UPPSALA
UNIVERSITET

VHDL Final Project

16-bit Microprocessor

Vivek Vivian
dept. of Information Technology
Uppsala Universitet
Uppsala, Sweden

***Abstract*—The procedure of design and verification of a dedicated 16-bit microprocessor kernel is introduced in this report. The proposed design has traditional microprocessor components that interact with each other. It makes use of a Finite State Machine using a 1-stage pipeline for instruction execution.The design architecture has been written in Very High Speed Integrated Circuit Hardware Descriptive Language(VHDL). Debugging and testing has been successfully carried out using Cyclone V 5SEMA5F31C6 in DE1_SoC platform.**

***Index Terms*—16-bit, Microprocessor,VHDL, Control unit(CU), 16-bitArithmetic Logic Unit (ALU), 16-bit RAM, CPU, RTL,FPGA,ASIC.**

I. Introduction

Microprocessors are capable of performing basic arithmetic operations, moving data from place to place, and making basic decisions based on the quantity of certain values as shown in Figure 1 below.[1] All microprocessors can be divided into two main categories: general-purpose microprocessors and dedicated microprocessors. Dedicated Processors which are also known as aplication specific integrated circuits(ASICs), are dedicated to perform particular tasks.[2] These instructions are coded onto the processor itself and they cannot be modified after manufacture. Processors also come in various sizes and the most common ones being 8-bit, 16-bit, and 32-bit depending on the demand, cost, power and programability.[3] A16-bit microprocessor, has higher performance power than 8-bit microprocessors and lower power consumption than 32-bit microprocessors, and mostly used in 16-bit applications such as disk driver controller, airbags and cellular communication.[4]

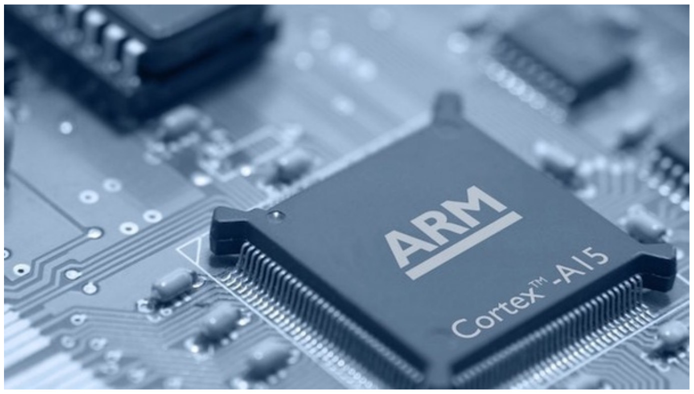


Figure 1: ARM Cortex Microprocessor

II. Project Description

A. Tools & Softwares

The 16-bit processor is used for assembly language programming. These requirements are met by using an FPGA based processor with VHDL language. This project has been designed usin Intel Quartus Prime 18.1 and the simulations have been done using ModelSim Altera. The debugging and testing has been performed on an Cyclone V FPGA using ARM Cortex A9 on a DE1_SoC.

B. The Design

The microprocess consists of a simple architecture made up of a processor which is interfaced with a memory unit as shown below in figure 2. The processor consists of few registers, an arithmetic logic unit(ALU), a Control Unit(CU) and a Memory Interface. All these components are interfaced using busses which act as a communication path.The Microrocessor can be divided into four parts:

1) Arithmetic Logic Unit(ALU)
2) Registers
3) Memory Interface
4) Control Unit(CU)

These four parts communicate with each other and together make up a microprocessor.

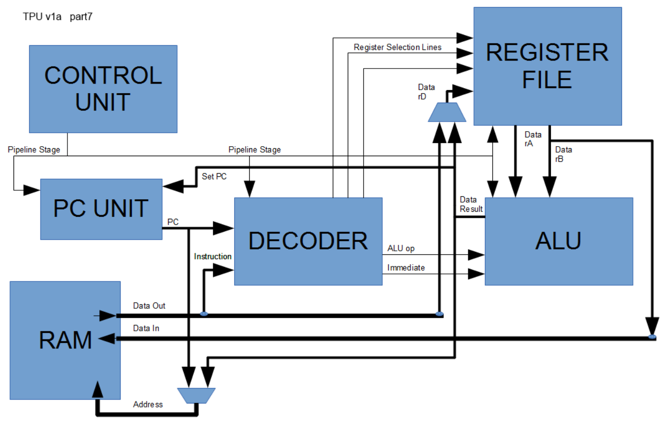


Figure 2: Simple 16-bit Microprocessor[9]

III. Theory

The design of the 16- bit microprocessor can be divided into two parts which is the datapath and the control unit. The datapath is responsible for all the operations that need to be performed on the data. It includes (1) adders, shifters, ALU (2) registers and elements for temporary storage of data (3) busses for the tansfer of datat between different components in the datapath.[2]
The control unit is responsible for controlling all the operations of the datapath by providing appropriate control signals. The control unit operates by transitioning from one state to another per clock cycle and hence the control unit is also referred to as a Finite State Machine(FSM).[2]

IV. Implementation

The implementation involves certain design rules and assumptions as mentioned below:

1) The design generated has both the data-path as well as the instruction set to be 16-bits long.
2) The processor design is based on Von-Neumann Architecture of a single memory with stored program.
3) The instruction set for this project has been designed the following wayas shown below, the opcode and address are each 8-bits long which means that 256 different operations and 256 different memory locations

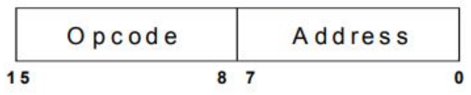
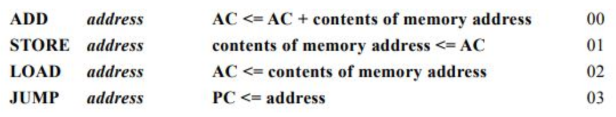


Figure 3: 16-bit Instruction

4) The opcode leads to an operation when it is decoded in the control unit and the address part is pointing on the data or the next instruction. The following are the instructions that are implemented in the project followed by its particular opcode.



ADD	*address*	AC <= AC + contents of memory address	00
STORE	*address*	contents of memory address <= AC	01
LOAD	*address*	AC <= contents of memory address	02
JUMP	*address*	PC <= address	03

Figure 4: Instruction set and Opcodes

5) The control unit as mentioned before is implemented using a state machine and the oder of the intruction fetch, decode and execute is shown below. This cycle keeps repeating. Each of these stages can contain many states. During the fetch stage an instruction is fetched from the memory, during the decode stage the instruction is decoded and the specific operation is determined and this stage then decides which operation should be executed.

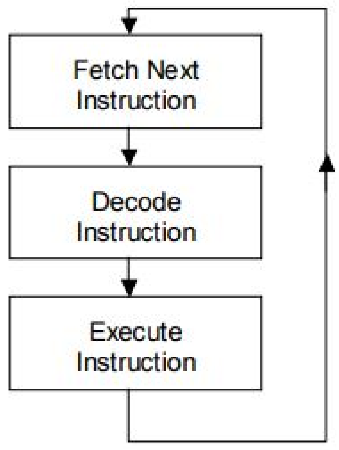
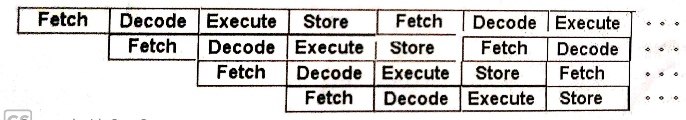


Figure 5: Instruction Cycle

6) The reason as to using three states eventhough it can be done using only the fetch and execute is due to the behaviour of modern processors which use pipelining. A flow of the pipelining is shown below and we implement only one of these lines in this project.



Fetch	Decode	Execute	Store	Fetch	Decode	Execute	...
	Fetch	Decode	Execute	Store	Fetch	Decode	...
		Fetch	Decode	Execute	Store	Fetch	...
			Fetch	Decode	Execute	Store	...

Figure 6: Pipelining stages

V. Components

1) Program Counter: The Program counter(PC) is a register in the processor that keeps the 8-bit address of the instruction that is being executed.It has the system clock and reset connections. When the reset state is executed the Program counter is set to 0.[5]
After the fetch cycle completes the program counter is incremented by 1, which means that after each instruction fetch the program counter points to the next instruction to be executed.
The pc_op_code which is a signal from the contol unit which is used to decide what operation to perform. The pc_count counts the number of instructions that is executed. pc_in and pc_out is for the 8-bit address bus.

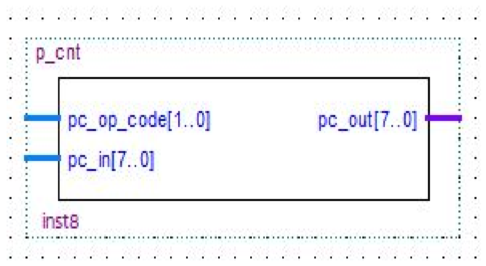


Figure 7: Program Counter

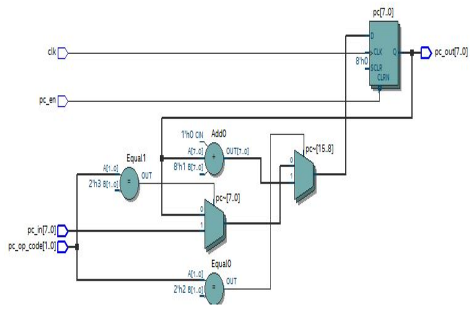


Figure 8: RTL View of Program Counter)

2) Instruction Register: The instruction Register is part of the processor that holds the address of the instruction that is ccurrently being executed.[6]It contains the global clock and reset. The Instruction Register gets a 16-bit input called as instruction from the memory interface. It then breaks the 16-bit instruction into the higher 8-bit opcode and the lower 8-bit address based on an enable signal. It the forwards the opcode to the control unit to be decoded and sends the address to the program counter.

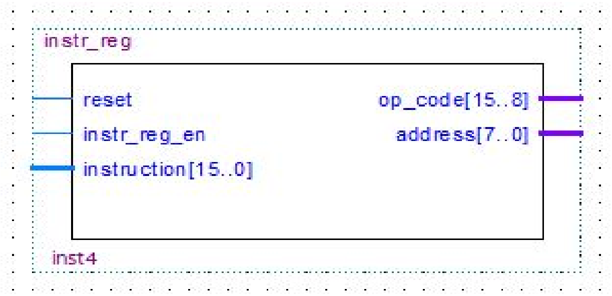


Figure 9: Instruction Register

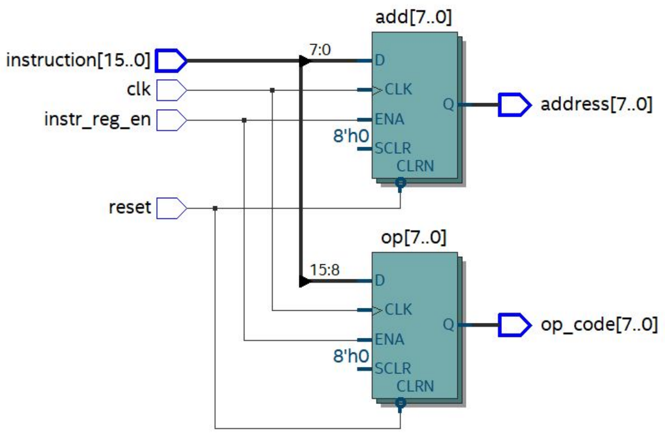


Figure 10: RTL View of Instruction Register)

3) Accumulator: An Accumulator is a register in whcih intermediate arithmetic and logic results are stored.[7] Data enters the accumulator from the memory interface. Once an operation has to take place the data from the accumulator moves to the arithmetic and logic unit and once the operation is complete the results are again stored back into the accumulator. The result stored in the accumulator then moves to the memory interface to be written onto the RAM when a store instruction has been executed. The block diagram of an accumulator is shown below.

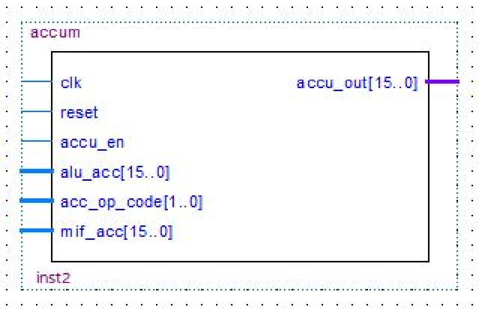


Figure 11: Accumulator

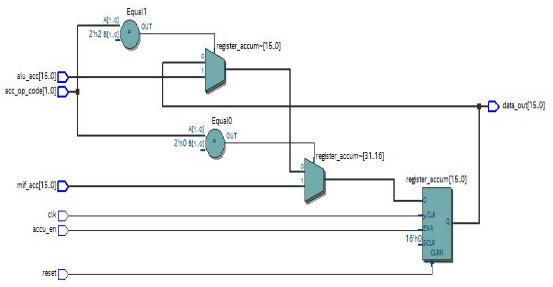


Figure 12: RTL View of Accumulator

4) Arithmetic Logic Unit(ALU): The arithmetic logic unit(ALU) is a combinational circuit that performs arithmetic and logical operations.[8] Various operations like add, subtract, logical and, logical or take place in the ALU. The data for the operations comes in from the accumulator and the contents of a specified memory location through the memory interface. After the operation has taken place the result is then moved to the accumulator where it is temporarily stored untill it gets written back into the memory. This ALU is a critical component in the processor and is shown below.

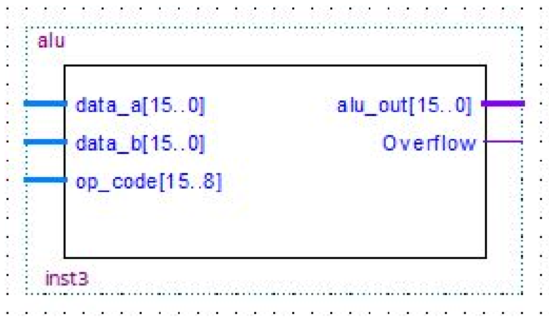


Figure 13: Arithmetic and Logic Unit(ALU)

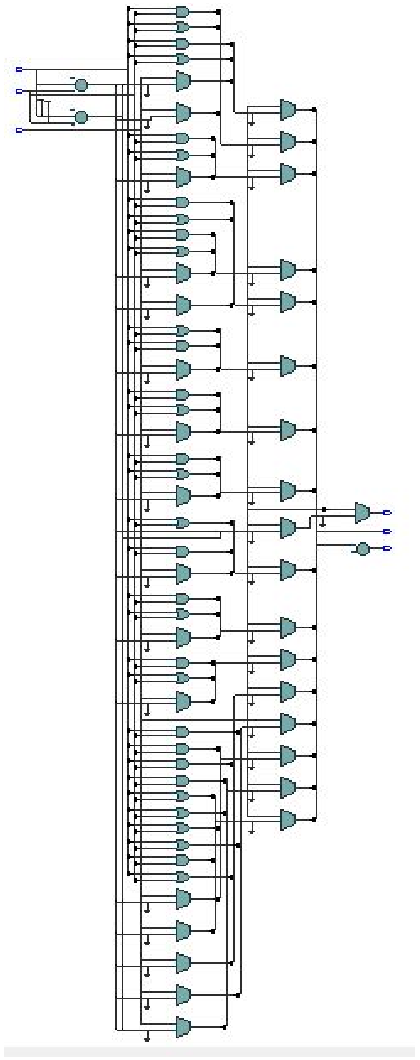


Figure 14: RTL View of ALU

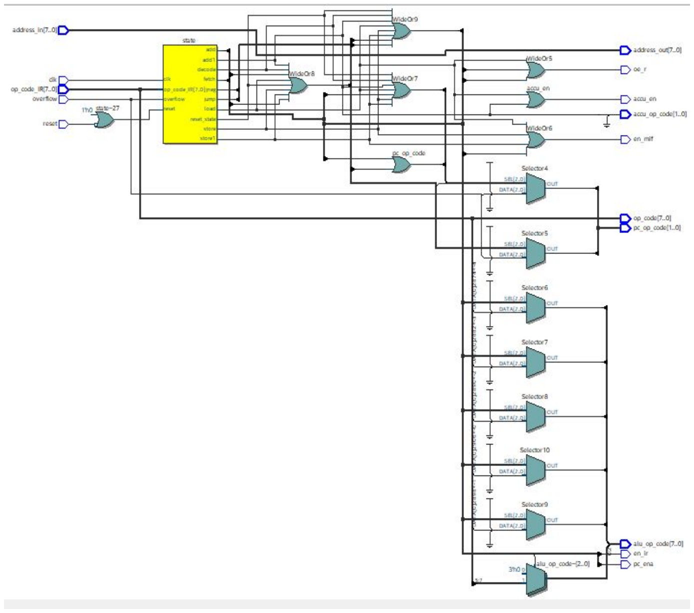


Figure 15: State Diagram of Control Unit

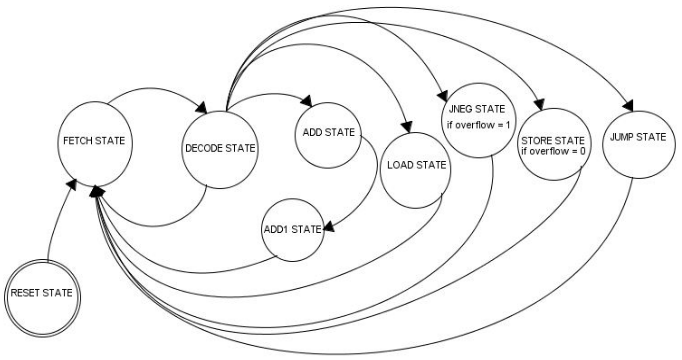


Figure 16: State Diagram of Control Unit

5) Control Unit(CU): The contol unit is like the brain of the microprocessor. It contols the timing and control signals to all the various components that are connected in the microprocessor. The contol unit is implemented as a Finite State Machine(FSM).

By stepping through a sequence of states, the contol unit controls the operations of the data path. For each state that the control unit is in, the output logic that is present inside the control unit will generate appropriate control signals for the datapath to perform one data operation.[2] The first state of the control unit is reset state in which the program counter and the accumulator is reset. The next state is fetch where the address from the program counter goes to the memory interface, read memory is performed and the 16-bit instruction set is retreived and then sent to the Instruction Register and the program counter is incremented by 1.

The next state is the decode state, where the instruction in the Instruction Register is split into 8-bit opcode and address is sent to the control unit to get decoded. The opcode is then read by the control unit and the specific operation to be performed is sent to the alu and the address to fetch the next instruction is sent to the memory interface. The next state is the execute state where the data from the memory is taken to the alu and a specific operation is performed based on the opcode and then the result is stored in the accumulator. When store state is executed the data from the accumulator moves to the memory interface to be written into the RAM. The program counter now has the next address to be executed and these states keep repeating untill forever. This is in brief as to how the control unit functions and the different states are executed.

6) Memory Interface: The memory interface here acts as the memory address register. It receives signals and addresses from different registers and based on the addresses it access the memory unit and provides the registers with the particular data or instructions. It acts as a bridge between

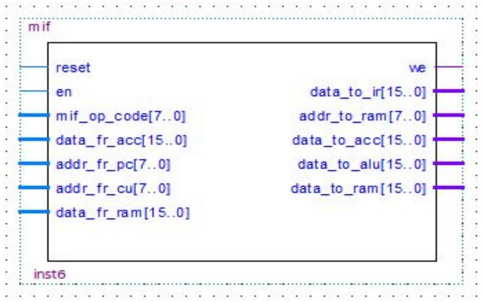


Figure 17: Memory Interface)

7) Memory - RAM: This design has been made using a structural VHDL code. In this design a 256 x 16 block RAM is used.This unit stores instructions and data. The memory unit is interfaced with the memory interface to which is communicates and tansfers instructions and data.

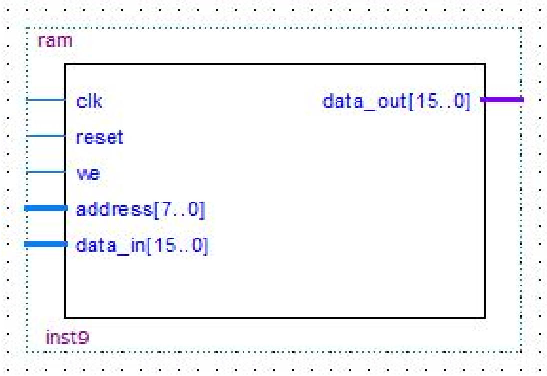


Figure 18: Memory Unit)

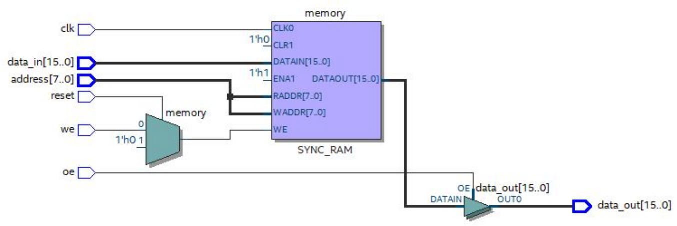


Figure 19: RTL View of RAM

VI. Complete Design

The RTL view of the complete Design is shown below:

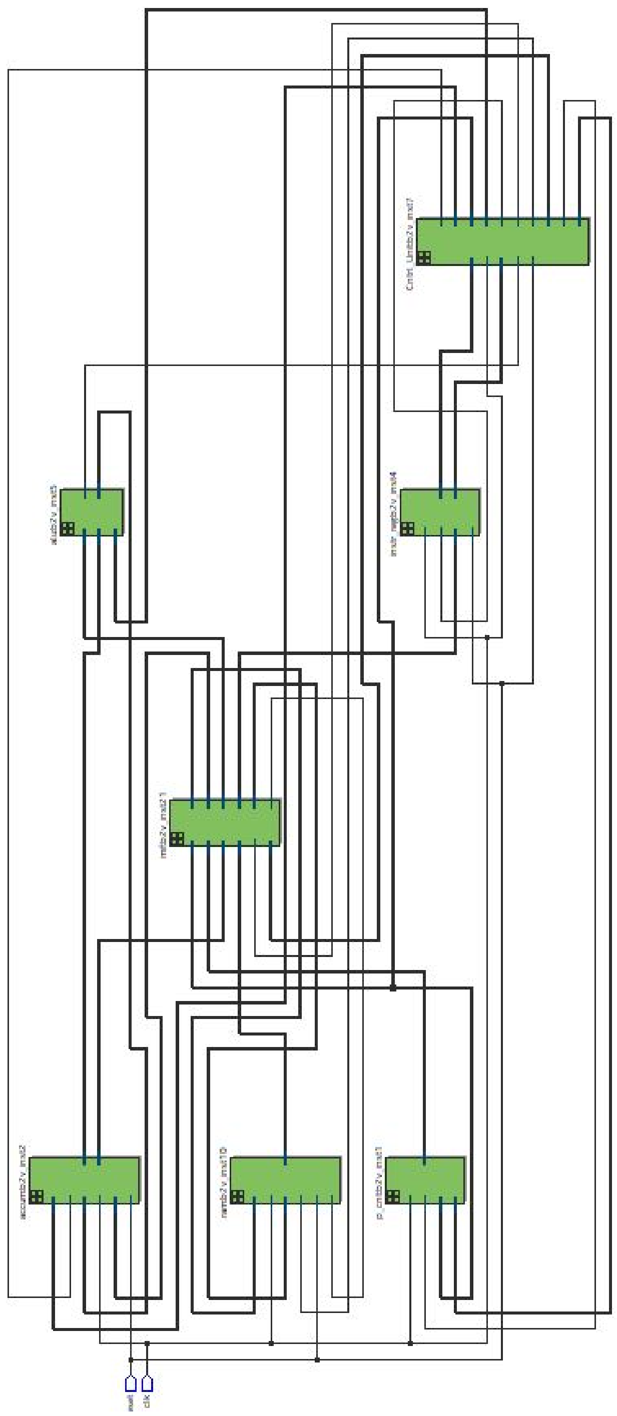


Figure 20: Complete Design RTL

The complete design consists the processor part which is connected to a memory part with the help of the memory Interface.

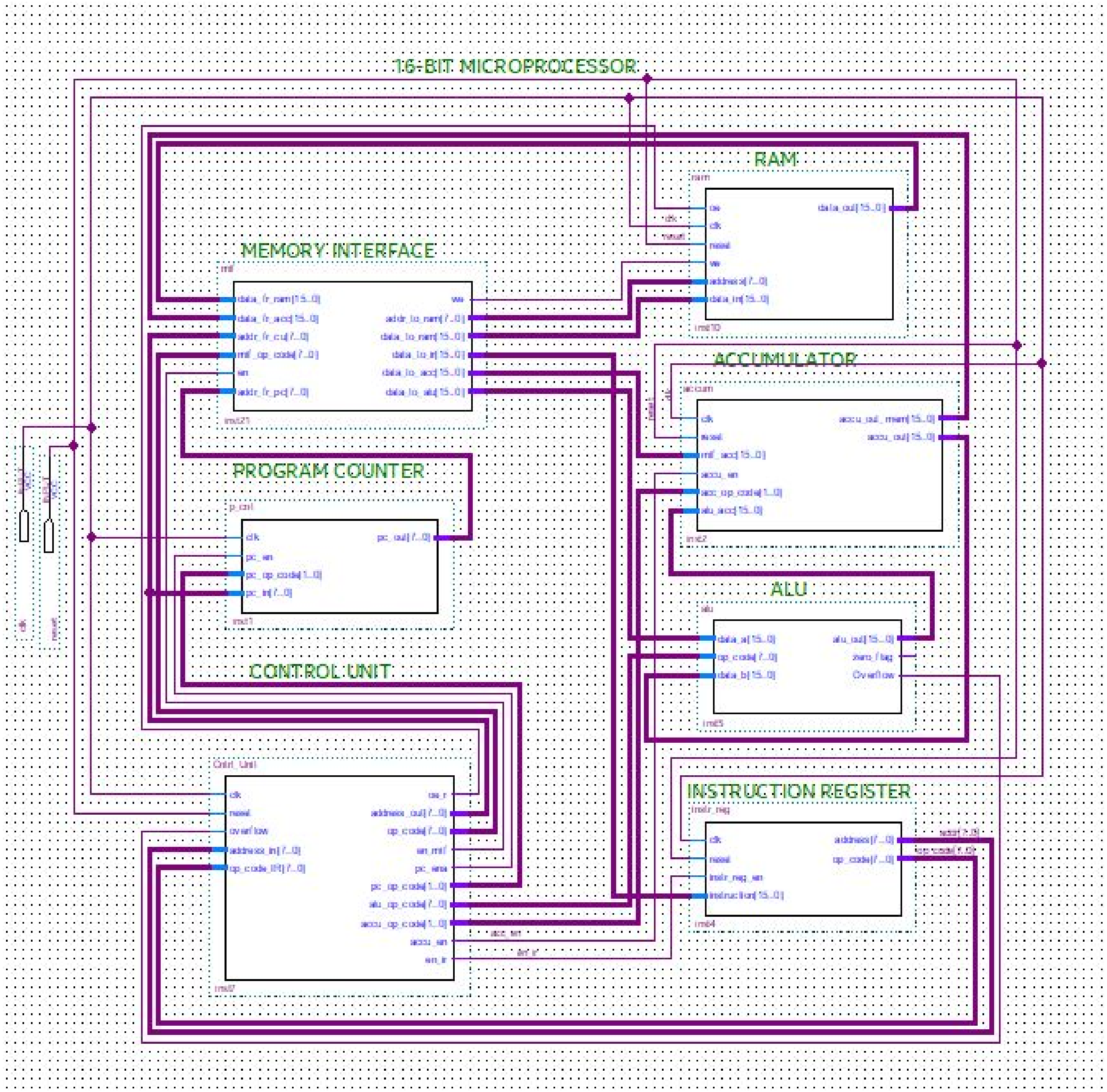


Figure 21: Complete Design

VII. Tests & Results

Testing is one of the most important part of development in a project. Here in this project we have tested two condtitions. The first one is: A = B + C.

1) Reset State: For the first equation we have stored the values of B = 7 and C = 4. The flow of the code goes this way:

Listing 1: Fetch State

```
when reset_state => pc_ena     <= '0';--reset PC
                    oe_r   <= '0';--disable read operation
```

2) Fetch State: It begins with a reset state where the program counter is made equal to an 8-bit value which is 0. The next state is the fetch state where the control unit requests the program counter to send the addess that's present in it to the Memory Interface(MIF) which then sends this 8-bit address to the memory unit which is the RAM and performas a read operation and sends this 16-bit data to the Instruction Register(IR). The IR then splits the 16-bit data into an 8-bit address and an 8-bit opcode and transfers it to the Control Unit(CU). The fetch cycle ends by incrementing the Program counter by 1. Following are the codes sent by the control unit to the various registers to execute the fetch cycle:

Listing 2: Fetch State

```
when fetch => pc_ena   <= '1';--enable PC
        en_mif   <= '0';--mif performs read and fetches
            the data
        oe_r      <= '1';--read enable for the ram
        en_ir     <= '1';--enable IR to split instruction
        pc_op_code    <= "10";--send address to mif
        alu_op_code   <= "00011111";
```

3) Decode State: In the decode state the control unit decodes the 8-bit op code that it receives from the Instruction Register to find out the operation that it has to execute next. The following codes are send:

Listing 3: Decode State

```
when decode =>   en_ir     <= '0';
            pc_op_code    <= "00";--send address to mif
            oe_r   <= '0';--Disable memory read
```

4) Load State: In this state the op code and the address from the control unit goes to the the Memory Interface(MIF). The MIF performs a read operation with the new address and gets a 16-bit value which is stored in the RAM as the value B. It then forwards the 16-bit value to the accumulator and stores it there temporarily. The code goes as follows:

Listing 4: Lode State

```
when load => op_code   <= op_code_IR;--send op-code to
    MIF
        address_out   <= address_in;--send address to MIF
        en_mif   <= '1';--enable MIF to perform load
            operation
        oe_r <= '1';
        accu_en  <= '1';--enable accumulator
        accu_op_code <= "00";--enable the accumulator to
             accept data
```

The Fetch and Decode gets executed again with the updated values of the Program counter and the next 16-bit Instruction to decode is made available to the Control Unit by the Instruction Register(IR) which then gets decoded to find out the next state.

5) Add State: In this state the 8ibit address and the op code is again sent to the Memory Interface which then performs a read operation on the RAM which gives out a 16-bit value which is stored as C. This value is then sent to the Arithmetic Logic Unit as one of the inputs and then the control unit signals the Accumulator to send the value stored in it which is B to the ALU as the other input. The ALU also receives the Op code which in this case is for addition and performs an add operation and then stores the result back in to the Accumulator. This state has been divided into two states so that all the following operations can take place. The code goes as follows:

Listing 5: Add & Add1 state

```
when add => op_code <= op_code_IR;--send op-code to MIF
        address_out   <= address_in;--send address to MIF
        oe_r   <= '1';
        accu_en  <= '0';--enable accumulator
        en_mif   <= '1';--enable MIF to perform add
            operation
        alu_op_code   <= op_code_IR;--op code to alu to
            perform add opreration
when add1 => accu_en  <= '1';--enable accumulator
        accu_op_code  <= "10";--take data from the alu
            and keep it in the accumulator
```

The cycle follows again with the Fetch and Decode State running again with the updated values, fetches the next instruction, splits it and send it to the control unit whic decodes the instruction to find out the next state.

6) Store State: In this state the Control Unit instructs the Accumulator to send the result which is A to the the Memory Interface and it signals the Memory Interface to store the sent 16-bit result to a particular address.

Listing 6: Store state

```
when store=>accu_en  <= '0';--enable accumulator
        op_code   <= op_code_IR;--send op-code to MIF
        address_out   <= address_in;--send address to MIF
        en_mif   <= '1';--enable MIF to perform store
            operation
when store1 =>op_code  <= op_code_IR;--send op-code to
    MIF
        address_out   <= address_in;--send address to MIF
        en_mif  <= '1';--enable MIF to perform store
            operation
```

This above mentioned flow is performed here to solve the equation A = B + C where the values of B and C are stored in the RAM at particular locations.

A = B + C			
Instrucion Address	Instruction	Machine Code	Description
0x00	LOAD	x0208	ACC[register_accum] <= B
0x01	ADD	x0009	ALU = C + ACC[register_accum]=> ACC
0x02	STORE	X010F	RAM <= @0x0F <= A
0X03	JUMP	x0303	Loop at location x 03

The secode equation is : If A >= 0 then B = C Here after adding the two number B and C the result is stored in a location. We now load the result A in the accumulator and check if the value of A is greater than 0. If the value is less than 0 then it goes to the JNEG state which means jump if negative and it loops in that particular location forever. If the value is greater than 0 then we load C into the accumulator and then store the value of C in the location of B which then satisfies A > 0 then B = C. Below in the table show you can see the respective machine code and instructions that get executed.

Listing 7: JNEG state

```
when jneg =>if overflow = '1' then
          address_out <= address_in;
          pc_op_code     <= "11";--jump to address
        else
          null;
        end if;
```

If A > 0 then B = C			
Instrucion Address	Instruction	Machine Code	Description
0x03	LOAD	x020F	ACC[register_accum] <= A
0x04	JNEG	x0404	A < 0 then Loop forever
0x05	LOAD	x0209	ACC[register_accum] <= C
0x06	STORE	X0108	RAM <= @0x08 <=C (B=C)
0X07	JUMP	x0307	Loop at location x 07

VIII. Simulations

Her are the figures which represent the simulation results:

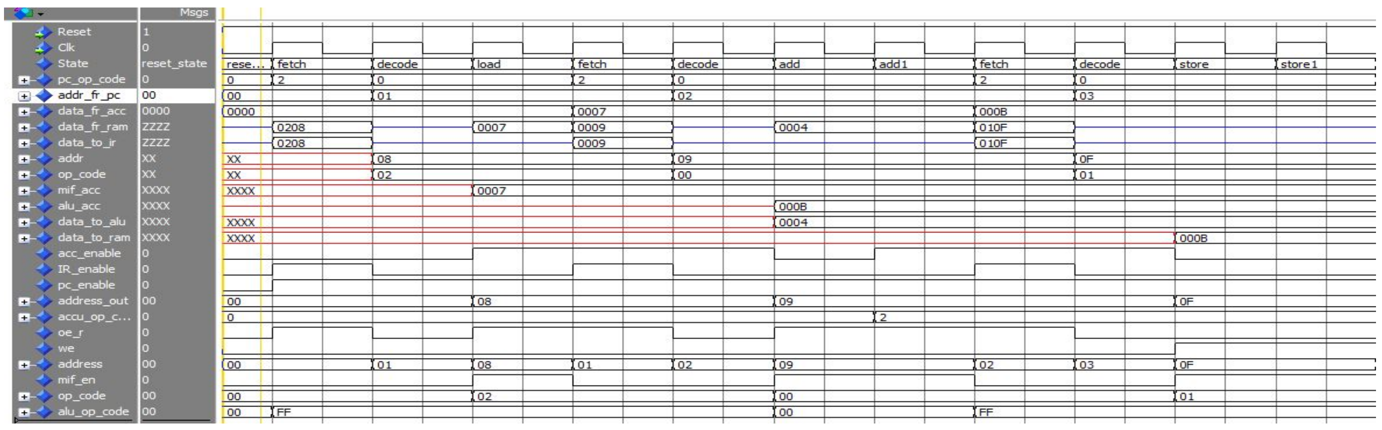


Figure 22: Simulation of A =B + C Instruction

Here we implement the JNEG which means jump if negative instruction.

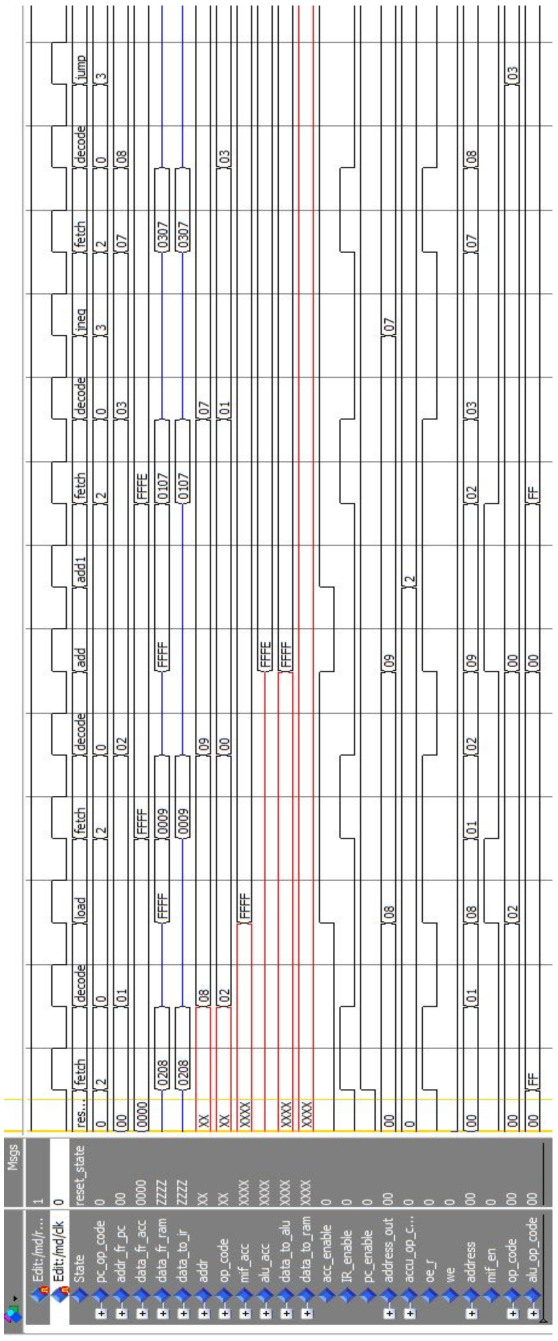


Figure 23: Simulation of JNEG Instruction

Here we can see that the if the value of A is greater than 0 then B= C

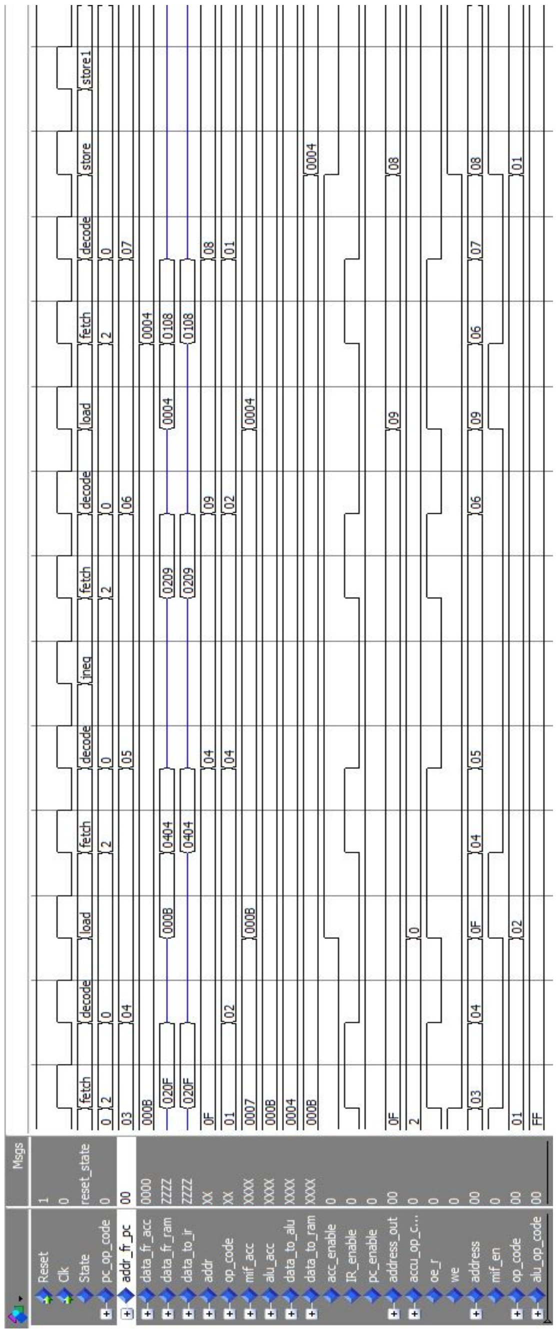


Figure 24: Simulation of A > 0 then B = C Instruction

IX. Conclusion & Evaluation

In this paper, we use a combination of "top-down" and "bottom-up" design technique which is sometimes called "sideways" to design a dedicated 16-Bit Microprocessor in which we design an instruction fetch unit, a decoder unit, and an instruction execution unit along with the different registers involved for their operations. VHDL design has been used to describe the system. Pipelining of instructions has been implemented but with just one line from the figure 6 mentioned. The 16-Bit Microprocessor has in-built instruction set designed to operate and execute the two equations mentioned. The design has been verified through the simulation results. The design, synthesis and simulation of the dedicated 16-bit Microprocssor has been achieved using Intel Quartus and ModelSim Altera for the simulation. I am also going to implementing this on the hardware using Cyclone V 5SEMA5F31C6 in DE1_SoC platform. The applications of microprocessors based on FPGA's are endless as they are so powerful and can be used to perform various operations. It can be used in areas where lots of data needs to be processed

X. Appendix A. Source Code

A. *Program Counter*

Listing 8: Program Counter Code

```
--Final Project - 16-BIT MICROPROCESSOR
--PROGRAM COUNTER
--Libraries
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity p_cnt is port(
    clk         : in std_logic;
    pc_en           : in std_logic;
    pc_op_code  : in    std_logic_vector(1 downto 0);
    pc_in           : in    std_logic_vector(7 downto 0)
        ;
    pc_out      : out std_logic_vector(7 downto 0)
);
end p_cnt;

architecture rtl of p_cnt is
    signal pc   : std_logic_vector(7 downto 0) := x"00";
         --set it to 0
begin
    process(clk,pc_op_code,pc_en,pc_in)
        begin
    if pc_en = '0' then
        pc <= x"00";
    elsif rising_edge(clk) then
        if pc_en = '1'then
            if pc_op_code = "10" then
                pc <= std_logic_vector(unsigned(pc) + 1)
                    ; --increment PC by 1
            elsif pc_op_code = "11" then
                pc <= pc_in;
            end if;
        end if;
    end if;
end process;
pc_out <= pc;
end rtl;
```

B. *Instruction Register*

Listing 9: Instruction Register Code

```
--Final Project - 16-BIT MICROPROCESSOR
--INSTRUCTION REGISTER
--Libraries
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity instr_reg is port(
    clk             : in std_logic;
    reset               : in std_logic;
    instr_reg_en    : in std_logic;
    instruction     : in std_logic_vector(15 downto 0);
    address         : out std_logic_vector(7 downto 0);
    op_code         : out std_logic_vector(7 downto 0)
);
end instr_reg;

architecture rtl of instr_reg is
signal add : std_logic_vector(7 downto 0);
signal op : std_logic_vector(7 downto 0);

begin
    process(clk,reset,instr_reg_en,instruction,add,op)
        begin
            if reset = '0' then
                add <= x"00";
                op <= x"00";
            elsif rising_edge(clk) then
                if instr_reg_en = '1' then
                    add <= instruction(7 downto 0);-- to
                         Program Counter
                    op <= instruction(15 downto 8);-- to
                         control unit
                end if;
            end if;
    end process;
op_code <= op;
address <= add;
end rtl;
```

C. *Accumulator*

Listing 10: Accumulator Code

```
    --Final Project - 16-BIT MICROPROCESSOR
--ACCUMULATOR
--Libraries
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity accum is port(
    clk         : in std_logic;
    reset           : in std_logic;
    mif_acc     : in std_logic_vector (15 downto 0); --
         data from the memory
    accu_en     : in std_logic;
    acc_op_code : in std_logic_vector(1 downto 0); --op
         code from the control unit
    alu_acc     : in std_logic_vector (15 downto 0);

    --accu_out_mem: out std_logic_vector (15 downto 0)
         ;--data to MIF
    data_out        : out std_logic_vector (15 downto 0)
          --data to memory or ALU
);
end accum;

architecture rtl of accum is
signal register_accum: std_logic_vector(15 downto 0) :=
     x"0000";

begin
    process(clk,reset,accu_en,acc_op_code,mif_acc,
         alu_acc)
    begin
        if reset = '0' then
            register_accum <= x"0000";
        elsif rising_edge(clk) then
            if accu_en = '1' then
```

```
                if acc_op_code = "00" then -- load
                    operation take data
                    register_accum  <= mif_acc;
                elsif acc_op_code = "10" then -- store
                    operation send data from accum to
                    mif
                    register_accum <= alu_acc;
                end if;
            end if;
        end if;
    end process;
data_out <= register_accum;
end rtl;
```

D. Memory Interface

Listing 11: Memory Interface Code

```
--Final Project - 16-BIT MICROPROCESSOR
--MEMORY INTERFACE
--Libraries
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity mif is port(
    we              : out std_logic;
    data_fr_ram : in std_logic_vector(15 downto 0);
    data_fr_acc : in std_logic_vector(15 downto 0);
    addr_fr_cu  : in std_logic_vector(7 downto 0);
    mif_op_code : in std_logic_vector(7 downto 0);
    en              : in std_logic;
    addr_fr_pc  : in std_logic_vector(7 downto 0);
    addr_to_ram : out std_logic_vector(7 downto 0);
    data_to_ram : out std_logic_vector(15 downto 0);
    data_to_ir  : out std_logic_vector(15 downto 0);
    data_to_acc : out std_logic_vector(15 downto 0);
    data_to_alu : out std_logic_vector(15 downto 0)
    );
end mif;

architecture rtl of mif is
signal value_disp : std_logic_vector(15 downto 0) := x"
    0000";
begin
process(en,mif_op_code,addr_fr_pc,data_fr_ram,addr_fr_cu
    ,data_fr_acc,value_disp)
begin
    we <= '0';
    addr_to_ram <= x"00";
    data_to_ram <= x"0000";
    data_to_ir  <= x"0000";
    data_to_acc <= x"0000";
    data_to_alu <= x"0000";
    value_disp <= x"0000";
    case en is
    when '0' =>
        addr_to_ram <= addr_fr_pc;
        data_to_ir <= data_fr_ram;
        we <= '0';
    when '1' =>
        case mif_op_code is
        when "00000010" => -- load
            addr_to_ram <= addr_fr_cu;
            data_to_acc <= data_fr_ram;

        when "00000000" => --add
            addr_to_ram  <= addr_fr_cu;
            data_to_alu <= data_fr_ram;

        when "00000001" => --store
            we <= '1';
            addr_to_ram <= addr_fr_cu;
            data_to_ram <= data_fr_acc;

        when "00000011" => --jump
            addr_to_ram <= addr_fr_pc;
            value_disp <= data_fr_ram;

        when others => null;
        end case;
    when others => null;
    end case;
end process;
end rtl;
```

E. Arithmetic Logic Code

Listing 12: Arithmetic Logic Unit

```
--Final Project - 16-BIT MICROPROCESSOR
--ARITHMETIC LOGIC UNIT
--Libraries
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity alu is port(
    data_a  : in  std_logic_vector(15 downto 0);
    op_code : in  std_logic_vector(7 downto 0);
    data_b  : in  std_logic_vector(15 downto 0);
    alu_out : out std_logic_vector(15 downto 0);
    zero_flag: out std_logic;
    Overflow : out std_logic
);
end alu;

architecture  rtl of alu is
signal ALU_Result : std_logic_vector (15 downto 0);
signal temp         : std_logic_vector(16 downto 0);
begin
process (op_code,data_a,data_b,ALU_Result,temp)
begin
    case op_code is
        when "00000000" => temp <= std_logic_vector(
            signed(data_a(15) & data_a) + signed(data_b
            (15) & data_b));
                ALU_Result <= temp(15 downto 0);
                Overflow <=  temp(16);

        when "00000001" => temp <= std_logic_vector(
            signed(data_a(15) & data_a) - signed(data_b
            (15) & data_b));
                ALU_Result <= temp(15 downto 0);
                Overflow <= temp(16);

        when "00000010" => ALU_Result <= data_a and
            data_b;

        when "00000011" => ALU_Result <= data_a or
            data_b;

        when others => ALU_Result <= data_a;
        end case;
if (ALU_RESULT = x"0000") then
    zero_flag <= '1';
else
    zero_flag <= '0';
end if;
end process;
alu_out <= ALU_Result;
end rtl;
```

F. Memory Unit - RAM

Listing 13: Memory Unit - RAM

```
--Final Project - 16-BIT MICROPROCESSOR
--RAM
--Libraries
library ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE ieee.std_logic_unsigned.all;

entity ram is port(
    oe      : in std_logic;
    clk     : in std_logic;
    reset       : in std_logic;
    we      : in std_logic;
    address : in std_logic_vector(7 downto 0);
    data_in : in std_logic_vector(15 downto 0);
    data_out    : out std_logic_vector(15 downto 0)
    );
end ram;

architecture rtl of ram is
--create an array
type memory_type is array (0 to 255) of std_logic_vector
     (15 downto 0);
signal memory : memory_type := (
                        0 => "0000001000001000",--load-
                            opcode address(8)-load B
                         1 => "0000000000001001",--add-
                           opcode address(9)-add C
                       2 => "0000000100000111",--store-
                          opcode address(15)-store in A

                       3 => "0000001000001111",--load-
                          opcode address(15)-load A
                       4 => "0000010000000100",--check for
                          jneg
                       5 => "0000001000001001",--load C
                       6 => "0000000100001000",--store C in
                           B
                       7 => "0000001100000111",--keep
                          jumping to (4) if negative
                       8 => "0000000000000111",--value B =
                          7
                       9 => "0000000000000100",--value C =
                          4
                        others => x"0000"

                   );
begin
process(clk,reset,we,address,data_in)
begin
if reset = '0' then
    if rising_edge(clk) then
        if (we = '1') then
            --Write Operation
            memory(conv_integer(address)) <= data_in;
        end if;
    end if;
end if;
end process;
--Read operation and convert index to an integer
data_out <= memory(conv_integer(address)) when oe = '1'
     else (others => 'Z');
end rtl;
```

G. *Control Unit*

Listing 14: Control Unit Code

```
--Final Project - 16-BIT MICROPROCESSOR
--CONTROL UNIT
--Libraries
library ieee;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

Entity Cntrl_Unit is port(
    clk   : in std_logic;
    reset : in std_logic;
    address_in : in std_logic_vector(7 downto 0);
    op_code_IR: in std_logic_vector(7 downto 0);
    oe_r : out std_logic := '0';
    address_out : out std_logic_vector(7 downto 0) := x"
        00";
    op_code : out std_logic_vector(7 downto 0) := x"00";
    en_mif : out std_logic := '0';
    pc_ena : out std_logic := '0';
    pc_op_code : out std_logic_vector(1 downto 0) := "00
        ";
    alu_op_code : out std_logic_vector(7 downto 0) := x"
        00";
    accu_op_code : out std_logic_vector(1 downto 0) := "
        00";
    accu_en : out std_logic := '0';
    en_ir : out std_logic := '0';
    overflow : in std_logic
);
end Cntrl_Unit;

Architecture FSM of Cntrl_Unit is
type state_type is (reset_state,fetch,decode,add,add1,
    store,store1,load,jump,jneg);
--Signals
signal state: state_type;
begin
    demo_process : process(clk,reset,op_code_IR,overflow
        )
    begin
        if(reset = '0') then
            state <= reset_state;
        elsif(rising_edge(clk)) then
            case state is
                when reset_state  => state <= fetch;
                when fetch  => state <= decode;
                when decode => case op_code_IR (7 downto
                     0) is
                        when x"00" => state <= add;
                        when x"01" => if overflow = '0'
                             then
                                   state <= store;
                              elsif overflow = '1'
                                  then
                                   state <= jneg;
                              end if;
                        when x"02" => state <= load;
                        when x"03" => state <= jump;
                        when x"04" => state <= jneg;
                        when others => state <= fetch;
                        end case ;
                when add => state   <= add1;
                when add1=> state <= fetch;
                when store => state <= store1;
                when store1 => state <= fetch;
                when load => state <= fetch;
                when jump => state <= fetch;
                when jneg => state <= fetch;
                when others => state <= fetch;
            end case;
        end if;
end process;

output_process : process(state,op_code_IR,address_in)
    begin
--default values for all the output signals
    oe_r    <= '0';
    address_out <= address_in;
    op_code <= op_code_IR;
    en_mif <= '0';
    pc_ena <= '0';
    pc_op_code <= "00";
    alu_op_code <= x"00";
    accu_op_code <= "00";
    accu_en <= '0';
    en_ir <= '0';
    case state is
        when reset_state    => pc_ena <= '0';--reset PC
                  oe_r <= '0';--disable read operation

            when fetch => pc_ena  <= '1';--enable PC
                    en_mif  <= '0';--mif performs read
                        and fetches the data
                    oe_r  <= '1';--read enable for the
                        ram
                    en_ir  <= '1';--enable IR to split
                        instruction
                    pc_op_code   <= "10";--send address
                        to mif
                    alu_op_code  <= "00011111";

            when decode => en_ir  <= '0';
                    pc_op_code   <= "00";--send address
                        to mif
                    oe_r  <= '0';--Disable memory read

            when add => op_code <= op_code_IR;--send op-
                code to MIF
                    address_out  <= address_in;--send
                        address to MIF
                    oe_r  <= '1';
```

```
                     accu_en   <= '0';--enable accumulator
                     en_mif   <= '1';--enable MIF to
                          perform add operation
                     alu_op_code   <= op_code_IR;--op code
                           to alu to perform add
                          opreration

             when add1 => accu_en   <= '1';--enable
                  accumulator
                     accu_op_code  <= "10";--take data
                          from the alu and keep it in the
                           accumulator

             when store   => accu_en  <= '0';--enable
                  accumulator
                     op_code  <= op_code_IR;--send op-code
                           to MIF
                     address_out   <= address_in;--send
                          address to MIF
                     en_mif   <= '1';--enable MIF to
                          perform store operation

             when store1 => op_code   <= op_code_IR;--send
                   op-code to MIF
                     address_out   <= address_in;--send
                          address to MIF
                     en_mif   <= '1';--enable MIF to
                          perform store operation

             when load => op_code  <= op_code_IR;--send op
                  -code to MIF
                     address_out   <= address_in;--send
                          address to MIF
                     en_mif   <= '1';--enable MIF to
                          perform load operation
                     oe_r   <= '1';
                     accu_en   <= '1';--enable accumulator
                     accu_op_code  <= "00";--enable the
                          accumulator to accept data

             when jump => address_out   <= address_in;--
                  send address to PC
                     op_code   <= op_code_IR;--send op-
                          code to MIF
                     pc_op_code    <= "11";--jump to
                          address


             when jneg => if overflow = '1' then
                         address_out  <= address_in;
                         pc_op_code   <= "11";--jump to
                              address
                    else
                         null;
                    end if;

             when others => null;
         end case;
     end process;
end FSM;
```

List of Figures

References

[1] Microprocessor Design Print Version, retrieved from: `https://www.kth.se/social/upload/62/microprocessor_design.pdf`

[2] Digital Logic and Microprocessor Design With VHDL - Enoch O. Hwang ,retrieved from: `https://theswissbay.ch/pdf/Gentoomen%20Library/Digital%20Design Digital%20Logic%20%26%20Microprocessor %20Design%20With%20VHDL%20-%20Hwang.pdf`

[3] An Embedded Low Power/Cost 16-Bit Data/Instruction Microprocessor Compatible with ARM7 Software Tools - Fu-Ching Yang & Ing-Jer Huang, retrieved from: `http://cecs.uci.edu/p̃apers/aspdac07/pdf/p902_9B-2.pdf`

[4] "Migrating from 8- to 16-bit processors", `Bannatyne, R., in proc. Northcon /98 conf., pp. 150-158, Oct. 1998`

[5] Techtarget, retrieved from: `https://whatis.techtarget.com/definition/program-counter/`

[6] Instruction Register, retrieved from: `https://en.wikipedia.org/wiki/Instruction_register`

[7] Accumulator, retrieved from: `https://en.wikipedia.org/wiki/Accumulator_(computing)r`

[8] Arintmetic Logic Unit, retrieved from: `http://vlab.amrita.edu/?sub=3&brch=81&sim=604&cnt=1r`

[9] Image, retrieved from: `http://labs.domipheus.com/blog/wp-content/uploads/2015/07/highlevel.`