The Basic Computer Design

ModelSim

Computer Architecture Lab

January 2017

By Ali Gholami

Under Supervision of

Masoumeh Karami

Computer Engineering Faculty

Amirkabir University of Technology

Contents

Introduction 3

1. Stored program organization 3

Architecture 5

1. Registers 5
2. Common Bus 9

Timing and Control 10

1. Timing Unit 10
2. Control Unit 11

Complete flowchart 13

VHDL Implementation 14

- **Microprocessor and Memory** 14

1. Entity Declaration 14
2. Signal Declarations 16
3. Instruction Decoder 17
4. Sequence Counter 17
5. Timing Signal Generator 18
6. Control Unit 19

- **RAM and Interface** 24
- **Test bench** 26

Introduction

1. Stored program organization

The processor designed in this project has one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.

The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

A 16-bit memory unit with 4096 words can be addressed by the microprocessor. Thus the microprocessor needs 12 address lines to address the memory unit with 4096 (= 2^{12}) words. As each instruction is stored in one 16-bit memory word, four bits are available for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.

The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

As the processor has a single-processor register it is named as accumulator and is labeled as AC. The operation is performed with the memory operand and the content of accumulator (AC).

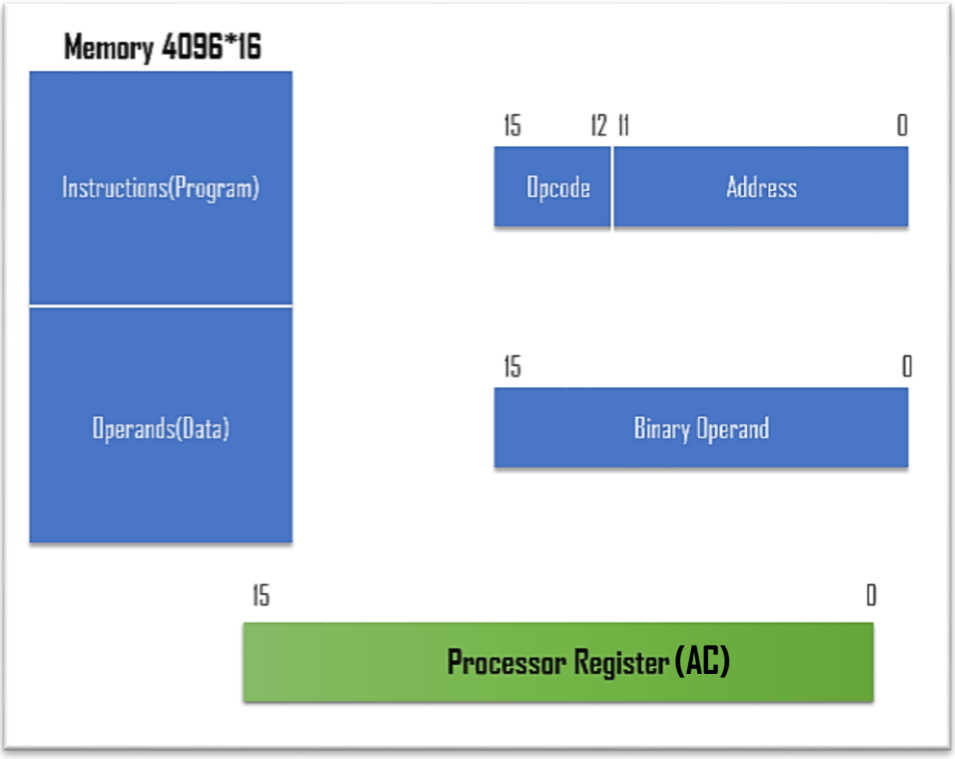

Figure 1.1 – The Stored Program Organization

Architecture

1. Registers

Processor instructions are stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on.

This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution if the current instruction is completed. A register in the control unit for storing the instruction code after it is read from memory is also required. The processor needs processor registers for manipulating data and a register for holding a memory address.

The register configuration for the given architecture is shown in Figure 2.1. The registers are listed in the Table 2.1 together with a brief description of their function and the number of bits that they contain.

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. 3 bits are used as the operation part of the instruction and one bit to specify whether the addressing mode is direct or indirect for memory reference instructions.

Figure 2.1 – Microprocessor Registers

The data register (DR) holds the operand read from memory. The accumulator (AC) is a general purpose processing register. The instruction read from memory is placed in the instruction register (IR). The temporary register (TR) is used for holding temporary data during the processing.

The memory address register (AR) has 12 bits since this is the width of a memory address. The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered.

A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction.

To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. The PC is then incremented by one, so it holds the address of the next instruction in sequence.

Symbol	Number of bits	Register Name	Function
DR	16	Data Register	Hold memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INPR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

Table 2.1 – List of Microprocessor Registers

2. Common Bus

The processor has eight registers and a control unit. A path needs to be provided to transfer information from one register to another and between memory and registers. An efficient scheme of common bus is used for transferring information in this system. Number of wires and connections is reduced in this approach. In this approach all the registers transfer data between themselves through a common bus.

Timing and Control

1. Timing Unit

The microprocessor performs some micro operations every at every clock pulse. The sequence counter (SC) counts from 00002 to 11112. The output of the sequence counter is fed to a 4 × 16 decoder which decodes the input binary number and generates a timing signal which will be called T0, T1 and so on in the sections to follow. Thus the timing unit generates a new timing signal every clock cycle. The SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 × 16 decoder.

The control unit starts fetching an instruction with the timing signal T0. The control unit performs various steps to execute the fetched instruction at each timing signal. Various instructions can take various numbers of timing signals. At the end of the instruction cycle the counter is cleared to 0, causing the next active timing signal to be T0 so that a new instruction cycle can start.

2. Control Unit

The block diagram of the control unit is shown in Figure 3.1. It consists of two decoders, a sequence counter, and a number of control logic gates.

An instruction read from memory is placed in the instruction register (IR), where it is divided into three parts: the I bit, the operation code, and bits 11 through 0.The I bit is the most significant bit (MSB) of the instruction code.

The I bit is stored in I flip-flop. The operation code in the bits 12 through 14 are decoded with a 3 × 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7.

The output of the decoder specifies the operation to be done. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to the control logic gates.

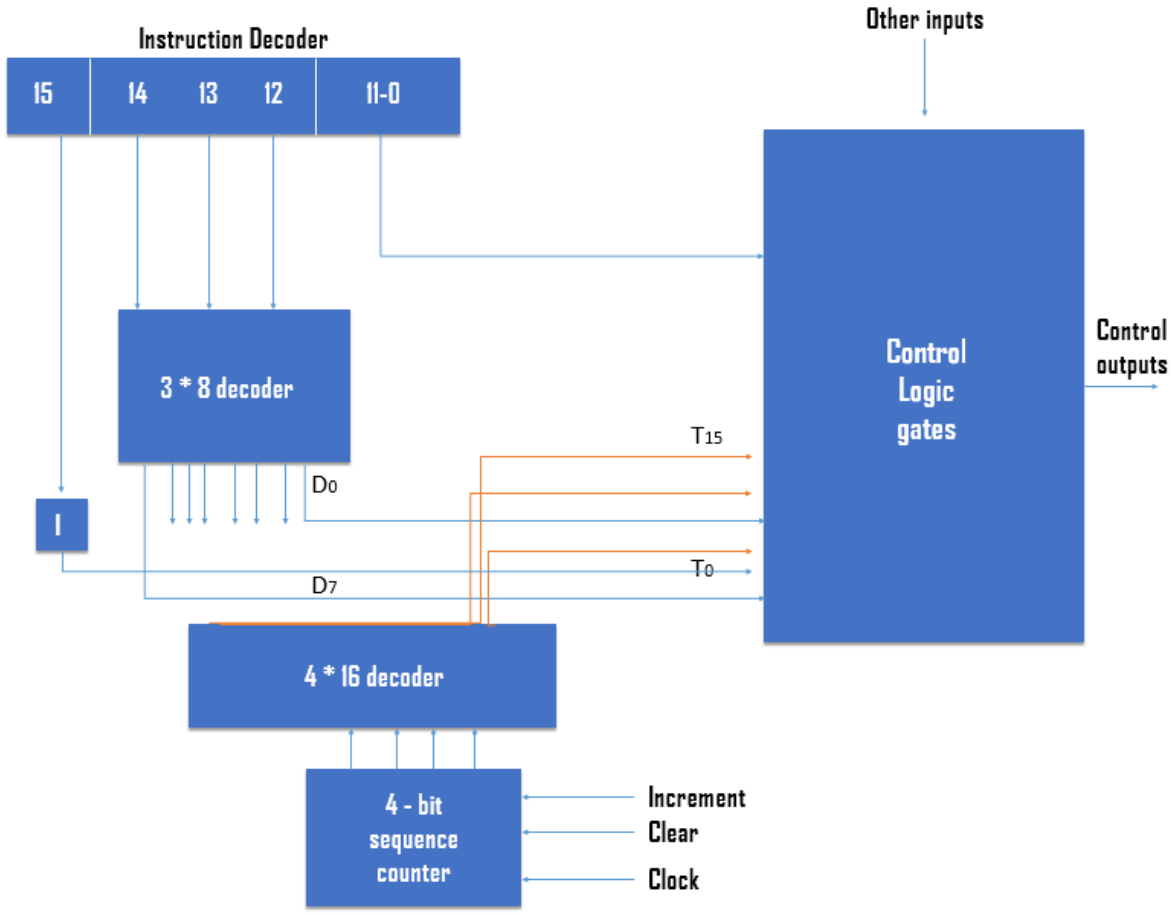

Figure 3.2 – The Control Unit

Complete flowchart

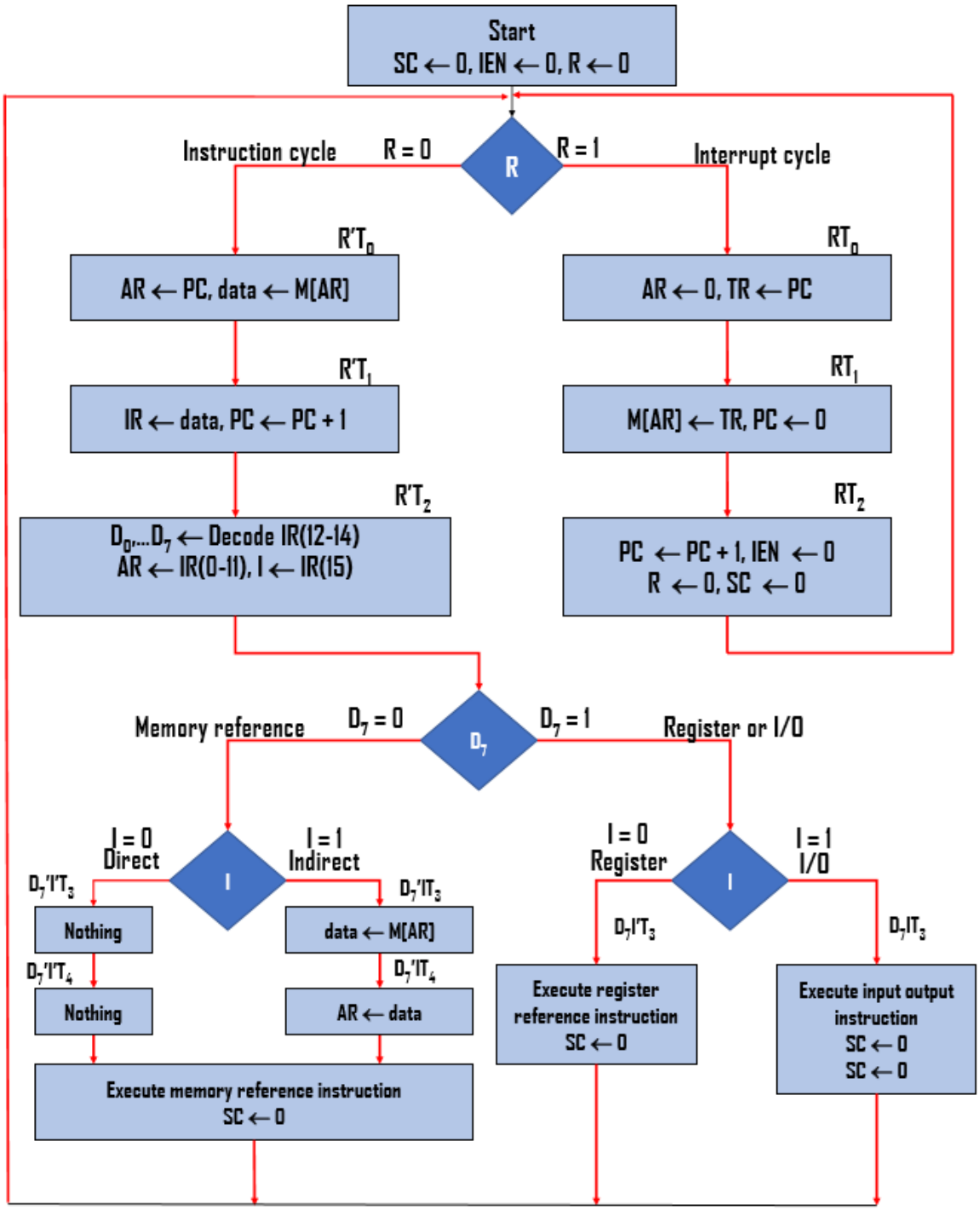

Figure 4.1 – Complete flowchart

VHDL Implementation

❖ Microprocessor and Memory

This section presents a complete program to code the microprocessor in VHDL.

1. Entity Declaration

```
entity microprocessor is
        port
        (
                reset           :in std_logic;                          -- reset
                clk             :in std_logic;                          -- clock
                data            :inout std_logic_vector(15 downto 0);   -- data lines
                address         :out std_logic_vector(11 downto 0);     -- address lines
                memr            :out std_logic;                         -- memory read
                memw            :out std_logic;                         -- memory write
                inport          :in std_logic_vector(7 downto 0);       -- input port
                outport         :out std_logic_vector(7 downto 0);      -- output port
                intr_in         :in std_logic;                          -- interrupt for input
                intr_out        :in std_logic                           -- interrupt for output
        );
end microprocessor;
```

The name of the entity is microprocessor. The entity declaration lists the set of interface ports. These are the signals through which the entity communicates with the external environment. The ports of the microprocessor are as follows:

i.	**clk**	This pin is connected to a clock generator of 4 MHz frequency.
ii.	**reset**	It is used to reset the microprocessor. When this pin goes high the program counter and all other flags and flip-flops are reset. On turning it low again, the microprocessor starts executing instructions from location 0.
iii.	**data**	These pins are connected to a 16-bit data bus. It is used to exchange data with the memory.
iv.	**address**	These pins are connected to a 12-bit address bus. These pins are used to address a specific location of the memory.

v.	**memr**	This pin is connected to the memory. It goes high whenever the microprocessor needs to read a word from the memory.
vi.	**memw**	This pin is connected to a memory. It goes high whenever the microprocessor needs to write a word into the memory.
vii.	**inport**	This is an 8-bit input port of the microprocessor. Any external input device from which the microprocessor wants to read data is connected to this port.
viii.	**outport**	This is an 8-bit output port of the microprocessor. Any external output device to which the microprocessor wants to output data is connected to this port.
ix.	**intr_in**	This is an interrupt pin connected to an input device. Whenever an input device is ready with data, which is to act as an input to the microprocessor, it interrupts the microprocessor by asserting this pin high.
x.	**intr_out**	This is an interrupt pin connected to an output device. Whenever an output device is ready to accept the next data, it interrupts the microprocessor by asserting this pin high.

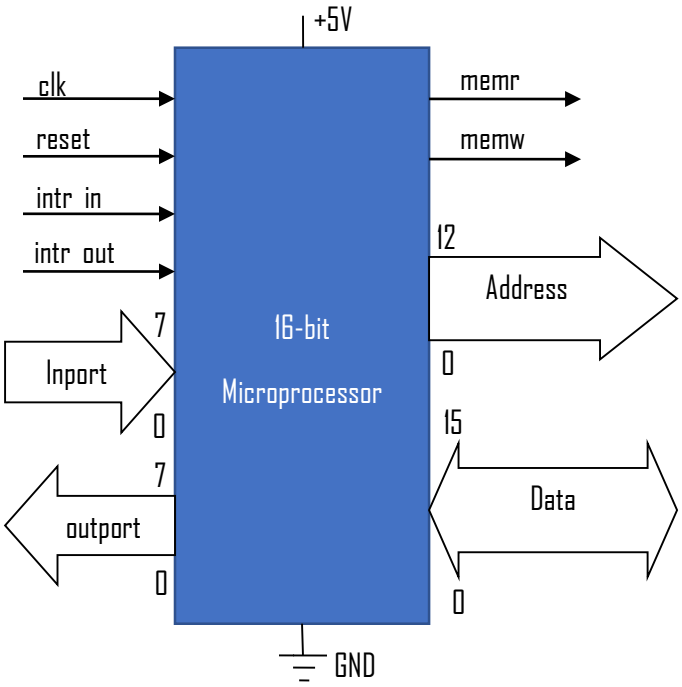

Figure 5.1 – Microprocessor Signals

2. Signal Declarations

```
architecture main of microprocessor is

                                                        -- list of registers
        signal dr           :std_logic_vector(15 downto 0);     -- data register
        signal ar           :std_logic_vector(11 downto 0);     -- address register
        signal ac           :std_logic_vector(15 downto 0);     -- accumulator
        signal ir           :std_logic_vector(15 downto 0);     -- instruction register
        signal pc           :std_logic_vector(11 downto 0);     -- program counter
        signal tr           :std_logic_vector(15 downto 0);     -- temporary register
        signal inpr         :std_logic_vector(7 downto 0);      -- input register
        signal outr         :std_logic_vector(7 downto 0);      -- output register

                                                        -- internals of microprocessor
        signal d            :std_logic_vector(7 downto 0);      -- instruction decoder output
        signal sc           :std_logic_vector(3 downto 0);      -- sequence counter output
        signal t            :std_logic_vector(15 downto 0);     -- timing signals
        signal i            :std_logic;                         -- I bit
        signal e            :std_logic;                         -- extended accumulator
        signal s            :std_logic;                         -- start-stop flip-flop
        signal en_id        :std_logic;                         -- enable signal for instruction decoder
        signal clr_sc       :std_logic;                         -- clear signal for sequence counter
        signal fgi          :std_logic;                         -- input flag
        signal fgo          :std_logic;                         -- output flag
        signal ien          :std_logic;                         -- interrupt enable flip-flop
        signal r            :std_logic;                         -- interrupt flip-flop
```

The first block of signals is the various registers of the microprocessor. The next block of signals is the internal signals used by the microprocessor. Most of them are self-explanatory and needs no further explanation. en_id signal is used to enable the instruction decoder during the instruction decode cycle. The clr_sc signal is used to clear the sequence counter to start a new instruction cycle.

3. Instruction Decoder

```
begin
process(en_id)                                          -- instruction decoder
        begin
        if en_id='1' then
                case ir(14 downto 12) is
                        when "000" => d <= "00000001";
                        when "001" => d <= "00000010";
                        when "010" => d <= "00000100";
                        when "011" => d <= "00001000";
                        when "100" => d <= "00010000";
                        when "101" => d <= "00100000";
                        when "110" => d <= "01000000";
                        when "111" => d <= "10000000";
                        when others => null;
                end case;
        end if;
end process;
```

It can be seen that the instruction decoder decodes when en_id = '1'. The instruction decoder decodes the bits represented by ir (14 downto 12) to make one of the bits of the register d (7 downto 0) high.

4. Sequence Counter

```
process(clk,s,clr_sc,inr_sc)                            -- 4-bit sequence counter
        begin
        if s='0' then
                if (clk'event and clk='1') then
                        if clr_sc='1' then
                                sc <= "0000";
                        else
                                sc <= sc + "0001";
                        end if;
                end if;
        end if;
end process;
```

The 4-bit sequence counter responds to the positive edge of every clock pulse. It counts from 00002 to 11112. When the

clr_sc goes high the sequence counter is reset to 00002. The output of the sequence counter is obtained in the SC register.

5. Timing Signal Generator

```
process(sc, reset)                                  -- 4-to-16 decoder to generate timing signals
        begin
        if reset='1' then
                                        t <= "0000000000000000";
        else
                case sc is
                        when "0000" => t <= "0000000000000001";
                        when "0001" => t <= "0000000000000010";
                        when "0010" => t <= "0000000000000100";
                        when "0011" => t <= "0000000000001000";
                        when "0100" => t <= "0000000000010000";
                        when "0101" => t <= "0000000000100000";
                        when "0110" => t <= "0000000001000000";
                        when "0111" => t <= "0000000010000000";
                        when "1000" => t <= "0000000100000000";
                        when "1001" => t <= "0000001000000000";
                        when "1010" => t <= "0000010000000000";
                        when "1011" => t <= "0000100000000000";
                        when "1100" => t <= "0001000000000000";
                        when "1101" => t <= "0010000000000000";
                        when "1110" => t <= "0100000000000000";
                        when "1111" => t <= "1000000000000000";
                        when others => null;
                end case;
        end if;
end process;
```

A 4-to-16 decoder decodes the sequence counter to generate the timing signals. It indicates a timing signal by making one of the bits of the t-register high. When the microprocessor is under reset condition, it doesn't generate any timing signal. When the reset pin of the microprocessor is low, it generates a timing signal depending on the value of SC register, which is the output of the sequence counter.

6. Control Unit

```
process(t)                                                  -- control unit
        variable temp           :std_logic;
        variable sum            :std_logic_vector(16 downto 0);
        variable ac_ext         :std_logic_vector(16 downto 0);
        variable dr_ext         :std_logic_vector(16 downto 0);
        begin
        if reset='1' then                                   -- reset microprocessor
                clr_sc <= '1';
                s <= '0';
                r <= '0';
                ien <= '0';
                fgi <= '0';
                fgo <= '0';
                memr <= '0';
                memw <= '0';
                en_id <= '0';
                pc <= "000000000000";
        elsif ((not r) and t(0))='1' then                   -- load 'ar' with the contents of 'pc'
                clr_sc <= '0';
                memr <= '1';
                memw <= '0';
                ar <= pc;
        elsif ((not r) and t(1))='1' then                   -- fetch instruction and increment 'pc'
                ir <= data;
                pc <= pc + 1;
        elsif ((not r) and t(2))='1' then                   -- decode opcode
                fgi <= intr_in;
                fgo <= intr_out;
                memr <= '0';
                en_id <= '1';
                ar <= ir(11 downto 0);
                i <= ir(15);
        elsif (r and t(0))='1' then                         -- store return address in tr
                clr_sc <= '0';
                ar <= "000000000000";
                tr <= "0000" & pc;
```

When the reset pin goes high some of the signals and registers are initialized bring back the microprocessor to its initial state. It can be seen that the following statements, which occur at the timing signals T0, T1 and T2, are executed only when the R flip-flop is cleared. Thus these statements represent the instruction fetch and decode of an instruction cycle. In the T2 cycle, interrupt pins intr_in and intr_out are read into the fgi and fgo flip-flops because the T3 cycle will

need these signals to determine whether an interrupt has occurred.

```
elsif (r and t(1))='1' then                    -- store return address in location 0
        data <= tr;
        memw <= '1';
        pc <= "000000000000";
elsif(r and t(2))='1' then                     -- increment pc, and reset ien and r
        pc <= pc + 1;
        ien <= '0';
        r <= '0';
        clr sc <= '1';
```

The statements of the code above are executed when R flip-flop is set. Thus it represents an interrupt cycle. At the end of the interrupt cycle IEN and R flip-flops are cleared and the sequence counter is reset to resume the normal execution of the instructions.

```
        elsif ir(7)='1' then                   -- ION  (interrupt enable on)
                ien <= '1';
        elsif ir(6)='0' then                   -- IOF  (interrupt enable off)
                ien <= '0';
        end if;
        clr_sc <= '1';
elsif (d(7) and (not i))='1' then              -- execute register reference instruction
        if ir(11)='1' then                     --  CLA (clear ac)
                ac <= "0000000000000000";
        elsif ir(10)='1' then                  -- CLE  (clear e)
                e <= '0';
        elsif ir(9)='1' then                   -- CMA  (complement ac)
                ac <= not ac;
        elsif ir(8)='1' then                   -- CME  (complement e)
                e <= not e;
        elsif ir(7)='1' then                   -- CIR  (cirulate right)
                temp := e;
                e <= ac(0);
                ac <= temp & ac(15 downto 1);
        elsif ir(6)='1' then                   -- CIL  (circulate left)
                temp := e;
                e <= ac(15);
                ac <= ac(14 downto 0) & temp;
        elsif ir(5)='1' then                   -- INC  (increment ac)
                ac <= ac + 1;
        elsif ir(4)='1' then                   -- SPA  (skip if positive)
                if ac(15)='0' then
                        pc <= pc + 1;
                end if;
        elsif ir(3)='1' then                   -- SNA  (skip if negative)
                if ac(15)='1' then
                        pc <= pc + 1;
                end if;
```

When D7 is 1 and I is 1, it is an input output instruction. This condition is tested in the beginning of this portion of the code and then the type of instruction is determined and executed. These statements are executed with the timing signal T3. After executing the instruction the sequence counter is cleared to fetch, decode and execute the next instruction.

```
                elsif ir(2)='1' then                    -- SZA  (skip if ac is zero)
                        if ac=0 then
                                pc <= pc + 1;
                        end if;
                elsif ir(1)='1' then                    -- SZE  (skip if e is zero)
                        if e='0' then
                                pc <= pc + 1;
                        end if;
                elsif ir(0)='1' then                    -- HLT  (halt)
                        s <= '1';
                end if;
                clr_sc <= '1';
        elsif (not d(7))='1' then
                if i='1' then                           -- fetch address for indirect addressing mode
                        memr <= '1';
                elsif i='0' then                        -- do nothing for direct addressing mode
                        null;
                end if;
        end if;


elsif t(4)='1' then
        fgi <= intr_in;
        fgo <= intr_out;
        r <= ien and (fgi or fgo);
        if d(7)='1' then
                if i='1' then                           -- fetch address for indirect addressing mode
                        ar <= data(11 downto 0);
                elsif i='0' then
                        null;                           -- do nothing for direct addressing mode
                end if;
        end if;
elsif t(5)='1' then
        fgi <= intr_in;
        fgo <= intr_out;
        r <= ien and (fgi or fgo);
```

During the timing signals from T4 to the end of the instruction cycle, the interrupt pins are continuously monitored and the R flip-flop is set whenever an interrupt occurs. It can be seen as the first three statements of the code for every timing signal from T4 to T8.

The code below represents the various cycles of various memory reference instructions. Different memory reference instructions need different number of timing signals.

```
        if d(0)='1' then                                -- AND (and to ac)
                memr <= '1';
        elsif d(1)='1' then                     -- ADD (add to ac)
                memr <= '1';
        elsif d(2)='1' then                     --  LDA (load to ac)
                memr <= '1';
        elsif d(3)='1' then                     -- STA (store ac)
                data <= ac;
                memw <= '1';
                clr_sc <= '1';
        elsif d(4)='1' then                     -- BUN (branch unconditionally)
                pc <= ar;
                clr_sc <= '1';
        elsif d(5)='1' then                     -- BSA (branch and save return address)
                data <= "0000" & pc;
                memw <= '1';
                ar <= ar + 1;
        elsif d(6)='1' then                     -- ISZ (increment and skip if zero)
                memr <= '1';
        end if;
elsif t(6)='1' then
        fgi <= intr_in;
        fgo <= intr_out;
        r <= ien and (fgi or fgo);
                                                -- memory read for AND, ADD, LDA and ISZ instructions
        if (d(0) or d(1) or d(2) or d(6)) = '1' then
                dr <= data;
        elsif d(5)='1' then                     -- BSA (branch and save return address)
                memw <= '0';
                pc <= ar;
                clr_sc <= '1';
        end if;
```

```
elsif t(7)='1' then
        fgi <= intr_in;
        fgo <= intr_out;
        r <= ien and (fgi or fgo);
        if d(0)='1' then                                    -- AND (and to ac)
                memr <= '0';
                ac <= ac and dr;
                                        clr_sc <= '1';
        elsif d(1)='1' then                                 -- ADD (add to ac)
                memr <= '0';
                ac_ext := '0' & ac;
                dr_ext := '0' & dr;
                sum := ac_ext + dr_ext;
                ac <= sum(15 downto 0);
                e <= sum(16);
                                        clr_sc <= '1';
        elsif d(2)='1' then                                 --  LDA (load to ac)
                memr <= '0';
                ac <= dr;
                clr_sc <= '1';
        elsif d(6)='1' then                                 -- ISZ (increment and skip if zero)
                memr <= '0';
                dr <= dr + 1;
        end if;
elsif t(8)='1' then
        fgi <= intr_in;
        fgo <= intr_out;
        r <= ien and (fgi or fgo);
```

The last part of the program uses dataflow modeling style.

```
                if d(6)='1' then                            -- ISZ (increment and skip if zero)
                        data <= dr;
                        memw <= '1';
                        if dr=0 then
                                pc <= pc + 1;
                        end if;
                                        clr_sc <= '1';
                end if;
        end if;
    end process;

    inpr <= inport;
    outport <= outr;
    address <= ar;
end main;
```

❖ RAM and Interface

The last section of the code belongs to the interface we are working on and the RAM, which is a simple a place to store the user input data such as addresses and their data. These data are sent to the ebscu unit and then according to the input mode, which mode “1” is for the data transfer between microprocessor’s memory and RAM and mode “0” stands for the data transfer between the RAM and User’s data.

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

entity ebscu is
        port
        (
                mode:           in std_logic;

                mpdata: inout std_logic_vector(15 downto 0);
                mpaddress:      in std_logic_vector(11 downto 0);
                userdata:               inout std_logic_vector(15 downto 0);
                useraddress:    in std_logic_vector(2 downto 0);
                ramdata:                inout std_logic_vector(15 downto 0);
                ramaddress:     out std_logic_vector(2 downto 0);

                mpread:         in std_logic;
                mpwrite:                in std_logic;
                userread:               in std_logic;
                userwrite:      in std_logic;
                ramread:                out std_logic;
                ramwrite:               out std_logic
        );
end ebscu;
```

```
architecture main of ebscu is
        begin
        process(mpread, mpwrite, userread, userwrite)
                begin
                if mode='0' then
                        ramread <= mpread;
                        ramwrite <= mpwrite;
                        ramaddress <= mpaddress(2 downto 0);
                        if mpread='1' then
                                mpdata <= ramdata;
                        elsif mpwrite='1' then
                                ramdata <= mpdata;
                        end if;
                elsif mode='1' then
                        ramread <= userread;
                        ramwrite <= userwrite;
                        ramaddress <= useraddress;
                        if userread='1' then
                                userdata <= ramdata;
                        elsif userwrite='1' then
                                ramdata <=  userdata;
                        end if;
                end if;
        end process;
end main;
```

❖ Test bench

```
entity testbench is
	port
	(
		reset	:in std_logic;
		clk	:in std_logic;
		mode	:in std_logic;

		inport	:in std_logic_vector(7 downto 0);
		outport	:out std_logic_vector(7 downto 0);

		data	:inout std_logic_vector(15 downto 0);
		address	:in std_logic_vector(2 downto 0);
		load	:in std_logic;
		debug	:in std_logic
	);
end testbench;
```

Going further and ignoring the boring component declaration we'll have:

```
process(databus1)
	variable busvar: std_logic_vector(15 downto 0);
	begin
	busvar := databus1;
	databus2 <= busvar;
end process;
process(databus2)
	variable busvar: std_logic_vector(15 downto 0);
	begin
	busvar := databus2;
	databus1 <= busvar;
end process;
process(databus3)
	variable busvar: std_logic_vector(15 downto 0);
	begin
	busvar := databus3;
	databus4 <= busvar;
end process;
process(databus4)
	variable busvar: std_logic_vector(15 downto 0);
	begin
	busvar := databus4;
	databus3 <= busvar;
end process;
d1: divider		port map(reset, clk, internclk);
d2: microprocessor		port map(mpreset, internclk, databus1, addressbus1, read1, write1, inport, outport,intr_in,intr_out);
d3: ebscu		port map(mode, databus2, addressbus1, data, address, databus3, addressbus2, read1, write1, debug, load, read2, write2);
d4: ram		port map('1', read2, write2, databus4, addressbus2);
```