FACULTY OF ENGINEERING AND NATURAL SCIENCES

A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL

Thesis By

Ömer KARSLIOĞLU

January, 2022

ANKARA

A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL

A Thesis Submitted to

Faculty of Engineering and Natural Sciences of Ankara Yıldırım Beyazıt University

In Partial Fulfillment of the Reqeuirement for the Degree of Bachelor in Electrical & Electronics Engineering, Department of Electrical & Electronics Engineering

by

Ömer KARSLIOĞLU

January, 2022

ANKARA

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Asst. Prof. Gökhan Koray GÜLTEKİN for the support and motivation that my project progressed. His knowledge, experience and advice contributed to the progress of this project. His guidance has helped me during the research and this thesis at all times.

2022 , 01 January **Ömer KARSLIOĞLU**

A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL

ABSTRACT

The general aim of the project is a computer system that can process the basic written code, perform logical-arithmetic operations according to the written code, and as a result, make decisions. I took the Harvard Architecture as a reference in my design because of its advantages such as the separate existence of instruction memory and data memory, and the fact that an instruction runs in a single clock cycle. I preferred the RISC architecture in my RISC design because the pipeline technique is used as the instruction set architecture and it has less and uncomplicated instructions. This project consists of two main modules, CPU and memory, and a computer system with 8 bit register sizes has been designed within the scope of the determined target.

Keywords: ISA, RISC, Harvard Architecture, CPU, Controller Unit, Data Path, Memory, RAM, ROM, Input-Output Port, ALU, Register, Multiplexer, HDL, RTL, Combinational Circuit, Sequantial Circuit, Opcode, Operand, Datapath, Pipeline, Fetch-Decode-Execute

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACRONYYMS	FULL FORM
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computer
HDL	Hardware Description Language
IR	Instruction Architecture
PC	Program Counter
MAR	Memory Address Register
CCR	Condition Code Register
ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
RAM	Read Access Memory
ROM	Read Only Memory
IMM	Immediate Addressing
DIR	Direct Addressing

1 – Introduction

1.1 - Computer General Working Principle

Computers are among the most advanced digital systems today. In order to understand the general working structure of computers, we need to examine two basic concepts: hardware and software . Computers are made up of software that does not combine various hardware and establishes/manages relations between these hardware. At the center of all computers is the CPU, which processes the instruction, which is the most basic set of meaningful code that the computer can understand, and activates and shuts down other units accordingly, thus enabling the computer to perform the most basic operations. There are two basic structures in the CPU. The first one is the control unit controls the whole system according to inputs datas (instruction , program counter , contition etc.) , and the second is the data path, which provides the program flow with various units such as registers and ALUs according to the information received from the control unit.

One of the most important units for the operation of a computer is the memory unit. In the memories, there is the code body that the computer will perform its operations, the instructions to generate the codes, and the temporary data that occurs while the computer is running. Although the areas where these data are located are different from each other, they are separate memories (RAM, ROM etc.) that enable the computer to work. According to the data coming from the CPU, data can be written to the memory or data can be read. In this way , the codes taken from the program memory are processed sequentially in the computer .

1.2 - Computer System Designed In The Project

Designed computer system can perform basic arithmetic and logical operations such as addition, subtraction, and, or, by retrieving code from program memory, and determining states with branches. In line with the objectives, this computer design was designed on an 8-bit basis, Harvard Architecture and RISC was chosen as computer architecture and ISA.

In general, the design consists of the combination of two main modules, CPU and memory, and many modules coming together in a hierarchical manner. I have mentioned these modules in detail in the following pages of my report .You can examine the block diagram below to see the combination of the main modules.

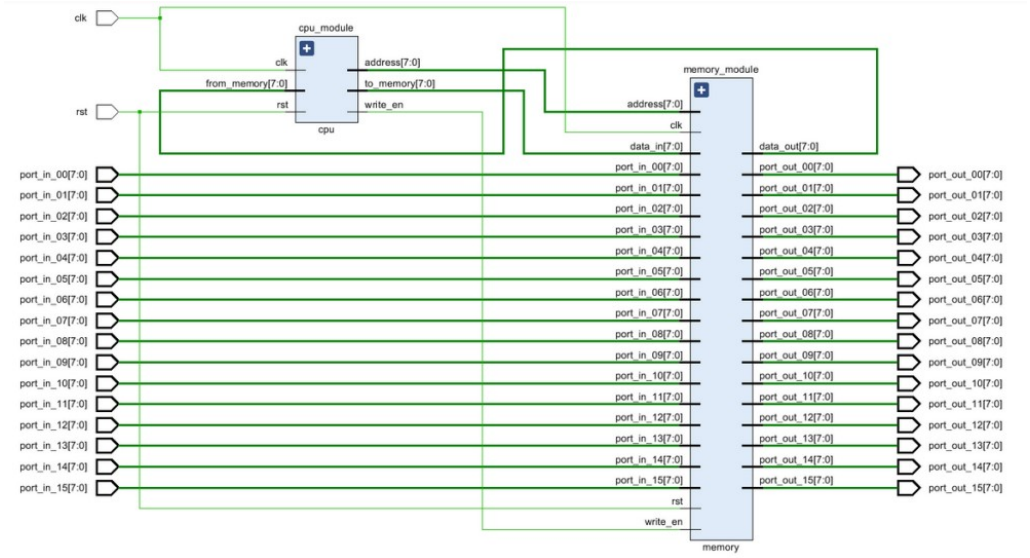

Figure 1 - Computer System Top Level Block Schematic : They are two main modules obtained after system design and elaborating in Xilinx Vivado.

1.2.1 - Harvard Architecture

The Harvard architecture is a computer architecture with separate storage and signal pathways for instructions and data. It contrasts with the von Neumann architecture, where program instructions and data share the same memory and pathways. The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters. These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data. Programs needed to be loaded by an operator; the processor could not initialize itself. Modern processors appear to the user to be von Neumann machines, with the program code stored in the same main memory as the data. For performance reasons, internally and largely invisible to the user, most designs have separate processor caches for the instructions and data, with separate pathways into the processor for each. This is one form of what is known as the modified Harvard architecture.In a Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions for preprogrammed tasks can be stored in read-only memory while data memory generally requires read-write memory. In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.

Instruction memory and data memory are separate units, the CPU can read instructions and provide memory access at the same time without a cache, an instruction is processed in one clock cycle instead of two, it is generally suitable for use in MCUs and signal processing. So I decided to use "Harvard Architecture".

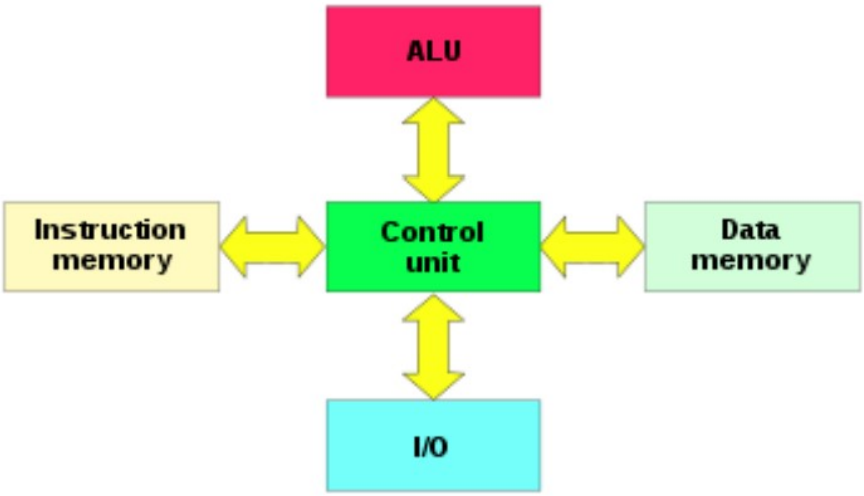

Figure 2 - Harvard Architecture : The separation of instruction memory and data memory is one of the most important features of Harvard Architecture.

1.2.2 – RISC

A Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions rather than the highlyspecialized set of instructions typically found in other architectures. RISC is an alternative to the Complex Instruction Set Computing (CISC) architecture and is often considered the most efficient CPU architecture technology available today. With RISC, a central processing unit (CPU) implements the processor design principle of simplified instructions that can do less but can execute more rapidly. The result is improved performance. A key RISC feature is that it allows developers to increase the register set and increase internal parallelism by increasing the number of parallel threads executed by the CPU and increasing the speed of the CPU's executing instructions. ARM, or “Advanced RISC Machine” is a specific family of instruction set architecture that’s based on reduced instruction set architecture developed by Arm Ltd. Processors based on this architecture are common in smartphones, tablets, laptops, gaming consoles and desktops, as well as a growing number of other intelligent devices.

RISC provides high performance per watt for battery operated devices where energy efficiency is key. A RISC processor executes one action per instruction. By taking just one cycle to complete, operation execution time is optimized. Because the architecture uses a fixed length of instruction, it's easier to pipeline. And because it lacks complex instruction decoding logic, it supports more registers and spends less time on loading and storing values to memory. For chip designers, RISC processors simplify the design and deployment process and provide a lower per-chip cost due to the smaller components required. Because of the reduced instruction set and simple decoding logic, less chip space is used, fewer transistors are required, and more general-purpose registers can fit into the central processing unit.

I chose RISC for this design because the execution time is one cycle, it is far from complex instructions, and it provides faster data processing using the pipeline technique.

1.2.3 - Design Environment and Tools Used

I chose Xilinx Vivado as the environment I will design for testing , simulation , synthesis operations , elaborating the schematic and implementing it in an FPGA . I made my design in VHDL language.

2 - Computer Hardware System

In this part, I will talk about the design of digital hardware design in computer system design and all the units in the design.Computer hardware refers to all of the physical components within the system. This hardware includes all circuit components in a computer such as the memory devices, registers, and finite state machines.

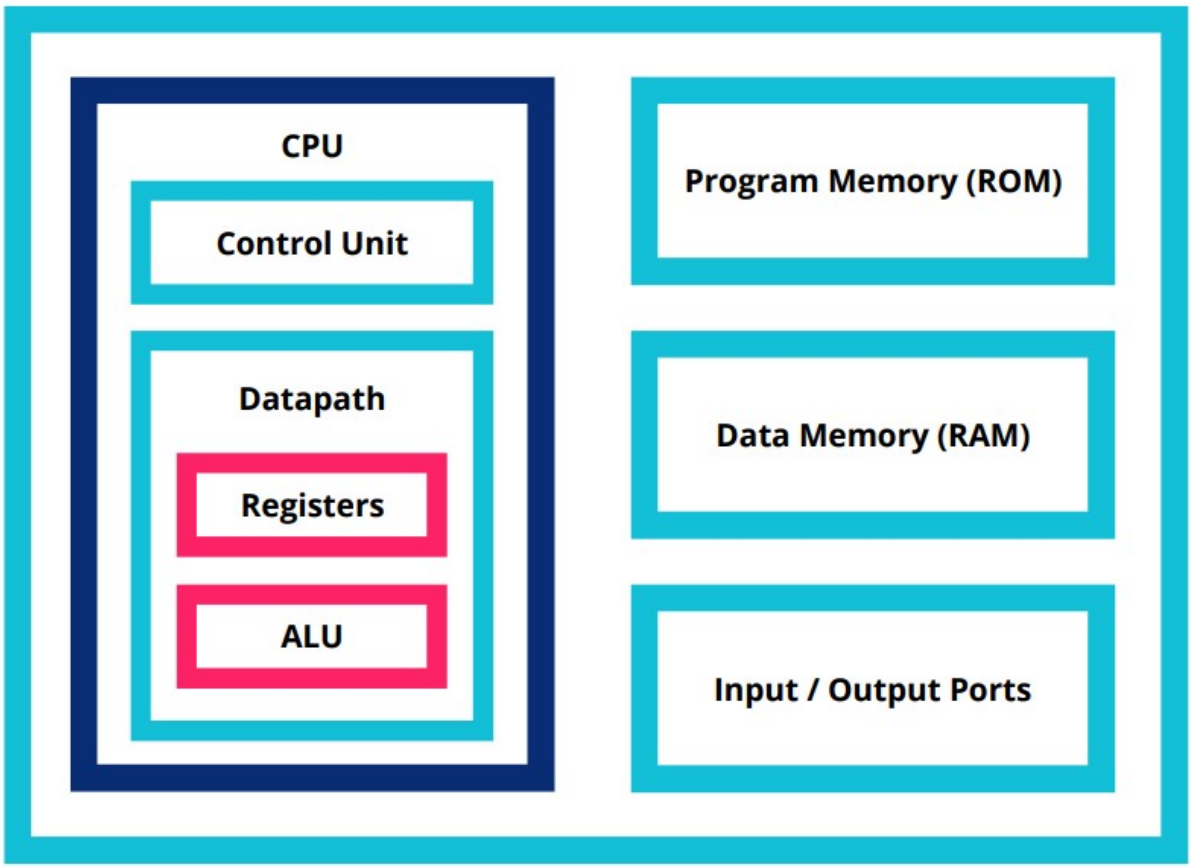

Figure 3 - Computer Hardware System General Schematic : It is the block schematized version of the computer system designed in this project. The computer design is designed as a hierarchical system as shown in the diagram above.

You can see the general scheme of the design in the picture above. We can think of computer system design as two main parts. The first of these is the CPU and the other is the Memory System. First, let's examine the CPU and the components that make up the CPU.

2.1 - Central Processing Unit (CPU)

The CPU can be defined as the brain that does the administrative work of the computer. When the units in it are examined, the working principle will be better understood. The CPU generally consists of two parts. The first of these is Control Unit. The other is Datapath.

2.1.1 - Control Unit

It is a state machine that provides fetching-decoding-execution of commands. This state machine is responsible for receiving the command from the relevant memory , detecting which command this command is and performing the related operation . Each instruction determines the operating mode of the CPU by changing the state of the state machine . The communication between Control Unit and Datapath takes place with control signals and status signals . The data path does what the control unit commands.

2.1.2 - Datapath

The CPU groups its registers and ALU into a sub-system called the data path. The data path refers to the fast storage and data manipulations within the CPU. All of these operations are initiated and managed by the control unit state machine. The CPU contains a variety of registers that are necessary to execute instructions and hold status information about the system. Basic computers have the following registers in their CPU:

2.1.2.1 - Registers

Registers are fast memory elements where data exchanges and manipulations are made. It is fast because it can read/write data in the same clock pulse. High priority data expected to be used at that moment are stored in the register . Non-priority data is stored in slow but large memory structures such as RAM. Five different registers were used in this computer design.

- **Instruction Register (IR):** It is the register that holds the 'binary' value of the next instruction to be processed. The instruction received in the program memory is kept in this register .
- **Memory Address Register (MAR):** The register where the address information required to reach the memory is kept and exported. If the CPU is going to read a data from the memory, the address to be read is held and transmitted by the MAR. The address information that MAR transmits to memory is needed for operations such as reading commands from memory or writing data to RAM.
- **Program Counter (PC):** It keeps the address of the next command to be processed in the program memory. Its value is zero when the processor starts, and the value increases by +1 after each instruction. In the program memory, the commands are stored and processed sequentially. PC holds this row. Whatever value the PC shows, the command kept at that address is read from the memory and transferred to the IR.
- **General Purpose Registers:** These registers are the types of registers that generally hold the data to be used in transactions.
- **Condition Code Register (CCR):** It keeps the status flags generated as a result of ALU operations. (Negative , Zero , Overflow and Carry) . The reason why this information is kept is for jump / branch commands in the conditional command type. For example, if there is an overflow, conditional operating modes and jumps can be added, such as restart the code on the computer.

2.1.2.2 - Arithmetic Logic Unit (ALU)

The arithmetic logic unit is the system that performs all mathematical (i.e., addition, subtraction, multiplication, and division) and logic operations (i.e., and, or, not, shifts, etc.). This system operates on data being held in CPU registers. The ALU has a unique symbol associated with it to distinguish it from other functional units in the CPU (see figure 4).

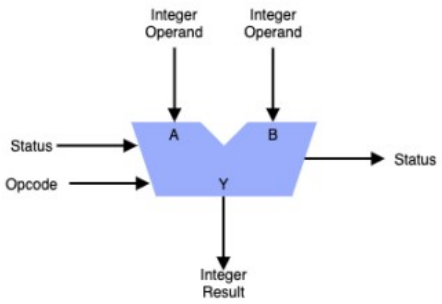

Figure 4 – ALU : ALU units are usually represented as in this figure. Generally, there are two entrances. All ALU designs must have an output and a status output to report the status.

2.2 - Memory System

The memory unit generally consists of four main parts. These consist of program memory, data memory and input-output ports. Communication between CPU and memory is provided by address, data and control signals between CPU and memory.

2.2.1 - Program Memory

The instructions to be processed by the computer to run the related program are kept in this memory by the computer. These commands are found in the ROM in their respective order. Because commands are processed sequentially. The program is of memory read only memory type. Otherwise , if data was written to the memory , the commands would be confused and the command operating system would not work . For this reason , the program consists of structures called memory rom - eprom - flashrom . Desired data is engraved on the ROM chips at the production stage. Even if the power of the computer is cut off, the code here cannot be deleted. Code cannot be loaded electrically. The desired program software in the design will be written into the program memory.

2.2.2 - Data Memory

This memory is used to hold temporary variables that are created by the software program. This memory expands the capability of the computer system by allowing large amounts of information to be created and stored by the program. Additionally, computations can be performed that are larger than the width of the computer system by holding interim portions of the calculation. Data memory is implemented with R/W memory, most often SRAM or DRAM.

2.2.3 - Input/Output Ports

The term port is used to describe the mechanism to get information from the output world into or out of the computer. Ports can be input, output, or bidirectional.

Although the memories are more than one, separate address signals/wires do not go to the memory. There is only one address line. In the design, instead of using four different memory structures, a single large memory was used and this memory was divided into four parts (Program Memory , Data Memory , Input - Output Ports).

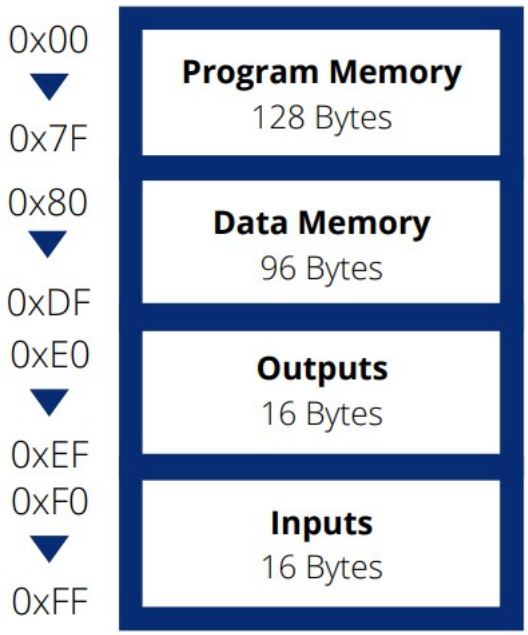

Figure 5 - Memory Mapped : The designed memory structure is as above. All memory can be accessed with a single address signal.

Each input/output port and each location in both program and data memory are assigned a unique address. This allows the CPU to access everything in the memory system with a dedicated address. This reduces the number of lines that must pass into the CPU. A bus system facilitates transferring information within the computer system. An address bus is driven by the CPU to identify which location in the memory system is being accessed. A data bus is used to transfer information to/from the CPU and the memory system. Finally, a control bus is used to provide other required information about the transactions such as read or write lines. The address bus between CPU and memory system can provide 256 unique locations. For this design, the memory system is also 8-bits wide, thus the entire memory system is 256x8 in size. In this design 128 bytes are allocated for program memory; 96 bytes are allocated for data memory; 16 bytes are allocated for output ports; and 16 bytes are allocated for input ports.

3 - Computer Software System

When a code is written to the computer, the written code represents a binary number. Binary codes, which are the machine language equivalent of the written code, are divided into parts by the computer. These binary commands, which were previously defined in the computer, are detected. The processor divides these commands into various parts and performs the operation by reading the relevant data in the relevant parts. For example, when a code is written to add the numbers a and b, the computer takes this code and processes the add command by dividing the values of the numbers a and b into parts, such as where the number resulting from the addition operation will be saved.

Computer software refers to the instructions that the computer can execute and how they are designed to accomplish various tasks. The specific group of instructions that a computer can execute is known as its instruction set. The instruction set of a computer needs to be defined first before the computer hardware can be implemented.

3.1 - Structure Of The Computer System Instruction

The instruction structure used in this design consists of two parts, opcode and operand. Each partition is 8 bits in size.

Figure 6 - Opcode & Operand : It is the instruction structure to be written into the program memory.

Opcode : The ID of the command. There is a special opcode value for each command. For example , the opcode value for the collection operation is 0x42 , and the opcode value for the upload operation is 0x86 .

Operand : It is the data required for the processing of the command (address information, integer value, etc.).

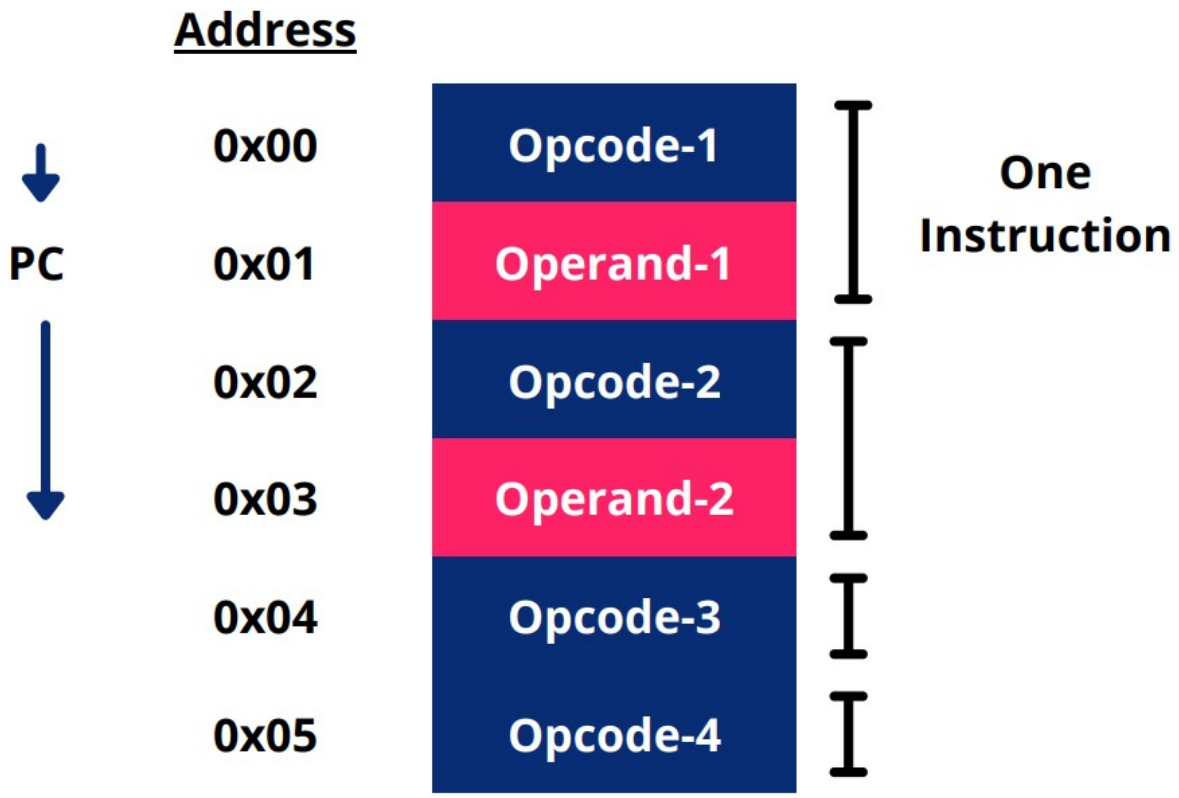

Figure 7 - Instruction Code Structure : It is a schematized version of the code structure consisting of instructions. The PC value increases by one after each opcode and operand.

As you can see above (figure 7), every instruction has an address definition. All of these instructions consist of either an opcode or both an opcode and an operand. The program counter starts from 0 and reads these opcodes and operands in the program memory, and then writes the read instruction to the instruction register in the datapath in the CPU. Afterwards, the PC increases by one and continues the operations in the same way.

3.2 - Processing Of Instructions

When the processor starts to process the command, three processes take place in sequence:

1- Fetch : Receiving the command from the program memory according to the value of the program counter.

2- Decode : It is to understand what the command is by the CPU by looking at the opcede value.

3- Execute : It is the execution of necessary mathematical/logical operations according to the purpose of the command.

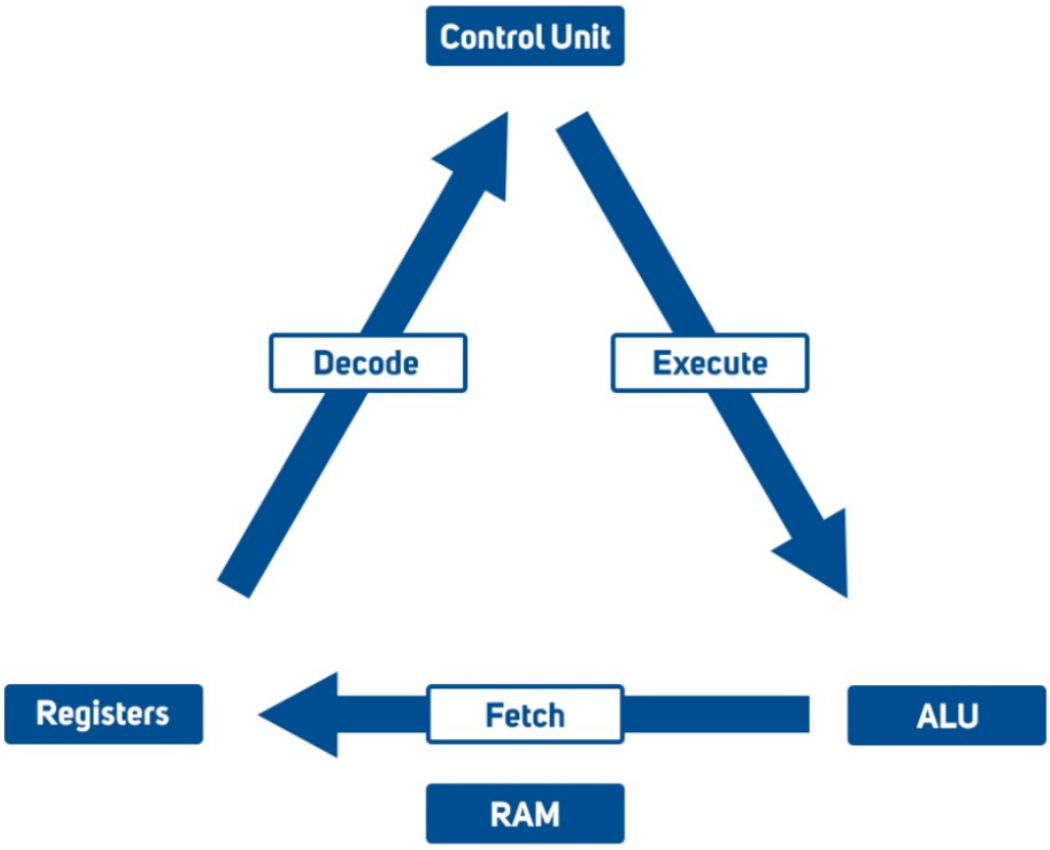

Figure 8 - Fetch-Decode-Execute : Each instruction must enter a fetch-decode-execute loop to be processed.

I would like to explain how the commands work with an example. In the image below, there is the opcode of the instruction at address 0 and its operand at the second address. When the program starts the PC is equal to 0 and reads the x"86" opcode and this value is transferred to the IR (Fetching). The control unit in the CPU understands that this is an immadiate loading to A register command and the PC increments to read the operand information it needs (Decoding). The decoding step is the detection of what the command is. The PC reads the value x"11" in the operand at the next clock pulse and loads/writes it to register A (Executing).

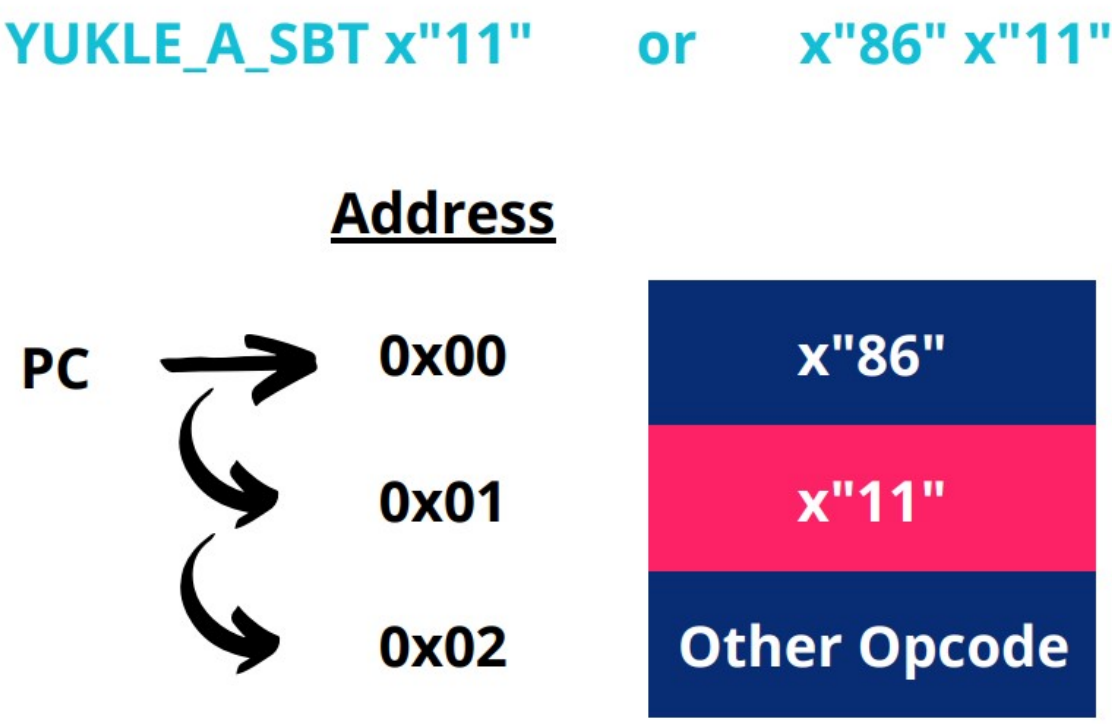

Figure 9 - Processing Of Instruction Example : It is a schematized structure of a piece of code written in Assembly language, as an instruction.

4 - Computer System Design And Implementation

This system is designed hierarchically . After the design of the lower modules , the design of the upper modules was realized .

Block names are addressed with the names of files with the ".vhd" extension. At the top level of the design is computer.vhd. There are two modules under the top level computer.vhd, cpu.vhd and memory.vhd.

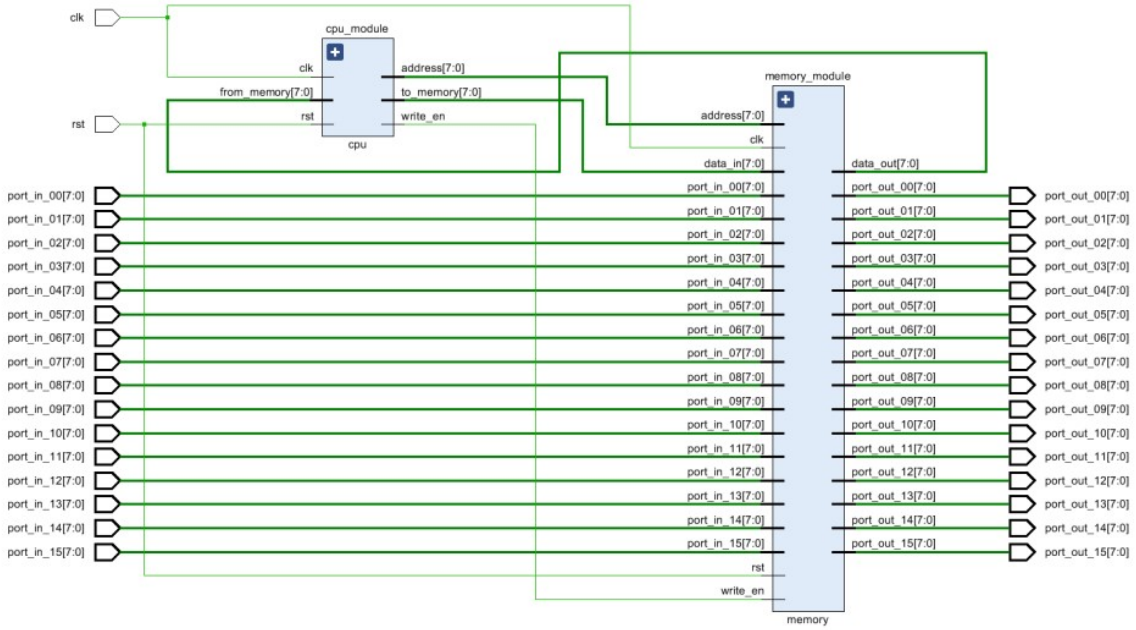

Figure 10 - Computer System Top Level Block Schematic : They are two main modules obtained after system design and elaborating in Xilinx Vivado.

There are 16 inputs and 16 outputs at the top level. It also includes CPU and memory.

The address signal, the write control signal, and the data signal are interconnected between the CPU and memory. Now let's examine the sub-blocks that make up the top level.

4.1 - Memory Design

Memory (memory.vhd) is the top module that combines the memory sub-blocks of the computer.

The sub-blocks of the memory.vhd module are program memory (ROM), data memory (RAM) and output ports. Each memory module inputs an address signal. Only data_in and write signals are input to data memory and output ports. Because only these memory elements can be written to data . Since the program is of memory ROM type, it does not need data entry and write permission. A 16x8 bit output is output from the output block. This enables the computer to export data to the outside world .

There is a multiplexer at the top level. The 8-bit data_out signal at the output of this multiplexer is the data signal going to the CPU. In other words, it is the data_out signal of the channel where the data read from the memory is sent. Multiplexer inputs are respectively 16x8 input ports (port_in_xx), data_out signal which is output of program memory and data_out signal which is output of data memory. Since the output port memory does not send any data to the CPU, it has no connection to the multiplexer.

The reason why the input ports are connected to the memory without being directly connected to the CPU and then go to the CPU according to the input of the multiplexer is the use of the routing method. In other words, input signals are transmitted from a single port (data_out) via memory in order not to pull 16 extra cables to the CPU. So the CPU gets the data from a single channel.

Another important issue is that the processors are designed sequentially. Therefore, it is imperative to design the design sequentially. Therefore, memory read and write operations are sensitive to clock pulses. As a result, the memory block is designed in a sequential structure and has clock and reset inputs. Since memory units are in a sequential structure, we can set and predict when data will be retrieved from memory.

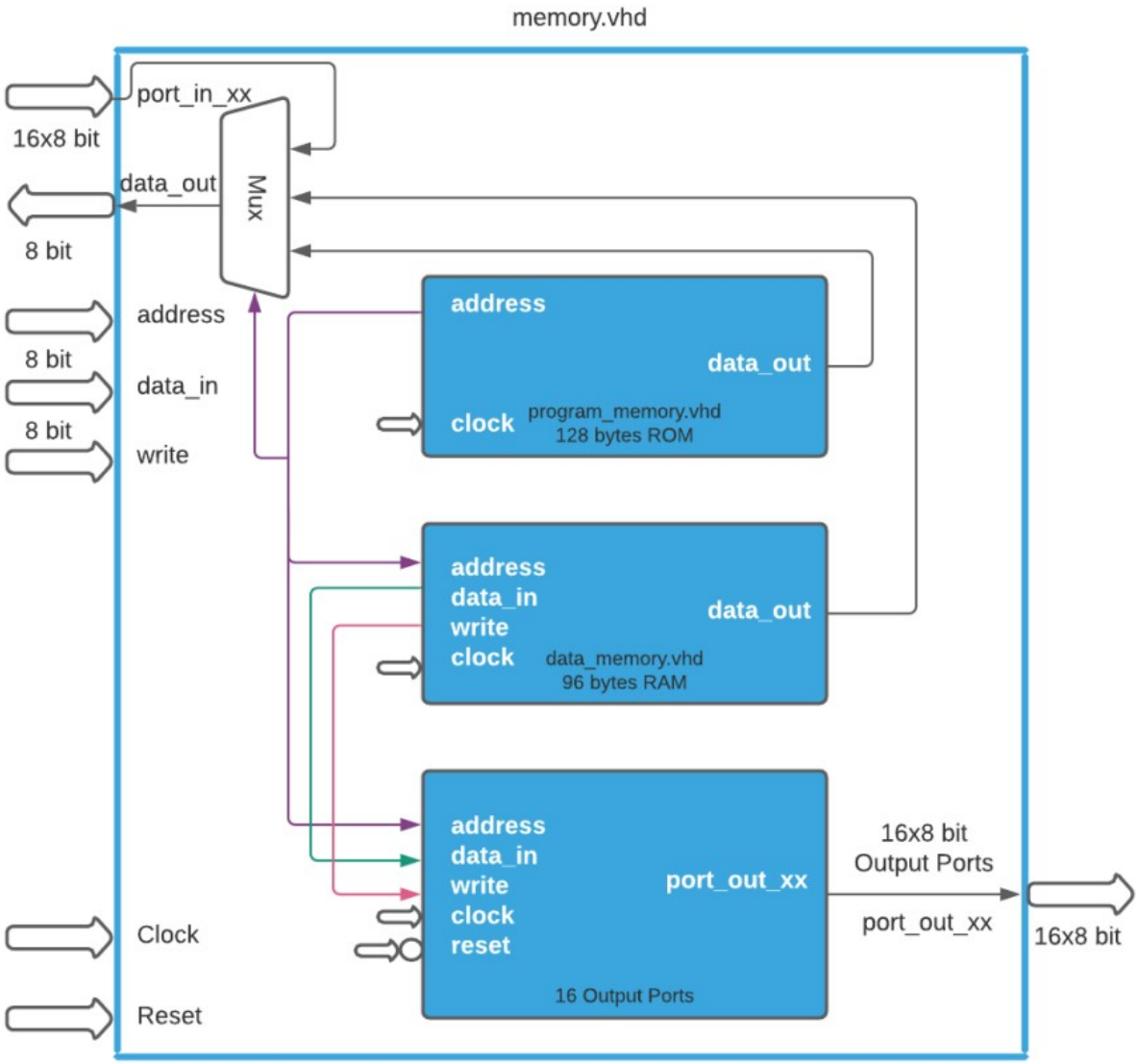

Figure 11 - Memory Module Schematic : It is schematized with all memory unit units in the memory block.

You can see the schematic of the memory structure I designed above (figure 11). The modules, hierarchical structure and connections of the memory structure are as above.

4.1.1 - Program Memory Design

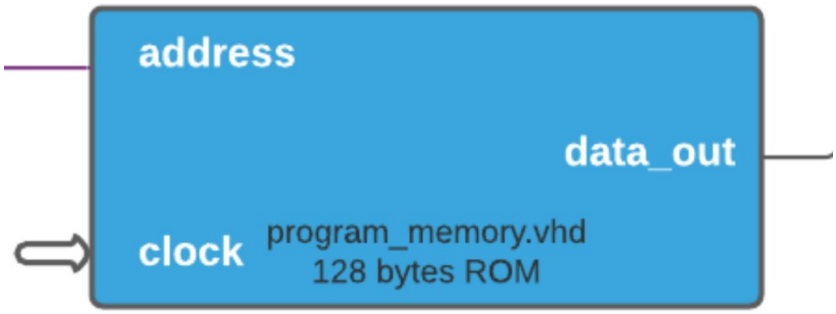

Figure 12 - Program Memory Module Schematic : The program memory block is schematized.

Program memory is a 128 byte ROM type memory that holds the instructions and the main program and outputs data according to the information coming from the CPU.

```
entity program_memory is
    port (
        clk          : in std_logic;
        address      : in std_logic_vector(7 downto 0);
        -- Output
        data_out     : out std_logic_vector(7 downto 0)
    );
end program_memory;
```

Figure 13 - Inputs & Output Of Program Memory In VHDL : It is the entity part in the VHDL code of the program memory.

4.1.1.1 - Instruction Set Architecture

All instructions are defined in program memory beforehand. To see the ISA used in the design, you can examine the instructions that I defined as constant in the architecture of the program memory in the vhdl language (see figure 13).

```
-- Loads and Stores Commands
constant YUKLE_A_SBT        :std_logic_vector(7 downto 0) := x"86"; -- Load Register A using Immediate Addressing
constant YUKLE_A            :std_logic_vector(7 downto 0) := x"87"; -- Load Register A using Direct Addressing
constant YUKLE_B_SBT        :std_logic_vector(7 downto 0) := x"88"; -- Load Register B using Immediate Addressing
constant YUKLE_B            :std_logic_vector(7 downto 0) := x"89"; -- Load Register B using Direct Addressing
constant KAYDET_A           :std_logic_vector(7 downto 0) := x"96"; -- Store Register A to Memory using Direct Addr.
constant KAYDET_B           :std_logic_vector(7 downto 0) := x"97"; -- Store Register B to Memory using Direct Addr.

-- Data Manipulations
constant TOPLA_AB           :std_logic_vector(7 downto 0) :=x"42";  -- A=A+B
constant CIKAR_AB           :std_logic_vector(7 downto 0) :=x"43";  -- A=A-B
constant AND_AB             :std_logic_vector(7 downto 0) :=x"44";  -- A=A&B
constant OR_AB              :std_logic_vector(7 downto 0) :=x"45";  -- A=A+B
constant ARTTIR_A           :std_logic_vector(7 downto 0) :=x"46";  -- A=A+1
constant ARTTIR_B           :std_logic_vector(7 downto 0) :=x"47";  -- B=B+1
constant DUSUR_A            :std_logic_vector(7 downto 0) :=x"48";  -- A=A-1
constant DUSUR_B            :std_logic_vector(7 downto 0) :=x"49";  -- B=B-1

-- Branches
constant ATLA               :std_logic_vector(7 downto 0) :=x"20";  -- Branch Always to Address Provided
constant ATLA_NEGATIFSE     :std_logic_vector(7 downto 0) :=x"21";  -- Branch to Address Provided if N=1
constant ATLA_POZITIFSE     :std_logic_vector(7 downto 0) :=x"22";  -- Branch to Address Provided if N=0
constant ATLA_ESITSE_SIFIR  :std_logic_vector(7 downto 0) :=x"23";  -- Branch to Address Provided if Z=1
constant ATLA_DEGILSE_SIFIR :std_logic_vector(7 downto 0) :=x"24";  -- Branch to Address Provided if Z=0
constant ATLA_OVERFLOW_VARSA:std_logic_vector(7 downto 0) :=x"25";  -- Branch to Address Provided if V=1
constant ATLA_OVERFLOW_YOKSA:std_logic_vector(7 downto 0) :=x"26";  -- Branch to Address Provided if V=0
constant ATLA_ELDE_VARSA    :std_logic_vector(7 downto 0) :=x"27";  -- Branch to Address Provided if C=1
constant ATLA_ELDE_YOKSA    :std_logic_vector(7 downto 0) :=x"28";  -- Branch to Address Provided if C=0
```

Figure 14 - Instruction Set Architecture : They are the opcodes defined to our CPU.

The definitions of the figure instructions above are expressed in comments. In this design, the size of the program memoy was kept small by using simple commands.

RISC is a simple and minimal architecture, where the bit positions of the opcode and operand are the same and fixed for each instruction. CISC is defined as architectures with complex commands and each command-specific opcode, operand, bit positions, and sizes are changeable.As you can see in this design, RISC architecture is used.

4.1.1.2 - ROM Type

```
type rom_type is array (0 to 127) of std_logic_vector(7 downto 0);
constant ROM : rom_type := (
                        0 => YUKLE_A_SBT,
                        1 => x"0F",
                        2 => KAYDET_A,
                        3 => x"80",
                        4 => ATLA,
                        5 => x"00",
                        others => x"00"
);
```

Figure 15 - ROM Type Design & Program : It is a sample code in the ROM.

Since the program is designed in memory rom type, data cannot be written into it, only data can be read from it. As seen in Figure 14, it was designed as 128 bytes and a simple code was written inside.

4.1.1.3 - Processes Of Program Memory

```
-- Signals :
signal enable : std_logic; -- to control interval of address
begin

process(address) begin
    if(address >=x"00" and address <= x"7F") then
        enable <= '1';
    else
        enable <= '0';
    end if;
end process;

process(clk) begin
    if(rising_edge(clk)) then
        if(enable = '1') then
            data_out <= ROM(to_integer(unsigned(address)));
        end if;
    end if;
end process;

end architecture;
```

Figure 16 - Processes Of Program Memory : It is the process part in the design of the ROM with VHDL.

Each time the program enters the memory address, it first checks it. If the address information is between x"00" and x"7F", the ability to read data from the program memory is activated. This enable signal, which is the enable signal, is checked at each clock pulse. If the enable signal is '1', the program performs the data assignment corresponding to the address value entered in memory data_out.

4.1.2 - Data Memory Design

Figure 17 - Data Memory Module Schematic : The data memory block is schematized.

Data memory is a RAM type memory structure with a size of 96 bytes where the data generated while the program is running is recorded and read.

```
entity data_memory is
port(
    clk : in std_logic;
    address :  in std_logic_vector(7 downto 0);
    data_in :  in std_logic_vector(7 downto 0);
    write_en : in std_logic; -- sended signal to write from cpu
    -- Output :
    data_out : out std_logic_vector(7 downto 0)
);
end data_memory;
```

Figure 18 - Inputs & Output Of Data Memory In VHDL : It is screenshot image of entity part of the data_memory unit.

Since data can be written into the data memory, there is a data input. Both data can be written into RAM and data can be read from RAM. According to the information coming from the processor, the data is transferred to the multiplexer as output.

4.1.2.1 - RAM Type

Since it is designed as data memory RAM type, data can be written and read in it. When the system is shut down or restarted, the data in RAM is deleted.

```
architecture arch of data_memory is

    type ram_type is array (128 to 223) of std_logic_vector(7 downto 0); -- 96x8 bit
    signal RAM : ram_type := (others => x"00"); -- initial values are zeros
```

Figure 19 - RAM Type : It is a screenshot of the VHDL code that defines the address range and initial value of the RAM unit.

4.1.2.2 - Processes Of Data Memory

When the address information is received, if the address range is between x"80" and x"DF", the enable signal is activated. If the enable pin, which is the address verification signal/flag, and the write_en signal, which controls the writing/reading of data from the CPU, is '1' in each clock pulse, the incoming data is written into the RAM. If the enable signal confirming the address range is '1' and the write_en signal is '0', it is understood that data will be read from the RAM and the RAM outputs data.

```
    signal enable : std_logic;

begin

    process(address)
    begin
        if(address >= x"80" and address <= x"DF") then -- addresses are between 128 and 223
            enable <= '1';
        else
            enable <= '0';
        end if;
    end process;

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(enable = '1' and write_en = '1') then -- Writing
                RAM(to_integer(unsigned(address))) <= data_in;
            elsif(enable='1' and write_en = '0') then
                data_out <= RAM(to_integer(unsigned(address))); -- Reading
            end if;
        end if;
    end process;
```

Figure 20 - Processes Of Data Memory : It is the process block of the data memory.

4.1.3 - Output Ports Design

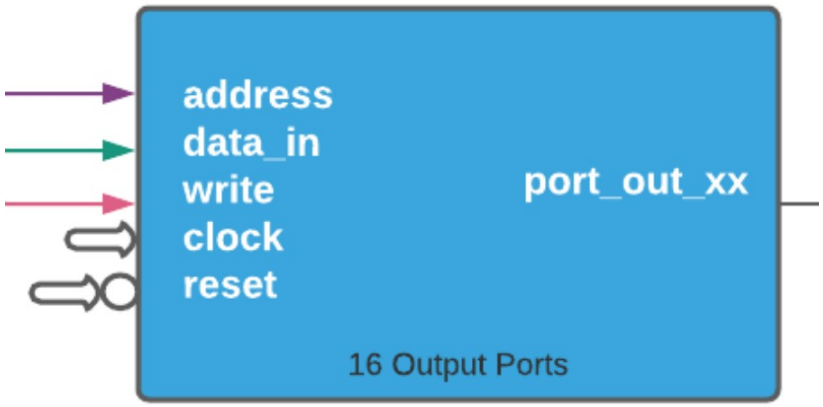

Figure 21 - Output Ports Module Schematic : The output ports block is schematized.

Output ports are the structure designed for the computer to communicate with the outside world. There are 16 outputs, each output being 8 bits.

```
entity output_ports is
    port(
        clk          : in std_logic;
        rst          : in std_logic;
        write_en     : in std_logic;
        address      : in std_logic_vector(7 downto 0);
        data_in      : in std_logic_vector(7 downto 0);
        -- Output
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end output_ports;
```

Figure 22 - Inputs & Outputs Of Output Ports Module In VHDL : The figure is the entity of output ports unit.

4.1.3.1 - Processes Of Output Ports Module

A reset input is defined for the output ports to be 0. In each rising edge clock cycle, if the write_en signal from the CPU to the output ports is active, the data coming to the entered address is written.

```
architecture arch of output_ports is
begin

    process(clk,rst)
    begin
        if(rst = '1') then
            port_out_00  <= (others => '0');
            port_out_01  <= (others => '0');
            port_out_02  <= (others => '0');
            port_out_03  <= (others => '0');
            port_out_04  <= (others => '0');
            port_out_05  <= (others => '0');
            port_out_06  <= (others => '0');
            port_out_07  <= (others => '0');
            port_out_08  <= (others => '0');
            port_out_09  <= (others => '0');
            port_out_10  <= (others => '0');
            port_out_11  <= (others => '0');
            port_out_12  <= (others => '0');
            port_out_13  <= (others => '0');
            port_out_14  <= (others => '0');
            port_out_15  <= (others => '0');
        elsif(rising_edge(clk)) then
            if(write_en = '1') then
                case address is
                    when x"E0" =>
                        port_out_00 <= data_in;
                    when x"E1" =>
                        port_out_01 <= data_in;
                    when x"E2" =>
                        port_out_02 <= data_in;
```

```
    when x"EF" =>
        port_out_15 <= data_in;
    when others =>
        port_out_00  <= (others => '0');
        port_out_01  <= (others => '0');
        port_out_02  <= (others => '0');
        port_out_03  <= (others => '0');
        port_out_04  <= (others => '0');
        port_out_05  <= (others => '0');
        port_out_06  <= (others => '0');
        port_out_07  <= (others => '0');
        port_out_08  <= (others => '0');
        port_out_09  <= (others => '0');
        port_out_10  <= (others => '0');
        port_out_11  <= (others => '0');
        port_out_12  <= (others => '0');
        port_out_13  <= (others => '0');
        port_out_14  <= (others => '0');
        port_out_15  <= (others => '0');
end case;
```

Figure 23 - Processes Of Output Ports Module : The figure is the process block of output ports unit in VHDL.

4.1.4 - Memory Top Level Design

A hierarchical memory block is created by combining the three modules I mentioned above. Since all units in the memory block are a sub-block of the memory, they are defined as components and port maps are created. Define a mux inside the memory block so that the data can be extracted systematically. See Figure 11 for a better understanding of this design. See the memory.vhd file for a more detailed look at the design.

```
entity memory is
port(
    clk         : in std_logic;
    rst         : in std_logic;
    address     : in std_logic_vector(7 downto 0);
    data_in     : in std_logic_vector(7 downto 0);
    write_en    : in std_logic; -- sended signal to write from cpu
    --
    port_in_00  : in std_logic_vector(7 downto 0);
    port_in_01  : in std_logic_vector(7 downto 0);
    port_in_02  : in std_logic_vector(7 downto 0);
    port_in_03  : in std_logic_vector(7 downto 0);
```

Figure 24 - Inputs & Outputs Of Memory Module : It is the entity block in the VHDL of the memory module.

4.1.4.1 - Processes Of Memory

When the address from any CPU, input data from the ports or ram/rom output is changed, data_out is set according to the entered address.

```
process(address , rom_out , ram_out, port_in_00 ,
port_in_01 , port_in_02 , port_in_03 , port_in_04 ,
port_in_05 , port_in_06 , port_in_07 , port_in_08 ,
port_in_09 , port_in_10 , port_in_11 , port_in_12 ,
port_in_13 , port_in_14 , port_in_15)
begin
    if(address >= x"00" and address <= x"7F") then
        data_out <= rom_out;
    elsif(address >= x"00" and address <= x"DF") then
        data_out <= ram_out;

    -- Input Routing
    elsif(address = x"F0") then
        data_out <= port_in_00;
    elsif(address = x"F1") then
        data_out <= port_in_01;
    elsif(address = x"F2") then
        data_out <= port_in_02;
    elsif(address = x"F3") then
        data_out <= port_in_03;
```

Figure 25 - Processes Of Memory : It is the process block of the memory unit.

4.2 - CPU Design

The CPU contains two main modules. These are control unit and datapath. The control unit can be defined as the brain of the computer. It organizes all the working steps of the processor here. There is a finite state machine inside. This FSM controls the retrieval, decoding, fetch, and execution of instructions. The control unit communicates with the memories via datapath.

Registers in Datapath (IR,MAR,PC , A,B,CCR), BUS structures, muxes and ALUs. The register where the instruction is kept after it is read from memory is the instruction register (IR). The control unit sets its operations according to the value in the IR. The memory access register (MAR) is the register that transmits the address information necessary to read/write the memory to the memory. The program counter keeps the address information indicating which instruction to read from the memory. It is '0' when the program starts. Its value increases by one after each command (except branch commands). A and B registers are general purpose data registers where operands are held. Condition code register (CCR) is the register that sets N,Z,V,C flags and reports status as a result of ALU operations. To understand the functions of registers in datapath in detail, see the "2.1.2.1 - Registers" section.

I can define it as the wire that connects the BUS1 and BUS2 registers. Each register has to communicate and exchange data with each other. If the connection between these registers is connected to each other through ports, the design area becomes too large and too many interconnections are used inside. Therefore, BUS1 and BUS2 structures were designed by simplifying the traffic in the datapath. PC, A and B registers are connected to BUS1, and BUS1, ALU and data that comes from memory are connected to BUS2. Which BUS will be selected is adjusted by the selection signal from the control unit. Data from BUS1 can be directly transferred to memory or sent to BUS2. The sent data can also be transferred to registers from BUS2. In fact, data from CPU to memory goes directly from BUS1. Which data will be sent to the BUS is also determined by the control unit.

BUS1 is responsible for transmitting the data in the CPU register to the memory. BUS2 is basically responsible for transmitting the data read from the memory to the CPU.

In addition, even if the address information in the MAR is sent to the memory and the data in the BUS1 is sent to the memory, the write signal in the control unit is set to perform these operations.

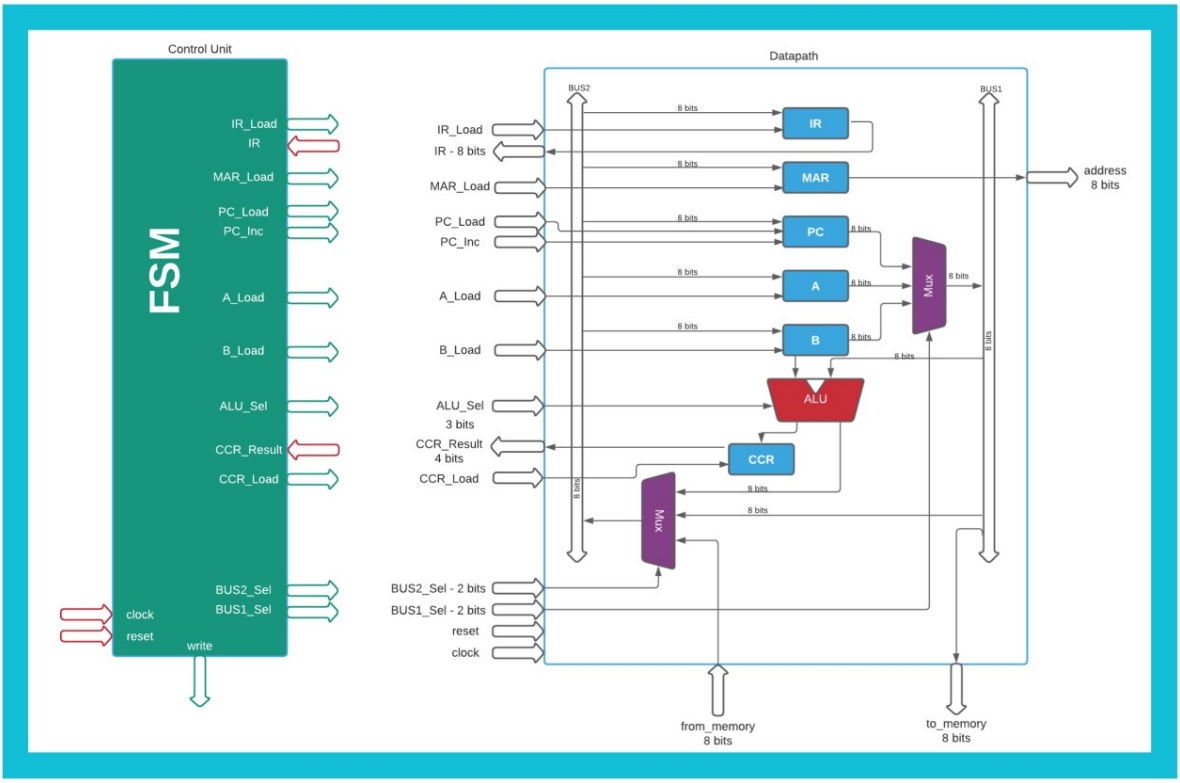

Figure 26 - CPU Module Schematic : All submodules and systems in the CPU module are schematized.

You can see the designs of the internal structure of the CPU module above. You can also examine the Control Unit and Datapath inside the CPU.

4.2.1 - Datapath Design

Datapath generally provides the communication between the Control Unit and the Memory Unit with the commands it receives from the Control Unit and the processing of the instructions.

Datapath includes IR, MAR, PC, general purpose registers A, B registers, CCR, Arithmatic Logic Unit (ALU), muxes and two bus structures. I explained what these units do in general and their working principles. In this section, I will try to explain based on design. In this way, I hope that Datapath will be better understood.

As can be seen on the schematic, 6 register load inputs (A_Load , IR_Load etc.), PC_Inc input that increases the program counter connected from the Control Unit, 3- bit ALU_Sel input that shows what operation will be done in the ALU, There are 2 2-bit bus selection input signals that inform which data to drive. Since the datapath is in a sequential structure, it is necessary to use the clock input. Data from Memory does not go to Control Unit, it only connect to Datapath. The data coming from the memory unit is processed in this module.

Address information is exported from the MAR in the Datapath (to the Memory Unit). Data information is transferred from BUS1 to Memory Unit. Data transfer from Datapath to Control Unit takes place with only two output signals. These output signals are the output signal that transfers the data in the IR and the output signal that carries the status information transferred to the CCR to the Control Unit as a result of the operation performed in the ALU.

```
entity data_path is
    port(
        clk         : in std_logic;
        rst         : in std_logic;
        IR_Load     : in std_logic; -- Instruction Register Load
        MAR_Load    : in std_logic; -- Memory Access Register Load
        PC_Load     : in std_logic; -- Program Counter Register Load
        PC_Inc      : in std_logic; -- Program Counter Register Incrementer
        A_Load      : in std_logic;
        B_Load      : in std_logic;
        ALU_Sel     : in std_logic_vector(2 downto 0);
        CCR_Load    : in std_logic; -- Condition code register
        BUS1_Sel    : in std_logic_vector(1 downto 0);
        BUS2_Sel    : in std_logic_vector(1 downto 0);
        from_memory : in std_logic_vector(7 downto 0);

        -- Outputs :
        IR          : out std_logic_vector(7 downto 0);
        address     : out std_logic_vector(7 downto 0); -- address to memory
        CCR_Result  : out std_logic_vector(3 downto 0); -- NZVC
        to_memory   : out std_logic_vector(7 downto 0)  -- data to memery

    );
end data_path;
```

Figure 27 - Inputs&Outputs Of Datapath Module In VHDL : The figure is screenshot image of entity of the Datapath

4.2.1.1 - Architecture Of Datapath

In Datapath, the ALU module is defined as a component. Afterwards, all Datapath internal signals were defined. See figure 26 to review these internal signals. The muxes that allow determining the values to be assigned to BUS1 and BUS2 are designed as combinational circuits. Data is transferred to buses according to bus selection inputs.

```
-- BUS1 Mux :
BUS1 <= PC          when BUS1_Sel <= "00" else
        A_reg       when BUS1_Sel <= "01" else
        B_reg       when BUS1_Sel <= "10" else (others => '0');
-- BUS2 Mux :
BUS2 <= ALU_result  when BUS2_Sel <= "00" else
        BUS1        when BUS2_Sel <= "01" else
        from_memory when BUS2_Sel <= "10" else (others => '0');
```

Figure 28 - BUS Multiplexers : Multiplexers were designed with when-else structures in VHDL.

All of the registers in the datapath are defined in a separate process block. The working logic of registers is the same. If the load input from the control unit is '1' at the rising edge of each clock pulse, data is transferred into the register from BUS2. To BUS2, data comes from memory, ALU or BUS1. Control Unit selects the data coming to BUS2. If the reset signal is '1', the inside of the registers is set to '0'. Here, as an extra operation, when PC_Inc is '1' as an extra in the PC register, the value in the register is increased by one.

```
-- Program Counter Register
process(clk,rst)
begin
    if(rst = '1') then
        PC <= (others => '0');
    elsif(rising_edge(clk)) then
        if(PC_Load = '1') then
            PC <= BUS2;
        elsif(PC_Inc = '1') then
            PC <= PC + X"01";
        end if;
    end if;
end process;
```

Figure 29 - A Process Of Register Design : A screenshot of a register (PC) process block in VHDL is shown. You can check datapath.vhd for the design of other registers.

4.2.1.1.1 - ALU (Arithmetic Logic Unit)

It is the unit designed in accordance with the hierarchical design to be used in Datapath to perform logical and arithmatic operations.B register and BUS1 are connected to the inputs of the ALU unit.

```
entity ALU is
    port(
        A           : in std_logic_vector(7 downto 0); -- 8 bit data to be processed
        B           : in std_logic_vector(7 downto 0); -- 8 bit data to be processed
        ALU_Sel     : in std_logic_vector(2 downto 0);

        -- Outputs
        NZVC        : out std_logic_vector(3 downto 0);
        ALU_RESULT  : out std_logic_vector(7 downto 0)
    );
end ALU;
```

Figure 30 - Input&Outputs Of ALU : It is the entity block of ALU in VHDL.

The ALU selection input is a 3-bit input signal from the Control Unit that shows what operation will be performed in the ALU. For example, if ALU_Sel is "001", the data in BUS1 is subtracted from B_Reg (A input) data, or if ALU_Sel is "101", 1 is subtracted from A input.

```
process(ALU_Sel,A,B) begin
    sum_unsigned <= (others => '0'); -- reset parameter
    case ALU_Sel is
        when "000" => -- Addition
            alu_signal      <= A+B;
            sum_unsigned    <= ('0' & A) + ('0' & B); --
        when "001" => -- Subtraction
            alu_signal      <= A-B;
            sum_unsigned    <= ('0' & A) - ('0' & B); --
```

Figure 31 - Process Block In ALU : The figure is the screenshot of process block of ALU.

The ALU unit has two outputs. One of them is the ALU_Result, which is the result of the operation, and the other is the 4-bit NZVC output, which reports the states that occur as the result of the operation. Bit 0 of the NZVC is set to the carry bit, bit 1 is the overflow bit, bit 2 is the zero bit, and bit 3 is the negative bit.

```
-- NZVC

NZVC(3) <= alu_signal(7); -- N
NZVC(2) <= '1' when alu_signal = x"00" else '0'; -- Z

-- V :
add_overflow <= (not(A(7)) and not(B(7)) and alu_signal(7)) or (A(7) and B(7) and not(alu_signal(7)));
sub_overflow <= (not(A(7)) and B(7) and alu_signal(7)) or (A(7) and not(B(7)) and not(alu_signal(7)));

NZVC(1)  <= add_overflow    when (ALU_Sel = "000") else
            sub_overflow    when (ALU_Sel = "001") else '0';

NZVC(0)  <= sum_unsigned(8) when (ALU_Sel = "000") else
            sum_unsigned(8) when (ALU_Sel = "001") else '0';
```

Figure 32 – NZVC : Shows how the negative , zero , overflow and cary bit states are set in VHDL codes.

For the carry bit, the 8th bit of the sum_unsigned signal, which is a 9-bit signal defined before, is checked. To find the overflow, the logical operations you see above are performed on the last bits of the inputs and the result. If the result is zero then Z (bit 2 of NZVC) is '1'. In order to understand whether the result of the operation is negative or positive, the last bit of the result of the operation is checked and assigned to the 3rd bit of NZVC.

ALU's port map was created in Datapath's architecture. In this port map, NZVC is directly connected to the CCR. ALU_result, which is the result of ALU operation, is directly connected to the multiplexer of BUS2.

4.2.2 - Control Unit Design

There is a finite state machine in the Control Unit that executes the Fetch-Decode-Execute operations. This part does all the management of the processor.

First, let's talk about the structure of this state machine. There are two status signals in the Finite state machine (FSM) that report the previous and next status.

The current state circuit is sequential and sensitive to clock and reset. The next state circuit is the combinational circuit, which depends on the inputs and the current state.

Two status signals and three separate process blocks are used in FSM. The reason for setting up such a system is that many control signals that the Control Unit sends to the Datapath and memory must be asynchronous. In other words, signals must be transmitted without waiting for the clock pulse. If we direct the system with a clock pulse, it will not be possible to streamline the operations and pull the correct data at the time intervals we expect, effectively.For this reason, it is more suitable for this system to establish a structure in which the necessary signals are assigned to the output logic side asynchronously, but the state is updated regularly according to the clock pulses.

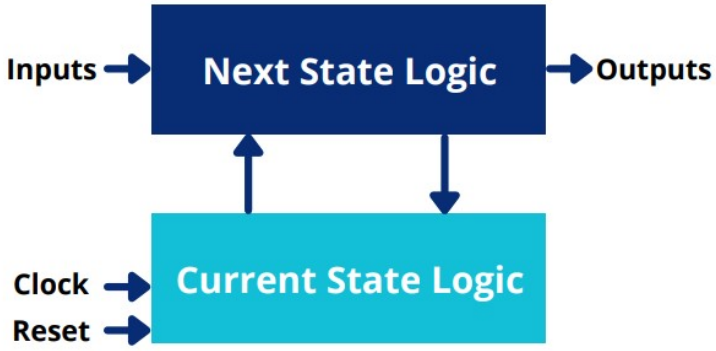

Figure 33 - FSM Schematic : It is the schematized structure of the FSM in the Control Unit.

```
entity control_unit is
 port(
       clk             :       in std_logic;
       rst             :       in std_logic;
       CCR_Result      :       in std_logic_vector(3 downto 0);
       IR              :       in std_logic_vector(7 downto 0);
       -- Outputs :
       IR_Load         :       out std_logic; -- Instruction Register Load
       MAR_Load        :       out std_logic; -- Memory Access Register Load
       PC_Load         :       out std_logic; -- Program Counter Register Load
       PC_Inc          :       out std_logic;  -- Program Counter Register Incrementer
       A_Load          :       out std_logic;
       B_Load          :       out std_logic;
       ALU_Sel         :       out std_logic_vector(2 downto 0);
       CCR_Load        :       out std_logic; -- Condition code register
       BUS1_Sel        :       out std_logic_vector(1 downto 0);
       BUS2_Sel        :       out std_logic_vector(1 downto 0);
       write_en        :       out std_logic
   );
end control_unit;
```

Figure 34 - Inputs&Outputs Of Control Unit In VHDL : It is the entity block of the control unit in VHDL.

All of the states are defined as state_type in Control Unit's architecture. State values are defined as current_state and next_state signals of state_type type.

```
type state_type is  (
                    STATE_FETCH_0 , STATE_FETCH_1 , STATE_FETCH_2 , STATE_DECODE_3 ,
                    STATE_LDA_IMM_4 , STATE_LDA_IMM_5 , STATE_LDA_IMM_6 , -- YUKLE A SABIT
                    STATE_LDA_DIR_4 , STATE_LDA_DIR_5 , STATE_LDA_DIR_6 , STATE_LDA_DIR_7 , STATE_LDA_DIR_8, -- YUKLE A DIRECT
```

Figure 35 - State Types : All states were defined for fetch-decode-execute process.

All instruction values are also defined in Control Unit's architecture as constant. This definition is important for the decode and execute steps as mentioned before.

```
-- Loads and Stores Commands
constant YUKLE_A_SBT        :std_logic_vector(7 downto 0) := x"86";
constant YUKLE_A            :std_logic_vector(7 downto 0) := x"87";
constant YUKLE_B_SBT        :std_logic_vector(7 downto 0) := x"88";
constant YUKLE_B            :std_logic_vector(7 downto 0) := x"89";
constant KAYDET_A           :std_logic_vector(7 downto 0) := x"96";
constant KAYDET_B           :std_logic_vector(7 downto 0) := x"97";

-- Data Manipulations
constant TOPLA_AB           :std_logic_vector(7 downto 0) :=x"42";
```

Figure 36 - Definiton The Instructions As Constant In Control Unit : The figure is the screenshot of definitions of the instructions in VHDL.

4.2.2.1 - Current State Logic Circuit Design

It is the structure designed to determine the current state. At each rising edge clock pulse, the next_state value is transferred to the current state.

```
-- Current Logic State
process(clk,rst)
begin
    if(rst = '1') then
        current_state <= STATE_FETCH_0;
    elsif(rising_edge(clk)) then
        current_state <= next_state;
    end if;
end process;
```

Figure 37 - Current State Process : The current state is set within this process block.

4.2.2.2 - Next State Logic Circuit Design

The next state logic is also implemented as a separate process. The next state logic depends on the current state, instruction register (IR), and the condition code register (CCR_Result).

```
-- Next State Logic
process(current_state , IR , CCR_Result)
begin
    case current_state is
        when STATE_FETCH_0  =>
            next_state <= STATE_FETCH_1;
        when STATE_FETCH_1  =>
            next_state <= STATE_FETCH_2;
        when STATE_FETCH_2  =>
            next_state <= STATE_DECODE_3;
        when STATE_DECODE_3  =>
            if(IR = YUKLE_A_SBT) then
                next_state <= STATE_LDA_IMM_4;
            elsif(IR = YUKLE_A) then
                next_state <= STATE_LDA_DIR_4;
            elsif(IR = YUKLE_B_SBT) then
                next_state <= STATE_LDB_IMM_4;
            elsif(IR = YUKLE_B) then
                next_state <= STATE_LDB_DIR_4;
            elsif(IR = KAYDET_A) then
                next_state <= STATE_STA_DIR_4;
            elsif(IR = KAYDET_B) then
                next_state <= STATE_STB_DIR_4;
            elsif(IR = TOPLA_AB) then
                next_state <= STATE_ADD_AB_4;
            elsif(IR = ATLA) then
                next_state <= STATE_BRA_4;
            elsif(IR = ATLA_ESITSE_SIFIR) then
                if(CCR_Result(2) = '1') then    --NZVC , Zero second bit
                    next_state <= STATE_BEQ_4;
                else    -- Z = '0'
                    next_state <= STATE_BEQ_7;
                end if;
            else
                next_state <= STATE_FETCH_0;
            end if;
-------------------------------------------------------
        when STATE_LDA_IMM_4 =>
            next_state <= STATE_LDA_IMM_5;
        when STATE_LDA_IMM_5 =>
            next_state <= STATE_LDA_IMM_6;
        when STATE_LDA_IMM_6 =>
            next_state <= STATE_FETCH_0;
-------------------------------------------------------

        when STATE_LDA_DIR_4 =>
            next_state <= STATE_LDA_DIR_5;
```

Figure 38 - Next State Assignment : Next state is set in this process block.

4.2.2.3 - FSM States

To understand the working principles of Control Unit, we need to know the Fetch-Decode and Execute steps.

- **Fetch :** It is the process of reading data from Program Memory according to the value of the PC and transferring this data to the IR in the CPU .
- **Decode :** It is the process of detecting this instruction transferred to IR.
- **Execute :** This is the step where this instruction, which has been identified, is applied.

Although the operation of these three steps is clear, more than one clock pulse is required to complete these steps. Because the process of reading from memory is performed. To read from memory, we must transmit the address signal and wait for the next clock pulse. When reading from synchronous memory structures, a certain period of time must be waited. When you examine the following schematic created for the fetch decode execute steps, the processes here will be better understood.

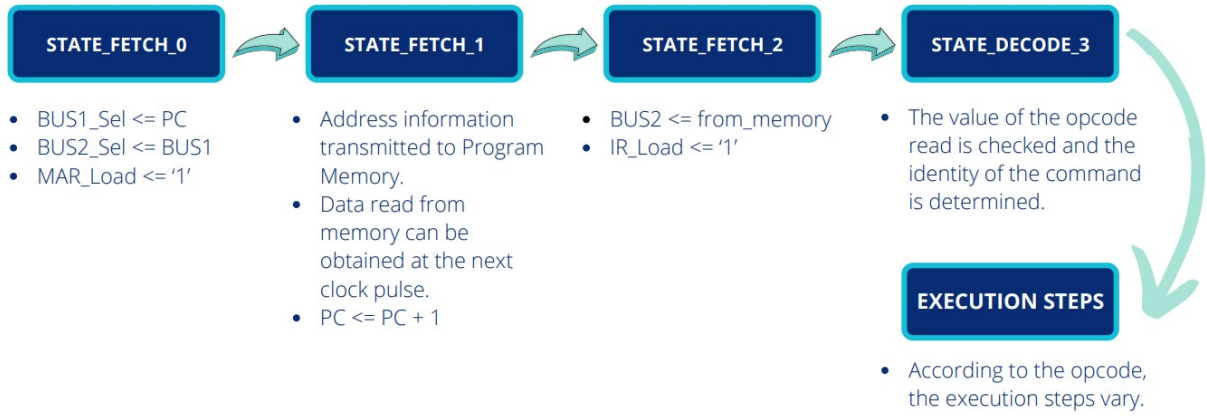

Figure 39 - Fetch - Decode Steps : It is the schematized structure of the Fetch and Decode steps.

In STATE_FETCH_0 state, the value inside the PC is transferred to BUS1. Then the value of BUS1 is driven to BUS2. That is, the PC value is indirectly transferred to BUS2. In the case of STATE_FETCH_1 it is necessary to wait for a clock cycle as the BUS2 value, that is, the address of the instruction to be read, is transmitted to the memory. While we waited, we increased the value of the PC by one. Afterwards, the data (instruction opcode) read from the program memory is read in STATE_FETCH_2 state. IR_Load is set to '1' to transfer this read opcode to IR.

In STATE_DECODE_3 state, this opcode is read and what this opcode is what is determined. All of these states are common to all instructions. What differentiates instructions is the state of execution.

4.2.2.4 - Execution Of Instructions

I explained in the previous thread that the fetch and decode steps are almost the same for all instructions. In this section, I will explain how commands complete the execute step. An important point in the execution step is the recognition of the operand here. Now let's explain the execution steps for instructions :

- **Execution Of Load Immediate Instructions :** In the first step of the load immediate command, PC value is transferred from BUS1 to BUS2 and MAR_Load is activated. In the next stage, data is expected to come from the memory and the PC is increased by one. Finally, the data read from memory is assigned to BUS2 and transferred from BUS2 to register A.

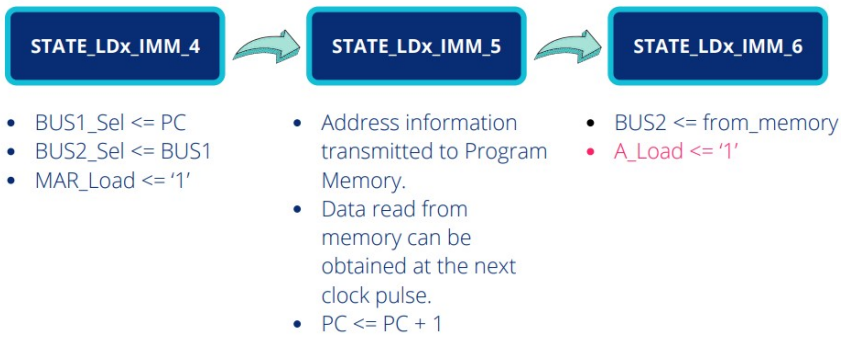

Figure 40 - Load Immediate Instruction Execution Steps : This figure is a schematized structure of what happens in the execution steps of load immediate instructions.

- **Execution Of Load Direct Instructions :** In the next state, data is transferred from memory to BUS2 and MAR_Load becomes '1' as data will be transferred from memory to register. In the next state, a clock pulse is waited for data to come from memory. Finally, the data coming to BUS2 is transferred and transferred from BUS2 to register A.

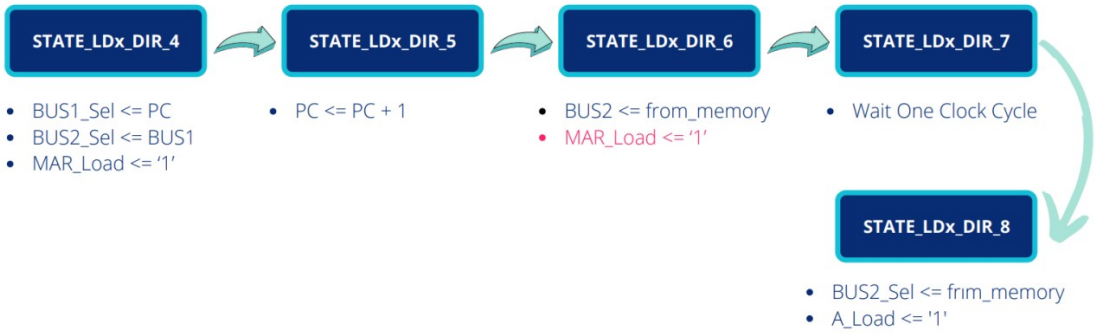

Figure 41 - Load Direct Instruction Execution Steps : This figure is a schematized structure of what happens in the execution steps of load direct instructions.

- **Execution Of Store Instructions :** In the first step, PC value is driven from BUS1 to BUS2 and assigned to MAR. In the second step, a clock cycle is expected and the PC value is increased by +1. In the next step, the operand is assigned to BUS2 and this operand is transferred to MAR. Finally, the data in A_reg or B_reg is transferred to BUS1 to BUS1 and transferred to memory (RAM) from here.

Figure 42 - Store Instruction Execution Steps : This figure is a schematized structure of what happens in the execution steps of store instructions.

- **Execution Of Data Manipulation Instructions :** There is only one step in the execution of data manipulation instructions. In this step, A_reg value is transferred to BUS1, the operation to be performed on the ALU is assigned, the result of the operation from the ALU is transferred to BUS2, this result is loaded from BUS2 to the A register and the NZVC value from the ALU is transferred to the CCR.

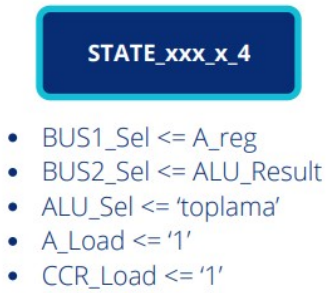

Figure 43 - Data Manipulation Instruction Execution Step : This figure is a schematized structure of what happens in the execution step of data manipulation instructions.

- **Execution Of Branch Instructions :** Branch commands are commands that direct the code according to the situations that occur. In the first execution step after the fetch and decode steps, the PC value is assigned to BUS1. It is also assigned from BUS1 to BUS2 and MAR_Load is set to '1'. In the second step, the data is expected to come from the memory. In the third step, the data is read from the memory and the data read from the BUS2 is transferred to the PC.

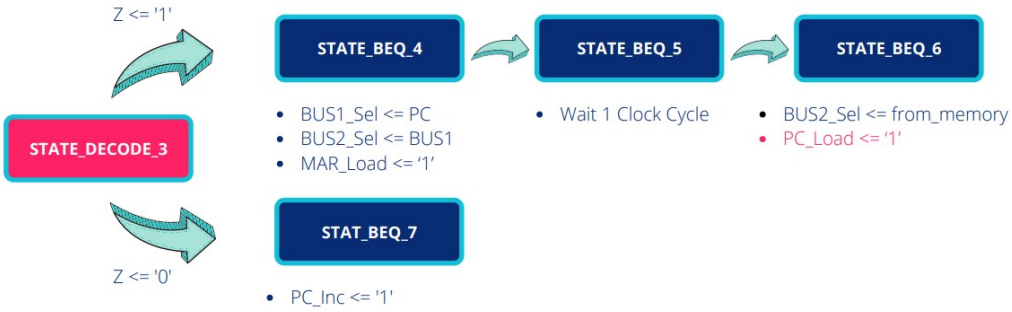

Figure 44 - Branch (i.e. BEQ) Instruction Execution Steps : This figure is a schematized structure of what happens in the execution steps of branch instructions.

Below, you can see the sample code snippet on the processing of current states after the fetch and decode steps in the third process block, which is designed as an output logic circuit in the Control Unit.

```
when STATE_BEQ_4 =>
    BUS1_Sel <= "00";
    BUS2_Sel <= "01";
    MAR_Load <= '1' ;
when STATE_BEQ_5 =>
    -- BOS
when STATE_BEQ_6 =>
    BUS2_Sel <= "10";
    PC_Load  <= '1';
when STATE_BEQ_7 =>
    PC_Inc <= '1';
```

Figure 45 - Sample Code Snippet Of BEQ Execution : A screenshot of the code in VHDL of one of the execution steps.

You can see more execution steps by looking at the code of Control Unit (control_unit.vhd).

Before each fetch-decode-execution operation, all load , enable and selection signals are reset in the Control Unit. In this way, there is no confusion in each new command operation.

5 - Simulation And Tests

In this section, we will examine the behavioral simulation by trying the codes prepared for test purposes before the program memory (ROM). These tests show us that the designed computer works correctly.

In the prepared testbench, a clock period is set to 20ns and the computer is reset first and then 40ns is waited.

```
clock_process: process
begin
        clk <= '0';
        wait for clock_period/2;
        clk <= '1';
        wait for clock_period/2;
end process;
```

```
process
begin
rst <= '1';
wait for 40ns;
rst <= '0';
wait for clock_period*2;

wait for clock_period*200;
wait;

end process;
```

Figure 46 - Clok And Process Parts Of Testbench : It is a screenshot of the place where the clock configurations prepared in testbench are made for simulations.

Code Test And Simulation

```
0 => YUKLE_B_SBT,
1 => x"0A",
2 => KAYDET_B,
3 => x"80",
4 => ATLA,
5 => x"00",
others => x"00"
```

The code you have seen above first saves the number 0x0A in register B. Later, it saves this data to the RAM block at address 0x80 (128 in decimal format and first 8 bits cell of RAM). Finally, the code returns to the beginning with the branch command.

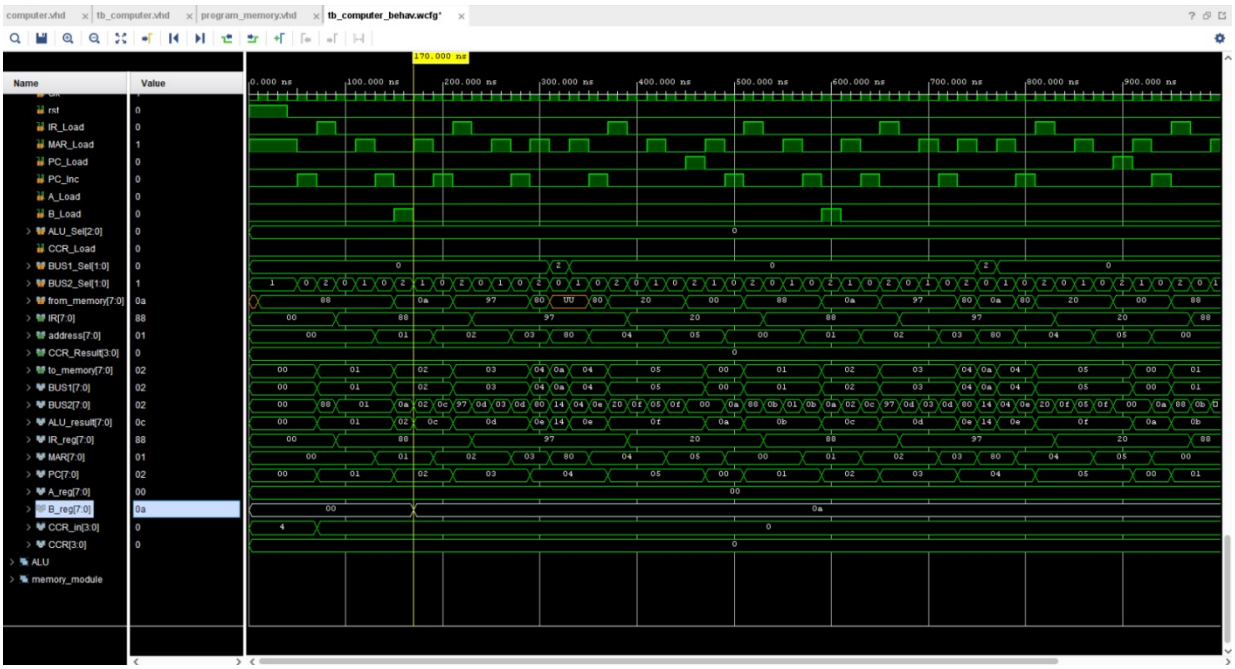

Figure 47 - Result Of Simulation That Shows Loading Data To Register : A screenshot of the simulation result to show one of the general purpose registers that data is being loaded.

In the image you have seen above, you can see that the value 0x0A is assigned to register B after the last execution step of the LDB state.

When the value in the Program Counter Register is 0, the value of the "YUKLE_B_SBT" opcode has been sent to the Instruction Register. After one clock cycle, the opcode value sent appears in the instruction register. Currently this value is 0x88. With this process, you can see that the value of the Program Counter Register has increased by one. The current value of the Program Counter is "01". In this way, the operand value that will come after the opcode will be read. The reading process of the operand varies according to the opcode. Since the opcode used here is included in a load instruction group, the control unit expects an operand in the execute step after this opcode. Since the value of the Program Counter Register is "01", the code written here has the value x"0A". The opcode written earlier is responsible for loading into the B register. As a result, the written code will save the value 0x"0A" in register B, which is a general purpose register type. As a result of the simulation, you can see that the desired value is loaded into the B register by looking at the B_reg value marked with white.

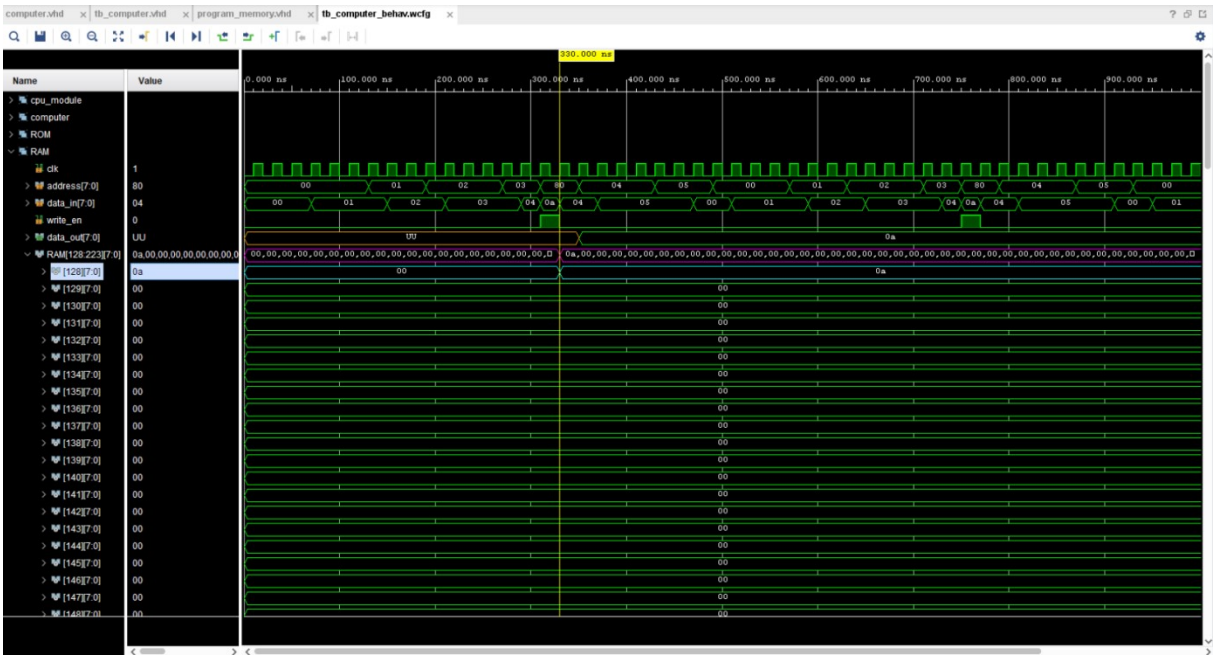

Figure 48 - Result Of Simulation That Shows Storing Data To Register : It is a screenshot of the simulation result to show that the data is stored in the memory. After the data input signal became 0x0A, 0x0A data was recorded at the 128th address in the memory map.

This computer system was able to load values from general purpose registers to B register in previous values of Program Counter Register. Once the value of the Program Counter Register is increased, it will start processing other instructions. The value of the Program Counter register is now "02". The opcode written here is the "KAYDET_B" opcode, which saves the value of register B to the entered address (in memory). The value of this opcode is x"97". After Fetch and Decode operations, Control Unit understands this value and then an oprand value is expected.

Afterwards, the Program Counter Register value is increased by one. The Program Counter Register value is now "03". The expected operand of the opcode that performed the storing operation written earlier, x"80", is read. Address x"80" is a free space in RAM. According to the written code, the value in the B register, which is one of the general purpose registers, is written to the x"80" address of the RAM unit in the computer.

When we examine the values at 330ns in the simulation image above, you can see that the address information transmitted to the RAM is x"80". At the same time, the write_en flag is active, which allows data to be written into RAM. When this activation flag is 1, the value of the data (data_in) entering the RAM is x"0a". Finally, x"0a" data is written to the x"80" (127 in decimal) address of the RAM. As a result of the simulation, this value written into the RAM is marked in blue.

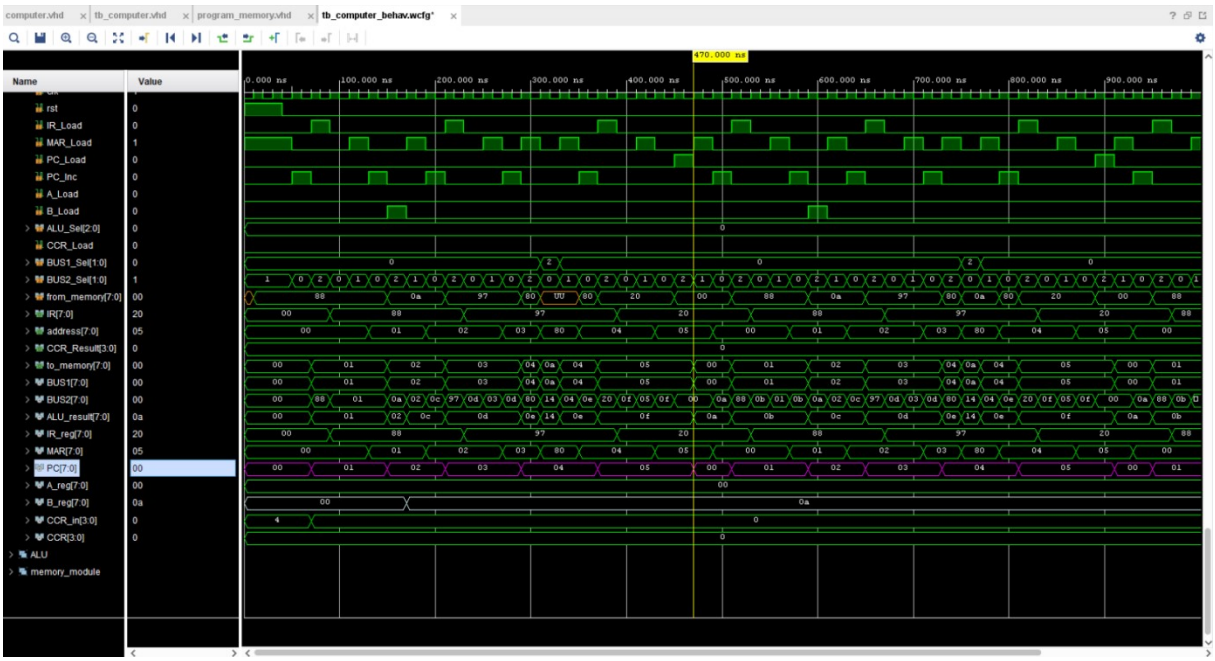

Figure 49 - Result Of Simulation That Shows Restart The Code : A screenshot of the simulation result to show that the code loop is in progress. You can understand that the code has been restarted by making the PC 0x00 with the branch command by looking at the PC values marked purple as a result of the simulation above.

According to the written code, the value in register B was saved in the last RAM. The Program Counter Register value is incremented by one after all operations have been successfully performed. The current Program Counter Register value is "04". After this step, the value in the Instruction register becomes x"20" as seen in the simulation. Because when the program is in Counter Register "04", "ATLA" opcode from branch commands is read. The value of this opcode in ISA is x"20". This opcode performs the process from which address the program will continue according to the operand value to be read after it. After this value is read and the Program Counter Register value increases by one, "x00" value is read in the program. After reading this value, as it can be seen in the simulation, the value of the Program Counter Register turned back to 0 right after it was 5. In this way, the program has become an endless loop that starts again.

6 - Conclusion

In this project, I designed an 8-bit computer architecture with RISC instruction set architecture and based on Harvard Architecture.

Computers are structures that process code sets consisting of instructions in computers in the CPU hardware and save/read the data processed or entered in the memory part. RISC is a processor architecture in which the bit positions of the opcode and operand in a simple structure are the same and fixed for each instruction. Instructions consist of opcode and operand. It has been designed to switch to another opcode or operand at every PC change.

There are many modules and various units in this designed computer. A finite state machine has been designed that manages the entire computure in the Control Unit inside the CPU. The processing of the instructions and the control of the registers and units in the Datapath according to the situations are done here.

Another important unit in the CPU is the Datapath, which regulates the communication between the Control Unit and the Memory Unit and enables various operations to be performed with the inputs. Datapath includes Instruction Register, Program Counter Register, General Purpose Register, Memory Access Register, Condition Control Register and Arithmetic Logic Unit, which are critical for computer management and processing of written codes. Each of these components communicate with each other through the bus systems structure.

The Memory Unit, which is in constant communication with the CPU in the designed computer, provides the recording / reading of data and communication with the outside world. Memory Unit is a hierarchical unit structure consisting of the combination of three separate modules. It is designed to follow each other in order to manage memory addresses easily and properly. All memory units are assigned a unique address. Program Memory Unit is a ROM type unit in which the main codes are written in which the instructions are stored and cannot be changed after writing. The data memory unit is the structure in which the data is saved and read when requested.

The Data Memory Unit is of RAM type and as the opposite of ROM, the data in it can be deleted when the hardware is reset. In the Memory unit, the other unit is Output Ports. Input Ports are transferred from the memory unit to the CPU side.

The capabilities of this designed computer are directly proportional to the instructions in it. The computer is generally designed to be able to implement the codes written in it. It can run algorithms that can perform operations such as addition-subtraction-logical operations with inputs or variables, evaluate the situation according to the results and continue the code flow accordingly.

7 – References

[1] https://en.wikipedia.org/wiki/Harvard_architecture

[2] https://www.arm.com/glossary/risc

[3] http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/1-circuits/alu.html

[4] https://www.geeksforgeeks.org/introduction-of-alu-and-data-path/

[5] https://www.fpga4student.com/2017/06/vhdl-code-for-arithmetic-logic-unit-alu.html

[6] https://www.geeksforgeeks.org/different-classes-of-cpu-registers/

[7] https://www.youtube.com/watch?v=cNN_tTXABUA

[8] Ian Grout, in Digital Systems Design with FPGAs and CPLDs, 2008

[9] Elmustafa Sayed Ali Ahmed Mohamed, Principles of Computer System, 2015

[10] Brock J. LaMeres, Introduction to Logic Circuits & Logic Design with VHDL, 2019

[11] Studies in Microprocessor Design Donald Alpcrt Technical Report NO. 232, June 1982

[12] David A. Patterson - John L. Hennessy, Computer Organization and Design: The Hardware/ Software Interface, Fifth Edition, 2014