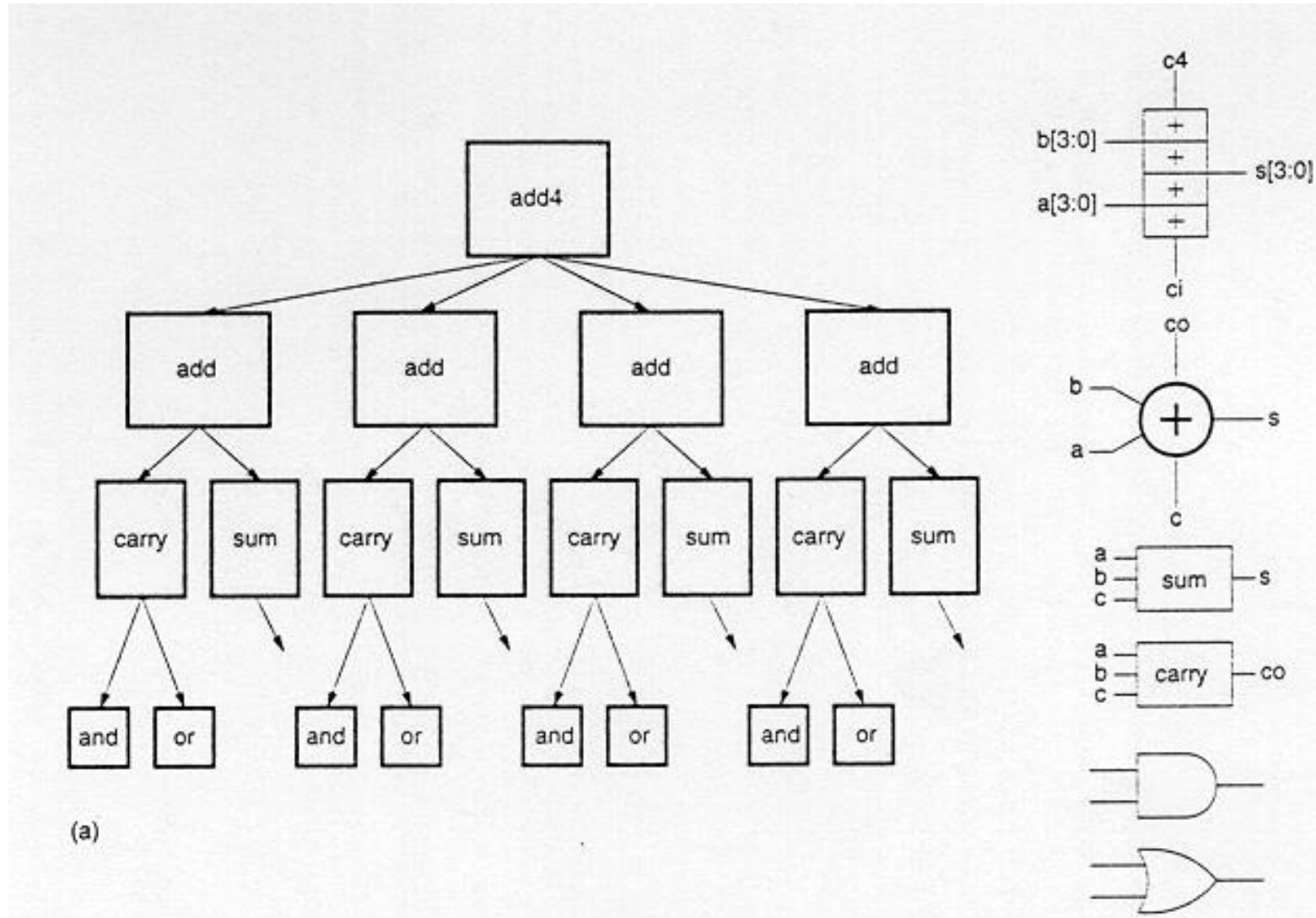
Disadvantages of Top-down or Bottom-up design

- No systematic procedure exists for decomposition (in case of Top-down) or composition (in case of Bottom-up)
- Depends on the expertise of the designer

Which is better?

- In practice both Top-down and Bottom-up design approaches are needed and used
 - Need top-down divide and conquer to handle the complexity
 - Need bottom-up because in a well designed system the structure is influenced by what primitives are available

Design Hierarchy



Structural decomposition of a four-bit adder circuit, showing the hierarchy down to gate level

- Fig. shows the structural decomposition of a CMOS four-bit adder into its components.
- The adder can be decomposed progressively into one-bit adders, separate carry and sum circuits, and finally, into individual logic gates.
- At this lower level of the hierarchy, the design of a simple circuit realizing a well-defined Boolean function is much more easier to handle than at the higher levels of the hierarchy.

Hardware Description Languages (HDL)

- HDLs are indeed similar to programming languages but not exactly the same.
- We utilize a programming language to build or create software, whereas we use a hardware description language to describe the structure, behaviour and timing of electronic circuits, and most commonly, digital logic circuits.
- In addition to their use in circuit design, HDLs serve the purpose of simulating the circuit and verifying its response.
- We utilize HDLs for designing processors, motherboards, CPUs (i.e., computer chips), as well as various other digital circuitry.
- Many HDLs are available, but the most popular HDLs so far are
 1. Verilog and
 2. VHDL.
- Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit(IC) designers.
- Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

Verilog vs VHDL

1. Verilog: Verilog stands for verification logic. It is used to model and stimulate the Digital circuits Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs).

Syntax:

```
module module_name(inputs,output)
//statements
endmodule
```

VHDL: VHDL stands for **Very High-speed Integrated Circuit Hardware Description Language (VHSIC)**. It is used to design digital circuits. It is often used to design complex Digital circuits such as Microprocessors and Digital Signal Processors.

Syntax:

```
library ieee;
use ieee.std_logic_1164.all;
entity Circuit_name is
    Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          out1 : out STD_LOGIC);
end Circuit_1;

-----

architecture Behavioral of Circuit_name is
begin
    // statements
end Behavioral;
```

Modelling techniques

When you think of any sequential or combination circuit, what modelling aspects come to mind?

- Schematic
- Truth table
- Logical expression

Similarly, when it's about Verilog HDL, three modelling aspects come to mind:

- **Structural (lowest level modelling)**
- **Data-flow**
- **Behavioral (highest level modelling)**

Structural Modelling Style

- The structural modelling style is the lowest level of abstraction obtained using logic gates.
- Similar to schematic or circuit diagrams of the digital circuit, Verilog uses primitive gates to compile and synthesize the program.
- The language supports multiple gates such as ***and***, ***or***, ***nand***, ***xor***, ***nor***, and ***xnor***.
- You can also use tri-state gates and multiple-output gates such as ***bufif1***, ***bufif0***, ***notif1***, ***notif0***, ***not***, and ***buf***.

Syntax
example:

```
and (out, in1, in2, in3,...); // an and gate  
or (y, a, b, c, d); // a 4-input or gate
```

Data Flow Modelling Style

- The data flow is a medium level abstraction, which is achieved by defining the data flow of the module.
- You can design the module by defining and expressing input signals which are assigned to the output, very much similar to logical expressions.

Syntax:

```
module and_gate(a,b,out);  
    input a,b;  
    output out;  
    assign out = a&b;  
endmodule
```


Behavioral Modelling Style

- Behavioural modelling is the highest of level abstraction that completely depends on the circuit behavior or on the truth table.
- It is utilized for complex circuits such as pure combinational or sequential circuits.
- A module can be implemented in terms of the desired design algorithm without looking into the hardware details using structured procedures (like **always** and **initial**), conditional statements (like **if** and **else**) and multi way branching (like **case**, **case x** and **case z**).
- A module developed using behavioural modelling contains ***initial*** or ***always*** statements, which are executed concurrently.
- The procedural statements in the module are executed sequentially.
- At time=0, both the ***initial*** and ***always*** will execute and then, ***always*** statements run for the remaining time.

Here's the syntax:

```
always [timing control] procedural_statements;  
initial [ timing control] procedural_statements;
```

```

module sample_circuit(
    output x,
    output y,
    input a,
    input b,
    input c
);

    wire e;

    and AND1 (e,a,b);
    not NOT1 (y,c);
    or OR1 (x,e,y);

endmodule

```

```

module test_sample_circuit( );

    reg a,b,c;
    wire x,y;

    sample_circuit CKT1 (x,y,a,b,c);

    initial
    begin
        a = 0; b = 0; c = 0;
        #10
        a = 0; b = 0; c = 1;
        #10
        a = 0; b = 1; c = 0;
        #10
        a = 0; b = 1; c = 1;
        #10
        a = 1; b = 0; c = 0;
        #10
        a = 1; b = 0; c = 1;
        #10
        a = 1; b = 1; c = 0;
        #10
        a = 1; b = 1; c = 1;
        #10
        $finish;
    end
endmodule

```

Operators

Arithmetic Operators

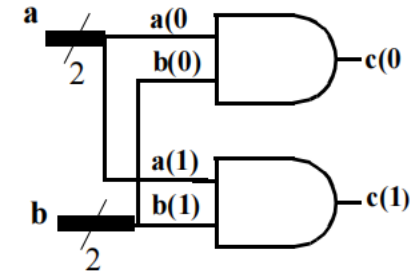
- These perform arithmetic operations. The + and - can be used as either unary (-z) or binary (x-y) operators.
- Operators
 - + (addition)
 - - (subtraction)
 - * (multiplication)
 - / (division)
 - % (modulus)

Relational Operators

- Relational operators compare two operands and return a single bit 1 or 0. These operators synthesize into comparators.
- **Operators**
 - < (less than)
 - <= (less than or equal to)
 - > (greater than)
 - >= (greater than or equal to)
 - == (equal to)
 - != (not equal to)

Bit-wise Operators

- Bit-wise operators do a bit-by-bit comparison between two operands.
- Operators
 - \sim (bitwise NOT)
 - $\&$ (bitwise AND)
 - $|$ (bitwise OR)
 - \wedge (bitwise XOR)
 - $\sim\wedge$ or $\wedge\sim$ (bitwise XNOR)

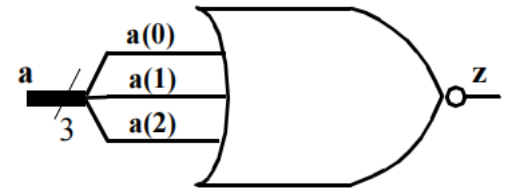


Logical Operators

- Logical operators return a single bit 1 or 0.
- They are the same as bit-wise operators only for single bit operands.
- They can work on expressions, integers or groups of bits, and treat all values that are nonzero as “1”.
- Logical operators are typically used in conditional (if ... else) statements since they work with expressions.
- Operators
 - ! (logical NOT)
 - && (logical AND)
 - || (logical OR)

Reduction Operators

- Reduction operators operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators above.
- Operators
 - $\&$ (reduction AND)
 - $|$ (reduction OR)
 - $\sim\&$ (reduction NAND)
 - $\sim|$ (reduction NOR)
 - \wedge (reduction XOR)
 - $\sim\wedge$ or $\wedge\sim$ (reduction XNOR)



Shift Operators

- Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).
- Operators
 - << (shift left)
 - >> (shift right)

Concatenation Operator

- The concatenation operator combines two or more operands to form a larger vector.
- Operators
 - { }(concatenation)

Replication Operator

- The replication operator makes multiple copies of an item.
- Operators
 - {n{item}} (n fold replication of an item)

Conditional Operator: “?”

Conditional operator is like those in C/C++. They evaluate one of the two expressions based on a condition. It will synthesize to a multiplexer (MUX).

Operators

- (cond) ? (result if cond true):
(result if cond false)

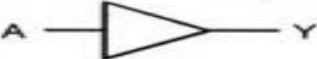
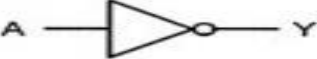






Verilog Reserved Key Words

always	endfunction	input	signed
and	endgenerate	integer	task
assign	endmodule	join	time
automatic	endprimitive	localparam	real
begin	endspecify	module	realtime
case	endtable	nand	reg
casex	endtask	negedge	unsigned
casez	event	nmos	wait
deassign	for	nor	while
default	force	not	wire
defparam	forever	or	
design	fork	output	
disable	function	parameter	
edge	generate	pmos	
else	genvar	posedge	
end	include	primitive	
endcase	initial	specify	
endconfig	inout	specparam	

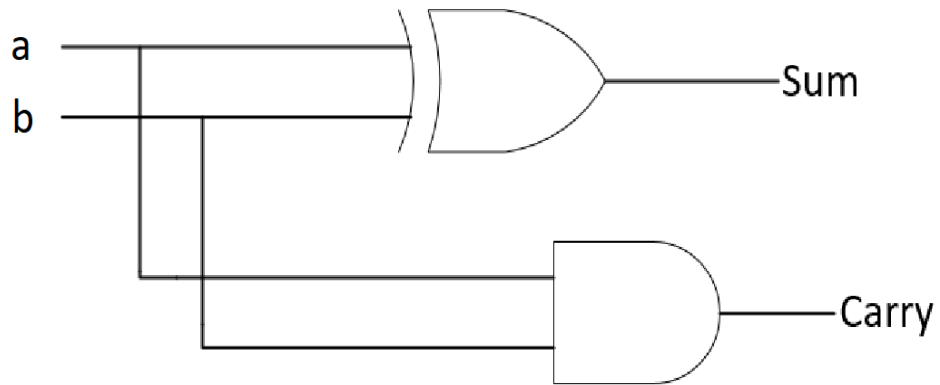
Logic gates

Keywords for Logic Gates:

buf , not, and,
nand, or, nor, xor,
xnor.

Logic function	Logic symbol	Truth table	Boolean expression															
Buffer		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	Y	0	0	1	1	$Y = A$									
A	Y																	
0	0																	
1	1																	
Inverter (NOT gate)		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0	$Y = \bar{A}$									
A	Y																	
0	1																	
1	0																	
2-input AND gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	$Y = A \cdot B$
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
2-input NAND gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0	$Y = \overline{A \cdot B}$
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
2-input OR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1	$Y = A + B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
2-input NOR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	$Y = \overline{A + B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
2-input EX-OR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0	$Y = A \oplus B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
2-input EX-NOR gate		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1	$Y = \overline{A \oplus B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Example: Half-adder implementation



a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half Adder Verilog Code: Gate level Modelling

- “module” is the basic building block in Verilog.
- In Verilog, a module is declared by the keyword module.
- A corresponding keyword endmodule must appear at the end of the module definition.
- Each module must have a module_name, which is the identifier for the module, and a port list, which describes the input and output terminals of the module

```
module half_adder(sum, carry, a, b);  
    input a, b;  
    output sum, carry;  
    xor sum1(sum, a, b);  
    and carry1(carry, a, b);  
endmodule
```

The module name

- The module name, formally called an identifier should best describe what the system is doing. Each identifier in Verilog, including module names must follow these rules:
- It can be composed of letters, digits, dollar sign (\$), and underscore characters (_) only.
- It must start with a letter or underscore.
- No spaces are allowed inside an identifier.
- Upper and lower case characters are distinguished (Verilog is case sensitive)
- Reserved keywords cannot be used as identifiers.

Half Adder Verilog Code: Dataflow Modelling

```
module half_adder(sum, carry, a, b);  
    input a,b;  
    output sum, carry; // sum and carry  
    assign sum = a^b;  
    assign carry = a&b ;  
endmodule
```

- In the data flow modeling style, we use the **assign** statements directly to describe the logic equations that determine the outputs (sum and carry) based on the inputs (a and b).
- The assign statements directly express the relationships between inputs and outputs, making it clear how the logic is interconnected.

Half Adder Verilog Code: Behavioral Modelling

```
module half_adder(sum, carry, a, b);  
    input a,b;  
    output reg sum, carry;  
    always @*  
    begin  
        sum_reg = a ^ b;  
        carry_reg = a & b;  
    end  
    assign sum = sum_reg;  
    assign carry = carry_reg;  
endmodule
```

- **always** is a **procedural block** is used to describe the behaviour of the circuit.
- **always@* begin** is used to define a procedural block in Verilog.
- @* is known as "wildcard" or "star" operator.
- It is used to infer sensitivity to all the signals referenced within the block.
- This means that whenever any of the signals used inside the block change, the block will execute, allowing you to describe the logic that depends on those signals.

Testbench for half-adder

```
module half_adder_testbench;
    reg a,b;
    wire sum,cout;
    half_adder h1(sum, carry, a, b); // circuit to be tested
    initial
    begin
        a = 0, b=0; //test case-1
        #10
        a = 0, b=1; //test case-2
        #10;
        a = 1, b=0; //test case-3
        #10;
        a=1, b=1; //test case-4
        $finish;
    end
endmodule
```

- Verilog data-type **reg** can be used to model hardware registers since it can hold values between assignments.

A **wire** represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block.

```
1  initial
2      [single statement]
3
4  initial begin
5      [multiple statements]
6  end
```

NOTE: Testbench is same for all types of modelling

Data Flow vs Behavioral Modeling Style

Data Flow Modelling

- In data flow modeling, we use continuous assignments (**assign statements**) to directly express the relationships between inputs and outputs.
- The logic is expressed as a set of equations that determine the outputs based on the inputs.
- These assignments are continuously evaluated in parallel, and whenever the inputs change, the outputs are updated immediately.
- Data flow modeling is more suitable for describing combinational logic.

Behavioral Modeling

- In behavioral modeling, we use procedural blocks like **always** or **initial** to describe the behavior of the circuit using procedural statements (like if, case, etc.).
- The logic is described as a sequence of operations or conditions, similar to a programming language.
- Behavioral modeling can capture both combinational and sequential logic, making it suitable for more complex designs that involve state changes.

Further reading...

VHDL vs Verilog

VHDL	Verilog
Strongly typed	Weakly typed
Easier to understand	Less code to write
More natural in use	More of a hardware modeling language
Wordy	Succinct
Non-C-like syntax	Similarities to the C language
Variables must be described by data type	A lower level of programming constructs
Widely used for FPGAs and military	A better grasp on hardware modeling
More difficult to learn	Simpler to learn

End of Module-1

You are also advised to refer the textbook for practice and further understanding.