

# Mano's Micro

N. Narasimhamurthi

<b>INTRODUCTION.....</b>	<b>1</b>
<b>BASIC COMPUTER.....</b>	<b>1</b>
ARCHITECTURE.....	1
Registers.....	1
Memory.....	2
Input/Output.....	2
Interrupt Structure.....	2
Instruction Set.....	2
RTL DESCRIPTION.....	3
<b>VHDL IMPLEMENTATION.....</b>	<b>3</b>
MAKING LIFE EASY WITH PACKAGES.....	3
Listing of <i>datatypes.vhd</i> .....	3
Listing of <i>defines.vhd</i> .....	4
GETTING STARTED.....	4
Decoders.....	5
Listing of <i>dec1x2e</i> (decoder with enable).....	5
Listing of <i>dec1x2</i> (decoder without enable).....	5
Listing of <i>dec2x4e</i> .....	5
Listing of <i>dec2x4</i> .....	6
Listing of <i>dec3x8.vhd</i> and <i>dec4x16.vhd</i> .....	7
Flipflops.....	8
Listing of <i>Dfflop.vhd</i> and <i>Jkfflop.vhd</i> .....	8
Registers.....	9
Listing of <i>reg1.vhd</i> .....	9
Listing of <i>regn.vhd</i> .....	10
Listing of <i>reg16.vhd</i> and <i>reg12.vhd</i> files.....	11
ALU FUNCTIONS.....	12
Listings of various ALU function files.....	12
Logic function (bitwise and operation).....	14
ALU UNIT.....	15
Listing of <i>alu.vhd</i> .....	15
TIMER DEVICE.....	16
MEMORY.....	17
Rom implementation: Listing of <i>rom.vhd</i> .....	18
RAM implementation: Listing of <i>ram.vhd</i> .....	19
Memory module: Listing of <i>mem.vhd</i> .....	19
BUS ARCHITECTURE.....	20
Listing of <i>buslines.vhd</i> .....	20
ARCHITECTURE OF THE BASIC COMPUTER.....	21
Listing of <i>struct.vhd</i> .....	21
CONTROL UNIT.....	22
Listing of <i>controller.vhd</i> .....	23
COLLECTING ALL THE COMPONENT DECLARATIONS.....	25
Listing of <i>Devices.vhd</i> .....	25
COMPLETE COMPUTER.....	29
Display of <i>mano1.gdf</i> .....	30
SIMULATING THE COMPUTER.....	30

**EXERCISES:..... 31**

# Mano's Micro

N. Narasimhamurthi

---

## ***Introduction***

This document describes a VHDL implementation of the Microprocessor described in Chapter 5 of Computer System Architecture (Third edition) by Morris Mano. It is assumed that the reader is familiar with the basic computer and design based on RTL description. However a brief overview of the computer and the RTL method is presented below.

---

## ***Basic Computer***

### ***Architecture***

The basic computer is a 16-bit machine, i.e. the basic unit is a 16-bit word. The memory size is 4096 words, and is addressed by a 12-bit address. The unit of memory is a 16-bit word. It has a multiplexed bus, with the following registers connected to the bus (the number denotes the bus address of the register):

### ***Registers***

1. AR: Address register. This register holds the memory address. Only the least-significant 12 bits are significant. The output of the register is directly connected to the address lines of memory.
2. PC: Program counter. Only the least significant 12 bits are significant.
3. DR: Data register. This register holds data for ALU operation. The output of the register is directly connected to the ALU on data line 0.
4. AC: The accumulator. The output of the register is directly connected to ALU data line 1.
5. IR: The instruction register. This register holds the current instruction
6. TR: Temporary register. Holds temporary data.

The inputs of these registers, except the AC, are connected to the bus. AC receives its input from the ALU. Each of these registers have the following control lines: LD, INR, and CLR. LD will load the data into the register, INR will increment the register, and CLR will clear the register.

## *Memory*

Memory has 16 bit input data lines connected to the bus, and 16 bit output lines connected at bus address 7. The memory is 4096 words long and is word addressed. The address lines are 12 bits wide ( $2^{12}=4096$ ) and is connected to the address register AR.

## *Input/Output*

Input/output is performed via two 8 bit registers, INPR and OTR. INPR is connected to the ALU, while the input lines of OTR are connected to the bus. I/O is request based. When an input character is available, a flip-flop FGI is to be turned on by the i/o unit. This flag will be turned off by the CPU when the input is read. Similarly, when the output is written to OTR, a flip-flop FGO will be turned off by the CPU. This flop-flop can be turned on by external hardware.

## *Interrupt Structure*

The CPU has IEN flip-flop that enables interrupts. When an interrupt is received, the CPU stores the current PC at location 0 and jumps to location 1. Typically there will be an unconditional branch to the service routine. The routine will branch back to the address in location 0. A flip flop R is used to initiate interrupt service. It is turned on when the interrupt is acknowledged and is turned off once the PC has been saved and the PC has been set to 1.

## *Instruction Set*

The opcode consists of 4 bits while the remaining 12 bits are used either to address memory or encode inherent (non-memory) instructions.

The instruction sets consists of 6 memory addressed instructions. The most significant bit determines the type of addressing, either direct or indirect. The other 3 bits determine the instruction as follows:

000:	AND
001:	ADD
010:	LDA
011:	STA
100:	BUN (Branch unconditional)
101:	BSA (Branch and save return address)
110:	ISZ (Increment memory and skip next instruction if zero)

## RTL Description

One way to describe and synthesize the machine is to describe the action of the computer at each clock cycle. For example, consider the LDA instruction. Here is the sequence of events

T0	Transfer PC to AR	AR <- PC
T1	Fetch instruction and increment PC	IR <-Memory, PC <- PC+1
T2:	Decode and xfer address	AR <= IR(11 -- 0)
T3	Indirection if needed	AR <= Memory, if IR(15) =1
T4:	Fetch data	DR <- Memory
T5:	Move to AC	AC <-DR, DONE

Note based on the above timing information, we first generate a sequencer which will have lines T0, T1, T2, etc. A counter (register) connected to a decoder generates this sequencer. The control module will then turn on the following signals:

T0	Connect PC to BUS, Turn on LD line of AR
T1	Connect Mem to Bus, Turn on LD line of AR and INC line of PC
T2:	Connect IR to bus, Turn on LD line of AR
T3	If I(15) connect Mem to Bus and turn on LD line of AR
T4:	Connect Mem to Bus, Turn on LD line of DR
T5:	Set ALU to pass through and turn on LD line of AC, Turn on CLR line of Sequence counter

---

## VHDL implementation

### Making life easy with packages

It gets to be both tedious and error prone to remember the bus address of the devices, the fact that they 16 bit vectors etc. So it is best to define them in a package and then include the package in all the VHDL files. We will define three packages: **defines.vhd**, **datatypes.vhd**, and **devices.vhd**, to keep track of definitions, datatypes and various devices. Here is a listing of the files (These files were updated as the design proceeded so some of the entries may not make sense at this stage). Note the use of subtypes to define user specific types. For example we can use **Tword** where ever **std\_logic\_vector(15 downto 0)** can be used. The **memUnitSize** will be explained later.

#### Listing of datatypes.vhd

```
library ieee; USE ieee.std_logic_1164.all;
-- Define the basic data types and constants
-- N. Narasimhamurthi
-- $Id: dataty~1.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $

PACKAGE datatypes IS
  -- Constants
  constant wordsize : integer := 16;
```

```

        constant adrsiz: integer := 12;
        constant memUnitSize: integer := 6;
    -- SubType Declaration
        SUBTYPE Tword IS std_logic_vector(wordsiz-1 downto 0);
        SUBTYPE Taddress IS std_logic_vector(adrsiz-1 downto 0);
        SUBTYPE TMemAddress IS std_logic_vector(memUnitSize-1 downto 0);
    -- Special constants
        constant zeroword : Tword := "0000000000000000";
        constant BusDefault: Tword := "1111111111111111";

end datatypes;

```

### *Listing of defines.vhd*

```

library ieee; USE ieee.std_logic_1164.all;

--Author: N. Narasimhamurthi
--
-- Defines constants used for ALU and BUS operations
--
--$Id: defines.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $
PACKAGE defines IS
    -- Subtypes
        subtype aluinst is std_logic_vector(2 downto 0);
        subtype busctrl is std_logic_vector(2 downto 0);
    -- Constants
        -- ALU instructions
        constant aluPass: aluinst := "000";
        constant aluAND:  aluinst := "001";
        constant aluADD:  aluinst := "010";
        constant aluCOM:  aluinst := "011";
        constant aluSHR:  aluinst := "100";
        constant aluSHL:  aluinst := "101";
    -- Bus select codes
        constant BusAR: busctrl := "001";
        constant BusPC: busctrl := "010";
        constant BusDR: busctrl := "011";
        constant BusAC: busctrl := "100";
        constant BusIR: busctrl := "101";
        constant BusTR: busctrl := "110";
        constant BusMem: busctrl := "111";

end defines;

```

## **Getting Started**

First we define a few basic devices. These serve two primary purposes, one gets us going, and more importantly everything we need is under our control (the only exception is when we define RAM). We start with a few devices from chapter 2 of Mano's book. First we start with 1x2 decoder with or without enable line. We use these to build bigger decoders as described in Mano's book.

## Decoders

### *Listing of dec1x2e (decoder with enable)*

```
--1 by 2 decoder with enable
-- Author: N. Narasimhamurthi
--
-- $Id: dec1x2e.vhd 1.1 10/0/3 .2:.0:4 unknown Exp Locker: unknown $
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity dec1x2e is
    port(
        sel: in std_logic;
        en: in std_logic;
        o: out std_logic_vector(1 downto 0)
    );
end dec1x2e ;

architecture a of dec1x2e is
begin
    o(0) <= (not sel) and en;
    o(1) <= sel and en;
end a;
```

### *Listing of dec1x2 (decoder without enable)*

```
--1 by 2 decoder (no enable line)
-- Author: N. Narasimhamurthi
-- This is the standard 1x2 decoder
-- $Id: dec1x2.vhd 1.1 10/0/3 .2:.0:4 unknown Exp Locker: unknown $
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity dec1x2 is
    port(
        sel: in std_logic;
        o: out std_logic_vector(1 downto 0)
    );
end dec1x2 ;

architecture a of dec1x2 is
begin
    o(0) <= (not sel);
    o(1) <= sel;
end a;
```

### *Listing of dec2x4e*

```
--2 by 4 decoder with enable
-- Author: N. Narasimhamurthi
--
-- Uses three 1x2e decoder (with enable)
-- The high bit is passed to a 1x2e decoder and the output
```

```

-- selects one of the other two 1x2e decoder.
-- $Id: dec2x4e.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity dec2x4e is
    port(
        sel: in std_logic_vector(1 downto 0);
        en: in std_logic;
        o: out std_logic_vector(3 downto 0)
    );
end dec2x4e ;

architecture a of dec2x4e is
    signal chipsel: std_logic_vector(1 downto 0);
begin
    sel1: dec1x2e port map(sel(1), en, chipsel);
    chip0: dec1x2e port map(sel(0), chipsel(0), o(1 downto 0));
    chip1: dec1x2e port map(sel(0), chipsel(1), o(3 downto 2));
end a;

```

#### *Listing of dec2x4*

```

-- 2 by 4 decoder (no enable)
-- Author: N. Narasimhamurthi
--
-- Principle of operation
-- We obtain 2x4 decoder by using two 1x2e decoder with enable and a
-- 1x2 decoder
-- The high bit is passed to the 1x2 decoder. The output is used to
-- enable one of the other two 1x2e decoder
--
-- $Id: dec2x4.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity dec2x4 is
    port(
        sel: in std_logic_vector(1 downto 0);
        o: out std_logic_vector(3 downto 0)
    );
end dec2x4 ;

architecture a of dec2x4 is
    signal chipsel: std_logic_vector(1 downto 0);
    signal one: std_logic;

begin
    sel1: dec1x2 port map(sel(1), chipsel);
    chip0: dec1x2e port map(sel(0), chipsel(0), o(1 downto 0));
    chip1: dec1x2e port map(sel(0), chipsel(1), o(3 downto 2));
end a;

```

*Listing of dec3x8.vhd and dec4x16.vhd*

```
-- 3 by 8 decoder
-- Author: N. Narasimhamurthi
--
-- The device consists of two 2x4e decoders and a 1x2 decoder
-- The 1x2 decoder selects one of the other two based on the
-- high bit
-- The other two decoders receive the low two bits
--
-- $Id: dec3x8.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity dec3x8 is
    port(
        sel: in std_logic_vector(2 downto 0);
        o: out std_logic_vector(7 downto 0)
    );
end dec3x8;

architecture a of dec3x8 is
    signal chipsel: std_logic_vector(1 downto 0);
begin
    sel1: dec1x2 port map(sel(2), chipsel);
    chip0: dec2x4e port map(sel(1 downto 0), chipsel(0), o(3 downto 0));
    chip1: dec2x4e port map(sel(1 downto 0), chipsel(1), o(7 downto 4));
end a;

-- 4 by 16 decoder -- Author: N. Narasimhamurthi
--
-- We build a 4x16 decoder by first using a 2x4 decoder on the
-- high two bits; The four output lines of this decoder is used
-- to select one of four 2x4e decoders, each of which receive the low
-- two bits
--
-- $Id: dec4x16.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
--
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity dec4x16 is
    port(
        sel: in std_logic_vector(3 downto 0);
        o: out std_logic_vector(15 downto 0)
    );
end dec4x16;

architecture a of dec4x16 is
    signal chipsel: std_logic_vector(3 downto 0);
begin
    sel1: dec2x4 port map(sel(3 downto 2), chipsel);
    chip0: dec2x4e port map(sel(1 downto 0), chipsel(0), o(3 downto 0));
    chip1: dec2x4e port map(sel(1 downto 0), chipsel(1), o(7 downto 4));
    chip2: dec2x4e port map(sel(1 downto 0), chipsel(2), o(11 downto 8));
    chip3: dec2x4e port map(sel(1 downto 0), chipsel(3), o(15 downto 12));
```

```
end a;
```

## Flipflops

We define the primitive D Flipflop and the JK flipflop

### Listing of Dfflop.vhd and Jkfflop.vhd

```
--D flip-flop
-- Author: N. Narasimhamurthi
--
-- $Id: dfflop.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
--
-- Component type: Generic (not specific to the problem)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dfflop IS
    PORT
    (
        D      : IN STD_LOGIC;
        clk    : IN STD_LOGIC;
        Q      : OUT STD_LOGIC
    );
end dfflop;

architecture a OF dfflop IS
    SIGNAL Q_signal : STD_LOGIC;
begin
    process (clk)
    begin
        if (clk'event AND clk = '1') then
            Q_signal <= D;
        end if;
    end process;
    Q <= Q_signal;
end a;

--JK flip-flop
-- Author: N. Narasimhamurthi
--
-- Standard JK flip flop
--
-- $Id: jkfflop.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
-- Component type: Generic (not specific to the problem)

LIBRARY ieee; USE ieee.std_logic_1164.all;

library work; use work.devices.all;

entity jkfflop is
    port(
```

```

        J,K      : IN STD_LOGIC;
        clk     : IN STD_LOGIC;
        Q,Qbar  : OUT  STD_LOGIC);
end jkfflop;

architecture a of jkfflop is

    signal qs, notqs, ds: std_logic;
begin
    notqs <= not qs;
    Q <= qs; Qbar <= notqs;
    ds <= (J and notqs) or (not K and qs);
    df: dfflop port map(D=>ds, clk=>clk, Q=>qs);
end a;

```

## Registers

First we define a 1 bit register using the JK flipflop. For details of this implementation, refer to Mano's book.

### Listing of reg1.vhd

```

-- One bit generic register with clr, ld, and data lines
-- Uses JK flipflop as the basic unit
-- See Mano's text book for details
-- N. Narasimhamurthi
--
-- $Id: reg1.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.devices.all;

entity reg1 is
    port
    (
        cin: in std_logic;
        data: in std_logic;

        clr: in std_logic;
        ld: in std_logic;

        clk: in std_logic;

        q, qbar: out std_logic;
        cout: out std_logic);
end reg1;

architecture a of reg1 is
    signal js, ks, qs, qbars: std_logic;
begin
    js <= cin or (data and ld);
    ks <= cin or (not data and ld) or clr;

    ff1: jkfflop port map (js, ks, clk, qs, qbars);

```

```

        q <= qs;
        qbar <= qbars;
        cout <=qs and cin;

end a;

```

We can now define a n-bit register. We shall use two important features of VHDL. One is generic declaration. This allows us to generalize the design for any n. Secondly, we use a for loop to generate the sequence of n 1-bit registers connected together. Also, we can provide a default value for n.

*Listing of regn.vhd*

```

-- General purpose n bit register
--
-- Note that we string n 1-bit registers in series
-- These registers are generated in a for-generate loop
--
-- N. Narasimhamurthi
--
-- $Id: regn.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $

LIBRARY ieee;
USE ieee.std_logic_1164.all;

library work; use work.devices.all;

entity regn is
    generic (n: integer := 16);

    port
    (
        inc: in std_logic;
        data: in std_logic_vector(n-1 downto 0);

        clr: in std_logic;
        load: in std_logic;

        clk: in std_logic;

        q, qbar: out std_logic_vector(n-1 downto 0);
        cout: out std_logic);
end regn;

architecture a of regn is
    signal coS: std_logic_vector(n-2 downto 0);
    signal cin, ld: std_logic;
begin
    cin <= inc and not load and not clr;
    ld <= load and not clr;

    r0: reg1 port map (cin, data(0), clr,ld, clk, q(0), qbar(0), coS(0));
    gen_loop: for i in 1 to n-2 generate
        ri: reg1 port map (coS(i-1), data(i), clr,ld, clk, q(i), qbar(i), coS(i));
    end generate;
    rlast: reg1 port map (coS(n-2), data(n-1), clr,ld, clk, q(n-1), qbar(n-1), cout);
end a;

```

A few remarks on the for generate loop. The entire loop should have a label. We have used **gen\_loop**. Also each instance of an element inside the loop must have a label (this is true even if the instance is not inside a loop). We have used **r0**, **ri**, and **rlast** for the three labels.

The above is a general purpose register with both **q**, **notq**, and **cout** outputs. For our purposes, these outputs are not needed. Rather than clutter our design with extra unwanted signals, we can define a specific 16 bit register. To keep our design simple, we shall also define a 12 bit register as a 16 bit register with leading four bits always zero (this will make connecting these simpler). Also, note the use of **generic map** which is similar to **port map**.

*Listing of reg16.vhd and reg12.vhd files*

```

-- General purpose 16 bit register
-- This is a special case of generic n-bit register
-- Only q is available at the output
-- Author: N. Narasimhamurthi
--
-- $Id: reg16.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

library work;
use work.devices.all;

entity reg16 is
  port
  (
    inc: in std_logic;
    data: in std_logic_vector(15 downto 0);

    clr: in std_logic;
    load: in std_logic;

    clk: in std_logic;

    q: out std_logic_vector(15 downto 0)
  );
end reg16;

architecture a of reg16 is
  signal qbar: std_logic_vector(15 downto 0);
  signal cout: std_logic;
begin
  chip: regn generic map(n=>16)
  port map(inc, data, clr, load, clk, q, qbar, cout);
end a;

-- 12 bit registers with 16 bit i/o lines
-- Leading 4 bits are ignored on input
-- and padded with zero at the output
-- Author: N. Narasimhamurthi

```

```

--
-- $Id: reg12.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $
--
-- Component type: Specific to 12 bit addressing

LIBRARY ieee;
USE ieee.std_logic_1164.all;

library work;
use work.devices.all;

entity reg12 is
  port
  (
    inc: in std_logic;
    data: in std_logic_vector(15 downto 0);

    clr: in std_logic;
    load: in std_logic;

    clk: in std_logic;

    q: out std_logic_vector(15 downto 0)
  );
end reg12;

architecture a of reg12 is

  signal qs: std_logic_vector(11 downto 0);
begin

  chip: regn generic map(n=>12)
    port map (inc, data(11 downto 0), clr, load, clk, qs);
  q <= "0000" & qs;
end a;

```

Some points to note. We could have written **generic map(16)** instead of **generic map(n=>16)**. However, it is a good idea to explicitly show what the generic parameter that gets mapped. This is especially true if the device has several generic parameters.

## ***ALU functions***

We next define a number of ALU functions such as 16 bit adders, 16 bit , term by term and functions etc. Note that we design our own and gate and use 16 of them to perform 16 bit and operation.

### *Listings of various ALU function files*

```

-- One bit half adder
-- Author: N. Narasimhamurthi
--
-- $Id: hadder.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $
--
-- Component type: Generic (not specific to the problem)

```

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

```
entity hadder is  
  port(  
    x,y: in std_logic;  
    s,c: out std_logic);  
end hadder;
```

```
ARCHITECTURE a OF hadder IS  
BEGIN  
  s <= x xor y;  
  c <= x and y;  
END a;
```

```
-- One bit full adder  
-- N. Narasimhamurthi  
--  
-- $Id: fa1.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
```

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

```
entity fa1 is  
  port(  
    x,y,z: in std_logic;  
    s,c: out std_logic);  
end fa1;
```

```
ARCHITECTURE a OF fa1 IS  
  SIGNAL xeory: STD_LOGIC;  
BEGIN  
  xeory <= x xor y;  
  s <= xeory xor z;  
  c <= (x and y) or (xeory and z);  
END a;
```

```
-- n-bit full adder with no initial carry-in  
-- Built using an half adder and n-1 full adders  
-- The adders are connected in series using for-generate loop  
-- Author: N. Narasimhamurthi  
--  
-- $Id: fan.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $  
--  
-- Component type: Generic (not specific to the problem)
```

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

```
library work;  
use work.devices.all;
```

```
entity fan is  
  generic (n: integer := 16);
```

```

        port(
            x,y: in std_logic_vector(n-1 downto 0);
            s: out std_logic_vector(n-1 downto 0);
            cout: out std_logic
        );
end fan;

architecture a of fan is
    signal coS: std_logic_vector(n-2 downto 0);
begin

    a0: hadder port map (x(0), y(0), s(0), coS(0));
    gen_loop: for i in 1 to n-2 generate
        ai: fa1 port map (x(i), y(i), coS(i-1), s(i), coS(i));
    end generate;
    alast: fa1 port map (x(n-1), y(n-1), coS(n-2), s(n-1), cout);
end a;

```

### *Logic function (bitwise and operation)*

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;
-- 1 bit AND gate
-- Author: N. Narasimhamurthi
--
-- Note: VHDL treats and as an operator /function call
-- This generates and gate as a device.
-- This is useful if we want to add an and gate with port map
--
-- $Id: and1bit.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
--
-- Component type: Generic (not specific to the problem)

entity and1bit is
    port(
        x,y: in std_logic;
        z: out std_logic
    );
end and1bit;

architecture a of and1bit is
begin
    z <= x and y;
end a;

```

```

-- An n bit
-- The two n-bit words are added bit by bit
-- N. Narasimhamurthi
--
-- $Id: andnbit.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $

```

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; use work.devices.all;

entity andnbit is
    generic (n: integer := 16);
    port(
        x,y: in std_logic_vector(n-1 downto 0);

```

```

        s: out std_logic_vector(n-1 downto 0)
    );
end andnbit;

architecture a of andnbit is
begin

    gen_loop: for i in 0 to n-1 generate
        chip: and1bit port map (x=>x(i), y=>y(i), z=>s(i));
    end generate;
end a;

```

Note the construct in the **port map**. We have explicitly stating which port is connected which signal. The general syntax is **portname=>signal**.

### **ALU unit**

We can now design the ALU unit. Only a few functions are implemented. The rest can be easily added.

#### *Listing of alu.vhd*

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; USE work.datatypes.all, work.defines.all, work.devices.all;

-- Simple multipurpose ALU with several functions
-- Input fn determines the functionality
-- Functionality codes use named constants defined in defines.vhd
--
-- Author: N. Narasimhamurthi
--
-- $Id: alu.vhd 1.1 10/0/3 .2:0:4 unknown Exp Locker: unknown $
--

entity alu is
port(
    Ein: in std_logic;
    D0, D1: in Tword;
    fn : in aluinst;
    E: out std_logic;
    q: out Tword
);
end alu;

architecture a of alu is
    signal Cout: std_logic;
    signal notD0, and2, add2, ror, rol: Tword;
begin
    chip1: andnbit generic map(n=>16) port map (D0, D1, and2);
    chip2: fan generic map(n=>16) port map(D0, D1, add2, Cout);

    q <= D0 when fn=aluPass
        else and2 when fn=aluAND
        else add2 when fn=aluADD
        else zeroword;
    E <= Cout when fn=aluAdd else

```

```
    '0';  
end a;
```

## Timer device

The sequence timer can be designed using a 4 bit register and a 4x16 decoder. (3 bits are actually sufficient as the basic computer does not require more than 7 clock cycles for the longest instruction). The code is shown below. First we define an n-bit counter and then connect it to a decoder.

```
-- 16 line timer  
-- Author: N. Narasimhamurthi  
--  
-- Principle of operation:  
-- The device consists of a 4 bit register and and 4x16 decoder  
-- The register is configured to be an upcounter except when  
-- a clr signal is received.  
-- I.e the not of clr is sent to the increment line of the register  
-- Component type: Generic (not specific to the problem)  
--  
-- Identification $Id: timer16.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $  
  
LIBRARY ieee; USE ieee.std_logic_1164.all;  
library work; use work.devices.all;  
  
entity timer16 is  
    port(  
        clk: in std_logic;  
        clr: in std_logic;  
        o: out std_logic_vector(15 downto 0)  
    );  
end timer16;  
  
architecture a of timer16 is  
    signal incS, zero, coutS: std_logic;  
    signal dataS, qs, qbars: std_logic_vector(3 downto 0);  
begin  
    zero <= '0'; -- Need this as we can not use constants in port map  
    incS <= not clr;  
    dataS <= "0000";  
    counter: regn  
        generic map( n=>4)  
        port map(  
            inc => incS, data=>dataS, clr=>clr, load=>zero, clk=>clk,  
            q=>qs, qbar=>qbars, cout=>couts);  
    dec: dec4x16 port map(sel=>qs, o);  
end a;
```

## Memory

Memory is usually organized into segments or pages. For example, when you buy a PC, it may come with less memory than you wish to have. Sooner or later, you outgrow the memory, and you go out and buy extra memory. You pop a chip into your motherboard, and almost by magic everything works (most of the time)! How is this done? Typically low order address lines are connected to all the memory units. Each memory unit has a decoder that will enable that particular unit when the addresses are in a certain range. As long as two units do not both get enabled, you get extra memory. Note that you can have gaps in memory. This happens if no unit looks for addresses in that particular range. Also, you can have I/O devices look for certain address ranges. As long as no memory unit decodes those addresses (or if such memory unit exists, they are disabled), then you can have I/O where memory should be. You must have seen this with 68HC11 where memory addresses \$1000 to \$103F actually are I/O ports. We will use the same principle.

Memory is organized in units of 64 words (6 bits of addresses). Thus we have the following memory units:

<b>Address range (binary)</b>	<b>Memory unit</b>
<b>0000 0000 0000 to 0000 0011 1111</b>	Unit 0
<b>0000 0100 0000 to 0000 0111 1111</b>	Unit 1
<b>0000 1000 0000 to 0000 1001 1111</b>	Unit 2
<b>0000 1100 0000 to 0000 1111 1111</b>	Unit 3

Note that the last 6 bits determine the address within the unit and the first bits determine the unit. The memory is essentially 64 units of 64 words each. In our implementation we shall have only two units namely unit 0 and unit 1. Also, we shall use bit #6 to determine the unit number. Thus as far as the hardware is concerned we have only 7 bits of addressing for a total of 128 words of memory. If you try to implement more than two units, you may run out of space in the altera board. Also remember as for as memory units are concerned, there are only 6 address lines!

The unit 0 will be ROM and unit 1 will be RAM. This choice makes power-up initialization simple, as we have to make sure that PC has a ROM address when the computer is started (for example, in 68HC11, the PC gets its value from ROM location \$FFFE). This choice also means that we can not implement interrupt service as the Interrupt service start-up stores the return address at memory location \$00. A more sensible choice will be to place RAM in unit 0 and ROM in unit 1. But then we have to make sure that PC is initialized to a non-zero location. We shall discuss power-up initialization in detail in the class.

ROM will be initialize to the following program. This program loads data from RAM at location \$40, increments the value, outputs the value to the output register, stores the incremented value back to location \$40. It then clears A and then loops back to top of the program. Note by clearing A before looping back,

we make sure that the ACC is really getting its data from memory and not from the value left there in the previous pass through the loop. This program also makes sure that we can read from and write to RAM. Also, the program is simple enough that we can test it with only partial implementation of the basic computer.

Location	Instruction
000	LDA \$40
001	INCA
002	OUT
003	STA \$40
004	CLRA
005	BUN 0
006	BUN 0
007	BUN 0
--	BUN 0

*Rom implementation: Listing of rom.vhd*

```

-- ROM
-- N. Narasimhamurthi
-- Contents of memory are preloaded
--
-- $Id: rom.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $

LIBRARY ieee; USE ieee.std_logic_1164.all;
LIBRARY lpm; USE lpm.lpm_components.all;
LIBRARY work;
USE work.datatypes.all;

ENTITY rom IS
PORT(
    address : IN TMemAddress;
    q       : OUT Tword
);
END rom;

ARCHITECTURE example OF rom IS
BEGIN
    q <= "0010000001000000" when address = "000000" -- LDA $40
        else "0111000000100000" when address = "000001" -- INA
        else "1111010000000000" when address = "000010" -- OUT
        else "0011000001000000" when address = "000011" -- STA $40
        else "0111100000000000" when address = "000100" -- CLA
        else "0100000000000000"; --BUN 0
END example;

```

### *RAM implementation: Listing of ram.vhd*

[Know when to hold up/ Know when to fold up/ Know when to cry uncle]

After several attempts at implementing RAM from registers, I gave up, as I could not get my design to fit inside a flex 10K. So I decided to use Altera's predefined dual clock (one for read and one of write) generic ram device. I copied the code from Altera's tutorial folder and added the my data definitions.

```
-- Ram
-- Author: N. Narasimhamurthi
-- Code taken directly from Altera examples
--
-- Uses lpm library
-- Size and width of RAM defined in datatypes.vhd
-- $Id: ram.vhd 1.1 10/0/3 .2.:0:.4 unknown Exp Locker: unknown $

-- MAX+plus II VHDL Example
-- LPM RAM Instantiation
-- Copyright (c) 1996 Altera Corporation

LIBRARY ieee; USE ieee.std_logic_1164.all;
LIBRARY lpm; USE lpm.lpm_components.all;
LIBRARY work;
USE work.datatypes.all;

ENTITY ram IS
PORT(
    data    :   IN Tword;
    address :   IN TMemAddress;
    we      :   IN STD_LOGIC;
    clock   :   IN STD_LOGIC;
    q       :   OUT Tword
);
END ram;

ARCHITECTURE example OF ram IS
constant addwidth : integer := 6;
BEGIN

inst_1  : LPM_RAM_DQ
        GENERIC MAP (lpm_widthad => memUnitSize,
                    lpm_width => wordsize)
        PORT MAP (data => data, address => address, we => we,
                 inclock => clock, outclock => clock,
                 q => q);

END example;
```

### *Memory module: Listing of mem.vhd*

```
-- Memory
-- Memory is organized in modules.
```

```

-- As implemented there are two modules
-- Module 0: ROM
-- Module 1: RAM
-- Note that rearranging ROM and RAM makes more sense but need
-- initial boot sequence to make sure that PC points to ROM on
-- power-up!
--
-- Author: N. Narasimhamurthi
--
-- $Id: mem.vhd 1.1 10/0/3 .2.:0:4 unknown Exp Locker: unknown $
--
LIBRARY ieee; USE ieee.std_logic_1164.all;
LIBRARY work; USE work.datatypes.all; USE work.devices.all;

ENTITY mem IS
PORT(
    data    : IN Tword;
    address : IN TAddress;
    we      : IN STD_LOGIC;
    clock   : IN STD_LOGIC;
    q       : OUT Tword
);
END mem;

ARCHITECTURE example OF mem IS
    signal ramS, writes: std_logic;
    signal qsram, qsrom: Tword;
BEGIN
    ramS <= address(memUnitSize); -- Determine RAM or ROM
    writes <= we and ramS; -- Enable write only if RAM

    ramchip: ram port map(data, address(memUnitSize-1 downto 0), writes, clock, qsram);
    romchip: rom port map(address(memUnitSize-1 downto 0), qsrom);
    q <= qsram when ramS='1' else qsrom; -- Select the appropriate output
END example;

```

## ***Bus Architecture***

The basic computer uses a simple multiplexed bus. This means that at any given time the bus has a data source. A three-bit select line selects the source. Rather than hard code the selection codes for each of the possible sources, constants defined in the **defines.vhd** package is used for clarity and coordination with other source files.

### *Listing of buslines.vhd*

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; USE work.datatypes.all, work.defines.all;
-- This is the multiplexed bus
-- Basically connects the bus based on the selection
-- The constants BusAR etc are defined in defines.vhd
-- Using named constants improves the readability of the code.
-- Author: N. Narasimhamurthi
-- $Id: buslines.vhd 1.1 10/0/3 .2.:0:4 unknown Exp Locker: unknown $

```

```

entity buslines is
  port(
    Ar, PC, DR, AC, IR, TR, Memory: in Tword;
    sel: in busctrl ;
    q: out Tword
  );
end buslines;

architecture a of buslines is
begin
  q <= Ar when sel = BusAR else
    PC when sel = BusPC else
    DR when sel = BusDR else
    AC when sel = BusAC else
    IR when sel = BusIR else
    TR when sel = BusTR else
    Memory when sel = BusMem else
    BusDefault;
end a;

```

## ***Architecture of the Basic Computer***

We now have all the elements of the basic computer except the control module. We can connect all the pieces to build the machine without the control module. This is done in the file struct.vhd. At the input to the architecture are all the control lines and at the output are the output registers and contents of registers that are needed by the control unit, as well as the output of the flags (flip-flops). Note that it does not matter if we use hardwired control logic as in chapter 5 of Mano's book or micro-programmed control unit as in chapter 7 of the book.

### *Listing of struct.vhd*

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; USE work.datatypes.all, work.defines.all, work.devices.all;

-- Basic Structure of the Mano Machine
-- Defines all the components w/o the control logic
-- N. Narasimhamurthi
-- $Id: struct.vhd 1.1 10/0/3 .2:0:4 unknown Exp Locker: unknown $

entity struct is
port (
  -- Control signals from controller

  Mwrite: in std_logic; -- 1 for memory write
  AluFn: in std_logic_vector(2 downto 0); -- Determines the ALU Function
  BusSel: in std_logic_vector(2 downto 0); -- Determines who connects to the Bus

  ArLd, ArInc, ArClr,           -- Signals for AR
  PcLd, PcInc, PcClr,          -- Signals for PC
  DrLd, DrInc, DrClr,          -- Signals for DR
  AcLd, AcInc, AcClr,          -- Signals for AC
  IrLd, IrInc, IrClr,          -- Signals for IR
  TrLd, TrInc, TrClr,          -- Signals for TR

```

```

OtLd, OtInc, OtClr,                -- Signals of Output Register

RJ, RK,                            -- Signals of R Flipflop
IENJ, IENK,                        -- IEN FF
FGIJ, FGIK,                        -- FGI FF
FGOJ, FGOK,                        -- FGO FF

clk:                                in std_logic;-- Clock

-- Control Signals to controller
IR, DR, AC: out TWord;
R, IEN, FGI, FGO: out std_logic;

-- Other outputs of interest
PC, Ot: out TWord
);
end struct;

architecture a of struct is
    signal busdata, aluout, Memout,
           ArS, PcS, DrS, AcS, IrS, TrS -- Output of the registers
           : Tword;
    signal E, Ein: std_logic; -- This is here temporarily. Should be a flipflop
begin
-- Connect output to internal signals
IR <= IrS;  DR <= DrS; AC <= AcS; PC <= PcS;
-- Flipflops
FFR: JKFFLOP port map(J=>RJ, K=>RK, clk=>clk, Q=>R);
FFIEN: JKFFLOP port map(J=>IENJ, K=>IENK, clk=>clk, Q=>IEN);
FFFGI: JKFFLOP port map(J=>FGIJ, K=>FGIK, clk=>clk, Q=>FGI);
FFFGO: JKFFLOP port map(J=>FGOJ, K=>FGOK, clk=>clk, Q=>FGO);

-- Registers
RegIr: reg16 port map(IrInc, busdata, IrClr, IrLd, clk, IrS);
RegDr: reg16 port map(DrInc, busdata, DrClr, DrLd, clk, DrS);
RegPc: reg12 port map(PcInc, busdata, PcClr, PcLd, clk, PcS);
RegTr: reg16 port map(TrInc, busdata, TrClr, TrLd, clk, TrS);
RegOt: reg16 port map(OtInc, busdata, OtClr, OtLd, clk, Ot);
RegAr: reg12 port map(ArInc, busdata, ArClr, ArLd, clk, ArS);

RegAc: reg16 port map(AcInc, aluout, AcClr, AcLd, clk, AcS);

-- Memory
Memchip: mem port map(busdata, Ars(11 downto 0), Mwrite, clk, Memout);

-- Bus
BusSwitch: buslines port map(ArS, PcS, DrS, AcS, IrS, TrS, Memout, Bussel, busdata);
-- Alu
AluUnit: Alu port map(Ein, DrS, AcS, AluFn, E, aluout);
end a;

```

Note that AR and PC are 12 bit registers while others are 16 bit registers.

### **Control Unit**

Control unit is at the heart of the computer. It should generate appropriate control signals based on the clock sequence and the state of the various registers,

most importantly the instruction register IR. For details on how to build the control unit, refer to chapter 5 in Mano's book. The following does ***not*** implement ***all*** the instructions! Only those that are needed for the program in ROM is implemented.

*Listing of controller.vhd*

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; USE work.datatypes.all, work.devices.all, work.defines.all;
-- This is the controller for Mano's computer
-- Refer to the book for details
-- N. Narasimhamurthi
--
-- $Id: contro~1.vhd 1.1 10/0/3 .2.:0.:4 unknown Exp Locker: unknown $
entity controller is
    port(
        T: in std_logic_vector(15 downto 0);
        IR: in Tword;
        DR: in Tword;
        R,IEN, FGI, FGO: in std_logic;

        MemWr: out std_logic;
        AluFn: out aluinst;
        bussel : out busctrl;

        ArLd, ArInc, ArClr,
        PcLd, PcInc, PcClr,
        DrLd, DrInc, DrClr,
        AcLd, AcInc, AcClr,
        IrLd, IrInc, IrClr,
        TRLd, TrInc, TrClr,
        OtLd, OtInc, OtClr,
        RJ, RK,
        IENJ, IENK,           -- IEN FF
        FGIK,                 -- FGI FF (Set by i/o device)
        FGOK,                 -- FGO FF (Set by i/o device);

        ScClr                -- Clear for sequencer (end of current
instruction)

        : out std_logic
    );
end controller;

architecture a of controller is
    signal I, LCr, Zflag, P: std_logic;
    signal SelPC, SelMem, SelTr, SelAc, Sellr, SelAr
        :std_logic;
    signal D: std_logic_vector(7 downto 0);
begin
-- R flipflip
    RJ <= not T(0) and not T(1) and not T(2) and IEN and (FGI or FGO);
    RK <= R and T(2);
-- IEN flipflop
    IENJ <= P and Ir(7);
    IENK <= P and Ir(6);
-- FGI flipflop
    FGIK <= P and Ir(10);
-- FGO flipflop

```

```

FGOK <= P and Ir(11);
--
I <= IR(15);
P <= D(7) and I and T(3);
LCr <= D(7) and (Not R) and T(3);
Zflag <= '1' when DR = "0000000000000000" else
    '0';
insdecode: dec3x8 port map(IR(14 downto 12), D);
-- Sequence Counter
ScClr <= (R and T(2)) or
    (D(0) and T(5)) or
    (D(1) and T(5)) or
    (D(2) and T(5)) or
    (D(3) and T(4)) or
    (D(4) and T(4)) or
    (D(5) and T(5)) or
    (D(6) and T(6)) or
    LCr or P;
-- Alu
AluFn <= AluPass; -- Passthru
-- Memory write line
MemWr <= (D(3) and T(4))
    or (D(5) and T(4))
    or (D(6) and T(6));
-- Address Register
ArLd <= ((not R) and T(0))
    or ((not R) and T(2))
    or ((not D(7)) and IR(15))
    ;
ArClr <= R and T(0);
ArInc <= D(5) and T(4);
-- Program Counter
PcLd <= D(4) and T(4);
PcClr <= R and T(1);
PcInc <= ((Not R ) and T(1))
    or (D(6) and T(6) and Zflag );
-- Data register DR
DrLd <= (D(0) and T(4))
    or (D(1) and T(4))
    or (D(2) and T(4))
    or (D(6) and T(4))
    ;
DrInc <= D(6) and T(5);
DrClr <= '0';
-- Temporary Register TR
TrLd <= R and T(0);
TrInc <= '0';
TrClr <= '0';
-- Instruction Register IR;
IrLd <= (Not R) and T(1);
IrInc <= '0';
IrClr <= '0';
-- Accumulator Ac;
AcLd <= (D(0) and T(5)) or
    (D(1) and T(5)) or
    (D(2) and T(5)) or
    (D(4) and T(5))
    ;
AcInc <= LcR and IR(5);
AcClr <= LcR and IR(11);
-- Output register OT
OtLd <= P and IR(10);

```

```

OtInc <= '0';
OtClr <= '0';
SelPc <=      (T(0) and '1')
              or
              (D(5) and T(4));

SelMem <=
              ((not R) and T(1))
              or (D(0) and T(4))
              or (D(2) and T(4))
              or ((not D(7)) and I and T(3));

SelTr <= R and T(1);
SelAc <= D(3) and T(4); -- And a lot more!
Sellr <= ((not R) and T(2));
SelAr <= D(4) and T(4);
-- Bus selection
bussel <= BusPc when SelPc='1'
          else BusMem when SelMem ='1'
          else BusTr when SelTr='1'
          else BusAc when SelAc='1'
          else Buslr when Sellr ='1'
          else BusAr when SelAr = '1'
          else BusDr;

end a;

```

### ***Collecting all the component declarations***

With all the devices defined, we can complete the **devices.vhd** file. Here is the completed listing of the file. Note that this file has to be continuously updated and compiled every time it is changed!

#### *Listing of Devices.vhd*

```

LIBRARY ieee; USE ieee.std_logic_1164.all;
library work; USE work.datatypes.all, work.defines.all;
-- This package defines all the devices used in the project
-- It is more logical to break this file into two separate files
-- with one containing generic devices such as decoders, registers, etc.
-- and the other containing specialized devices for the project on hand.
--
-- Author: N. Narasimhamurthi
--
-- $Id: devices.vhd 1.1 10/0/3 .2:.0:.4 unknown Exp Locker: unknown $
PACKAGE devices IS
  -- Component Declaration
  component fa1
    port(
      x,y,z: in std_logic;
      s,c: out std_logic);
  end component;

  component hadder
    port(
      x,y: in std_logic;
      s,c: out std_logic);
  end component;

```

```

component fan
  generic (n: integer);
  port(
    x,y: in std_logic_vector(n-1 downto 0);
    s: out std_logic_vector(n-1 downto 0);
    cout: out std_logic
  );
end component ;

component and1bit
  port (
    x, y: in std_logic;
    z: out std_logic
  );
end component;

component andnbit
  generic (n: integer);
  port(
    x,y: in std_logic_vector(n-1 downto 0);
    s: out std_logic_vector(n-1 downto 0)
  );
end component ;

component dec1x2e
  port(
    sel: in std_logic;
    en: in std_logic;
    o: out std_logic_vector(1 downto 0)
  );
end component ;
component dec1x2
  port(
    sel: in std_logic;
    o: out std_logic_vector(1 downto 0)
  );
end component ;

component dec2x4e
  port(
    sel: in std_logic_vector(1 downto 0);
    en: in std_logic;
    o: out std_logic_vector(3 downto 0)
  );
end component;

component dec2x4
  port(
    sel: in std_logic_vector(1 downto 0);
    o: out std_logic_vector(3 downto 0)
  );
end component ;

component dec3x8
  port(
    sel: in std_logic_vector(2 downto 0);
    o: out std_logic_vector(7 downto 0)
  );
end component ;

component dec4x16

```

```

        port(
            sel: in std_logic_vector(3 downto 0);
            o: out std_logic_vector(15 downto 0)
        );
    end component ;

    component jkfflop
    port(
        J,K      : IN STD_LOGIC;
        clk      : IN STD_LOGIC;
        Q,Qbar   : OUT  STD_LOGIC);
    end component;

    component dfflop
    PORT
    (
        D      : IN STD_LOGIC;
        clk    : IN STD_LOGIC;
        Q      : OUT  STD_LOGIC
    );
    end component;

    component reg1
    port
    (
        cin: in std_logic;
        data: in std_logic;

        clr: in std_logic;
        ld: in std_logic;

        clk: in std_logic;

        q, qbar: out std_logic;
        cout: out std_logic);
    end component;

    component regn
    generic (n: integer);
    port (
        inc: in std_logic;
        data: in std_logic_vector(n-1 downto 0);

        clr: in std_logic;
        load: in std_logic;

        clk: in std_logic;

        q, qbar: out std_logic_vector(n-1 downto 0);
        cout: out std_logic);

    end component;

    component reg16
    port
    (
        inc: in std_logic;
        data: in std_logic_vector(15 downto 0);

        clr: in std_logic;
        load: in std_logic;
    
```

```

        clk: in std_logic;

        q: out std_logic_vector(15 downto 0)
    );
end component;

component reg12 -- 12 bit register with 4 zeros padded to the left
port
(
    inc: in std_logic;
    data: in std_logic_vector(15 downto 0);

    clr: in std_logic;
    load: in std_logic;

    clk: in std_logic;

    q: out std_logic_vector(15 downto 0)
);
end component;

component timer16
port(
    clk: in std_logic;
    clr: in std_logic;
    o: out std_logic_vector(15 downto 0)
);
end component;

component ram
PORT(
    data    : IN Tword;
    address : IN TMemAddress;
    we      : IN STD_LOGIC;
    clock   : IN STD_LOGIC;
    q       : OUT Tword
);
END component ;

component rom
PORT(
    address : IN TMemAddress;
    q       : OUT Tword
);
END component ;

component mem
PORT(
    data    : IN Tword;
    address : IN TAddress;
    we      : IN STD_LOGIC;
    clock   : IN STD_LOGIC;
    q       : OUT Tword
);
END component;

component alu
port(

```

```

        Ein: in std_logic;
        D0, D1: in Tword;
        fn : in aluinst;
        E: out std_logic;
        q: out Tword
    );
end component;

component buslines
    port(
        Ar, PC, DR, AC, IR, TR, Memory: in Tword;
        sel: in busctrl ;
        q: out Tword
    );
end component;

component controller
    port(
        T: in std_logic_vector(15 downto 0);
        IR: in Tword;
        DR: in Tword;
        R: in std_logic;

        MemWr: out std_logic;
        AluFn: out aluinst;

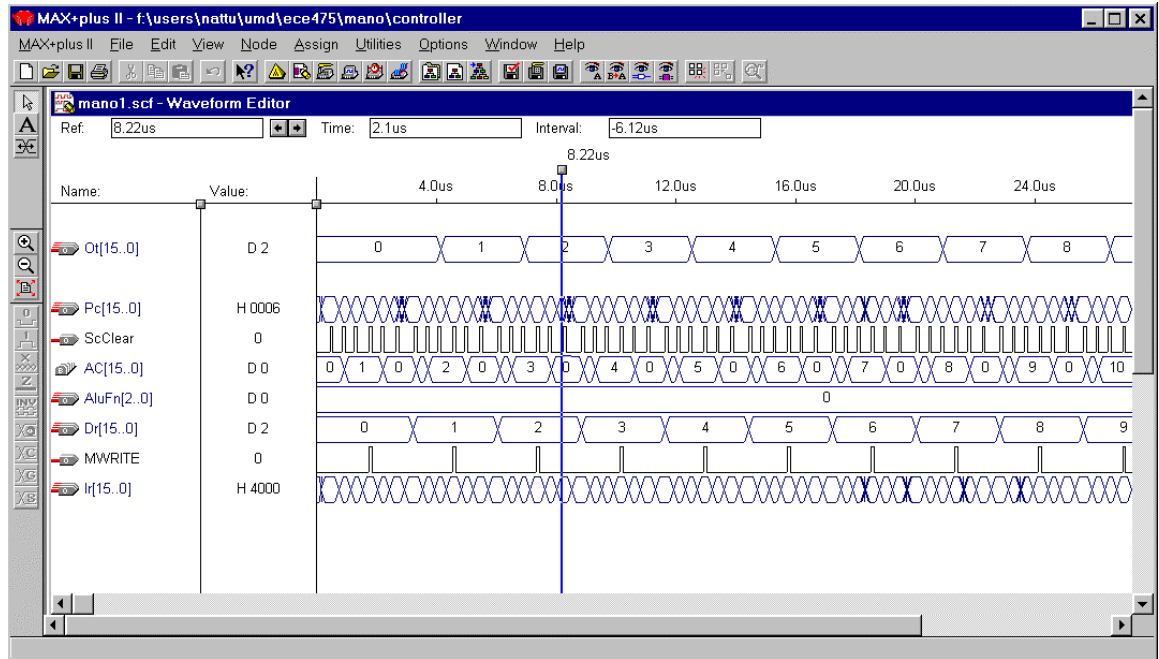
        ScClr,
        ArLd, ArInc, ArClr,
        PcLd, PcInc, PcClr,
        DrLd, DrInc, DrClr,
        AcLd, AcInc, AcClr,
        IrLd, IrInc, IrClr,
        TrLd, TrInc, TrClr,
        OtLd, OtInc, OtClr
        : out std_logic
    );
end component;
END devices;

```

### ***Complete computer***

Now we are in a position to design the complete computer. Rather than use VHDL code, we shall use graphical interface. Note that to convert it to a computer, we have to use port maps to connect the sequence counter, the controller and the structure. The graphical interface will permit us to place I/O pins and also permit us to simulate the computer. All we do is connect the control output lines to the corresponding structure input lines, and the structure output lines to the corresponding input lines of the controller module. The timer is also connected to the controller.





## Exercises:

1. Complete the ALU unit and also implement all the instructions (except interrupt handling and inputs) in the controller.
2. Write a test program that will copy a block of memory from ROM to RAM. Use this to copy a prewritten function. Test the function with a jump to subroutine (note you can JSR to ROM as the return address is stored inside the function).
3. Suppose we want to implement a minimal 68HC11. Determine all the opcodes for which the result is stored in accumulator A (LDAA, ABA, ANDA, NEGA INCA etc.).