

# SoftCore CPU

THE IMPEMNTATION OF SOFTCORE CPU AND ITS ASSEMBLER

AYMEN SEKHRI

# Acknowledgements

We would like to thank our supervisor Dr. Metidji for the patient guidance, encouragement and advices he has provided throughout the time he has been our project advisor. We would like also express our gratitude to Dr. Maache and Dr. Benzekri for helping us and giving us a strong knowledge in both computer engineering and digital systems design with VHDL through their courses, which helped us a lot to finish the thesis.

In addition, we would like to thank our Institute for providing us resources and our teachers for being always approachable and helping us on innumerable occasions over the last three years.

# Abstract

This project demonstrates the implementation of a soft-core CPU on an FPGA and the design of its different components using VHDL and block diagram in Quartus II software. This report covers the implementation of different registers, the ALU, the RAM and its different data and address decoders (to perform the one byte addressable memory), different I/O peripherals and finally the control unit. In addition to the hardware implementation of the CPU, an assembler software has been made to make the programming of the CPU easier and allow a serial communication from any PC to the CPU. This report digs through all the steps that have been taken to archive the final version of the project. In the end, the report will cover some future edits that may improve the performance and reliability of the CPU.

# Table of Contents

Acknowledgements .....	0
Abstract.....	2
Table of Contents .....	3
Table of Figures and Tables.....	5
Chapter 1: Introduction.....	7
Chapter 2: Background.....	8
1. The Central Processing Unit and its elements.....	8
2. Main types of computer architecture designs (RISC vs. CISC).....	8
3. Von Neumann Architecture .....	8
4. Programmable Devices .....	10
5. Mixing the high performance of CPUs and flexibility of the FPGAs .....	10
Chapter 3: Design and Implementation.....	12
1. Registers.....	12
2. Arithmetic Logic Unit (ALU) .....	13
3. Random Access Memory (RAM).....	14
3.1. Main RAM Block .....	17
3.2. Address Controller.....	18
3.3. Data Controller .....	20
4. Control Unit.....	21
4.1. Fetch Cycle.....	21
4.2. Execute Cycle.....	22
5. I/O Device's Ports .....	24
5.1. Seven Segment Output.....	24
5.2. LCD Port.....	25
5.3. Pulse Width Modulator (PWM) .....	25
5.4. Timers .....	26
5.5. RS232 Port .....	27
6. Assembler.....	29
6.1. Comments .....	30
6.2. Labels.....	30
6.3. Data Definitions.....	31
Chapter 4: Results and Evaluation .....	32
1. Example 1: Counter .....	32
2. Example 2: Servo Motor Controller .....	33

3. Example 3: LCD Display .....	34
Chapter 5: General Conclusion .....	37
Chapter 6: Appendix .....	38
Table of Instructions .....	38
Register OP Codes .....	39
Fetch Cycle .....	40
Bibliography .....	42

# Table of Figures and Tables

FIGURE 2-1: VON NEUMANN ARCHITECTURE. ....	9
FIGURE 3-1: TOP LEVEL CPU ARCHITECTURE.....	12
FIGURE 3-2: REGISTER PINOUT. ....	13
FIGURE 3-3: ALU PINOUT. ....	13
FIGURE 3-4: FOUR BYTE ADDRESSING MODE.....	15
FIGURE 3-5: ONE BYTE ADDRESSING MODE .....	15
FIGURE 3-6: RAM PINOUT .....	15
FIGURE 3-7: READ OPERATION WAVEFORMS .....	16
FIGURE 3-8: WRITE OPERATION WAVEFORMS.....	17
FIGURE 3-9: INTERNAL DESIGN OF RAM .....	17
FIGURE 3-10: MAIN RAM PINOUT .....	18
FIGURE 3-11: MAIN RAM WAVEFORMS. ....	18
FIGURE 3-12: UNALIGNED MEMORY ACCESS. ....	19
FIGURE 3-13: ADDRESS CONTROLLER PINOUT .....	19
FIGURE 3-14: ADDRESS CONTROLLER WAVEFORMS.....	19
FIGURE 3-15: DATA CONTROLLER PINOUT.....	20
FIGURE 3-16: LITTLE ENDIAN AND BIG ENDIAN FORMATS.....	20
FIGURE 3-17: INDETERMINATION PROBLEM .....	21
FIGURE 3-18: SAFE SYNCHRONIZATION .....	21
FIGURE 3-19: THE CONTENT OF THE BUS DURING FETCH CYCLE OF INSTRUCTION AT ADDRESS 0x00 .....	22
FIGURE 3-20: MICROPROGRAMMABLE CONTROL UNIT DESIGN [8] .....	23
FIGURE 3-21: SIMPLIFIED FSM DESIGN OF 2 INSTRUCTIONS .....	23
FIGURE 3-22: I/O PORTS DESIGN. ....	24
FIGURE 3-23: 4 DIGITS SEVEN SEGMENT SCHEME. ....	25
FIGURE 3-24: SEVENSIG PORT PINOUT.....	25
FIGURE 3-25: LCD PORT PINOUT.....	25
FIGURE 3-26: PWM PINOUT. ....	26
FIGURE 3-27: PWM DUTY CYCLE. ....	26
FIGURE 3-28: TIMERS PINOUT.....	26
FIGURE 3-29:RS232 PHYSICAL PORT PINOUT. [9] .....	27
FIGURE 3-30: RS232 COMMUNICATION. [11] .....	28

FIGURE 3-31: RS232 PROTOCOL DATAFLOW. ....	28
FIGURE 3-32: USB TO RS232 CABLE. ....	29
FIGURE 3-33: EXAMPLE OF PROGRAM THAT PRINTS STRING TO LCD. ....	30
FIGURE 3-34: USING LABELS ....	31
FIGURE 3-35: DATA DEFINITION TYPES. ....	31
FIGURE 3-36: THE OUTPUT OF THE PREVIOUS PROGRAM.....	31
FIGURE 4-1: EXAMPLE 1: COUNTER PROGRAM. ....	32
FIGURE 4-2: EXAMPLE 1 OUTPUT.....	33
FIGURE 4-3: EXAMPLE 2: SERVO-CONTROLLER PROGRAM.....	33
FIGURE 4-4: OUTPUT WAVEFORM.....	34
FIGURE 4-5: MEASUREMENTS OF THE WAVEFORM. ....	34
FIGURE 4-6: EXAMPLE 3: LCD DISPLAY. ....	35
FIGURE 4-7: LCD OUTPUT.....	36
TABLE 3-1: ALU OPERATION. ....	14
TABLE 3-2: ALU FLAGS. ....	14
TABLE 3-3: PORTS AND THEIR CORRESPONDING ID. ....	24
TABLE 3-4: RS232 PINOUT.....	28
TABLE 6-1: INSTRUCTIONS OP CODES.....	39
TABLE 6-2: REGISTERS OP CODES ....	39
TABLE 6-3: FETCH CYCLE CONTROL WORDS. ....	40

# Chapter 1: Introduction

The Central Processing Unit is a propose specific hardware circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions. The modern CPUs are designed and printed into a silicon plates in such a way that it is not possible to modify any logic circuit, they are general purpose oriented, normal computer applications do not need any special hardware. Though there are some applications where the needs of a CPU that's capable to be tuned to specific requirements, more features, custom instructions, and the ability of accumulation of the same CPU core in one hardware to improve multi-tasking problems, one important field that needs such type of CPUs is machine learning.

In order to implement such hardware that fits this requirements, the CPU has been fully designed in software, usually in a VHDL which can be synthesized in programmable hardware, such as FPGA. A soft-core processor targeting FPGAs is flexible because its parameters can be changed at any time by reprogramming the device .In our design we used the basic architecture generally used for any CPU. The Von-Neumann architecture consists of the basic parts which are well defined in the coming chapters.



# Chapter 2: Background

## 1. The Central Processing Unit and its elements

The CPU is the heart and the brain of a computer. It receives data input, executes instructions, and processes information. It communicates with input/output (I/O) devices, which send and receive data to and from the CPU. Additionally, the CPU has an internal bus for communication with the internal cache memory, called the backside bus. The main bus for data transfer to and from the CPU, memory, chipset, and AGP socket is called the front-side bus. The CPU is the unit which performs most of the processing inside a computer. To control instructions and data flow to and from other parts of the computer, the CPU relies heavily on a chipset, which is a group of microchips located on the motherboard. The CPU has two components: Control Unit; extracts instructions from memory and decodes and executes them, Arithmetic Logic Unit (ALU); handles arithmetic and logical operations. To function properly, the CPU relies on the system clock, memory, secondary storage, and data and address buses. [1]

## 2. Main types of computer architecture designs (RISC vs. CISC)

Modern CPU mainly differentiated into two types: RISC (Reduced Instruction Set Computing) which is dominated by *ARM* and *AVR* processors and CISC (Complex Instruction Set Computing) which is ran by *Intel* and *AMD*. CISC has the capacity to perform multi-step operations or addressing modes within one instruction set. It is the CPU design where one instruction works several low-level acts. For instance, memory storage, loading from memory, and an arithmetic operation but takes up multiple clock cycles (ranged from 2 to 15 cycles). Reduced instruction set computing is a CPU design strategy based on the vision that basic instruction set gives a great performance when combined with a microprocessor architecture which has the capacity to perform the instructions by using few cycles per instruction (average of 1.5 clock cycle per instruction). Also the RISC's architecture Performance is optimized with more focus on software and has multiple register sets to compensate the lack of direct access to memory by the instructions, though, the CISC's architecture performance is optimized with more focus on hardware and has much fewer registers than the RISC. [2]

## 3. Von Neumann Architecture

The modern computers are based on a stored-program concept introduced by John Von Neumann. In this stored-program concept, programs and data are stored in a separate storage unit called memories and are treated the same. This novel idea meant that a computer built with this architecture would be much easier to reprogram. Let's consider this type of architecture in details, it is mainly consisted of four elements: Control Unit; which

handles all processor control signals. It directs all input and output flow, fetches code for instructions and controlling how data moves around the system. Arithmetic and Logic Unit (ALU); is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons, It performs Logical Operations, Bit Shifting Operations, and Arithmetic Operation. Main Memory Unit (Registers) which are different and have different tasks; Accumulator: Stores the results of calculations made by ALU, Program Counter (PC): Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR). Memory Address Register (MAR): It stores the memory locations of instructions that need to be fetched from memory or stored into memory. Memory Data Register (MDR): It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory, Instruction Pointer Register (IP): It stores the most recently fetched instructions while it is waiting to be coded and executed, Instruction Buffer Register (IBR): The instruction that is to be executed immediately is placed in the instruction buffer register IBR. The fourth main element of the CPU is Buses, which are: Data Bus; It carries data among the memory unit, the I/O devices, and the processor. Address Bus; It carries the address of data (not the actual data) between memory and processor. Control Bus; It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer. [3]

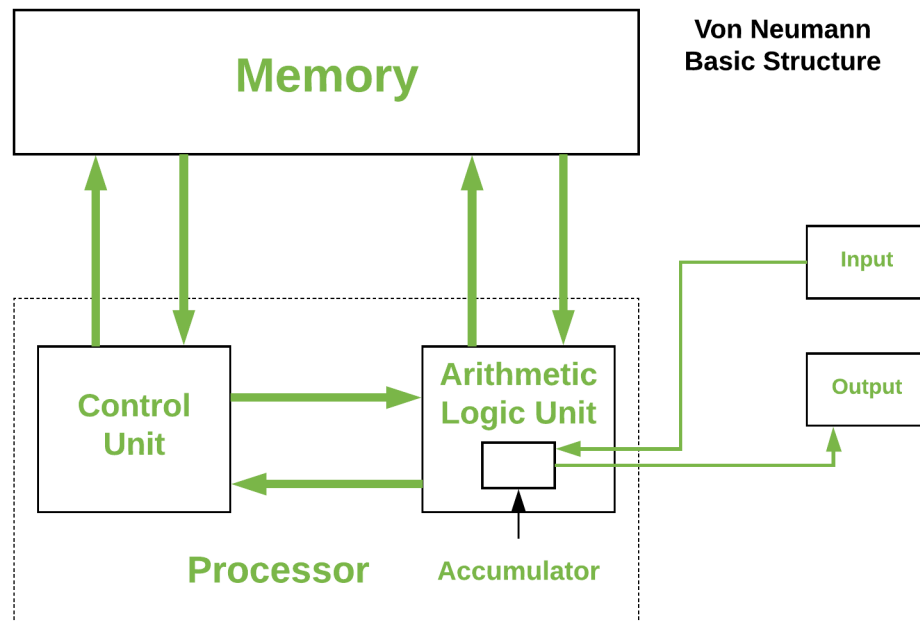


Figure 2-1: Von Neumann Architecture.

## 4. Programmable Devices

Processors are placed in a single chip with all of the necessary components like memory and timers embedded inside. It is programmed to do some simple tasks for other hardware, also it already has a fixed set of instructions, which the programmers need to learn in order to create the appropriate working program. Each of these instructions has their own corresponding block that is already hardwired into the microprocessor. Some industrial applications need a flexible devices that are capable of reconfiguration to a specific task, which lead to the invention of the programmable device. Programmable devices usually refers to chips that incorporate configurable logic circuits. These include Field Programmable Logic Devices (FPGAs), Complex Programmable Logic Devices (CPLD) and Programmable Logic Devices (PLD, PLA, PAL, GAL). These devices invariably have a number of ways to program them, and may contain internal flash to store configuration data. Such programming interfaces include serial and parallel connections to external non-volatile memory, a bus connection to a microcontroller, or JTAG connection directly to a computer for download and debugging in development. [4]

FPGAs that are constructed from the main two competitors in the market Altera and Xilinx contain arrays of configurable logic blocks that are interconnected by configurable switch arrays. They can also incorporate memory, digital signal processing, memory interface, transceiver and clock modules that are configurable, and can be wired to and from the logic blocks. The logic elements (also referred to as logic cells or complex logic blocks) are arrayed in yet larger structures (logic array blocks) to efficiently implement complex logic functions. They have highly flexible I/O banks that can be configured to support different voltages and bus standards. There may also be special hardwired memory controllers or processors available on the die for connection to the logic. The internal logic and configuration of FPGAs and CPLDs is usually performed at power up by a serial shift register style loading process that sets an invisible layer of configuration registers to set wire MUXs and logic connections. This configuration image is compiled by sophisticated software tools that interpret hardware descriptive languages or schematics. Descriptions are analyzed, optimized and synthesized into net lists. The net list is then targeted to the physical resources of the particular device in a mapping, fitting, floor-planning and optimization process so as to meet the timing requirements of the circuit. [4]

## 5. Mixing the high performance of CPUs and flexibility of the FPGAs

In order to exploit the high speeds of CPUs and the flexibility and customization of the FPGA, CPU has been implemented in the programmable device in what so-called *Soft Core CPU*. A soft-core processor is a microprocessor fully described in software using VHDL, which can be synthesized in programmable hardware, such as FPGAs. A soft-core processor targeting FPGAs is flexible because its parameters can be changed at any time by reprogramming the device. Traditionally, systems have been built using general-purpose processors

implemented as Application Specific Integrated Circuits (ASIC), placed on printed circuit boards that may have included FPGAs if flexible user logic was required. Using soft-core processors, such systems can be integrated on a single FPGA chip, assuming that the soft-core processor provides adequate performance. Recently, two commercial soft-core processors have become available: *Nios* from Altera Corporation, and *MicroBlaze* from Xilinx Inc. [5]

The main problem with the existing soft-core processor implementations targeting FPGAs is that they provide very little detail of the implementation and choices made during the development process. In this thesis, the methodology of soft-core processor development is investigated and provides the details of the steps involved to design each main component of the CPU including the registers, ALU, control unit using the FSM to reduce the Logic Units used, the RAM with one byte addressable mode and the different I/O peripherals.

# Chapter 3: Design and Implementation

Any CPU essentially has two main tasks to do; data transfer and data operations, and how fast a CPU can execute these tasks is a way to evaluate how good it is. The implemented CPU has clock speed of 50MHz along with 32 bit shared data/address bus that allows it to operate on 32bit data and access up to 4GB of ram. The Architecture implementation was based on Von Neumann architecture so it consists of registers, ALU (Arithmetic Logic Unit) and memory which are all connected to the same 32 bit bus along with Control Unit which controls the flow of data by control signals.

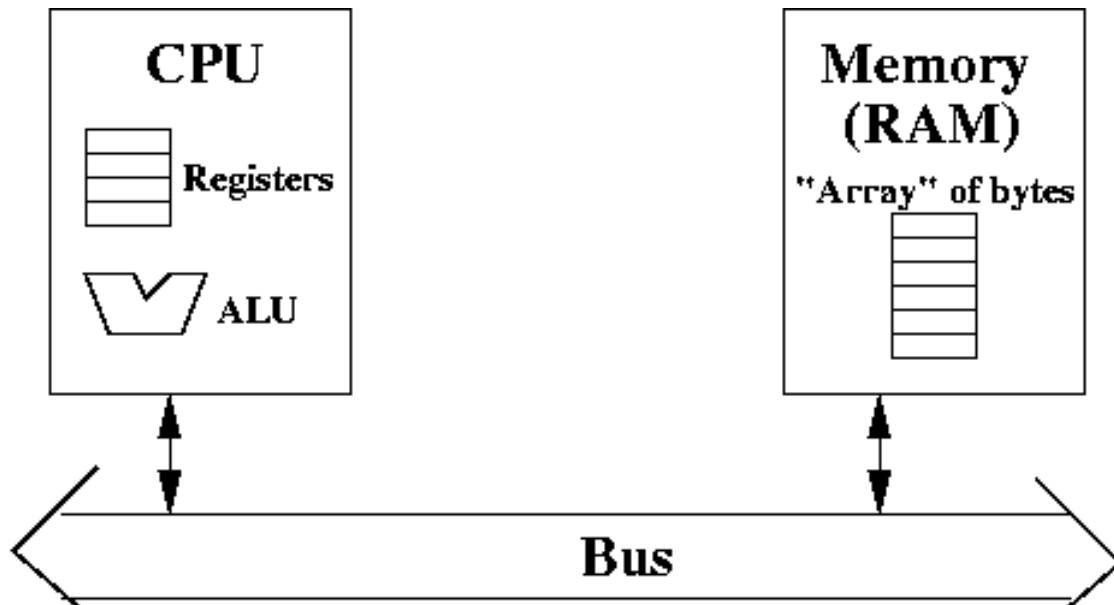


Figure 3-1: Top Level CPU Architecture

## 1. Registers

Register are the fastest way to store data however they are limited in capacity. This CPU has seven registers; four are general purpose registers (AX/BX/CX/DX), one as instruction pointer (IP) and two for stack (SP/BP).

- 1- General Purpose Registers: are used for all kind of data transfer and operations.
- 2- Instruction Pointer: indicates the address of the memory that holds the next instruction to be executed.
- 3- Stack Registers: both are related to stack but has different purposes. SP is used to hold the address of the current pointer of the stack. And BP is used as base pointer or a reference to the local variables.

All registers has 2 control signals and 32bit in/out data lines.

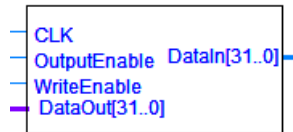


Figure 3-2: Register Pinout.

*OutputEnable*: is asynchronous read enable signal to output the inner data of the register into the data bus regardless of the clock. If it is 0, the register will output a high impedance.

*WriteEable*: is synchronous write enable signal to overwrite the internal data of the register with the one from the data bus.

## 2. Arithmetic Logic Unit (ALU)

The ALU is the responsible on the data operations in the CPU, it has 15 different operations and 3 control signal, 4 operation select signals, 5bit flags signals and 32bit data in/out data lines.

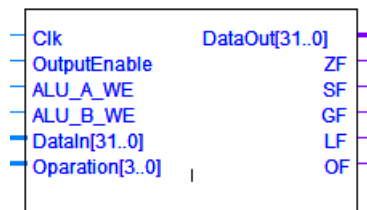


Figure 3-3: ALU Pinout.

*OutputEnable*: is asynchronous read enable signal to output the inner data of the register into the data bus regardless of the clock. If it is 0, the register will outputs a high impedance.

*ALU\_A\_WE*: is synchronous write enable signal to overwrite the internal data of register A in the ALU with the one from the data bus.

*ALU\_B\_WE*: is synchronous write enable signal to overwrite the internal data of register B in the ALU with the one from the data bus.

*Operation*: is 4 bit signals to choose the operation desired to perform.

*ZF/SF/GF/LF/OF*: are 5 signals to indicate the status of the last operation done in the ALU.

The ALU has 2 internal registers to store the data of the operands of the operation. The data is written to the registers at the falling edge of the clock and when OE signal is active the ALU calculates the result and outputs into the data bus, the flags are changes by the operation and keep its value until next operation is calculated.

Operation	Code	Effectuated Flags
Comparison	00	GF/LF/ZF
Addition	01	ZF/SF/OF
Subtraction	02	ZF/SF/OF
Logic AND	03	ZF
Logic OR	04	ZF
Logic NOT	05	ZF
Logic XOR	06	ZF
Right Rotation	07	ZF
Left Rotation	08	ZF
Increment By 1	09	ZF/SF/OF
Decrement By 1	10	ZF/SF/OF
Increment By 4	11	None
Decrement By 4	12	None
Multiplication	13	ZF/SF/OF
Addition without Flags	14	None

Table 3-1: ALU Operation.

Flag Name	Description
ZF	Set if the result is Zero
SF	Set if the result is Signed
GF	Set if value of register A is greater than register B
LF	Set if value of register A is less than register B
OF	Set if there is an overflow in the result

Table 3-2: ALU Flags.

### 3. Random Access Memory (RAM)

SRAM blocks in the FPGA are designed to address 32bit memory chunks at a time; at each unique address the ram will output different 4 bytes without intersection with other addresses. This might be easy to implement but it is possible to get some unused bytes, say you have a 1 byte size instruction at address 0x500, the next instruction is at the next address which is 0x501, now because the 4 bytes of address 0x500 are different than the ones of address 0x501 we ended up wasting 3 bytes in the address 0x500 since we need just 1 byte at that address.

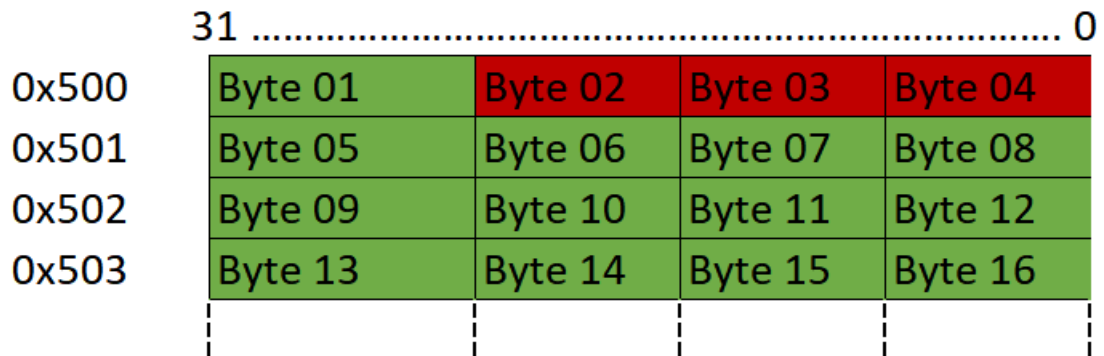


Figure 3-4: Four Byte Addressing Mode

Byte 01 (address 0x500) is the first instruction and Byte 05 (address 0x501) is the second instruction. Byte 02, 03 and 04 are unused.

So to overtake this problem the CPU has additional controllers to the RAM to address 1 byte at time, thus some memory can be saved and gives more flexibility to the RAM.

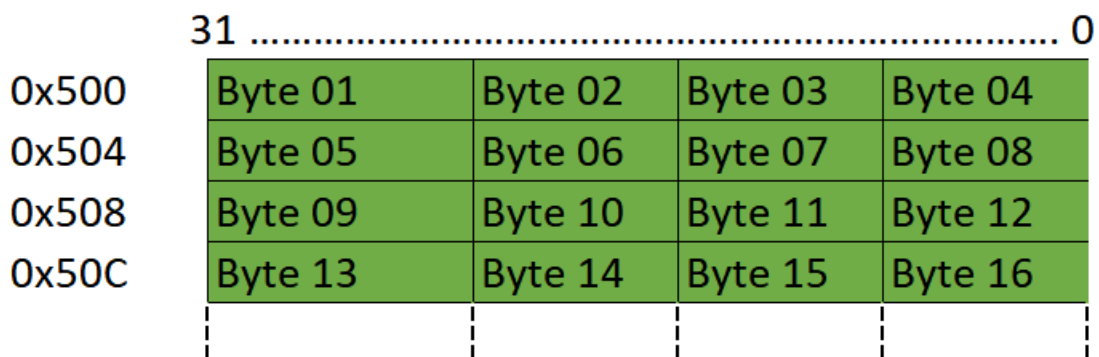


Figure 3-5: One Byte Addressing Mode

Byte 01 (address 0x500) is the first instruction and Byte 02 (address 0x501) is the second instruction. There are no unused bytes.

Now look at the top level view of the RAM :

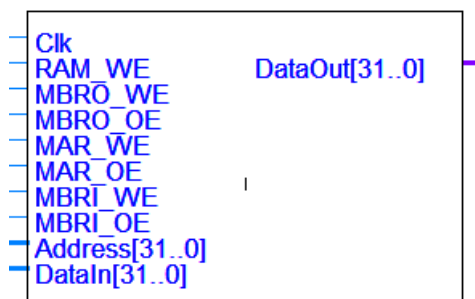


Figure 3-6: RAM Pinout



The RAM has 3 internal 32bit registers called MBRO,MBRI and MAR.The later (Memory Address Register) is the resopnsible of storing the address of the location desired to be read or write into it. The MBRO (Memory Output Buffer Register OUT) holds the data after the operation of reading the memory. The MBRI (Memory Output Buffer Register IN) holds data being written to the memory. So in order to write or read data into/from RAM a sequence of activations to the signals that controls these registers must be taken. Let's dig into these signals :

*Clk*: Clock Signal.

*RAM\_WE*: Write Enable signal of the RAM.

*MAR\_WE*: Write Enable signal of the MAR register.

*MAR\_OE*: Read Enable signal of the MAR register.

*MBRI\_WE*: Write Enable signal of the MBRI register.

*MBRI\_OE*: Read Enable signal of the MBRI register.

*MBRO\_WE*: Write Enable signal of the MBRO register.

*MBRO\_OE*: Read Enable signal of the MBRO register.

*DataIn*: 32bit input data bus.

*Address*: 32bit input address bus.

*DataOut*: 32bit output data bus.

So in order to read 4 bytes from memory the input signals have to folow the sequence :



Figure 3-7: Read Operation Waveforms

Note: Since the memory is divided into 32bit chunks and due to how the internal RAM controllers are designed it is possible to reduce the number of cycles required to read by skipping the 4<sup>th</sup> and 5<sup>th</sup> cycles of the previous waveform if the address is aligned, meaning it is a multiple of 4.

To write 4 bytes into RAM the input signals have to follow the sequence :

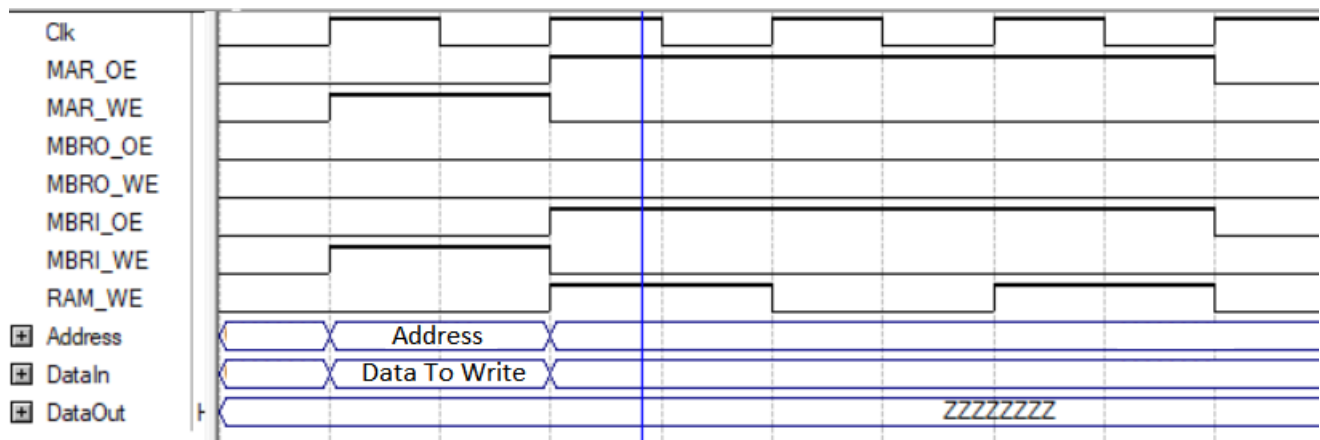


Figure 3-8: Write Operation Waveforms

Note: It is possible to pass the data being written and the address at the same time since they have different ports, but in this CPU they are both connected in the same bus so they have to be passed at different clock cycles.

As we discussed before the RAM has 2 controllers in addition to the main RAM, one for the address and other for the data in order to provide 1 byte addressing mode. Let's go into the design of the RAM module shown before and discover its controllers.

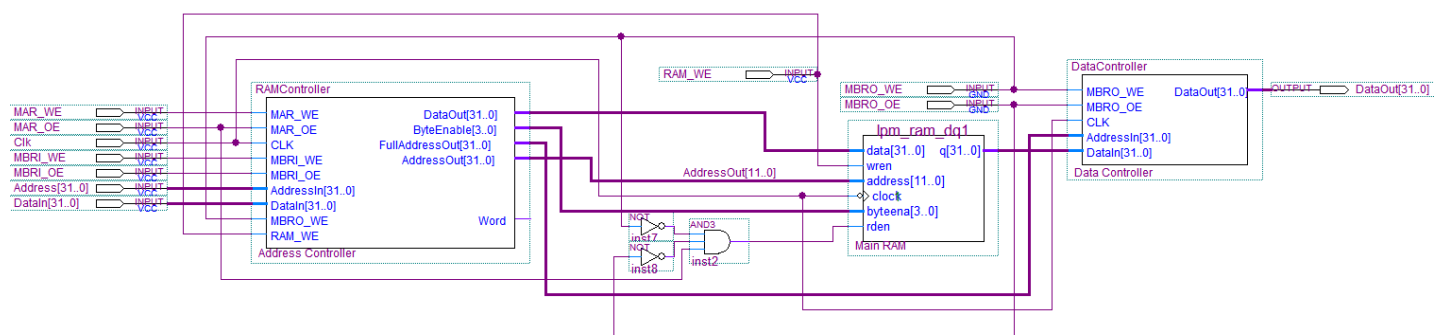


Figure 3-9: Internal Design of RAM

### 3.1. Main RAM Block

Main Ram is the FPGA's RAM block provided by Megafunctions of Quartus II. It is responsible for doing the read/write operations of the ram in the FPGA with 4byte addressing mode.

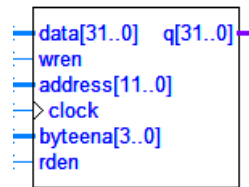


Figure 3-10: Main RAM Pinout

*q*: The output of the RAM.

*data*: 32bit input data bus being written into the RAM.

*address*: registered bus (means its value is accessed at the clock edge only) specifies the address data being read or written.

*byteena*: 4bit signal to mask written bytes into the ram. These signals allow to choose which bytes of the 4 is being written. Say you want the second byte of 32bit memory chunk in the address to be overwritten and keep the other bytes as they were before, you pass  $(0010)_2$  to byteena bus.

*rden*: Output Enable signal of the RAM. Note that RAM registers the address if this signal is set but if it is not the RAM outputs the content of previously registered address.

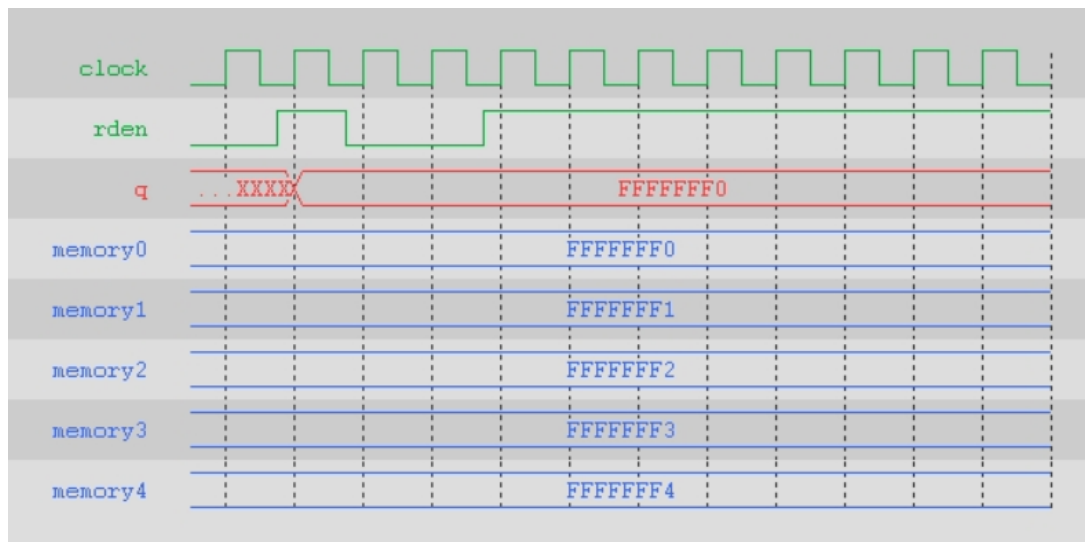


Figure 3-11: Main RAM waveforms.

### 3.2. Address Controller

As stated before the main RAM has 4 byte addressing mode though the desired RAM has 1 byte addressing mode. This module is the responsible of converting the two different modes.

Suppose you want to read the memory at address 0x13 as shown in the figure:

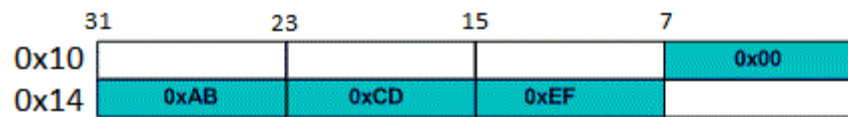


Figure 3-12: Unaligned Memory Access.

You may notice that the first byte is located in the first 32bit chunk of memory and the other three are located in second chunk. So in order to read 32bit the RAM controllers have to read the first 32bit chunk at address 0x4 (which is the first 32bit chunk's address and can be derived by last 30 bit of the desired address which is 0x13) and read the next chunk at address 0x5, Then combine the specific bytes of the two 32bit that has been read (1<sup>st</sup> byte of the first chunk and last 3 bytes of the second chunk). The purpose of Address Controller module in this task is to pass the last 30 bit of the desired address to the main ram then after reading that memory chunk increase the address by one and pass it to main ram again.

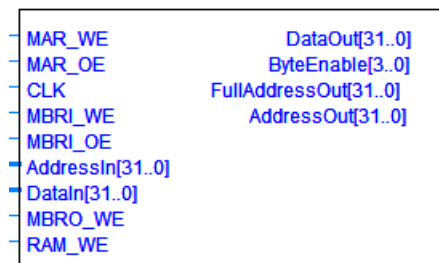


Figure 3-13: Address Controller Pinout

**FullAddressOut:** is the desired address to be read/written (1 byte addressing mode).

**AddressOut:** is the address memory chunk to be read/written (4 bytes addressing mode).

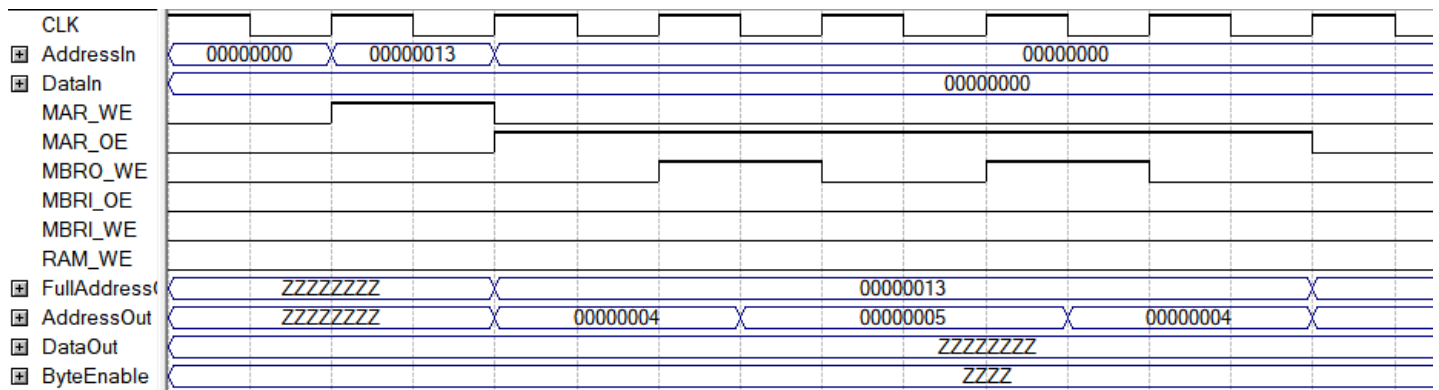


Figure 3-14: Address Controller Waveforms

At the 1<sup>st</sup> clock cycle *MAR\_WE* is set and *AddressIn* bus contains the address of the memory being read, this will register the address into the MAR register. The 2<sup>nd</sup> clock cycle is when the read operation started, *MAR\_OE* is set for the next five cycles and *FullAddressOut* outputs the value entered before in *AddressIn*. Notice that *AddressOut* is 0x4 in 2<sup>nd</sup> and 3<sup>rd</sup> cycles Then changes to 0x5, 0x4 is the address of the first memory chunk of address 0x13 (shifting 0x13 by 2 to the right) and 0x5 is the address of the next chunk.

### 3.3. Data Controller

After passing the address of the two memory chunks to main RAM which do read the data in these chunk then pass it to Data Controller. Now the data controller combine these two 32bit chunks and outputs one 32bit data which is the result of read operation.

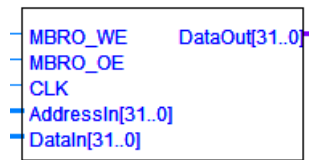


Figure 3-15: Data Controller Pinout

#### Note

The RAM uses big endian format to represent 32bit data. Endianness is the sequential order in which bytes are arranged into larger numerical values when stored in memory [6, 6]

Big Endian Byte Order: The most significant byte (the "big end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

Little Endian Byte Order: The least significant byte (the "little end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory. [7]

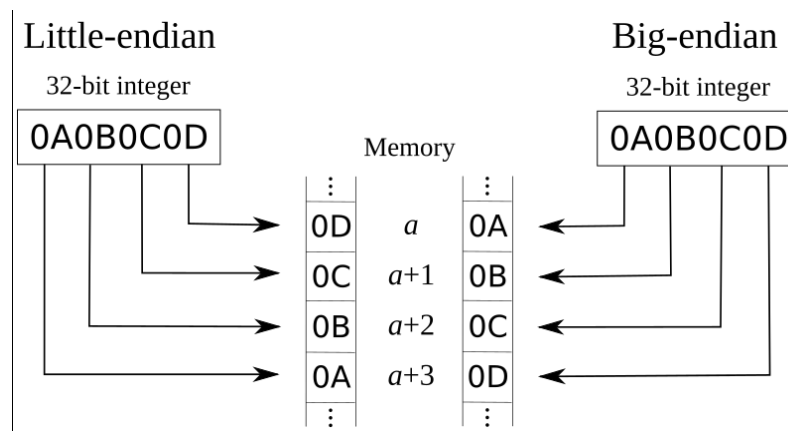


Figure 3-16: Little Endian and Big Endian Formats

## 4. Control Unit

The control unit is the most important module in the CPU it is the responsible of leading the input signals of all other components (ALU/Registers/RAM/IO) and that happens with precise clock synchronization for these components.

The CPU operations simply are flow of data between modules, each module has two data flow control signals which are *OutputEnable* and *WriteEnable*, *OutputEnable* signal is asynchronous and *WriteEnable* is synchronous. If an instruction moves data from module A to module B, the CPU sets *OutputEnable* of module A and sets *WriteEnable* of module B, though, there is a timing problem if *OutputEnable* and *WriteEnable* are set at the same time and both modules are synchronized with same clock. Since module A outputs its value at the moment *OutputEnable* is set which is the raising edge (when the control unit changes the control words) of the clock and module B registers the value in the bus at the raising edge of clock, the new value registered in module B is unknown because there is no enough time to the outputs of module A to settle up.

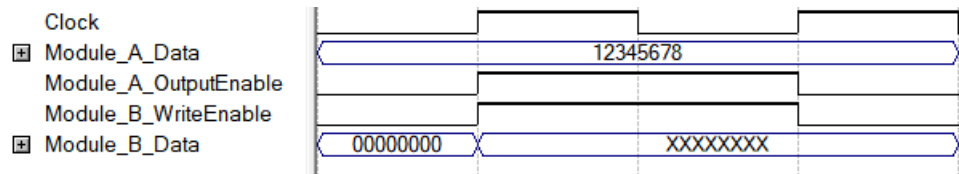


Figure 3-17: Indetermination Problem

To solve this problem the control unit which controls the input signals of modules is synchronized with different timing to the modules. In this CPU the control unit synchronized at the raising edge of the clock and all modules are synchronized at the falling edge. This means that that the value of *OutputEnable* and *WriteEnable* change at the raising edge but the write operation happens on the falling edge.

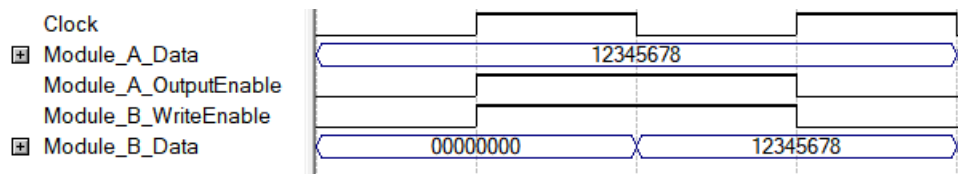


Figure 3-18: Safe Synchronization

The Execution of all CPU instruction is a matter of control the input signals (control words) of all modules, which is the task of control unit. The execution of certain instruction is done by a task called Fetch-Decode-Execute cycle.

### 4.1. Fetch Cycle

Before executing any instruction the CPU needs to load this instruction to control unit and decode it, Fetch cycle takes 16 clock cycles at most can be described as follow:

- 1- Move the pointer of the next instruction from IP register to MAR register in the RAM
- 2- Read 4bytes from RAM and move it to control unit, the control unit decode the instruction and check if the size of this instruction is more than 4 bytes, if so adds 4 to the current value of the MAR register and read the memory again and move it to the control unit.
- 3- Determine the size of the instruction (can be varied from 1 byte to 6 bytes) and add it to the current value of IP register.

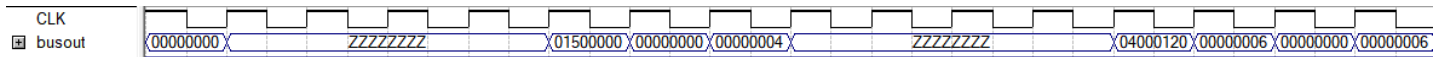


Figure 3-19: The content of The Bus During Fetch Cycle of instruction at address 0x00

Check the appendix to get the exact control words during the 16 cycles of fetch.

## 4.2. Execute Cycle

After loading the next instruction to control unit, now it is time to execute it. Unlike the fetch cycle the execution cycle differs depending on the instruction, the CPU needs a way to store each instruction control words. One way to do that is by using hard wired logic circuit which is a chain of IF/CASE statements in VHDL, though, after implementing just simple MOV instruction with its variations (all register combinations) the FPGA used nearly 120 Logic Units for that, this number is not accepted when developing CPU with large set of instructions. A more efficient way to do that is using Microprogrammable memory, the control words are stored in a special memory (not accessible by the programmer) and decoded by the instruction op code and microinstruction counter. This might be better solution than hard wired control unit, but it is still not good enough since the control word size is 36bit and there are 50 instructions in the CPU with at least 7 variations (different registers as operand), the micro programmable memory needs at least 1600 Bytes of RAM in addition to a 1 cycle delay for each microinstruction because the memory in FPGA needs at least one cycle to read data.

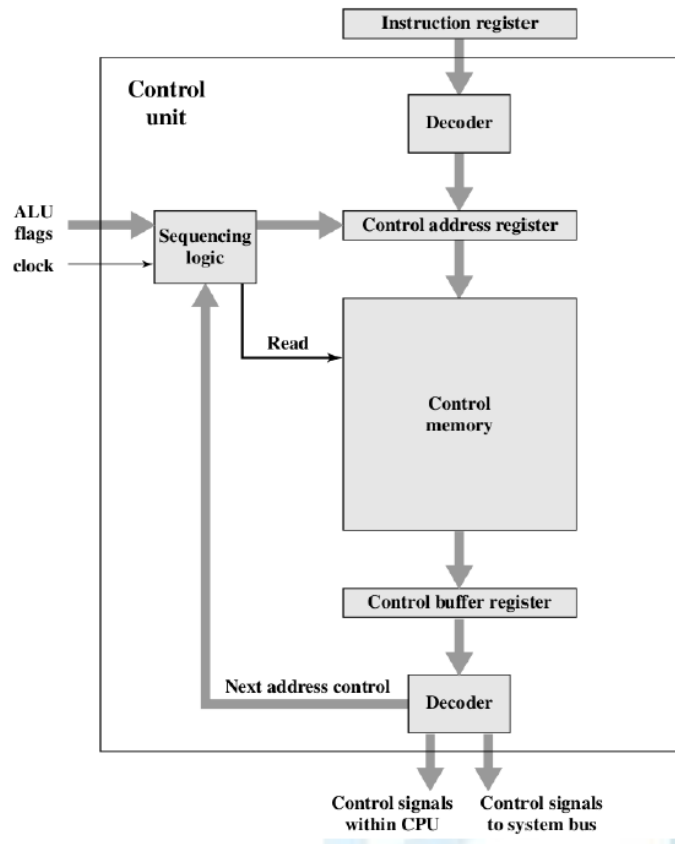


Figure 3-20: Microprogrammable Control Unit Design [8]

A better solution is to use a combination of the two method, combination logic plus a memory using Finite State Machine (FSM). It is considered the best solution because we can define 3 main states which are register data transfer, memory read and memory write, all instructions are just sequence of these states. So the FSM needs just to decode which state the CPU will execute for each instruction instead of decoding the control words directly (which is used in microprogrammable and hard wired CU).

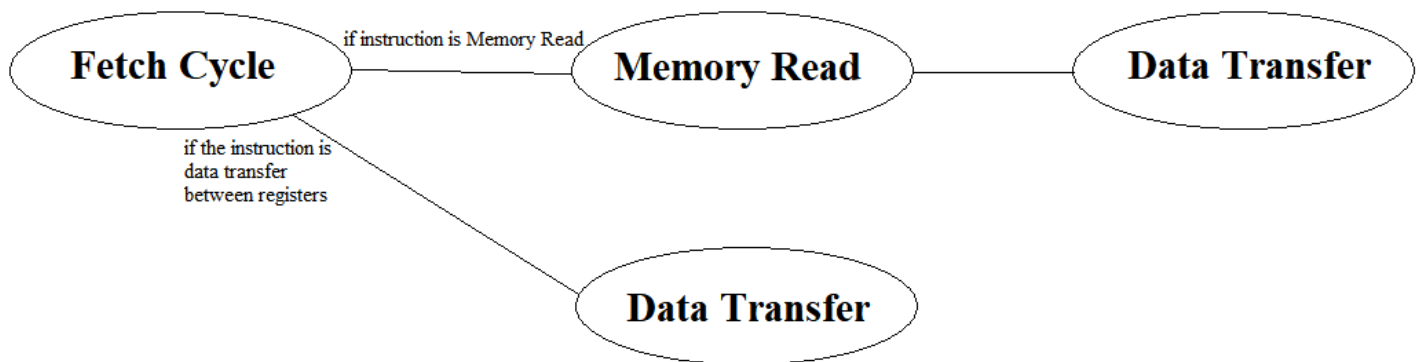


Figure 3-21: Simplified FSM Design of 2 instructions



## 5. I/O Device's Ports

Doing internal calculation is not enough for a CPU to be useful, a set of peripherals ports is needed to transfer data from/into other CPUs, sensors or output devices. Our CPU currently has 5 I/O ports: Seven Segment Output, PWM Generator, LCD Controller, Timers and RS233 port. The CPU has external 32bit bus connect all I/O ports in addition to 8bit IO\_Selector signal which indicates which peripheral is being used.

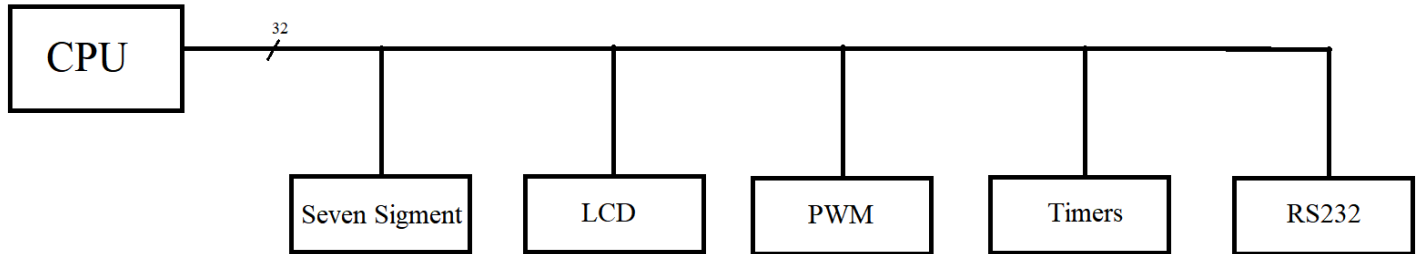


Figure 3-22: I/O Ports Design.

Port Name	ID
PWM Output	0x0-0x7
LCD	0x21
Seven Segment	0x20
Timer	0x25-0x28

Table 3-3: Ports And Their Corresponding ID.

### 5.1. Seven Segment Output

The Seven Segment Output is simplest I/O device but it was highly useful in the developing stage of the CPU specially for debugging. The FPGA development board we have used has 4 digit seven segment, it has 8 pins for digit display and another 4 pins for digit selection, and this allows to display one digit at a time though if the clock is fast enough it is able to display each digit at slight period and move to the other without noticing any blinking with human eyes.

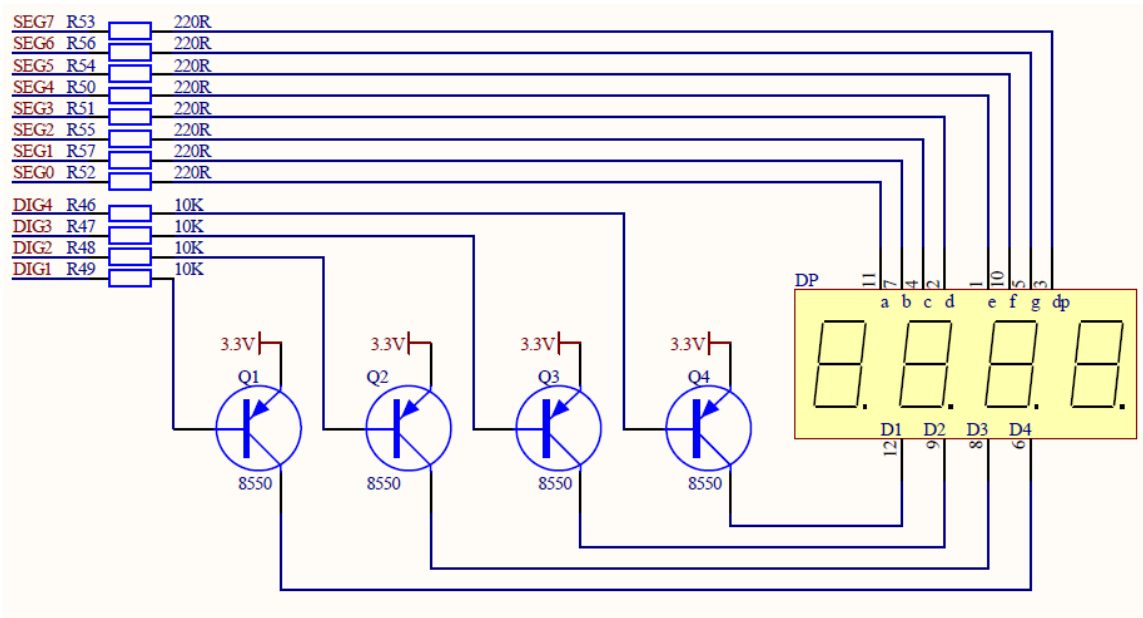


Figure 3-23: 4 Digits Seven Segment Scheme.

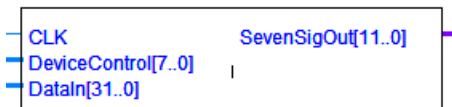


Figure 3-24: SevenSig Port Pinout.

*SevenSigOut*: the higher 4 digit are used for digit selection and the lower 8 are for seven segment display.

If the *DeviceControl* (IO\_Selector) signal of SevenSig is set to this port's ID, an internal register will store the value of *DataIn* input, then an internal counter will be used to decode a digit every fraction of time and set the corresponding digit selector.

## 5.2. LCD Port

The LCD Port is just simple 32bit register because the LCD does not require high speed timing thus it is managed via software rather than hardware. it registers the *DataIn* input whenever *DeviceControl* is set to this port's ID.

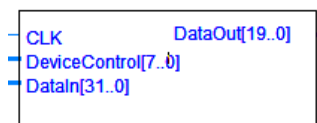


Figure 3-25: LCD Port Pinout.

## 5.3. Pulse Width Modulator (PWM)

Some devices needs analog output to operate, though it not possible to do analog output using merely digital electronics, that's why Pulse Width Modulation is used in most microprocessors. PWM is simple square

wave signal with 3.3v peak, where the duty cycle is varied, if the duty cycle is 100% taking the average of this signal will be the peak which is 3.3v, if the duty cycle is 40% then the output acts like 40% of 3.3v which is 1.32v.

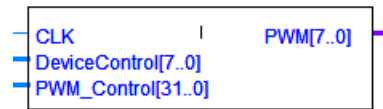


Figure 3-26: PWM Pinout.

The CPU has 8 PWM output ports their ID varies from 0 to 7, if these IDs is registered in *DeviceControl* signal, an internal register will store the value of *PWM\_Control* bus which indicates the duty cycle of generated signal on a scale of 256.

Note: each one of the 8 PMW output has its independent register to store the duty cycle control value.

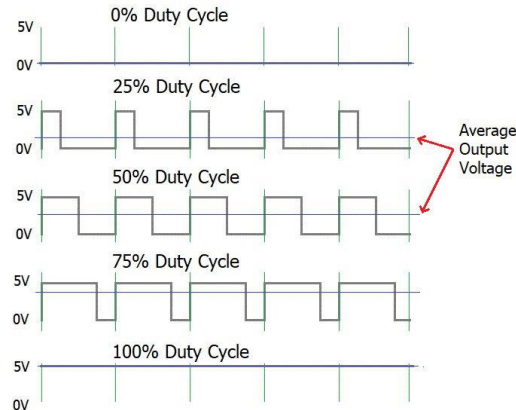


Figure 3-27: PWM Duty Cycle.

## 5.4. Timers

Many of the microcontroller applications require counting of external events such as frequency of the pulse trains and generation of precise internal time delays between CPU outputs. Just like PWM the timers output a square wave signal with controlled duty cycle though in timers the frequency can be controlled too. It has internal counter that divides the input clock which is 3.125MHz into lower frequencies.



Figure 3-28: Timers Pinout.

The CPU has 4 Timer output ports their ID varies from 0x25 to 0x28, if these IDs is registered in *DeviceControl* signal, an internal register will store the value of *DataIn* bus which indicates the timer settings,

the higher 16bits specifies the maximum value of the clock divider counter ( $T_{Counter\ Ticks}$ ), and the lower 16bits specifies the duty cycle ( $T_{Duty\ Ticks}$ ).

Example: Suppose it is required to control a servo motor which requires pulse length between 1ms and 2ms and a new pulse needs to be sent regularly every 20ms.

The timer needed has 50Hz (20ms pules) so to get  $T_{Counter\ Ticks}$

$$T_{Counter\ Ticks} = \frac{f}{f_{Desired\ Frequency}} = \frac{3.125MHz}{50Hz} = 62500$$

$f$ : input frequency which is 3.125 MHz

$T_{Counter\ Ticks}$ : the value to be set into the higher 16bits of *DataIn*

Now the counter of the timer resets whenever reaches the value 62500 (0xF424). To get the appropriate  $T_{Duty\ Ticks}$  to set the pulse length to 1ms it is required

$$T_{Duty\ Ticks} = f \times T_{Desired\ Pulse\ Legth} = 3.125MHz \times 1ms = 3125$$

$f$ : input frequency which is 3.125 MHz

$T_{Duty\ Ticks}$  : the value to be set into the lower 16bits of *DataIn*

So to control the servo motor the CPU has to register 0xF424 in the higher 16bits of *DataIn* bus and value between 0xC35 (3125d) and 0x186A (6250d) in the lower 16bits depending on the desired angle of the servo.

## 5.5. RS232 Port

RS232 is asynchronous Serial Communication Standard developed by the Electronic Industry Association in 1960s it is used for linking computer and its peripheral devices to allow serial data exchange.

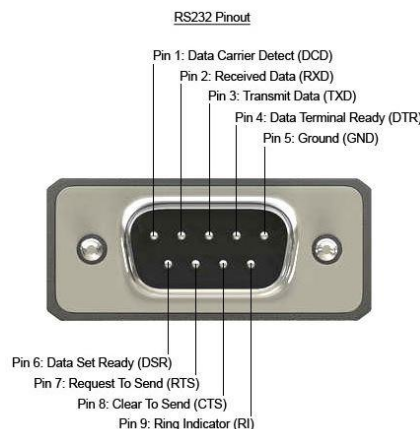


Figure 3-29:RS232 Physical Port Pinout. [9]

RS232 works as the two-way communication that exchanges data to one another. There are two devices connected to each other, (DTE) Data Transmission Equipment & (DCE) Data Communication Equipment which has the pins like TXD, RXD, and RTS& CTS. Now, from DTE source, the RTS generates the request to send the data. Then from the other side DCE, the CTS, clears the path for receiving the data. After clearing a path, it will give a signal to RTS of the DTE source to send the signal. Then the bits are transmitted from DTE to DCE. Now again from DCE source, the request can be generated by RTS and CTS of DTE sources clears the path for receiving the data and gives a signal to send the data. This is the whole process through which data transmission takes place. [10]

TXD	TRANSMITTER
RXD	RECEIVER
RTS	REQUEST TO SEND
CTS	CLEAR TO SEND
GND	GROUND

Table 3-4: RS232 Pinout

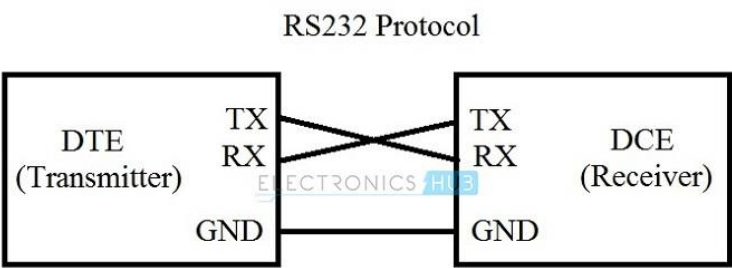


Figure 3-30: RS232 Communcation. [11]

On idle state, the transmission signal is set to logic 1, to send 8bit data, the signal is set to 0 which indicates start bit then it is followed with 8 bits of transmitted data, at the end, the signal is set to 1 as a stop bit, some transmitters use parity bit before stop bit to detect errors. The frequency of transmitted bits must be predefined by both transmitter and receiver and it is called *Baud Rate*.

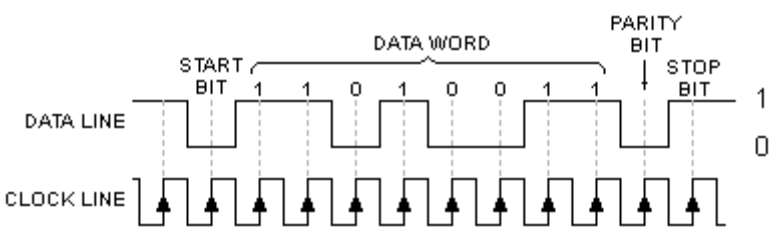


Figure 3-31: RS232 Protocol Dataflow.

In this CPU the RS232 port is used as memory programmer so it is possible to write program into CPU memory via a software in any PC. The transmitter software initiates programming mode of CPU by sending the byte 0x55, the RS232 port sends an interrupt to CPU which stops the execution of the current program and start writing the receiving data into memory. The CPU stops the writing sequence after processing predefined number of bytes, then it resets all registers and start program execution from address 0.

Note: The RS232 is an old technology and has been overthrown by the USB technology so it is rare to see a computer with RS232 port, thus it is possible to use a USB to RS232 cable which transforms USB into RS232 protocol though keeping the low speeds of RS232.



*Figure 3-32: USB to RS232 Cable.*

## **6. Assembler**

Currently the only way to write program to the CPU is by writing the code then manually translating each instruction and its operands into the corresponding machine code then send the generated bytes to the CPU via transmission software by RS232 port. This task is pretty much exhaustive when it comes to writing tens lines programs, thus, an assembler has been implemented to overtake this task and convert the assembly code directly into machine code and upload it to the CPU without tiring the user to know any information about instructions machine codes.

```

1  MOV SP, 400H
2  MOV AX, 38H
3  MOV BX, 0H
4  CALL @SEND          //INITIATE LCD
5  MOV AX, FH
6  MOV BX, 0H
7  CALL @SEND          //CLEAR LCD & SETUP CURSOR
8  MOV AX, 1H
9  MOV BX, 0H
10 CALL @SEND
11 MOV CX, @STRING
12 NEXT: MOV AX, [CX]  //CHARACTERS LOOP
13 ROR AX, 18H
14 AND AX, FFH
15 JZ @SK              //CONTINUE IF IT IS NON-ZERO CHARACTER
16 MOV BX, 1H
17 CALL @SEND          //PRINT CHARACTER
18 INC CX
19 JMP @NEXT
20 SK: NOP
21 HLT
22 STRING: DS "Aymen Sekhri" //NULL TERMINATED STRING
23 // RW : 0 WRITE      1 READ

```

Figure 3-33: Example of Program That Prints String to LCD.

The Assembler software was designed by Visual Basic .Net Framework using Object Oriented Programming, the software has a class named *instructionBuilder* which translates the assembly code of an instruction into its machine code, this class is called for each line of code to build the full machine of the program. Then, the Assembler searches for the available COM ports in the PC and displays it in a list, the user chooses the port then the assembler initiates the Serial Port Communication class in the .Net Framework and sends the machine code to the CPU.

## 6.1. Comments

It is possible to insert comments anywhere the code by using double forward slash “//”, the next words of line will be commented (colored by dark green) and not translated into machine code.

## 6.2. Labels

Since the address of certain instruction or data is dependent on the previous instructions, any modification of the first instructions must be followed with updating all addresses, here is where labels are useful they can allow the user to avoid dealing with addresses in any way. A label can be defined by using any name followed by a colon then the instruction or data desired to address. The name of the label must be alphanumeric characters or an underscore. To use the label, the user must write the @ symbol followed by the name of the label.

```

1 //Ways to use the label
2 CALL @NEWLABEL
3 JMP @NEWLABEL
4 MOV AX,[@NEWLABEL]
5 MOV AX,@NEWLABEL
6
7 //Defining a label
8 NEWLABEL: HLT
9 NOP
10 NOP

```

Figure 3-34: Using Labels

### 6.3. Data Definitions

Sometimes programmers need to define certain constants or strings to be used by the program, this assembler allows to use 3 types of data: bytes, integers and strings. The bytes are simple 8bit numbers that can be defined by DB keyword followed by the bytes, it is possible to define array of bytes by separating them with spaces or tabs. The integers are 32bit numbers which can be defined by the keyword DD, it is possible to define an array with the same way as bytes too. Finally, Strings can be defined using keyword DS followed by the ASCII string between double quotes (note that the strings are null terminated).

```

1
2 DB 10H //Defining a byte
3 DB 10H 20H 30H // Defining a byte array
4 DD 10H //Defining a integer
5 DD 10H 20H 30H //Defining a integer array
6 DS "Aymen Sekhri" //Defining a Null-Terminated ASCII string
7
8 Message: DS "Hello World" // data can be used with labels

```

Figure 3-35: Data Definition Types.

```

0000 : 10
0001 : 10 20 30
0004 : 00 00 00 10
0008 : 00 00 00 10 00 00 00 20 00 00 00 30
0014 : 41 79 6D 65 6E 20 53 65 6B 68 72 69 00
0021 : 48 65 6C 6C 6F 20 57 6F 72 6C 64 00

```

Figure 3-36: The output of the previous program.

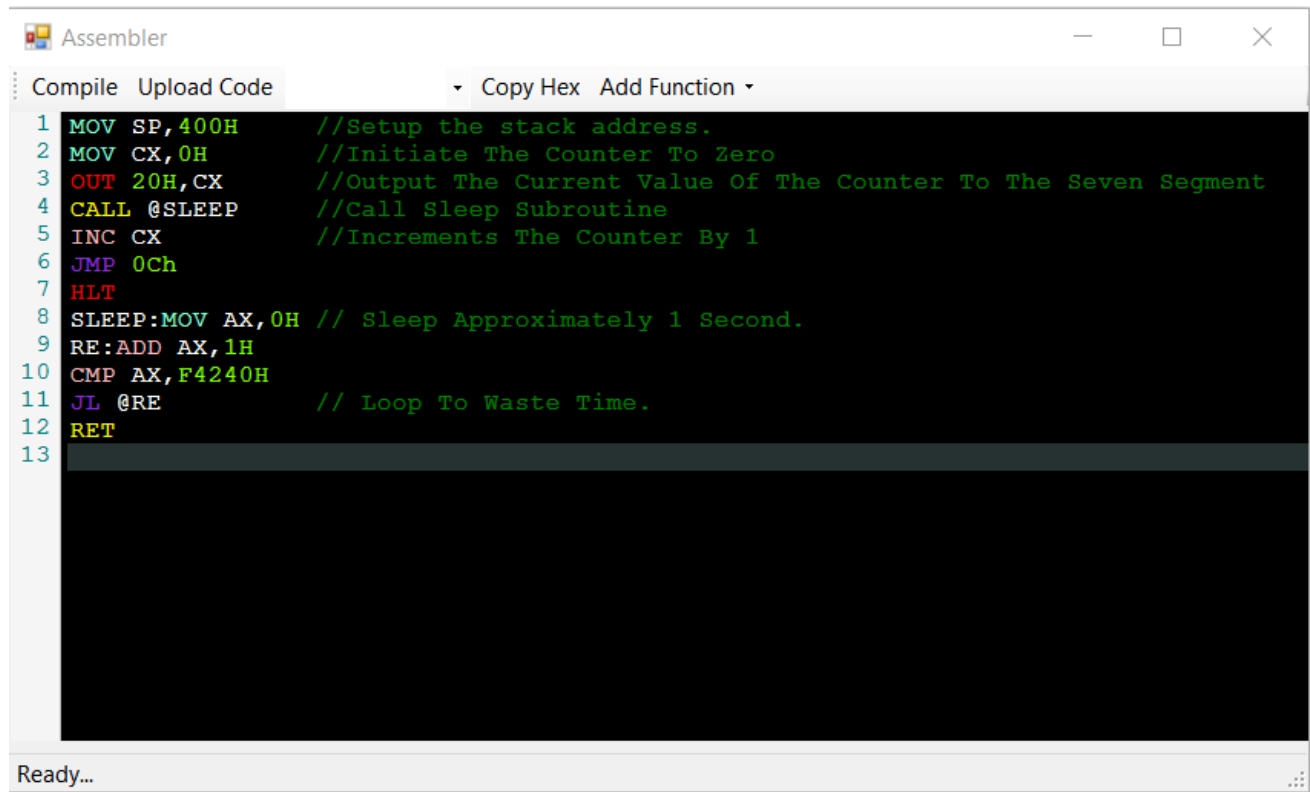


# Chapter 4: Results and Evaluation

In this chapter various code examples will be demonstrated to exploit all CPU potential and its different peripherals.

## 1. Example 1: Counter

The following program counts numbers starting from zero.

A screenshot of a software window titled "Assembler". The window has a menu bar with "Compile", "Upload Code", "Copy Hex", and "Add Function". Below the menu bar is a text area containing assembly code. The code is as follows:

```
1 MOV SP,400H //Setup the stack address.
2 MOV CX,0H //Initiate The Counter To Zero
3 OUT 20H,CX //Output The Current Value Of The Counter To The Seven Segment
4 CALL @SLEEP //Call Sleep Subroutine
5 INC CX //Increments The Counter By 1
6 JMP 0Ch
7 HLT
8 SLEEP:MOV AX,0H // Sleep Approximately 1 Second.
9 RE:ADD AX,1H
10 CMP AX,F4240H
11 JL @RE // Loop To Waste Time.
12 RET
13
```

The status bar at the bottom of the window displays "Ready...".

Figure 4-1: Example 1: Counter Program.



Figure 4-2: Example 1 Output.

## 2. Example 2: Servo Motor Controller

The following program uses the timer to control the servo motor, by sending 1ms to 2ms pulses every 20ms, so it rotates from its lowest angle to its greatest.

```

Assembler
Compile Upload Code Copy Hex Add Function
1 MOV SP,400H //Setup the stack address.
2 MOV CX,FFFF0900h //Send Pulse Each 2ms With Duty Cycle Of Almost 1ms.
3 DO:OUT 25h,CX
4 CALL @SLEEP //Call Sleep Subroutine.
5 ADD CX,50h //Add 0.02ms To The Duty Cycle (increase Servo Angle).
6 MOV DX,CX
7 AND DX,0000FFFFH
8 CMP DX,1D6AH //If Duty Cycle is Greater Than 2ms reset it to 1ms.
9 JG @SKIP
10 MOV CX,FFFF0900h
11 SKIP:MOV BX,CX
12 JMP @DO
13 HLT
14
15
16 SLEEP:PUSH AX // Sleep Approximately 1 Second.
17 MOV AX,0H
18 RE:ADD AX,1H
19 CMP AX,F424H
20 JG @RE // Loop To Waste Time.
21 POP AX
22 RET
Ready...
```

Figure 4-3: Example 2: Servo-Controller Program.

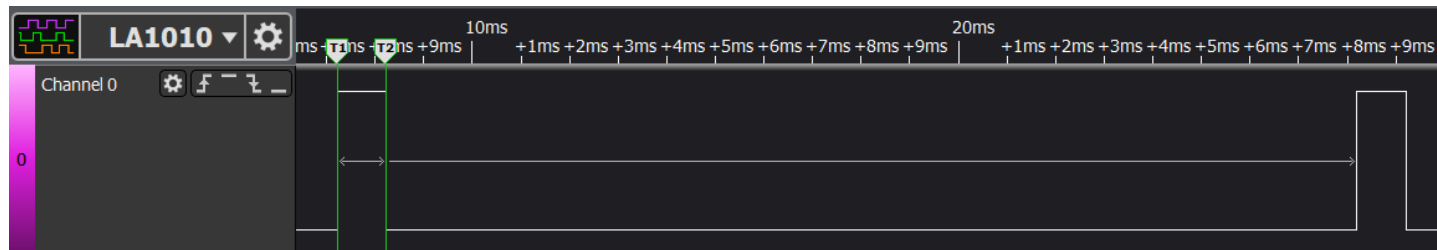


Figure 4-4: Output Waveform.

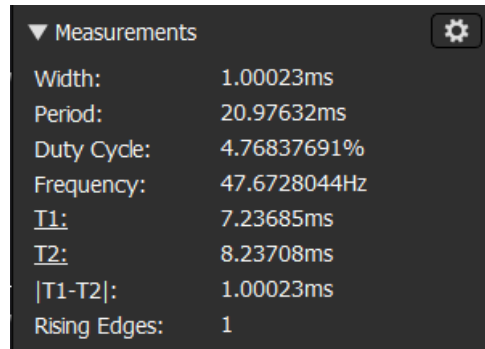


Figure 4-5: Measurements of the Waveform.

### 3. Example 3: LCD Display

The following program uses the LCD IO to display predefined string into 16×2 LCD and processes the strings greater than 16 character by displaying it in a new line.

```

1  MOV SP,400H
2  MOV AX,38H
3  MOV BX,0H
4  CALL @SEND      //INTIATE LCD
5  MOV AX,FH
6  MOV BX,0H
7  CALL @SEND      //CLEAR LCD AND SETUP CURSOR
8  MOV AX,1H
9  MOV BX,0H
10 CALL @SEND
11 MOV CX,@STRING
12 NEXT:MOV AX,[CX] //CHARACTERS LOOP
13 MOV BX,1H
14 ROR AX,18H
15 AND AX,FFH
16 JZ @SK          //CONTINUE IF IT IS NON-ZERO CHARACTER
17 MOV BX,1H
18 CALL @SEND      //PRINT CHARACTER
19 INC CX
20 MOV AX,CX
21 SUB AX,@STRING
22 OUT 20H,AX
23 CMP AX,10H      // CHECK IF's The 16th character
24 JNZ @NEXT
25 MOV AX,A9H
26 MOV BX,0H
27 CALL @SEND      // IF so,Mov THE CURSOR TO THE NEW LINE
28 JMP @NEXT
29 SK:HLT
30 STRING: DS "Welcome To Soft      Core CPU" //NULL TERMINATED STRING
31 // RW : 0 WRITE      1 READ
32 // RS : 0 COMMAND    1 DATA
33 SEND:ROL AX,1H
34 ROL AX,1H
35 ROL AX,1H
36 OR AX,BX
37 OUT 21H,AX
38 CALL @SLEEP
39 OR AX,4H
40 OUT 21H,AX
41 CALL @SLEEP
42 AND AX,FFFBH
43 OUT 21H,AX
44 CALL @SLEEP
45 MOV AX,0H
46 OUT 21H,AX
47 CALL @SLEEP
48 RET
49 SLEEP:MOV DX,0H
50 RE:ADD DX,1H
51 CMP DX,1524H
52 JL @RE

```

Ready...

Figure 4-6: Example 3: LCD Display.

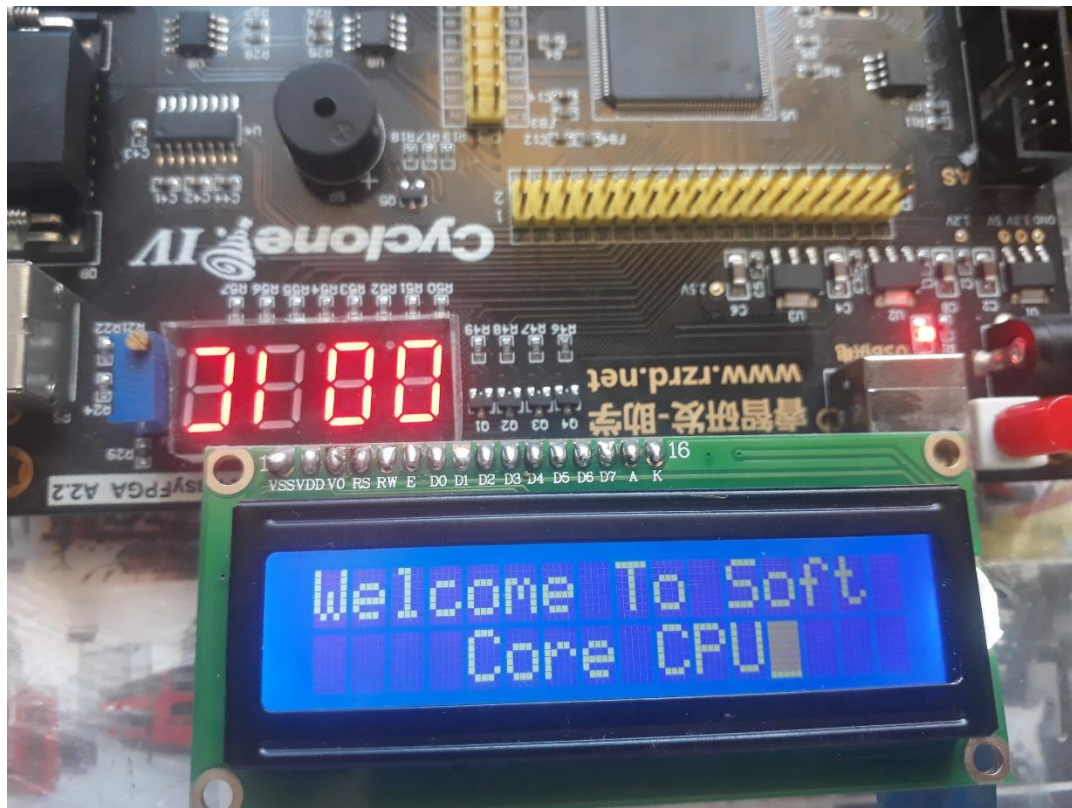


Figure 4-7: LCD Output.

## Chapter 5: General Conclusion

At the end of this project, we have improved our knowledge related to the modern CPUs architecture (for both hardware and software) in particular, and in digital systems in general. New concepts are also discovered which allows us to add more enhanced features to our CPU to be more efficient such as one byte addressing mode and peripheral mapped IO devices. It is also noticeable that all obvious courses that we learnt in both computer engineering and digital electronics with VHDL were so helpful to implement the project.

Due the limited time, the project did not go exactly as planned, a few more points has been abandoned but can be added in the future releases. One interrupt has been implemented in the CPU which was the RS232 port write operation, though it was too simple and has no Interrupt Service Routine (ISP) which will be executed to handle the interrupt whenever it happens. In a future work it is possible to implement custom interrupts with its corresponding ISP such as mouse and keyboard interrupt. Also the communication protocol used by the RS232 was too simple and not totally secure, hashing algorithm could be used to authenticate the received data. In addition, in the current implementation of the CPU, it can only execute one flow of program, it could be designed with multithreading which is the ability of a CPU to provide multiple threads of execution concurrently and perform different task simultaneously. And lastly, it is possible to design a Protected Mode which gain some features including virtual memory, file paging, multi-processing and privilege levels. This mode isolates the hardware from the user using an application layer.

# Chapter 6: Appendix

## 1. Table of Instructions

m32: memory direct addressing mode.

imm32: immediate addressing mode.

r32: Register addressing mode.

Instruction	Op code	Size	Architecture	Cycles - 16	Description
MOV r32,imm32	0x01	6	0x01/r32/imm32	1	
MOV r32,r32	0x02	2	0x02/r32r32	1	
MOV r32,m32	0x03	6	0x03/r32/m32	6	
MOV r32,[r32]	0x04	2	0x04/r32r32	6	
MOV [r32],r32	0x05	2	0x05/r32r32	5	
MOV [r32],imm32	0x14	6	0x05/r32r32	5	
ADD r32,r32	0x06	2	0x6/r32r32	2	
SUB r32,r32	0x07	2	0x7/r32r32	2	
MUL r32,r32	0x08	2	0x8/r32r32	2	
AND r32,r32	0x09	2	0x9/r32r32	2	
OR r32,r32	0x0A	2	0xA/r32r32	2	
CMP r32,r32	0x0B	2	0xB/r32r32	2	
ADD r32,imm32	0x0C	6	0xC/r32/imm32	2	
SUB r32,imm32	0x0D	6	0xD/r32/imm32	2	
MUL r32,imm32	0x0E	6	0xE/r32/imm32	2	
AND r32,imm32	0x0F	6	0xF/r32/imm32	2	
OR r32,imm32	0x10	6	0x10/r32/imm32	2	
CMP r32,imm32	0x11	6	0x11/r32/imm32	2	
ROR r32,imm32	0x12	6	0x12/r32/imm32	2	Accepts values 1,8,16 and 24
ROL r32,imm32	0x13	6	0x13/r32/imm32	2	Accepts values 1,8,16 and 24
INC	0x15	2	0x15/r32.00	2	
DEC	0x16	2	0x16/r32.00	2	
PUSH r32	0x20	2	0x20/00.r32	6	
PUSH imm32	0x21	5	0x21/imm32	6	
POP r32	0x22	2	0x22/r32.00	7	
JMP m32	0x30	5	0x30/m32	1	
JMP r32	0x31	2	0x31/r32	1	

CALL m32	0x23	5	0x23/m32	8	
CALL r32	0x24	2	0x24/00.r32	8	
RET	0x22	2	0x22/4.00	7	RET = POP IP
JZ imm32	0x40	5	0x40/m32	1	If zero flag is set
JS imm32	0x41	5	0x41/m32	1	If sign flag is set
JO imm32	0x42	5	0x42/m32	1	If overflow flag is set
JL imm32	0x43	5	0x43/m32	1	If less flag is set
JG imm32	0x44	5	0x44/m32	1	If greater flag is set
JNZ imm32	0x45	5	0x45/m32	1	
JNS imm32	0x46	5	0x46/m32	1	
JNO imm32	0x47	5	0x47/m32	1	
JZ r32	0x48	2	0x48/00.r32	1	
JS r32	0x49	2	0x49/00.r32	1	
JO r32	0x4A	2	0x4A/00.r32	1	
JL r32	0x4B	2	0x4B/00.r32	1	
JG r32	0x4C	2	0x4C/00.r32	1	
JNZ r32	0x4D	2	0x4D/00.r32	1	
JNS r32	0x4E	2	0x4E/00.r32	1	
JNO r32	0x4F	2	0x4F/00.r32	1	
OUT Port,r32	0x50	3	0x50/r32/imm8	1	
HLT	0xF0	1	0xF0	1	

Table 6-1: Instructions OP Codes.

## 2. Register OP Codes

Register	Code	Description
AX	0	General Purpose Register
BX	1	General Purpose Register
CX	2	General Purpose Register
DX	3	General Purpose Register
IP	4	Instruction Pointer
SP	5	Stack Pointer
BP	6	Base Pointer

Table 6-2: Registers Op Codes



### 3. Fetch Cycle

Cycle Number	Active Control Words	Description
01	MAR_WE , IP_OE	Move Address Of next instruction to MAR register
02	MAR_OE	Read 4 bytes from Memory at the address of the next instruction and store it into IR register.
03	MAR_OE , MBRO_WE	
04	MAR_OE	
05	MAR_OE , MBRO_WE	
06	MAR_OE , MBRO_OE , IR_WE	
07	IP_OE , ALU_A_WE	Add 4 to the address of the next instruction.
08	MAR_WE , ALU_OE , ALU_OP = 0xB	
09	MAR_OE	Read Memory of next 4 bytes of the instruction and store it into IR register.
10	MAR_OE , MBRO_WE	
11	MAR_OE	
12	MAR_OE , MBRO_WE	
13	MAR_OE , MBRO_OE , IR_WE	
14	ALU_B_WE	Decode instruction size and add it to IP register.
15	IP_OE , ALU_A_WE	
16	IP_WE , ALU_OE , ALU_OP = 0xE	

Table 6-3: Fetch Cycle Control Words.



# Bibliography

- [1] "What does Central Processing Unit (CPU) mean?," techopedia, [Online]. Available: [www.techopedia.com/definition/2851/central-processing-unit-cpu](http://www.techopedia.com/definition/2851/central-processing-unit-cpu).
- [2] "What is The Difference Between RISC and CISC Architecture," elprocus, [Online]. Available: <https://www.elprocus.com/difference-between-risc-and-cisc-architecture/>.
- [3] "Computer Organization | Von Neumann architecture," geeksforgeeks, [Online]. Available: <https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/>.
- [4] "Programmable Devices," Arrow, [Online]. Available: <https://www.arrow.com/en/categories/programmable-devices>.
- [5] F. Plavec, "SOFT-CORE PROCESSOR DESIGN," University of Toronto, Toronto - Canada, 2004.
- [6] A. S. Tanenbaum, Structured Computer Organization, Todd M. Austin: Prentice Hall PTR, 4 August 2012.
- [7] "Big Endian and Little Endian," chortle.ccsu, 2019. [Online]. Available: [chortle.ccsu.edu/assemblytutorial/Chapter-15/ass15\\_3.html](http://chortle.ccsu.edu/assemblytutorial/Chapter-15/ass15_3.html).
- [8] L. Tarrataca, "Chapter 20 - Microprogrammed Control (9th edition) - Federal Center for Technological Education of Rio de Janeiro," Rio de Janeiro.
- [9] B. Kirstein, "RS232 9 Pin Pinout," stratusengineering, 20 July 2016. [Online]. Available: [www.stratusengineering.com/rs232-9-pin-pinout](http://www.stratusengineering.com/rs232-9-pin-pinout).
- [10] T. Sharma, "RS232 Serial Communication Protocol: Basics, Working & Specifications," [Online]. Available: [circuitdigest.com/article/rs232-serial-communication-protocol-basics-specifications](http://circuitdigest.com/article/rs232-serial-communication-protocol-basics-specifications).
- [11] ANUSHA, "RS232 Protocol – Basics," 1 JULY 2017. [Online]. Available: [www.electronicshub.org/rs232-protocol-basics](http://www.electronicshub.org/rs232-protocol-basics).