CAD Tools:

- Logic circuits for complex systems such as today’s computers cannot be designed manually. They are designed using CAD tools.

- To design a logic circuit a no. of CAD tools are required. They are usually packaged together into a CAD system which typically includes tools for design, synthesis, simulation and physical design.

- To design a logic circuit the source code is written using HDL (Hardware Description Language)
- HDL is similar to typical computer programming language, but it is used to describe hardware rather than a program to be executed on a computer
- IEEE standard languages are used such as VHDL(Very high speed integrated circuit Hardware Description Language), Verilog HDL etc.
- VHDL code can be written in a modular way that facilitates hierarchical design. Both large and small logic ckts can be efficiently represented in VHDL.

Max + plus II is one of the most easiest flatform to use CAD systems

- Each logic ckt designed in Max + plus II is called a project. The s/w works on one project at a time and keeps all information for that project in a single directory
- So to begin the first step is to create a directory to hold the files of the project.

Steps to follow:

1. **Start → Programs → Altera → MAX + plus II**

   The MAX + plus II manager window will pop up.
2. Select **File** from the menu bar, then select **Project → Name** and give project name. The project name will come on the title bar. This step is optional. If skipped, before compilation follow **File → Project → Set Project to Current File**.
3. Then follow **MAX + plus II → Text Editor**. Text Editor is used to type the source code.
4. Save the source code with file name same as project name but with .vhd extension in the required directory.
5. Then follow **MAX + plus II → Compiler**. If step 2 is skipped, follow **File → Project → Set Project to Current File** first and then select Compiler. The compiler window will pop up.

6. Click **Start**. The source file will be compiled and errors if any will be displayed. Double click on the first error to rectify it. Rectify all the errors and recompile to get error free compiled file.
7. Select **MAX + plus II → Waveform Editor** . The Waveform Editor window will pop up. Then select **Node →Enter Nodes from SNF** (*simulator netlist file)*and then in the popped up box click **List** button to display the names of input and output nodes in the box labeled **Available Nodes & Groups.** Select the nodes if not selected, and click **=>** button to copy them into **Selected Nodes & Groups** box. Click **OK** to return to waveform editor. The nodes are now displayed in waveform display.
8. Now the values to the inputs will be given. Select **File → End Time** to specify the total amount of time for simulation. If not specified it will take default time.

9. Select **View → Fit in Window** so that entire time range is visible in the waveform editor display. Select **Options → Grid Size** to give appropriate grid size
10. Either the single value or different values at different grids can be given to the input signals. To give single value select the entire signal and click the required value which is activated in the left side of the window. To give different values select a section of the signal by dragging the mouse over it and then by clicking the required value. After giving the values save the file with same name as before but with .scf (*simulator channel file)* extension.
11. Select **MAX + plus II → Simulator.** Click **Start.** A message box will be displayed indicating no errors. Click **OK**. And then **Open SCF** to view the waveforms after simulation.

Any data information in VHDL is represented as data object.

3 types:

1. Signals
2. Constants
3. Variables

Data object names

1. Cannot be a VHDL keyword.
2. It must begin with a letter
3. It cannot end with an _
4. It cannot have two successive _ _

Data object values:

Individual values: with in single quote eg. ‘0‘

Multiple bits : with in double quotes eg. “1001” it means decimal 9

Signal data object:

Is used in 3 places:

1. In entity declaration.
2. Declarative section of Architecture body
3. Declarative section of package.

Declarative section of architecture is inside the architecture body before begin.

Signal declaration:

signal *signal name* : type_name

Types:

std_logic: 0 or 1 or X

std_logic_vector: std_logic_vector(0 to 4) lsb has highest index
or std_logic_vector(4 downto 0) msb has highest index

Integer Types:

Can be used for arithmetic operations.

32 bit representation

Integer with fewer bits can also be declared

Eg. Signal X: integer range -128 to +127.

signal assignment operator is <=

Constant data objects:

constant *const_name* : type_name := const_value

Eg. constant zero : std_logic_vector(3 downto 0) := “0000”

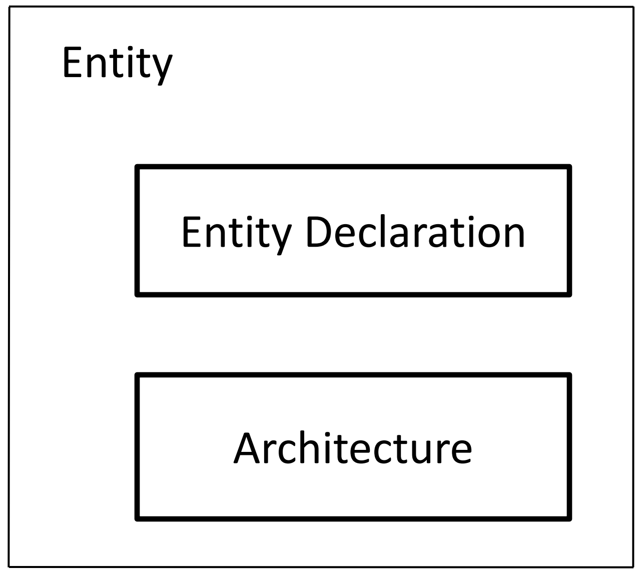
Variable data objects:

variable *variable_name* : type_name

Variable assignment operator is :=

The only place where the variable can be declared is inside the process and its scope is limited to the process.

Structure of VHDL Design Entity



A circuit described with VHDL code is called a design entity

It has two main parts:
Entity declaration: specifies the input and output signals for the entity
Architecture: gives the circuit details

General form of entity declaration:

```
ENTITY entity_name IS
PORT (signal_name {, signal name}:  mode type_name; ---------);
END entity_name;
```

The words in capitals are keywords. VHDL is case insensitive. entity_name must be same as VHDL file name (source file). Mode may be either IN or OUT. Type name may be std_logic or std_logic _vector or integer

The project name, file name and the entity name are same.

General form of architecture:

```
ARCHITECTURE architecture_name OF entity_name IS

[declarations if any]

BEGIN

Statements

END architecture_name
```

The general form has two parts:

Declarative region – appears preceding the BEGIN keyword

Architecture body – where the functionality of the entity is specified and it follows the BEGIN keyword

An example:

-- is used for comment

```
library ieee;     --standard library which must be included always
use ieee.std_logic_1164.all;  -- a package of ieee library which
                                         --must be included always
 entity andgate is        -- entity name is andgate , so file name
                          --should be andgate.vhd.
port ( ax,bx : in std_logic;
      cx : out std_logic);
end andgate;
architecture arch_andgate of andgate is
begin
cx<=ax and bx;
end arch_andgate;
```

3 styles of programming:

1.Data flow

2. Behavioral

3. Structural

1.Data flow:

The logic circuit is described using concurrent expressions in architecture body.

2. Behavioral:

The behavior of the logic circuit is described by a set of assignment statements.

3. Structural:

This is a hierarchical design. The big design is divided into different components and described separately. And then they are integrated.

A VHDL entity defined in one source code file can be used as a component in another source code file.

The general form of component declaration:

COMPONENT component_name

PORT (signal_name : mode type_name;……..);

END COMPONENT;

component_name is the VHDL file name. PORT is the PORT of the entity of that file.

The component can be instantiated using component instantiation whose general form is

instance_name : component_name PORT MAP (formal_name => actual_name, ………..);

Each formal_ name is the name of port in the component. Each actual_name is the name of a signal in the entity that instantiates the component. The syntax “formal_name => “ is provided so that the order of the signals following PORT MAP need not be in the same order as in the component. This is known as named association. Otherwise “formal_name => “ is not needed and that is called positional associttion.

An example: structural code for 4 bit adder

Behavioral code for full adder:

```
library ieee;
use ieee.std_logic_1164.all;
 entity fulladder is
port ( af,bf,cf : in std_logic;
       sf,cof : out std_logic);
end fulladder;
architecture arch_fulladder of fulladder is
begin
sf<= (af XOR bf) XOR (cf);
cof<= (af AND bf) OR (af AND cf) OR (bf AND cf);
end arch_fulladder;
```

Code for 4 bit adder using fulladder as the component:

```
library ieee;
use ieee.std_logic_1164.all;
 entity  adder4bit is
 port(a ,b  : in std_logic_vector(3 downto 0); c  : in std_logic;
      s  : out std_logic_vector(3 downto 0); co : out std_logic);
end  adder4bit;
architecture arch_4bitadder of  adder4bit is
component fulladder is
port ( af,bf,cf : in std_logic; sf,cof : out std_logic);
end component;
signal carry: std_logic_vector(2 downto 0);
begin
fa_0: fulladder port map(a(0),b(0),c,s(0),carry(0));
fa_1: fulladder port map(a(1),b(1),carry(0),s(1),carry(1));
fa_2: fulladder port map(a(2),b(2),carry(1),s(2),carry(2));
fa_3: fulladder port map(a(3),b(3),carry(2),s(3),co);
end arch_4bitadder;
```

Sequential assignment statements:

Order of the statements in the code affect the program execution

Three variants:

1. if statement
2. case statement
3. loop statement

Process statement:

Sequential statements must be separated from the concurrent statements. This is accomplished using process statement

The process statement appears inside the architecture body and it encloses other sequential statements with in it. The if,case and loop statements can appear only within process statement

process (*sensitivity list)*

variable declaration;

begin

end process;

if statement:

if *expression* then
statement;

else if *expression* then
statement;

else
statement;

end if;
end if

case statement:

case *expression* is
when *const_value* => *statement* ;
when *const_value* => *statement* ;
.
.
.
.when others => *statement* ;

end case;

case *sel* is

when ‘0’ => f<= x1;

when others => f<=x2;

end case;

loop statements:

Two types:	for loop and while loop

for loop:

```
        loop lable:
                for variable_name in range  loop
                        statement;
                end loop loop lable;
```

```
loop label:
while *boolean expression*   loop
        statement;
end loop loop lable;
```

Concurrent Assignment Statements:

These are used to assign a value to a signal in architecture body.

VHDL provides 4 different types of concurrent assignment statements :

- Simple signal assignment
- Selected signal assignment
- Conditional signal assignment
- Generate statement

Simple signal assignment statement:

It is used for a logic or an arithmetic expression.

The general form is

Signal_name <= expression;

VHDL also supports multibit logic expression

Eg1:

```
Signal a,b,c: std_logic_vector(0 to 3);
.
.
c <= a AND b;
```

Here c(0) = a(0) AND b(0), c(1) = a(1) AND b(1),
          c(2) = a(2) AND b(2), c(3) = a(3) AND b(3)

Eg2:

Signal a,b,c: std_logic_vector(0 to 3);

.

.

c <= a+b;

This is a 4 bit adder without carry in and carry out

Assigning signal values using others:

To set all bits in the signal S to 0, we can write as follows:

```
S<= (OTHERS => ‘0’)
```

The meaning of (OTHERS => value) is set each bit of the destination operand to value

Selected signal assignment statement:

It is used to set the value of a signal to one of several alternatives based on a selection criterion. The general form is

```
with  expression select
   signal_name <=  expression when const_value,
                   expression when const_value,
                   .
                   .
                   expression when const_value;
```

Eg:

2 to 1 mux with x1and x2 as data inputs and sel as the select input and f as output:

```
with  sel select
f<= x1 when ‘0’,
     x2 when others;
```

Conditional signal assignment statement:

It is used to set the value of a signal to one of several alternatives. The general form is

```
Signal_name <= expression when logic expression else
               expression when logic expression else
               .
               .
                expression;
```

```
Eg1:   f <= '1' when x1=x2 else
             '0';
```

Eg2:

```
F <=    “01” when req1 = ‘1’ else
        “10” when req2 = ‘1’ else
        “11” when req3 = ‘1’ else
        “00”;
```

Generate statement:

Two types:

- For generate
- If generate

The general formats are

generate_ label:

for index variable *in* range *generate*

statement;

statement;

end generate;

```
generate_ label:
 if expression generate
statement;
statement;
end generate;
```

Defining an entity with generic:

A program can be made more general by defining the entity with generic

Eg:

entity addern is

generic (n:integer := 4);

Port (x,y : *in std_logic_vector*(n-1 *downto* 0);

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity nbitadder1 is
generic(n:integer:=4);
port(anba,bnba : in std_logic_vector((n-1) downto 0);
    cnba : in std_logic;
    snba : out std_logic_vector((n-1) downto 0);
    conba: out std_logic);
end nbitadder1;
```

```
architecture arch_nbitadder of nbitadder1 is
component fulladder1 is
port ( af,bf,cf : in std_logic;
       sf,cof : out std_logic);
end component;
signal carry: std_logic_vector(n downto 0);
```

```
begin
 carry(0) <= cnba;
ga_0: for i in 0 to (n-1) generate
fa_i: fulladder1
port map(anba(i),bnba(i),carry(i),snba(i),carry(i+1));
end generate;
conba <= carry(n);
end arch_nbitadder;
```

Sequential circuits: (FF’s, registers, counters etc)

VHDL uses a construct called an attribute .

An attribute refers to a property of an object such as signal .

Eg. Assume that *clock* is a signal.

clock ’event → here *‘event* is an attribute. It refers to any change in the *clock* signal.

clock ’event and clock = 0 → this means that the value of the clock signal has just changed and the value is now equal to 0. This implies a –ve clock edge.

D FF:

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
port(d,clock : in std_logic;
        q,qbar : out std_logic);
end dff;
architecture arch_dff of dff is
begin
process(clock)
begin
  if clock’event and clock=‘1’ then
    q<=d;
    qbar <= not (d);
end if;
end process;
end  arch_dff;
```

Using wait until statement:

```
 library ieee;
use ieee.std_logic_1164.all;
entity dff is
port(d,clock : in std_logic;
        q,qbar : out std_logic);
end dff;
architecture arch_dff of dff is
begin
process
begin
   wait until clock’event and clock = ‘1’;
     q<=d;
qbar<=not(d);
 end process;
end  arch_dff;
```

A process that uses wait until statement is a special case because the sensitivity list is omitted. It implies that the sensitivity list only includes clock. And we can use wait until statement only if it is the first statement in the process

If a process only defines flip flops then it makes no difference which construct is used. But in practical designs a process often includes many statements. If one or more of these statements specify a combinational sub circuit, then it is necessary to use if statement to infer flip flops. In this case any signals that are assigned values inside the if statement are implemented as the outputs of the flip flop. If the wait until statement is used, which has to be the first statement in the process then there will be the flip flops inferred for all statements in the process. i.e., any signal that is assigned a value in the entire process is implemented as the output of the flip flop

Asynchronous clear:

The device is cleared asynchronously.

Eg.

```
process(reset,clock)
begin
If reset = ‘0’ then
Q<=‘0’;
elsif clock’event and clock=‘1’ then
Q<=d;
end if;
end process;
```

synchronous clear:

The device is cleared synchronously.

Eg.

```
process
Begin
wait until clock’event and clock=‘1’ ;
if reset = ‘0’ then
Q<=‘0’;
 else
Q<=d;
end if;
end process;
```

The mode of a signal inside the entity can be *buffer* .

If the mode is *buffer* it is an output from the entity and its value can be used inside the entity. i.e., it can appear on both sides of assignment operator.

T FF

```
library ieee;
use ieee.std_logic_1164.all;
entity tff is
port(t,clock : in std_logic;
        q,qbar : buffer std_logic);
end tff;
architecture arch_tff of tff is
begin
process(clock)
variable temp : std_logic;
begin
  if clock’event and clock=‘1’ then
     temp:=q;
   if t=‘0’ then null;
   elsif t=‘1’ then q<=not(temp);
                              qbar <= temp;
end if;
end if;
end process;
end  arch_tff;
```

JK FF

```
library ieee;
use ieee.std_logic_1164.all;
entity jkff is
port(j,k,clock : in std_logic;
        q,qbar : buffer std_logic);
end jkff;
architecture arch_jkff of jkff is
begin
process(clock)
variable temp : std_logic;
begin
```

```
if clock’event and clock=‘1’ then
    temp:=q;
  if  j=‘0’ and k=‘0’ then q<=temp;
                         qbar <= not(temp);
  elsif  j=‘0’ and k=‘1’ then q<=‘0’;
                             qbar<=‘1’;
  elsif  j=‘1’ and k=‘0’ then q<=‘1’;
                             qbar<=‘0’;
  elsif  j=‘1’ and k=‘1’ then q<=not(temp);
                             qbar<=temp;
end if;
end if;
end process;
end  arch_jkff;
```