Computer Architecture Lab Manual

Experiment 1: Data representation

Objective:

1. To illustrate the concept of data representation

Theory:

Data representation: It refers to the format of data that are stored in the memory, processed and transmitted. In computer, data are stored in digital formats that can be handled by electronic circuitry. There are various forms of representing data, some of them are:

a) Binary: Binary number system uses 1's and 0's called bits. It is also called a positional natation. E.g.: 10011, 11000
b) Hexadecimal:
c) Octal:
d) 1's and 2's complement:

Algorithm:

```
i. Start
ii. Initialize a function to find binary 1's and 2's complement, hexadecimal, octal
    equivalent of number
iii. Input a number to be converted
iv. Find the equivalent numbers
v. Print the binary 1's and 2's complement, hexadecimal and octal equivalent
   numbers
vi. Stop
```

Source code:

```
#include <stdio.h>
#include <conio.h>
void binary (int );
int main()
{
int n;
printf("enter a number");
scanf("%d",&n);
printf("\n the hex equivalent is %x",
n);
printf("\n the octal equivalent is %o",
n);
printf("\n the binary equivalent is:");
binary(n);
printf("\n the 1's complement is:");
binary(-n-1);
printf("\n the 2's complement
is:");
binary(-n);
return 0;
}

void binary (int a)
{
int i, k;
for (i=3;i>=0;i--)
{
k=(a>>i)&1;
printf("%d", k);
}
```

Experiment 2: Data overflow

Objective:

1. To understand the concept of data overflow

Theory:

In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is not in the range that can be represented with a given number of digits- either too larger than the maximum or lower than the minimum representable value.

Condition for overflow:

```
((AS==BS) &(AS==RS) || (AS! =BS))
        {
                NO overflow, display result;
        }
```

Algorithm:

i. Start
ii. Observe carry into the sign bit position & carry out the sign bit position
iii. If the two carry aren't equal, overflow should be detected
iv. If the two carry are applied to an X-OR gate overflow will be detected when output of gate is 1.
v. Stop

Source code:

```
#include <stdio.h>
int main()
{
 int a,b,r,as,bs,rs;
        printf("enter the two numbers a and b");
        scanf("%d%d", &a,&b);
        r=a+b;
        as=(a>>3)&1;
        bs=(b>>3)&1;
        rs=(r>>3)&1;
        if(((as==bs)&&(as==rs))||(as!=bs))
        {
                printf("\nresult=%d",r);
        }
        else
        {
                printf("\n overflow detected\t the result is:%d",r);
        }
        }
```

Experiment 3: Introduction to VHDL

Objective:

1. To implement the Basic gates and universal gates using VHDL

Theory:

VHDL (VHSIC Hardware Description Language) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general purpose parallel programming language.

VHDL is commonly used to write text models that describe a logic circuit. Such a model is processed by a synthesis program, only if it is part of the logic design. A simulation program is used to test the logic design using simulation models to represent the logic circuits that interface to the design. This collection of simulation models is commonly called a ***testbench.***

A VHDL simulator is typically an event-driven simulator. The simulation alters between two modes: statement execution, where triggered statements are evaluated, and event processing, where events in the queue are processed.

VHDL has constructs to handle the parallelism inherent in hardware designs, but these constructs (*processes*) differ in syntax from the parallel constructs in Ada (*tasks*). Like Ada, VHDL is strongly typed and is not case sensitive.

In VHDL, entity is used to describe a hardware module. An entity can be described using:

i. Entity declaration
ii. Architecture
iii. Configuration
iv. Package declaration
v. Package body

Entity Declaration:

It defines the name, input/output signals and modes of hardware module.

Syntax:

```
entity entity_name is
        port declaration;
end entity_name;
```

Architecture:

Can be described using structural, data flow, behavior or mixed type.

Syntax:

```
architecture architecture_name of entity_name is
architecture architecture_declarative part;
begin
statements;
end architecture_name;
logic operation: NAND gate
```

Architecture Program:

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

entity And_gate is
   Port ( A : in STD_LOGIC;
          B : in STD_LOGIC;
          y : out STD_LOGIC);
end And_gate;

architecture Behavioral of And_gate is
begin
y <= A and B;

end Behavioral;
```

Test bench:

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

ENTITY AND_test IS
END AND_test;

ARCHITECTURE behavior OF AND_test IS

   -- Component Declaration for the Unit Under Test (UUT)

   COMPONENT And_Gate
   PORT(
       A : IN  std_logic;
       B : IN  std_logic;
       y : OUT  std_logic
      );
   END COMPONENT;

  --Inputs
  signal A : std_logic := '0';
  signal B : std_logic := '0';

         --Outputs
  signal y : std_logic;
BEGIN

         -- Instantiate the Unit Under Test (UUT)
  uut: And_Gate PORT MAP (
        A => A,
        B => B,
        y => y
      );
```

```
        A_process: PROCESS
        BEGIN
         A <= NOT A;
         wait for 25 ns;
         END PROCESS;
         B_PROCESS:PROCESS
         BEGIN
         B<=NOT B;
         wait for 75 ns;
         END PROCESS;

END;
```

Experiment 4: 4-bit parallel adder

Objective:

1. To implement the 4-bit parallel adder using VHDL

Theory:

Parallel adder is a digital circuit capable of finding the arithmetic sum of two binary numbers that is greater than one bit in length by operating on corresponding pairs of bits in parallel. It consists of full adder connected in a chain where the output carry from each full adder is connected to the carry input of the next higher order full adder in a chain. A n-bit parallel adder requires n-full adder to perform the operation.

Architecture:

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

entity parallel_adder is
port(   A,B: in STD_LOGIC_VECTOR(3 downto 0);
                S: out STD_LOGIC_VECTOR(3 downto 0);
                Cin: in STD_LOGIC;
                Cout: out STD_LOGIC
        );
end parallel_adder;

architecture Behavioral of parallel_adder is
begin
process (A,B,Cin)
variable t: STD_LOGIC;
begin
t :=Cin;
for i in 0 to 3 loop
S(i)<=A(i) Xor B(i) Xor t;
t := (A(i) and B(i)) or (t and A(i) or (t and B(i)));
```

```
          end loop;
          Cout<=t;
          end process;
          end Behavioral;
```

TestBench:

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

entity tb_paraller_adder is
end tb_paraller_adder;

architecture Behavioral of tb_paraller_adder is
component parallel_adder
Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
         B : in STD_LOGIC_VECTOR (3 downto 0);
         S : out STD_LOGIC_VECTOR (3 downto 0);
         Cin : in STD_LOGIC;
         Cout : out STD_LOGIC);
end component;
--input signal
signal A,B:STD_LOGIC_VECTOR (3 downto 0):= (others => '0');
signal Cin:STD_LOGIC;
--output signal
signal S:STD_LOGIC_VECTOR (3 downto 0):= (others => '0');
signal Cout:STD_LOGIC;
begin
-- Instantiate the Unit Under Test (UUT)
UUT: parallel_adder port map(
A=>A,B=>B,S=>S,Cin=>Cin,Cout=>Cout);
tb: process
    begin
      A<="0101";
      B<="0100";
      Cin<='0';
      wait for 20 ns;
           A<="1101";
           B<="0011";
           Cin<='1';
           wait for 20 ns;
                A<="1100";
                B<="1001";
                Cin<='1';
                wait for 20 ns;
      wait;
end process tb;
end Behavioral;
```

Experiment 5: 3-Segment Pipeline

Objective:

1. To implement 3-Segment Pipeline using VHDL

Theory:

Y= (a+b)*c

Architecture:

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

entity pipeline is
   Port ( a : in integer;
          b : in integer;
          c : in integer;
          clk : in STD_LOGIC;
          y : out integer);
end pipeline;

architecture Behavioral of pipeline is
signal r1,r2,r3,r4 : integer :=0;
begin
y<=r6;
process(clk)
begin
if(rising_edge(clk)) then
   for i in 0 to 2 loop
   case(i) is
   when 0=>
   r1<=a;
   r2<=b;
   r3<=c;
   when 1=>
   r4<=r1+r2;
   when 2=>
   r5<=r4*r3;
   when others=>
   null;
   end case;
   end loop;
end if;
end process;
end Behavioral;
```

Testbench:

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tb_pipeline is
end tb_pipeline;

architecture Behavioral of tb_pipeline is
component pipeline
Port ( a,b,c : in integer;
        clk : in STD_LOGIC;
        y : out integer);
end component;
--input/output signals
signal a,b,c,y: integer;
signal clk: std_logic;
--constant clk_period: time:=2 ns;
begin
uut: pipeline port map(a=>a,b=>b,c=>c,y=>y,clk=>clk);
clock_process :process
begin
   clk <= '0';
   wait for 10 ns;
   clk <= '1';
   wait for 10 ns;
end process;
tb: process
  begin
  a<=2;
  b<=3;
  c<=1;
  wait for 10 ns;
  a<=3;
  b<=4;
  c<=1;
  wait for 10 ns;
  a<=2;
  b<=3;
  c<=1;
  wait for 10 ns;
  a<=5;
  b<=3;
  c<=4;
  wait for 10 ns;
wait;
end process tb;
end Behavioral;
```