Chapter 3

RTL and HDL

Assistant Professor

Er. Shiva Ram Dam

Contents:

1. Micro-operations and RTL
2. Using RTL to specify a Digital System
3. Specification of Digital Components
4. Specification and Implementation of Simple System
5. Introduction to VHDL: Syntax. Levels of Abstraction in Design

3.1 Micro-operations and RTL

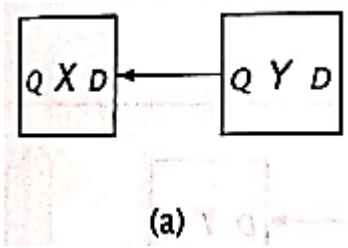
- Micro-operations are the basis for most sequential digital systems.
- Micro-operations are simpler actions like:
  - Movement (copying) of data form one register, memory location, or I/O device to another
  - Modifying stored values
  - Performing arithmetic or logical functions.
- It is also necessary to specify the conditions under which each transfers occurs.

Micro-operations

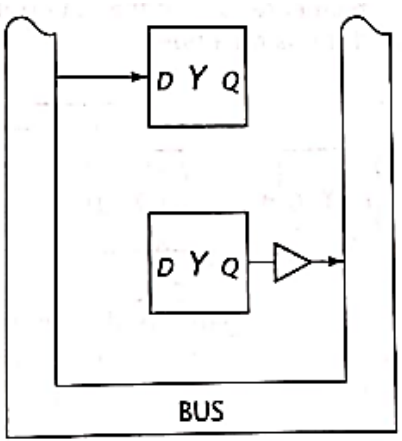
- A Micro-operation , or **μ-**op, specifies an operation whose result os stored, typically in a register or memory location.
- The operation can be as simple as copying data from one register to another, or more complex, such as adding the contents of two registers and storing the result in a third register,
- When designing a sequential digital system, the designer can specify the behavior of the system first, using micro-operations and then design the hardware to match this specification.

Implementations of micro-operation X←Y

- Consider a digital system with two 1-bit registers, X and Y. The µop that copies the contents of register Y to register X can be expressed as X←Y.
- Sometimes, this is expressed as Y→X.
- The micro-operation may be implemented via a direct connection, or by way of a bus.
- Both are valid implementation.



(a)



(b)

Implementations of the micro-operation X←Y using (a) a direct connection and (b) a bus connection

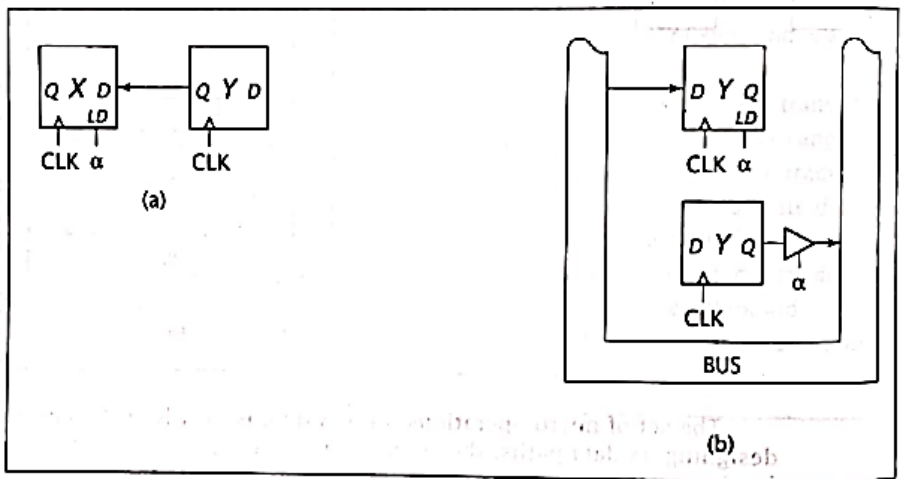
Implementations of the data transfer α : X←Y

- Both designs provide a path for data to flow from register Y to register X, but neither specifies when X should load this data.
- Assume that the transfer should occur when control input α is high.
- The transfer could be written as:

  IF α THEN X←Y
- This can be written as:

  α: X←Y
- When all conditions to the left of the colon are asserted, the data transfers specified by the µops are performed.
- α is used to load register X and, in the bus-based implementation, to enable the tri-state buffer so that the contents of register Y are placed on the bus.

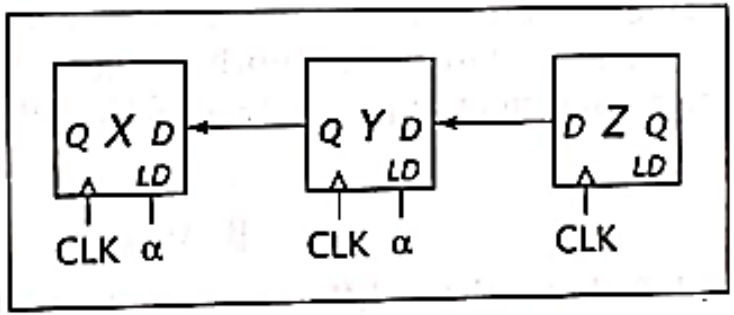
Implementations of the data transfer α: X←Y with control signals: (a) with direct path, and (b) using a bus



Implementations of the data transfer α : X←Y, Y←Z

- One way to improve system performance is to perform two or more µops simultaneously.
- The µops are separated by commas; the order in which they are written is unimportant because they are performed concurrently.
- Consider:

  α: X←Y, Y←Z or α: Y←Z, X←Y
- If X=0, Y=1 and Z=0 just before α becomes 1, these µops set X=1(the original value of Y) and Y=0.
- Note that a single bus cannot be used here because a bus can hold only one value at a time.
- When α=1, both Y and Z must travel on the data paths simultaneously.



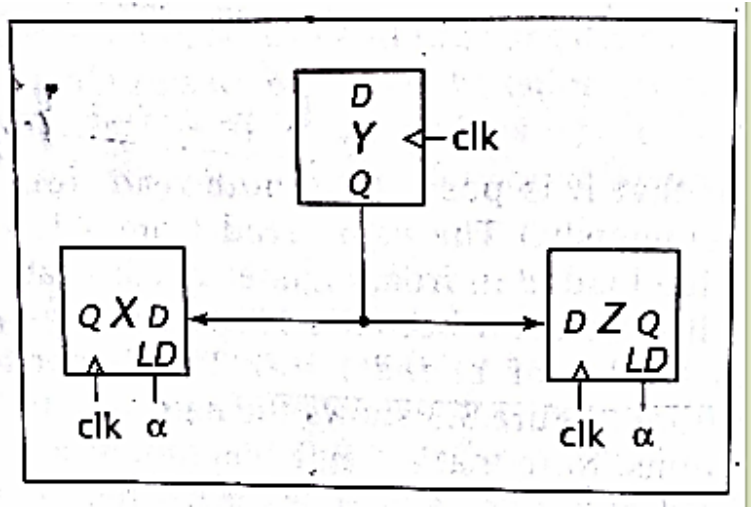
Implementation of the data transfer α: X←Y, Y←Z

Implementations of the data transfer α : X←Y , Z←Y

- Consider the transfers that occur when α = 1.

  α: X←Y, Z←Y

- Register Y can be read by many other registers simultaneously; both micro operations can be performed concurrently.



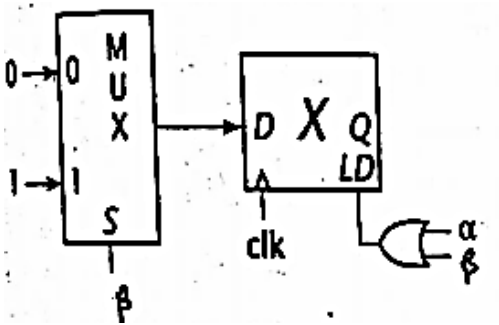
Implementation of the data transfer α: X←Y, Z←Y

Three implementations of the data transfers
α : X← 0 and β: X←1

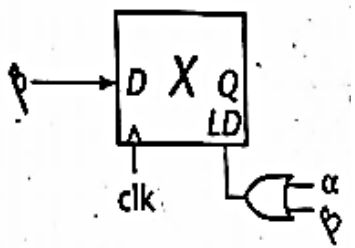
- Sometimes it is necessary to move a constant value into a register, rather than data from another register.

  α: X←0

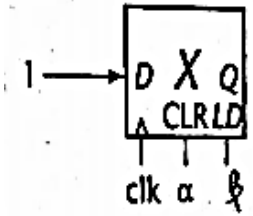
  β: X←1
- The first circuit sets X by loading data for both transfers.
  - Either α or β can assert the register's load signal, and the MUX ensures that the proper data is made available.
  - If α=0 and β=0, the MUX still outputs data 0. Register X ignores this data and retains its value.
- The second circuit loads data exactly same as in the first circuit;
- However, its data is generated directly by signal β.
  - When , β=0 and α=1, register X loads 0
  - When , β=1 and α=0, register X loads 1
  - When , β=0 and α=0, register X ignores the input 0, and retains its value.
- The final circuit makes use of the register's clear input to simplify the hardware.
  - When α=1, register X is cleared, i.e set to 0.
  - When β=1, the register is loaded with the data value 1 as before.



a) Using a multiplexer to select the data input



b) Using β as the data input



c) Using the CLR signal

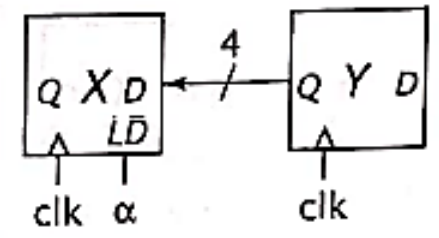
Implementations of the 4-bit data transfer α:X← Y

- All of the operations so far dealt are with individual 1-bit registers.
- Real systems use multibit registers to store larger data values.
- For eg: consider the case, when α=1 in which date is transferred from 4-bit register Y to 4-bit register X, this can be represented as: α: X← Y
- Sometimes it is necessary to access individual bits or groups of bits within a register. Individual bits can be referred to by subscripts, such as X_3 or Y_2.
- Groups of bits cane be referred to as ranges in RTL as X(3-1) or X(3:1)
- Below statements are valid:

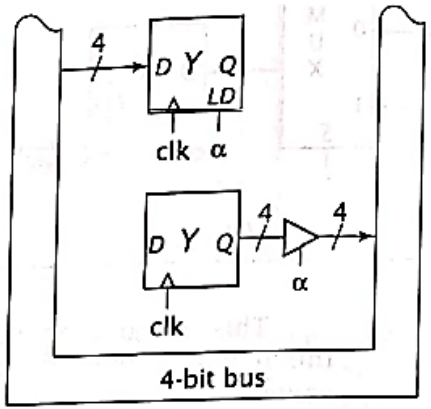
  α : X(3-1) ← Y(2-0)

  β : X_3←X_2

  τ : X(3-0) ← X(2-0), X_3



a) Using a direct connection



b) Using a bus

Types of Micro-operations

- In digital computer, there are four types of micro-operations:
  1. Arithmetic Micro-operations
  2. Register Transfer Micro-operations
  3. Logical micro-operations
  4. Shift Micro-operations

1. Arithmetic micro-operation

- Used to perform some basic arithmetic operation.
- Eg. R1 ← R1+R2
- R ← R+1
- R ← R-1
- R3 ← 1's complement of R3 + 1

2. Register transfer micro-operation

- Used to transfer content of register
- Eg. R1 ← R2 content of R2 transferred to R1

3. Logic micro-operation

- Bitwise logical operation are performed; i.e. perform bit manipulation operation on data stored in registers.
- Eg. AND, OR, NOT, EX-OR, EX-NOR, etc.

4. Shift micro-operation

- Serial transfer or bits in a register is called shift operation.
  - Left shift/ right shift
  - Arithmetic right shift
  - Circular shift
  - Decimal Shift

Arithmetic and Logical Micro-operations

Operation	Example
Add	$X \leftarrow X + Y$
Subtract	$X \leftarrow X - Y$ or $X \leftarrow X + Y' + 1$
Increment	$X \leftarrow X + 1$
Decrement	$X \leftarrow X - 1$
AND	$X \leftarrow X \wedge Y$ or $X \leftarrow XY$
OR	$X \leftarrow X \vee Y$
XOR	$X \leftarrow X \oplus Y$
NOT	$X \leftarrow /X$ or $X \leftarrow X'$

Shift Micro-operations

- There are four basic types of shift Micro-operations:
  1. Linear shift
  2. Circular shift
  3. Arithmetic shift
  4. Decimal shift

Operation	Notation
Linear shift left	shl(*X*)
Linear shift right	shr(*X*)
Circular shift left	cil(*X*)
Circular shift right	cir(*X*)
Arithmetic shift left	ashl(*X*)
Arithmetic shift right	ashr(*X*)
Decimal shift left	dshl(*X*)
Decimal shift right	dshr(*X*)

a) Linear / Logical shift

- It transfers 0 by the serial input.
- The symbol **"shl"** can be used for logical shift left and **"shr"** can be used for logical shift right.

  R1 ←R1 shl R1

  R2 ←R1 shr R2

- The register symbol should be the equivalent on both sides of the arrows.

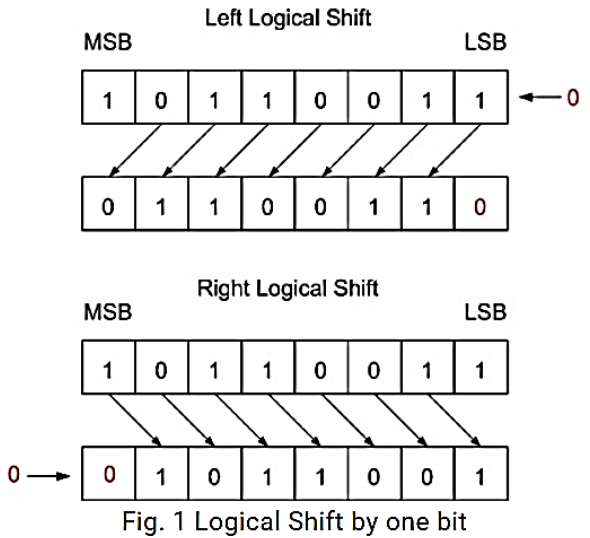
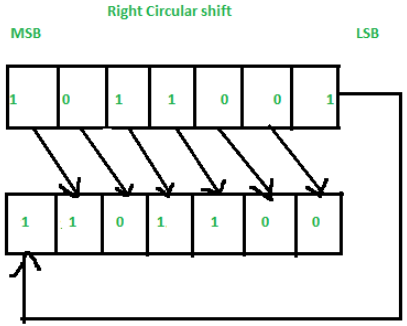


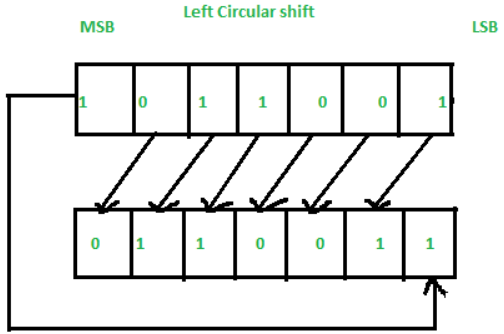
Fig. 1 Logical Shift by one bit

b) Circular shift

- This circulates or pivots the bits of register around the two ends without any trouble of data or contents.
- In circular shift, the serial output of the shift register is linked to its serial input.
- **"cil"** and **"cir"** is used for circular shift left and right respectively.
- The symbolic documentation for the shift micro-operations is demonstrated in the table.

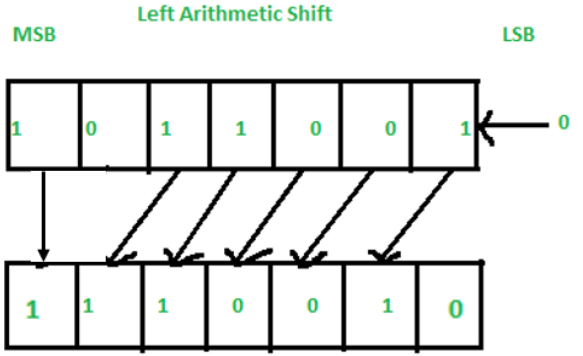
Symbolic Designation	Description
R ←R1 shl R	Shift-left register R
R ←R1 shr R	Shift-right register R
R ←R1 cil R	Circular shift-left register R
R ←R1 cir R	Circular shift-right register R
R ←R1 ashl R	Arithmetic shift-left R
R ←R1 ashr R	Arithmetic shift-right R

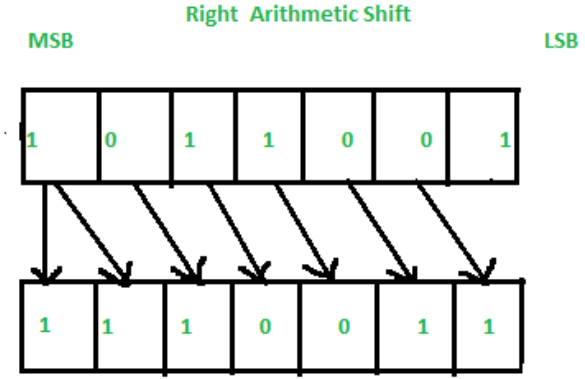




c) Arithmetic shift

- This shift was developed to work with numbers that have signed formats.
- This shifts a signed binary number to left or right.
- In such, the sign bit remains unchanged. Otherwise, it is similar to the logical shift.
- For eg: X=1011
- Arithmetic Left shift yields: 1110. Here, the value of the second MSB is lost.
- Arithmetic Right shift yields: 1101. Here, the LSB is lost.





d) Decimal Shift

- Decimal shift was developed specifically for BCD representation.
- It acts like a linear shift, except it shifts one digit, or four bits, instead of just one bit.
- For eg: X= 1001 0111
- The decimal shift left results in the value 0111 0000
- The decimal shift right results in the value 0000 1001

3.2 Using RTL to specify a Digital System

- RTL can be used to specify the behavior of any sequential digital system, simplest to complex.
- RTL specify the function of digital components and simple systems.

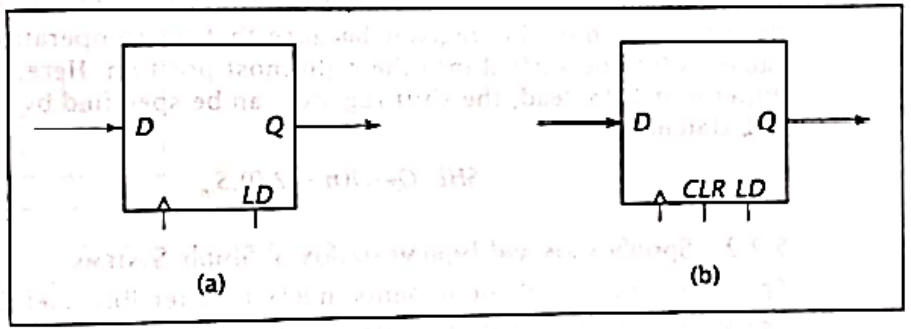
3.3 Specification of Digital Components

- Consider the D flip-flop as shown in figure (a).
- Its function can be expressed by single RTL statement.

  LD: Q←D

- When LD input is high, the value on the D input is loaded and made available on the Q output. This happens on rising edge of clock pulse.

D flip-flop: (a) without and (b) with clear input



Clk	D	Qn+1
0	x	Qn
1	0	0
1	1	1

D Flipflop truth table

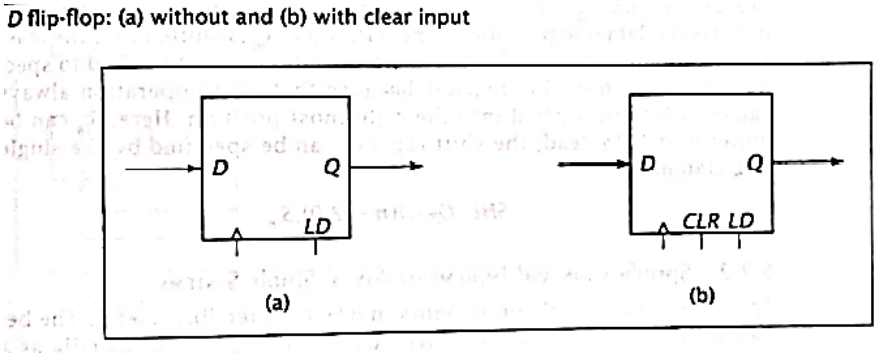
- In figure b, when CLR =1, the flipflop set to 0.

  i.e.

  LD: Q← D

  CLR: Q←0

- However, When D, LD and CLR are equal to 1, the above system fails.
  - When LD=1, the Q is set to 1
  - And when CLR=1, Q is clear to 0, Both set Q simultaneously.
  - Which is not possible.



Clk	D	Qn+1
0	x	Qn
1	0	0
1	1	1

D Flipflop truth table

- The solution is to modify the conditions so they are mutually exclusive.
- Either of the following is valid.

$CLR'LD: Q \leftarrow D$ $\qquad$ $LD: Q \leftarrow D$

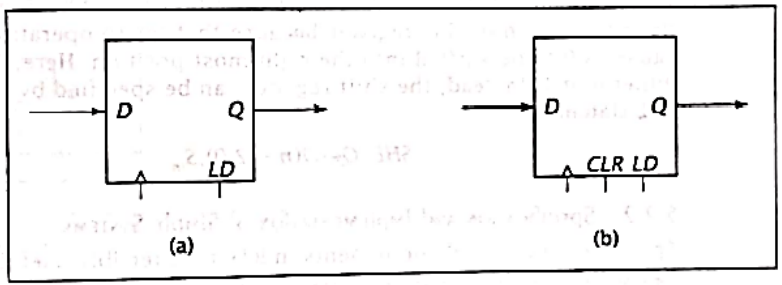
$CLR: Q \leftarrow 0$ $\qquad$ $LD'CLR: Q \leftarrow 0$

- In this RTL code, note the combined condition in the RTL statement.

$$CLR'LD: Q \leftarrow D$$

- It is possible to require more than one condition to be met in order for a micro-operation to occur.
- Here, Q is set to 1 if both CLR=0 and LD=1.

D flip-flop: (a) without and (b) with clear input



Clk	D	Q_{n+1}
0	x	Q_n
1	0	0
1	1	1

D Flipflop truth table

Another Example: JK

- Consider the JK flip flop without a CLR input.
- Its truth table is:

Truth Table

J	K	CLK	Q
0	0	↑	Q_0 (no change)
1	0	↑	1
0	1	↑	0
1	1	↑	$\overline{Q}_0$ (toggles)

- It behavior can be specified in RTL as follow:

$J'K$: $Q \leftarrow 0$
JK': $Q \leftarrow 1$
JK: $Q \leftarrow Q'$

- When J=0 and K=0, no condition is met and the flip-flop retains its previous value. No RTL statement is included for this case because no transfer is needed.

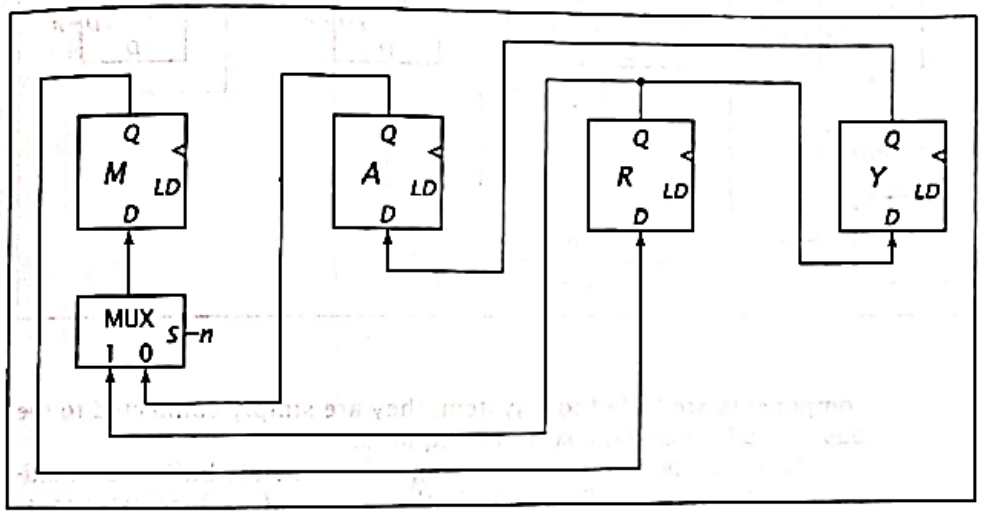
3.4 Specification and Implementation of Simple Systems

- RTL can be used to specify behavior of an entire system , independent of the components used to implement the system.
- Consider a system with four 1-bit flipflops. Given RTL code where j, o, h and n are conditions which are mutually exclusive.

j: $M \leftarrow A$
o: $A \leftarrow Y$
h: $R \leftarrow M$
n: $Y \leftarrow R$, $M \leftarrow R$

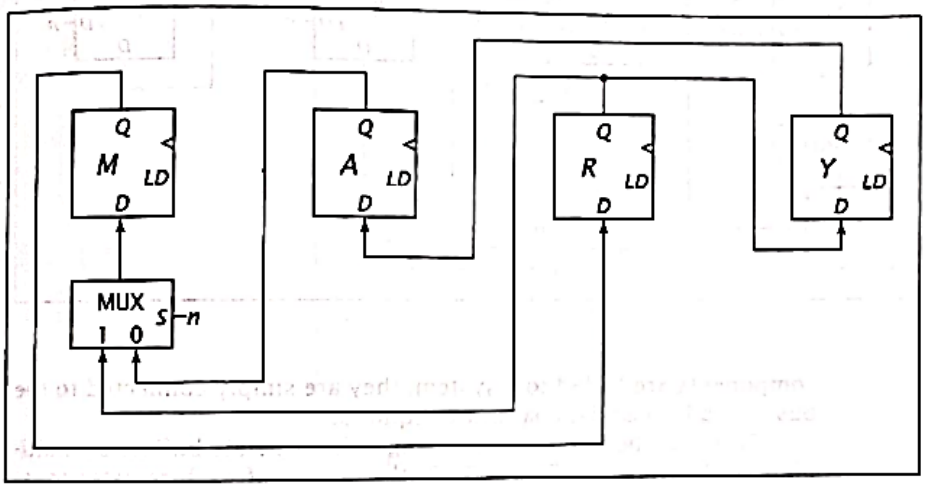
- This RTL can be implemented using direct connections.

j: $M \leftarrow A$
o: $A \leftarrow Y$
h: $R \leftarrow M$
n: $Y \leftarrow R$, $M \leftarrow R$



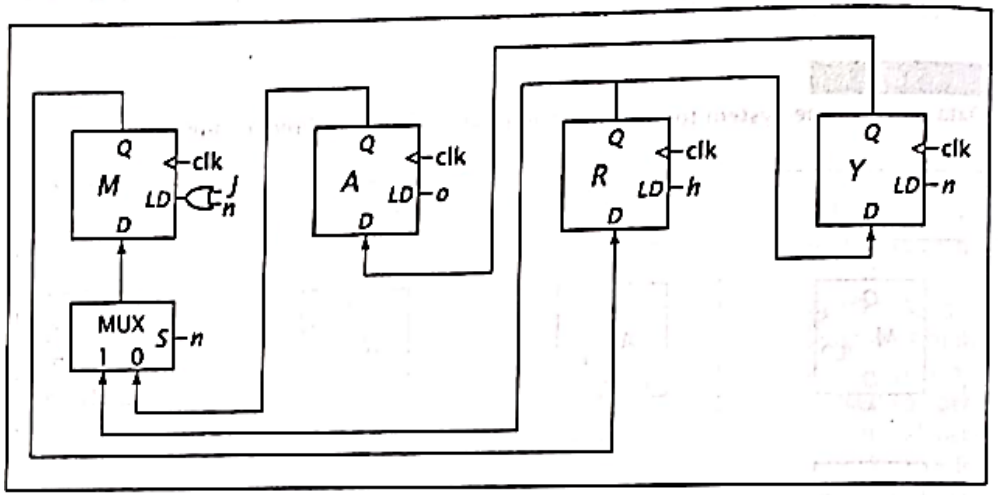
Data paths of the system to implement the RTL code using direct connections

- But it does not complete the transfer.
- No register has actually loaded any data yet.
- It is necessary to assert LD signal.



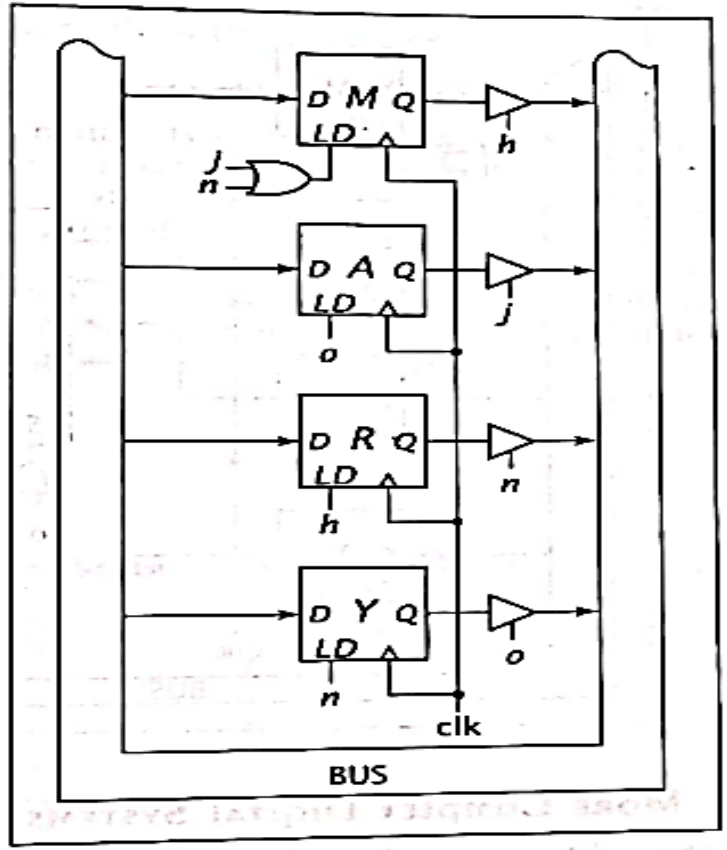
Data paths of the system to implement the RTL code using direct connections

- The complete circuit, with LD signal is shown as below.
- The flipflop M is loaded when j=1 (M←A) or n=1 (M←R).
- Logically, ORing these signals together produces LD input for M flipflop.
- Similarly for other conditions.



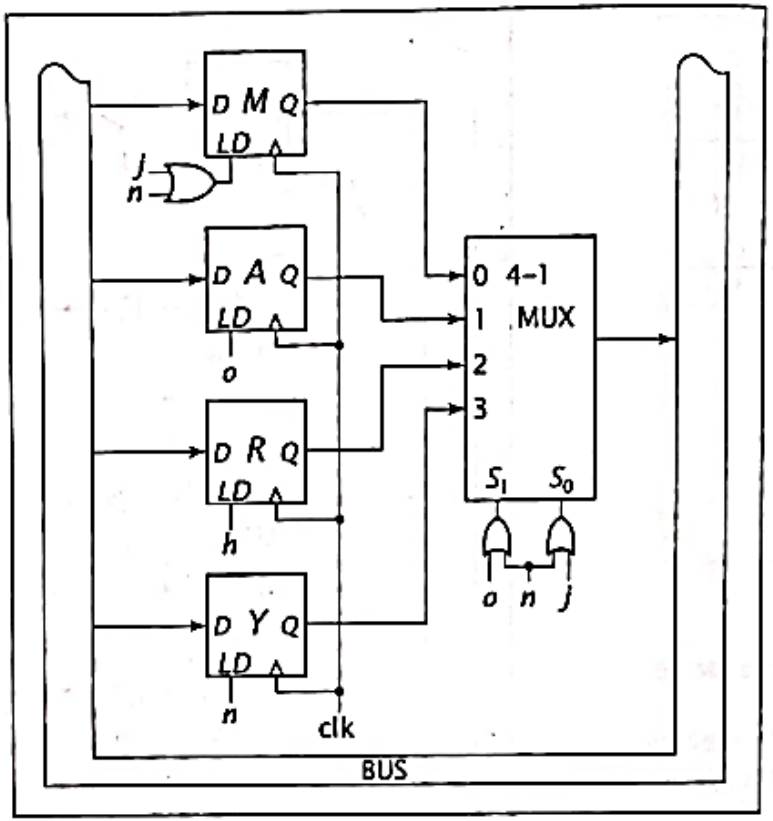
Complete design of the system to implement the RTL code using direct connections

- Buses can be implemented using either tri-state buffers or multiplexers.
- Figure aside shows the complete design of the system to implement RTL code using a bus and tri-state buffers.



Complete design of the system to implement the RTL code using a bus and tri-state buffers

- It is also possible to implement a system bus using a multiplexer.
- MUX can be used instead of tri-state buffers.



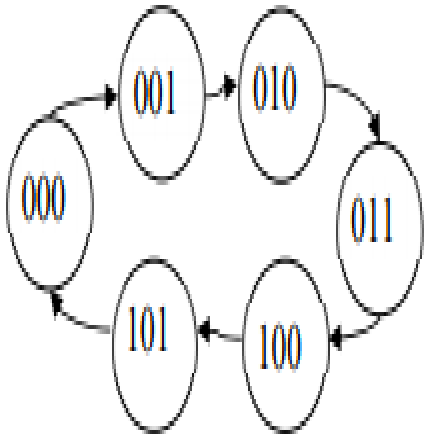
Complete design of the system to implement the RTL code using a bus and multiplexer

Modulo 6 Counter

- To design a modulo 6 counter, first, we need to specify the function of counter using RTL.
- Then, we implement RTL code using digital logic.
- The modulo 6 counter is a 3 bit counter that counts through the sequence:

000→001→010→011→100→101→000→... (0→1→2→3→4→5→0→...)

- Its input U controls the counter.
  - When U=1, the counter increments its value on the rising edge of the clock.
  - When U=0, it retains its current value of the clock.
- The value of count is represented as 3-bit output $V_2V_1V_0$.
- An additional output, C is 1 when going from 5 to 0, and 0 otherwise.
- In this example, C output remains at 1 until the counter goes from 0 to 1.

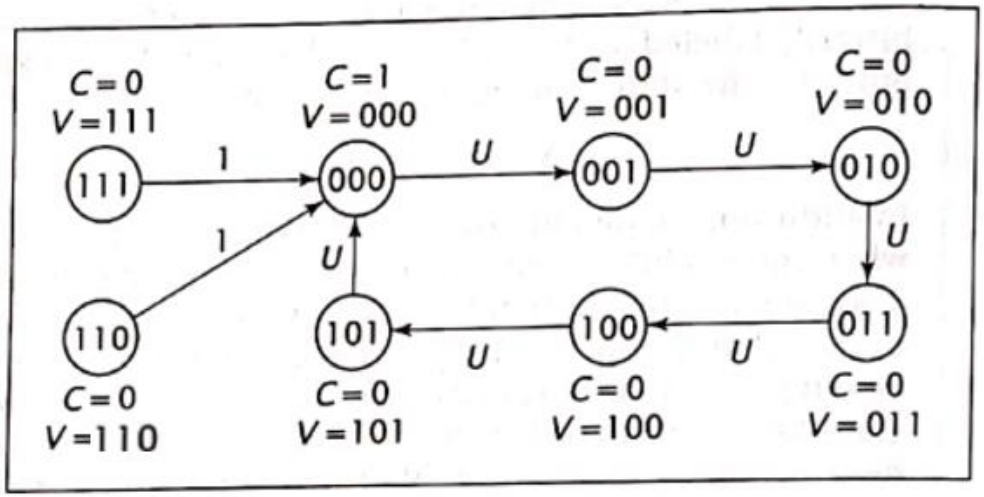


- State table for Modulo 6 counter:
- The additional states S6 and S7 are to handle the case when mod-6 counter powers up in invalid state.

Present State		U	Next State	$V_2V_1V_0$	C
S_0	000	0	S_0	000	1
S_0	000	1	S_1	001	0
S_1	001	0	S_1	001	0
S_1	001	1	S_2	010	0
S_2	010	0	S_2	010	0
S_2	010	1	S_3	011	0
S_3	011	0	S_3	011	0
S_3	011	1	S_4	100	0
S_4	100	0	S_4	100	0
S_4	100	1	S_5	101	0
S_5	101	0	S_5	101	0
S_5	101	1	S_0	000	1
S_6		X	S_0	000	1
S_7		X	S_0	000	1

State diagram

Present State		U	Next State	$V_2V_1V_0$	C
S_0	000	0	S_0	000	1
S_0	000	1	S_1	001	0
S_1	001	0	S_1	001	0
S_1	001	1	S_2	010	0
S_2	010	0	S_2	010	0
S_2	010	1	S_3	011	0
S_3	011	0	S_3	011	0
S_3	011	1	S_4	100	0
S_4	100	0	S_4	100	0
S_4	100	1	S_5	101	0
S_5	101	0	S_5	101	0
S_5	101	1	S_0	000	1
S_6		X	S_0	000	1
S_7		X	S_0	000	1



- When the counter's value is 000 to 100 and its U(up) signal is asserted, the output of the counter is incremented.
- This corresponds to condition $(S_0+S_1+S_2+S_3+S_4)U$.
- Under this condition, C is also set to 0.

  So **$V \leftarrow V+1, C \leftarrow 0$**

  Thus **$(S_0+S_1+S_2+S_3+S_4)U : V \leftarrow V+1, C \leftarrow 0$**
- When counter is in state S_5 (V=101) and U=1, the counter must be reset to 000 and C must be 1,

  i.e. **$S_5\ U : V \leftarrow 0, C \leftarrow 1$**
- For invalid state, regardless of value of U,

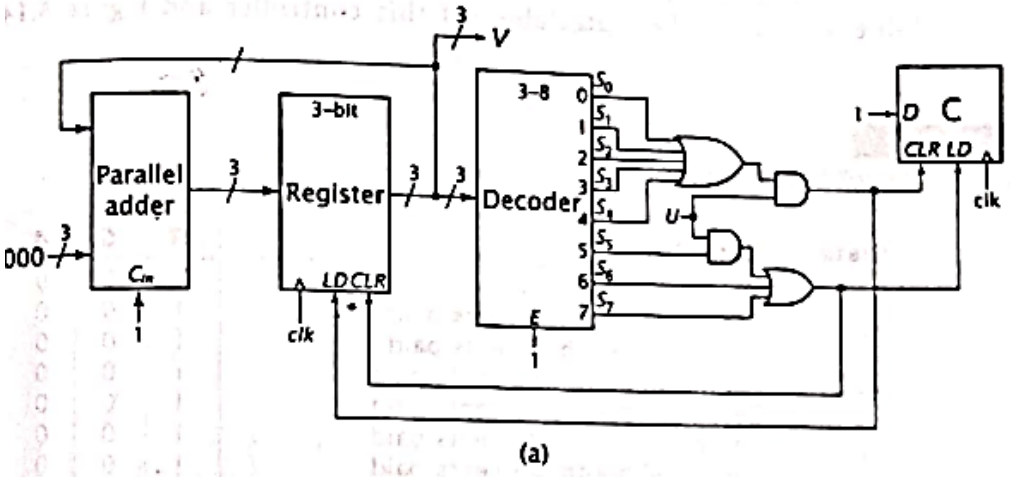
  $(S_6+S_7) : V \leftarrow 0, C \leftarrow 1$
- Thus the behavior of the modulo 6 counter can be expressed by RTL as:

  $(S_0+S_1+S_2+S_3+S_4)U : V \leftarrow V+1, C \leftarrow 0$

  $S_5\ U + S_6+S_7 : V \leftarrow 0, C \leftarrow 1$

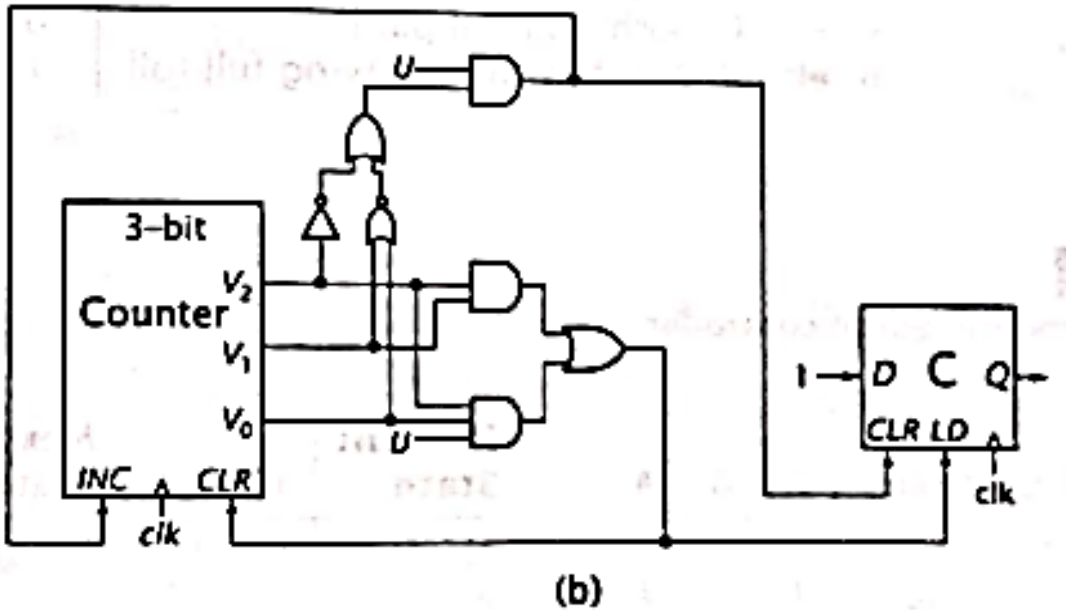
- Figure shows the RTL code implementation using a register.
- This uses a 3-bit parallel adder to generate V+1 and sets C separately.
- Notice that the decoder converts $V_2V_1V_0$ to the state values S0 to S7.

$(S_0+S_1+S_2+S_3+S_4)U$: V←V+1, C←0
$S_5U+S_6+S_7$: V←0, C←1



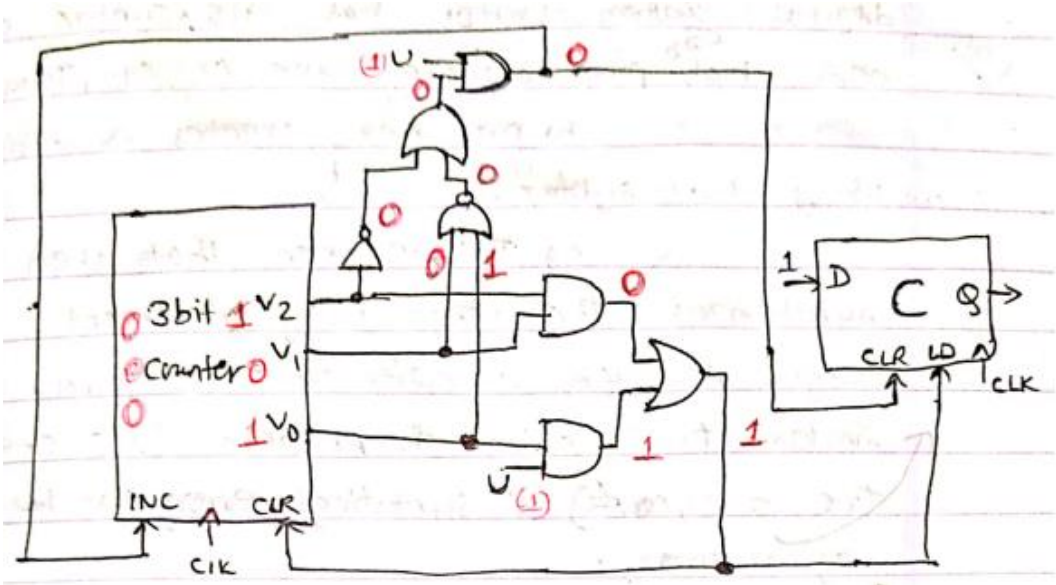
Two implementations of the RTL code for the modulo 6 counter: (a) using a register, and (b) using a counter

- Another way of implementation is the use of a counter.
- This implementation uses a 3-bit counter.
- C is set to 1 either when counting from 101 to 000 or when it is reset from an invalid state.



Two implementations of the RTL code for the modulo 6 counter: (a) using a register, and (b) using a counter

Present State		U	Next State	$V_2V_1V_0$	C
S_0	000	0	S_0	000	1
S_0	000	1	S_1	001	0
S_1	001	0	S_1	001	0
S_1	001	1	S_2	010	0
S_2	010	0	S_2	010	0
S_2	010	1	S_3	011	0
S_3	011	0	S_3	011	0
S_3	011	1	S_4	100	0
S_4	100	0	S_4	100	0
S_4	100	1	S_5	101	0
S_5	101	0	S_5	101	0
S_5	101	1	S_0	000	1
S_6		X	S_0	000	1
S_7		X	S_0	000	1



Problems and Solutions

1 Write valid RTL statements that realize the following transitions. All registers are 1-bit wide.

a) IF $\alpha = 1$ THEN copy X to W and copy Z to Y
b) IF $\alpha = 1$ THEN copy X to W; otherwise copy Z to Y
c) IF $\alpha = 0$ THEN copy X to W

a) α: $W \leftarrow X,\ Y \leftarrow Z$

b) α: $W \leftarrow X$

α': $Y \leftarrow Z$

c) α': $W \leftarrow X$

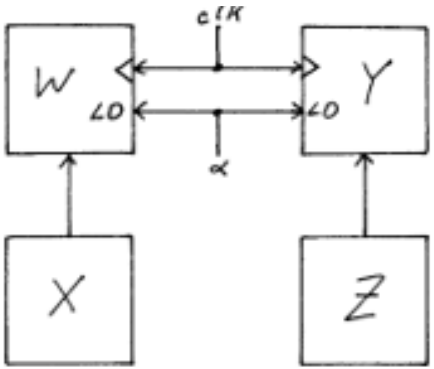
1 Write valid RTL statements that realize the following transitions. All registers are 1-bit wide.

a) IF α = 1 THEN copy *X* to *W* and copy *Z* to *Y*
b) IF α = 1 THEN copy *X* to *W*; otherwise copy *Z* to *Y*
c) IF α = 0 THEN copy *X* to *W*

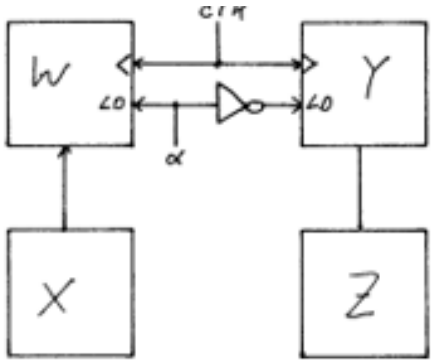
a) $\alpha: W \leftarrow X,\ Y \leftarrow Z$
b) $\alpha: W \leftarrow X$
$\alpha': Y \leftarrow Z$
c) $\alpha': W \leftarrow X$

2 Show the hardware to implement the RTL statements developed for Problem 1.

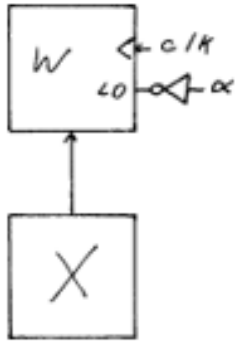
a)



b)



c)



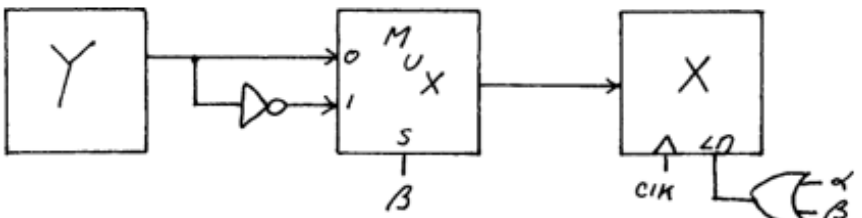
3 Write the RTL code for the following transitions. All registers are 1-bit wide. Signals α and β are never equal to 1 simultaneously.

a) IF α = 1 THEN copy Y to X
   IF β = 1 THEN copy Y' to X
b) IF α = 1 THEN set X to 0
   IF β = 1 THEN set X to X'

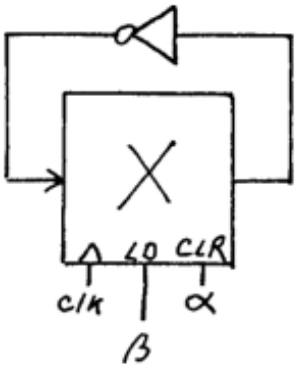
a) $\alpha: X \leftarrow Y$
   $\beta: X \leftarrow Y'$
b) $\alpha: X \leftarrow 0$
   $\beta: X \leftarrow X'$

4 Show the hardware to implement the RTL statements developed for Problem 3.

a)



b)



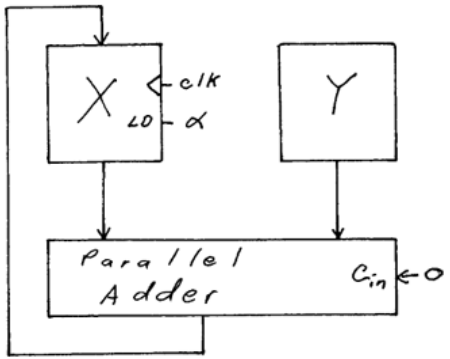
6 Show the hardware to implement the following RTL code.

a) α: $X \leftarrow X + Y$

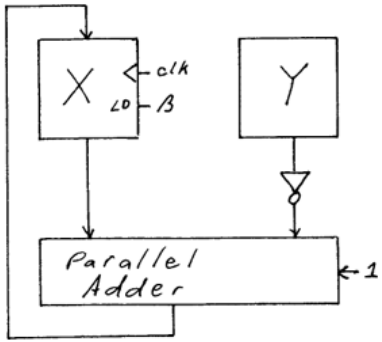
b) β: $X \leftarrow X + Y' + 1$

c) Γ: $X \leftarrow X \oplus Y$

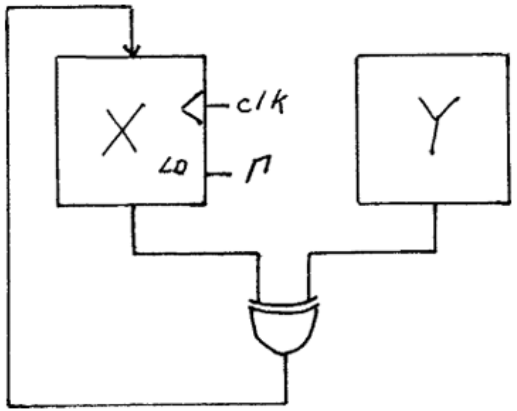
a)



b)



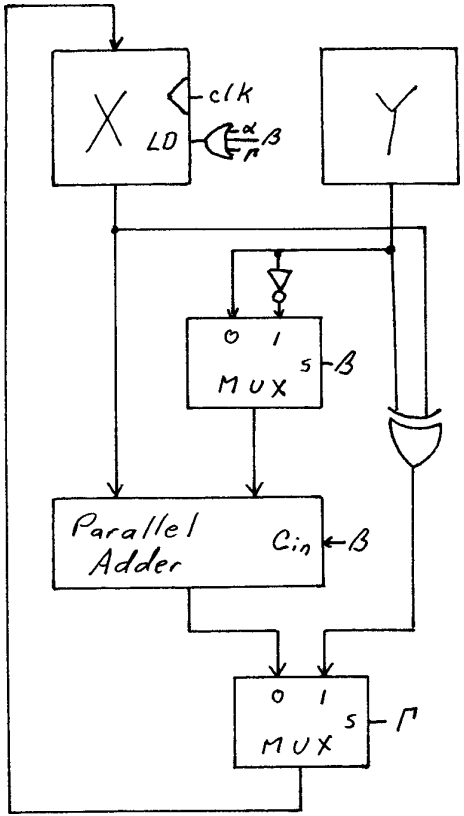
c)



6 Show the hardware to implement the following RTL code.

a) α: $X \leftarrow X + Y$
b) β: $X \leftarrow X + Y' + 1$
c) Γ: $X \leftarrow X \oplus Y$

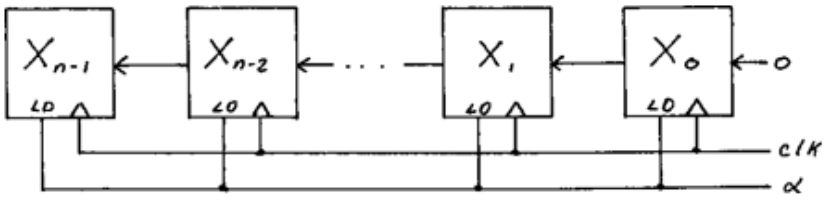
7 Show the hardware to implement all three of the RTL statements of Problem 6 in one combined system. Control hardware ensures that no more than one of the control signals α, β, and Γ is active at any time.



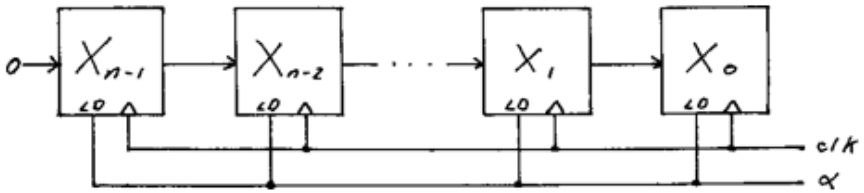
11 Show the hardware to implement the following micro-operations. X consists of four D flip-flops. Each micro-operation occurs when $\alpha = 1$.

a) shl(X)
b) shr(X)
c) cil(X)
d) cir(X)

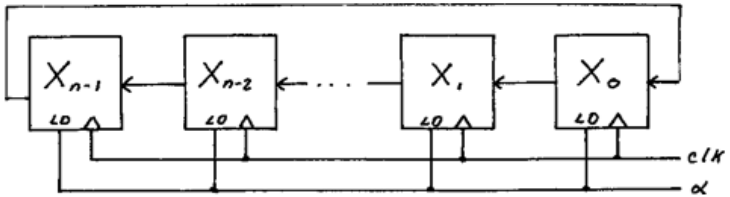
a)



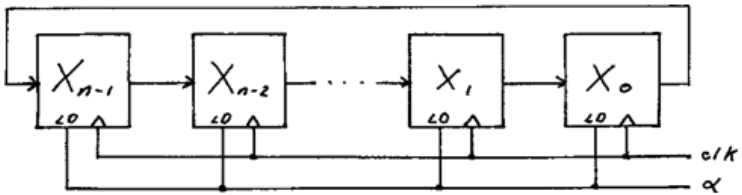
b)



c)



d)



Assignment:

1. Implement the given RTL codes into a single system using bus. . [PU 2017 Fall]

   R1← R2, R3← R2

   R2 ← R3

   R3 ← R1, R2←R1
2. Perform the circular left shift, circular right shift, linear left shift, arithmetic left shift and arithmetic right shift operations on a register holding the value 10100100. [PU 2017 Spring]
3. Define RTL. Perform eight different shift operations on X= 110001010001
4. List out the RTL code for arithmetic and logical operations. Also show the implantation of addition and subtraction using parallel adder. [PU 2018 Fall]
5. Define RTL. Write RTL codes for arithmetic operations with circuitry. [PU 2018 Spring]
6. Design a system that could implement the given RTL codes sing direct connection: [PU 2019 Spring]

   R: x←x+y
   T: x←x-y
   L: y←y^x

VHDL

- Very high speed integrated circuit (VHSIC) Hardware Description Language (VHDL)
- VHDL was developed to provide a standard for designing digital systems.
- VHDL specifies a formal syntax. The designer creates a design file using that syntax just as a programmer writes a C-program.
- The designer then synthesizes the design using only design package that can accept VHDL files. This is equivalent to the programmer compiling the C-program. It checks for errors in syntax and declaration, but not in logic.
- Finally, the designer debugs the design using simulation tools

Advantages of VHDL:

1. **PORTABILITY:** Just as a valid C-program can be compiled by any compliant C compiler, a VHDL design can be synthesized by any design system that supports VHDL.
2. **DEVICE INDEPENDENT:** The VHDL file is device independent. The same file can be used to implement the design on a custom ICs, on ASICs or any PLD that is capable of containing the design.
3. **SIMULATION:** VHDL designs can be simulated by the design system, allowing the designer to verify the design performance before committing it to hardware.
4. **SYSTEM SPECIFICATION:** The designer can design the system using a high level of abstraction, such as a finite state machine, down to a low level digital logic implementation.

Advantages of VHDL:

Summary:

1. VHDL is used to design ICs and application specific ICs (ASICs), PLD.
2. VHDL offers portability. VHDL can be synthesized by any design system supported by VHDL.
3. VHDL files are device independent.
4. VHDL designs can be simulated.
5. VHDL files provide good documentation of system design.

Disadvantages of VHDL:

- VHDL source code often becomes long and difficult to follow, especially at a low level of abstraction.
- It is often less intuitive than a block diagram or RTL description of same system.
- Different design tools may produce different valid design for the same system, especially for high level of abstraction, providing no details about the implementation

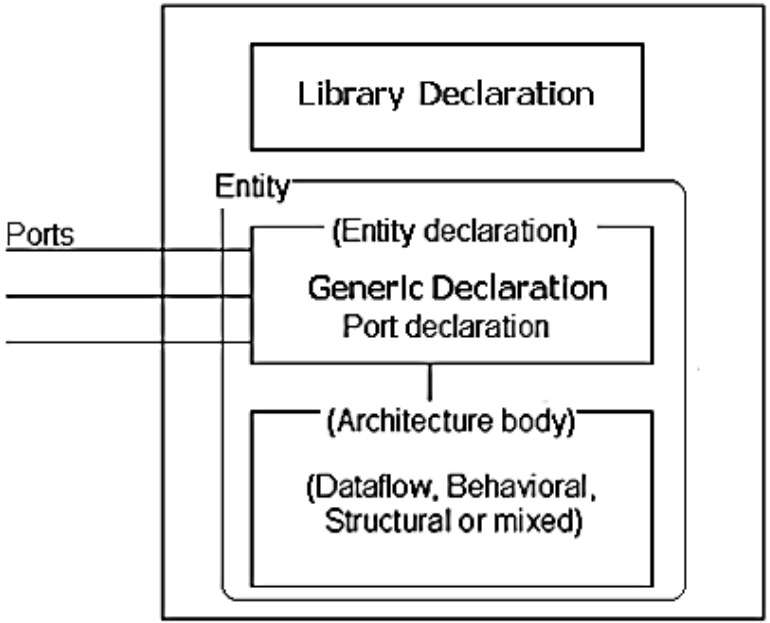
Disadvantages of VHDL:

Summary:

1. VHDL source code often are long and difficult to follow at low level of abstraction.
2. VHDL are difficult to understand.
3. VHDL does not provide detail about its design implementation.

VHDL syntax:

- VHDL design code has 3 primary sections:
  - Library declarations
  - Entity section and
  - Architecture section



Structure of VHDL Code

i) Library declaration:

- It consists of statements that specify libraries to be accessed and modules of these libraries to be used.(such as “include” in C)
- The most commonly used library is called IEEE.
- In this library, std_logic_1164 is used most often, which specifies i/p and o/p declarations for the designer to use (such as “stdio.h” in C).

General Syntax is:

```
library   <library_name>;              //Library IEEE will be used almost in every code
use <library_name> . <package_name> . all;   //This is format for importing package
```

Example:

```
Library IEEE;
use IEEE.std_logic_1164.all;
```

ii) Entity section:

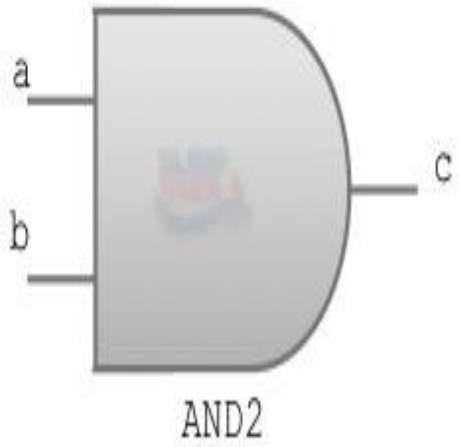
- In the entity section, the designer specifies the name of the design and its inputs and outputs.
- The designer does not specify the logic that uses and drives these signals here; that is done in the architecture section.

General format is:

```
entity <entity_name> is
port
(
    <signal_name> : mode <type>;
    <signal_name> : mode <type> : = default_value;
    <signal-name> : inout <type>;
    <signal-name> : out <type>: = default_value;
);
End <entity_name> ;
```

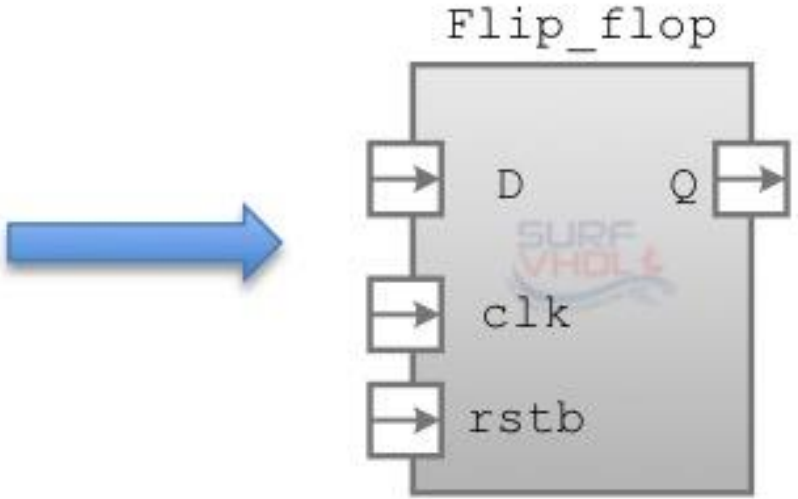
VHDL Entity representing an "**And gate**" with two input ports **a** and **b** and output port **c**.

```
entity and2 is
port(
    a       : in  std_logic;
    b       : in  std_logic;
    c       : out std_logic);
end and2;
```



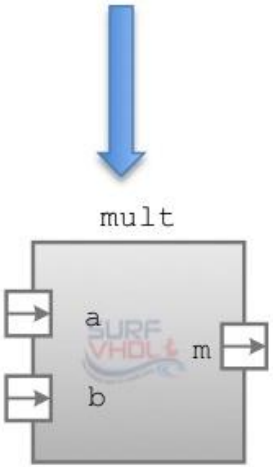
VHDL Entity representing a flip-flop type D with input port: **clock** and **reset** active low, data **D**, and output port **Q**.

```
entity flip_flop is
port(
    clk     : in  std_logic;
    rstb    : in  std_logic;
    d       : in  std_logic;
    q       : out std_logic);
end flip_flop;
```



VHDL Entity representing a multiplier with input operand **a** and **b** of 8 bit and output **m** of 16 bit.

```
entity mult is
port(
    a       : in  std_logic_vector( 7 downto 0);
    b       : in  std_logic_vector( 7 downto 0);
    m       : out std_logic_vector(15 downto 0));
end mult;
```



iii) Architecture Section:

- This is the final section .
- This section specifies the behavior and internal logic of system under design.
- The basic format is:

```
architecture  <architecture_name>  of <entity_name> is
begin
        ..................
        //statements
        ..................
End <architecture_name> ;
```

iii) Architecture Section:


```
entity and2 is
port(
    a       : in  std_logic;
    b       : in  std_logic;
    c       : out std_logic);
end and2;

architecture and2_a of and2 is
begin
    c <= a and b;
end and2_a;
```

Architecture name

Different types of Modeling to describe architecture:

i. Behavioral Modeling
ii. Structural Modeling
iii. Data flow Modeling
iv. Mixed Modeling

i) Behavioral Modeling

- In Behavioral modeling, we describe a system in terms of what it does or how it behaves rather than in terms of its component and interconnection between them.
- Such description specifies the relationship between input and output signal.

a) OR gate:

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  ------------------------------
5.
6.  entity OR_ent is
7.  port( x: in std_logic;
8.        y: in std_logic;
9.        F: out std_logic
10.       );
11.       end OR_ent;
12.
13.       ------------------------
14.
15.       architecture OR_arch of OR_ent is
16.       begin
17.
18.           process(x, y)
19.           begin
20.               -- compare to truth table
21.               if ((x='0') and (y='0')) then
22.                   F <= '0';
23.               else
24.                   F <= '1';
25.               end if;
26.           end process;
27.
28.       end OR_arch;
29.
30.       ---------------------------------------
31.
32.       architecture OR_beh of OR_ent is
33.       begin
34.
35.           F <= x or y;
36.
37.       end OR_beh;
```

b) AND gate

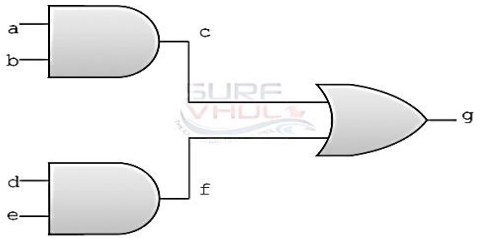
```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  ------------------------------
5.
6.  entity AND_ent is
7.  port
8.  ( x: in std_logic;
9.    y: in std_logic;
10.   F: out std_logic
11.   );
12.   end AND_ent;
13.
14.   ------------------------------
15.
16.  architecture behav1 of AND_ent is
17.  begin
18.
19.      process(x, y)
20.      begin
21.          -- compare to truth table
22.          if ((x='1') and (y='1')) then
23.              F <= '1';
24.          else
25.              F <= '0';
26.          end if;
27.      end process;
28.
29.  end behav1;
30.
31.  ------------------------------------
32.
33.  architecture behav2 of AND_ent is
34.  begin
35.
36.      F <= x and y;
37.
38.  end behav2;
```

c) XOR gate

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3.
4. ----------------------------
5.
6. entity XOR_ent is
7. port(
8.      x: in std_logic;
9.      y: in std_logic;
10.     F: out std_logic
11. );
12. end XOR_ent;
13.
14. ----------------------------
```

```
15.
16. architecture behv1 of XOR_ent is
17. begin
18.
19.     process(x, y)
20.     begin
21.         -- compare to truth table
22.         if (x/=y) then
23.             F <= '1';
24.         else
25.             F <= '0';
26.         end if;
27.     end process;
28.
29. end behv1;
30.
31. ----------------------------------------
32.
33. architecture behv2 of XOR_ent is
34. begin
35.
36.     F <= x xor y;
37.
38. end behv2;
39.
40. ----------------------------------------
```

d) VHDL concurrency



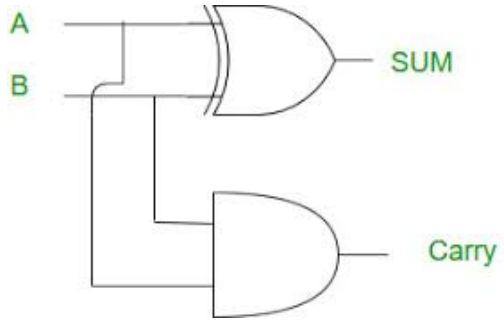

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3. entity and_or is
4. port(
5.          a : in std_logic;
6.          b : in std_logic;
7.          d  : in std_logic;
8.          e : in std_logic;
9.          g : out std_logic);
10.end and_or;
11.
12.architecture and_or_a of and_or is
13.         signal c : std_logic;
14.         signal f : std_logic;
15.begin
```

```
16.          c <= a and b; -- c assigment
17.          f <= d and e; -- f assigment
18.        g <= c or f; -- g assigment
19.end and_or_a;
```

It is clear that the same result will be evaluated from this implementation

```
1. architecture and_or_a of and_or is
2. signal c : std_logic;
3. signal f : std_logic;
4. begin
5. g <= c or f; -- g assigment
6. f <= d and e; -- f assigment
7. c <= a and b; -- c assigment
8. end and_or_a;
```

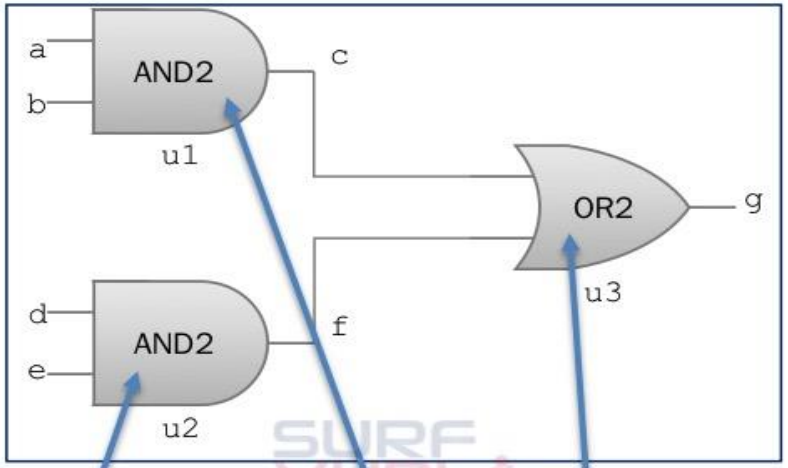
e) VHDL Code for half-adder using dataflow via logic equation




```
library IEEE;
use IEEE.std_logic_1164.all;

entity H_adder is
port(
     a,b : IN std_logic;
     sum,carry : OUT std logic);
end H_adder;

architecture dataflow of H_adder is
begin
     sum <= a xor b;
     carry <= a and b;

end dataflow;
```

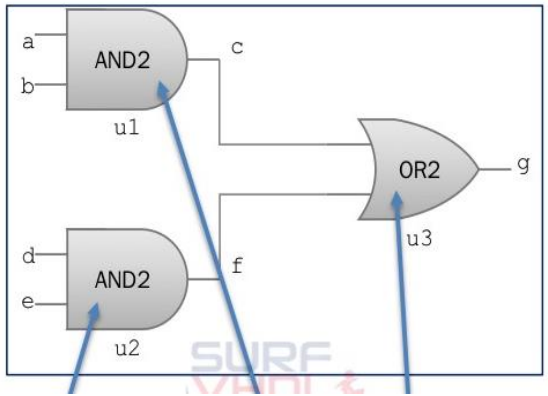
ii) Structural Modeling

- In structural modeling, different interconnection of different components are described.
- If you are designing a complex project, you should split in **two or more simple design** in order to easy handle the complexity.




```
entity and2 is
port(
    a       : in  std_logic;
    b       : in  std_logic;
    c       : out std_logic);
end and2;
```

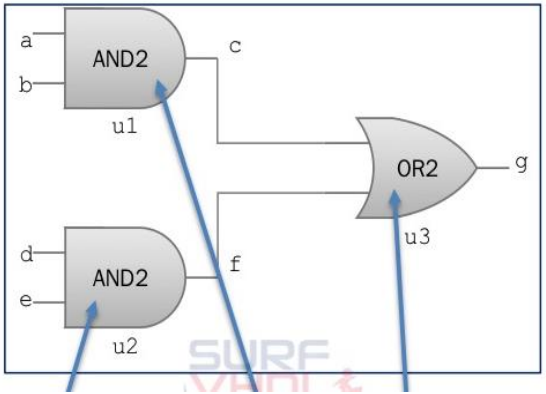
```
entity or2 is
port(
    a       : in  std_logic;
    b       : in  std_logic;
    c       : out std_logic);
end or2;
```




```
library IEEE;
use IEEE.std_logic_1164.all;
entity ckt is
port
        (a, b, d, e : in std_logic;
        g: out std_logic);
end ckt;
```

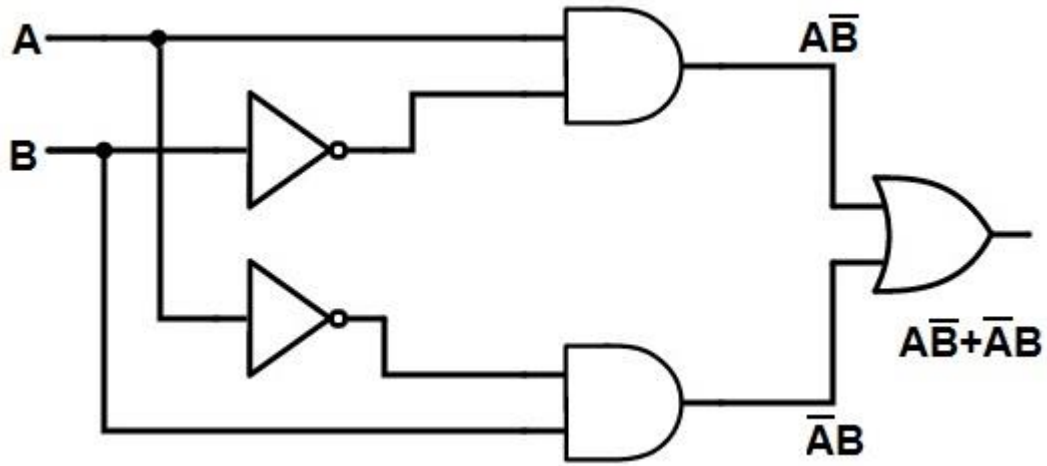
```
architecture structure of ckt
is
component AND2
        port
        (x, y: in std_logic;
        z: out std_logic);
        )
end component;

component OR2
        port
        (p, q: in std_logic;
        r: out std_logic);
        )
end component;

signal  c, f : std_logic;
```

```
begin
 u1: AND2 portmap (x=>a, y=> b, z=> c);
 u2: AND2 portmap (x=>d ,y=>e, z=> f);
 u3: OR2 portmap (p=>c, q=>f, z<= g);
end structure
```




```
entity and_or is
port(
   a : in std_logic;
   b : in std_logic;
   d : in std_logic;
   e : in std_logic;
   g : out std_logic);
end and_or;

architecture and_or_a of and_or is

    component and2 -- and2 component declaration
    port(
        a : in std_logic;
        b : in std_logic;
        c : out std_logic);
    end component;
```

```
    component or2 -- or2 component declaration
    port(
        a : in std_logic;
        b : in std_logic;
        c : out std_logic);
    end component;

    signal c : std_logic; -- wire used to connect
    signal f : std_logic; -- the component
begin
    -- and2 component instance
    u1: and2
    port map(
        a => a,
        b => b,
        c => c);

    -- and2 component instance
    u2: and2
    port map(
        a => d,
        b => e,
        c => f);

    -- or2 component instance
    u3: or2
    port map(
        a => c,
        b => f,
        c => g);
end and_or_a;
```


A
B
$A\overline{B}$
$\overline{A}B$
$A\overline{B}+\overline{A}B$

VHDL design with high level of abstraction

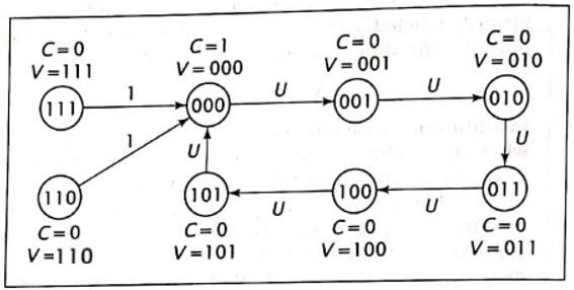
- A high level of abstraction is a finite state machine description of a system (using MUX, decoder, ROM)
- A low level of abstraction is a digital logical implementation (using gates)
- The design of modulo 6 counter as a finite state machine is high level of abstraction.

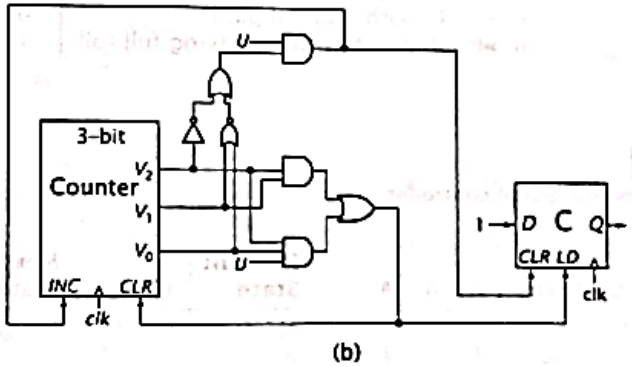
VHDL design with high level of abstraction

- For Modulo 6 counter

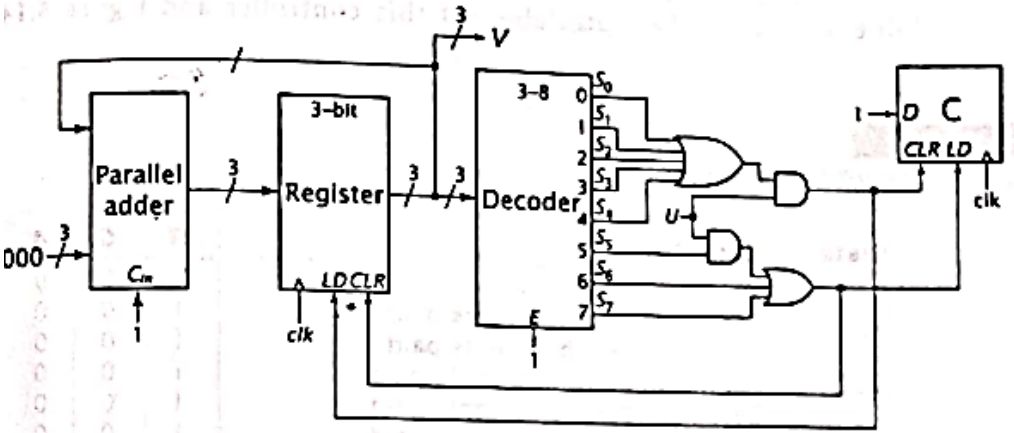
$(S_0+S_1+S_2+S_3+S_4)U$: $V \leftarrow V+1$, $C \leftarrow 0$

$S_5U+S_6+S_7$: $V \leftarrow 0$, $C \leftarrow 1$





(b)



(a)

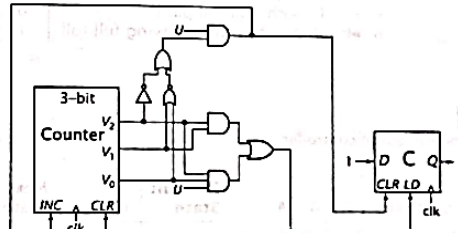
Present State		U	Next State	$V_2V_1V_0$	C
S_0	000	0	S_0	000	1
S_0	000	1	S_1	001	0
S_1	001	0	S_1	001	0
S_1	001	1	S_2	010	0
S_2	010	0	S_2	010	0
S_2	010	1	S_3	011	0
S_3	011	0	S_3	011	0
S_3	011	1	S_4	100	0
S_4	100	0	S_4	100	0
S_4	100	1	S_5	101	0
S_5	101	0	S_5	101	0
S_5	101	1	S_0	000	1
S_6		X	S_0	000	1
S_7		X	S_0	000	1

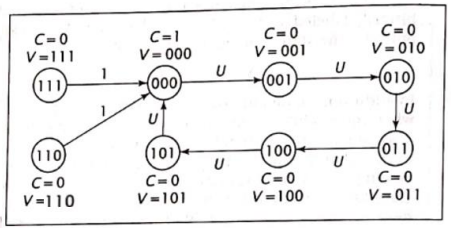
VHDL file to implement Modulo 6 counter using a high level of abstraction

$(S_0+S_1+S_2+S_3+S_4)U$: $V \leftarrow V+1$, $C \leftarrow 0$

$S_5U+S_6+S_7$: $V \leftarrow 0$, $C \leftarrow 1$

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mod6 is
      port(
            U,clk: in std_logic;
                C: out std_logic;
                V: out std_logic_vector(2 downto 0)
      );
end mod6;

architecture amod6 of mod6 is
      type states is (S0, S1, S2, S3, S4, S5, S6, S7);
      signal present_state, next_state: states;
begin
      state_mod6: process(present_state,u)
            begin
                  case present_state is
                        when S0 => V<="000"; C<='1';
                              if (U='1') then next_state <= S1;
                                    else next_state <= S0;
                              end if;
                        when S1 => V<="001"; C<='0';
                              if (U='1') then next_state <= S2;
                                    else next_state <= S1;
                              end if;
                        when S2 => V<="010"; C<='0';
                              if (U='1') then next_state <= S3;
                                    else next_state <= S2;
                              end if;
```






```
                        when S3 => V<="011"; C<='0';
                              if (U='1') then next_state <= S4;
                                    else next_state <= S3;
                              end if;
                        when S4 => V<="100"; C<='0';
                              if (U='1') then next_state <= S5;
                                    else next_state <= S4;
                              end if;
                        when S5 => V<="101"; C<='0';
                              if (U='1') then next_state <= S0;
                                    else next_state <= S5;
                              end if;
                        when S6 => V<="110"; C<='0';
                              next_state <= S0;
                        when S7 => V<="111"; C<='0';
                              next_state <= S0;
                  end case;
            end process state_mod6;
      state_transition: process(clk)
            begin
              if rising_edge(clk) then present_state <= next_state;
              end if;
            end process state_transition;
end amod6;
```

Explanation:

- To declare the VHDL design for this counter, we begin with the library declarations and entity section.
- In the library declarations, we will only need the standard IEEE library and its *std_logic 1164* module.
- In the entity section, we declare the system's inputs and outputs.
- The modulo 6 counter has two inputs. U and the system clock which we label clk.
- It has 1-bit output C and 3-bit output V.
- We define V as a 3-bit vector with indices 2, 1 and 0.
- The library declarations and entity section of this design are:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mod6 is
      port(
            U,clk: in std_logic;
                C: out std_logic;
                V: out std_logic_vector(2 downto 0)

      );
end mod6;
```

- Now we must develop the architecture section of the modulo 6 counter.
- But before specifying the system's behavior, the architecture declares a new enumerated type, states.
- This type has eight possible values, S_0 to S_7, which correspond to the six valid and two invalid sates of the counter.
- It declares two signals: *present_state* and *next_state*, to be of this type.

```
architecture amod6 of mod6 is
      type states is (S0, S1, S2, S3, S4, S5, S6, S7);
      signal present_state, next_state: states;
```

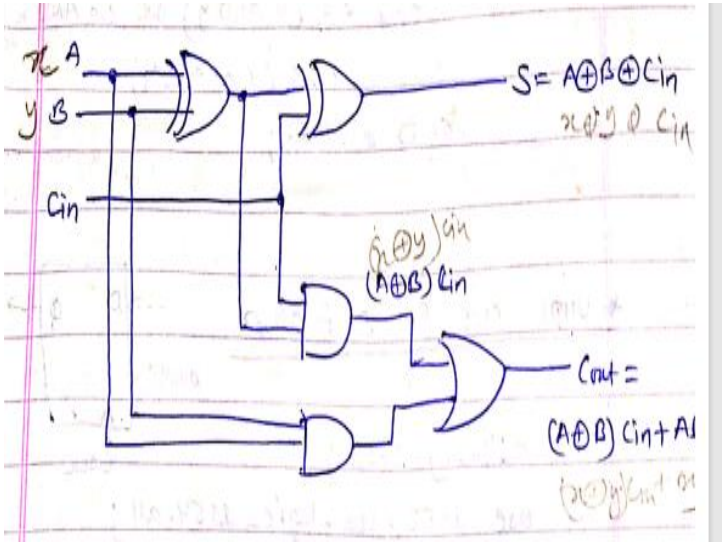
- Beyond these declarations, we partition this design into two processes.
- The first looks at the present state of the counter and its input values, and generates the desired outputs and next state value.
- The second process makes the actual transition from the present state to the next state.
- The first procedure is names state_mod6. it uses a case statement.

```
begin
      state_mod6: process(present_state,u)
            begin
                  case present_state is
                        when S0 => V<="000"; C<='1';
                              if (U='1') then next_state <= S1;
                                    else next_state <= S0;
                              end if;
                        when S1 => V<="001"; C<='0';
                              if (U='1') then next_state <= S2;
                                    else next_state <= S1;
                              end if;
                        when S2 => V<="010"; C<='0';
                              if (U='1') then next_state <= S3;
                                    else next_state <= S2;
                              end if;
                        when S3 => V<="011"; C<='0';
                              if (U='1') then next_state <= S4;
                                    else next_state <= S3;
                              end if;
                        when S4 => V<="100"; C<='0';
                              if (U='1') then next_state <= S5;
                                    else next_state <= S4;
                              end if;
                        when S5 => V<="101"; C<='0';
                              if (U='1') then next_state <= S0;
                                    else next_state <= S5;
                              end if;
                        when S6 => V<="110"; C<='0';
                              next_state <= S0;
                        when S7 => V<="111"; C<='0';
                              next_state <= S0;
                  end case;
            end process state_mod6;
      state_transition: process(clk)
            begin
              if rising_edge(clk) then present_state <= next_state;
              end if;
          end process state_transition;
end amod6;
```

- The second process is **state_transition.**
- On the rising edge of the clock it copies the ***next_state*** value generated bus process state_mod6 to present_state.
- Although *state_mod6* generates the next state value, the transition to this state occurs in process *state_transition.*
- Once the new value of *present_state* is set, it is used by state_mod6 to select the next value for the case statement.
- The code for this module is:

```
state_transition: process(clk)
      begin
            if rising_edge(clk) then present_state <= next_state;
            end if;
            end process state_transition;
```

VHDL code for Full adder (Low level abstraction)




```
Library IEEE;
use IEEE.std_logic_1164.all;

Entity fulladd is
port (
    Cin, x, y : in std_logic;
    s, Cout : out std_logic
);
End fulladd;
```

```
architecture afulladd of fulladd is
begin
    s <= x xor y xor Cin;
    Cout <= (x AND y) OR (Cin AND x)
            OR (Cin AND y);
END afulladd;
```

Assignment:

1. Write a VHDL code for generating the combinational circuit with three inputs A, B and C and output F=(A^B) v A' v (A' ^B'). [PU 2017 Fall]
2. Write a VHD code to generate a function F= ABC + A'B'C'. [PU 2018 Spring]
3. Write a VHDL code for generating the combinational circuit for a function F(x, y, z) $= \sum(1,3,4,6)$. [PU 2017 Spring]
4. Write a VHDL code for generating the combinational circuit with three inputs A, B, C and an output F, where F(A, B, C) $= \sum(1,3,4,6)$. [PU 2018 Fall]
5. Write VHDL code for designing a full adder using half adder in structural model. [PU 2020 Spring]

Assignment:

1. Write a VHDL code for generating the combinational circuit with three inputs A, B and C and output F=(A^B) v A' v (A' ^B'). [PU 2017 Fall]
2. Write a VHDL code for generating the combinational circuit for a function F(x, y, z) = $\sum(1,3,4,6)$. [PU 2017 Spring]
3. Write a VHDL code for generating the combinational circuit with three inputs A, B, C and an output F, where F(A, B, C) = $\sum(1,3,4,6)$. [PU 2018 Fall]
4. Write a VHD code to generate a function F= ABC + A'B'C'. [PU 2018 Spring]
5. Write VHDL code for designing a full adder using half adder in structural model. [PU 2020 Spring]

Partial End
of
chapter 3

Some References for VHDL code

Some more about

VHDL

Assistant Professor

Er. Shiva Ram Dam

Contents:

1. Background and basic concepts
2. Structural specification of hardware and design organization
3. VHDL realization of basic digital circuits
   - Binary adder
   - Multiplier
   - Decoder
   - Multiplexer
   - Counters
   - Shift Registers
   - Sequence Detectors

8.1 VHDL

- Very high speed integrated circuit (VHSIC) Hardware Description Language (VHDL)
- VHDL was developed to provide a standard for designing digital systems.
- VHDL specifies a formal syntax. The designer creates a design file using that syntax just as a programmer writes a C-program.
- The designer then synthesizes the design using only design package that can accept VHDL files. This is equivalent to the programmer compiling the C-program. It checks for errors in syntax and declaration, but not in logic.
- Finally, the designer debugs the design using simulation tools

Advantages of VHDL:

1. **PORTABILITY:** Just as a valid C-program can be compiled by any compliant C compiler, a VHDL design can be synthesized by any design system that supports VHDL.
2. **DEVICE INDEPENDENT:** The VHDL file is device independent. The same file can be used to implement the design on a custom ICs, on ASICs or any PLD that is capable of containing the design.
3. **SIMULATION:** VHDL designs can be simulated by the design system, allowing the designer to verify the design performance before committing it to hardware.
4. **SYSTEM SPECIFICATION:** The designer can design the system using a high level of abstraction, such as a finite state machine, down to a low level digital logic implementation.

Advantages of VHDL:

Summary:

1. VHDL is used to design ICs and application specific ICs (ASICs), PLD.
2. VHDL offers portability. VHDL can be synthesized by any design system supported by VHDL.
3. VHDL files are device independent.
4. VHDL designs can be simulated.
5. VHDL files provide good documentation of system design.

Disadvantages of VHDL:

- VHDL source code often becomes long and difficult to follow, especially at a low level of abstraction.
- It is often less intuitive than a block diagram or RTL description of same system.
- Different design tools may produce different valid design for the same system, especially for high level of abstraction, providing no details about the implementation

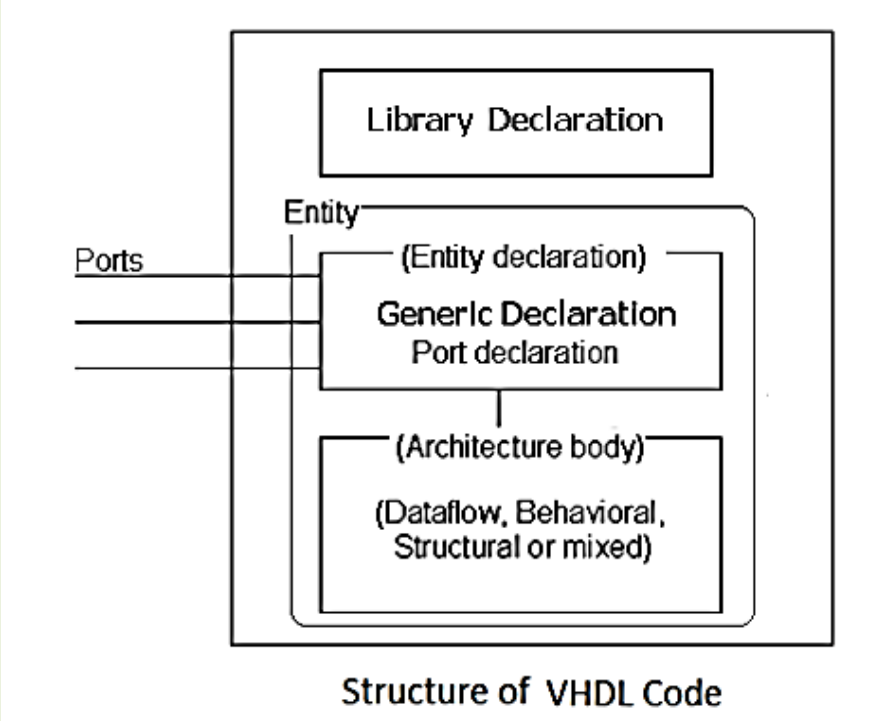
Disadvantages of VHDL:

Summary:

1. VHDL source code often are long and difficult to follow at low level of abstraction.
2. VHDL are difficult to understand.
3. VHDL does not provide detail about its design implementation.

8.2 VHDL Structure:

- VHDL design code has 3 primary sections:
  - Library declarations
  - Entity section and
  - Architecture section



Structure of VHDL Code

i) Library declaration:

- It consists of statements that specify libraries to be accessed and modules of these libraries to be used.(such as “include” in C)
- The most commonly used library is called IEEE.
- In this library, std_logic_1164 is used most often, which specifies i/p and o/p declarations for the designer to use (such as “stdio.h” in C).

General Syntax is:

```
library   <library_name>;              //Library IEEE will be used almost in every code
use <library_name> . <package_name> . all;   //This is format for importing package
```

Example:

```
Library IEEE;
use IEEE.std_logic_1164.all;
```

ii) Entity section:

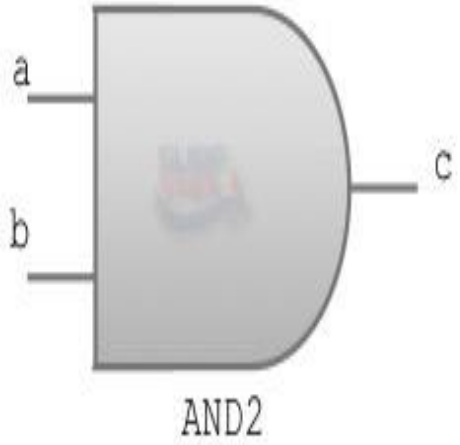
- In the entity section, the designer specifies the name of the design and its inputs and outputs.
- The designer does not specify the logic that uses and drives these signals here; that is done in the architecture section.

General format is:

```
entity <entity_name> is
port
(
    <signal_name> : mode <type>;
    <signal_name> : mode <type> : = default_value;
    <signal-name> : inout <type>;
    <signal-name> : out <type>: = default_value;
);
End <entity_name> ;
```

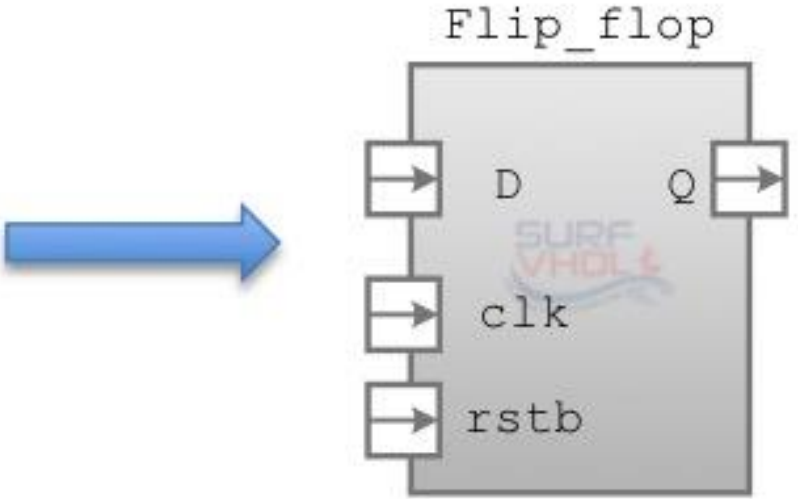
VHDL Entity representing an "**And gate**" with two input ports **a** and **b** and output port **c**.

```
entity and2 is
port(
    a       : in  std_logic;
    b       : in  std_logic;
    c       : out std_logic);
end and2;
```



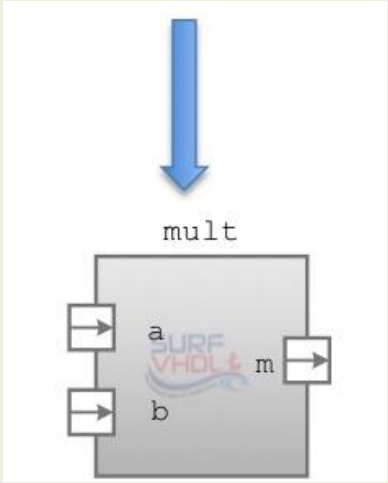
VHDL Entity representing a flip-flop type D with input port: **clock** and **reset** active low, data **D**, and output port **Q**.

```
entity flip_flop is
port(
    clk     : in  std_logic;
    rstb    : in  std_logic;
    d       : in  std_logic;
    q       : out std_logic);
end flip_flop;
```



VHDL Entity representing a multiplier with input operand **a** and **b** of 8 bit and output **m** of 16 bit.

```
entity mult is
port(
    a       : in  std_logic_vector( 7 downto 0);
    b       : in  std_logic_vector( 7 downto 0);
    m       : out std_logic_vector(15 downto 0));
end mult;
```

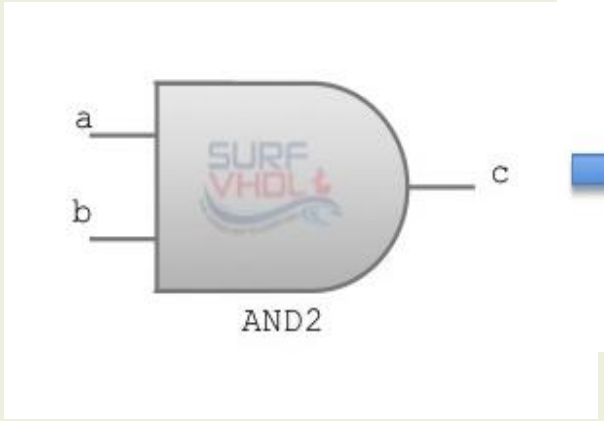


iii) Architecture Section:

- This is the final section .
- This section specifies the behavior and internal logic of system under design.
- The basic format is:

```
architecture  <architecture_name>  of <entity_name> is
begin
        ..................
        //statements
        ..................
End <architecture_name> ;
```

iii) Architecture Section:




```
entity and2 is
port(
    a       : in  std_logic;
    b       : in  std_logic;
    c       : out std_logic);
end and2;

architecture and2_a of and2 is
begin
    c <= a and b;
end and2_a;
```

Architecture name

Different types of Modeling to describe architecture:

i. **Data flow Modeling**

Dataflow describes how the circuit signals flow from the inputs to the outputs.

ii. **Behavioral Modeling**

This model describes a system in terms of what it does or how it behaves rather than in terms of its component and interconnection between them.

iii. **Structural Modeling**

In structural modeling, different interconnection of different components are described.

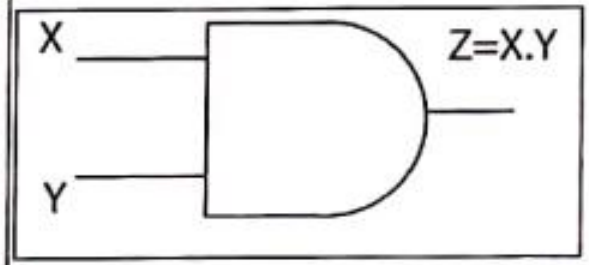
i) Dataflow Modeling

- Dataflow describes how the circuit signals flow from the inputs to the outputs.
- A dataflow model specifies the functionality of the entity without explicitly specifying its structure.
- The functionality shows the flow of information through the entity, which is expressed primarily using concurrent signal assignment statements and block statements.

1. Write VHDL code for AND gate (using dataflow model)

```
Library ieee;
use ieee.std_logic_1164.all;
Entity andgate is
Port(
        x, y: in std_logic;
        z : out std_logic;
        );
End andgate;
architecture arc_andgate of andgate is
begin
z <= x and y;
end arc_andgate;
```

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

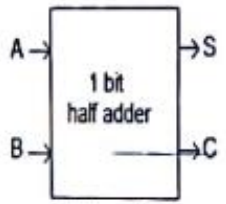


2. Write VHDL code for Half Adder (using Dataflow model)

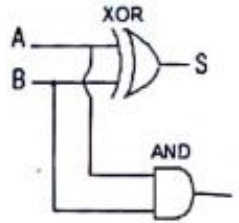
```
Library ieee;
use ieee.std_logic_1164.all;
Entity half_adder is
Port(
        a, b: in std_logic;
        c, s : out std_logic;
        );
End half_adder;
architecture arc_half_adder of half_adder  is
begin
c<= x and y;
s<= x xor y;
end arc_half_adder;
```

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table

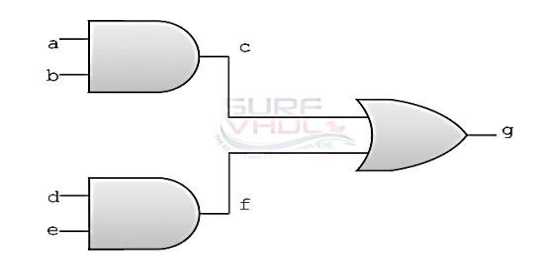


Schematic



Realization

3. Write VHDL concurrency coding for below circuit (using dataflow model)




```
1. library ieee;
2. use ieee.std_logic_1164.all;
3. entity and_or is
4. port(
5.          a : in std_logic;
6.          b : in std_logic;
7.          d  : in std_logic;
8.          e : in std_logic;
9.          g : out std_logic);
10.end and_or;
11.
12.architecture and_or_a of and_or is
13.         signal c : std_logic;
14.         signal f : std_logic;
15.begin
```

```
16.          c <= a and b; -- c assigment
17.          f <= d and e; -- f assigment
18.         g <= c or f; -- g assigment
19.end and_or_a;
```

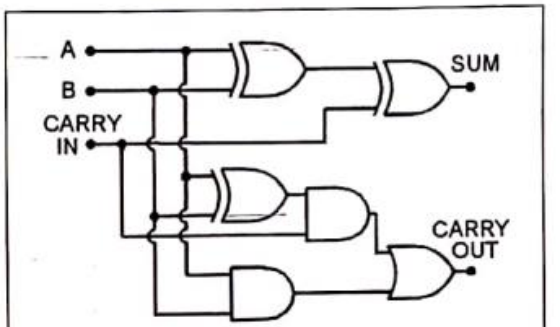
It is clear that the same result will be evaluated from this implementation

```
1. architecture and_or_a of and_or is
2. signal c : std_logic;
3. signal f : std_logic;
4. begin
5. g <= c or f; -- g assigment
6. f <= d and e; -- f assigment
7. c <= a and b; -- c assigment
8. end and_or_a;
```

4. Write VHDL code for Full adder (using dataflow model)

 # Write the VHDL code for full adder circuit.

```
Library ieee;
use ieee.std_logic_1164.all;
Entity full_adder is
Port(
        a, b, c: in std_logic;
        carry, sum : out std_logic;
        );
End full_adder;
architecture arc_full_adder of full_adder  is
signal x, y, z : std_logic;
begin
x<= a xor b;
sum<= x xor c;
y<= x and c;
z<= a and b;
carry <= y or z;
end arc_full_adder;
```



A	B	CARRY IN	SUM	CARRY OUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

ii) Behavioral Modeling

- In Behavioral modeling, we describe a system in terms of what it does or how it behaves rather than in terms of its component and interconnection between them.
- Such description specifies the relationship between input and output signal.
- This style specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order.

5. Write VHDL for OR gate (using behavioral model)

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  ------------------------------
5.
6.  entity OR_ent is
7.  port( x: in std_logic;
8.        y: in std_logic;
9.        F: out std_logic
10.       );
11.       end OR_ent;
12.
13.       ------------------------
```

```
14.
15.       architecture OR_arch of OR_ent is
16.       begin
17.
18.           process(x, y)
19.           begin
20.               -- compare to truth table
21.               if ((x='0') and (y='0')) then
22.                   F <= '0';
23.               else
24.                   F <= '1';
25.               end if;
26.           end process;
27.
28.       end OR_arch;
```

6. Write VHDL for AND gate (using behavioral model)

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  ------------------------------
5.
6.  entity AND_ent is
7.  port
8.  ( x: in std_logic;
9.    y: in std_logic;
10.   F: out std_logic
11.   );
12.   end AND_ent;
13.
14.   ------------------------------
15.
```

```
16.  architecture behav1 of AND_ent is
17.  begin
18.
19.      process(x, y)
20.      begin
21.          -- compare to truth table
22.          if ((x='1') and (y='1')) then
23.              F <= '1';
24.          else
25.              F <= '0';
26.          end if;
27.      end process;
28.
29.  end behav1;
30.
```

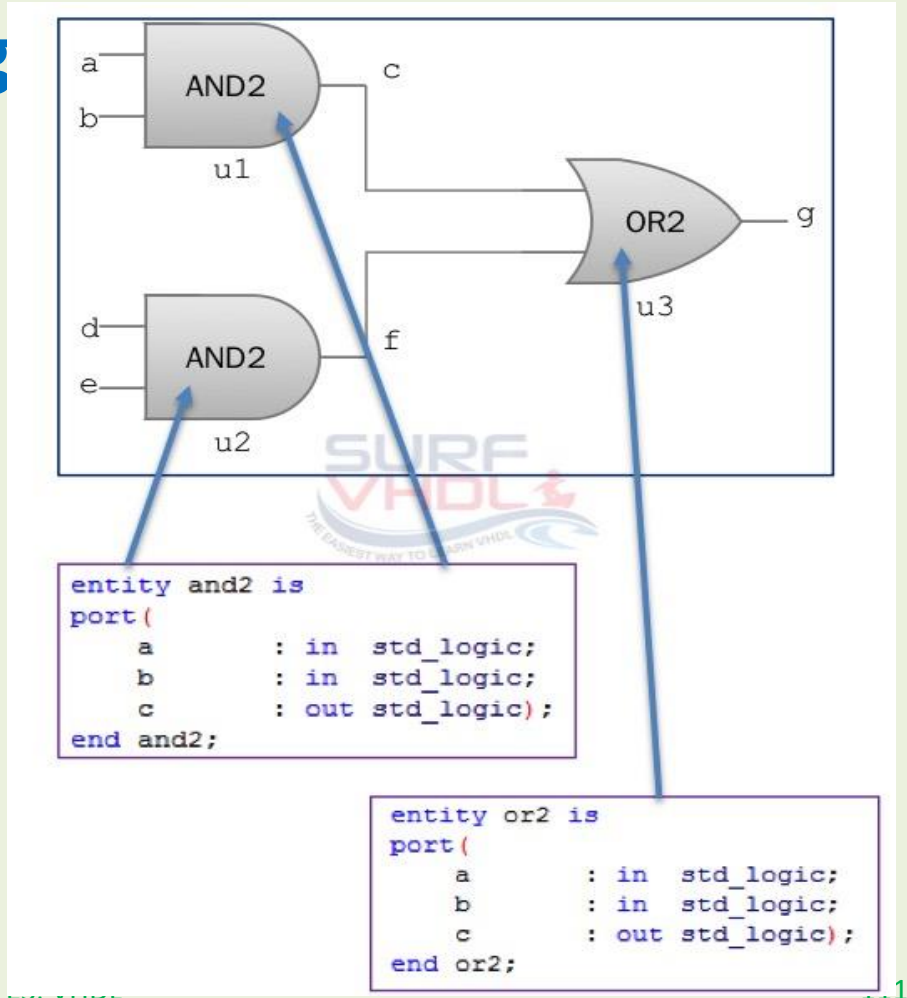
7. Write VHDL for OR gate (using behavioral model)

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3.
4. ----------------------------
5.
6. entity XOR_ent is
7. port(
8.      x: in std_logic;
9.      y: in std_logic;
10.     F: out std_logic
11.  );
12.  end XOR_ent;
13.
14.  ----------------------------
```

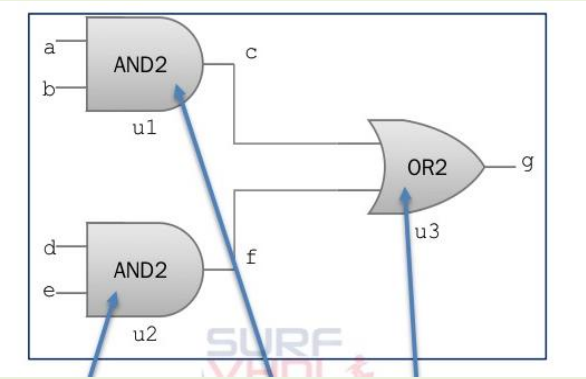
```
15.
16.  architecture behv1 of XOR_ent is
17.  begin
18.
19.      process(x, y)
20.      begin
21.          -- compare to truth table
22.          if (x/=y) then
23.              F <= '1';
24.          else
25.              F <= '0';
26.          end if;
27.      end process;
28.
29.  end behv1;
```

ii) Structural Modeling

- In structural modeling, different interconnection of different components are described.
- If you are designing a complex project, you should split in **two or more simple design** in order to easy handle the complexity.



8. Write VHDL for below combinational circuit (using structural model but no behavioral/dataflow specified)




```
library IEEE;
use IEEE.std_logic_1164.all;
entity ANDgate is
port
        (a, b, d, e : in std_logic;
        g: out std_logic);
end ckt;
```

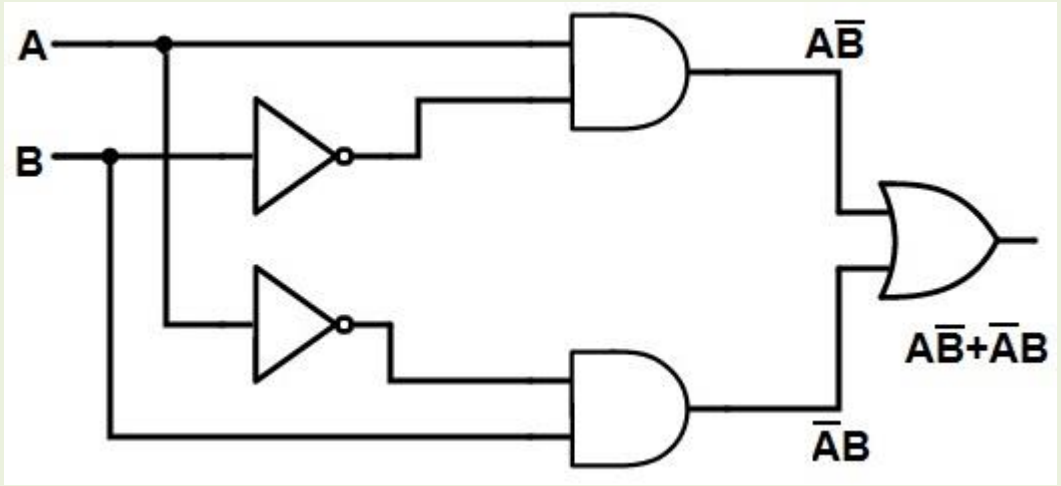
```
architecture structure of ckt is
component AND2
        port
        (x, y: in std_logic;
        z: out std_logic);
end component;
```

```
component OR2
        port
        (p, q: in std_logic;
        r: out std_logic);
end component;
```

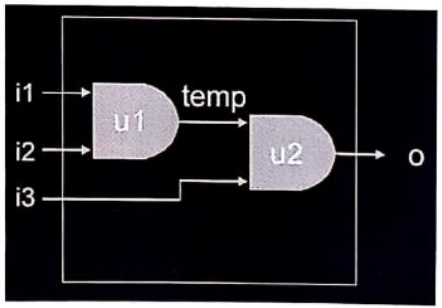
```
signal  c, f : std_logic;

begin
 u1: AND2 port map (a, b, c);
 u2: AND2 port map (d ,e, f);
 u3: OR2 port map (c, f, g);
end structure
```

Assignment:

9. Write VHDL code for below circuit using structural modeling.



10. Write VHDL code for below (using just structural model)




```
Library ieee;
use ieee.std_logic_1164.all;
entity and3gate is
port
(       o : out std_logic;
        i1 : in std_logic;
        i2 : in std_logic;
        i3 : in std_logic
);
end and3gate;
```

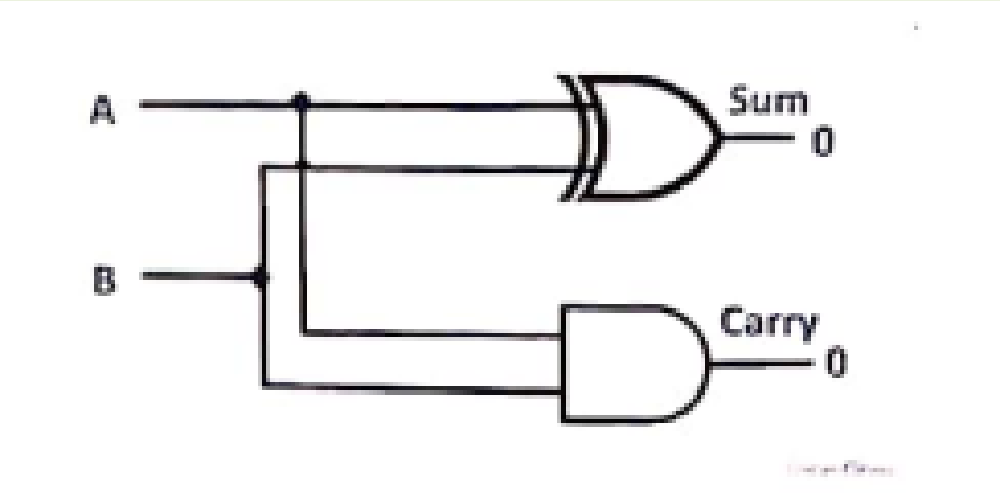
```
architecture arc_and3gate of and3gate is
 componcnt andgate is
port
(       c : out std_logic;
        a : in std_logic;
        b : in std_logic
);
end component;
signal temp1 : std_logic;
begin
                u1:     andgate
                        port map(temp1, i1, i2);
                u2:     andgate
                        port map(o, temp1, i3);
end arc_and3gate;
```

8.3 VHDL realization of basic digital circuits

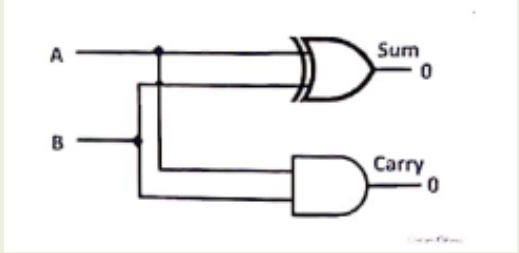
A. Binary adder
B. Multiplier
C. Decoder
D. Multiplexer
E. Counters
F. Shift Registers
G. Sequence Detectors

8.3.1 Binary adders

11. Write a VHDL code to implement the half adder using structural modeling or component.



Solution:



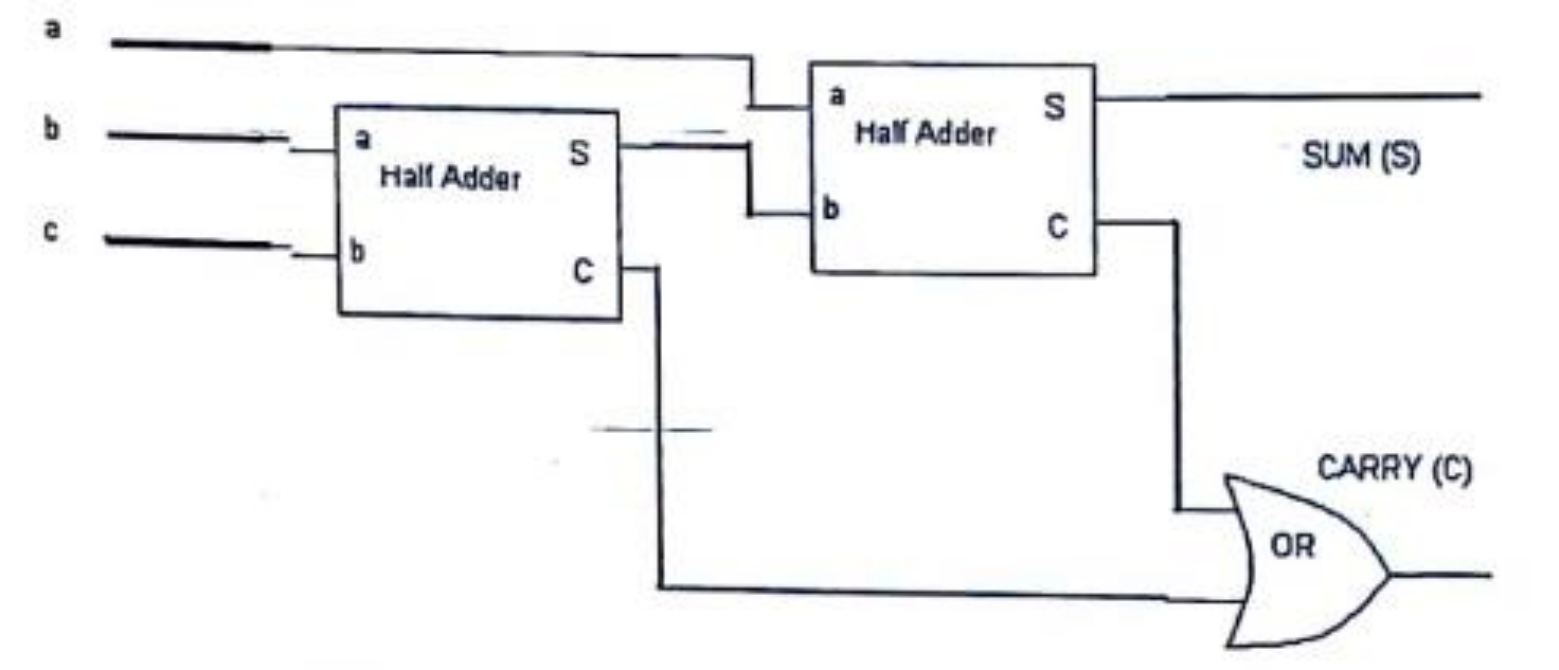

```
Library ieee;
use ieee.std_logic_1164.all;
entity andgate is
port
(       c : out std_logic;
        a : in std_logic;
        b: in std_logic;
 );
end andgate;
```

```
architecture arch_halfadder of halfadder is
component andgate
port
(
        a, b : in std_logic;
        c: out std_logic;
);
End component;
```

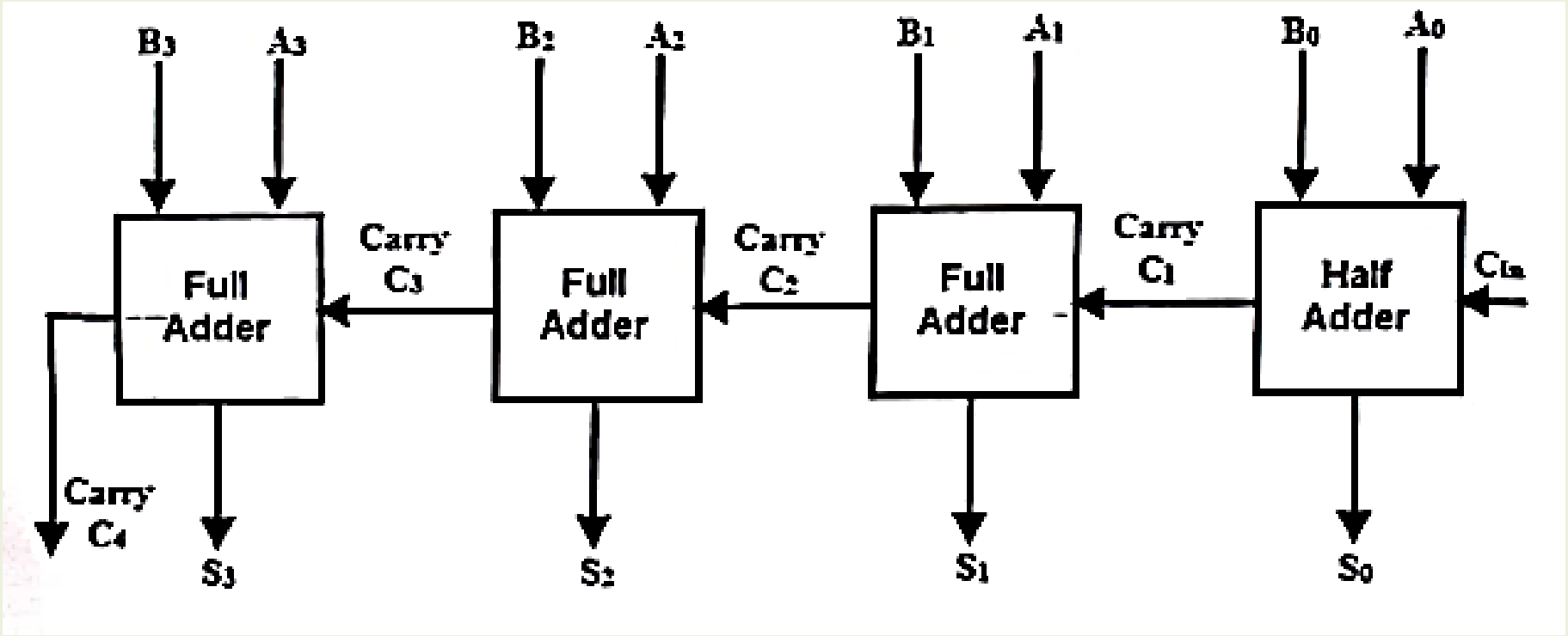
```
Component xorgate
(       --
        a, b : in std_logic;
        c : out std_logic;
);
End component;
```

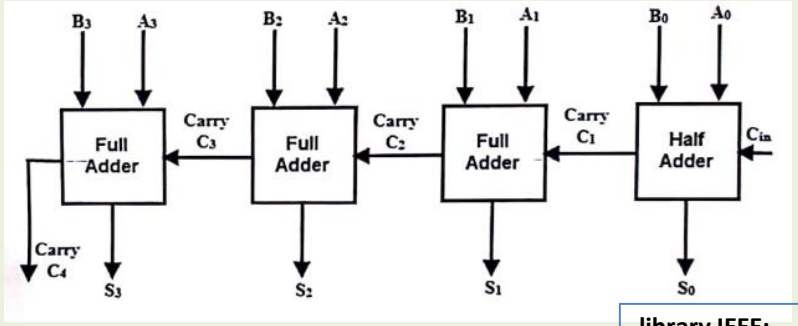
```
Begin
        U0: andgate portmap(carry, a, b);
        U1: xorgate portmap  (sum, a, b);
End arch_halfadder;
```

12. Write a VHDL code to implement the full-adder using two half adders.



13. Write VHDL code for 4-bit binary parallel adder.






```
library IEEE;
use IEEE.STD_LOGIC_1164.all;


----------------------------------------
--VHDL code for Full Adder
----------------------------------------
entity FA is
   port (  a : in STD_LOGIC;
           b : in STD_LOGIC;
           c : in STD_LOGIC;
           sum : out STD_LOGIC;
           carry : out STD_LOGIC
           );
end FA;


architecture FA_arc of FA is
begin
   sum <= a xor b xor c;
   carry <= (a and b) or (b and c) or (c and a);

end FA_arc;
```

```
----------------------------------------
--VHDL code for Parallel  Adder
----------------------------------------

entity PA is
    port(
           a : in STD_LOGIC_VECTOR(3 downto 0);
           b : in STD_LOGIC_VECTOR(3 downto 0);
           carry : out STD_LOGIC;
           sum : out STD_LOGIC_VECTOR(3 downto 0)
           );
end PA;

architecture PA_arch of PA is

Component FA is
   port (  a, b, c  : in STD_LOGIC;
           sum , carry : out STD_LOGIC
           );
end component;

signal c : std_logic_vector (3 downto 1);

begin

   u0 : FA port map (a(0) , b(0) ,  '0'  , sum(0) , c1));
   u1 : FA port map (a(1) , b(1) , s(0) , sum(1) , c2);
   u2 : FA port map (a(2) , b(2) , s(1) , sum(2) , c3);
   u3 : FA port map (a(3) , b(3) , s(2) , sum(3) , carry);

end PA_arch ;
```

8.3.2 Multiplier

14. VHDL for 2 bit binary multiplier using dataflow modeling

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

entity multiply is
    port (A, B : in bit_vector(1 downto 0);
    P : out bit_vector(3 downto 0)
);
end multiply;

architecture dataflow of multiply is
    begin
        P(0) <= A(0) AND B(0);
        P(1) <= (A(1) AND B(0)) XOR (A(0) AND B(1));
        P(2) <= ((A(1) AND B(0)) AND (A(0) AND B(1))) XOR (A(1) AND B(1));
        P(3) <= ((A(1) AND B(0)) AND (A(0) AND B(1))) AND (A(1) AND B(1));
end architecture;
```

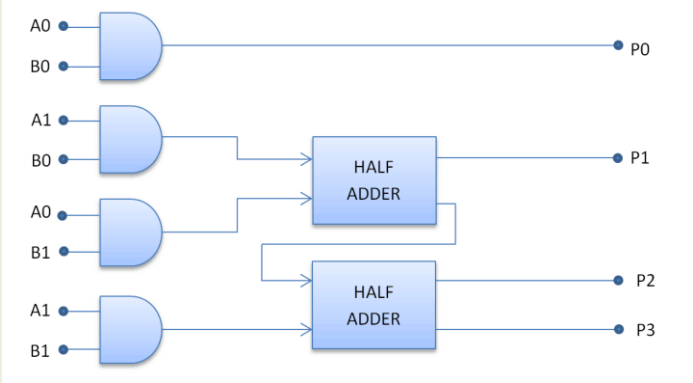
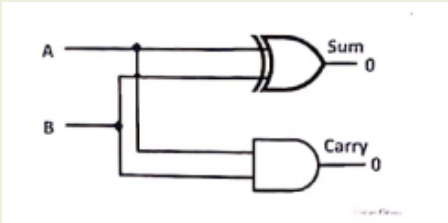


Fig: The logic circuit of a 2-bit binary multiplier



15. VHDL for 2 bit binary multiplier using behavioral modeling

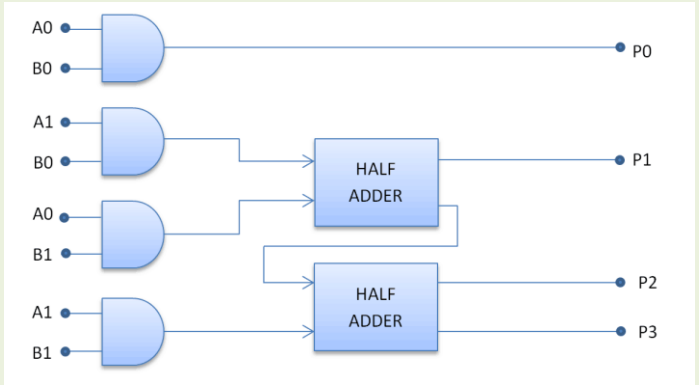
```
library ieee;
use ieee.std_logic_1164.all;

entity multiply_behav is
  port (A, B : in bit_vector(1 downto 0);
       P : out bit_vector(3 downto 0)
   );
end multiply_behav;

architecture behavioral of multiply_behav
is
begin
   process(A,B) is
   begin
      case A is
         when "00" =>
            if B="00" then P<="0000";
            elsif B="01" then P<="0000";
            elsif B="10" then P<="0000";
            else P<="0000";
            end if;
         when "01" =>
            if B="00" then P<="0000";
            elsif B="01" then P<="0001";
            elsif B="10" then P<="0010";
            else P<="0011";
            end if;
         when "10" =>
            if B="00" then P<="0000";
            elsif B="01" then P<="0010";
            elsif B="10" then P<="0100";
            else P<="0110";
            end if;
         when "11" =>
            if B="00" then P<="0000";
            elsif B="01" then P<="0011";
            elsif B="10" then P<="0110";
            else P<="1001";
            end if;
      end case;
end process;
end architecture;
```

16. VHDL for 2 bit binary multiplier using structural modeling

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

entity AND2 is
port(
    A,B: in BIT;
    x : out BIT);
end AND2;

architecture behavioral of AND2 is
begin
    x <= A and B;
end behavioral;
```

```
entity half_adder is
port (
    a, b : in BIT;
    sum, carry : out BIT
);
end half_adder;

architecture arch of half_adder is
begin
    sum <= a xor b;
    carry <= a and b;
end arch;
```

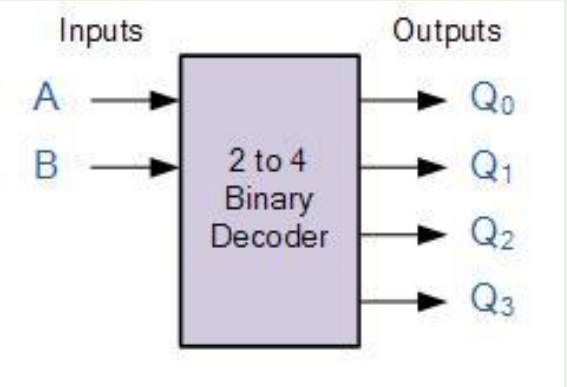
```
entity multiply_struct is
port (
    A, B : in bit_vector(1 downto 0);
    P : buffer bit_vector(3 downto 0)
);
end multiply_struct;

architecture structural of multiply_struct is
    component AND2
    port(
        A,B: in BIT;
        X : out BIT);
    end component;

    component half_adder
    port (
        A, B : in BIT;
        sum, carry : out BIT);
    end component;

signal S1,S2,S3,S4:BIT;

begin
    A1: AND2 port map(A(0),B(0),P(0));
    A2: AND2 port map(A(1),B(0),S1);
    A3: AND2 port map(A(0),B(1),S2);
    A4: AND2 port map(A(1),B(1),S3);
    H1: half_adder port map(S1,S2,P(1),S4);
    H2: half_adder port map(S4,S3,P(2),P(3));

end architecture;
```



8.3.3 Decoders

17. Write the VHDL code to implement 2:4 decoder

```
Library ieee;
use ieee.std_logic_1164.all;

entity Decoder is
port
   (  SW : in std_logic_vector (1 down to 0);
      Q : out std_logic_vector(3 down to 0);
);
end decoder;

architecture Dec_Arch of Decoder is
begin
          if (SW = “00”) then
            Q<= “0001” ;
          elseif (SW = “01”) then
            Q<= “0010” ;
          elseif (SW = “10”) then
            Q<= “0100”;
          else
            Q<= “1000”;
          Endif;
end Dec_Arch ;
```

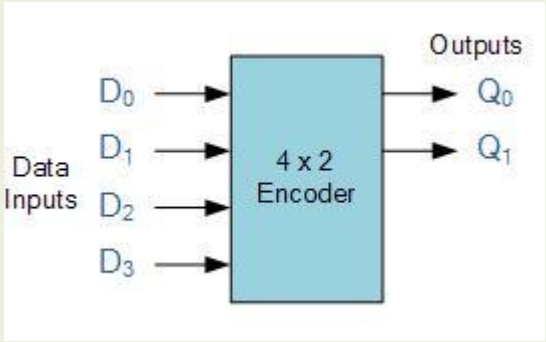


Truth Table

A	B	Q_0	Q_1	Q_2	Q_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

18. Write VHDL for 4:2 Encoder

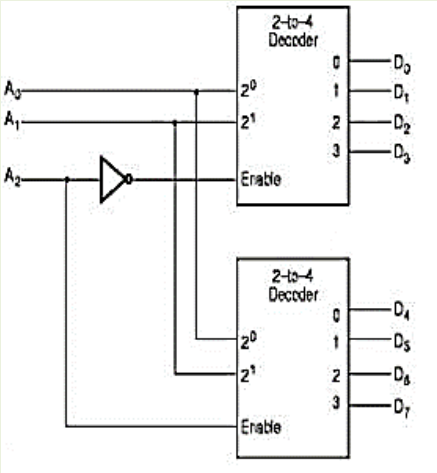
```
Library ieee;
use ieee.std_logic_1164.all;

entity Encoder is
port
      (  Q : out std_logic_vector (1 down to 0);
         D: in std_logic_vector(3 down to 0);
      );
end Encoder;

architecture arc_encoder of Encoder is
   begin
      if (D = “0001”) then
            Q<= “00” ;
      elseif (Q = “0001”) then
            Q<= “01” ;
      elseif (D= “10”) then
            Q<= “10”;
      else
            Q<= “11”;
      Endif;
end arc_encoder;
```



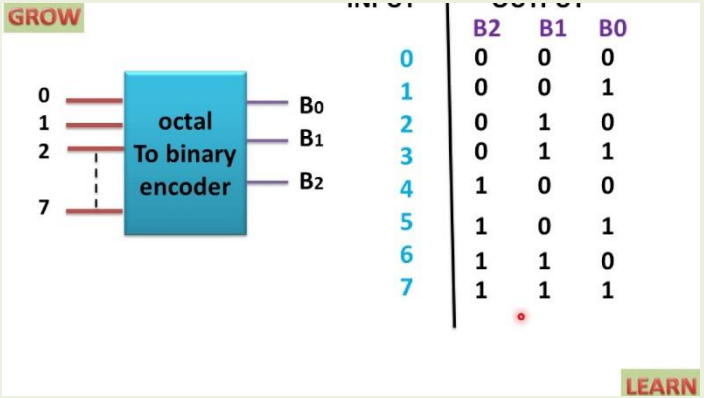
Inputs				Outputs	
D_3	D_2	D_1	D_0	Q_1	Q_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	0	x	x

Assignment:

19. Write the VHDL code to implement 3 to 8 decoder using two 2 to 4 decoder.



20. Write VHDL code to simulate Octal to Binary Encoder.



8.3.4 Multiplexers

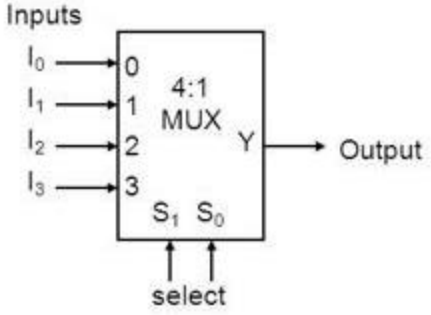
21. Write VHDL code to simulate 4:1 MUX.

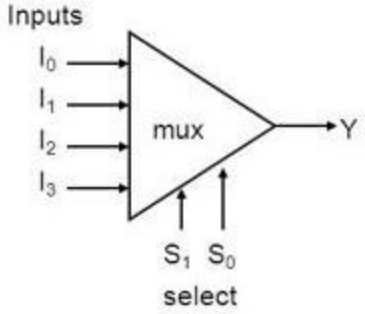
```
Library ieee;
use ieee.std_logic_1164.all;

entity MUX is
port
      (  Q : out std_logic;
         I0,I1,I2,I3 : in std_logic;
         SL: in std_logic_vector (1 down to 0);
      );
end MUX;

architecture arc_MUX of MUX is
      begin
         if (SL= “00”) then
               Q<= I0;
         elseif (SL= “01”) then
               Q<= I1 ;
         elseif (SL= “10”) then
               Q<= I2;
         else
               Q<= I4;
         Endif;
end arc_MUX;
```

Truth table for a 4-to-1 multiplexer:

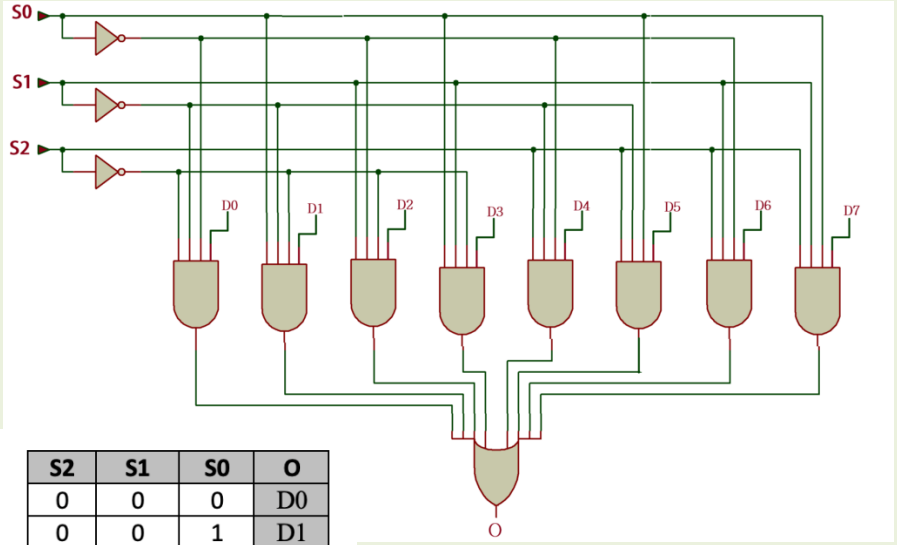
I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3





22. Write VHDL code to simulate 8:1 MUX.

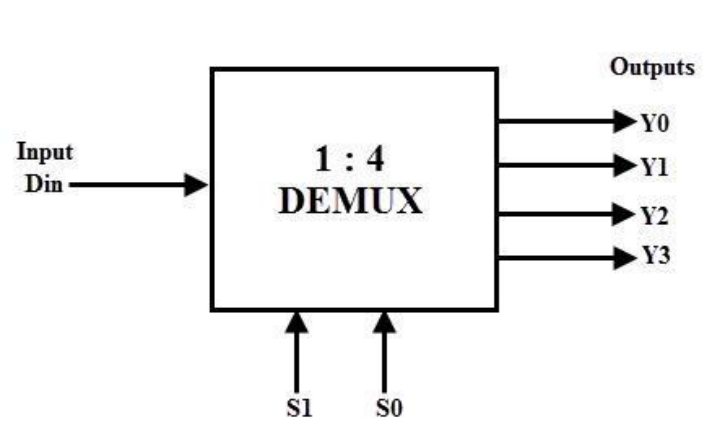


S2	S1	S0	O
0	0	0	D0
0	0	1	D1
0	1	0	D2
0	1	1	D3
1	0	0	D4
1	0	1	D5
1	1	0	D6
1	1	1	D7

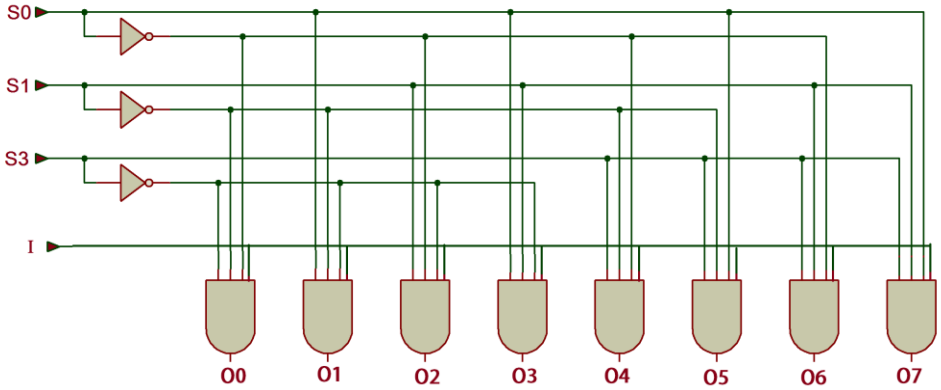
```
library ieee;
use ieee.std_logic_1164.all;

entity mux_8x1 is
port (d : in std_logic_vector (0 to 7);
      s : in std_logic_vector (0 to 2);
      o : out std_logic);
end mux_8x1;

architecture mux_arch of mux_8x1 is
  begin
   process (d,s)
    begin
      case s is
            when "000" => o<= d(0);
            when "001" => o<= d(1);
            when "010" => o<= d(2);
            when "011" => o<= d(3);
            when "100" => o<= d(4);
            when "101" => o<= d(5);
            when "110" => o<= d(6);
            when "111" => o<= d(7);
            when others => o<= '0';
       end case;
    end process;
end mux_arch;
```

23. Write VHDL code to simulate 1:4 DEMUX.

```
Library ieee;
use ieee.std_logic_1164.all;

entity DMUX is
port
      (  Din : in std_logic;
         Y0,Y1,Y2,Y3 : out std_logic;
         SL: in std_logic_vector (1 down to 0);
      );
end DMUX;

architecture arc_DMUX of DMUX is
   begin
         if (SL= “00”) then
            Q0<= Din;
         elseif (SL= “01”) then
            Q1<= Din ;
         elseif (SL= “10”) then
            Q2<= Din;
         else
            Q3<= Din;
         Endif;
end arc_MUX;
```



24. Write VHDL code to simulate 1:8 DEMUX.



S2	S1	S0	O0	O1	O2	O3	O4	O5	O6	O7
0	0	0	I	0	0	0	0	0	0	0
0	0	1	0	I	0	0	0	0	0	0
0	1	0	0	0	I	0	0	0	0	0
0	1	1	0	0	0	I	0	0	0	0
1	0	0	0	0	0	0	I	0	0	0
1	0	1	0	0	0	0	0	I	0	0
1	1	0	0	0	0	0	0	0	I	0
1	1	1	0	0	0	0	0	0	0	I

Truth Table

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

entity demux_1x8 is
port (i : in std_logic;
      s : in std_logic_vector (0 to 2);
      o : out std_logic_vector (0 to 7));
end demux_1x8;

architecture demux_arch of demux_1x8 is
 begin
  process (i,s)
   begin
     o <= "00000000";
      case s is
            when "000" => o(0)<= i;
            when "001" => o(1)<= i;
            when "010" => o(2)<= i;
            when "011" => o(3)<= i;
            when "100" => o(4)<= i;
            when "101" => o(5)<= i;
            when "110" => o(6)<= i;
            when "111" => o(7)<= i;
            when others => o<= "00000000";
      end case;
  end process;
end demux_arch;
```

8.3.5 Flipflops

25. VHDL for simple D latch

```
-- Latch is simply controlled by enable bit
-- But has nothing to do with clock signal
-- Notice this difference from flip-flops

library ieee ;
use ieee.std_logic_1164.all;

Entity D_latch is
port(data_in, enable: in std_logic;
data_out: out std_logic);
end D_latch;
```

```
Architecture behv of D_latch is
begin
  -- compare this to D flip flop

process(data_in, enable)
begin
if (enable = '1') then
  -- no clock signal here
data_out <= data_in;
end if;
end process;
end behv;
```

26. VHDL for D flipflop

```
-- Flip-flop is the basic component in sequential logic design
-- we assign input signal to the output at the clock rising edge

library ieee ;
use ieee.std_logic_1164.all;
use work.all;

entity dff is
port(data_in,clock: in std_logic;
data_out: out std_logic);
enddff;
```

```
architecture behv of dff is
begin
process(data_in,clock)
begin
    -- clock rising edge
if rising_edge(clock then
data_out <= data_in;
end if;
end process;
end behv;
```

27. VHDL for JK Flipflop

```
-- JK Flip-Flop with reset –
-- The description of JK Flip-Flop is based on functional truth table
-- Concurrent statement and signal assignment are used in this example
library ieee;
use ieee.std_logic_1164.all;

entity JK_FF is
port (clock,J,K,reset: in std_logic;
Q,Qbar: out std_logic);
end JK_FF;
```

```
architecture behv of JK_FF is
  -- define the useful signals here
signal state: std_logic;
signal input: std_logic_vector(1 downto 0);
begin
  -- combine inputs into vector
input <= J & K;
p: process(clock,reset) is
begin
if (reset = '1') then
state <= '0';
elsif (rising_edge(clock)) then
      -- compare to the truth table
```

```
case (input) is
when "11" =>
state <= not state;
when "10" =>
state <= '1';
when "01" =>
state <= '0';
when others =>
null;
end case;
end if;
end process;
  -- concurrent statements
  Q <= state;
Qbar <= not state;
end behv;
```

8.3.6 Counters

28. VHDL for n-bit counter

```
-- Behavioral description of n-bit counter
-- Another way can be used is FSM model.
Library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
generic (n: natural := 2);
port(clock, clear, count: in std_logic;
Q: out std_logic_vector(n – 1 downto 0));
end counter;

architecture behv of counter is
signal Pre_Q: std_logic_vector(n – 1 downto 0);
begin
  -- behavior describe the counter
process(clock, count, clear)
begin
if clear = '1' then
Pre_Q <= Pre_Q - Pre_Q;
elsif (clock = '1' and clock'event) then
if count = '1' then
Pre_Q <= Pre_Q + 1;
end if;
end if;
end process;
  -- concurrent assignment statement
  Q <= Pre_Q;
end behv;
```

8.3.7 Shift Registers

29. VHDL for n-bit register

```
-- KEY WORD: concurrent, generic and range
Library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity reg is
generic(n: natural := 2);
port(I: in std_logic_vector(n – 1 downto 0);
clock, load, clear: in std_logic;
Q: out std_logic_vector(n – 1 downto 0));
end reg;
```

```
architecture behv of reg is
signal Q_tmp: std_logic_vector(n – 1 downto 0);
begin
process(I, clock, load, clear)
begin
if clear = '0' then
    -- use 'range in signal assigment
Q_tmp <= (Q_tmp'range => '0');
elsif (clock = '1' and clock'event) then
if load = '1' then
Q_tmp <= I;
end if;
end if;
end process;
  -- concurrent statement
  Q <= Q_tmp;
end behv;
```

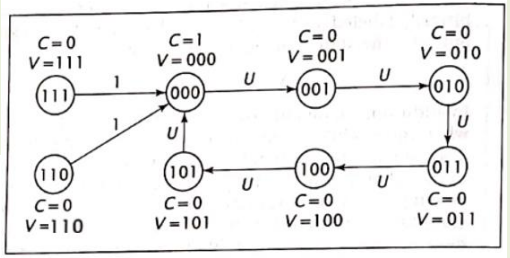
30. VHDL for n-bit shift register

```
-- 3-bit Shift-Register/Shifter
-- reset is ignored
library ieee ;
use ieee.std_logic_1164.all;

entity shift_reg is
port(I, clock, shift: in std_logic;
Q: out std_logic);
end shift_reg;
```

```
architecturebehv of shift_reg is
  -- initialize the declared signal
signal S: std_logic_vector(2 downto 0):= "111";
begin
process(I, clock, shift, S)
begin
-- everything happens upon the clock changing
If clock'event and clock = '1' then
if shift = '1' then
S <= I &S (2 downto 1);
end if;
end if;
end process;
  -- concurrent assignment
  Q <= S(0);
end behv;
```

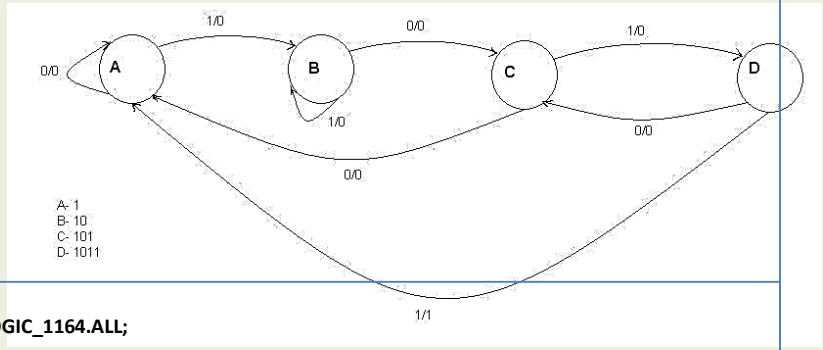
8.3.8 Sequence Detectors

31. VHDL program for Modulo 6 Counter



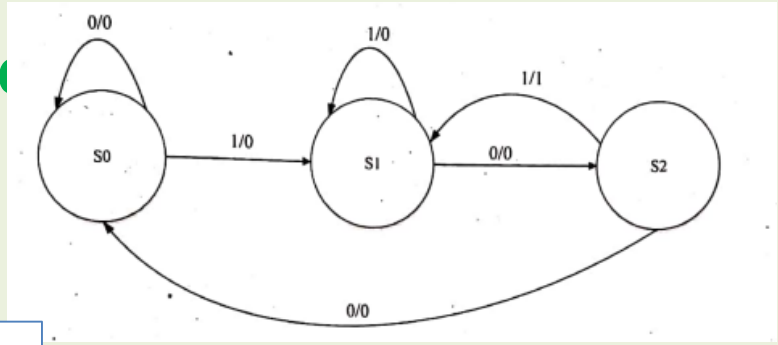

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mod6 is
      port(
            U,clk: in std_logic;
                C: out std_logic;
                V: out std_logic_vector(2 downto 0)
      );
end mod6;

architecture amod6 of mod6 is
      type states is (S0, S1, S2, S3, S4, S5, S6, S7);
      signal present_state, next_state: states;
begin
      state_mod6: process(present_state,u)
            begin
                  case present_state is
                        when S0 => V<="000"; C<='1';
                              if (U='1') then next_state <= S1;
                                    else next_state <= S0;
                              end if;
                        when S1 => V<="001"; C<='0';
                              if (U='1') then next_state <= S2;
                                    else next_state <= S1;
                              end if;
                        when S2 => V<="010"; C<='0';
                              if (U='1') then next_state <= S3;
                                    else next_state <= S2;
                              end if;
                        when S3 => V<="011"; C<='0';
                              if (U='1') then next_state <= S4;
                                    else next_state <= S3;
                              end if;
                        when S4 => V<="100"; C<='0';
                              if (U='1') then next_state <= S5;
                                    else next_state <= S4;
                              end if;
                        when S5 => V<="101"; C<='0';
                              if (U='1') then next_state <= S0;
                                    else next_state <= S5;
                              end if;
                        when S6 => V<="110"; C<='0';
                              next_state <= S0;
                        when S7 => V<="111"; C<='0';
                              next_state <= S0;
                  end case;
            end process state_mod6;
      state_transition: process(clk)
            begin
              if rising_edge(clk) then present_state <= next_state;
              end if;
            end process state_transition;
end amod6;
```

32. VHDL program for which output will be 1 when the sequence 1101 is detected considering overlapping condition.




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

--Sequence detector for detecting the sequence "1011".
--Non overlapping type.

entity seq_det is
port(    clk : in std_logic;              --clock signal
         reset : in std_logic;            --reset signal
         seq : in std_logic;              --serial bit sequence
         det_vld : out std_logic          --A '1' indicates the pattern "1011" is detected in the sequence.
);
end seq_det;

architecture Behavioral of seq_det is
   type state_dec is (A,B,C,D);           --Defines the type for states in the state machine
   signal state : state_dec ;             --Declare the signal with the corresponding state.

   begin
      process(clk)
      begin
         if( reset = '1' ) then           --resets state and output signal when reset is asserted.
            det_vld <= '0';
            state <= A;
elsif ( rising_edge(clk) ) then      --calculates the next state based on current state and input bit.
   case state is
      when A => det_vld <= '0';
            if ( seq = '0' ) then
               state <= A;
            else
               state <= B;
            end if;

      when B =>                      --when the current state is B.
            if ( seq = '0' ) then
               state <= C;
            else
               state <= B;
            end if;

      when C =>                      --when the current state is C.
            if ( seq = '0' ) then
               state <= A;
            else
               state <= D;
            end if;

      when D =>                      --when the current state is D.
            if ( seq = '0' ) then
               state <= C;
            else
               state <= A;
               det_vld <= '1';       --Output is asserted when the pattern "1011" is found in the sequence.
            end if;
      when others =>
            NULL;
   end case;
end if;
end process;
end Behavioral;
```

33. VHDL for 101 Sequence dete




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

entity mealy is
Port ( clk, din, rst : in STD_LOGIC;
dout : out STD_LOGIC);
end mealy;

architecture Behavioral of mealy is
type state is (st0, st1, st2, st3);
signal present_state, next_state : state;
begin
syncronous_process : process (clk)
begin
        if (rst = '1') then
                present_state <= st0;
        elsif rising_edge(clk) then
                present_state <= next_state;
        end if;
end process;

next_state_and_output_decoder : process(present_state, din)
begin
case (present_state) is
```

```
when st0 =>
        if (din = '1') then
                next_state <= st1;
                dout <= '0';
        else
                next_state <= st0;
                dout <= '0';
        end if;
when St1 =>
        if (din = '1') then
                next_state <= st1;
                dout <= '0';
        else
                next_state <= st2;
                dout <= '0';
                end if;
```

```
when St2 =>
        if (din = '1') then
                next_state <= st1;
                dout <= '1';
        else
                next_state <= st0;
                dout <= '0';
        end if;
when others =>
        next_state <= st0;
        dout <= '0';
end case;
end process;
end Behavioral;
```

Some More Useful Links for VHDL coding:

- https://www.engineersgarage.com/vhdl-tutorial-1-introduction-to-vhdl/
- https://www.eng.auburn.edu/~nelsovp/courses/elec4200/Slides/VHDL%202%20Combinational.pdf
- https://www.javatpoint.com/vhdl
- https://nandland.com/introduction-to-vhdl-for-beginners-with-code-examples/
- https://www.tutorialspoint.com/vlsi_design/vlsi_design_vhdl_introduction.htm
- https://www.asic-world.com/vhdl/tutorial.html

End of chapter 3