Hardware Description Language

by

Dr. Dubacharla Gyaneshwar

Department of Computer Science and Engineering

Indian Institute of Information Technology Raichur (IIIT-R)

R slot – EE121 – 2 credit course

Evaluation Breakdown

- **Evaluation Criteria**:
  - ❑*80% Attendance or more*: 5% (*)
  - ❑*1 Quiz*: 25% --- **completed**
  - ❑*Lab-exam/practical/project*: 20% -- **will be given in this week**
  - ❑*End-course*: 40%
  - ❑Assignments: 10% **(*) – part 1 is done and part 2 will be given**

- Dates of the exams or any changes will be notified in advance in class or through an email.

HDL – Design Flow

Hardware description languages need the ability to

- Describe
- Simulate at
  - Behavioural
  - Structural
  - and mixed

  level.
- and to synthesize (structure from behaviour).
- Timing
- Concurrency
- Hardware Simulation process which involves:
  - Analysis
  - Elaboration
  - and Simulation
- Simulation proceeds in two distinct phases
  - Signal update
  - Selective re-simulation

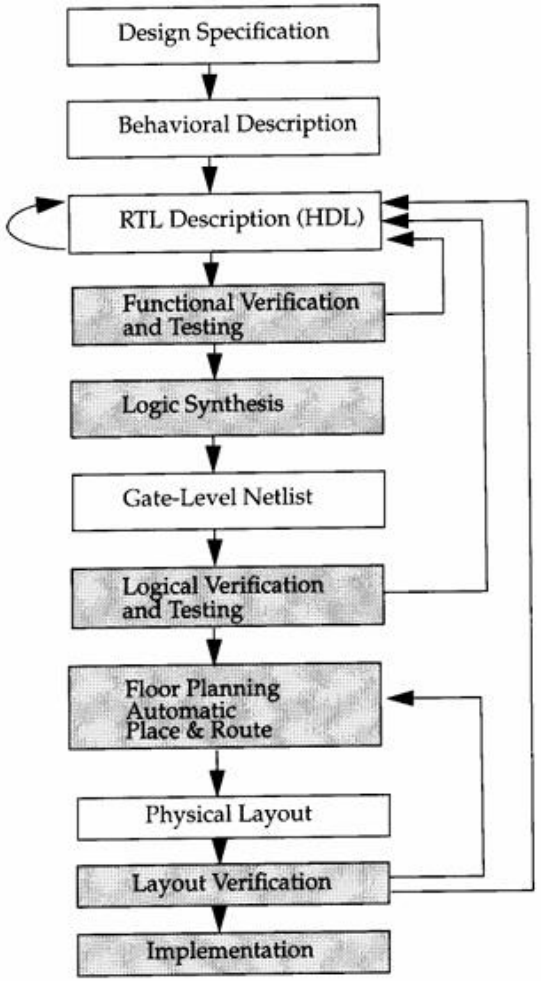
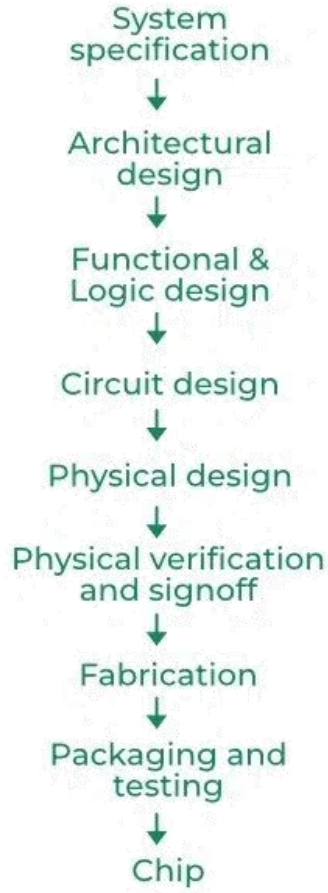
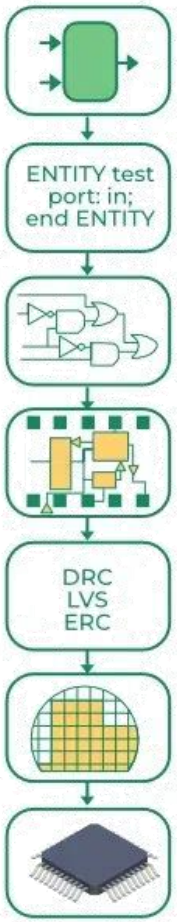


Figure 1-1 Typical Design Flow

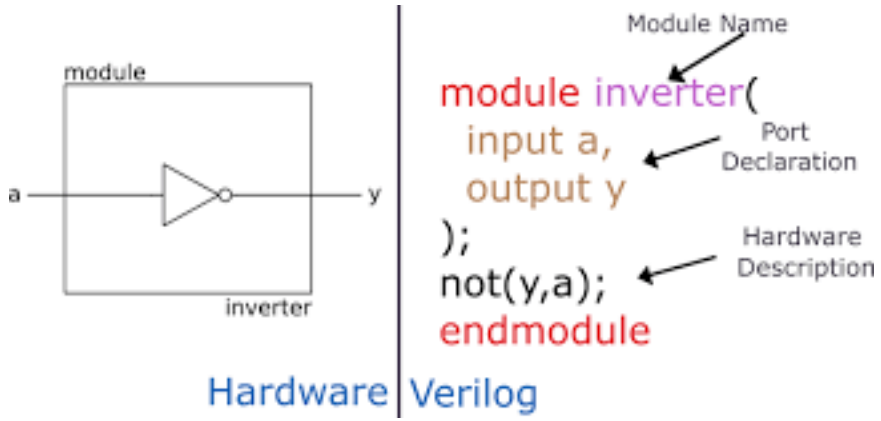




Hardware Description Languages (HDLs)

Hardware Description Languages are used for:

- Description of
  - Interfaces
  - Behaviour
  - Structure
- Test Benches
- Synthesis



Languages:

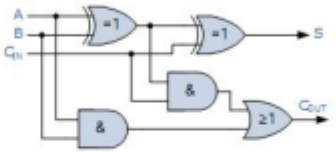
- **Verilog** – created in 1984 by Philip Moorby of Gateway Design Automation (merged with Cadence)
  - IEEE Standard 1364-1995/2001/2005
  - Based on the C language
  - Verilog-AMS – analog & mixed-signal extensions
  - IEEE Std. 1800-2012 "System Verilog" – Unified hardware design, spec, verification
- **VHDL** = VHSIC Hardware Description Language

  (VHSIC = Very High Speed Integrated Circuits)
  - Developed by DOD from 1983 – based on ADA language
  - IEEE Standard 1076-1987/1993/2002/2008
  - VHDL-AMS supports analog & mixed-signal extensions

Different levels

Different Levels of Abstraction in Verilog HDL

- **Gate-Level modeling**:
  - Verilog HDL supports built-in primitive gates modeling.




```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;
      wire s1,c1,c2,c3;
      xor(s1,x,y);
      xor(sum,cin,s1);
      and(c1,x,y);
      and(c2,y,cin);
      and(c3,x,cin);
      or(cout,c1,c2,c3);
endmodule
```

- **Dataflow modeling:**
  - Mainly used to describe combinational circuits. The basic mechanism used is the continuous assignment.

`assign` [delay] LHS_net = RHS_expression;

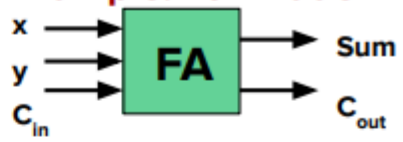
```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;

      assign {cout, sum} = x + y + cin;
endmodule
```

- **Behavioral modeling**:
  - Used to describe complex circuits (primarily sequential). Mechanisms: **`initial`** and **`always`** statements.

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output reg sum, cout;

    always @ (x or y or cin) begin
        {cout, sum} = a + b + cin;
    end
endmodule
```

Behavioral: describes what should be done in a module
- **module contents are C-like assignment statements, loops**

Example: Full Adder



- Boolean expressions and continuous assignments

- **Structural** connects components hierarchically.
- **Dataflow** uses concurrent signal assignment.
- **Behavioral** uses a process block to describe logic behavior.

HDL Modeling styles

- **In HDL like Verilog and VHDL, there are three primary modeling styles used to describe digital circuits:**
  - ❑***Gate-Level/Structural Modeling***
    - ❑*The lowest-level abstraction in HDL, where circuits are described using basic logic gates like AND, OR, NOT, NAND, etc.*
    - ❑*Uses primitive gate instantiations (built-in gate primitives in Verilog).*
    - ❑*Similar to drawing a circuit using actual gates in a schematic.*
    - ❑*Best suited for: Understanding hardware behavior at the fundamental level.*
    - ❑*Limitations: Tedious for complex designs.*

```
module AND_Gate (output Y, input A, B);
  and G1(Y, A, B); // Using Verilog's built-in AND gate
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_Gate_Structural is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Y : out STD_LOGIC);
end AND_Gate_Structural;

architecture Structural of AND_Gate_Structural is
    component AND_Gate
        Port ( A : in STD_LOGIC;
               B : in STD_LOGIC;
               Y : out STD_LOGIC);
    end component;
begin
    U1: AND_Gate port map(A => A, B => B, Y => Y);
end Structural;
```

HDL Modeling styles

- **In HDL like Verilog and VHDL, there are three primary modeling styles used to describe digital circuits:**
  - ❑ ***Dataflow Modeling***
    - ❑ *Uses Boolean expressions and continuous assignments (assign) to describe the circuit's functionality.*
    - ❑ *More abstract than gate-level modeling but still closely tied to the circuit's logic.*
    - ❑ *Uses assign in Verilog or concurrent signal assignments in VHDL.*
    - ❑ *Best suited for: Designs that can be described using simple equations, such as adders, multiplexers, etc.*
    - ❑ *Limitations: Does not describe sequential circuits like flip-flops directly.*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_Gate_Dataflow is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Y : out STD_LOGIC);
end AND_Gate_Dataflow;

architecture Dataflow of AND_Gate_Dataflow is
begin
    Y <= A AND B;  -- Dataflow expression
end Dataflow;
```

```
module AND_Gate (output Y, input A, B);
  assign Y = A & B;  // Boolean expression for AND gate
endmodule
```

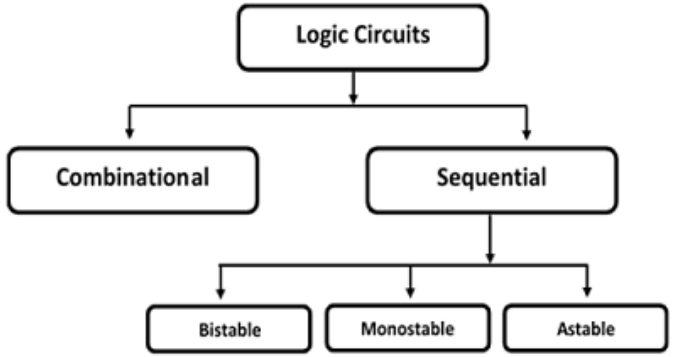
HDL Modeling styles

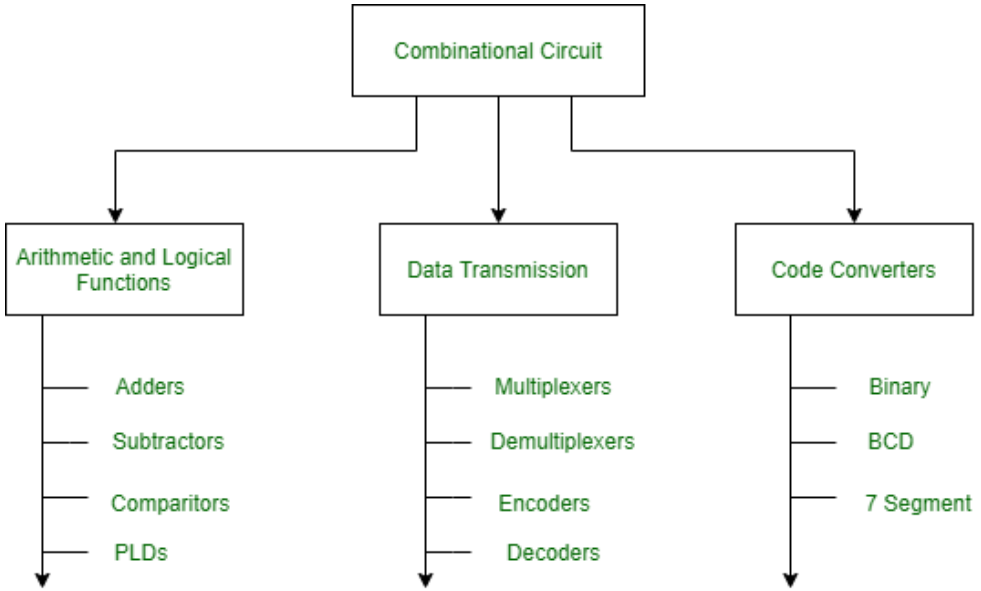
- **In HDL like Verilog and VHDL, there are three primary modeling styles used to describe digital circuits:**
  - ***Behavioral Modeling***
    - *The highest level of abstraction, describing what the circuit does rather than how it is implemented.*
    - *Uses procedural blocks (always, initial), conditional statements (if, case), and loops.*
    - *Allows designing complex circuits like FSMs, ALUs, and processors.*
    - *Best suited for: Complex logic, sequential circuits, and algorithmic design.*
    - *Limitations: May not be directly synthesizable in some cases.*

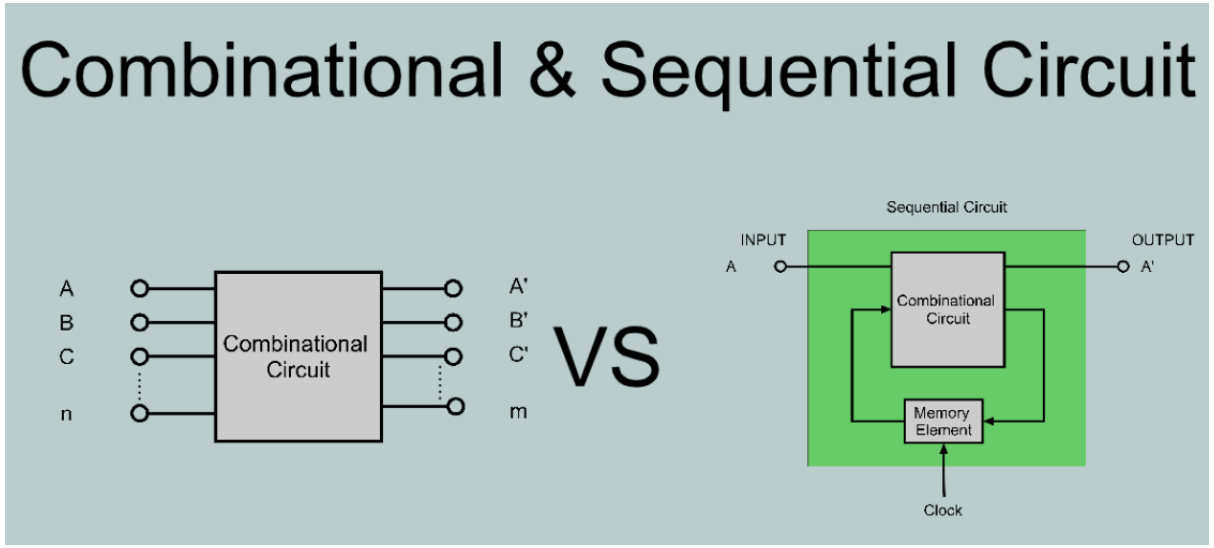
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_Gate_Behavioral is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Y : out STD_LOGIC);
end AND_Gate_Behavioral;

architecture Behavioral of AND_Gate_Behavioral is
begin
    process(A, B)
    begin
        if (A = '1' and B = '1') then
            Y <= '1';
        else
            Y <= '0';
        end if;
    end process;
end Behavioral;
```

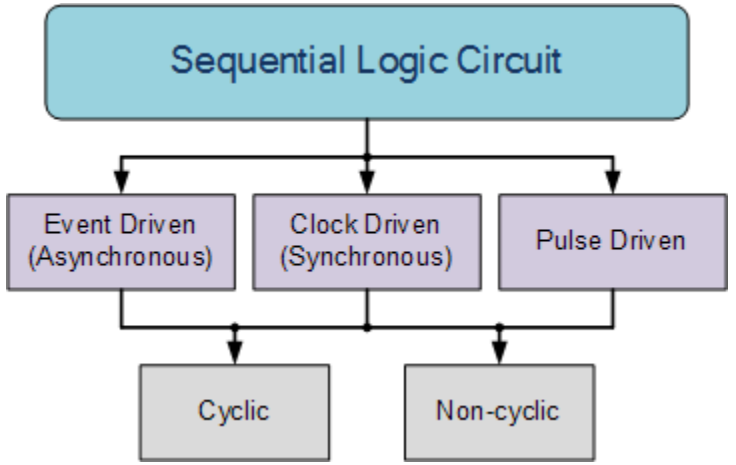
```
module AND_Gate (output reg Y, input A, B);
  always @(*) begin
    Y = A & B;  // Using procedural block
  end
endmodule
```

Fundamental Blocks in Digital Circuits

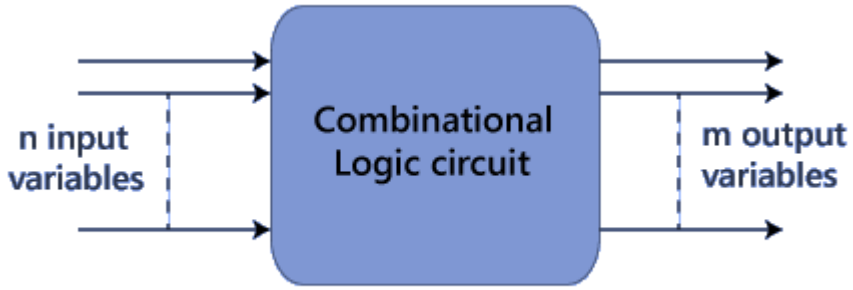








Combinational circuits



Block diagram of a combinational circuit

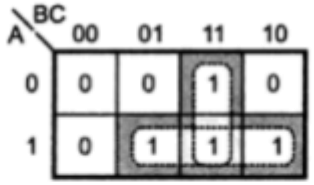
- Binary Adders
- Binary Subtractors
- Multiplexers (MUX)
- Demultiplexers (DEMUX)
- Encoders
- Decoders
- Comparators

Example : Design a combinational logic circuit with three inputs , the output is at logic 1 when more than one inputs are at logic 1.

Truth table

Inputs			Output
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

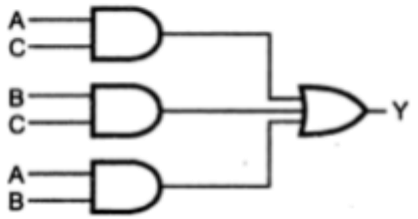
K map Simplification



Boolean Expression

Y=AC + BC + AB

Logic Diagram



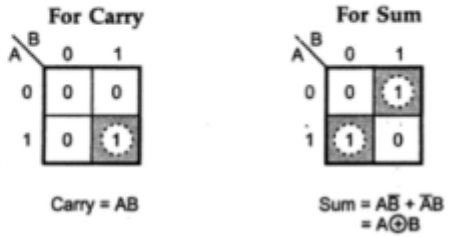
Combinational circuits

Block diagram of Half adder

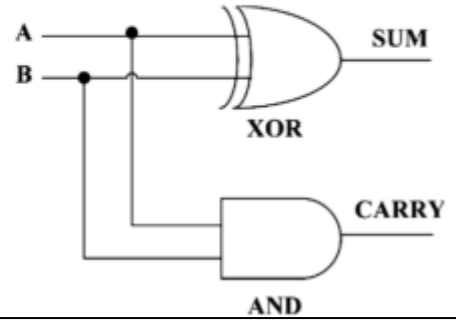
Truth table of Half adder

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

K-map simplification

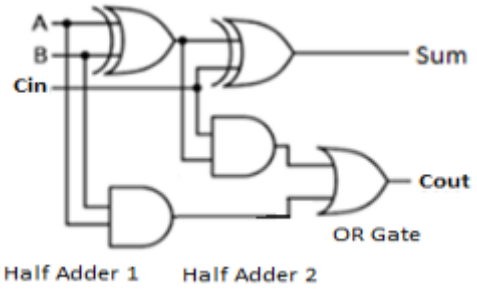


Logic diagram



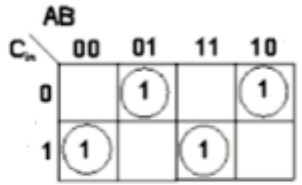
Truth table

Inputs			Outputs	
A	B	Cin	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

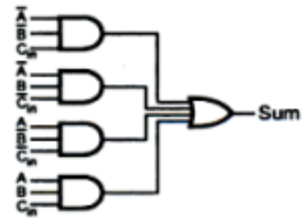


K-map simplifications

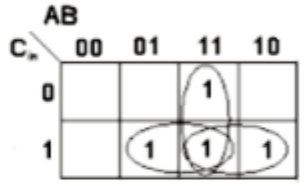
For Sum



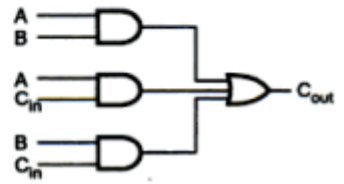
$Sum = \bar{A}\bar{B}Cin + \bar{A}B\bar{C}in + A\bar{B}\bar{C}in + ABCin$



For Cout



$Cout = AB + BCin + ACin$

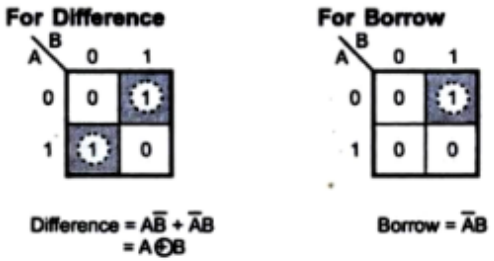


Combinational circuits

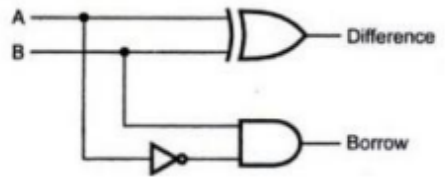
Truth table of Half adder

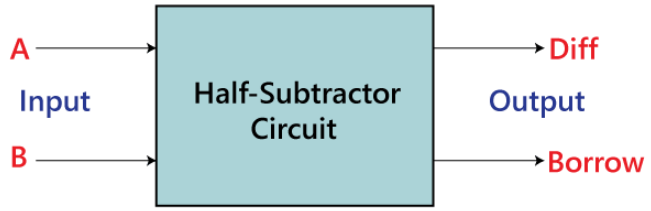
Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

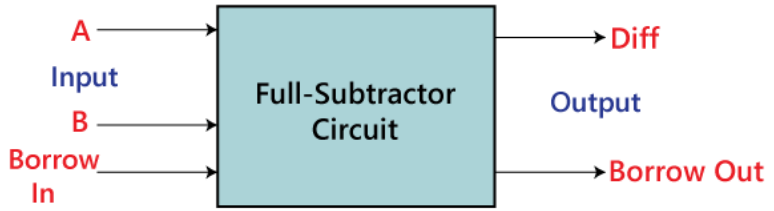
K-map for Difference and Borrow



Logic Diagram



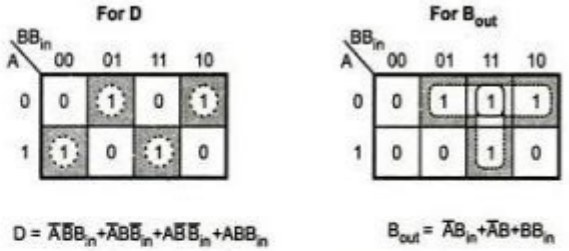




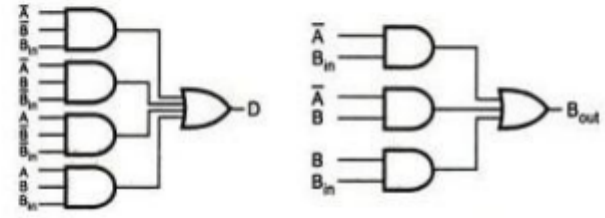
Truth table

Inputs			Outputs	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

K-map for D and Bout



Logic Diagram



Combinational circuits

Mutliplexer (Mux)

Multiplexer is a combinational circuit that selects binary information from one of many inputs and directs it into single output.

The selection of particular input is controlled by a set of selection line

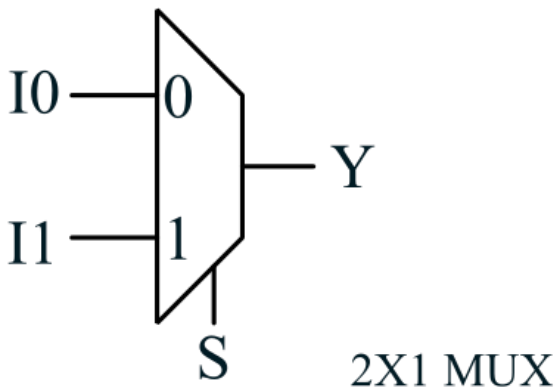
Mutliplexer has 2^n inputs, n select line (control input) and one output

It also called as Data selector

2 to 1 Multiplexer

has 2^1 inputs, 1 select line and one output

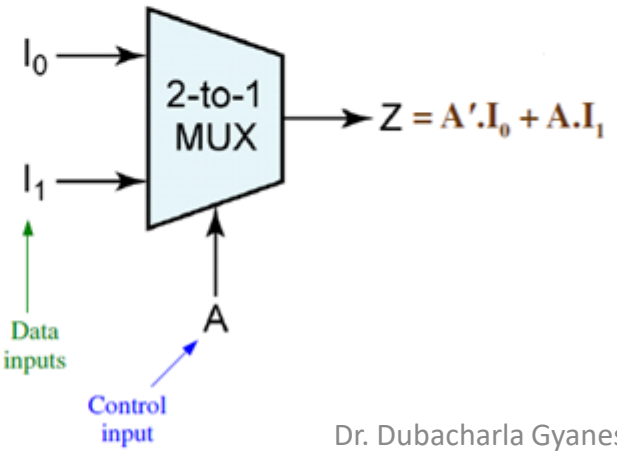
Circuit diagram



2X1 MUX

Truth Table

S_0	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

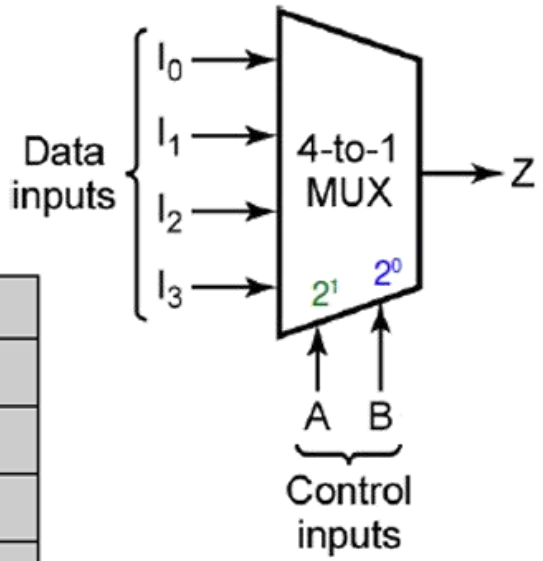


Combinational circuits

Mutliplexer (Mux)

4 to 1 MUX

4 to 1 MUX has 2^2 = 4 inputs, 2 select line and one output

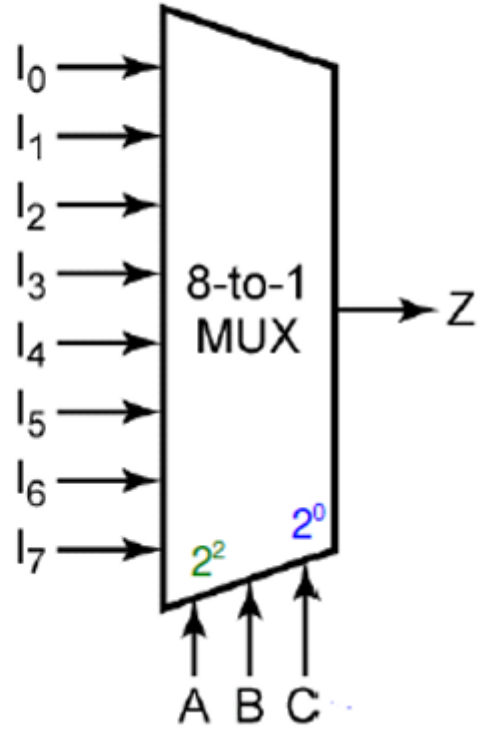


A	B	Z
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

$$Z = A'.B'.I_0 + A'.B.I_1 + A.B'.I_2 + A.B.I_3$$

8 to1 MUX

8 to1 MUX has 2^3 = 8 inputs, 3 select line and one output



Combinational circuits

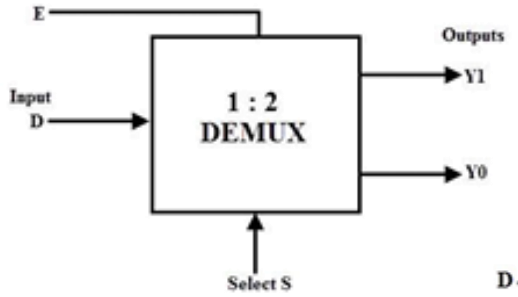
Demultiplexer (DEMUX)

Demultiplexer has 2^n outputs , n select lines, one input.

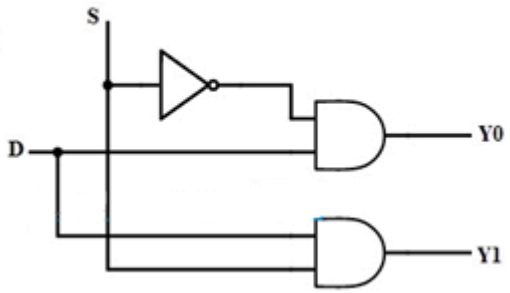
A **demultiplexer** is also called a data distributor.

1-to-2 demultiplexer

has 2^2 outputs , 2 select lines, one input.



Logic diagram

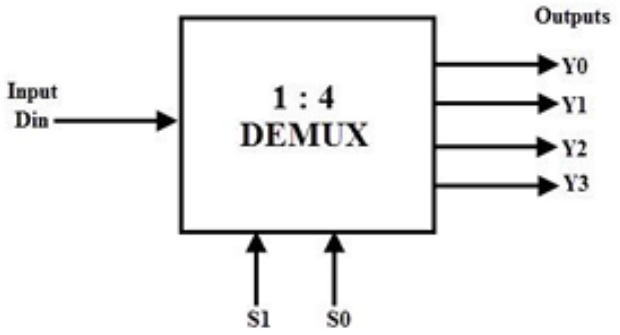


The truth table

Select	Input	Outputs	
S	D	Y_1	Y_0
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

1-to-4 Demultiplexer

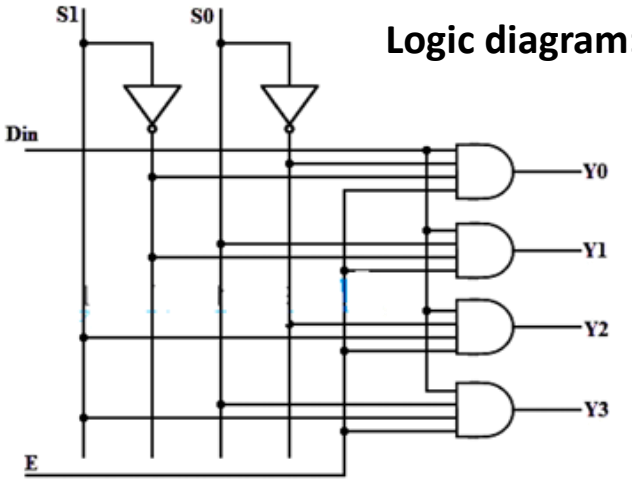
It has one input,2 select lines,4 outputs



Truth table:

Data Input	Select Inputs		Outputs			
D	S_1	S_0	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	D
D	0	1	0	0	D	0
D	1	0	0	D	0	0
D	1	1	D	0	0	0

Logic diagram:



$$Y0 = \overline{S1}\ \overline{S0}\ D$$

$$Y1 = \overline{S1}\ S0\ D$$

$$Y2 = S1\ \overline{S0}\ D$$

$$Y3 = S1\ S0\ D$$

Combinational circuits

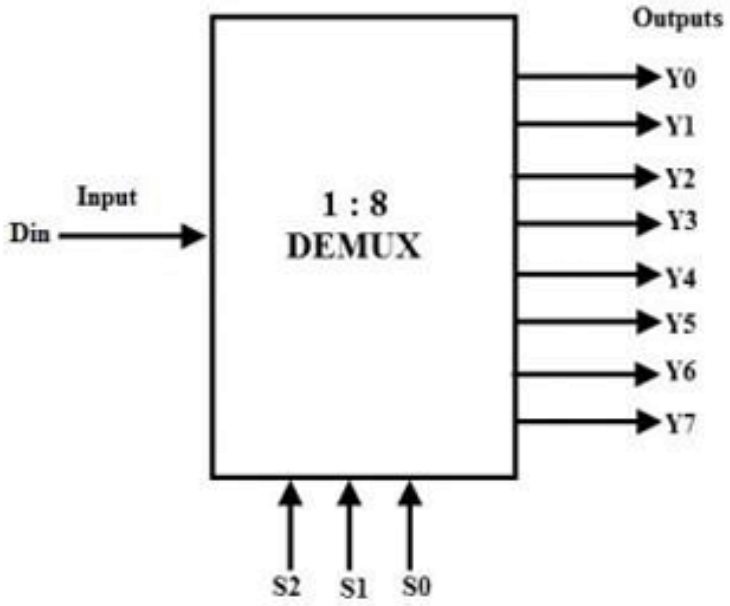
Demultiplexer (DEMUX)

1-to-8 Demultiplexer

Has one input

3-select lines

8-outputs



Truth table:

Data Input	Select Inputs			Outputs							
D	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	0	0	0	0	0	D
D	0	0	1	0	0	0	0	0	0	D	0
D	0	1	0	0	0	0	0	0	D	0	0
D	0	1	1	0	0	0	0	D	0	0	0
D	1	0	0	0	0	0	D	0	0	0	0
D	1	0	1	0	0	D	0	0	0	0	0
D	1	1	0	0	D	0	0	0	0	0	0
D	1	1	1	D	0	0	0	0	0	0	0

$Y0 = D\,\overline{S2}\,\overline{S1}\,\overline{S0}$

$Y1 = D\,\overline{S2}\,\overline{S1}\,S0$

$Y2 = D\,\overline{S2}\,S1\,\overline{S0}$

$Y3 = D\,\overline{S2}\,S1\,S0$

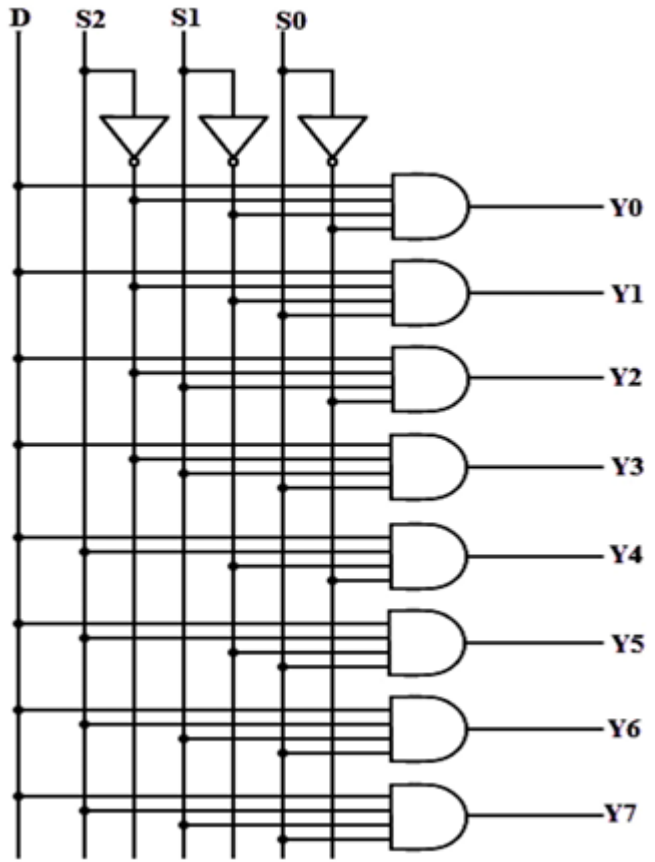
$Y4 = D\,S2\,\overline{S1}\,\overline{S0}$

$Y5 = D\,S2\,\overline{S1}\,S0$

$Y6 = D\,S2\,S1\,\overline{S0}$

$Y7 = D\,S2\,S1\,S0$

Logic diagram:



Combinational circuits

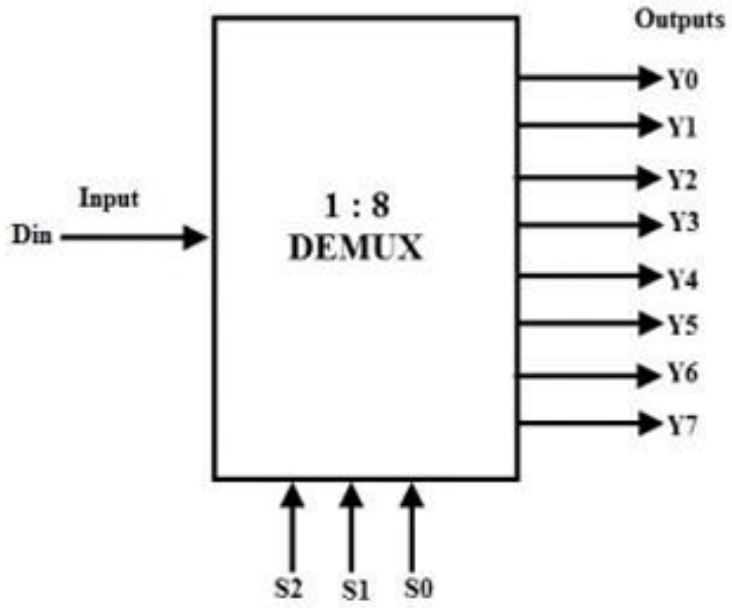
Demultiplexer (DEMUX)

1-to-8 Demultiplexer

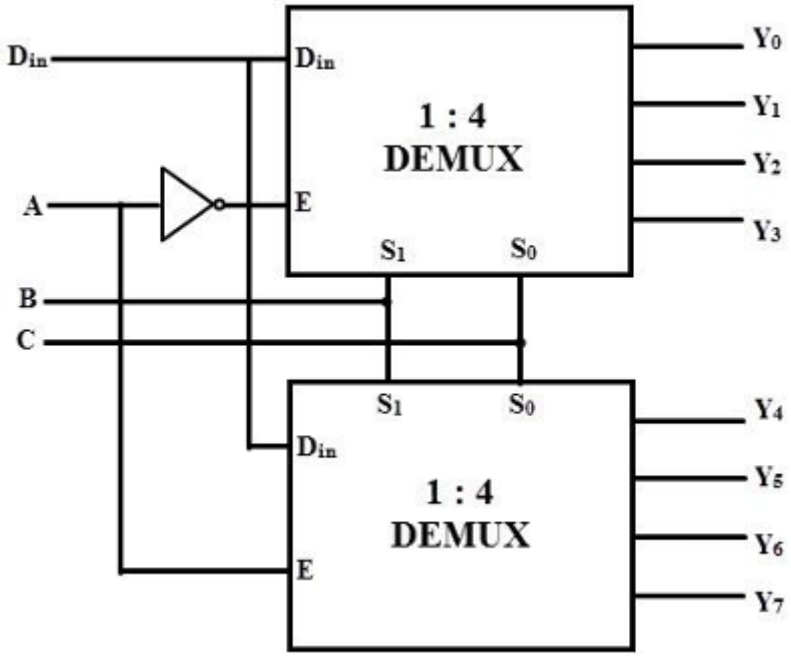
Has one input

3-select lines

8-outputs



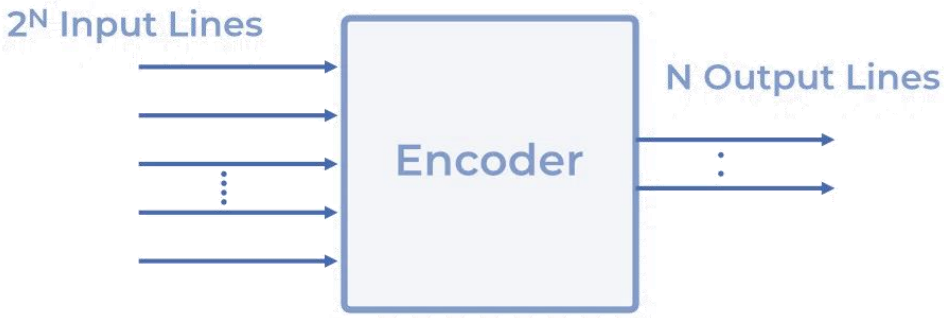
≡



Combinational circuits

Encoders

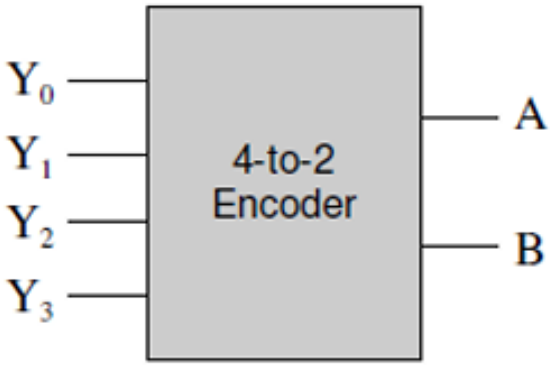
Encoders is a combinational circuit which takes 2^N inputs and gives out N outputs, the enable pin should be kept 1 for enabling the circuit.

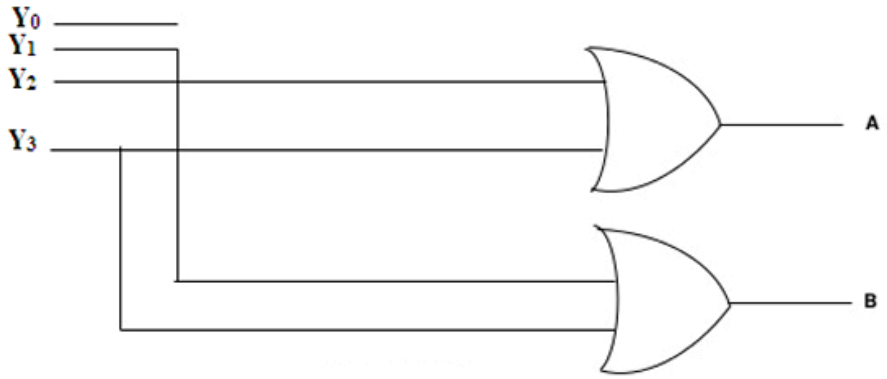


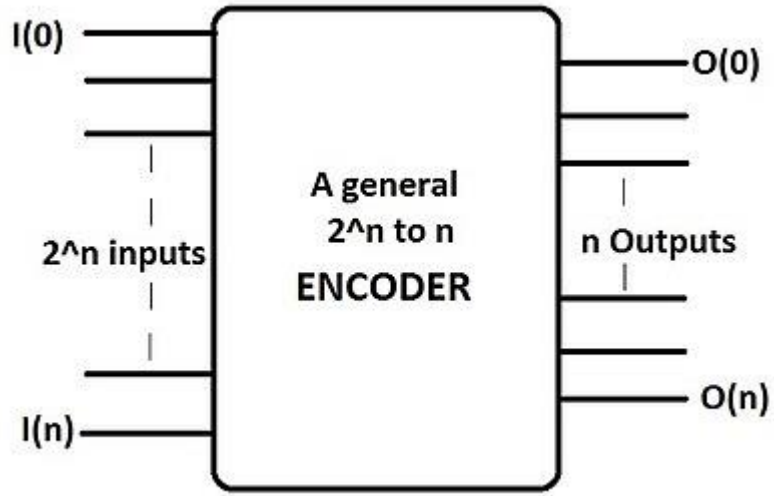
4 to 2 Encoder

It has 2^2 inputs and 2 outputs.

Y_0	Y_1	Y_2	Y_3	A	B
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1



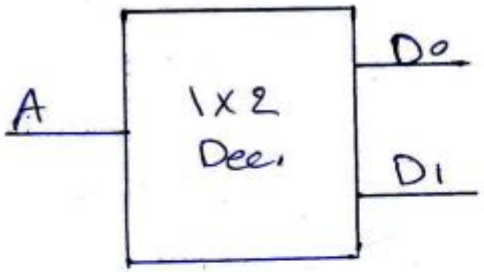




Combinational circuits

Decoder Decoder is a combinational circuit.

It has N inputs and 2^N outputs.



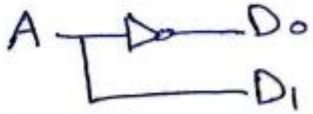
A	D0	D1
0	1	0
1	0	1

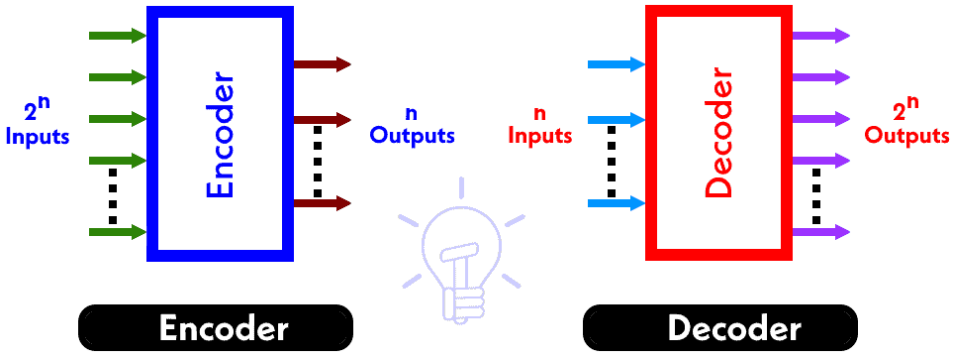
"Truth tabe"

From the truth table:-

$D_0 = \bar{A}$

$D_1 = A$





- ***Uses***:
  - *Address Decoding in Memory Systems*
  - *Digital Display Systems*
  - *Multiplexing and Demultiplexing*
  - *Control Units in Microprocessors*
  - *IoT Devices*
  - *Security and Authentication Systems*
  - *Analog-to-Digital Conversion*
  - *Robotics and Automation*

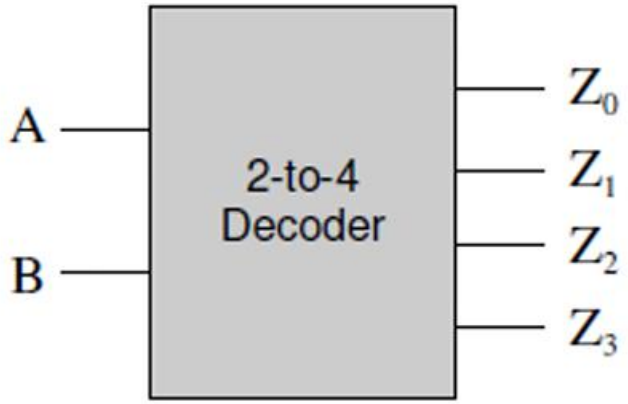
Combinational circuits

Decoder

2 to 4 Decoder

It has 2 inputs and 2^2 = 4 outputs.

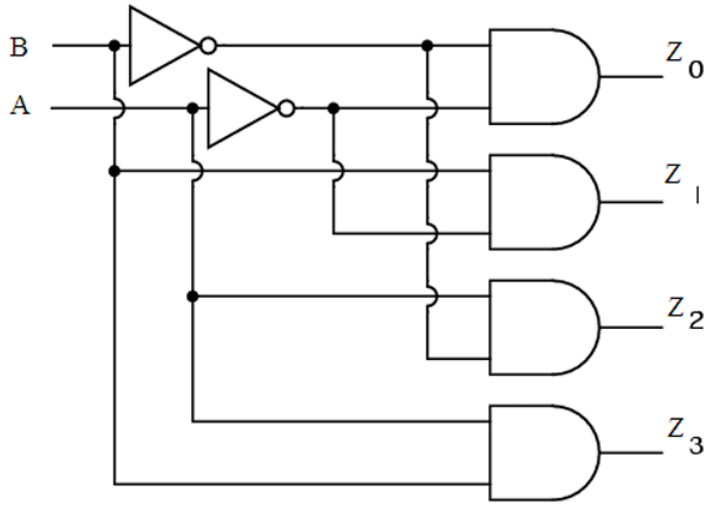
Circuit Diagram



Truth Table

A	B	Z_0	Z_1	Z_2	Z_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Logic Diagram

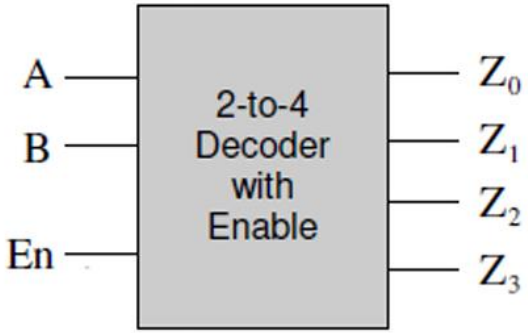


Combinational circuits

Decoder

2 to 4 Decoder with Enable input

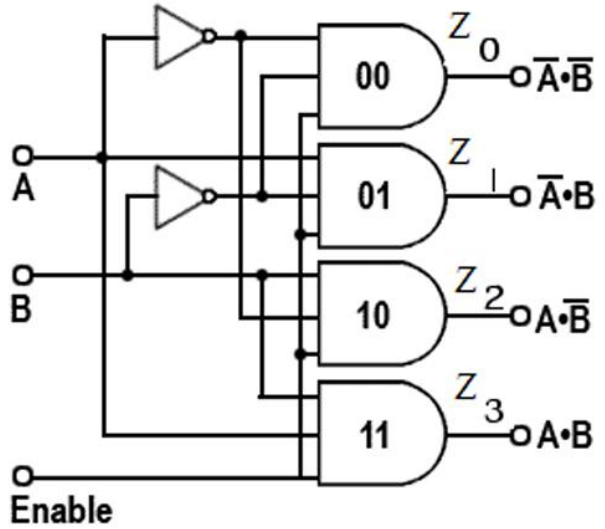
Circuit Diagram



Truth Table

	En	A	B	Z_0	Z_1	Z_2	Z_3
enabled	1	0	0	1	0	0	0
	1	0	1	0	1	0	0
	1	1	0	0	0	1	0
	1	1	1	0	0	0	1
disabled	0	x	x	0	0	0	0

Logic Diagram

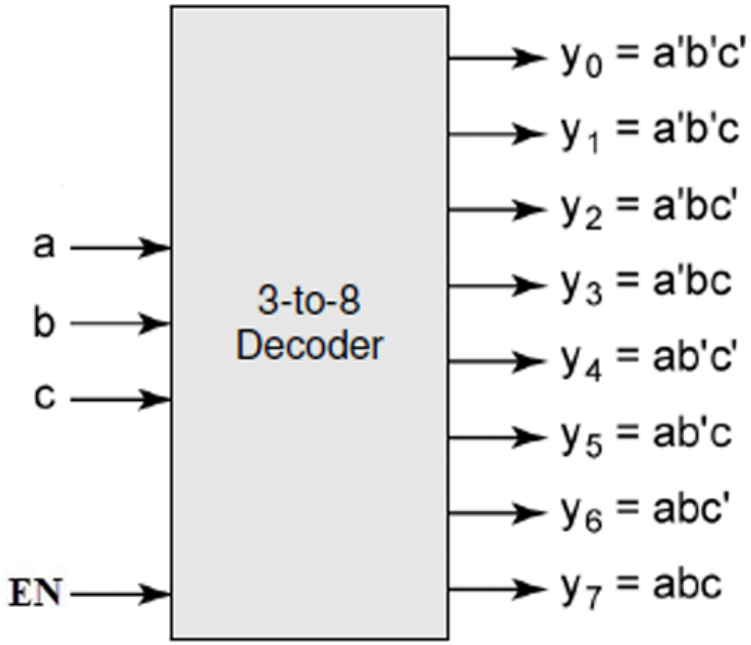


Combinational circuits

Decoder

3 to 8 Decoder

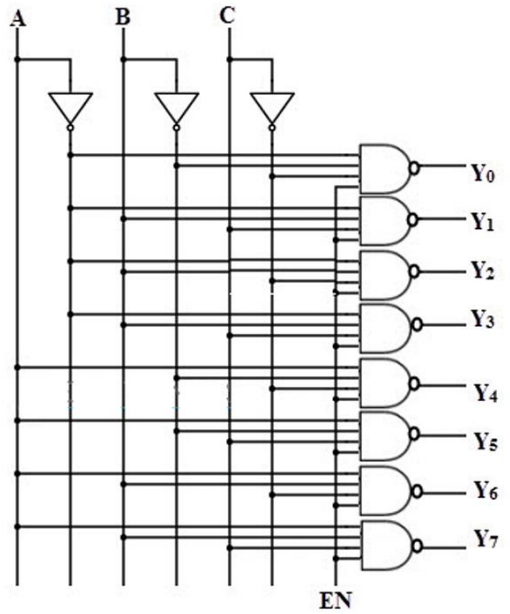
Circuit Diagram



Truth Table

Inputs				Outputs							
EN	A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Logic Diagram



Sequential circuits

- **Memory-Based**: Sequential circuits store past inputs using memory elements like flip-flops, enabling time-dependent behaviour.
- **Clock-Driven**: They rely on a clock signal to synchronize state changes, ensuring orderly and predictable operation.
- **State Dependency**: Sequential circuits rely on memory to store past states and are time-dependent, while combinational circuits are stateless and depend solely on current inputs for their outputs.

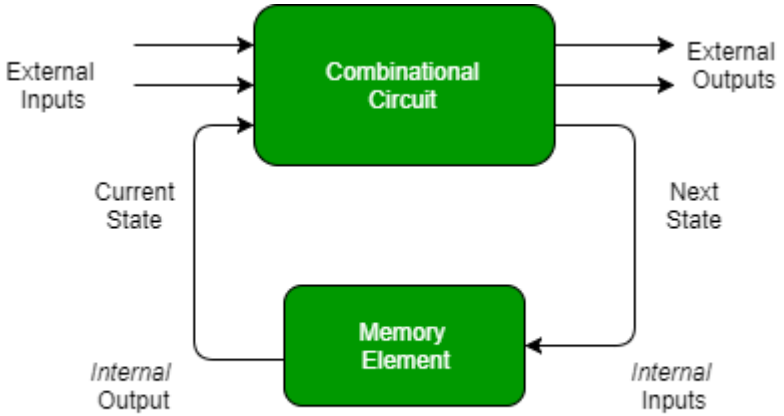
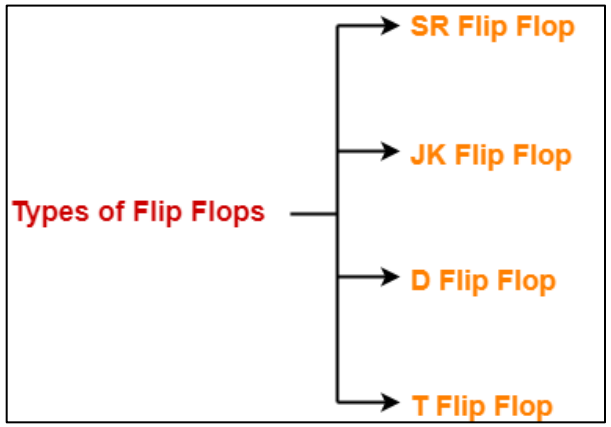
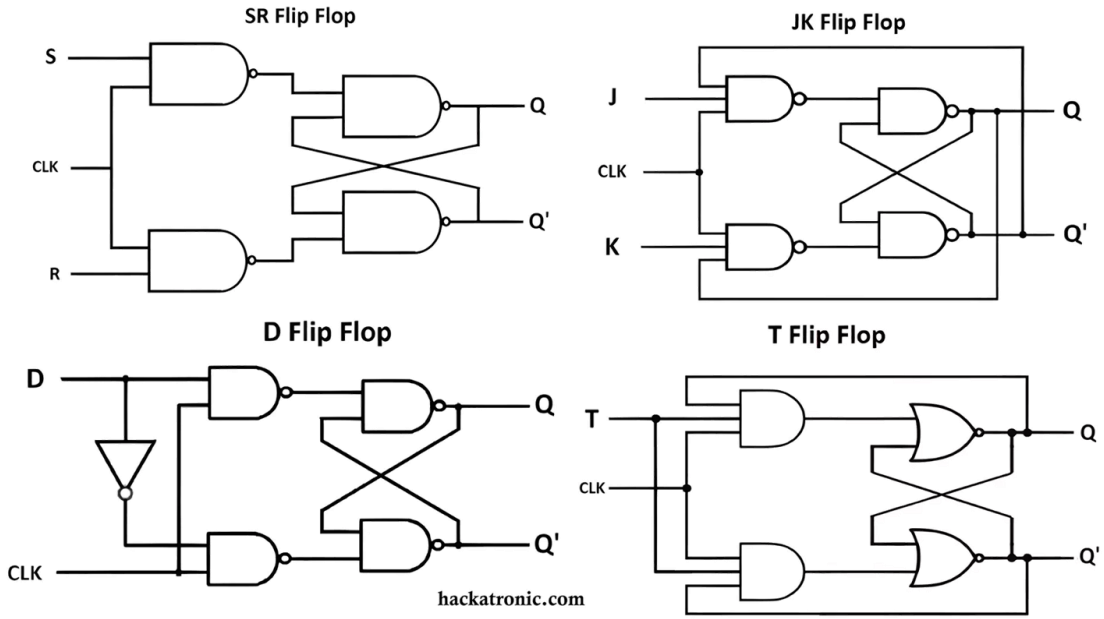


Figure: Sequential Circuit





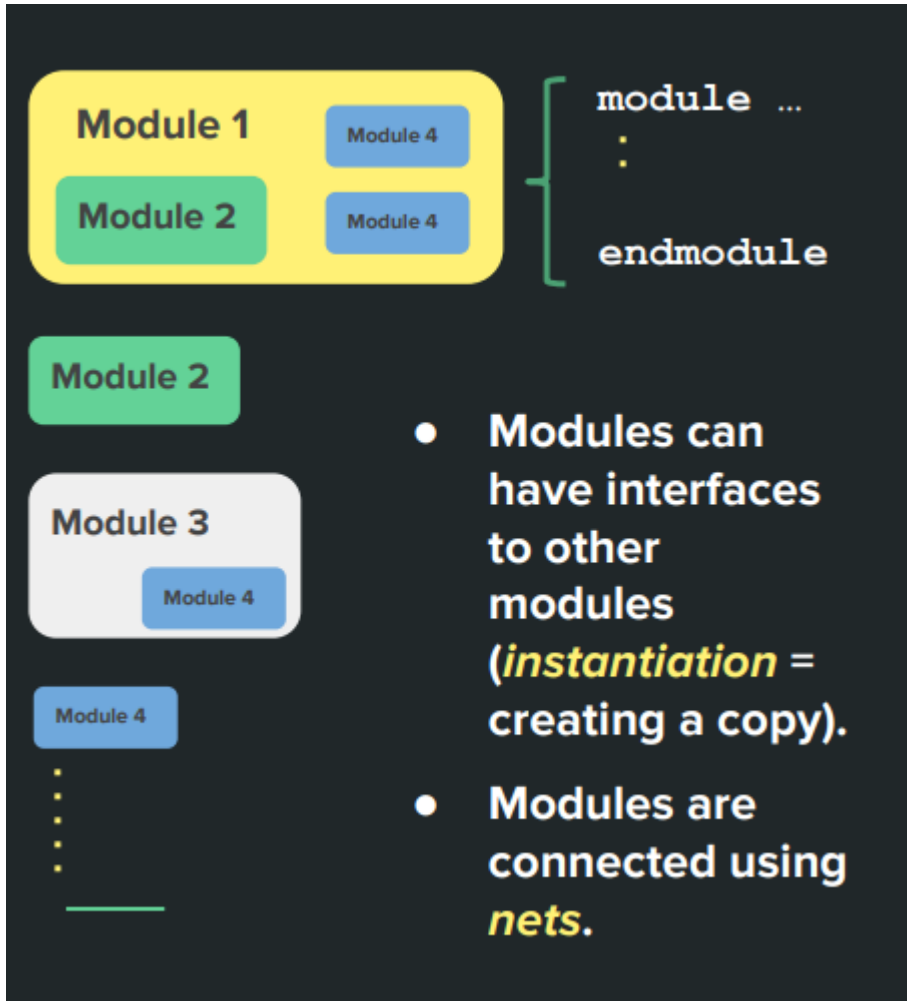
Comparison

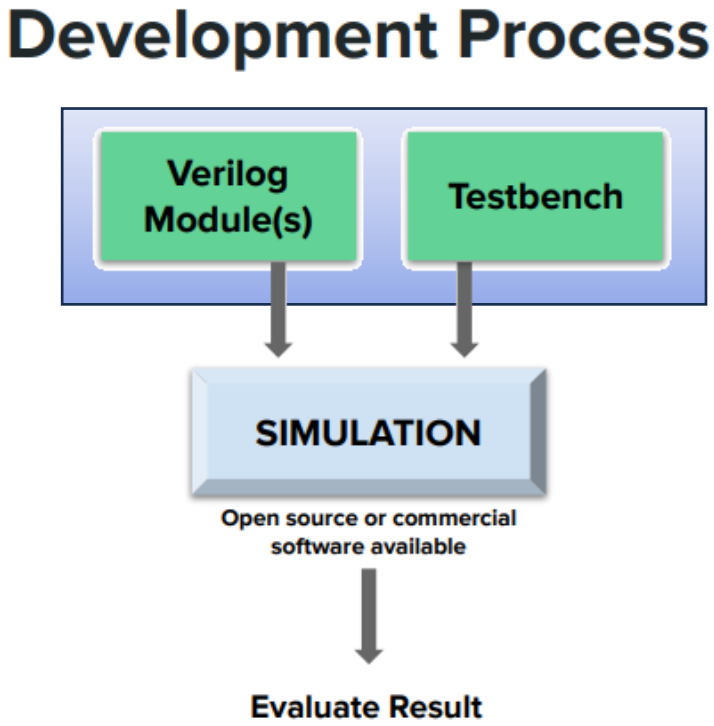
➢ Feature	➢ Combinational Circuits	➢ Sequential Circuits
Dependency	Outputs depend only on current inputs.	Outputs depend on current inputs and past states.
Memory	No memory elements (stateless).	Requires memory elements (stateful).
Clock Requirement	Operates without a clock.	Requires a clock signal for state changes.
Examples	• Basic Logic Gates: AND, OR, NOT. • Adders: Half Adder, Full Adder. • Subtractor: Half Subtractor, Full Subtractor. • Encoders and Decoders. • Multiplexers (MUX) and Demultiplexers (DEMUX). • Comparators and Parity Generators.	• Flip-Flops: SR, D, JK, T Flip-Flops. • Counters: Ripple Counter, Synchronous Counter, Up/Down Counter. • Registers: Shift Registers, Parallel Registers. • Finite State Machines (FSMs): Mealy and Moore machines. • Memory: Random Access Memory (RAM), Read-Only Memory (ROM).
Applications	• Arithmetic operations • Data routing and selection, etc	• Data storage and transfer • Timing and control circuits ,etc

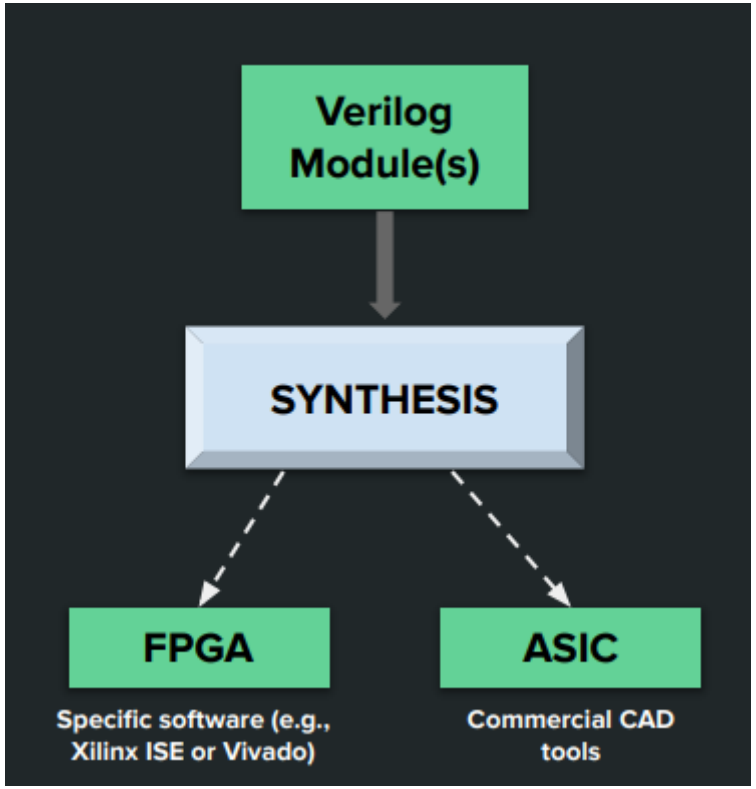
Fundamental concepts in HDL

- **Signals and Variables**: Represent data storage and transfer within a digital system.
- **Control Structures**: Conditionals (if, case) and loops to direct data flow and operations.
- **Modules and Architectures**: Define hierarchical structure and organization of digital components.
- **Libraries and Packages**: Reusable collections of commonly used functions and definitions.
- **Syntax and Semantics**: Specific rules and conventions for writing valid HDL code.
- **Simulation and Synthesis Tools**: Software for verifying design behavior and translating HDL to hardware.

Verilog cycle



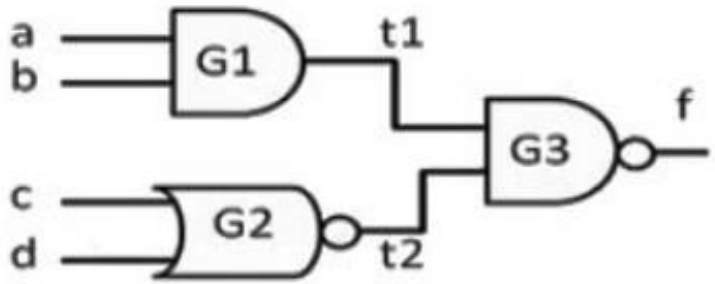




Verilog example

Module example 2: A 2-level combinational circuit

```
// A 2-level combinational circuit
module two_level(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2; // intermediate lines
    assign t1 = a & b;
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```




```
'timescale 1ns/100ps
module(port_list)
    Declarations:
        Inputs, Outputs,
        Wire, Reg, Parameter

    Statements:
        Initial Statement
        Always Statement
        Continuous Statement
        Module Instantiation
endmodule
```

Verilog

- **Keywords**:
  - **Module-related**: *module, endmodule*
  - **Data types**: *wire, reg, integer, real, logic*
  - **Control flow**: *if, else, case, for, while, repeat, forever*
  - **Concurrency**: *always, initial, assign*
  - **Procedural blocks**: *begin, end, function, task*
  - **Delays & Timing**: *#, @, posedge, negedge*
  - **Miscellaneous**: *generate, parameter, localparam, define, include*

Verilog

- **Control Flow Statements**:
  - **Conditional**: if, else, case
  - **Loops**: for, while, repeat, forever
  - **Procedural**: always, initial, begin … end
  - **Wait/Event**: @(posedge clk), #10

```
always @(posedge clk)
  begin
    if (reset)
      counter
```

```
for (initialization; condition; increment/decrement)
  begin
    // Statements to execute in each iteration
  end
```

```
if (condition)
  begin
    // Statements to execute when the condition is true
  end
else
  begin
    // Statements to execute when the condition is false
  end
```

```
case (value)
  value1: begin
            // Statements to execute when value equals value1
          end
  value2: begin
            // Statements to execute when value equals value2
          end
  default: begin
             // Statements to execute when none of the cases match
           end
endcase
```

Verilog

- **Variables and Data Types**:
  - **Bit/Logic:** wire, reg, logic, bit
  - **Integer:** integer, real
  - **Arrays:** reg [7:0] array [0:15]
  - **Constants:** parameter, localparam
  - **String:** Not directly supported

- **Bit** – represents a single binary value (0 or 1).
- **Integer** – represents a signed or unsigned whole number.
- **Real** – represents a real number with decimal points.
- **Reg** – represents a register that can store binary values.
- **Wire** – represents connections between different modules.
- **Enum** – represents a set of named values.
- **Array** – represents a collection of elements of the same data type.

```
 integer count; // declares an integer variable named 'count'
bit [7:0] data; // declares an 8-bit vector variable named 'data'
reg [3:0] enable = 4'd7; // declares a 4-bit register variable named 'enable' with an in
itial value of 7
```

Verilog vs VHDL

➢Feature	➢Verilog	➢VHDL
• **Signal Declaration**	• wire is used for connecting components (continuous assignment).	• signal is used for connecting components and holds values across simulation cycles.
• **Storage Elements**	• reg is used to represent variables or sequential elements (flip-flops).	• variable is used for temporary storage within a process, signal for inter-process communication.
• **Concurrency**	• Concurrent by nature with explicit always and continuous assignments.	• Concurrent by nature with processes that describe sequential operations.
• **Hardware Modeling**	• Procedural blocks: always, initial.	• Procedural blocks: process, wait.
• **Data Types**	• Fewer, simpler types (e.g., wire, reg, logic).	• Rich data types (e.g., std_logic, integer, real, enumerations).
• **Bit-Vector Type**	• wire or reg for buses and vectors.	• std_logic_vector for buses and vectors.
• **Case Sensitivity**	• Case-sensitive (e.g., Signal_A ≠ signal_a).	• Case-insensitive (Signal_A = signal_a).
• **Testbench Style**	• Procedural, supports tasks and functions.	• Behavioral with more structured testbench writing, supports procedures and functions.
• **Hierarchy**	• Supports modules for hierarchy and reuse.	• Supports entities and architectures for hierarchy and reuse.
• **Comments**	• // for single-line, /* … */ for multi-line.	• -- for single-line, no multi-line comments.
• **Library Usage**	• Lacks formal library mechanism (uses include).	• Strong library mechanism (library, use).

Verilog: Number representation

Type	Format	Example
Decimal	<bit-width>'d<decimal_value>	8'd42 (Represents an 8-bit unsigned decimal 42)
Binary	<bit-width>'b<binary_value>	4'b1010 (Represents a 4-bit binary number 1010)
Hexadecimal	<bit-width>'h<hex_value>	8'h1F (Represents 8-bit hexadecimal 1F = binary 00011111)
Octal	<bit-width>'o<octal_value>	6'o77 (Represents 6-bit octal 77 = binary 0111111)
Signed	<bit-width>'s<signed_decimal_value>	8's-25 (Represents 8-bit signed decimal -25)
Unsigned	Implicit for all values unless signed is used.	reg [7:0] my_value = 42;
Real Numbers	Use the real datatype (simulation only).	real pi_value = 3.1415;

VHDL: Number representation

Type	Format	Example
Decimal	Use integer or real.	signal my_value : integer := 42;
Binary	Use std_logic_vector or bit_vector.	signal my_binary : std_logic_vector(3 downto 0) := "1010";
Hexadecimal	Use X"..." for compact binary.	signal my_hex : std_logic_vector(7 downto 0) := X"1F";
Octal	No direct representation (convert manually).	Use binary equivalent of octal (e.g., "111" for octal 7).
Signed	Use the signed datatype from numeric_std.	signal signed_num : signed(7 downto 0) := "10101010";
Unsigned	Use the unsigned datatype from numeric_std.	signal unsigned_num : unsigned(7 downto 0) := "10101010";
Real Numbers	Use real (simulation only).	signal my_real : real := 12.3456;

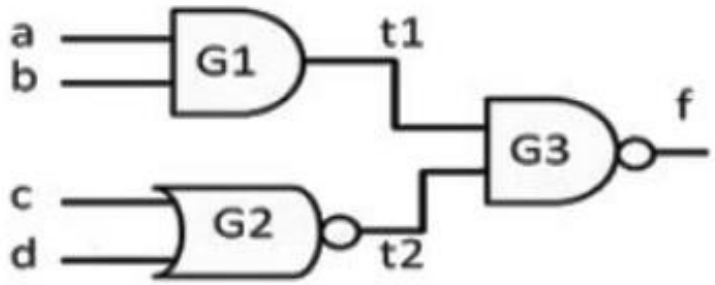
VHDL vs Verilog Operators

	VHDL	Verilog		VHDL	Verilog
Add	+	+	Bitwise Negation	not	~
Subtract	-	-	Bitwise NAND	nand	~&
Multiplication	*	*	Bitwise NOR	nor	~|
Division	/	/	Bitwise XNOR	xnor	~^
Modulo	mod	%	Greater (or Equal)	>, >=	>, >=
Absolute	abs	N/A	Less (or Equal)	<, <=	<, <=
Exponentiation	**	**	Logical Equality	=	==
Concatenation	&	{ , }	Logical Inequality	/=	!=
Left Shift	sll	<<	Logical AND	and	&&
Right Shift	srl	>>	Logical OR	or	||
Bitwise AND	and	&	Logical Negation	not	!
Bitwise OR	or	|	Case Equality	N/A	===
Bitwise XOR	xor	^	Case Inequality	N/A	!==

Verilog Structure Example

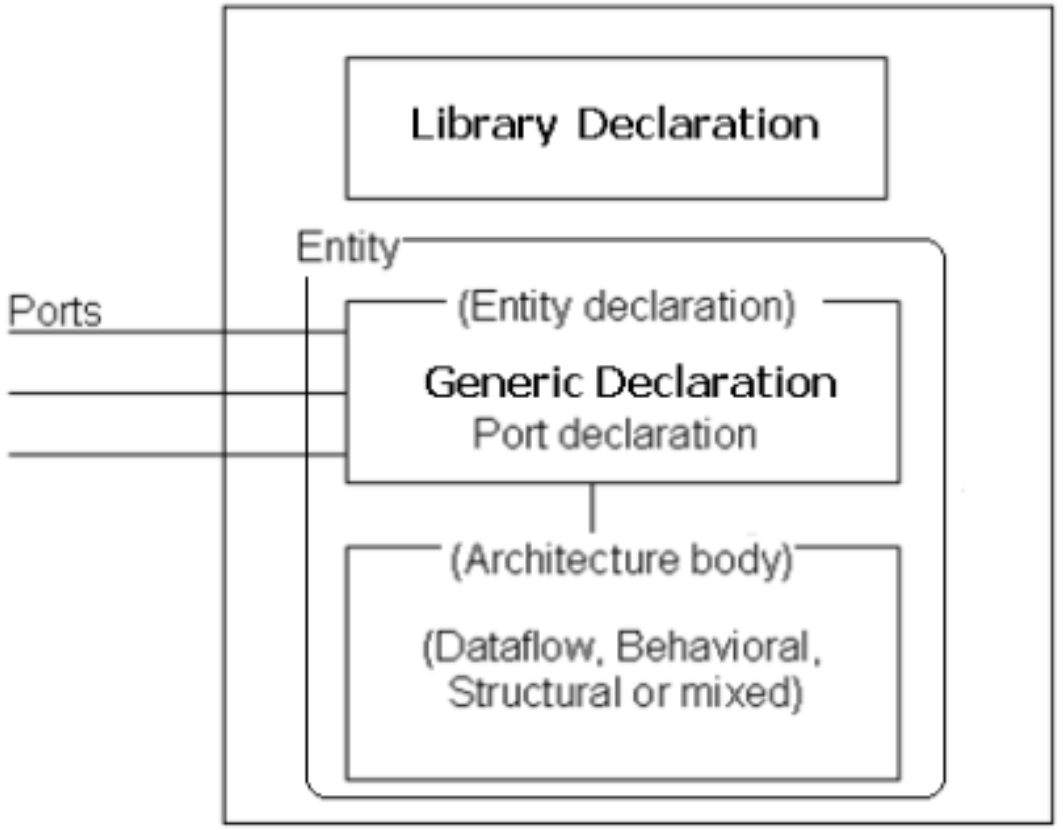
Module example 2: A 2-level combinational circuit

```
// A 2-level combinational circuit
module two_level(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2; // intermediate lines
    assign t1 = a & b;
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```




```
'timescale 1ns/100ps
module(port_list)
    Declarations:
        Inputs, Outputs,
        Wire, Reg, Parameter

    Statements:
        Initial Statement
        Always Statement
        Continuous Statement
        Module Instantiation
endmodule
```

VHDL Structure Example



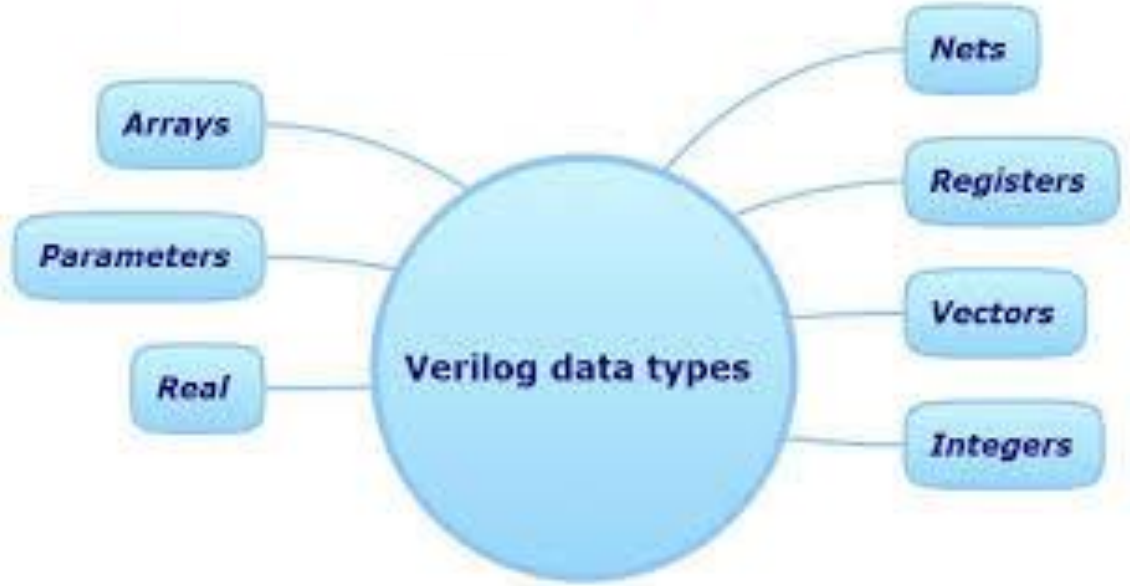
Structure of VHDL Code

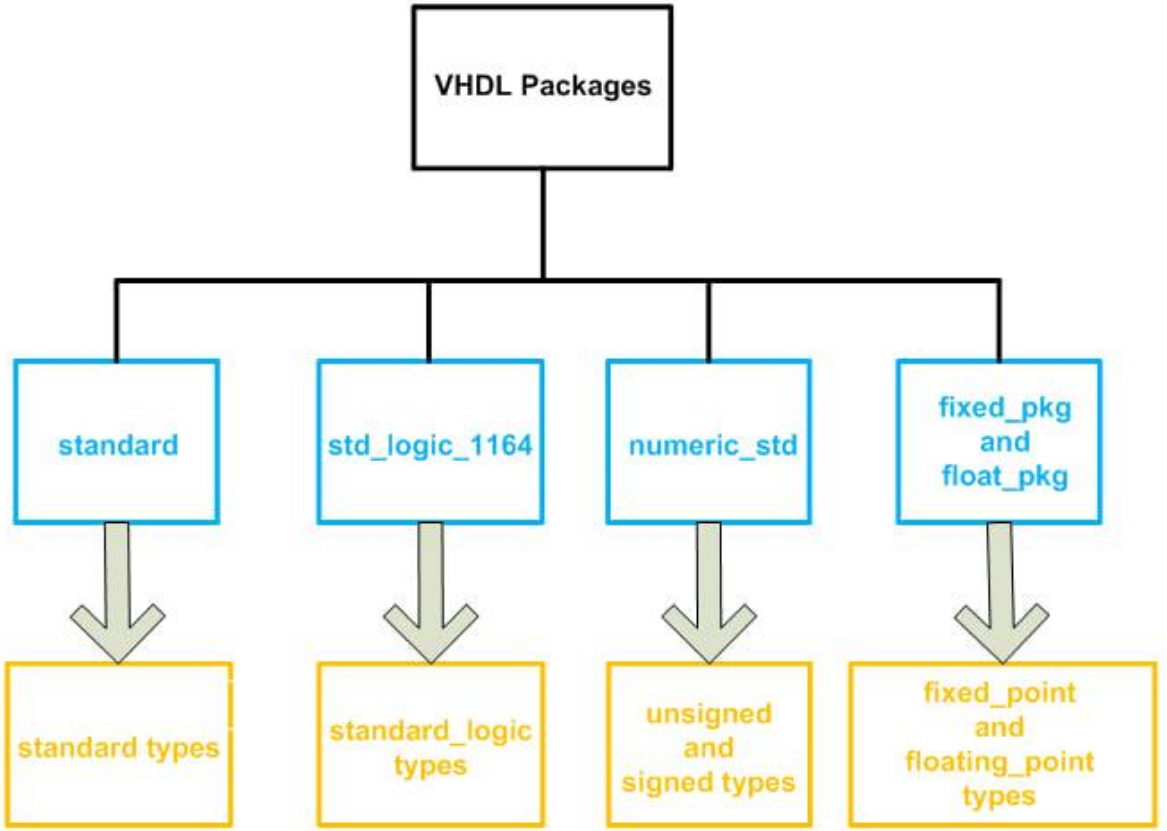
```
LIBRARY<library_Name>; — Library IEEE will be used almost in every code

USE<library_name>.<package_name>.ALL; –this is the format for importing packages

USEIEEE.STD_LOGIC_1164.ALL; — this package is required for bit, std_logic,

–& std_logic_vector declarations and some related operations

— arithmetic functions with Signed or Unsigned values

USEIEEE.NUMERIC_STD.ALL;

ENTITY<entity_name>IS

GENERIC( generic_name : generic_type : generic_value);
```

VHDL Structure Example

```
LIBRARY<library_Name>; — Library IEEE will be used almost in every code
USE<library_name>.<package_name>.ALL; –this is the format for importing packages
USEIEEE.STD_LOGIC_1164.ALL; — this package is required for bit, std_logic,
USEIEEE.NUMERIC_STD.ALL;
ENTITY<entity_name>IS
GENERIC( generic_name : generic_type : generic_value);
PORT
(
    <signal_name> : mode <type>;
    <signal_name> : mode <type> := default_value;
    <signal_name> : INOUT<type>;
    <signal_name> : OUT<type> := <default_value>);
END<entity_name>;
— Next we write the architecture for the entity
ARCHITECTURE<architecture_name>OF <Entity_name>IS
    — declaration of optional signals, constants etc;
BEGIN
    –here come the statement that describe the behaviour of the design
    –commonly used statements are:
    –Optional Concurrent signal Assignment
    –Optional Process Statement
    –Optional Conditional Signal assignment statement
    –optional Generate statement
end [Architecture_name];
```

Data Types: Verilog vs VHDL





VHDL vs Verilog Structure

- To use standard logic data types place at top of source file
  - LIBRARY ieee; -- library
  - USE ieee.std_logic_1164.ALL; -- package
- Note: VHDL is case insensitive, free format.
- Semicolon (;) terminates statement
- Comments are preceded by two consecutive dashes.
  - comment ends at end of current line
- A digital component is described using an
  - ENTITY DECLARATION and a corresponding
  - ARCHITECTURE BODY.
- Std_logic is an enumeration type defined in an IEEE package
  - has the values '0' and '1' and 'Z' (and others)
- Ports are like IC pins, connected by wires called SIGNALS

VHDL (.vhd)	Verilog (.v)
-- Library Declaration	// One Module
library IEEE;	module mux (...);
...	
-- Entity Declaration	
entity mux is	
...	
end mux	...
-- Architecture Body	
architecture arch of mux is	
begin	
...	
end arch;	endmodule

VHDL vs Verilog I/O Declaration

VHDL

```
-- Library Declaration
...
-- Entity Declaration
entity mux is
port(a,b,s: in std_logic;
     y: out std_logic);
end mux
-- Architecture Body
architecture arch of mux is
begin
...
end arch;
```

Verilog

```
// One Module
module mux (a, b, s, y);
input a,b,s;
output y;
...
endmodule
```

VHDL vs Verilog Statements

VHDL

```
-- Entity Declaration
entity mux is
port(a,b,s: in std_logic;
     y: out std_logic);
end mux
architecture arch of mux is
begin
-- concurrent statements
process (...) begin
  -- sequential statements
end process;
end arch;
```

Verilog

```
// One Module
module mux (a, b, s, y);
input a,b,s;
output y;
-- concurrent statements
always @(...)
begin
  -- sequential statements
end
endmodule
```

Verilog

Operators

Bitwise Operators:

~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Examples:

```
wire a, b, c, d, f1, f2, f3, f4;
assign f1 = ~a | b;
assign f2 = (a & b) | (b & c) | (c & a);
assign f3 = a ^ b ^ c;
assign f4 = (a & ~b) | (b & c & ~d);
```

Operators: Logical vs. Bitwise

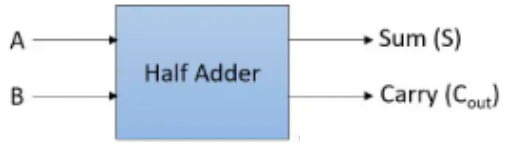
1'b1 && 1'b1	= 1	1'b1 & 1'b1	= 1
1'b1 && 1'b0	= 0	1'b1 & 1'b0	= 0
! 1'b1	= 0	~ 1'b1	= 0
! 3'b000	= 1	~ 3'b000	= 111
3'b111 && 3'b111	= 1	3'b111 & 3'b111	= 111
3'b100 && 3'b111	= 0	3'b000 & 3'b111	= 000

```
module logical_op;
  reg [3:0] i1, i2;
  initial begin
    i1 = 4'h6; i2 = 4'h2;
    $display("For operator: (&&): i1 = %0h && i2 = %0h: %h", i1, i2, i1 && i2);
    $display("For operator: (||): i1 = %0h || i2 = %0h: %h", i1, i2, i1 || i2);
    $display("For operator: (!) : i1 = %0h !  i2 = %0h: %h", i1, i2, !i1);

    i1 = 4'b1x0z; i2 = 4'b0x1x;
    $display("For operator: (&&): i1 = %0b && i2 = %0b: %h", i1, i2, i1 && i2);
    $display("For operator: (||): i1 = %0b || i2 = %0b: %h", i1, i2, i1 || i2);
  end
endmodule
```

```
For operator: (&&): i1 = 6 && i2 = 2: 1
For operator: (||): i1 = 6 || i2 = 2: 1
For operator: (!) : i1 = 6 !  i2 = 2: 0
For operator: (&&): i1 = 1x0z && i2 = x1x: 1
For operator: (||): i1 = 1x0z || i2 = x1x: 1
```

Verilog: examples

Block Diagram



Truth Table

A	B	Sum (S)	Carry (Cout)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Output:

S = A ^ B

Cout = A · B

```
module half_adder(input a, b, output s, Cout);
  assign S = a ^ b;
  assign Cout = a & b;
endmodule
```

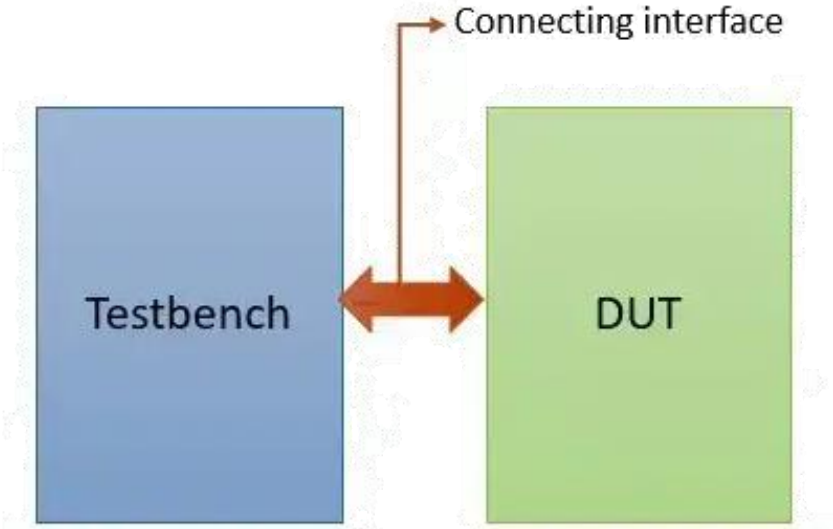
Testbench Code

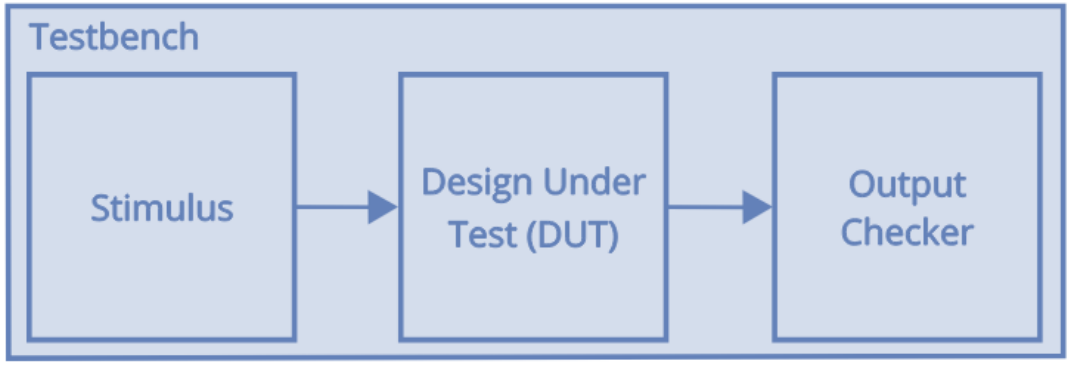
```
module tb_top;
  reg a, b;
  wire s, c_out;

  half_adder ha(a, b, s, c_out);

  initial begin
    $monitor("At time %0t: a=%b b=%b, sum=%b, carry=%b",$time, a,b,s,c_out);
    a = 0; b = 0;
    #1;
    a = 0; b = 1;
    #1;
    a = 1; b = 0;
    #1;
    a = 1; b = 1;
  end
endmodule
```

```
At time 0: a=0 b=0, sum=z, carry=0
At time 1: a=0 b=1, sum=z, carry=0
At time 2: a=1 b=0, sum=z, carry=0
At time 3: a=1 b=1, sum=z, carry=1
```

Verilog: Testbench

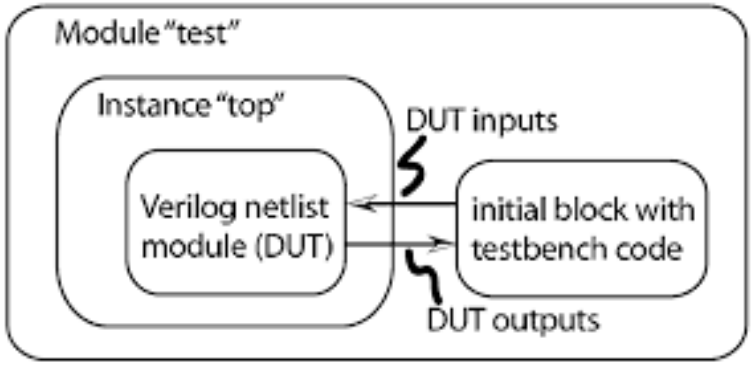
- The testbench is written to check the functional correctness based on design behavior.
- The connections between design and testbench are established using a port interface.
- A testbench generates and drives a stimulus to the design to check its behavior. Since the behavior of the design has to be tested, the design module is known to be "**Design Under Test**" (DUT)





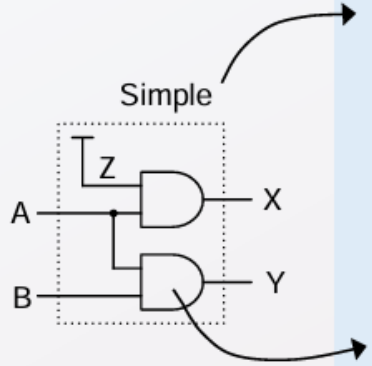
Verilog: Testbench

- **`timescale directive**: Specifies the time unit and precision for the simulation.
- **Module Declaration (module testbench_name)**: Defines the test bench module.
- Input and Output Declarations: Inputs are declared as reg and outputs as wire.
- **Instantiation of DUT (Design Under Test)**: Creates an instance of the design module being tested.
- **Clock Generation (optional):** Generates a clock signal if needed.
- **Initial Block**: Initializes inputs, applies test vectors, and terminates the simulation.
- The initial block initializes the input signals and provides test vectors.
- The always block generates a clock signal.
- The always @ (posedge clk) block in the design module would describe the behavior of the design on the positive edge of the clock.
- **Monitoring Outputs (optional):** Monitors and prints the values of specified signals during the simulation.



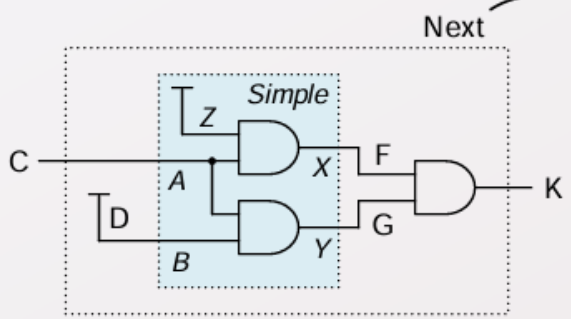
Verilog: Testbench Example

The module declaration statement defines the “bounding box”, and the module body statements define the circuit functions




```
module simple(
    input A, B;
    output X, Y;
);

wire Z = 1b'1;
assign X = A & Z;
assign Y = A & B;
end module
```

Any module can be “instantiated” as a component in another module by listing it’s name and port connections. Here, “named association” is used.



.component_port(topmodule_port)

```
module next(
    input C;
    output K;
);

wire D = 1b'1;
wire F,G;

simple(.A(C),.B(D),.F(X),.G(Y))
assign K = F & G;
end module
```

Combinational circuits – Verilog

Mutliplexer (Mux)

Multiplexer is a combinational circuit that selects binary information from one of many inputs and directs it into single output.

The selection of particular input is controlled by a set of selection line

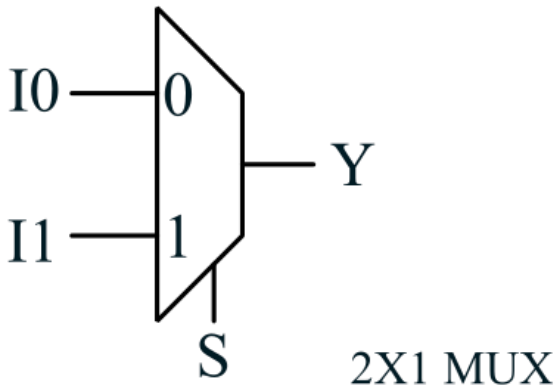
Mutliplexer has 2^n inputs, n select line (control input) and one output

It also called as Data selector

2 to 1 Multiplexer

has 2^1 inputs, 1 select line and one output

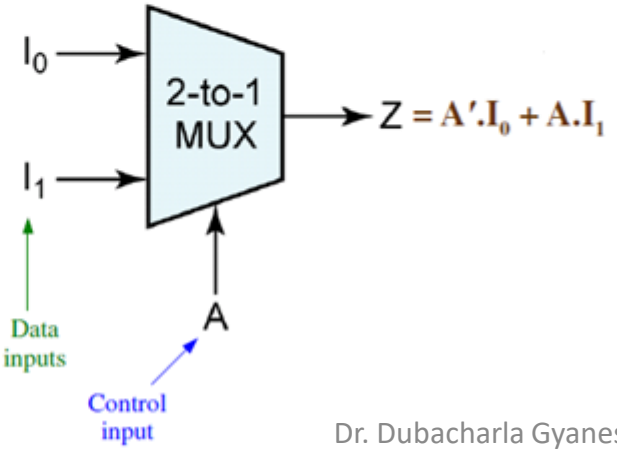
Circuit diagram



2X1 MUX

Truth Table

S_0	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1



Verilog: Source Code and Testbench

```
1  module mux_2_1(
2    input sel,
3    input i0, i1,
4    output y);
5
6    assign y = sel ? i1 : i0;
7  endmodule
```

```
module <testbench_name>;
Example: module mux_tb;
```

- Instantiate top-level design and connect DUT port interface with testbench variables or signals.

```
<dut_module> <dut_inst> (<TB signals>)
mux_2_1 mux(.sel(sel), .i0(i0), .i1(i1), .y(y));
or
mux_2_1 mux(sel, i0, i1, y);
```

```
1  module mux_tb;
2    reg i0, i1, sel;
3    wire y;
4
5    mux_2_1 mux(sel, i0, i1, y);
6    initial begin
7      $monitor("sel = %h: i0 = %h, i1 = %h --> y = %h", sel, i0, i1, y);
8      i0 = 0; i1 = 1;
9      sel = 0;
10     #1;
11     sel = 1;
12   end
13 endmodule
```

Verilog: Example

Write a Verilog module for a 4-bit adder that takes two 4-bit inputs (`a` and `b`) and produces a 4-bit sum (`sum`) and a carry-out (`carry_out`). Also, write a testbench to verify the functionality of this adder.

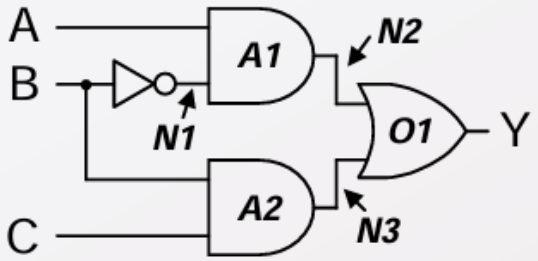
Verilog: Example

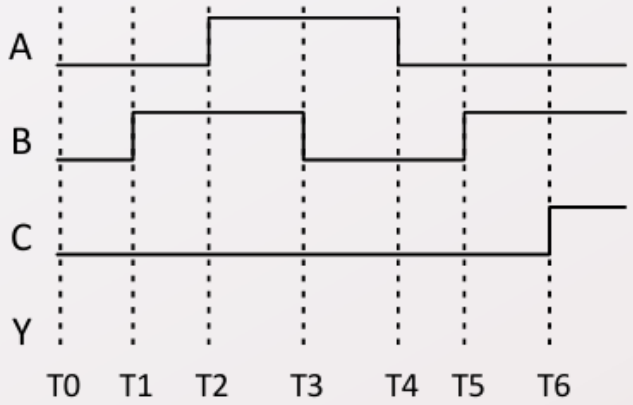
```
module adder4bit (
  input [3:0] a,
  input [3:0] b,
  output [3:0] sum,
  output carry_out
);
  assign {carry_out, sum} = a + b;
endmodule
```

```
module tb_adder4bit;
  reg [3:0] a;
  reg [3:0] b;
  wire [3:0] sum;
  wire carry_out;

  adder4bit uut (
    .a(a),
    .b(b),
    .sum(sum),
    .carry_out(carry_out)
  );

  initial begin
    // Initialize inputs
    a = 4'b0000; b = 4'b0000;
    #10 a = 4'b0010; b = 4'b0011;  // Test case 1
    #10 a = 4'b1111; b = 4'b0001;  // Test case 2
    #10 a = 4'b1001; b = 4'b0110;  // Test case 3
  end

  initial begin
    $monitor("At time %t, a = %b, b = %b, sum = %b, carry_out = %b",
             $time, a, b, sum, carry_out);
  end
endmodule
```

Concept: Sequential vs. Concurrent Models

- A sequential processing algorithm defines a set of steps that are taken in a specific order
- Concurrent processing steps occur whenever new input data is available, with no implied sequence relationship between separate concurrent processes
- Consider a simulation of a 2-input mux: At what time(s) should gate A1 be simulated? What about gate O1?



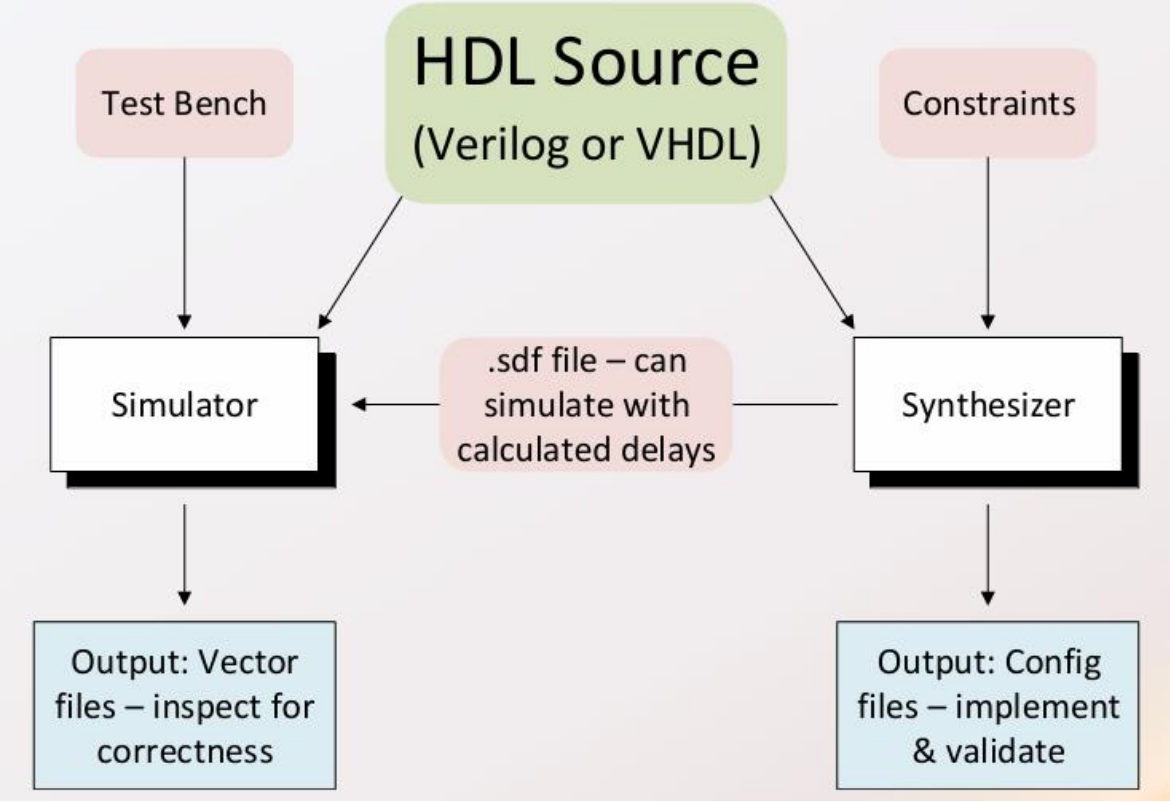


- Computer time vs. Simulation time

VHDL cycle

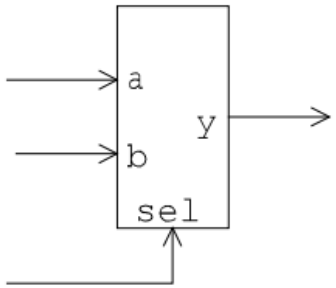
VHDL
Very High Speed Integrated Circuit Hardware Description Language

- VHDL and Verilog descriptions can be simulated to check for logical correctness, and synthesized to automatically create a physical circuit definition.



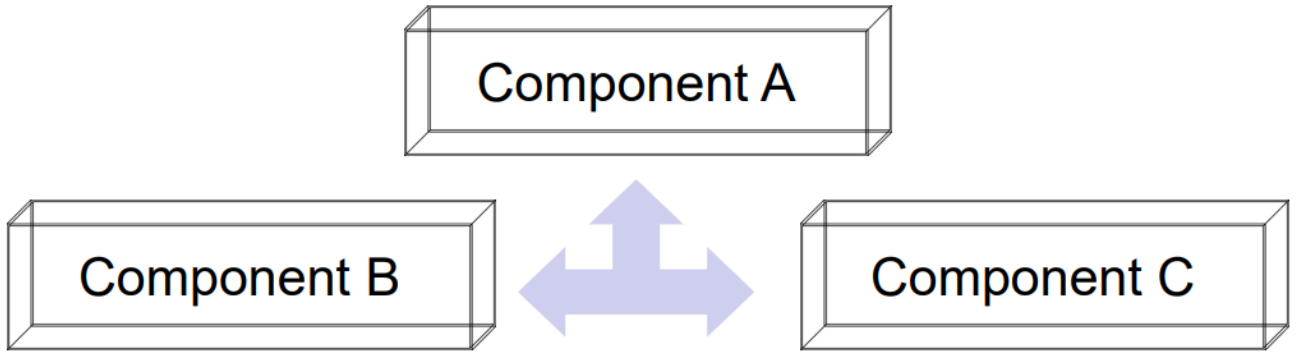
- A way of ***describing*** the operation of a system.
  - Example: a 2-input multiplexer

```
ENTITY mux IS
  PORT (a, b, sel :  IN std_logic;
        y :  OUT std_logic);
END mux;

ARCHITECTURE behavior OF mux IS
BEGIN

  y <= a WHEN sel = '0' ELSE
       b;

END behavior;
```



VHDL

- VHDL uses top-down approach to partition a design into small building blocks (i.e., **`components`**).
  - **Entity**: Describe interface signals and basic building blocks.
  - **Architecture**: Describe behavior, each entity can have multiple Architectures.

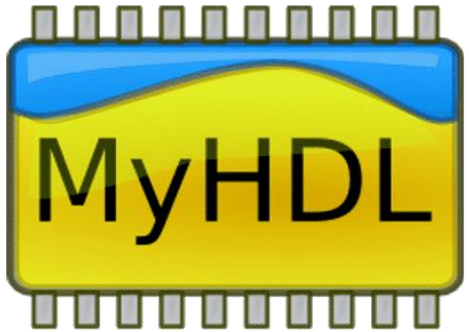


Connected by **`port map`** in architecture body

Popularity	
VHDL is more popular with European companies.	**Verilog** is more popular with US companies.

Programming Style (Syntax)	
VHDL is similar to Ada programming language.	**Verilog** is similar to C/Pascal programming language.
VHDL is NOT case-sensitive.	**Verilog** is case-sensitive.
VHDL is more "verbose" than **Verilog**.	

Tool for experimenting with Verilog & VHDL



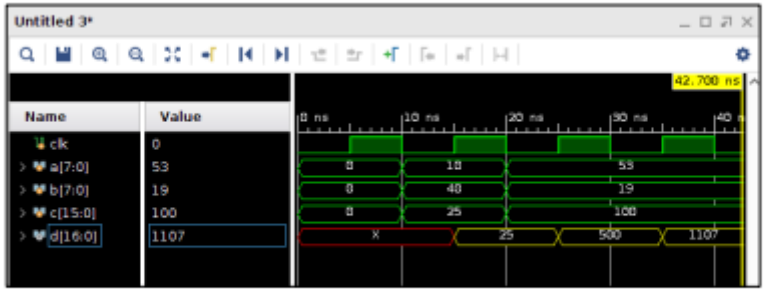
Tool: Xilinx Vivado

Create a Simulation Source

- Open design1_tb.v and write the following example Verilog testbench code and save it.

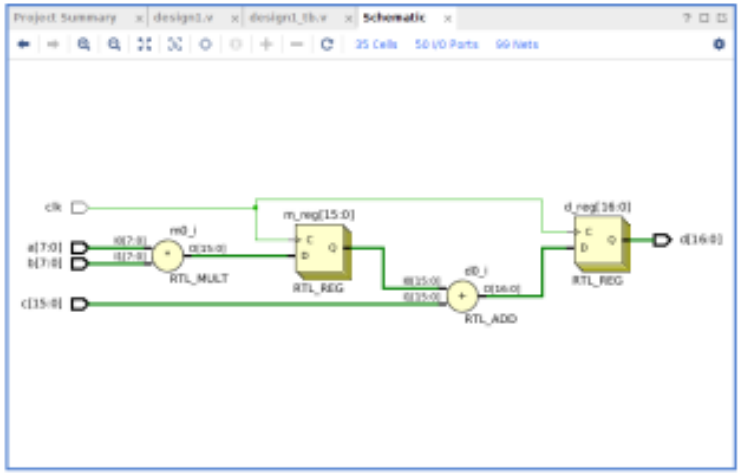
```
` timescale 1ns / 1ps

module design1_tb();

reg [7:0] a,b;
reg [15:0] c;
wire [16:0] d;

design1 dut (a,b,c,d);

initial begin
      a=0; b=0; c=0;
      #10;
      a=10; b=40; c=25;
      #10;
      a=53; b=19; c=100;
end

endmodule
```

Behavioral Simulation



Design Elaboration

- Design Elaboration and RTL Analysis
  - It compiles RTL code and loads RTL netlist
  - You can check RTL structure, syntax, and logic definitions
  - You can view the schematic of your design



Q&A session

Source code and Testbench code of Half adder using Verilog and VHDL
?

Q&A session

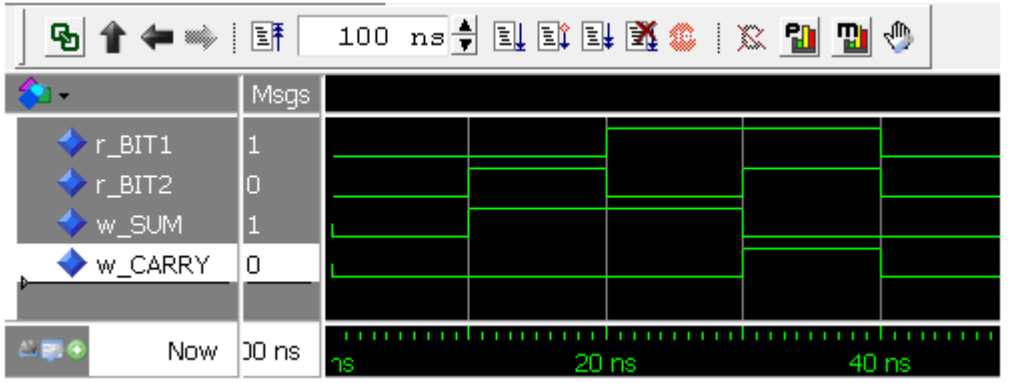
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity half_adder is
  port (
    i_bit1  : in std_logic;
    i_bit2  : in std_logic;
    --
    o_sum   : out std_logic;
    o_carry : out std_logic
    );
end half_adder;

architecture rtl of half_adder is
begin
  o_sum   <= i_bit1 xor i_bit2;
  o_carry <= i_bit1 and i_bit2;
end rtl;
```

```verilog
module half_adder
  (
   i_bit1,
   i_bit2,
   o_sum,
   o_carry
   );

  input  i_bit1;
  input  i_bit2;
  output o_sum;
  output o_carry;

  assign o_sum   = i_bit1 ^ i_bit2;  // bitwise xor
  assign o_carry = i_bit1 & i_bit2;  // bitwise and

endmodule // half_adder
```

Q&A session

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity half_adder_tb is
end half_adder_tb;

architecture behave of half_adder_tb is
  signal r_BIT1  : std_logic := '0';
  signal r_BIT2  : std_logic := '0';
  signal w_SUM   : std_logic;
  signal w_CARRY : std_logic;
begin

  UUT : entity work.half_adder  -- uses default binding
    port map (
      i_bit1  => r_BIT1,
      i_bit2  => r_BIT2,
      o_sum   => w_SUM,
      o_carry => w_CARRY
      );
```

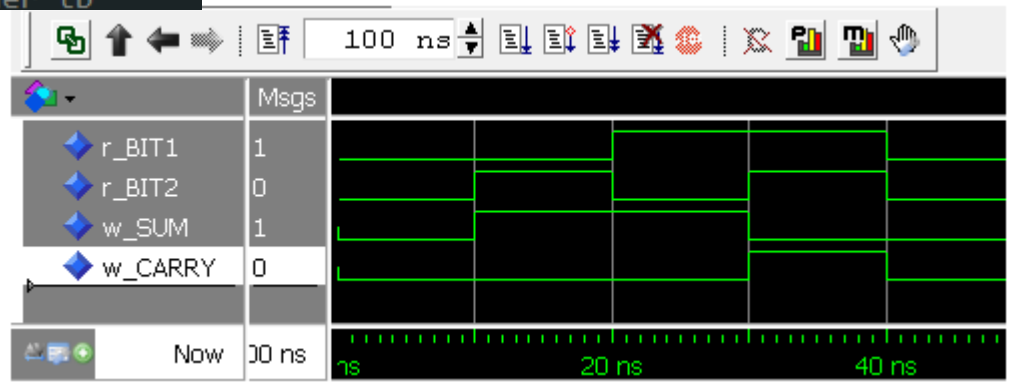
```vhdl
    process is
    begin
      r_BIT1 <= '0';
      r_BIT2 <= '0';
      wait for 10 ns;
      r_BIT1 <= '0';
      r_BIT2 <= '1';
      wait for 10 ns;
      r_BIT1 <= '1';
      r_BIT2 <= '0';
      wait for 10 ns;
      r_BIT1 <= '1';
      r_BIT2 <= '1';
      wait for 10 ns;
    end process;
  end behave;
```



Q&A session

```
module half_adder_tb;

  reg r_BIT1 = 0;
  reg r_BIT2 = 0;
  wire w_SUM;
  wire w_CARRY;

  half_adder half_adder_inst
    (
     .i_bit1(r_BIT1),
     .i_bit2(r_BIT2),
     .o_sum(w_SUM),
     .o_carry(w_CARRY)
     );
```

```
  initial
    begin
      r_BIT1 = 1'b0;
      r_BIT2 = 1'b0;
      #10;
      r_BIT1 = 1'b0;
      r_BIT2 = 1'b1;
      #10;
      r_BIT1 = 1'b1;
      r_BIT2 = 1'b0;
      #10;
      r_BIT1 = 1'b1;
      r_BIT2 = 1'b1;
      #10;
    end

endmodule // half_adder_tb
```

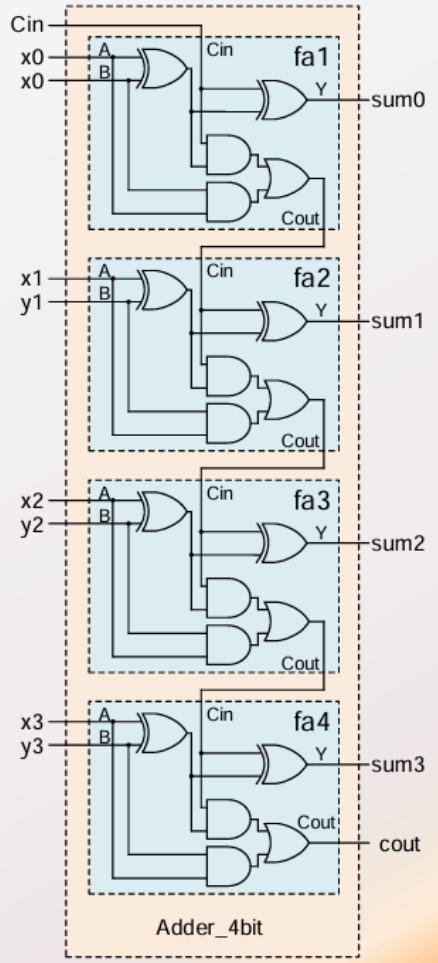


Case studies

Structural Verilog: Using modules as building blocks

```
module full_adder(
    input a,b,cin;
    output y,cout
);
    wire s1,c1,c2;
    xor(s1,a,b);
    xor(y,cin,s1);
    and(c1,s1,cin);
    and(c2,a,b);
    or(cout,c1,c2);
endmodule

module adder_4bit(
    input [3:0] x,y;
    input cin;
    output[3:0] sum;
    output cout
);
    wire c1,c2,c3;
    full_adder fa1(.a(x[0]),.b(y[0]),.cin(cin),.y(sum[0]),.cout(c1));
    full_adder fa2(.a(x[1]),.b(y[1]),.cin(c1),.y(sum[1]),.cout(c2));
    full_adder fa3(.a(x[2]),.b(y[2]),.cin(c2),.y(sum[2]),.cout(c3));
    full_adder fa4(.a(x[3]),.b(y[3]),.cin(c3),.y(sum[3]),.cout(cout));
endmodule
```

This example uses the Verilog “built in” logic modules of and, or, and xor. These module instantiations use “positional association”, which means the order of signals must match in the module and in the instantiation – they are matched left to right. In this case, the format is output name, followed by inputs.



Verilog

Case studies

```
Module decoder_3_8(
   input[2:0] I,
   output[7:0] Y
);
reg [7:0] Y;
always @(I)
begin
   case(I)
      3'd0: Y = 8'd1;
      3'd1: Y = 8'd2;
      3'd2: Y = 8'd4;
      3'd3: Y = 8'd8;
      3'd4: Y = 8'd16;
      3'd5: Y = 8'd32;
      3'd6: Y = 8'd64;
      3'd7: Y = 8'd128;
      default: 8'd0;
   endcase
end;
```

ALU Example

```
module alu(
   input [7:0] A,B,
   input [3:0] Sel,
   output [7:0] Y
);
reg [7:0] Y;
always @ (Sel, A, B) begin
 case (Sel)
      3'd0: Y = A + B;
      3'd1: Y = A - B;
      3'd2: Y = A + 1;
      3'd3: Y = A - 1;
      3'd4: Y = A | B;
      3'd5: Y = A & B;
      3'd6: Y = A ^ B;
      3'd7: Y = ~A;
      default: Y = 7'd0;
   endcase
end
```

Case studies

Verilog Code for 1MB BRAM

```
module BRAM_1MB (
    input wire clk,                 // Clock input
    input wire [19:0] addr,         // 20-bit address for
1MB (2^20 bytes)
    input wire [7:0] din,           // 8-bit data input
    input wire we,                  // Write enable
    output reg [7:0] dout           // 8-bit data output
);
    // Declare the memory array (1M x 8 bits)
    reg [7:0] memory_array [0:1048575]; // 2^20 =
1,048,576 addresses
```

```
always @(posedge clk) begin
        if (we) begin
            // Write operation
            memory_array[addr] <= din;
        end else begin
            // Read operation
            dout <= memory_array[addr];
        end
    end
endmodule
```

Q&A session

?

Thank you!