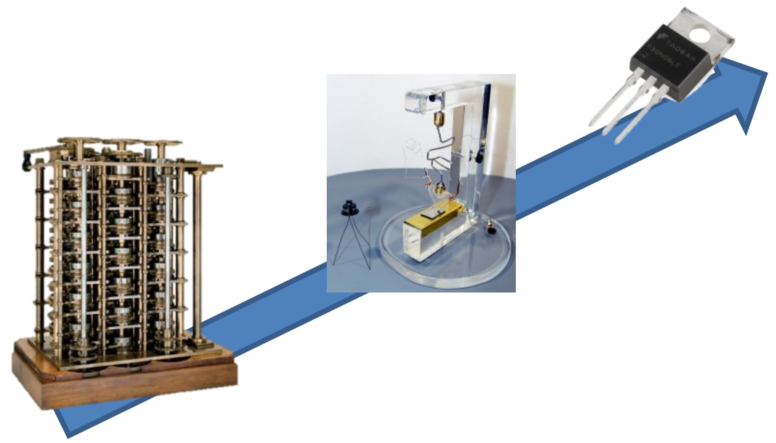
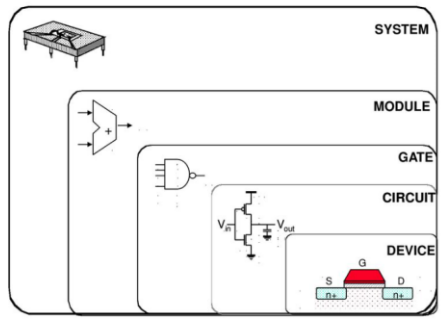
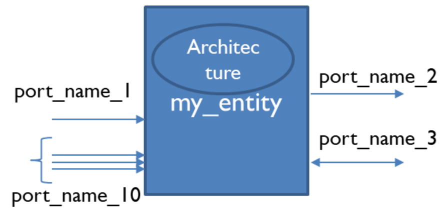
Lecture2: Concurrent vs. Sequential statements in VHDL & Combinatorial Circuits: Demultiplexer, Decoder, Encoder

Vincent Helbig
Original Author: Emad Samuel Malki Ebeid

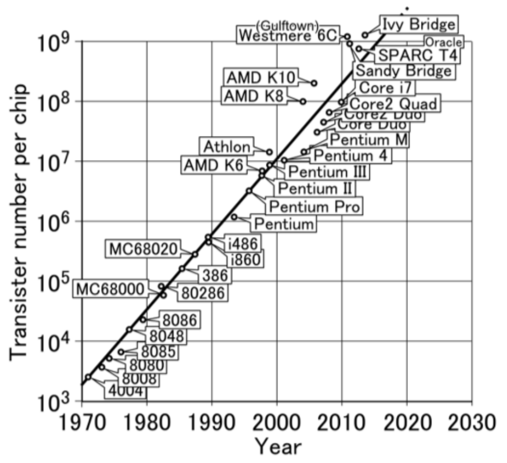
Summary of Lecture 1



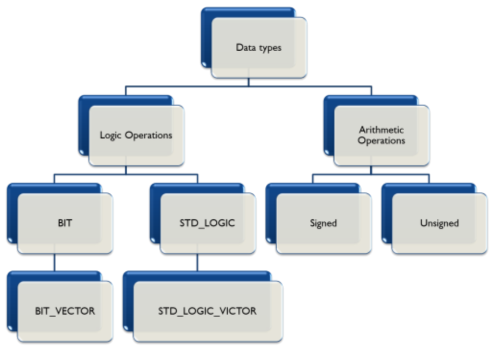


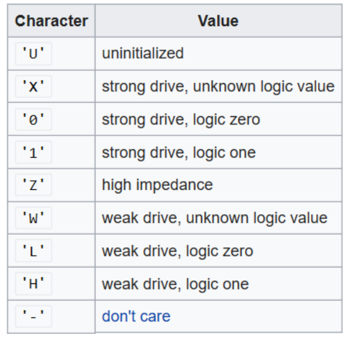


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity AND_Gate is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           C : out STD_LOGIC);
end AND_Gate;

architecture Behavioral of AND_Gate is

begin
C <= A and B;

end Behavioral;
```

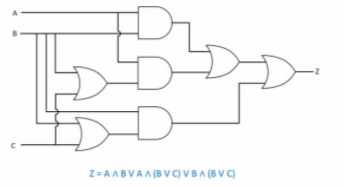


VHDL

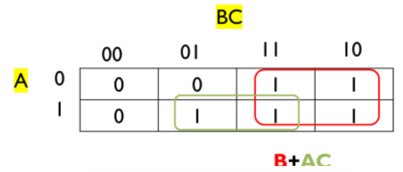




Character	Value
'U'	uninitialized
'X'	strong drive, unknown logic value
'0'	strong drive, logic zero
'1'	strong drive, logic one
'Z'	high impedance
'W'	weak drive, unknown logic value
'L'	weak drive, logic zero
'H'	weak drive, logic one
'-'	don't care

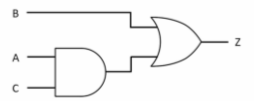


Z = A ∧ B ∨ A ∧ (B ∨ C) ∨ B ∧ (B ∨ C)

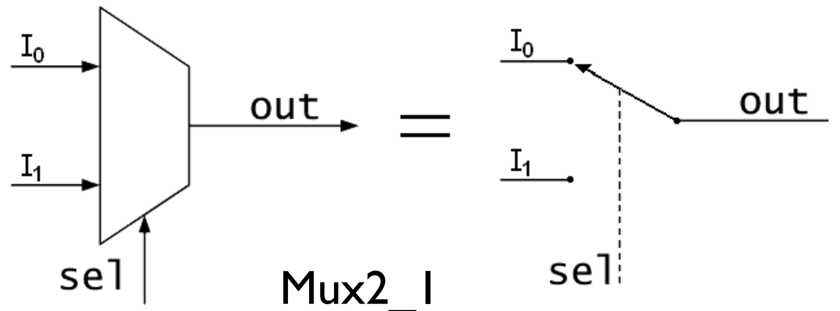


A \ BC	00	01	11	10
0	0	0	1	1
1	0	1	1	1

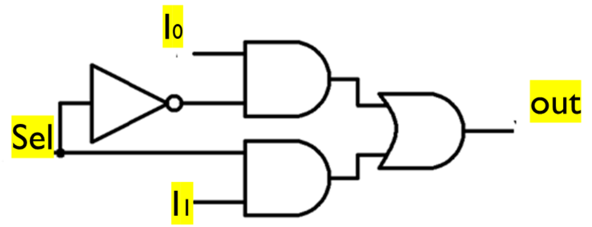
B+AC



Multiplexer



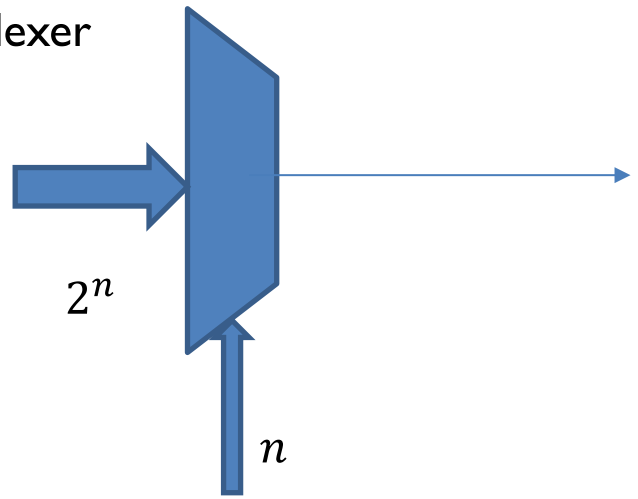
Mux2_1



Out = Sel' * I0+Sel*I1

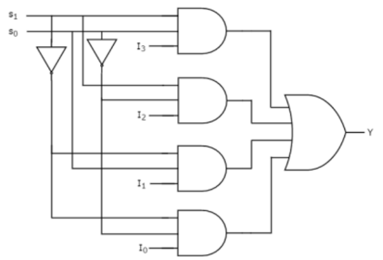
Out<= (not(sel) AND I0) OR (sel AND I1)

General Multiplexer



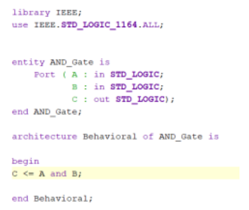
n select inputs can select from one of up to 2^n inputs.

Mux4_1



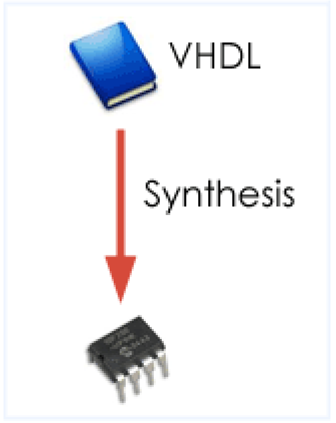
Will it be always described by gates?

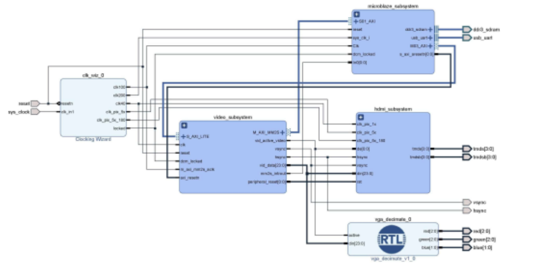
VHDL Coding Styles



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


entity AND_Gate is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           C : out STD_LOGIC);
end AND_Gate;

architecture Behavioral of AND_Gate is

begin
C <= A and B;

end Behavioral;
```

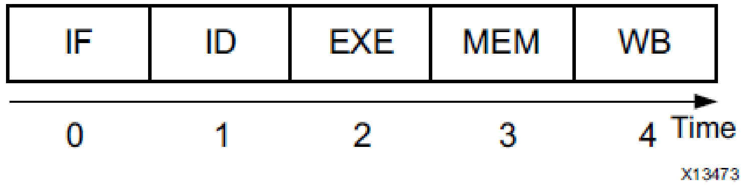
- **Dataflow/Register-Transfer Level** (RTL): describes how data is transformed as it is passed from register to register. The transformation of the data is performed by the combinational logic that exists between the registers. For example logic gates, mux, flip flops, etc.
- **Behavioral:** describes how the circuit should behave without an intention for synthesis. The behavioral design may include operations such as integer division or propagation delays that are difficult or impossible to synthesize. It can be used in the simulation. For example infinite loops, waits, test benches, etc.
- **Structural**: describes the interconnection of components within an architecture. For example: block design
- **Hybrid Style:** A combination of the previous design styles.



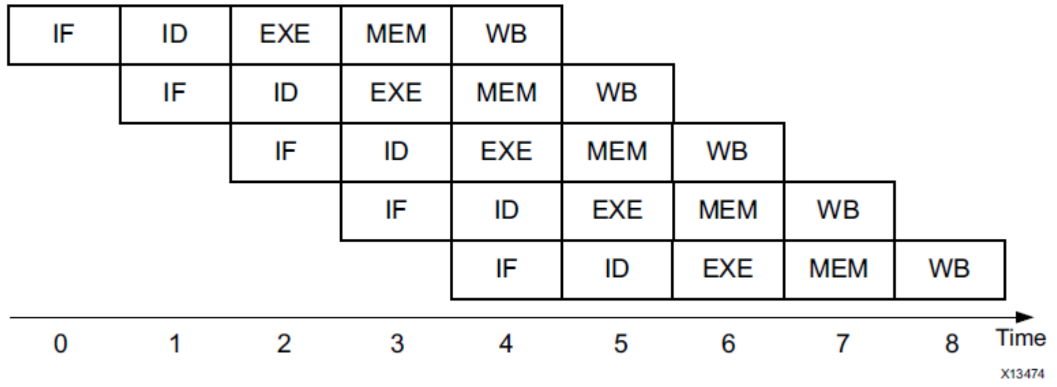


Programming Paradigm (Ch.4)

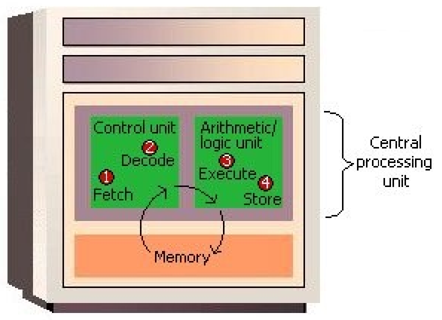
- Most programming languages (C, Java, python, etc) implement functionalities in a sequential manner, one instruction at a time.
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Execute (EXE)
  - Memory operations (MEM)
  - Write back (WB)



Processor Instruction Execution Stages

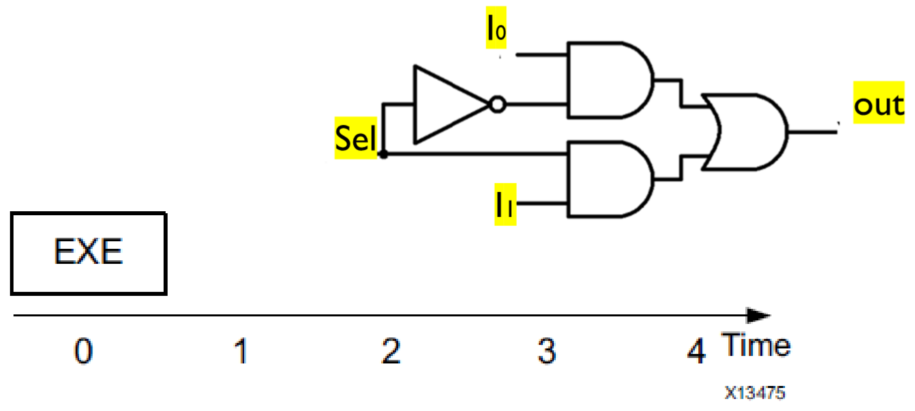


Processor with Multiple Instruction Execution Units

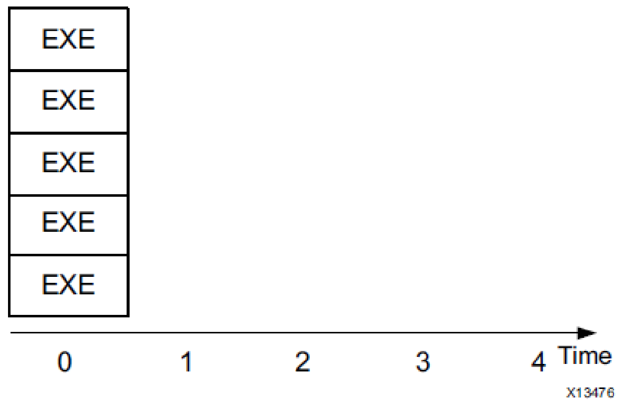


VHDL programming paradigm

- VHDL programming is significantly different than any processor-based programming languages.
- VHDL has the ability to execute a virtually *unlimited number of statements* at the same time and in a concurrent manner (i.e., in parallel).



FPGA Instruction Execution Stages



FPGA with Multiple Instruction Execution Units

Example of concurrent statements

Listing 4.1: VHDL code for the circuit of Figure 4.1.

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

-- entity
entity my_circuit is
    port ( A_1,A_2,B_1,B_2,D_1 : in  std_logic;
           E_out               : out std_logic);
end my_circuit;

-- architecture
architecture my_circuit_arc of my_circuit is
    signal A_out, B_out, C_out : std_logic;
begin
    A_out <= A_1 and A_2;
    B_out <= B_1 or B_2;
    C_out <= (not D_1) and B_2;
    E_out <= A_out or B_out or C_out;
end my_circuit_arc;
```

- The statement's order does not matter!

Listing 4.2: Equivalent VHDL code for the circuit of Figure 4.1.

```
C_out <= (not D_1) and B_2;
A_out <= A_1 and A_2;
B_out <= B_1 or B_2;
E_out <= A_out or B_out or C_out;
```

Listing 4.3: Equivalent VHDL code for the circuit of Figure 4.1.

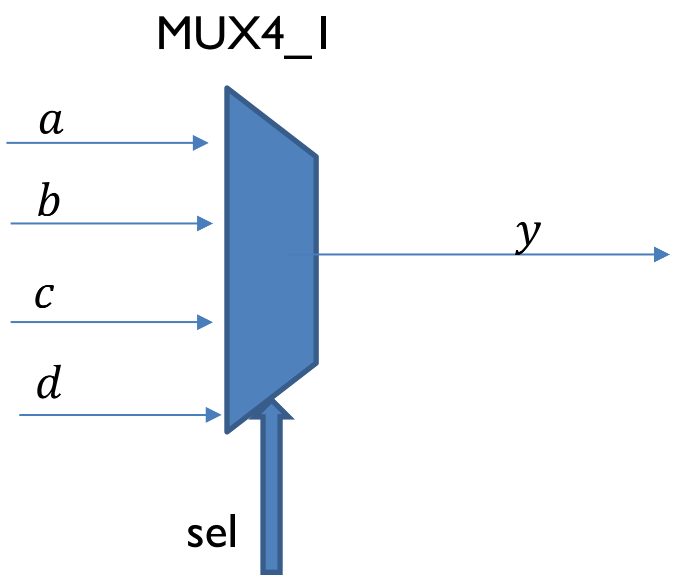
```
A_out <= A_1 and A_2;
E_out <= A_out or B_out or C_out;
B_out <= B_1 or B_2;
C_out <= (not D_1) and B_2;
```

Listing 4.4: Equivalent VHDL code for the circuit of Figure 4.1.

```
B_out <= B_1 or B_2;
A_out <= A_1 and A_2;
E_out <= A_out or B_out or C_out;
C_out <= (not D_1) and B_2;
```

Conditional signal assignment **(when)**

- The term conditional signal assignment is used to describe statements that have only one target but can have more than one associated expression assigned to the target.
- The individual conditions are evaluated *sequentially* in the conditional signal assignment statement until the first condition evaluates as true. When true, the associated expression is evaluated and assigned to the target.
- It is a more generic style of describing the system.

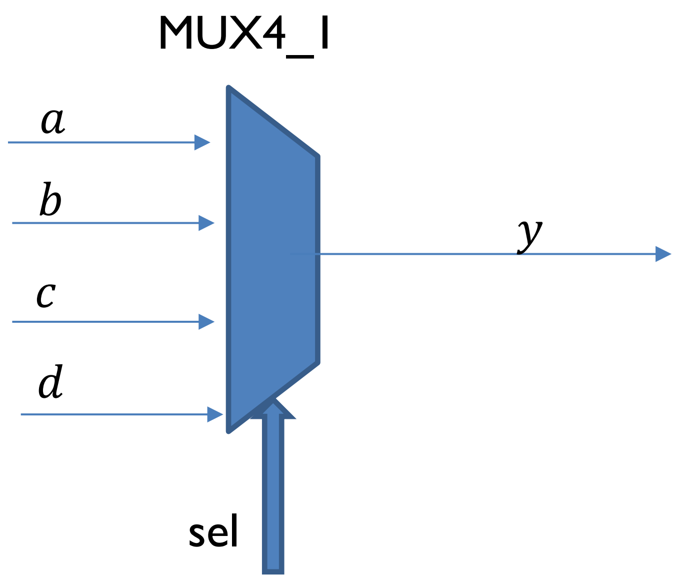


```
y <= a when sel = "00" else
     b when sel = "01" else
     c when sel = "10" else
     d; --when sel = "11" end
```

Selected Signal Assignment (**with select)**

- No redundancy in the code here compares with (when)

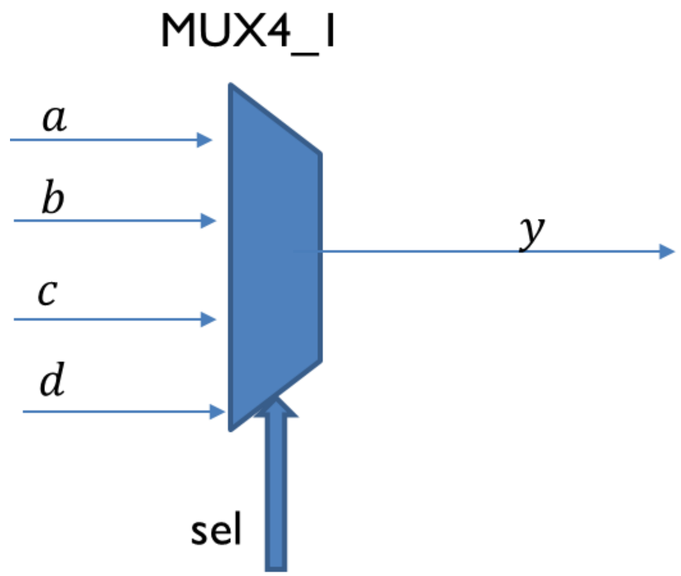
```
with sel select
   y <= a when "00",
        b when "01",
        c when "10",
        d when "11",
        a when others;
```



Process statement **(if/else)**

- It is a statement that contains a certain number of instructions that, when the process statement is executed, are executed sequentially

```
process (sel, a, b, c, d)
begin
  if (sel = "00") then
    y <= a;
  elsif (sel = "01") then
    y <= b;
  elsif (sel = "10") then
    y <= c;
  else
    y <= d;
  end if;
end process;
```



- The process statement in itself is a concurrent statement and will be executed together with the other concurrent statements in the body of the architecture.

Process statement **(case)**

- The case statement is similar to the ***if*** statement in that a sequence of statements is executed if an associated expression is true.
- The case statement differs from the **if** statement in that the resulting choice is made depending upon the value of *the single control expression*.

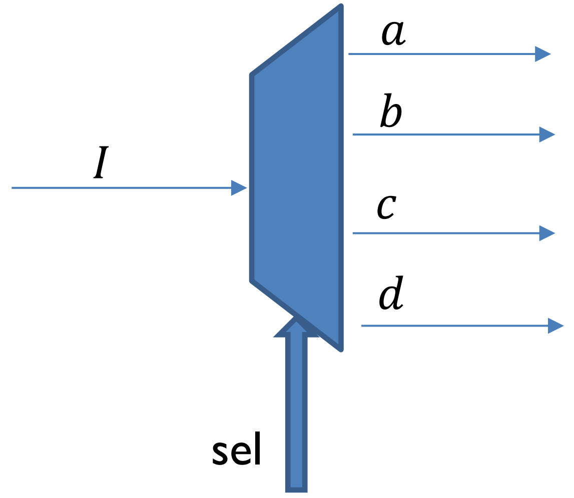
```
process (sel, a, b, c, d)
begin
  case sel is
    when “00”     => y <= a;
    when “01”     => y <= b;
    when “10”     => y <= c;
    when “11”     => y <= d;
    when others   => y <= a;
  end case;
end process,
```

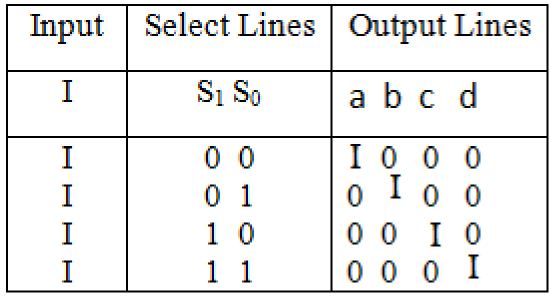
Demultiplexer (**DEMUX**)

- It selects one output from the multiple output line and fetches the single input through the selection line.
- If we use the *process* statement to write the VHDL code, what should go to the sensitivity list?

10 minutes: write the VHDL code of DEMUX4_1 using process statement (if/else)

DEMUX4_1

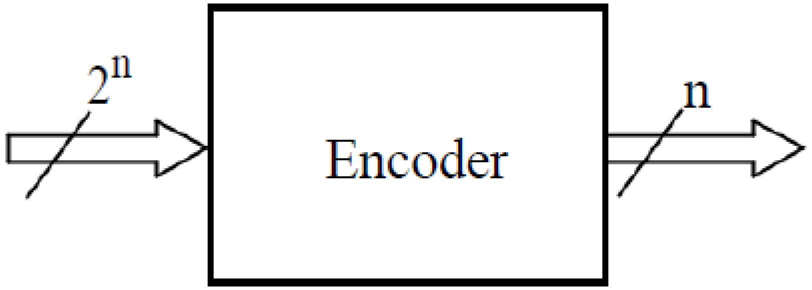


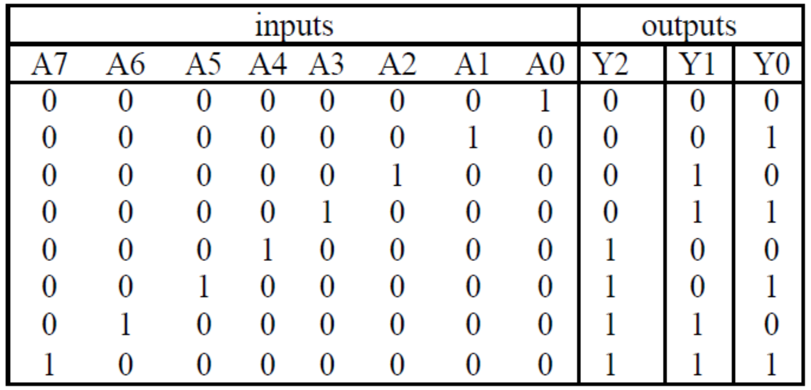


Input	Select Lines	Output Lines
I	S_1 S_0	a b c d
I	0 0	I 0 0 0
I	0 1	0 I 0 0
I	1 0	0 0 I 0
I	1 1	0 0 0 I

Encoders

- Discrete quantities of digital information, data are often represented in a coded form; binary being the most popular.
- Encoders are used to encode data into a coded form and decoders are used to convert it back into its original uncoded form.
- An encoder that has 2^n(or less) input lines encodes input data to provide *n* encoded output lines.
- It is assumed that only one input has a value of 1 at any given time, otherwise the output has some undefined value and the circuit is meaningless.





inputs								outputs		
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Truth table of a 8-3 binary encoder

Encoders

- All models of encoders circuit must use a default "don't care" value to minimize the synthesized circuit as only 8 of 256 (28) input conditions need to be specified.
- The synthesis tool, if capable, replaces "don't care" values with logic 0 or logic 1 values as necessary in order to minimize the circuit's logic.

```
use ieee.std_logic_arith.all;

entity encoder8_3 is
  port (a: in   std_logic_vector(7 downto 0);
        y: out std_logic_vector(2 downto
0)); end entity encoder8_3;

architecture first of encoder8_3
is begin
  y <=  "000" when a = "00000001" else
        "001" when a = "00000010" else
        "010" when a = "00000100" else
        "011" when a = "00001000" else
        "100" when a = "00010000" else
        "101" when a = "00100000" else
        "110" when a = "01000000" else
        "111" when a = "10000000" else
        "---";
end architecture first;
```

Encoder 8-3

```
entity encoder8_3 is
  port (a: in   std_logic_vector(7 downto 0);
        y: out std_logic_vector(2 downto
0)); end entity encoder8_3;
architecture first of encoder8_3
is begin
  y <=  "000" when a = "00000001" else
        "001" when a = "00000010" else
        "010" when a = "00000100" else
        "011" when a = "00001000" else
        "100" when a = "00010000" else
        "101" when a = "00100000" else
        "110" when a = "01000000" else
        "111" when a = "10000000" else
        "---";
end architecture first;
```

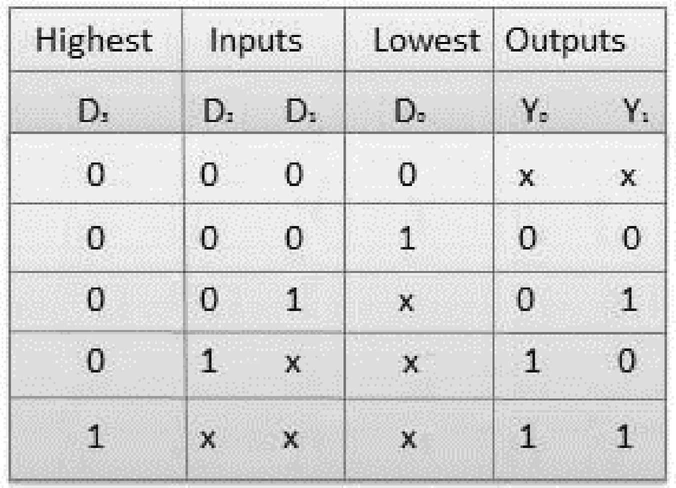
```
  architecture second of encoder8_3 is
  begin
   with a  select
     y <= "000" when "00000001",
          "001" when "00000010",
          "010" when "00000100",
          "011" when "00001000",
          "100" when "00010000",
          "101" when "00100000",
          "110" when "01000000",
          "111" when "10000000",
          "---" when others;
 end architecture second;
```

```
architecture fourth of encoder8_3
is begin
  process (a)
   begin
    case a is
      when "00000001" => y <= "000";
      when "00000010" => y <= "001";
      when "00000100" => y <= "010";
      when "00001000" => y <= "011";
      when "00010000" => y <= "100";
      when "00100000" => y <= "101";
      when "01000000" => y <= "110";
      when "10000000" => y <= "111";
      when others => y <= "---";
    end case;
  end process;
end architecture fourth;
```

```
architecture third of encoder8_3
is begin
  process (a)
  begin
    if (a = "00000001") then y <= "000";
    elsif (a = "00000010") then y <= "001";
    elsif (a = "00000100") then y <= "010";
    elsif (a = "00001000") then y <= "011";
    elsif (a = "00010000") then y <= "100";
    elsif (a = "00100000") then y <= "101";
    elsif (a = "01000000") then y <= "110";
    elsif (a = "10000000") then y <= "111";
    else y <= "---";
    end if;
  end process;
end architecture third;
```

Priority Encoders

- A modified version of an encoder. It gives a priority to an input signal and provides an output based on that priority.

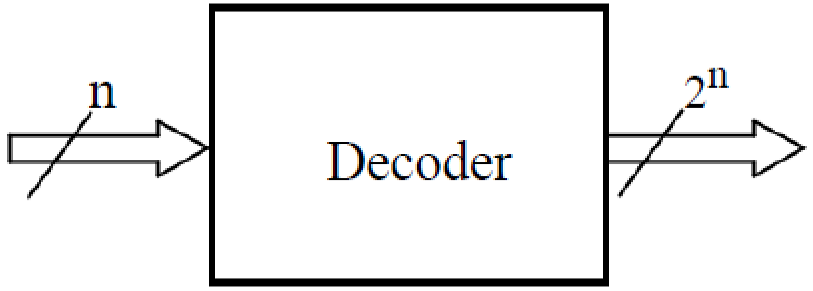


Highest	Inputs		Lowest	Outputs	
D_3	D_2	D_1	D_0	Y_0	Y_1
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

10 minutes: write the VHDL code of Priority Encoders

Decoders

- Decoders are used to decode data that has been previously encoded using a binary or possibly another type of coded format.
- An n-bit code can represent up to 2^n distinct bits of coded information, so a decoder with n inputs can decode up to 2^n outputs.



inputs			outputs							
A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Truth table of a 3-8 binary decoder

3-6 binary decoder with enable

```
entity decoder3_6 is
  port (en: in std_logic;
        a: in std_logic_vector(2 downto 0);
        y: out std_logic_vector(5 downto 0));
end entity decoder3_6;

architecture first of decoder3_6
is begin
  process (a, en)
  begin
    if (en = '0') then
      y <= "000000";
    else
      case a is
        when "000" => y <= "000001";
        when "001" => y <= "000010";
        when "010" => y <= "000100";
        when "011" => y <= "001000";
        when "100" => y <= "010000";
        when "101" => y <= "100000";
        when others => y <= "000000";
      end case;
    end if;
  end process;
end architecture first;
```

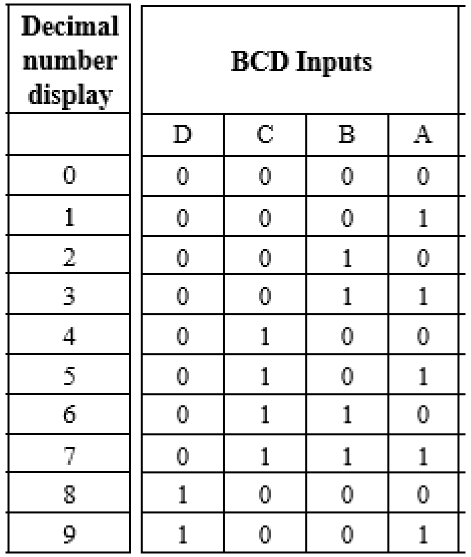
Inputs				outputs					
En	A2	A1	A0	Y5	Y4	Y3	Y2	Y1	Y0
0	–	–	–	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	1	0
1	0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

– =don't care

Binary-coded Decimal (BCD) & 7 segments

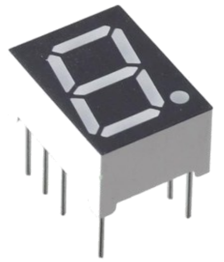
BCD

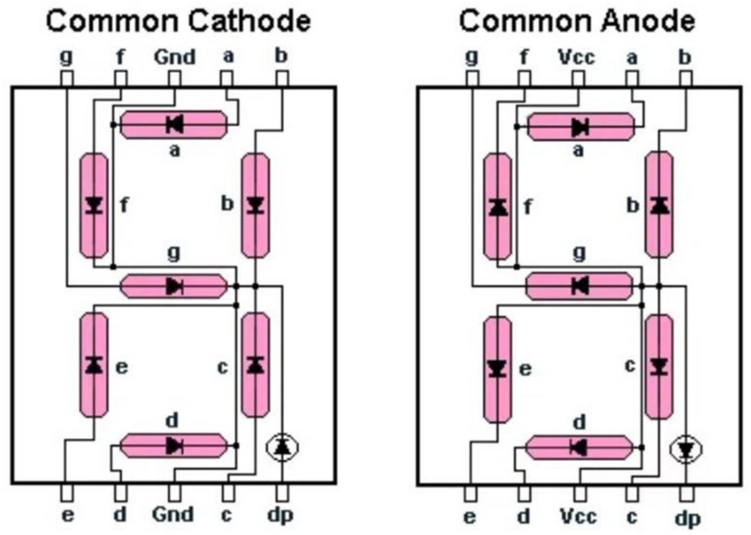
- BCD is a binary encodings of decimal numbers where each digit is represented by a fixed number of bits.

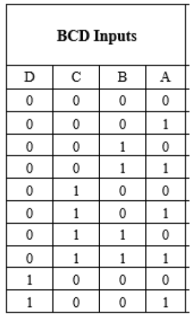


Decimal number display	BCD Inputs			
	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

7 segments

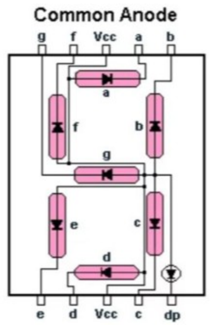






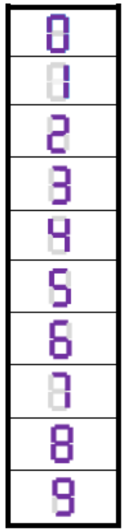
BCD Inputs			
D	C	B	A
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1

Binary-coded Decimal (BCD) to 7 segment decoder

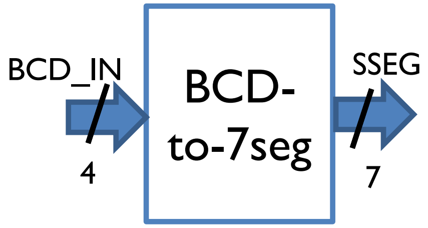


Truth table for 7 segment common anode displaying decimal digits:

a	b	c	d	e	f	g
0	0	0	0	0	0	1
1	0	0	1	1	1	1
0	0	1	0	0	1	0
0	0	0	0	1	1	0
1	0	0	1	1	0	0
0	1	0	0	1	0	0
1	1	0	0	0	0	0
0	0	0	1	1	1	1
0	0	0	0	0	0	0
0	0	0	1	1	0	0



BCD-to-7seg decoder in VHDL



```
with BCD_IN select
    SSEG <= "0000001" when "0000",    -- 0
            "1001111" when "0001",    -- 1
            "0010010" when "0010",    -- 2
            "0000110" when "0011",    -- 3
            "1001100" when "0100",    -- 4
            "0100100" when "0101",    -- 5
            "0100000" when "0110",    -- 6
            "0001111" when "0111",    -- 7
            "0000000" when "1000",    -- 8
            "0000100" when "1001",    -- 9
            "0001000" when "1010",    -- A
            "1100000" when "1011",    -- b
            "0110001" when "1100",    -- C
            "1000010" when "1101",    -- d
            "0110000" when "1110",    -- E
            "0111000" when "1111",    -- F
            "1111111" when others;    -- turn off all LEDs
```