Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

APPENDIX

J

VHDL

J.1 Introduction to VHDL

Each VHDL description contains two blocks. These are input/output and architectural components. The input/output description specifies the input and output connections (ports) to the hardware. The architectural component defines the behavior of the hardware entity being designed. A typical VHDL description includes a `port` statement contained within an `entity` statement. All keywords in VHDL are reserved. This means that they cannot be used for any other purpose. A typical VHDL entity is given below:

```
entity  EXAMPLE is  --  Entity  Statement
port                --  port     Statement
 (X,Y,Z : in   BIT;
  W     : out  BIT);
end     EXAMPLE
```

The `entity` statement begins with the keyword `entity` followed by the name of the entity EXAMPLE followed by the word `is`. Note that all keywords in VHDL are case sensitive. The `port` statement is contained within an `entity` statement. The VHDL design entity is comprised of two parts: an interface and a body. The interface is specified by the keyword `entity` and the body is denoted by the keyword `architecture`. Typical logic and arithmetic operators along with port modes are listed below:

LOGIC OPERATORS

and	AND Operation
or	OR Operation
xor	Exclusive-OR Operation
xnor	Exclusive-NOR Operation
nand	NAND Operation
nor	NOR Operation
not	NOT Operation

ARITHMETIC OPERATORS

+	Positive sign or addition
–	Negative sign or subtraction
*	Multiplication
/	Division
mod	Modulus
rem	Remainder
abs	Absolute value
**	Exponential

TYPICAL PORT MODES

in	Information from the signal flows into the entity.
out	Information from the signal flows out of the entity, the value of the signal cannot be used inside the entity. Therefore, the value can appear on the left of the <= symbol.
inout	Information from the signal can flow into and out of the entity.
buffer	Information from the signal flows out of the entity; however, the signal can be used the entity. Therefore, the signal can appear on both sides of the <= symbol.

In the following, a simple VHDL programming example is provided. A comment is indicated by the symbol – – before a statement. A VHDL program for an Exclusive-NOR operation between two Boolean variables X and Y is provided below:

```
-- Exclusive-NOR Operation
entity XNOR   is
port(X,Y : in   BIT; Z  : out BIT);
end  XNOR;
-- Body
architecture BEHAVIOR of XNOR is
begin
Z<=X xnor Y;
end  BEHAVIOR;
```

In the above example, `architecture` declares the name XNOR to associate the architecture with the XNOR design entity interface. VHDL provides a library where the intermediate files about a particular design can be stored. These files can be used during analysis, synthesis and simulation of the design using IEEE standards. For example, the statement `library ieee;` can be used at the beginning of each program to specify the IEEE library. Also, IEEE developed the 1164 standard logic package to satisfy the requirements of most of the designers. The statement `library ieee;use.std_logic_1164.all;` written at the start of a VHDL program can use all the definitions of the IEEE standard 1164 logic package. Some more features of VHDL are discussed in the following.

For instance, in the architecture definition, `signal` declaration can be used for providing wire (internal connection) in a circuit. The `signal` declaration is similar to `port` declaration except that no modes (`in or out`) need to be specified. Predefined data types such as `bit` and `bit_vector` can be used with the `signal` declaration. `bit` data type can have values of 0 or 1 while `bit_vector` data type can be used to define a binary number. For example, the statement `signal c:bit_vector (3 downto 0);` defines bits 3 and 0 as the most significant bit and the least significant bit of a 4-bit number respectively. VHDL provides `wait` keyword which can be used in a test program to stop an operation for a specified period of time and then verify the outputs based on the predefined inputs.

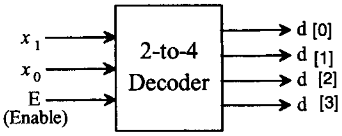
VHDL provides a `case` statement that executes one of several sequences of statements based on the value of a single expression. A simple example illustrating the use of the `case` statement is given below for a 2-to-1 multiplexer:

```
case sel is
when "0"=>
       z<=a;
when "1"=>
       z<=b;
endcase;
```

In the above, `sel` is used as the select input for the 2-to-1 multiplexer. When `sel`=0, output, z of the multiplexer is assigned with input, a. On the other hand, when `sel`=1, output, z will be assigned with input, b. As mentioned before, in order to design a system using HDL such as VHDL, two basic levels of abstractions or modeling are used. These are structural, modeling (used to describe a schematic or a logic diagram) and behavioral modeling (used to describe what the system does and how it behaves; uses both concurrent and sequential statements). Dataflow modeling is behavioral modeling with concurrent statements. Hierarchical structural model is used to decompose a large digital system into smaller blocks or modules. The three levels of abstractions (Structural, Dataflow, and Behavioral) are illustrated in the following by means of VHDL programs for the 2-to-4 decoder described in section 4.5.3.

J.1.1 Structural Modeling

The following VHDL structural description is provided for the 2-to-4 decoder of Figure 4.14. The figure is redrawn below for convenience:



```
library IEEE;
useIEEE.std_logic_1164.all;
entity decoder2to4 is
    port (x1,x0,E: in BIT;d: out BIT_VECTOR(0 to 3));
end decoder2to4;
architecture STRUCTURAL_DEC of decoder2to4 is
    component inv
        port (u: in BIT; v: out BIT);
    end component;
--VHDL code for inv
library IEEE;
useIEEE.std_logic_1164.all;
entity  inv is
    port (u: in BIT; v: out BIT);
end inv;
architecture LOGIC1 of  inv is
begin
    v<=not u;
end LOGIC1;

    component and3
         port (a, b, c: in BIT; f: out BIT);
    end component;
--VHDL code for and3
library IEEE;
useIEEE.std_logic_1164.all;
entity   and3   is
    port (a, b, c: in BIT; f: out BIT);
end and3;
architecture LOGIC of and3 is
begin
    f<= a and b and c;
end  LOGIC;
    signal x11, x00: BIT;
begin
    f0: inv port map (x1, x11);

    f1: inv port map (x0, x00);
    f2: and3 port map (E, x11, x00, d(0));
    f3: and3 port map (E, x11, x0, d(1));
    f4: and3 port map (E, x1, x00, d(2));
    f5: and3 port map (E, x1, x0, d(3));
end  STRUCTURAL_DEC;
```

As mentioned before, a VHDL program should include the statements :

```
library IEEE;
useIEEE.std_logic_1164.all;
```

The first statement provides access to the library called IEEE. This library contains the directory in the computer file system where the std_logic_1164 package is stored. The IEEE library files are plain text files that can be checked using any text editor. One can look at the IEEE library files after installing Altera Quartus II running under Microsoft Windows Operating System. The file that specifies the std_ logic type is called std 1164. vhd. Also note that VHDL is a strongly typed language unlike C. This means that VHDL compiler does not allow one to assign a value to a signal or a variable unless the type of the value exactly matches the declared type of the signal or variable. The VHDL compiler checks to see if data objects on both sides of assignment statements are identical. The VHDL compiler will not compile the program if there is a descrepency. For simplicity, all VHDL programs in this book will mostly use only the std_logic type. IEEE 1164 standard logic package defines many functions that operate on the standard data types such as std_ logic and std_logic_vector. Besides defining a number of user-defined data types, the IEEE 1164 package also defines the basic logic operations such as AND and OR on these data types . Because VHDL is a strongly typed language, it is often necessary to convert a signal from one type to another. IEEE 1164 package provides several conversion functions such as from bit to std_logic or vice versa. It should be mentioned that the IEEE 1164 does not include some of the common conversion functions such as from std_logic _vector to a corresponding integer value. However, the user can write such a conversion program. In the above example, all data objects for the inverter are defined as bits; this means that they can only have values of 0 or 1. In order to provide more flexibility, VHDL offers the data type called std_logic. Signals can have several different values when represented using this data type. In the above VHDL program, the statement (after component inv) port (u: in BIT; v: out: BIT); can be written as port (u: in std_logic; v: out std_logic); . The std_logic provides several data types including 0, 1, Z (High impedance state), and - (don't care condition).

Three types of data objects are used to represent information in VHDL programs. These are signals, constants, and variables. Signals are very common in logic circuits since they provide wires (connections) in the circuit. Constants and variables are also used in logic circuits. Furthermore, in order to implement arithmetic operators for signed and unsigned numbers,a package called std_logic_arith along with std_logic_signed (for signed numbers) and std_logic_unsigned (for unsigned numbers) can be used.

The entity called decoder2to4 in the above VHDL program contains three input ports and four output ports. E, x1, and x0 are defined as inputs with widths of one bit each while the output , d is defined as a vector with an array size of four bits. In this example, the name of the architecture body is STRUCTURE_DEC. There are two component declarations (inv, and3), and one signal declaration. The signal declaration declares two signals of type BIT named, x11 and x00. These signals represent wires that are used to connect the various components of the decoder. Note that the statements inside a component are concurrent. Therefore, these statements can be written in any order within a component. The Structural model considers the components as black boxes for only interconnecting them without taking behavior of components into consideration. In the architecture body of STRUCTURAL_DEC, signals x1, x0, and E are declared as input ports in the decoder2to4 entity declaration. Next, consider the statement labeled f5. In f5, port E is connected to input a of component and3, port x1 is connected to input b of component and3, port x0 is connected to input c of component and3, and port d(3) of the decoder2to4 entity is connected to the output port f of component and3. Note that separate entity along with architecture and appropriate declarations are included for components inv and and3.

The component statement is used to describe the Structural model of an entity. Two component names are used in the above program. These are inv and and3. The component name is the name of a defined entity to be used in the current architecture body. Each component is declared with port declarations. The component declaration is included in the declaration part of an architecture declaration. The keyword `port map` defines a list that associates ports of the named entity with signals in the current architecture. A component instantiation statement associates the signals in the entity with the ports. There are two ways to represent the association. These are positional association and named association. In positional association, each signal in the port map is mapped by position with each port in the component declaration. This means that the first port in the component declaration corresponds to the first signal in the component instantiation, the second port with the second signal, and so on. For example, consider the following component instantiation statement in the above program f0: inv port map (x1, x11); in which f0 is the component label for the current instantiation of the inv component. Signal x1 is associated with port u of the inv component and signal x11 receives the output value (inverted x1 in this case) from the component. The ordering of signals must be done properly.

In the named association, each of the entity's ports is connected using the operator <= or => and the order of listing is unimportant. The named association is illustrated by a two-input OR gate example provided below.

```
entity comb  is
    port (a, b: in BIT; c: out BIT);
end comb;
architecture structural of comb is
  component  OR2
    port (x, y: in BIT; z: out BIT);
  end component;
  signal s1: BIT;
begin
  g1: OR2 port map(x=>a, y=>s1, z=>c);
end structural;
entity OR2 is
    port (x, y: in BIT; z: out BIT);
end OR2;
architecture LOGIC of OR2  is
begin
    z<= x or y;
end LOGIC;
```

In the above, signal a (declared in the entity port list) is associated with x declared in the component port list, signal c is associated with z, and signal s1 is associated with y. In this named association, the ordering of the associations is not required.

J.1.2 Behavioral Modeling

The behavioral model contains statements that are executed sequentially in a predefined order. These sequential statements are defined using a `process` statement inside an architecture body. A VHDL program for a 2-to-4 decoder using Behavioral modeling is given in the following:

```
library IEEE;
useIEEE.std_logic_1164.all;
entity  decoder2to4 is
port (x1, x0, E: in BIT; d: out BIT_VECTOR (0 to 3));
end  decoder2to4;
architecture  BEHAVIOR_DEC of decoder2to4  is
        begin
        process  (x1, x0, E)
variable x11, x00:BIT;
           begin
x11:= not x1;
x00:= not x0;
        if E = '1' then
                d(0)<= x11 and x00;
                d(1)<= x11 and x0;
                d(2)<= x1 and x00;
                d(3)<= x1 and x0;
        else
                d<="0000";
        end if;
        end process;
end BEHAVIOR_DEC;
```

In the above, two variables x11 and x00 are declared using the keyword `variable`. A variable is always assigned with a value instantaneously using the assignment operator :=. A signal, on the other hand, is assigned with a value always after a certain delay using the assignment operator <=. Signal and variable assignment statements in a process are executed sequentially regardless of whether or not any event occurs on the right hand side of the expression. The general form of `process` statement is given below:

process (sensitivitylist)
process declarations
begin
list of sequential statements such as signal assignments, variable assignments, and if statements
end process;

The sensitivitylist includes signals to which the `process` is sensitive. The `process` will be executed as soon as any changes in the values of these signals occur. As mentioned before, variables and constants inside a `process` must be defined in the process declarations part before the keyword `begin`. The statements that follow after the keyword `begin` are executed sequentially. Variable assignments inside a `process` are denoted by the := operator, and are executed immediately. This is in contrast to signal assignment denoted by the operator <= in which changes occur after a delay. Therefore, variables will be available immediately to all subsequent statements within the same `process`. In the above program, if-else construct is used. The general form of if-else construct is as follows:

if condition then
sequential statements
elseif condition then
sequential statements
else
sequential statements
end if;

The if statement is executed by checking each condition (Boolean expression) in the order they are written in the program until a true condition is found. In the above program, E=1 is the true condition. If an event occurs on any signal E, x1, or x0, variable assignment statements are executed. When the if statement is executed , and if E=1, then four signal assignment statements are executed. On the other hand, if E=0, the four-bit vector, d receives the four-bit value 0000. When end of process is reached, the process halts itself and waits for another event to occur on a signal in the sensitivity list.

J.1.3 Dataflow Modeling

As mentioned before, dataflow modeling is a form of behavioral modeling. A VHDL program for the 2-to-4 decoder using dataflow modeling is provided in the following:

```
library IEEE;
useIEEE.std_logic_1164.all;
entity decoder2to4  is
    port (x1, x0, E: in BIT; out BIT_VECTOR (0 to 3));
end decoder2to4;
architecture DATAFLOW_DEC of  decoder2to4  is
  signal x11, x00: BIT;
begin
  x11 <= not x1;
  x00 <= not x0;
  d(0)<=E and x11 and x00;
  d(1)<=E and x11 and x0;
  d(2)<=E and x1 and x00;
  d(3)<=E and x1 and x0;
end DATAFLOW_DEC;
```

Note that VHDL programs written using dataflow modeling contain assignment statements. These statements are executed if one of the values on the right hand side of the assignment statement changes. The architecture body contains one signal declaration and six concurrent signal assignment statements. Note that concurrent signal assignment statements are concurrent statements, and hence, the ordering of these statements in the architecture body is unimportant. The signal declaration declares x11 and x00 to be used with the architecture body. Since no `after` clause is used for defining delays for each signal assignment statement, a default delay of 0ns is assumed. This delay of 0ns is called delta time and is denoted by a very small time delay. Now, Suppose that input signal, x0 in the above program changes. This will affect the signal assignment statements for x00, d(1), and d(3). Therefore, the right hand sides of these expressions will be evaluated , and the corresponding values of x00, d(1), and d(3) will be assigned after certain time delay (for example, t) during simulation. Since the value of x00 is affected due to changes in x0, this, in turn, will affect the values of d(0) and d(2). Therefore, new values will also be calculated for d(0) and d(2) after further time delays (for example, t+nt). The meaning of this concurrent behavior shows that the simulation is event-triggered. Hence, the simulation time proceeds to the next time unit when an event occurs. In the above program, the library and entity statements are same as before. Signal declarations are made for x11and x00. Signals x11 and x00 are obtained by applying logical not operations on x1 and x0 respectively. d(0), d(1), d(2), and d(3) are then obtained by performing logical and operations on E, x1, x0, x11, x00 as defined by the Boolean equations of the 2-to-4 decoder.

There are two other ways of writing VHDL programs with dataflow modeling. These are called conditional dataflow modeling, and are obtained by using when-else and with-select constructs. The following VHDL program is written for the 2-to-4 decoder using when-else construct:

```
library IEEE;
useIEEE.std_logic_1164.all;
entity decoder is
    port (x: in bit_vector(1 downto 0);
            E:in bit;
            d: out bit_vector(3 downto 0));
end decoder;
architecture  when_else of decoder is
signal Ex: in bit_vector(2 downto 0);
begin
Ex<= E & x;
d<= "0001" when Ex = "100" else
    "0010" when Ex = "101" else
    "0100" when Ex = "110" else
    "1000" when Ex = "111" else
    "0000"
end architecture when_else;
```

The truth table for the above decoder is given in table 4.8. The inputs in this table are shown in the order E x1 x0. In the above program, these three signals are represented as a three-bit signal called Ex. In order to express Ex, the VHDL concatenate operator & is used in the expression Ex<= E & x;. Thus, E and x are combined into Ex signal where Ex(2) = E, Ex(1) = x1, and Ex (0) = x0. Ex is used as a condition in the above when-else construct. This when-else conditional assignment is used to assign a signal value with one of several choices. The syntax is as follows:

signalname<= expression when Boolean_condition else
expression when Boolean_condition else
expression when Boolean_condition else
....
....
expression

The signalname will have the value of the first expression whose Boolean condition is true. If more than one condition is true, the signalname will be assigned with the value associated with the first true condition. If no true condition is found, the signalname will be assigned with the final expression. For example, if E=1, x1 = 0, x0 = 1, then Ex = 101. This means that the four-bit vector d will be assigned with the value 0010; hence, d3=0, d2=0, d1=1, and d0=0. However, if Ex = 011, then the four-bit vector, d will be assigned with the value 0000.

The following VHDL program is written for 2-to-4 decoder using with-select construct:

```
library IEEE;
useIEEE.std_logic_1164.all;
entity decoder is
port (x: in bit_vector(1 downto 0);
      E:in bit;
      d: out bit_vector(3 downto 0));
end decoder;

architecture  with_select of decoder is
signal Ex: in bit_vector(2 downto 0);
begin
Ex<= E & x;
      with Ex select
```

```
    d<= "0001" when "100",
        "0010" when "101",
        "0100" when "110",
        "1000" when "111",
        "0000" when others;
end architecture with_select;
```

The syntax for with-select construct is given below:
with choice_input select
signalvalue <= expression when value,
                  expression when value,
                  expression when value,
                  ....
                  expression when others;

In the above, choice_input (the value of choice_input is to be used for decision) is placed between with and select. When choice_input equals value, the expression associated with the value is assigned to signalvalue. For example, consider E=1, x1=1, and x0=0. This means that Ex=110. Hence, 0100 is assigned to the four-bit vector, d. Therefore, d3=0, d2=1, d1=0, and d0=0. All other values not listed are represented by the word, others. Hence, if Ex = 011, then d will be assigned with the value 0000.

J.1.4 Mixed Modeling

In the following, an example is provided in which all three levels of modeling (Structural, Dataflow, and Behavioral) are used. This is called mixed modeling. The full adder is used for this purpose. The equations for the full adder can be written as follows:

$S = w \oplus z$, where $w = x \oplus y$

$C = xy + yz + xz$

The following VHDL program implements the above equations as follows:

$w = x \oplus y$ (Structural), $S = S = w \oplus z$ (Dataflow), $C = xy + yz + xz$ (Behavioral)

```
--VHDL program for Full Adder using mixed modeling
library IEEE;
useIEEE.std_logic_1164.all;
entity  FA is
    port (x,y,z: in BIT; S, C: out BIT);
end FA;
-- Structural
architecture MIXED of FA is
  component XOR0
    port (a,b: in BIT; c: out BIT);
  end component;
  signal w:BIT;
begin
  g: XOR0 port map (x,y,w);
--Behavioral

  process (x,y,w)
    variable f1, f2, f3: BIT;
begin
    f1:=x and y;
    f2:=y and z;
    f3:=x and z;
    C<=f1 or  f2 or f3;
  end process;
--dataflow
    S<=w xor z;
end MIXED;
```

```
--VHDL code for XOR0
entity XOR0 is
  port (m, n: in BIT; v: out BIT);
end XOR0;
architecture LOGIC of XOR0 is
begin
  v<=m xor n;
end LOGIC;
```

J.2 VHDL descriptions of typical combinational logic circuits

EXAMPLE J.1

Write a VHDL description for $f = A + B\overline{C}$ (Section 3.6) using dataflow modeling.

Solution

The program written using Dataflow modeling as follows.

Program:

```
-- file name: FUNC.vhd

library ieee;
use ieee.std_logic_1164.all;
entity FUNC is
        port(a,b,c:in std_logic;
                f:out std_logic);
end FUNC;

architecture FUNC_arch of FUNC is
        signal y0,y1: std_logic;
begin
        y0 <= not c;
        y1 <= b and y0;
        f <= y1 or a;
end FUNC_arch;
```

EXAMPLE J.2

Write a VHDL description for a two-input Exclusive-OR gate using dataflow modeling.

Solution

This program is written using dataflow modeling as follows:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY xor_bit IS
        PORT (a,b: IN bit; y: OUT bit);
END xor_bit;
ARCHITECTURE behave OF xor_bit IS
BEGIN
        y <= a XOR b;
END behave;
```

EXAMPLE J.3

Write a VHDL description using dataflow modeling for the 2-to-1 multiplexer of figure 4.21 using dataflow modeling.

Solution

```
-- 2 to 1 MUX
-- file name: MUX2.vhd
library ieee;
use ieee.std_logic_1164.all;
entity MUX2 is
       port(a,b,sel:in std_logic;
                 cout:out std_logic);
end MUX2;
architecture MUX_arch of MUX2 is
signal y0,y1,y2: std_logic;
begin
       y0 <= not sel;
       y1 <= a and y0;
       y2 <= b and sel;
    cout <= y1 or y2;
end MUX_arch;
```

EXAMPLE J.4

Write a VHDL description using dataflow modeling for a 4-bit binary adder.

Solution

```
-- 4 bit binary adder
-- file name: adder4.vhd
library ieee;
use ieee.std_logic_1164.all;
entity adder4 is
       port(a,b:in bit_vector(3 downto 0);
              cin:in bit;
              cout:out bit;
              s:out bit_vector(3 downto 0));
end adder4;
architecture adder_arch of adder4 is
       signal c:bit_vector(3 downto 1);
begin
       s(0)<=a(0) xor b(0) xor cin;
       c(1)<=(a(0) and b(0)) or (a(0) and cin) or (b(0) and cin);
       s(1)<=a(1) xor b(1) xor c(1);
       c(2)<=(a(1) and b(1)) or (a(1) and c(1)) or (b(1) and c(1));
       s(2)<=a(2) xor b(2) xor c(2);
       c(3)<=(a(2) and b(2)) or (a(2) and c(2)) or (b(2) and c(2));
       s(3)<=a(3) xor b(3) xor c(3);
       cout<=(a(3) and b(3)) or (a(3) and c(3)) or (b(3) and c(3));
end adder_arch;
```

EXAMPLE J.5

Write a VHDL description using hierarchical modeling for a 4-bit binary adder.

Solution

VHDL (Using Hierarchical)

```
--One full adder program
library ieee;
use ieee.std_logic_1164.all;
-- full-adder
--Define outputs and inputs
entity full_adder is
  port (a, b, cin: in std_logic;
        sum, carry: out std_logic);
end full_adder;
--Use Boolean equations
architecture eqns of full_adder is
begin
```

```
  sum <= a xor b xor cin;
  carry <= (a and b) or (cin and (a xor b));
end eqns;

--4 bit full adder using the full adder program
-- 4-bit full adder using hierarchical logic
library ieee;
use ieee.std_logic_1164.all;

-- module interface
entity hier_full_adder is
  port ( a, b  : in std_logic_vector(3 downto 0);
         cin   : in std_logic  ;
         sum   : out std_logic_vector(3 downto 0);
         carry : out std_logic) ;
end hier_full_adder;
-- module hierarchical
architecture structural of hier_full_adder is
  component full_adder
    port (a, b, cin: in std_logic; sum, carry: out std_logic);
  end component;
  signal c0, c1, c2: std_logic;
begin
  fa0: full_adder port map (a(0), b(0), cin, sum(0), c0);
  fa1: full_adder port map (a(1), b(1), c0, sum(1), c1);
  fa2: full_adder port map (a(2), b(2), c1, sum(2), c2);
  fa3: full_adder port map (a(3), b(3), c2, sum(3), carry);
end structural;
```

EXAMPLE J.6

Write a VHDL description for a full-adder using 74138 decoder and gates (Figure 4.17).

Solution

The 74138 decoder is implemented using conditional dataflow. The Full-adder is implemented using structural modeling. The VHDL program is provided below:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY Dec3to8 IS
       PORT (  A                             :  in      STD_LOGIC_VECTOR(2
DOWNTO 0);
                 G1, NOT_G2A, NOT_G2B : in   STD_LOGIC ;
                 D                            :  out     STD_LOGIC_VECTOR(7
DOWNTO 0) );
END Dec3to8;
ARCHITECTURE Behavior OF Dec3to8 IS
SIGNAL Sel : std_logic_vector ( 5 downto 0);
BEGIN
               Sel <= ((NOT_G2A & NOT_G2B) & G1) & A;
               WITH Sel SELECT
                    D <=    “11111110” WHEN “001000”,
                            “11111101” WHEN “001001”,
                            “11111011” WHEN “001010”,
                            “11110111” WHEN “001011”,
                            “11101111” WHEN “001100”,
                            “11011111” WHEN “001101”,
                            “10111111” WHEN “001110”,
                            “01111111” WHEN “001111”,
                            “11111111” WHEN OTHERS;

END Behavior;
```

```
--   IMPLEMENTATION OF A FULL ADDER USING 74138 & 4-INPUT AND GATES
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY Full_Adder IS
       PORT (  X       : in    STD_LOGIC_VECTOR (2 DOWNTO 0);
               S, C    : out    STD_LOGIC );
END Full_Adder;
ARCHITECTURE Structural OF Full_Adder IS
SIGNAL g1, g2, g3 :  std_logic;
SIGNAL M : STD_LOGIC_VECTOR (7 DOWNTO 0);
COMPONENT Dec3to8
       PORT (  A                          :  in    STD_LOGIC_VECTOR(2
DOWNTO 0);
               G1, NOT_G2A, NOT_G2B: in  STD_LOGIC ;
               D                          :  out  STD_LOGIC_VECTOR(7
DOWNTO 0) );
END COMPONENT;
BEGIN
       g1 <= ‘1’;
       g2 <= ‘0’;
       g3 <= ‘0’;
       Dec: Dec3to8 port map ( X, g1, g2, g3, M);
       S <=  (M(0) and M(3) and M(5) and M(6));
       C <=  (M(0) and M(1) and M(2) and M(4));
END Structural;
```

J.3 VHDL descriptions of typical synchronous sequential circuits

VHDL keyword `process`, described in section J.1.2 for behavioral modeling, is used to describe sequential circuits. Furthermore, state machines are normally modeled using a `case` statement in a `process`. Since the `case` statement provides multiple branching, the behavior of a state in a state machine is represented using `case` statement. Also, the statement `clock'event and clock='1';` is used to obtain positive clock. This is because the syntax clock’event uses a VHDL attribute. An attribute basically implies the property of an object such as signal. The attribute ‘event means a change in the clock signal. By logically anding clock’event with clock=1 will indicate that the clock signal has just changed and the value of the clock signal is 1. This means a positive clock edge.

EXAMPLE J.7

Write a VHDL description for a D flip-flop using Behavioral modeling.

Solution

```
-- D Flip-Flop (Behaviorally)
-- Module DFF with synchronous reset
-- file name: dfflop.vhd

library ieee;
use ieee.std_logic_1164.all;
entity dfflop is


      port(d, clk, reset: in std_logic;
                          q: out std_logic);
end dfflop;

architecture dfflop_arch of dfflop is
```

```
begin
        process (clk, reset) is
        begin
                if reset = '1' then
                        q <= '0' ;
                elsif clk'event and clk = '1' then
                        q <= d ;
                end if ;
        end process;
end dflop_arch;
```

Tabular form of simulation:

```
INPUTS reset d clk ;
OUTPUTS q ;
PATTERN
%         r         %
%         e         %
%         s   c     %

%         e   l     %
%         t d k   q %
    0.0> 0 0 0 = 0
 1000.0> 0 0 1 = 0 2000.0> 0 1 0 = 0  2006.5> 0 1 0 = 1
 3000.0> 0 1 1 = 1  4000.0> 1 0 0 = 1  4006.5> 1 0 0 = 0
 5000.0> 1 0 1 = 0  6000.0> 1 1 0 = 0  7000.0> 1 1 1 = 0
 8000.0> 0 0 0 = 0  9000.0> 0 0 1 = 0 10000.0> X X X = X
```

EXAMPLE J.8

Write a VHDL description for a T flip-flop using behavioral modeling.

Solution

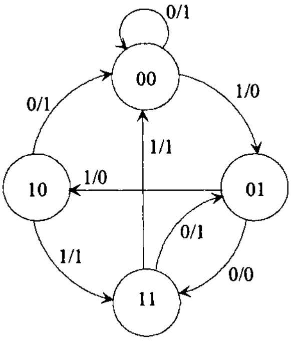
Implementation of T Flip Flop using Behavioral method:

```
--****************************************************************


--   T FLIPFLOP IMPLEMENTATION.

--****************************************************************
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY tff IS
    PORT ( T ,preset, reset, Clock        : IN          STD_LOGIC ;
                   q ,qnot                : buffer      STD_LOGIC) ;
END tff ;

ARCHITECTURE Behavior OF tff IS

            SIGNAL temp :STD_LOGIC;
            begin
            PROCESS (preset, reset, Clock )
                    BEGIN
                    IF reset = '0' THEN
                    temp <= '0' ;
```

```
               ELSIF preset = '0' then
               temp <= '1';
               ELSIF Clock'EVENT AND Clock = '1' THEN
               temp <= T xor temp ;
               END IF ;
        END PROCESS ;

               q <= temp;
               qnot <= not temp;
END Behavior;
```

EXAMPLE J.9

Write a VHDL description of the state machine of figure 5.21of Example 5.2
(a) using mixed modeling (dataflow and behavioral) (b) using behavioral modeling with `case` statement. Figure 5.21 is redrawn below:



Solution

(a)

The following equations are obtained in Example 5.2:

$D_X = X\overline{Y}A + \overline{X}Y \qquad D_Y = \overline{Y}A + Y\overline{A} = Y \oplus A \qquad Z = \overline{Y}\,\overline{A} + X$

```
These equations are used to write the following program.
-- Example 5.2: Sequential circuit

-- file name: ex52_seq1.vhd
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.std_logic_arith.all;

entity ex52_seq1 is
        port    (clk, a, reset: in std_logic; -- inputs for example 5.2
           z,x_out,y_out: out std_logic);   -- output for example 5.2
end ex52_seq1;

architecture dfflop_arch of ex52_seq1 is
        signal data_d1, data_d2, x, y :std_logic;
        signal x1,y1: std_logic;
begin
        data_d1 <= (( x and (not y) and a ) or ( (not x) and y ));
        data_d2 <= ( y xor a );
        dff1: process (clk)
```

```
   begin
       if (reset='1') then
              x<= '0';
              y<= '0';
       end if;
       begin
              if (clk'event and clk= '1') then
                     x <= data_d1;
                     y <= data_d2;
              end if;
       end process dff1;
       z <= x or (    (not y) and (not a));
       x_out <= x;
       y_out <= y;
end dfflop_arch;
```

(b) Behavioral Modeling using `case` statement:

VHDL PROGRAM:

```
    --*******************************************************************
    **
    --  IMPLEMENTATION OF SYNCHRONOUS SEQUENTIAL     *
    --                                 CIRCUIT (Example 5.2)
    *
    --*******************************************************************
    **
    LIBRARY ieee ;
    USE ieee.std_logic_1164.all ;
    ENTITY Mealy IS
        PORT (  x, reset, clock          : IN    STD_LOGIC ;
                                       z  : OUT STD_LOGIC ) ;
    END Mealy ;

    ARCHITECTURE M OF Mealy IS
        type state_type is (S0, S1, S2, S3);
        signal Yn  : state_type;
        begin

    --         State Transition       AND     Next State Calculation

    process (clock, reset)
               begin
                      if reset = '0' then
                             Yn <= S0;
                      elsif clock'event and clock = '1' then

                        case Yn is
                             when S0 =>     if x = '0' then Yn <= S0;
                                                    else Yn <= S1;
                                                    end if;
                             when S1 =>     if x = '0' then Yn <= S3;
                                                    else Yn <= S2;
                                                    end if;
                             when S2 =>     if x = '0' then Yn <= S0;
                                                    else Yn <= S3;
                                                    end if;
                             when S3 =>     if x = '0' then Yn <= S1;
                                                    else Yn <= S0;
                                             end if;
```

```
            end case;
           end if;
    end process;

--         Output Calculation

    process (x, Yn)
    begin
           case Yn is
                  when S0 => if x = '0' then z <= '1';
                                    else z <= '0';
                                    end if;
                  when S1 => z <= '0';
                  when S2 => z <= '1';
                  when S3 => z <= '1';
           end case;
    end process;
end M;
```

Note:

In the above VHDL program, the state table of the machine is defined using a `case` statement. Each `when` construct corresponds to a present state of the machine, and the `if` statement inside the `when` construct defines the next state at the positive edge of the clock Note that in VHDL clock'event and clock='1' means positive edge of the clock.. In the above, a `type` declaration is used for the signal Yn. The `type` declaration allows one to specify new types analogous to existing types such as `std_logic`. A `type` declaration starts with the keyword `type` followed by the name of the new type, the keyword `is`, and the list of the values of the signals of the new type in parentheses. The signal named Yn represents the state of the machine. It is defined as state_type with four possibilities S0, S1, S2, and S3. When the VHDL program is compiled, the compiler automatically performs a state assignment to select appropriate bit patterns for the four states. The behavior of the Mealy machine is defined by the inputs reset, clock, and input, x. The program contains an asynchronous reset input that places the machine in state S0. Consider the last four `when` statements between `case Yn is` and `end case`. The first statement means that when Yn=S0 (state 0), if input x=0 then output z=1. When Yn=S1(state 1), output z=0 for either input x=0 or 1; when Yn=S2 (state 2), output z=1 for either input x=0 or 1; when Yn=S3 (state 3), output z=1 for either input x=0 or 1. These transitions agree with the state diagram of figure 5.21.

EXAMPLE J.10

Write a VHDL description for the two-bit counter of Example 5.5 to count in the sequence 0, 1, 2, 3, and repeat. Use T flip-flops.

Solution

BEHAVIORAL METHOD:

```
--*****************************************************************
*
--. IMPLEMENTATION OF COUNTER

-- (Example 5-5)

--*****************************************************************
```

```
*
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all;

ENTITY Counter_2IN IS
PORT (      EN, reset, clock      : IN      STD_LOGIC ;
                       count      : OUT STD_LOGIC_VECTOR (1 DOWNTO 0) ) ;
END Counter_2IN ;

ARCHITECTURE M OF Counter_2IN IS .
    signal count_up  : std_logic_vector (1 downto 0);
    begin
    process (clock, reset)
            begin
                    if reset = '0' then
                            count_up <= (others => '0');
                    elsif clock'event and clock = '1' then
                            if EN ='1' then
                                    count_up <= count_up + 1;
                    end if;
            end if;
    end process;
                     count <= count_up;
end M;
```

Note: In the above, the statement count_up <= (others => '0'); is equivalent to count_up <="00" since count_up is declared as a two-bit vector earlier in the code. The (others=>'0') syntax will assign a '0' digit to each bit of count_up regardless of the size of count_up. Therefore, the above VHDL code can be used for any size of count_up rather than only for the two-bit count_up.

EXAMPLE J.11

Write a VHDL description for the three-bit counter of Example 5.7.

Solution

```
--      AND -T FLIP FLOP:

--****************************************************************
*
--   AND_T FLIPFLOP IMPLEMENTATION
--   (Example 5-7)
--****************************************************************
*

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY AND_tff IS
    PORT ( x0, x1, Clock   : IN    STD_LOGIC ;
                    q             : out STD_LOGIC) ;
END AND_tff ;

ARCHITECTURE Behavior OF AND_tff IS
signal T, temp : std_logic;
```

```
   BEGIN
          T <= x0 and x1;
          PROCESS
                 BEGIN
                 wait until Clock'EVENT AND Clock = '1';
                        temp <= T xor temp;//temp is 0 or 1

          END PROCESS;
                        q <= temp;
   END Behavior;
```

--OR-T FLIP FLOP:

```
--*****************************************************************
*
--   OR_T FLIPFLOP IMPLEMENTATION
--  (Example 5-7)

--*****************************************************************
*
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY OR_tff IS

PORT (x0, x1,     Clock   : IN   STD_LOGIC ;
                  q              : out STD_LOGIC) ;
END OR_tff ;

ARCHITECTURE Behavior OF OR_tff IS
signal T, temp : std_logic;
   BEGIN
          T <= x0 or x1;
          PROCESS
                 BEGIN
                 wait until Clock'EVENT AND Clock = '1';
                        temp <= T xor temp;
       END PROCESS;
                        q <= temp;
   END Behavior;
```

--AND-OR-T FLIP FLOP:

```
--*****************************************************************
*
--   AND_OR_T FLIPFLOP IMPLEMENTATION
--  (Example 5-7)

--*****************************************************************
*
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY AND_OR_tff IS
   PORT (x0, x1, x2, Clock             : IN   STD_LOGIC ;
                   q                      : OUT STD_LOGIC) ;
END AND_OR_tff;

ARCHITECTURE Behavior OF AND_OR_tff IS
signal T, temp : std_logic;
    BEGIN
           T <= (x0 and x1) or x2;
           PROCESS
                   BEGIN
                   wait until Clock’EVENT AND Clock = ‘1’;
                          temp <= T xor temp;
        END PROCESS;
                   q <= temp;
    END Behavior;
```

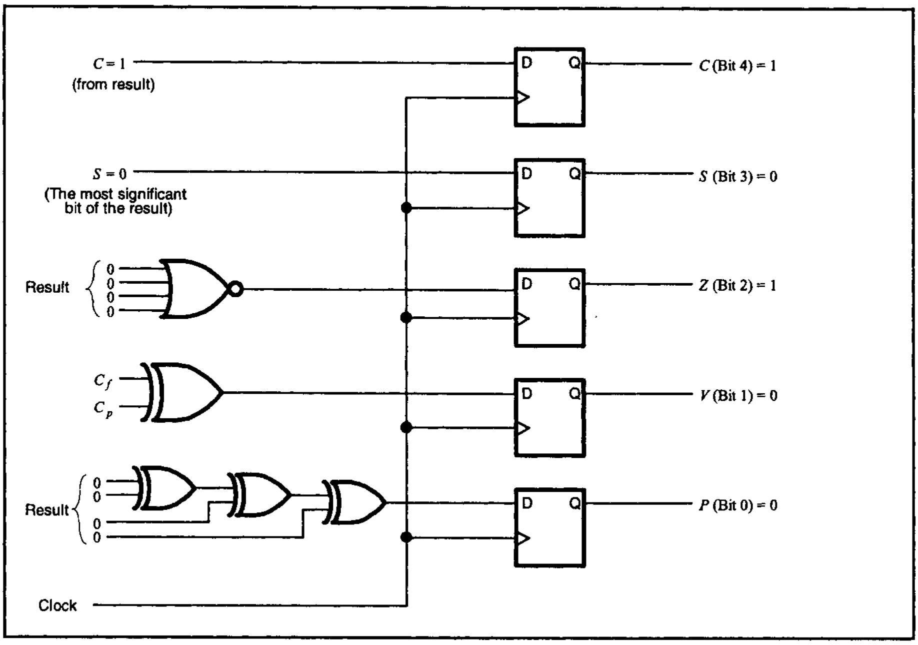
--THE MAIN PROGRAM OF NONBINARY COUNTER:

```
--**********************************************************************
*
--   NON BINARY COUNTER IMPLEMENTATION
--   (Example 5-7)

--**********************************************************************
*

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Non_Binary_Count IS
    PORT ( CLK              : in  std_logic;
           A                : buffer std_logic_vector ( 2 downto 0) );
END Non_Binary_Count;

ARCHITECTURE Structure OF Non_Binary_Count IS
signal t : std_logic_vector(2 downto 0);
COMPONENT AND_tff
    PORT (  x0, x1,Clock: IN      STD_LOGIC ;
                   q              : OUT STD_LOGIC) ;
END COMPONENT;
COMPONENT AND_OR_tff
    PORT (x0, x1, x2,Clock : IN   STD_LOGIC;
                   q              : OUT STD_LOGIC) ;
END COMPONENT;
COMPONENT OR_tff
    PORT (x0, x1,  Clock   : IN   STD_LOGIC ;
                   q              : OUT STD_LOGIC) ;
END COMPONENT;
    Begin
            t(0) <= not A(0);
            t(1) <= not A(1);
            t(2) <= not A(2);

            Tf0:    AND_tff port map ( A(0), A(1), CLK, A(2));
            Tf1:    OR_tff port map ( t(1), A(0), CLK, A(1));
            Tf2:    AND_OR_tff port map ( t(0), A(1), A(2), CLK, A(0));

END Structure;
```

Note: In the above VHDL code, `wait until` is used with the clock. This statement has the same effect as the `if` statement previously used with the clock. The sensitivity list is omitted from the process since `wait until` construct is used. The `wait until` construct means that the sensitivity list automatically contains only the clock signal.

J.4 Status register design using VHDL

In this section, the VHDL description of the Status register of Example 6.1 will be provided. The VHDL program for the Status register is written using structural modeling. Schematic for the Status register is redrawn below.



The VHDL description for the D flip-flop (required by the Status register program) is written using behavioral modeling.

EXAMPLE J.12

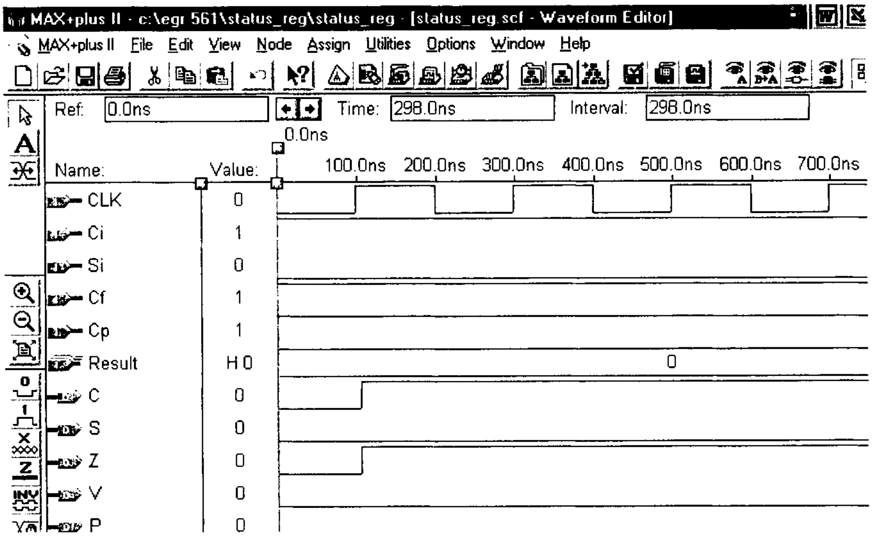
Write a VHDL description of the Status register of Example 6.1.

Solution

```
LIBRARY IEEE:
USE IEEE.STD LOGIC 1164.ALL;
ENTITY Status Reg IS
PORT (Ci, Si, Cf, Cp, CLK: in std logic;
                    Result: in std logic vector (3 downto 0);
             C, S, Z, V, P: buffer std logic);
end Status Reg;
ARCHITECTURE Structure OF Status Reg IS
COMPONENT DFF
       PORT ( D, CLK: in std logic; Q: buffer std logic);
END COMPONENT
SIGNAL m, n, r : std logic;
BEGIN
```

```
        m <= not ( Result (0) or Result (1) or Result (2) or Result (3));
        n <= Cf xor Cp;
        r <= (( Result(0) xor Result (1)) xor Result (2)) xor Result (3);
        D1: DFF PORT MAP (Ci, CLK, C);
        D2: DFF PORT MAP (Si, CLK, S);
        D3: DFF PORT MAP (m, CLK, Z);
        D4: DFF PORT MAP (n, CLK, V);
        D5: DFF PORT MAP (r, CLK, P);
END  Structure;
LIBRARY IEEE:
USE IEEE.STD LOGIC 1164.ALL;
ENTITY DFF IS
PORT ( D, CLK : in std logic; Q : buffer std logic);
end DFF;

ARCHITECTURE Behavior  OF  DFF  IS
begin
        process
                begin
                        wait until CLK'EVENT AND CLK = "1" ;
                                Q <= D ;
        end process;
end Behavior;
```

Waveform:


MAX+plus II - c:\egr 561\status_reg\status_reg - [status_reg.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 0.0ns
Time: 298.0ns
Interval: 298.0ns
0.0ns
100.0ns 200.0ns 300.0ns 400.0ns 500.0ns 600.0ns 700.0ns
Name:
Value:
CLK 0
Ci 1
Si 0
Cf 1
Cp 1
Result H 0
0
C 0
S 0
Z 0
V 0
P 0

After the clock is set to one, the outputs are generated. From the waveform, it can be verified that Ci = 1, Si = 0, Cf = Cp =1, and result = 0000. That gets the output C = 1, S = 0, Z = 1, V = 0, P = 0.

J.5 CPU design using VHDL

In writing VHDL description for the CPU in Example 7.5, some of the VHDL statements and keywords such as `generate` , `generic` , `generic map`, type-conversion functions, and `constant` are used. Therefore, these will be discussed below. The `generate` statement can be used in applications where it is necessary to create multiple copies of a particular structure within an architecture. For example, an n-bit ripple carry adder can be obtained by connecting n full-adders. The `generate` statement in VHDL can be used to create such repetitive structures. There are two types of `generate`. These are `for generate` and `if generate`. The `for generate` allows concurrent statements to be selected a predetermined number of times. The general form of `for generate` loop is given below:

```
label_name : for  k in  1 to n generate
                         concurrent statements
                        end generate;
```

In the above, the identifier k must be declared as the same type as the range 1 to n (integer in this case). The concurrent statements are executed once for each possible value of the identifier within the range.

The `if generate`, on the other hand, allows concurrent statements to be conditionally selected based on the value of an expression. The general form of `if generate` is given below:

```
label_name : if  k=n generate
                         concurrent statements
                        end generate;
```

In order to illustrate the applications of `for generate` and `if generate` statements, consider VHDL code for a 4-to16 decoder using five 2-to-4 decoders of figure 4.16 as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity 4to16dec is
  port (x:in std_logic_vector (3 downto 0);
         e:in std_logic;
        d: out std_logic_vector (0 to 15));
end 4to16dec;
architecture decoder of 4to16dec is
 component 2to4dec
   port (x:in std_logic_vector (1 downto 0);
         e:in std_logic;
           d: out std_logic_vector (0 to 3);
 end component;
 signal k: std_logic_vector (0 to 3));
begin
 f1: for i in 0 to 3 generate
  dec_1: 2to4dec port map(x(1 downto 0), k(i), d(4*i to 4*i+3));
 f2: if i=3 generate
  dec_2: 2to4dec port map (x(i downto i-1), e, k);
  end generate;
end decoder;
```

In the above, after the `component` declaration, signal k is defined as the outputs of the left 2-to-4 decoder of figure 4.16. Also, in figure 4.16, the outputs are instantiated by the `for generate` statement. For each iteration, the statement with label `dec_1` instantiates a 2-to-4 decoder component that corresponds to one of the four 2-to-4 decoders on the right side of figure 4.16. The first iteration produces `2to4dec component` with inputs x1 and x0, enable input k0 and, generates outputs d0, d1, d2, d3. The other outputs of the 4-to-16 decoder are similarly generated.

For the last iteration, the `if generate` statement with label `f2` instantiates a `2to4dec component`. Note that i=3 condition is true for this iteration. This defines the 2-to-4 decoder on the left of figure 4.16 with x3 and x2 as inputs, enable e, generating outputs k0, k1, k2, and k3. It should be pointed out that the `for generate` statement could have been used by instantiating this `component` outside the `for generate` statement rather than using the `if generate` statement as above. This is done in order to illustrate the use of `if generate` statement.

Digital circuits such as registers of different sizes are needed in many applications. It is convenient to specify a register entity for which the number of flip-flops can be readily changed to conform to the size of the required register. Therefore, a `generic` parameter (integer for a register) specifying the number of flip-flops needs to be defined before port declarations using the `generic` construct. By altering this parameter, the VHDL code can be used for register of any size. The `generic map` clause can then be used to specify a different value for the register size. In order to illustrate the use of `generic and generic map`, a 4-bit inverter (bitwise 4-bit NOT operation; this can be considered as four independent inverters with four inputs and four outputs) is first defined with an entity called `inv4` using `generic` and `generate` statements. Next, copies of this 4-bit inverter are instantiated to obtain 8-bit and 16-bit inverters using `generic map` and `port map` statements. The following VHDL code illustrates this:

```
library ieee;
use ieee.std_logic-1164.all;
entity inv4 is
       generic(size:positive);
       port(a:in std_logic_vector(size-1 downto 0);
       b:out std_logic_vector(size-1 downto 0));
end inv4;
architecture inv4_example of inv4 is
component inv
       port(x:in  std_logic;
            y:out std_logic);
end component;
--VHDL code for inv
library IEEE;
useIEEE.std_logic_1164.all;
entity  inv is
    port (x: in BIT; y: out BIT);
end inv;
architecture LOGIC1 of  inv is
begin
    y<=not x;
end LOGIC1;
begin
  f1: for n in size-1 downto 0 generate
       f2: inv port map(a(n),b(n));
       end generate;
end inv4_example;
library ieee;
use ieee.std_logic_1164.all;
entity inv8_16 is
       port(a1:in std_logic_vector(7 downto 0);
            b1:out std_logic_vector(7 downto 0);
          a2:in std_logic_vector(15 downto 0);
          b2:out std_logic_vector(15 downto 0));
end inv8_16;
architecture inv_diffsize of inv8_16 is
component inv4
       generic(size:positive);
       port(a:in std_logic_vector(size-1 downto 0);
            b:out std_logic-vector(size-1 downto 0));
```

```
end component;
begin
g1:inv4 generic map(size=>8) port map(a1,b1);
g2:inv4 generic map(size=>16) port map(a2,b2);
end inv_diffsize;
```

Since VHDL is a strongly typed language, the value of a signal of one type is not permitted to be used with another signal of a different type. This means that signals of the types `bit` and `std_logic` cannot be mixed. In order to mix signals of different types, type-conversion functions can be used. For example, consider converting `std_logic` type to an `integer` type. Suppose it is desired to convert a four-bit `std_logic_vector` signal (a) into an `integer` signal (b) in the range from 0 to 15. Conversion function for assigning the value of 'b' to 'a' can be written as: `a<= conv_std_logic_vector (b,4);`.

The conversion function can be obtained by writing `use ieee.std_logic_arith.all;` at the beginning of the VHDL code after `library` and `use` statements. This conversion function is included as part of the `std_logic_arith` package. In the above, the conversion function has two parameters. These are the name of the signal to be converted (b in this case) and the number of bits in the `std_logic_vector` signal, a (four bits in this case).

Finally, VHDL keyword `constant` can be used to assign a constant value to a name which cannot be altered during simulation. The syntax for `constant` is as follows: `constant name: type := value;`. For example, the declaration `constant numb:std_logic_vector (7 downto 0) := "00001111";` will assign `numb` with the value 00001111 whenever `numb` appears in the VHDL code. This improves readability of the code.

EXAMPLE J.13

Write a VHDL description to implement the ALU of figure 7.24.

Solution

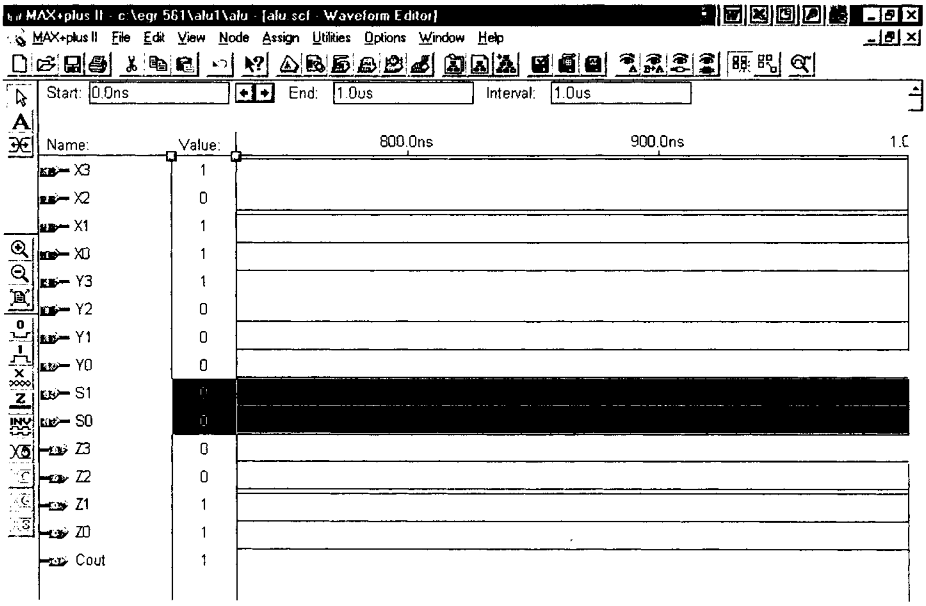
```
 LIBRARY ieee ;
 USE ieee.std logic 1164.all ;
 ENTITY mux21 IS
        PORT (wl, w0, s         : IN   STD_LOGIC ;
               fl           : OUT STD_LOGIC )
 END mux21 ;
 ARCHITECTURE Behavior OF mux21 IS
 BEGIN
        WITH s SELECT
               f1 <= w0 WHEN '0',
                    wl WHEN OTHERS ;
END Behavior ;
 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 LogicFunc OF fulladd IS
 BEGIN
        s <= x XOR y XOR Cin ;
       Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y);
END LogicFunc ;
 LIBRARY ieee   ;
 USE ieee.std   logic 1164.all ;
 ENTITY Four bitadder IS
        PORT (Cin                     : IN   STD_LOGIC ;
               x3, x2, x1, x0          : IN   STD_LOGIC ;
               y3, y2, y1, y0          : IN   STD_LOGIC ;
               s3, s2, s1, s0          : OUT  STD_LOGIC ;
               Cout                    : OUT  STD_LOGIC );
 END Four bitadder ;
 ARCHITECTURE Structure OF Four bitadder IS
        SIGNAL c1, c2, c3 :STD_LOGIC ;
        COMPONENT fulladd
               PORT (        Cin, x, y        : IN   STD_LOGIC ;
                             s, Cout          : OUT  STD_LOGIC );
        END COMPONENT ;
 BEGIN
        stage0:  fulladd   PORT MAP  ( Cin, x0, y0, s0, c1 ) ;
        stage1:  fulladd   PORT MAP  ( c1, x1, y1, s1, c2 ) ;
        stage2:  fulladd   PORT MAP  ( c2, x2, y2, s2, c3 ) ;
        stage3:  fulladd   PORT MAP  ( c3, x3, y3, s3, Cout );
               --Cin => Cout, x=>x3, y=>y3, s=>s3;
END  structure;
--Arithmetic Unit design
 LIBRARY IEEE; USE IEEE.STD LOGIC 1164.ALL;
 ENTITY Arithmetic Unit IS
 PORT (   X3, X2, X1, X0        :IN    STD_LOGIC;
          Y3, Y2, Y1, Y0         : IN   STD_LOGIC;
          S0     : IN    STD_LOGIC;
          Cout   :OUT STD_LOGIC;
      f3, f2, f1, f0: BUFFER STD_LOGIC);
 end Arithmetic Unit;
ARCHITECTURE Structure OF Arithmetic Unit IS
  COMPONENT Mux21
          PORT ( w1, w0, s       : IN    STD LOGIC; ;
          f1      : OUT STD_LOGIC; ) ;
  END COMPONENT;
  COMPONENT Four bitadder
         PORT ( Cin      : IN    STD_LOGIC;
          x3, x2, x1, x0        : IN    STD_LOGIC;
          y3, y2, y1, y0        : IN    STD_LOGIC;
                         s3, s2, s1, s0 : OUT STD_LOGIC;
                         Cout     : OUT STD_LOGIC );
  END COMPONENT;
         signal c3, c2, c1, c0 :std_logic;
         signal d3, d2, d1, d0 :std_logic;
 BEGIN
                d3 <= ( not Y3);
                d2 <= ( not Y2);
                d1 <= ( not Y1);
                d0 <= ( not Y0);
         Mux3      : MuX21 PORT MAP ( d3, Y3, S0 , c3);
         Mux2      : Mux21 PORT MAP ( d2, Y2, S0 , c2);
         Mux1      : Mux21 PORT MAP ( d1, Y1, S0 , c1);
         Mux0      : Mux21 PORT MAP ( d0, Y0, S0 , c0);
         Adder     : Four bitAdder PORT MAP ( S0, X3, X2, X1, X0, c3, c2,
                           c1, c0,f3, f2, f1, f0, Cout ) ;
 end Structure;
 -- 4-bit Two-Function Logic unit design
 LIBRARY IEEE;
 USE IEEE.STD LOGIC 1164.ALL;
 ENTITY Logic Function IS
 PORT  (  X3, X2, X1, X0                : in   std_logic;
          Y3, Y2, Y1, Y0                : in   std_logic;
```

```
     S0              : in  std_logic ;
     g3, g2, gl, g0  : buffer std_logic );
end Logic Function ;
 ARCHITECTURE Structure OF Logic Function IS
  COMPONENT Mux21
 PORT  ( wl, w0, s     : IN  STD_LOGIC ;
         fl            : OUT STD_LOGIC ) ;
 END COMPONENT;
 signal m3, m2, ml, m0 : std_logic;
 signal n3, n2, nl, n0 :std _logic;
begin
    m3  <=  (X3 and  Y3);
    m2  <=  (X2 and  Y2);
    ml  <=  (Xl and  Yl);
    m0  <=  (X0 and  Y0);
    n3  <=  (X3 xor  Y3);
    n2  <=  (X2 xor  Y2);
    nl  <=  (Xl xor  Yl);
    n0  <=  (X0 xor  Y0);
    Mux3: Mux21 Port map ( n3, m3, S0, g3);
    Mux2: Mux21 Port map ( n2, m2, S0, g2);
    Muxl: Mux21 Port map ( nl, ml, S0, gl);
    Mux0: Mux21 Port map ( n0, m0, S0, g0);
End Structure
--ALU Design
 LIBRARY IEEE;
 USE IEEE.STD LOGIC 1164.ALL;
 ENTITY ALU IS
PORT    (   X3, X2, Xl, X0 : in   std_logic
            Y3, Y2, Yl, Y0 : in   std_logic;
               S1, S0     : in std_logic ;
                Cout        : out   std_logic ;
            Z3, Z2, Z1, Z0 : buffer std_logic );
 end ALU;
 ARCHITECTURE Structure OF ALU IS
  COMPONENT Arithmetic Unit
    PORT (  X3, X2, Xl, X0          : in   std_ logic;
            Y3, Y2, Yl, Y0          : in   std_ logic;
            S0                      : in   std_ logic ;
            Cout                    : out  std_ logic ;
            f3, f2, fl, f0          : buffer std_ logic );
  END COMPONENT;
  COMPONENT Logic Function
    PORT (  X3, X2, Xl, X0          : in   std_ logic;
            Y3, Y2, Yl, Y0          : in   std_ logic;
            S0                      : in   std_ logic ;
            g3, g2, gl, g0          : buffer std_ logic );
  END COMPONENT;
  COMPONENT Mux21
    PORT ( wl, w0, s     : IN  STD_LOGIC ;
        f1         : OUT STD_LOGIC );
  END COMPONENT;
    signal m3, m2, ml, m0 : std_logic;
    signal n3, n2, nl, n0 : std_logic;
 BEGIN
  Arith: Arithmetic Unit Port map
          ( X3, X2, X1, X0, Y3, Y2, Y1, Y0, S0, Cout, m3, m2, ml, m0
  );
  Logic: Logic Function Port map
         ( X3, X2, X1, X0, Y3, Y2, Y1, Y0, S0, n3, n2, nl, n0 ) ;
 Selection3: Mux21 Port map (n3, m3, S1, Z3);
 Selection2: Mux21 Port map (n2, m2, S1, Z2);
 Selectionl: Mux21 Port map (nl, ml, S1, Z1);
```

```
Selection0: Mux21 Port map (n0, m0, S1, Z0);
end Structure;
```

SIMULATION RESULTS:

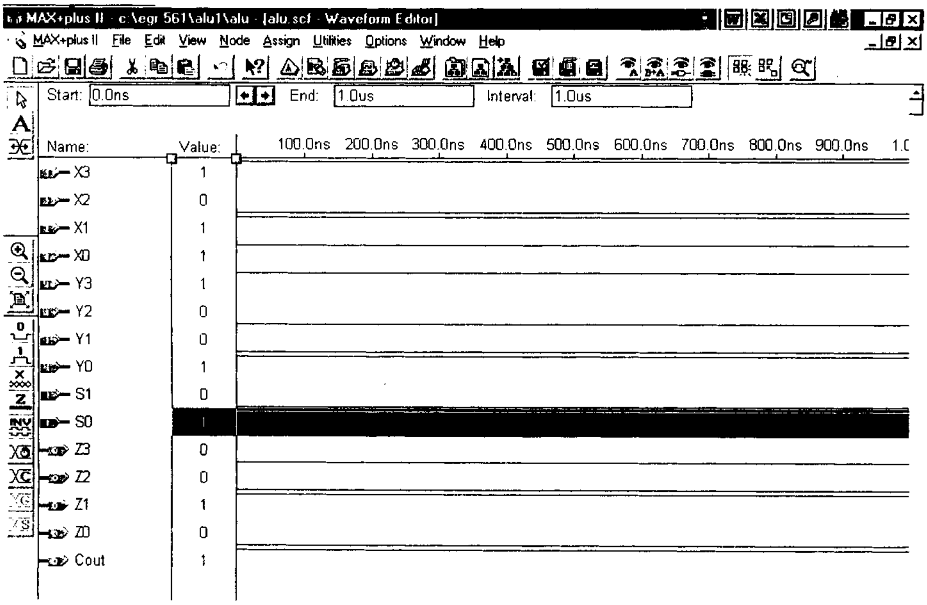
ADD Operation:


MAX+plus II - c:\egr 561\alu1\alu - [alu.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
800.0ns
900.0ns
1.0

Start: 0.0ns End: 1.0us Interval: 1.0us

Name:	Value:
X3	1
X2	0
X1	1
X0	1
Y3	1
Y2	0
Y1	0
Y0	0
S1	0
S0	0
Z3	0
Z2	0
Z1	1
Z0	1
Cout	1

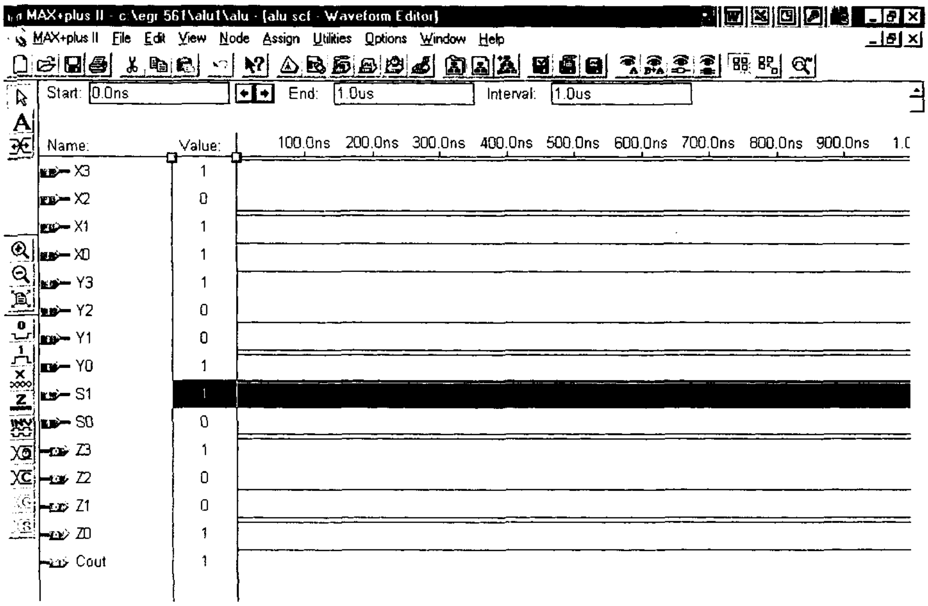
SUB Operation:


MAX+plus II - c:\egr 561\alu1\alu - [alu.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
100.0ns
200.0ns
300.0ns
400.0ns
500.0ns
600.0ns
700.0ns
800.0ns
900.0ns
1.0

Start: 0.0ns End: 1.0us Interval: 1.0us

Name:	Value:
X3	1
X2	0
X1	1
X0	1
Y3	1
Y2	0
Y1	0
Y0	1
S1	0
S0	1
Z3	0
Z2	0
Z1	1
Z0	0
Cout	1

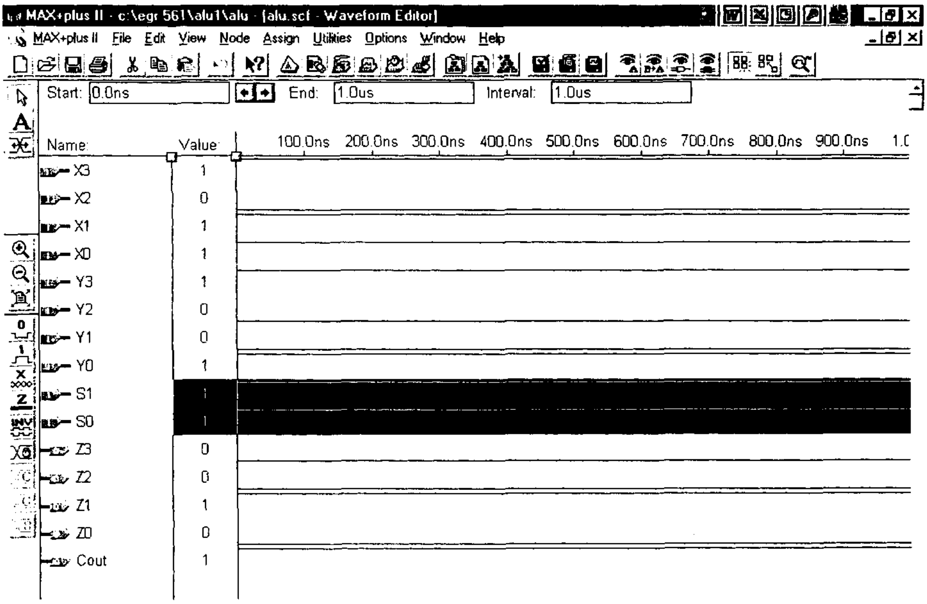
AND Operation:


MAX+plus II - c:\egr 561\alu1\alu - [alu.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
100.0ns 200.0ns 300.0ns 400.0ns 500.0ns 600.0ns 700.0ns 800.0ns 900.0ns 1.C

Start: 0.0ns End: 1.0us Interval: 1.0us

Name:	Value:
X3	1
X2	0
X1	1
X0	1
Y3	1
Y2	0
Y1	0
Y0	1
S1	1
S0	0
Z3	1
Z2	0
Z1	0
Z0	1
Cout	1

XOR Operation:


MAX+plus II - c:\egr 561\alu1\alu - [alu.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
100.0ns 200.0ns 300.0ns 400.0ns 500.0ns 600.0ns 700.0ns 800.0ns 900.0ns 1.C

Start: 0.0ns End: 1.0us Interval: 1.0us

Name:	Value:
X3	1
X2	0
X1	1
X0	1
Y3	1
Y2	0
Y1	0
Y0	1
S1	1
S0	1
Z3	0
Z2	0
Z1	1
Z0	0
Cout	1

EXAMPLE J.14

Write a VHDL description for the microprogrammed CPU described in section 7.4.

Solution

This example illustrates the design of the microprogrammed CPU by using VHDL. ModelSim simulator of Xilinx is used to implement the microprogrammed CPU. All VHDL codes of the CPU is written in Xilinx WebPack 4.2. General purpose register is used for instruction register (IR), memory address register (MAR), register A, and buffer. The VHDL module name of general purpose register is reg.

ModelSim simulator is used to simulate the VHDL program. The results can be illustrated by the timing diagrams. Figure 7.65 depicts one such timing diagram.

Fifteen modules are created in the VHDL program to implement the microprogrammed CPU. The modules are **cpu, micro1, micro2, cntr, cm, pctr, reg, alu, memory, cpu_rom, cpu_ram, ir_toxc, mux9to1 mux2to1 and fa1.** The design is created using hierarchical design. The **cpu** module is at the top of the hierarchy, **micro1** and **micro2** are under cpu module, and cntr, cm and mux9to1 are under micro1. Finally **pctr, memory, alu, ir_toxc, reg, mux2to1** and rest of the modules are under **micro2.**

Program Counter (PC)

The **pctr** module is the program counter for the instructions inside the memory.

Memory Module

The **memory** module contains **cpu_rom** and **cpu_ram** modules. Instructions are stored in the **cpu_rom**, read only memory. The instructions test a few instructions of the CPU like LOAD, STO, ADD, and HALT.

Memory Control Unit (module CM)

The **memcntrol** contains the ROM, which is filled with a 23-bit value which contains a 4-bit condition select, a 6-bit branch address, and 13-bit control input (C12 - C0) for the registers, ALU, and RAM. It also has the conditional statement that will make the Microprogram Counter (MPC) to count up by one if the load $/\overline{\text{increment}}$ is low, or will load the branch address passed by the control memory buffer.

Micro1 module

The **micro1** module connects **cntr**, **cm** and **mux9to1.**

Micro2 module

The processor module connects mux, alu, registers (regA, regIR, regMAR, regPC, regBUFF), and the memory module. It also includes the instruction decoder and does the following :

if condition select field = 0, load increment = 0, no branch,
if condition select = 1 and Z = 1, branch, if condition select = 2 and C =1, branch, if condition select = 3 and I3 = 1, branch, if condition select = 4 and XC2 = 1, branch,
if condition select = 5 and XC1 = 1, branch, if condition select = 6 and XC0 = 1, branch
if condition select = 7 and I0 = 1, branch.

CPU module

The CPU module has only two inputs: reset and clock. It connects the **micro1** module with the **micro2** module to complete the hierarchy of the microporgrammed CPU design.

--VHDL code for Microprogrammed CPU

--General Purpose Register

```
-- General purpose register
  library ieee;
use ieee.std_logic_1164.all;
entity reg is
generic ( n : integer := 8);           -- Port declarations
       port ( clk, load : in std_logic;-- clk: clock, load: load data to
reg
               x  :  in std_logic_vector ((n-1) downto 0);    -- x: input
               d  : out std_logic_vector ((n-1) downto 0) ); -- d: output
end reg;
architecture reg_arch of reg is
```

```
begin                         -- Process when clock and load change
        p1 : process ( clk, load )  -- if the clocking signal (clk)
        begin                       -- represents the rising edge
            if clk = ‘1’ and clk’event then -- and if load pin is high then
                if load = ‘1’ then        -- stores the data into
                              d <= x;     -- the reg
                       end if;
                end if;
        end process;
end reg_arch;
```

--Program Counter   (PC)

```
-- program counter
  library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity pctr is
generic ( n : integer := 8 );                 -- Port declarations
    port ( clk, clr, inc, load : in std_logic; -- clk: clock, clr: clear PC
    x    : in std_logic_vector ((n-1) downto 0);
    d    : out std_logic_vector ((n-1) downto 0) ); --,load: load
        --branch address, x: input
            -- d: output
end pctr;
  architecture pctr_arch of pctr is
    signal in_d : unsigned (x’range);      -- in_d: connect d
    signal in_x : unsigned (x’range);      -- in_x: connect x
begin
    p1 : process ( clk, clr, inc, load )   -- if clk = rising edge
    begin                                  -- and clr = 1
        if clk = ‘1’ and clk’event then    -- then PC <- 0
            if clr = ‘1’ then              -- if clk = rising edge
            in_d <= conv_unsigned(0,n);    -- and clr=0,inc = 1, load = 0
            else                           -- then PC <- PC + 1
                if inc = ‘1’ then          -- if clk = rising edge
                in_d <= in_d + 1;          -- and clr= 0, inc = 0, load = 1
                else                       -- then PC <- x
                                           if load = ‘1’ then
                              in_d <= in_x;
                       end if;
                end if;
            end if;
        end if;
    end process;
    g1 : for i in x’range generate   -- for i = 0 to 7 loop
                in_x(i) <= x(i);
                d(i) <= in_d(i);

       end generate;
end pctr_arch;
```

--Full adder

```
-- Full adder
  library ieee;
use ieee.std_logic_1164.all;
entity fal is                                             -- Port
declarations
    port ( a, b, c          : in std_logic;   -- c: carry input
        s, cout, anda, nota : out std_logic );-- s: sum, cout: carry output
end fal;                                     -- anda: a AND b, nota: NOT a
architecture fal_arch of fal is
```

```
      signal in_anda : std_logic;       -- in_anda: connect anda
begin
      s       <= a xor b xor c;
      cout    <= in_anda or (b and c) or (c and a);
      in_anda <= a and b;
      nota    <= not a;
      anda    <= in_anda;
end fal_arch;
```

--ALU module

```
-- Arithmetic logic unit
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity alu is                                   -- Port declarations
   generic ( n : integer := 8 );
   port (CTRL : in STD_LOGIC_VECTOR (0 to 2);-- CTRL: control input
      L, R : in STD_LOGIC_VECTOR ((n-1) downto 0);-- L, R: source inputs
      F    : out STD_LOGIC_VECTOR ((n-1) downto 0);-- F: result output
      C, Z : out STD_LOGIC ); -- C: carry flag, Z: zero flag
end alu;
architecture alu_arch of alu is
component fal
   port (  a, b, c              : in STD_LOGIC;
               s, cout, anda, nota : out STD_LOGIC );
end component;
      signal in_L, in_R, in_xR, in_F : unsigned (L'range);
      -- in_L: connect L, a, in_R: connect R

   signal in_zer, in_sum, in_and,   -- in_xR: connect b, in_F: connect F
     in_not, in_inc, in_dec  : unsigned (L'range); -- in_zer: connect 0,
                                                  -- in_sum: connect s
   signal in_c : STD_LOGIC_VECTOR (n downto 0);
       -- in_and: connect anda, in_zf: connect Z
   signal in_zf                 : boolean;-- in_not: connect nota,
begin                                         -- in_c: connect C,
CTRL(2), cout
   gen : for i in L'range generate          -- for i = 0 to 7 loop
      fa_i : fal port map ( in_L(i), in_xR(i), in_c(i), in_sum(i),
                            in_c(i+1), in_and(i), in_not(i) );
      in_xR(i) <= in_R(i) xor CTRL(2); -- CTRL(2) can determine add
                                              -- CTRL(2) = 0
      in_R(i)<= R(i);         -- or subtract CTRL(2) = 1
      in_L(i)<= L(i);         -- if CTRL(2) = 1, in_R(i) xor CTRL(2)
   F(i)   <= in_F(i) after 200 ps;-- performs 1's complement of R
   end generate;
   in_zer  <= CONV_UNSIGNED(0, n);
   in_inc  <= in_L + 1 after 500 ps;
   in_dec  <= in_L - 1 after 500 ps;
   in_c(0) <= CTRL(2);                -- performs 2's complement of R
   C       <= in_c(n);
   in_zf   <= ( in_F = 0 ) after 500 ps;
   with CTRL select
   in_F <=     in_zer when "000",   -- f=0 if ctrl=0
               in_R   when "001",    -- f=R if ctrl=1
                 in_sum when "010", -- f=L+R if ctrl=2
                 in_sum when "011", -- f=L-R if ctrl=3
                 in_inc when "100", -- f=L+1 if ctrl=4
                 in_dec when "101", -- f=L-1 if ctrl=5
                 in_and when "110",  -- f=L&R if ctrl=6
                 in_not when others; -- f=~L if ctrl=others
```

```
        with in_zf select
               z <= ‘1’ when True,       -- z = 1 if in_zf = true
                     ‘0’ when others; -- z= 0 if in_zf = others
end alu_arch;
```

--ROM

```
-- Read only memory (ROM)
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY cpu_rom IS
    PORT ( addr : in std_logic_vector (6 downto 0);-- addr: address input
           data : out std_logic_vector (7 downto 0));-- data: data output
end cpu_rom;
ARCHITECTURE Arch_rom OF cpu_rom IS                    -- Programming ROM
        -- Define instruction to opcode
        constant LDA   : std_logic_vector := “00001000”;--08h
               constant STA   : std_logic_vector := “00001001”;--09h
        constant ADD   : std_logic_vector := “00001010”;--0Ah
        constant SUB   : std_logic_vector := “00001011”;--0Bh
        constant JZ    : std_logic_vector := “00001100”;--0Ch
        constant JC    : std_logic_vector := “00001101”;--0Dh
        constant A_ND  : std_logic_vector := “00001110”;--0Eh
        constant CMA   : std_logic_vector := “00000000”;--00h
        constant INCA  : std_logic_vector := “00000010”;--02h
        constant DCRA  : std_logic_vector := “00000100”;--04h
        constant HLT   : std_logic_vector := “00000110”;--06h
        constant OUTPR : std_logic_vector := “10010000”;--90h
               -- Define label to memory address
        constant D1    : std_logic_vector := “00000110”;--06h
        constant D2    : std_logic_vector := “00000111”;--07h
        constant D3    : std_logic_vector := “00001000”;--08h
        constant D4    : std_logic_vector := “00001001”;--09h
        constant D5    : std_logic_vector := “00001010”;--0Ah
        constant PROD  : std_logic_vector := “10000000”;--80h
        constant CNTR  : std_logic_vector := “10000001”;--81h
        constant V2    : std_logic_vector := “10000010”;--82h
        constant V3    : std_logic_vector := “10000011”;--83h
        constant V4    : std_logic_vector := “10000100”;--84h
        constant V5    : std_logic_vector := “10000101”;--85h
        constant V6    : std_logic_vector := “10000110”;--86h
        constant V7    : std_logic_vector := “10000111”;--87h
        constant V8    : std_logic_vector := “10001000”;--88h
        constant V9    : std_logic_vector := “10001001”;--89h
        constant VA    : std_logic_vector := “10001010”;--8Ah
        constant VB    : std_logic_vector := “10001011”;--8Bh
        constant VC    : std_logic_vector := “10001100”;--8Ch
        constant VD    : std_logic_vector := “10001101”;--8Dh
        constant VE    : std_logic_vector := “10001110”;--8Eh
        constant VF    : std_logic_vector := “10001111”;--8Fh
        constant BEG   : std_logic_vector := “00010010”;--12h
        constant LOP   : std_logic_vector := “00101101”;--2Dh
        constant ENDS  : std_logic_vector := “01000000”;--40h
        signal in_data : std_logic_vector (7 downto 0);

 -- Signal declaration
begin
        with addr select
            in_data <=   LDA      when “0000000”,-- 0 A <- D1 (A = 80h)
                        D1       when “0000001”,-- 1 D1 = 80h
                        ADD      when “0000010”,-- 2 A <- A + D1(A=0,CF=1)
                        D1       when “0000011”,-- 3 D1 = 80h
```

```
                              JC         when “0000100”,-- 4 Jump to begin if A=0
                              BEG        when “0000101”,-- 5 BEG :=”00010010” = 12
                            “10000000”   when “0000110”,-- 6 D1            80h
                            “01001011”   when “0000111”,-- 7 D2            4Bh
                            “01010001”   when “0001000”,-- 8 D3            51h
                            “00110010”   when “0001001”,-- 9 D4            32h
                            “00000100”   when “0001010”,-- A D5            04h
                            ADD          when “0010010”,-- 12 A <- A + D2.(A = 4Bh)
                            D2           when “0010011”,-- 13 D2 = 4Bh
                            STA          when “0010100”,-- 14 Outport <- 4Bh
                            OUTPR        when “0010101”,-- 15
                            A_ND         when “0010110”,-- 16 A <- 4Bh&51h(A = 41h)
                            D3           when “0010111”,-- 17 D3 = 51h
                            STA          when “0011000”,-- 18 Outport <- 41h
                            OUTPR        when “0011001”,-- 19
                            CMA          when “0011010”,-- 1A A <- ~A (A = BEh)
                            STA          when “0011011”,-- 1B Outport <- BEh
                            OUTPR        when “0011100”,-- 1C
                            INCA         when “0011101”,-- 1D A <- A + 1 (A=BFh)
                            STA          when “0011110”,-- 1E Outport <- BFh
                            OUTPR        when “0011111”,-- 1F
                            DCRA         when “0100000”,-- 20 A <- A - 1 (A=BEh)
                            STA          when “0100001”,-- 21 Outport <- BEh
                            OUTPR        when “0100010”,-- 22
                            LDA          when “0100011”,-- 23 A <- D4 (A = 32h)
                            D4           when “0100100”,-- 24 D4 = 32h
                            SUB          when “0100101”,-- 25 A <- A - D4 (A = 00h)
                            D4           when “0100110”,-- 26 D4 = 32h
                            STA          when “0100111”,-- 27 PROD <- A(PROD = 00h)
                            PROD         when “0101000”,-- 28
                            LDA          when “0101001”,-- 29 A <- D5 (A = 04h)
                            D5           when “0101010”,-- 2A D5 = 04h
                            STA          when “0101011”,-- 2B CNTR <-A (CNTR = 04h)
                            CNTR         when “0101100”,-- 2C
                            LDA          when “0101101”,-- 2D LOOP:PROD<-PROD +D4
                            PROD         when “0101110”,-- 2E
                            ADD          when “0101111”,-- 2F A <- A + D4
                            D4           when “0110000”,-- 30 D4 = 32h
                            STA          when “0110001”,-- 31 PROD <- A
                            PROD         when “0110010”,-- 32
                            LDA          when “0110011”,-- 33    CNTR <- CNTR -1
                            CNTR         when “0110100”,-- 34
                            DCRA         when “0110101”,-- 35 A <- A - 1
                            JZ           when “0110110”,-- 36 If CNTR = 0 then
                            ENDS         when “0110111”,-- 37 Goto End, ENDS
                            STA          when “0111000”,-- 38 CNTA <- A
                            CNTR         when “0111001”,-- 39
                            LDA          when “0111010”,-- 3A Goto Loop
                            D1           when “0111011”,-- 3B D1 = 80h
                            SUB          when “0111100”,-- 3C A <- A - D1 (A = 00h)
                            D1           when “0111101”,-- 3D D1 = 80h
                            JZ           when “0111110”,-- 3E If A = 0 then
                            LOP          when “0111111”,-- 3F
                            LDA          when “1000000”,-- 40 End: Outport <- PROD
                            PROD         when “1000001”,-- 41
                            STA          when “1000010”,-- 42 Outport <- A
                            OUTPR        when “1000011”,-- 43
                            HLT          when others;   -- n
        data <= in_data after 200 ps;
end Arch_rom;
```

--RAM

```
-- Random access memory (RAM)
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity cpu_ram is
generic ( nw : integer := 8;
       n1 : integer := 4 );
    port ( rw, en : in STD_LOGIC;-- rw: read/write, en: enable RAM
       addr   : in STD_LOGIC_VECTOR ((n1-1) downto 0);
-- addr: address input
       d_in   : in STD_LOGIC_VECTOR ((nw-1) downto 0);
-- d_in: data input
       d_out  : out STD_LOGIC_VECTOR ((nw-1) downto 0) );  -- d_out
                                                            -- data output
end cpu_ram;
architecture cpu_ram_arch of cpu_ram is
     type Ram_Word is array ( d_in'range ) of STD_LOGIC;-- type declaration
     type Ram_Array is array ( 0 to ((2**n1)-1)) of Ram_Word;-- type
                                                            -- declaration
    signal in_din, dout1, dout2, in_dout     : Ram_Word;-- in_din: connect
d_in,                                                   --dout2: connect 0
    signal in_addr                      : unsigned (addr'range);
-- in_out: connect d_out
    signal Ram_Mem                      : Ram_Array;-- in_addr: connect
                                                         --addr
begin
    p: process ( rw, en, in_addr )
        variable intaddr : integer;
    begin
        intaddr := CONV_INTEGER (in_addr);   --convert binary number
                                              -- to integer
        dout1 <= Ram_Mem(intaddr);
        if en = '0' and rw = '0' then
-- if en = 0 and rw = 0
            Ram_Mem(intaddr) <= in_din after 500 ps;
-- then write data into the RAM
        end if;
    end process;
    with en select
        in_dout <= dout1 when '0',
                    dout2 when others;
    g1: for i in d_out'range generate
-- for i = 0 to 7 loop
        in_din(i) <= d_in(i);
        d_out(i)  <= in_dout(i) after 200 ps;
        dout2(i)  <= '0';
-- set dout2 := "00000000"
    end generate;
    g2: for i in addr'range generate
-- for i = 0 to 3 loop
        in_addr(i) <= addr(i) after 100 ps;
    end generate;
end cpu_ram_arch;
```

--Memory for CPU (ROM + RAM)

```
-- memory for cpu
library IEEE;
use IEEE.std_logic_1164.all;
entity memory is
        port ( RW, EN    : in STD_LOGIC;
-- RW: read/write, EN: enable memory
              addr, din : in STD_LOGIC_VECTOR (7 downto 0);

-- addr: address input, din: data input
              dout      : out STD_LOGIC_VECTOR (7 downto 0);
-- dout: data output
              ioout     : out STD_LOGIC_VECTOR (7 downto 0) );
-- ioout: data io output
end memory;
architecture memory_arch of memory is
component cpu_ram
       generic ( nw, nl : integer );
              port ( rw, en : in STD_LOGIC;
                     addr   : in STD_LOGIC_VECTOR ((nl-1) downto 0);
                     d_in   : in STD_LOGIC_VECTOR ((nw-1) downto 0);
                     d_out   : out STD_LOGIC_VECTOR ((nw-1) downto 0) );
end component;
component cpu_rom
       port ( addr : in STD_LOGIC_VECTOR (6 downto 0);
              data : out STD_LOGIC_VECTOR (7 downto 0) );
end component;   -- in_d1: connect data
       signal in_d1, in_d2 : STD_LOGIC_VECTOR ( 7 downto 0);
-- in_d2: connect d_out
       signal in_EnRAM     : STD_LOGIC;

-- in_EnRAM: connect en
begin
       rom1 : cpu_rom port map (addr=>addr(6 downto 0), data =>in_d1);
       ram1 : cpu_ram generic map (8, 4)
              port map (rw=>RW, en=>in_EnRAM, addr=>addr(3 downto 0),
                          d_in=>din, d_out=>in_d2);
       in_EnRAM <= EN or ( not addr(7) ) or addr(6) or addr(5) or addr(4);
-- memory mapping:
       with addr(7) select

-- programmed ROM when address =
              dout <= in_d2 when '1',

-- 00000000 to 01111111 (128 bytes)
                      in_d1 when others;

-- RAM when address =
              with addr select

-- 10000000 to 10001111 (16 bytes)
                      ioout <= din after 1 ns when "10010000",

-- IO when address =
                              "00000000" after 800 ps when others;
-- 10010000 (1 byte)
end memory_arch;
```

--Multiplexer 2 to 1

```
-- Multiplexer 2 to 1
  library IEEE;
use IEEE.std_logic_1164.all;
entity mux2to1 is
generic ( n : integer :=8);
       port ( s1, s0 : in STD_LOGIC_VECTOR ((n-1) downto 0);
-- s0, s1: source inputs
              s      : in STD_LOGIC;

-- s: select line
              f      : out STD_LOGIC_VECTOR ((n-1) downto 0) );
-- f: output
end mux2to1;
architecture arch_mux of mux2to1 is
begin
       with s select
              f <= s0 when '0',
                   s1 when others;
end arch_mux;
```

--Instruction Decoder

```
-- Instruction decoder
library IEEE;
use IEEE.std_logic_1164.all;
entity ir_to_xc is
       port ( i  : in STD_LOGIC_VECTOR (1 downto 0);

-- i: op-code bit 1 & 2
              xc : out STD_LOGIC_VECTOR ( 2 downto 0) );

-- xc: group number output
end ir_to_xc;
architecture ir_to_xc_arch of ir_to_xc is
begin
       with i select

              xc <= "001" when "00",

-- group 0
                    "010" when "01",

-- group 1
                    "100" when "10",

-- group 2
                    "000" when others;

-- group 3
end ir_to_xc_arch;
```

--Micro2 module

```
-- Overall hardware2 ( PC + Reg + Mux2to1 + ALU + Memory + IR_to_XC )
library ieee;
use ieee.std_logic_1164.all;
entity micro2 is
       port ( ctrl          : in STD_LOGIC_VECTOR (0 TO 12);
-- ctrl: control inputs C0-C12
              clr, clk      : in STD_LOGIC;

-- clk: clock, clr: clear
              dataout       : out STD_LOGIC_VECTOR ( 7 downto 0);
-- dataout: data output
              z, c, i3, i0 : out STD_LOGIC;
-- z: zero flag, c: carry flag
               xc             : out std_logic_vector ( 2 downto 0) );
-- i3, i0: op-code bit 3 & 0
end micro2;

-- xc: group number
architecture micro2_arch of micro2 is
component pctr
        generic ( n: integer);
                port ( clk, clr,

-- clr: C0, inc: C1, load: C2
                         inc, load : in STD_LOGIC;
                        x          : in STD_LOGIC_VECTOR ((n-1) downto 0);
-- x: branch
                        d          : out STD_LOGIC_VECTOR ((n-1) downto 0) );
-- d: memory reference
end component;
component reg     -- instantiate Register
        generic ( n: integer );
                port ( clk, load : in STD_LOGIC;

-- load: C4, C7, C8, C9
                        x          : in STD_LOGIC_VECTOR ((n-1) downto 0);
-- x: data input
                        d          : out STD_LOGIC_VECTOR ((n-1) downto 0) );
-- d: data output
end component;
component mux2to1     -- instantiate mux 2 to 1
        generic ( n: integer );
                port ( s1, s0 : in STD_LOGIC_VECTOR ((n-1) downto 0);
-- s1: from buffer, s0: from PC
                        s      : in STD_LOGIC;

-- s: C3
                        f      : out STD_LOGIC_VECTOR ((n-1) downto 0 ) );
-- f: to MAR
end component;
component alu      -- instantiate ALU
        generic ( n: integer );
                port ( CTRL : in STD_LOGIC_VECTOR (0 to 2);
-- CTRL: C10, C11, C12
                        L, R : in STD_LOGIC_VECTOR ((n-1) downto 0);
-- L, R: data input
                        F    : out STD_LOGIC_VECTOR ((n-1) downto 0);
-- F: data output
                        C, Z : out STD_LOGIC );

-- C: carry flag, Z: zero flag
end component;
component memory                -- instantiate memory
        port ( RW, EN      : in STD_LOGIC;

-- RW: C5, EN: C6
               addr, din   : in STD_LOGIC_VECTOR (7 downto 0);

-- addr: from MAR, din: from reg A
               dout        : out STD_LOGIC_VECTOR (7 downto 0);
-- dout: to PC, IR, buffer
               ioout       : out STD_LOGIC_VECTOR (7 downto 0) );
```

```
-- ioout: to IO
end component;
component ir_to_xc    -- instantiate instruction decoder
       port ( i  : in STD_LOGIC_VECTOR (1 downto 0);

-- i: from IR, I1 & I2
              xc : out STD_LOGIC_VECTOR ( 2 downto 0) );

-- xc: group number
end component;
       signal opc, oir, omux, omar,

-- opc: connect PC & MUX
              orega, obuf, oalu, omem      : STD_LOGIC_VECTOR ( 7 downto
0);
-- oir: connect IR & instruction decoding
       signal in_clr, en_flag, incf : STD_LOGIC;
-- omux: connect MUX & MAR
       signal i_cf, o_cf       : STD_LOGIC_VECTOR (0 downto 0);
-- omar: connect MAR & memory
begin

-- orega: connect Reg A & ALU (L)
       the_pc    : pctr generic map (8)

-- obuf: connect Buffer & ALU (R)
                      port map (clk, in_clr, ctrl(1), ctrl(2), omem, opc);
-- oalu: connect Reg A & ALU (F)
       the_ir    : reg generic map (8)

-- omem: connect memory & PC, IR, Buffer
                      port map (clk, ctrl(8), omem, oir);

-- in_clr: connect C0 or clr
       the_mar   : reg generic map (8)

-- en_flag: connect Z, C
                      port map (clk, ctrl(4), omux, omar);

-- inzf: connect ALU
       the_rega : reg generic map (8)

-- incf: connect ALU
                      port map (clk, ctrl(9), oalu, orega);

-- i_zf: connect Z, i_cf: connect C
       the_buf   : reg generic map (8)

-- o_zf: connect Z, o_cf: connect C
                      port map (clk, ctrl(7), omem, obuf);

       the_mux   : mux2to1 generic map (8)

                      port map (obuf, opc, ctrl(3), omux);
       the_alu   : alu generic map (8)
                      port map (CTRL=>ctrl(10 to 12), L=>orega,
                                   R=>obuf, F=>oalu, C=>incf, Z=>inzf);
--The zero flag is connected directly to the alu, the carry flag is
--instantiated.
       the_cf    : reg generic map (1)
```

```
                    port map (clk, en_flag, i_cf, o_cf);
       the_mem   : memory port map (ctrl(5), ctrl(6), omar,
                                 orega, omem, dataout);
       the_dec   : ir_to_xc port map (i=>oir(2 downto 1), xc=>xc);
       in_clr  <= ctrl(0) or clr;

-- ctrl(0): PC <- 0
       c       <= o_cf(0);
       i_cf(0) <= incf;
       i3      <= oir(3);

-- i3: type classifier
       i0      <= oir(0);

-- i0: subcategory within a group
       en_flag <= ctrl(10) or ctrl(11) or ctrl(12);

-- ctrl(10), ctrl(11), ctrl(12):   -- ALU control input
end micro2_arch;
```

--Memory Control Unit (module CM)

```
-- Control Unit
  LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY cm IS
       PORT ( addr : in std_logic_vector (5 downto 0);

-- addr: address input
              cmdb : out std_logic_vector (22 downto 0) );
-- cmbd: data output
end cm;
ARCHITECTURE Arch_cm OF cm IS
       signal in_cmdb : std_logic_vector (22 downto 0);

-- in_cmbd: connect cmbd
-- Binary microprogram
-- The size of the control memory is 53 x 23 bits.  The 23-bit control word
-- consists of 13- bit control function containing C0 through C12 with C0
-- as bit 12 and C12 as bit 0. The branch address field is 6-bit wide (bits
-- 13-18).  For example, consider the code for line 0 with the operation
-- PC <- 0 in the following. Since there is no condition in this operation,
-- condition select field ( CS ) and  branch address field ( Brn ) are all
-- 0's.  To clear PC to  0, C0 = 1 .  To disable RAM,  C6 = 1 and, C5(R/W')
-- is arbitrarily set to one.
begin
       with addr select
       --  22 19    12            0
       --  |CS| Brn  | CTR FUNC  |
n_cmdb <= "00000000001000011000000" when "000000",-- 0  PC <- 0
"00000000000000111000000" when "000001",    --1 FETCH   MAR<-PC
"00000000000100010010000" when "000010",    --2  IR<- M(MAR), PC <- PC +1
"00110011100000011000000" when "000011",    --3  IF I3=1, goto MEMR(14)
"01100010000000011000000" when "000100",    --4  IF XC0=1, goto CMA(8)
"01010010100000011000000" when "000101",    -- 5 IF XC1=1, goto INCA(10)
"01000011000000011000000" when "000110",    -- 6 IF XC2=1, goto DCRA(12)
"10001101000000011000000" when "000111",    -- 7 goto HALT(50)
"00000000000000011001111" when "001000",    -- 8 CMA   A <- ~A
"10000000010000011000000" when "001001",    -- 9       goto FETCH
"00000000000000011001100" when "001010",    -- 10 INCA A <- A + 1
"10000000010000011000000" when "001011",    -- 11      goto FETCH
```

```
“00000000000000011001101” when “001100”,    -- 12 DCRA A <- A - 1
“10000000010000011000000” when “001101”,    -- 13      goto FETCH
“01100101110000011000000” when “001110”,    -- 14 MEMREF IF XC0=1, goto
                                            --    LDSTO(23)
“01011000000000011000000” when “001111”,    -- 15 IF XC1=1, goto ADSUB(32)
“01001010010000011000000” when “010000”,    -- 16 IF XC2=1, goto JMPS(41)
“00000000000000111000000” when “010001”,    -- 17 AND    MAR <- PC
“00000000000100010100000” when “010010”,    -- 18 BUFFER <- M(MAR),
                                            --    PC <- PC+1
“00000000000001111000000” when “010011”,    -- 19 MAR <- BUFFER
“00000000000000010100000” when “010100”,    -- 20 BUFFER <- M(MAR)
“00000000000000011001110” when “010101”,    -- 21 A <- A ^ BUFFER
“10000000010000011000000” when “010110”,    -- 22 goto FETCH
“00000000000000111000000” when “010111”,    -- 23 LDSTO  MAR <- PC
“00000000000100010100000” when “011000”,  -- 24 BUFFER <- M(MAR),
                                          --    PC <- PC + 1
“00000000000001111000000” when “011001”,  -- 25 MAR <- BUFFER
“01110111100000011000000” when “011010”,  -- 26 IF I0=1, goto STO(30)
“00000000000000010100000” when “011011”,  -- 27 LOAD   BUFFER <- M(MAR)
“00000000000000011001001” when “011100”,  -- 28 A <- BUFFER
“10000000010000011000000” when “011101”,  -- 29 goto FETCH
“00000000000000000000000” when “011110”,  -- 30 STO M(MAR) <- A
“10000000010000011000000” when “011111”,  -- 31 goto FETCH
“00000000000000111000000” when “100000”,  -- 32 ADSUB  MAR <- PC
“00000000000100010100000” when “100001”,  -- 33 BUFFER <- M(MAR),
                                          --    PC <- PC +1
“00000000000001111000000” when “100010”,  -- 34 MAR <- BUFFER
“00000000000000010100000” when “100011”,  -- 35 BUFFER <- M(MAR)
“01111001110000011000000” when “100100”,  -- 36 IF I0=1, goto SUB(39)
“00000000000000011001010” when “100101”,  -- 37 ADD A <- A + BUFFER
“10000000010000011000000” when “100110”,  -- 38 goto FETCH
“00000000000000011001011” when “100111”,  -- 39 SUB A <- A - BUFFER
“10000000010000011000000” when “101000”,  -- 40 goto FETCH
“00000000000000111000000” when “101001”,  -- 41 JMPS MAR <- PC
“00000000000000011000000” when “101010”,  -- 42
“01111011110000011000000” when “101011”,  -- 43 IF I0=1, goto JOC(47)
“00011100100000011000000” when “101100”,  -- 44 JOZ IF Z=1, goto LOADPC
“00000000000100011000000” when “101101”,  -- 45 PC <- PC + 1
“10000000010000011000000” when “101110”,  -- 46 goto FETCH
“00101100100000011000000” when “101111”,  -- 47 JOC IF C=1, goto LOADPC(50)
“00000000000100011000000” when “110000”,  -- 48 PC <- PC+ 1
“10000000010000011000000” when “110001”,  -- 49 goto FETCH
“00000000000010010000000” when “110010”,  -- 50 LOADPC PC <- M(MAR)
“10000000010000011000000” when “110011”,  -- 51 goto FETCH
“10001101000000011000000” when others;    -- 52 HALT goto HALT
       cmdb <= in_cmdb after 200 ps;
end Arch_cm;
```

--Microprogram Counter Module(MPC)

```
-- Microprogramming counter
  library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity cntr is
generic ( n : integer := 6 );
       port( clk : in STD_LOGIC;  -- clk: clock
             clr : in STD_LOGIC;-- clr: clear MPC
             li  : in STD_LOGIC;-- li: load/increase
             x   : in STD_LOGIC_VECTOR ((n-1) downto 0);-- x: data input
             d   : out STD_LOGIC_VECTOR ((n-1) downto 0) );--d:data output
end cntr;
```

```
architecture cntr_arch of cntr is
       signal in_d : UNSIGNED (x'range);-- in_d: connect d
       signal in_x : UNSIGNED (x'range);-- in_x: connect x
begin
       p1 : process ( clk, clr, li)
       begin
              if clk = '1' and clk'event then     -- if clk = rising edge
                     if clr = '1' then            -- and clr = 1
              in_d <= CONV_UNSIGNED(0, n) after 200 ps; -- then MPC <- 0
                     else                         -- if clk = rising edge
                     if li = '0' then            -- and clr = 0, li = 0
                       in_d <= in_d + 1 after 500 ps;-- MPC <- MPC + 1
                            else                 -- if clk = rising edge
                      in_d <= in_x after 500 ps;-- and clr = 0, li = 1
                            end if;                     -- MPC <- x
                     end if;
              end if;
       end process;
       g1 : for i in x'range generate -- for i = 0 to 5 loop
              in_x(i) <= x(i);
              d(i) <= in_d(i);
       end generate;
end cntr_arch;
```

--Mux 9 to 1

```
-- Multiplexer 9 to 1

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux9to1 IS
PORT ( w : in std_logic_vector (8 downto 0);-- w: input
       s : in std_logic_vector (3 downto 0);-- s: select line
       f : out std_logic );                 -- f: output
end mux9to1;
ARCHITECTURE Arch_Mux OF mux9to1 IS
       begin
              with s select

                     f <= w(0) when "0000",

                            w(1) when "0001",

                            w(2) when "0010",

                            w(3) when "0011",

                            w(4) when "0100",

                            w(5) when "0101",

                            w(6) when "0110",

                            w(7) when "0111",

                            w(8) when others;

end Arch_Mux;
```

--Micro1 (MPC + decoder + CM)

```
-- Overall hardware1 ( MPC + Mux9to1 + CM )
 library IEEE;
use IEEE.std_logic_1164.all;
entity micro1 is
    port( Z   : in STD_LOGIC;      -- Z: zero flag
        C   : in STD_LOGIC;    -- C: carry flag
        I3  : in STD_LOGIC;    -- I3: type classifier( if I3=1, then
        XC  : in STD_LOGIC_VECTOR (2 downto 0);-- it is a MRL, othewise
                                                --it is a NMRI)
           I0  : in STD_LOGIC;                    -- XC: group number
           CLR : in STD_LOGIC;          -- I0: subcategory within a group
           CLK : in STD_LOGIC;          -- CLR: clear MPC
           CTN : out STD_LOGIC_VECTOR (0 to 12) );-- CLK: clock
end micro1;                                       -- CTN: control functions
architecture micro1_arch of micro1 is
    component cntr
        generic ( n : integer );
        port (    clk : in STD_LOGIC;
                       clr : in STD_LOGIC;
                       li  : in STD_LOGIC;
                       x   : in STD_LOGIC_VECTOR ((n-1) downto 0);
                       d   : out STD_LOGIC_VECTOR ((n-1) downto 0) );
    end component;
    component mux9to1
        port ( w : in std_logic_vector (8 downto 0);
                      s : in std_logic_vector (3 downto 0);
                    f : out std_logic );
        end component;
        component cm
               port ( addr : in std_logic_vector (5 downto 0);
                       cmdb : out std_logic_vector (22 downto 0) );
        end component;
        signal in_addr, in_brnh : STD_LOGIC_VECTOR (5 downto 0);
-- in_addr: connect MPC & CM
        signal in_cs            : STD_LOGIC_VECTOR (3 downto 0);
-- in_brnh: connect MPC cmbd(18 downto 13)
        signal in_li, IH, IL    : STD_LOGIC;

 -- in_cs: connect s & cmbd(22 downto 19)
begin                                  -- in_li: connect MUX & MPC
        cntr1 : cntr generic map (6)  -- IH: connect Vcc, IL: connect GND
               port map (clk=>clk, clr=>clr, li=>in_li, x=>in_brnh,
               d=>in_addr);
        mux91 : mux9to1 port map (w(8)=>IH, w(7)=>I0, w(6)=>XC(0),

                           w(5)=>XC(1), w(4)=>XC(2), w(3)=>I3,
                             w(2)=>C, w(1)=>Z, w(0)=>IL, s=>in_cs, f=>in_
li);
        cm1 : cm port map (addr=>in_addr, cmdb(22 downto 19)=>in_cs,
                           cmdb(18 downto 13)=>in_brnh, cmdb(12 downto
                                                         0)=>CTN);
        IH <= ‘1’;
        IL <= ‘0’;
end micro1_arch;
```

--CPU module

```
-- Microprogrammed CPU
  library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity CPU is
       port ( clk, reset: in STD_LOGIC;-- clk: clock
            d_out: out STD_LOGIC_VECTOR (7 downto 0) );- d_out:data output
end CPU;
architecture CPU_arch of CPU is
component micro1
       port (  Z   : in STD_LOGIC;
              C   : in STD_LOGIC;
              I3  : in STD_LOGIC;
              XC  : in STD_LOGIC_VECTOR (2 downto 0);
              I0  : in STD_LOGIC;
              CLR : in STD_LOGIC;
              CLK : in STD_LOGIC;
              CTN : out STD_LOGIC_VECTOR (0 to 12) );
end component;
component micro2
       port (   ctrl        : in STD_LOGIC_VECTOR (0 to 12);
              clr, clk     : in STD_LOGIC;
              dataout      : out STD_LOGIC_VECTOR (7 downto 0);
              Z, C, I3,I0  : out STD_LOGIC;
              XC           : out STD_LOGIC_VECTOR (2 downto 0));
end component;
       signal in_Z, in_C, in_I3, in_I0 : STD_LOGIC;
 -- in_Z: connect Z, in_C: connect C
-- in_I3: connect I3, in_I0: connect I0

       signal ctrl                      : STD_LOGIC_VECTOR (0 to 12);
       signal in_XC                     : STD_LOGIC_VECTOR (2 downto 0);
-- ctrl: connect CTN, in_xc: XC
begin
       the_mpc : micro1 port map ( in_Z, in_C, in_I3, in_XC, in_I0,
                                   reset, clk, ctrl );
       the_hdw : micro2 port map ( ctrl, reset, clk, d_out, in_Z, in_C,
                                   in_I3, in_I0, in_XC );
end CPU_arch;
```

--Test Bench for CPU module

```
-- CPU test bench
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS   -- Architecture of the test bench
COMPONENT cpu                            -- instantiate CPU module
PORT ( clk   : IN std_logic;
       reset : IN std_logic;
       d_out : OUT std_logic_vector (7 downto 0) );
END COMPONENT;
       SIGNAL clk   : std_logic;
       SIGNAL reset : std_logic;
       SIGNAL d_out : std_logic_vector (7 downto 0);
BEGIN
       uut : cpu PORT MAP( clk => clk,          -- port map CPU module
                       reset => reset,
                       d_out => d_out );
-- Shortest period : 2001 ps = Highest frequency ; 500 MHz
clk_process : PROCESS         -- Process for Clock generator
BEGIN
       for i in 0 to 600 loop -- generate clock with period of 2ns
```

```
               CLK <= ‘0’;
               wait for 1001 ps;
               CLK <= ‘1’;
               wait for 1000 ps;
end loop;
               wait;
       END PROCESS;
       rst_test : PROCESS     -- Process for Test stimulus
       BEGIN
               reset <= ‘1’;  -- reset goes high for 3.5 ns then goes low
           wait for 3500 ps;
               reset <= ‘0’;
               wait;
       END PROCESS;
END;
```

Timing Diagram

Figure J.1 shows a portion of the timing diagrams obtained by simulating the test program inside the 256 x 8 RAM. This program successfully tests all eleven instructions. Note that PC is the program counter for the test program in the module cpu_rom, and MPC is the microprogram counter for the symbolic program in the memory control module cm.

From figure K.1, we can see that the first instruction executed is LDA. LDA (PC=0) instruction using reference memory 06H, goes through the following subroutines in the symbolic program. FETCH (MPC=1 at t=6ns), branching to MEMREF(MPC=14 at t=12ns), then to LDSTO(MPC=23 at t=14ns), all the way through LOAD (MPC = 27 at t=22ns), and back to FETCH (Figure K.1). Next, ADD (PC=2) operation is performed using reference memory 06H. At this point, ADD goes through the following subroutines in the symbolic program: FETCH (MPC=1 at t=28ns), branching to MEMREF(MPC=14 at t=34ns), then to ADDSUB(MPC=32 at t=38ns), all the way through ADD (MPC=37 at t=48ns), then back to FETCH. At this point, the ALU generates the result with a carry. Hence, the carry flag becomes high (Figure J.1).

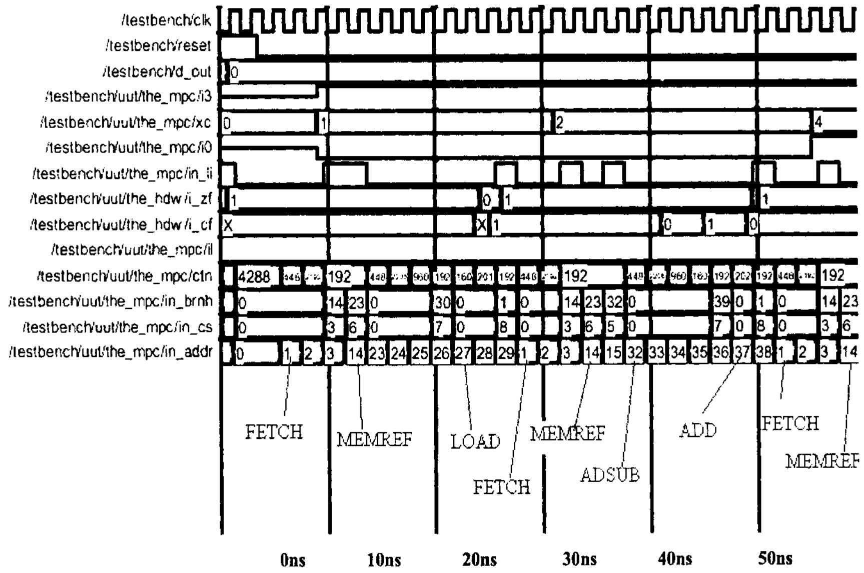
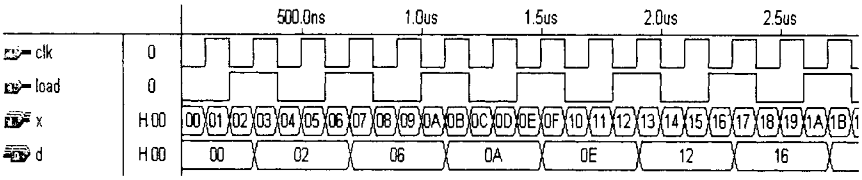


Figure J.1 VHDL Timing Diagram (Top diagram-testbench clock, Next-reset,

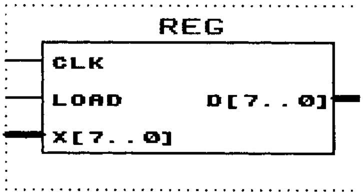
Next-cpu data_out, 8th from top-Zflag, 9th from top Carry flag, Bottom-mpc)
Several modules in the VHDL code are individually simulated for the CPU shown above. The simulation result of each module along with the corresponding block diagram is provided below:

REGISTER

- Simulation result:

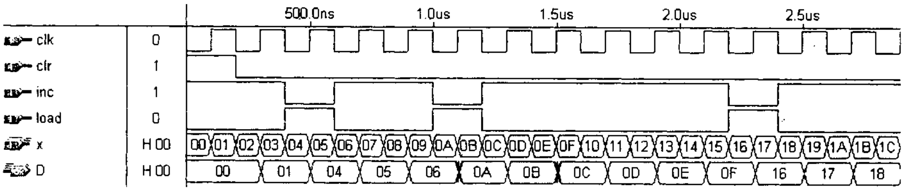


- Block diagram:

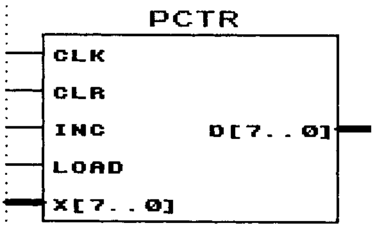


PROGRAM COUNTER

- Simulation result:

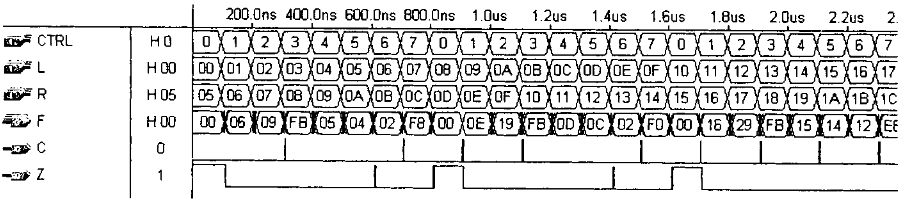


- Block diagram:

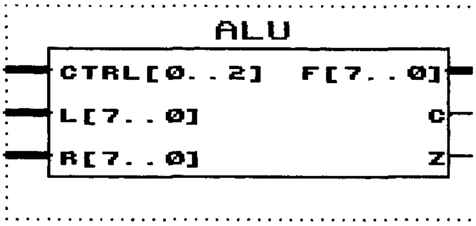


ALU

- Simulation result:

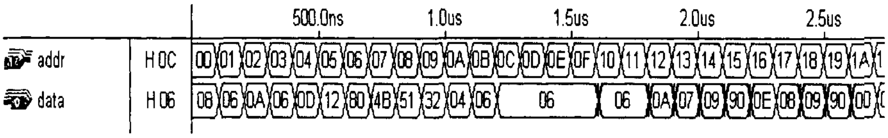


- Block diagram:

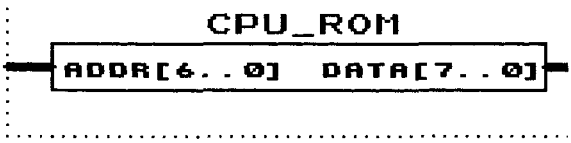


ROM

- Simulation result:

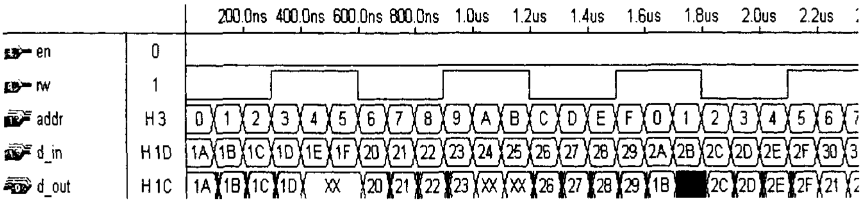


- Block diagram

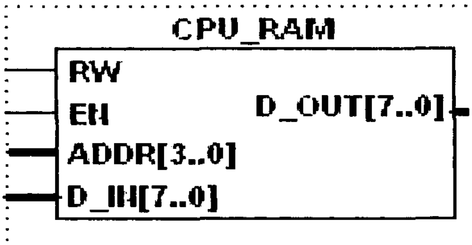


RAM

- Simulation result:

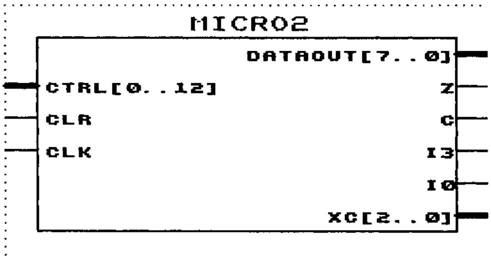


- Block diagram



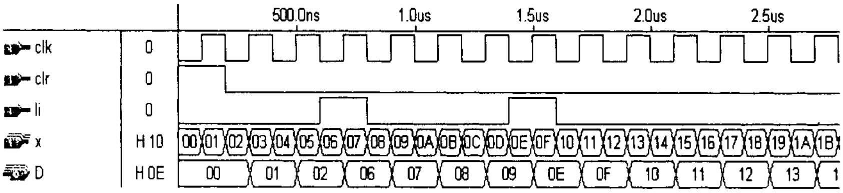
MICRO2

- Black diagram:

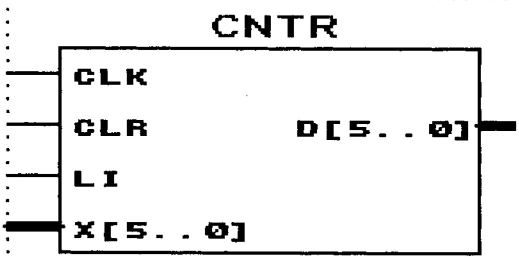


MICROPROGRAM COUNTER

- Simulation result:

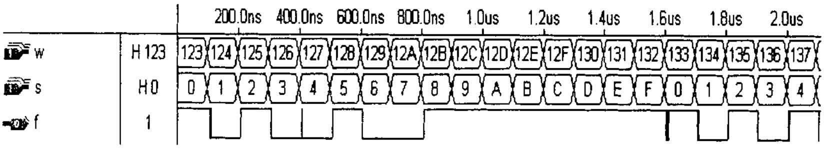


- Block diagram:

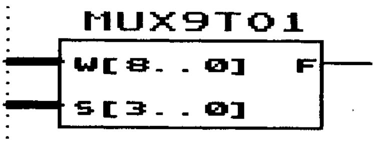


MUX 9 TO1

- Simulation result:

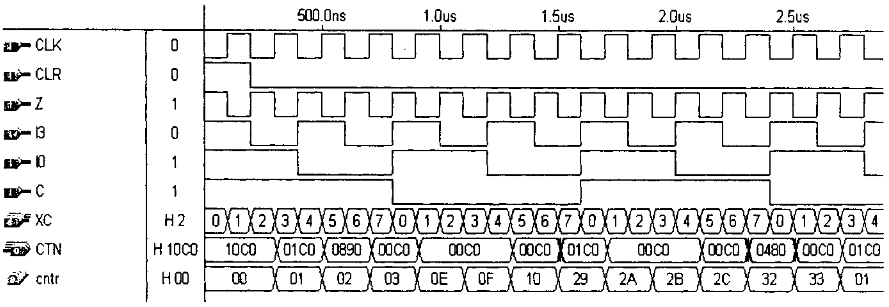


- Block diagram:

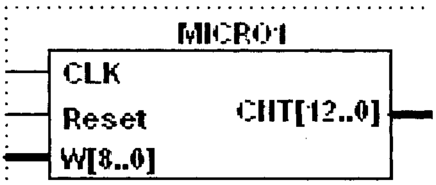


MICRO1

- Simulation result:



- Block diagram:



QUESTIONS AND PROBLEMS

J.1 Write a VHDL description for each of the following using modeling description of your choice:

(a) a 2-to-4 decoder, generating a low output when selected by a high enable.

(b) a 3-to-8 decoder, generating a high output when selected by a high enable.

(c) the 4 -to-16 decoder of Problem 4.15.

(d) a 4-to-1 multiplexer.

(e) a BCD to seven-segment converter for a common cathode display.

(f) the 2-bit unsigned comparator of Section 4.5.2.

J.2 Write a VHDL description for:

(a) the SR latch of Figure 5.1.

(b) the gated D flip-flop of Figure 5.5a.

(c) a D flip-flop with a synchronous reset input and a positive edge triggered clock. Use synchronous reset such that if reset ==0, the flip-flop is cleared to 0; on the other hand, if reset==1, the output of the flip-flop is unchanged until the procedural statements are evaluated at the positive edge of the clock.

(d) the T flip-flop (using D-ff and XOR gate) of Problem 5.13(b).

(e) the state machine of Problem 5.19.

(f) the counters of Problems 5.24(a) through 5.24(c).

(g) the general purpose register of Problem 5.25.

J.3 Write a VHDL description for an 8-bit register with a clear input. If clear is low, the register is loaded with 0. On the other hand, if clear is high, an 8-bit data is transferred to the register at the positive edge of the clock. Use behavioral modeling.

J.4 Write a VHDL description for the Status register of Example 6.1 using behavioral modeling.

J.5 Write a VHDL description for the four-bit by four-bit unsigned multiplier (repeated addition) using:
(a) Hardwired control (Section 7.3.5.2).
(b) Microprogramming (Section 7.3.5.3).