CPU Implementation With VHDL

by
Student BRAZZA Walid Mokrane

Professor TOUZOUT Walid

A report submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Electrical Engineering and Electronics

Institute of Electrical Engineering and Electronics
Boumerdes
May 17, 2024

To my father,my mother,my siblings and anybody that wishes me the best

Abstract

This report is about Central Processing unit (CPU) implementation with VHDL using Structural modeling,where the whole design is separated into smaller parts that are linked together to complete the design. This shows the power of abstraction when designing complex modules,since we reduce them to their essential parts. The idea is to solve the problem using a hierarchy from basic electronic components, like multiplexers,registers and other logic elements, to then make arithmetic and logic units,memory elements and controlling them to execute desired instructions.

Executive Summary

The processor, also known as Central Processing Unit is a very important element in almost any electronic device, such as smart phones and computers ,It is responsible for accessing/manipulating data,storing it or controlling IO operations. CPUs -as long as they are powered- are in a constant *fetch* , *decode* and *execute* cycle, this way it controls the flow of data based on the instructions that are provided.This behaviour makes it the **Brain** of the electronic device that it is controlling. Fetching means reading data from memory in order to determine the next task to perform. Decoding and Executing is determining the following *decisions* that must be taken in order the execute the given action.
Processors can be devided to Two categories. **First**, Application Specific Integrated Circuits (ASIC), those are designed for specific goals such as CPUs in digital watches, electronic toys and controllers in control systems, They cannot be programmed to execute other tasks since they are optimized to execute a specific kind of operation. **Second**, General Purpose Microprocessors, this kind is used in desktop computers due to there power and speed in executing instructions. They can be programmed to suit the user's needs,They are extremely flexible and only limited by the user's skill or budget.
Even though ASICs and general Purpose CPUs have different purposes they can use the same design philosophy, for example, according to Von Neumann (1945) that was published in 1945, also known as stored program computer, it consists of Arithmetic and logic unit, control unit , memory, and input/output devices.The four parts them selves are divided to smaller parts that will be discussed in the future chapters.

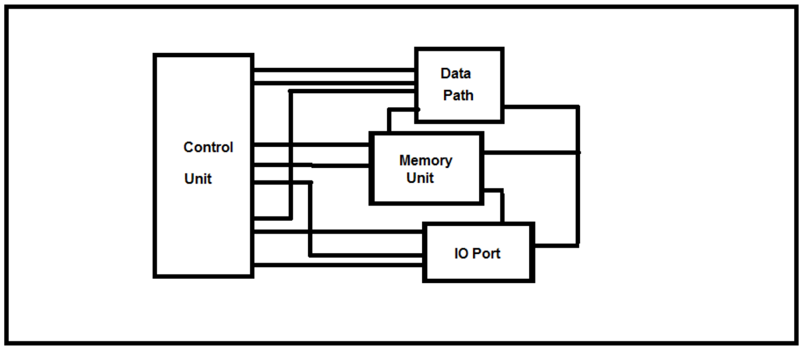


Figure 1: Von neumann architecture of a CPU

Acknowledgments

In this part I would like to thank my parents for their scarification for my learning, my siblings and my friends for supporting me through the good and the hard times and drawing a smile on my face all the time .
I would like to say thanks to the professors for their efforts and sacrifices to make the hardest topics seem simpler. for example my design was inspired by Z80 CPU, after our teachers explained the material and the architecture of computers the implementation with VHDL was easier.

- Mr A.KHOAS
- Mrs D.BELAIDI
- Mr A.BENZEKRI
- Mr E.BOUTLAA

I would like to thank my advisor for guiding me through this report and providing me with the necessary means to debug and implement my design, his experience paved the way for me to work with ease.
From the depths of my heart THANK YOU!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As stated in Executive Summary CPUs are devided into four modules that can be divided into even simpler modules, like adders, shifters, comparators ...etc.

The challenge is to implement the design with VHDL, it is a double acronym for Very High Speed Integrated Circuits Hardware Description Language. It is not a programming language as C++, where the code is compiled to generate a binary file that the CPU executes, Hardware Description Languages generate a circuit that describes the code written. The wonderful part about VHDL is that it allows for structural modeling where the user links different modules to get more complicated modules, by using Von neumann architecture and abstraction, we can determine that the CPU is devided to its core components that are simpler in terms of complexity, and VHDL allows us to design these components and link them, this makes the CPU implementation less complicated as we can see in the following chapters.

1.1 Early stages

Processors date to as early as the 1930s where vacuum tubes and relays were used, those components formed the foundation for digital computing. The largest achievement was the development of the Electronic Numerical Integrator and Computer, ENIAC for short, was developed in 1945 consisting of 1700 vacuum tubes, forming one of the first General purpose computers.The main draw back was the Large power consumption of those computers.

1.2 Transistor's Revolution

The invention of transistor in 1950s revolutionized the evolution of processors, since intense and complex computations were made faster and consumed less power, Unlike vacuum tubes, transistors are smaller enabling **Denser** integrated circuits that allowed for more components on a single chip, making them more reliable resulting in more efficient computer systems.

1.3 Parallel Processing and Multi Threading

The invention of transistors lead to development of integrated circuits that sparked rapid success that improved computing accessibility, resulting in microprocessors like the intel 8086 microprocessor that set the x86 architecture standard used in many computer systems. In the 2000s, microprocessors took a massive leap with the introduction of multi core systems that enhanced computing power and energy efficiency. By using multiple processing cores,multi threading enables multitasking abilities that led to software development, developers use the full power of distributing tasks over the given cores.

1.4 Some Challenges

Even though multi threading provides great power consumption and stronger computing power, poor management of the available cores may cause increased consumption that could damage the component's life time. Software developers need to adapt their practices in order to consume power efficiently without compromising performance.

1.5 Chapter Outline

Chapter 2: is about the hardware that forms the internal structure of the CPU.
Chapter 3: is about the software implementation and the virtual language that is used, this makes coding the CPU easier and simpler since writing assembly is more understandable than programming in assembly, Actually the more Human readable the language the easier it is to use, but the harder is for the CPU to use.
Chapter 4: is about the functionality of the designed cpu where a program is assembled, loaded to the FPGA and disassembled.
appendix A: provides opcodes, clock cycles and machine cycles of all the instructions in my instruction set.
appendix B: provides some insight into how memory is initialized in quartus II, and how to use MIF files.

Chapter 2

Hardware System Design

2.1 My Design

In a former semester we studied computer architecture and microprocessor design of the Z80, I found studying it very fun so I decided to implement my own CPU with an instruction set that is similar to the Z80's instruction set. After careful planing I settled for and 8-bit CPU, for a more detailed view on instruction set, take a look on appendix A.
Visit: "github.com/dWALIDb/CPU-implementation-with-VHDL" to have access to the code base for vhdl implementaion, as well as the C++ assembler code.
The next sections Will explain my implementation of von neumann architecture -see figure 1-.

2.2 The Data Path

The data Path is responsible for all the data manipulation for example addition, subtraction, comparison...etc.The data is stored in a register that is used for all the operations, it is called accumulator, so the result of the last operation is written to the accumulator.The flag register is not affected by loading or branching instructions, it is affected by arithmetic operations and comparison instructions.The flag register consists of :

- GREATER: Set when comparison instruction and the accumulator is greater.
- EQUAL: Set when comparison instruction and both operands are equal.
- LOWER: Set when comparison instruction and the accumulator is lower.
- CARRY: Set when a carry is present in the ALU or in the shifter.
- ZERO: Set when accumulator is ZERO or when result in ALU is ZERO.

- SIGN: Set when accumulator MSB is HIGH.
- OVERFLOW: Set when operands and result have different signs indicating signed overflow.

The following figure shows the Internal design of the DataPath:

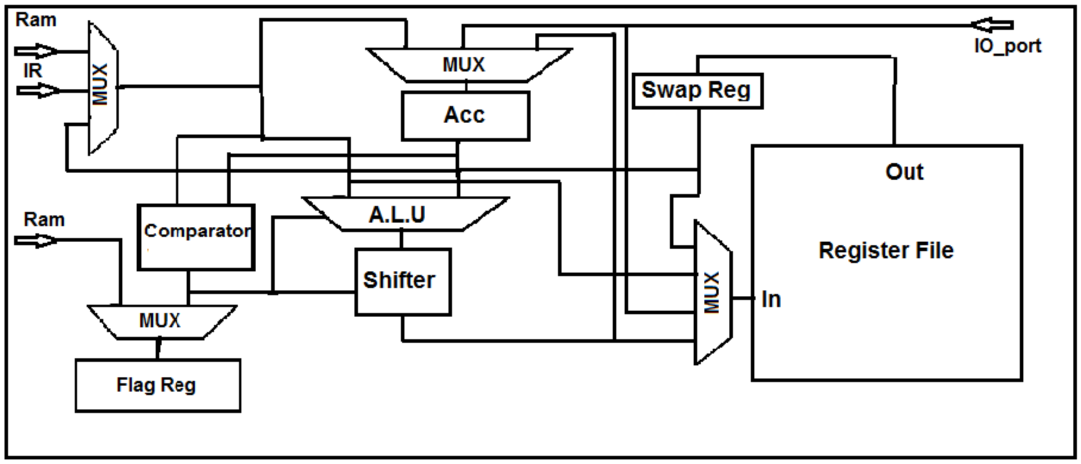


Figure 2.1: Internal Design Of the Data Path

2.2.1 The Arithmetic and Logic Unit

The arithmetic and logic unit -ALU for short-, it is the heart of the data path where all logical and arithmetic operations occur. this module takes 2 inputs and gives and output according to the ALU operation input, It provides a variety of flags that are extremely handy for the control unit 2.5.

The Flags provided are:

- CARRY:it is set when output exceeds 8-bits indicating unsigned overflow.
- ZERO: it is set when accumulator is zero or when the result is zero.
- SIGN: it is set when the MSB is set indicating negative number.
- OVERFLOW:it is set when input and output have different signs indicating signed overflow.

The following table shows the Different ALU operations:

Op-code(HEXADECIMAL)	Operation
1	ADD
2	SUB
3	INC A
4	INC B
5	DEC A
6	DEC B
7	A AND B
8	A OR B
9	NOT A
A	NEG A
B	A XOR B
C	PASS B
Others	PASS A

Table 2.1: ALU Operations and their corresponding opcodes

2.2.2 Shifter

The shifter does a binary shift of data concatenating a zero on the left or the right depending on the op-code.

Op-code(binary)	operation
00	PASS
01	SR
10	SL
11	ROT

Table 2.2: Shifter operations and their Op-codes

When using the ROT op-code the MSB is copied to an output bit that is read by the flag reg.

2.2.3 Comparator

To perform signed comparison, it has three outputs:

- LOWER: when the accumulator is lower than the second operand.
- GREATER: when the accumulator is greater than the second operand.
- EQUAL: when the accumulator is equal the second operand.

These outputs are very helpful in the Control Unit, since they are used execute conditional branching.

2.2.4 Register File

The Register File is a tiny memory, that is used to store temporary data to be used in the execution of the program. it has read and write signals that are used to read from/write to the corresponding address. A register is connected from its output to its input to be used for swapping, the register is called swap register.

2.3 Memory Unit

The memory unit contains all the components that ensure that the program is ran sequentially.

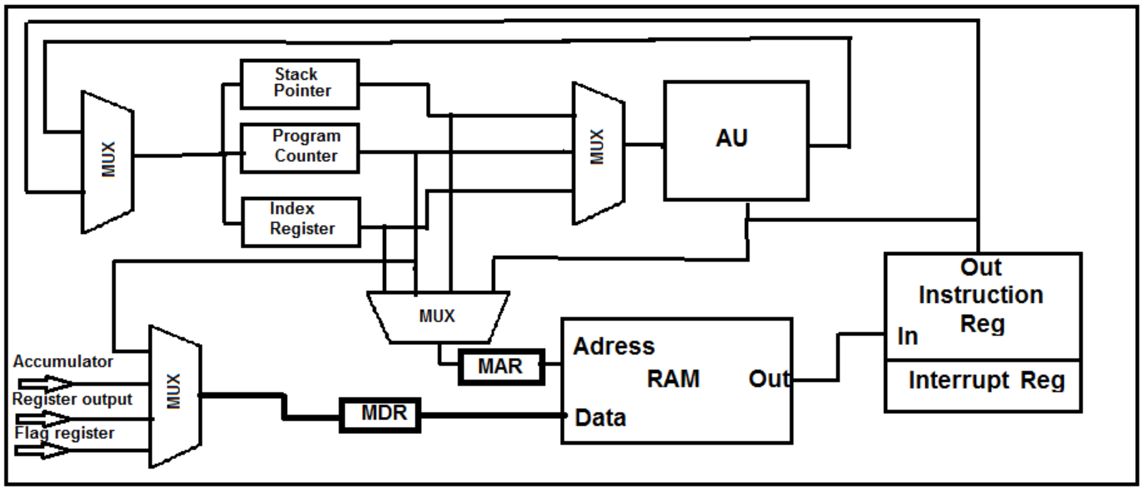


Figure 2.2: Internal Connection of Memory Components

2.3.1 Special registers

These special register point to desired memory locations such as:

- Program counter: A register that holds the current instruction to be performed, it is incremented every instruction ensuring that instructions are ran sequentially. It could be loaded with a new or offset with a quantity by calling a sub routine, using relative jumps or absolute jumps.refer to appendixA.
- Stack Pointer: A register that holds the address of the top of the stack, it is very useful for subroutines,interrupts and even to store/retrieve data that shouldn't appear outside subroutines via the push/pop instructions.

- Index Register: It is useful for arrays where data is stored contiguously, this enables dynamic programming by initializing the index register and performing the desired manipulation for all the data in the array.
- Memory Address Bus: it is connected directly to the address bus of the random access memory.
- Memory data Bus: it is connected directly to the data bus of the random access memory.
- Instruction Register: This register holds the Opcode and the operands for the control unit to manipulate.
- Interrupt: This register holds the Interrupt Subroutine Location in memory with the instruction "ei" that means enable interrupt.

2.3.2 Random Access Memory

The location where the program is stored. The user can read from the memory to the accumulator as well as the other registers,and vise versa. Memories in Quartus II are initialised using two types of files, using hexadecimal format or mif format that stands for Memory Initialization File, they can be referenced via attributes. The issue is that my simulation tool (Model-sim altera) doesn't recognize the attribute so I programmed a function that takes a text file and initializes the RAM for simulation and I use them depending if i want to test on FPGA or to simulate for debugging.Refer to appendixB or refer to Pedroni (2010) in Memory Initialization Files section for further details.

2.3.3 Arithmetic Unit

The arithmetic unit Is used t increment the address pointers or calculate the offset depending on the op-code.

Op-code	Operation
00	PASS
01	ADD
10	INC
11	DEC

Table 2.3: AU operations and their Op-codes

2.4 The Input Output Port

This is a buffer that holds data from the user, or it could be used to display data to the user. It has 16 Ports that can be written to from a single Input port, But can access the data from all the ports simultaneously. The ports are connected to binary to seven segment converters to make them easily displayable.

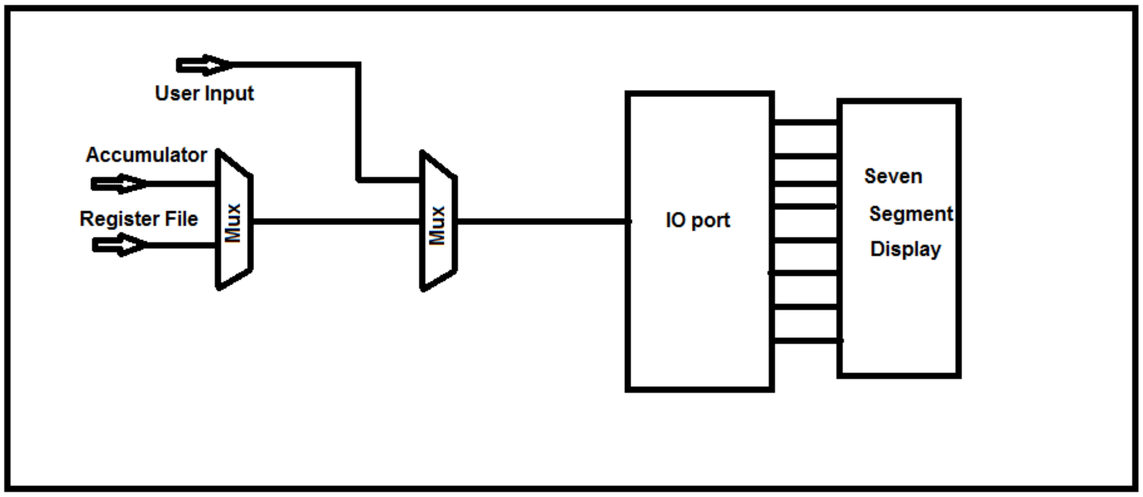


Figure 2.3: Internal Connection of Input Output Port

2.5 The Control Unit

The Control Unit provides the necessary signals for the internal components to let data flow from register to register. It uses The flag register to control some branching operations that depend on flags, for example: the jump on carry instruction uses the carry flag to set the program counter's enable to read the data or not.

2.5.1 Implementation of The Control Unit

Control Units can be implemented using various methods, The one I chose is the micro-program approach, according to Hwang (2005) the control unit is a *huge* Finite State Machine that determines its output based on the current instruction received from The instruction register.In other words, the control unit defines the instructions as states that determine the required micro instruction to be sent to the components. The states may take more than one clock cycle to be executed so a counter keeps track of the current T state of each instruction, For example, *fetch* state takes three T states. The first is that The memory address register reads the content of Program Counter, Second It enables the RAM and asserts the read signal for the RAM to display the memory pointed by MAR, Instruction Register reads the ram output, mean while Program Counter is incremented too, two operations happen in this T state because

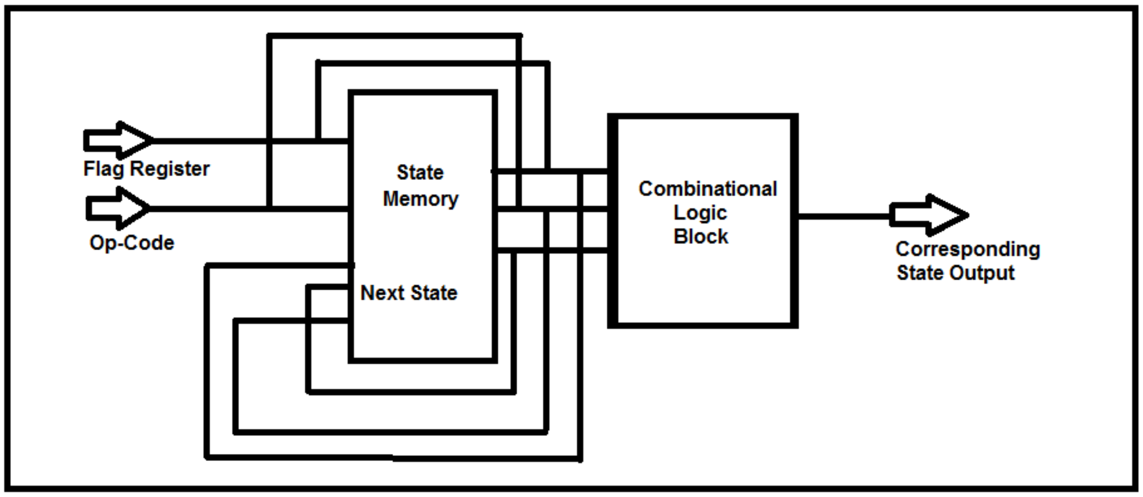


Figure 2.4: Internal Connection of Control Unit

they do not clash together, Finally it asserts the read signal for Instruction register to display for the control unit to determine the next state.

2.6 Timing Issues and meta stability

Since the Control Unit directs the components for the desired operation. It is important to respect the set up and hold timing of the controlled modules. therefore all the components are made to function on the *positive* edge of the clock, and the Control Unit operates on the *negative* edge of the clock. This ensures that timing requirements are respected for all the modules that depend on the clock. The data should be present before the negative edge so that when Control Unit asserts the signals, data will not be wrong.
Another issue is that the Micro-instructions must be set so that when two or more operations are executed on the same clock cycle, they should not read from and write to the same register at the same time. For example, incrementing the Program Counter and writing it on the same clock cycle, another rule is to not write to a register that is reading from another at the **same time**.An example is writing the Index Register to MAR when Index Register is reading the content of RAM, lastly, A register shouldn't write to a register that is writing to another register at the same time.For example RAM writing to the accumulator that is writing at the register file at the same clock cycle.
Those issues produce *meta stability* where the signal is not stable for enough time for registers to interpret it as a logic high or low,this will affect the data causing corruption and problems executing the desired operations.

2.7 Interrupts

Interrupts in this design are handled by the pooling method, a signal inside the control unit is set when the "ei" instruction is executed,it is a two byte instruction that specifies the Interrupt Sub-Routine and sets the interrupt enable so that when the interrupt signal is set and the current instruction is finished the next state will be interrupt instead of fetch, the content of the program counter is pushed to stack and the interrupt sub routine is loaded to the program counter, the interrupt enable is reset so the user must re-enable the interrupt again with the "ei" instruction when need, another way to reset interrupt enable is to use "di" instruction that stands for disable interrupt.
It is recommended to use the "ret" instruction when using interrupts because it pops the program counter back to the next instruction before the interrupt, it is very useful for the "call" instruction too, which works as the interrupt except it has the subroutine's address as an operand and it works as a normal instruction, no need for interrupts.

Chapter 3

Software System Design

We have discussed in Chapter2 about the hardware that is implemented in my design, and by joining the large modules we obtain a processor that runs very well and can executes a good amount of instructions.
Even though programming in machine code could sound pleasing, setting up programs using machine code can be tedious, and debugging a code written in machine code is a nightmare.Therefore, i decided to add a virtual language for my CPU so programming it gives a good experience. The language should include labels, origins and comments. Using the C++ language I programmed an assembler that takes the path to a text file that contains the program to be assembled, and convert it to machine code, then another program that converts the program to mif file that quartus uses for memory initialization.
The dis assembler takes the file path for the text file that has the hexadecimal data and converts the machine code to assembly
Visit: "github.com/dWALIDb/CPU-implementation-with-VHDL" to have access to the code base for vhdl implementaion, as well as the C++ assembler code.

3.1 The assembler

As discussed earlier the assembler takes two text file paths and converts the input file path to machine codes in hexadecimal. each line contains either a label, an origin or an instruction, this is done by a tokenizer that divides the text into tokens for the other programs to use.

3.1.1 Labels

It is very important for assemblers to be able to label a line and refer to it, it makes the code cleaner and easier to read and debug.The syntax for the label is:

$$.label\ name:$$

It is important to parse all labels before the tokenizer starts taking op-codes, therefore, the text is parsed first to get all labels and their corresponding addresses in decimal then we start to get all instructions.

3.1.2 Origins

Depending on the user's applications certain subroutines must be written in certain memory locations, origins enable the user to set the memory location that the program is written into.The syntax is as follows:

`&` *address in decimal*

the '`&`' in C++ stands for 'reference of' meaning the address of the object it is used on, so i used it to assert desired memory location that sub routine starts from.

3.1.3 Comments

Some users might want to explain the purpose of an operation, or different steps that are taken in the program, this is implemented simply with the '`#`' so what ever is to the right of the symbol is not considered.

`#`*this is commented :)*

3.1.4 The Mnemonics

As stated earlier, each line contains one instruction, so the tokenizer starts from the start of the line until it finds a white space, then the op-code is fetched and it will be compared with elements in an unordered map, since we want to determine whether the instruction exists in design or not, if not and error is reported and assembler halts with a failure message, otherwise we check for the number of bytes of the instruction then the tokenizer fetches for its operand if they exist.

- Single Byte instructions: no opcode is present so the instruction is converted to machine code then written into the output file.

- Two Byte instruction: most instructions are two bytes and their operands could be of different addressing modes:
  Immediate Addressing mode: the operand is fetched in decimal and converted to hexadecimal to be written to the file along with its opcode.

  Syntax:opcode value in decimal

  Register Addressing mode: Registers can be referenced to manipulate data.

  Syntax:opcode <register reference>

For instruction with io port they have the same syntax as registers

Syntax:opcode <io port reference>

Absolute/relative Jumps: this addressing mode is used for referencing memory locations too, for example loading from memory, the memory location should be a reference except for relative jumps where the user specifies the offset.

Syntax:opcode (memory location)

Important Note: All memory references must be provided in decimal, the assembler takes care of the conversion.

- Three Byte Instruction: Those instruction may manipulate data from `to` memory and registers.

Syntax:opcode (memory location),<register reference>

3.2 The disassembler

This tool is used to convert from hexadecimal format to the assembly defined in section3.1.4 .The idea is to get the assembly opcode along with the operands, and incrementing a counter accordingly to keep track of which value is an opcode or operand, this tool takes a text file that has the machine code written in hexadecimal and prints the program in hexadecimal.

Chapter 4

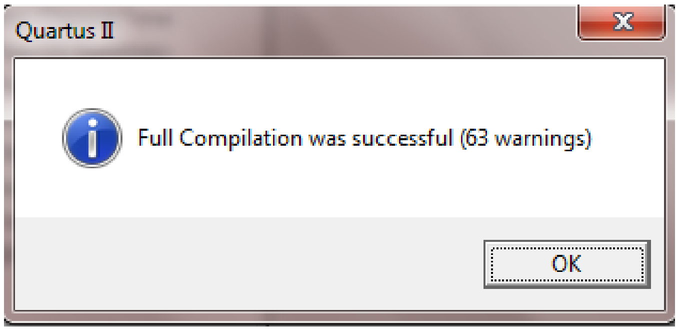
Testing and Verifying

We have discussed in the preceding chapters about the Hardware and Software design of the CPU.In this chapter, the main objective is to demonstrate to display the implementation and to make a tiny application to prove the functionality of the design.

4.1 VHDL Implementation

For the VHDL code every module is written in its own file, a package is used to encapsulate the constants like data bus/address bus width, the used functions or procedures and even user defined types.

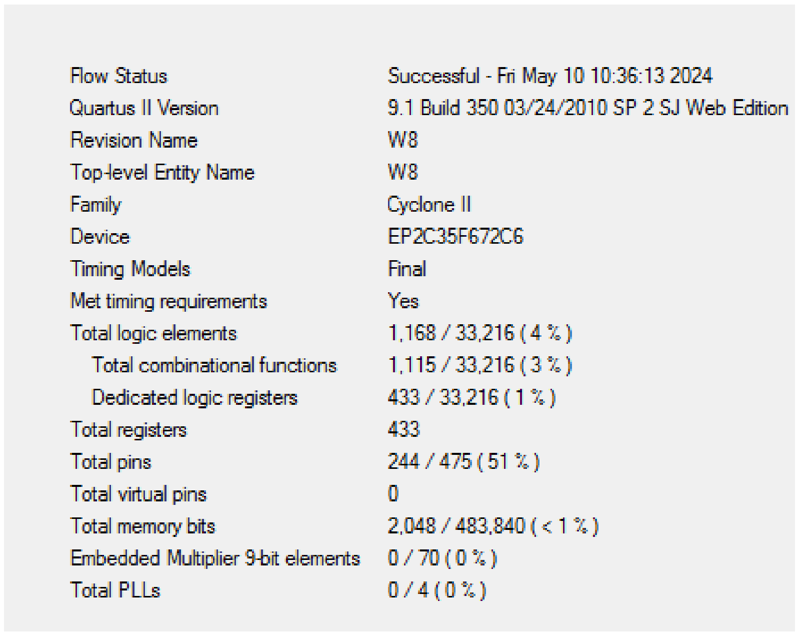
a pop up window shows upon successful compilation:



Full Compilation was successful (63 warnings)

Figure 4.1: Quartus II successful compilation of the design

This proves the validity of the design that I used. The top module is named ”W8”, and the compilation report is as follows:

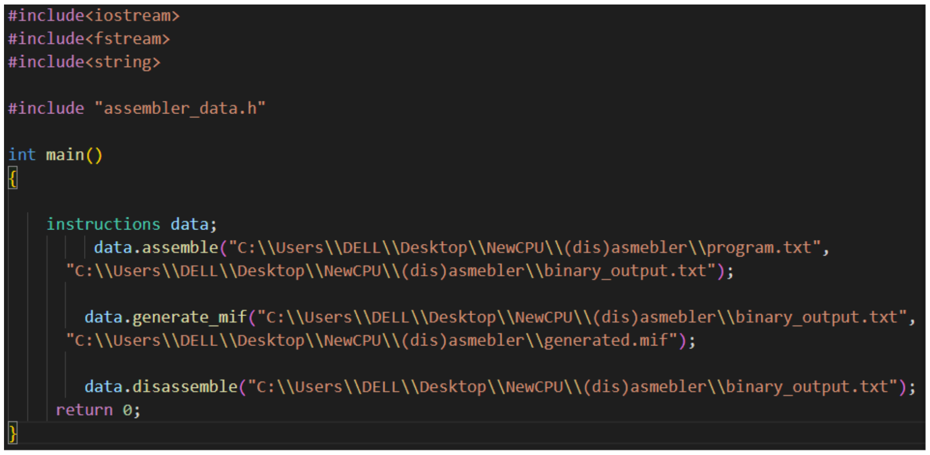


Flow Status	Successful - Fri May 10 10:36:13 2024
Quartus II Version	9.1 Build 350 03/24/2010 SP 2 SJ Web Edition
Revision Name	W8
Top-level Entity Name	W8
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	1,168 / 33,216 (4 %)
Total combinational functions	1,115 / 33,216 (3 %)
Dedicated logic registers	433 / 33,216 (1 %)
Total registers	433
Total pins	244 / 475 (51 %)
Total virtual pins	0
Total memory bits	2,048 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 4.2: Quartus II compilation report for CPU design with VHDL

4.2 The Functionality of the design

The next step is to test the functionality of the CPU by initializing the memory with desired instructions and loading the design on the FPGA to examine the output. The program to be loaded to memory is the **delay program**, it uses lots a variety of instructions such as conditional jumps, register to register data flow, comparison and usage of flag register and the delay could be seen on the output pins. The driving C++ program is as follows:



```
#include<iostream>
#include<fstream>
#include<string>

#include "assembler_data.h"

int main()
{

    instructions data;
         data.assemble("C:\\Users\\DELL\\Desktop\\NewCPU\\(dis)asmebler\\program.txt",
      "C:\\Users\\DELL\\Desktop\\NewCPU\\(dis)asmebler\\binary_output.txt");

        data.generate_mif("C:\\Users\\DELL\\Desktop\\NewCPU\\(dis)asmebler\\binary_output.txt",
      "C:\\Users\\DELL\\Desktop\\NewCPU\\(dis)asmebler\\generated.mif");

        data.disassemble("C:\\Users\\DELL\\Desktop\\NewCPU\\(dis)asmebler\\binary_output.txt");
     return 0;
}
```

Figure 4.3: The driver code for assembler and disassembler

The assembly program and the corresponding MIF file are shown below:

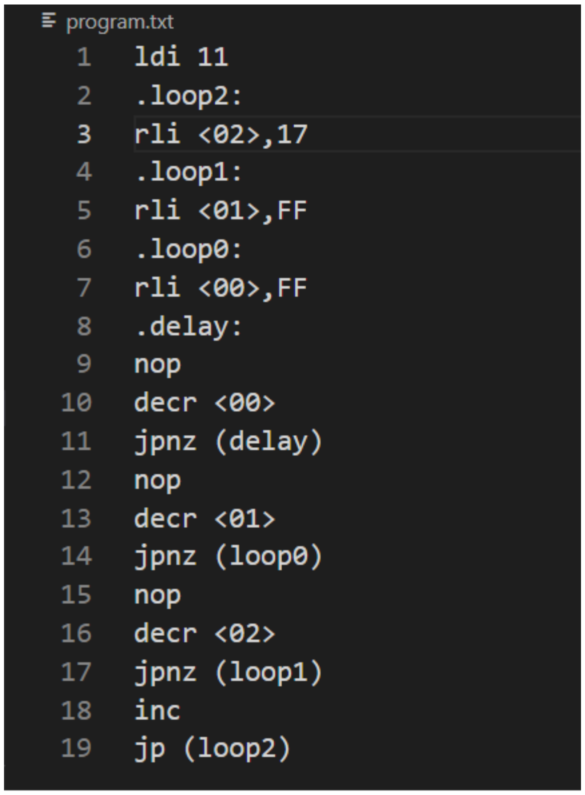
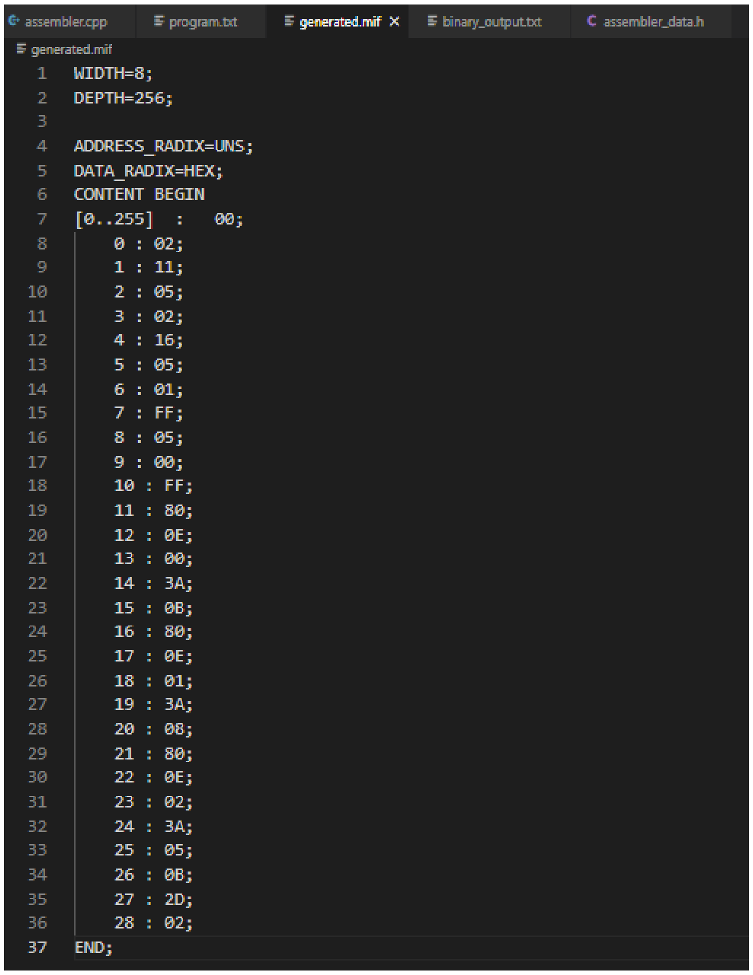

```
program.txt
ldi 11
.loop2:
rli <02>,17
.loop1:
rli <01>,FF
.loop0:
rli <00>,FF
.delay:
nop
decr <00>
jpnz (delay)
nop
decr <01>
jpnz (loop0)
nop
decr <02>
jpnz (loop1)
inc
jp (loop2)
```

Figure 4.4: Assembly code for a counter with 1 second delay

The program initializes three registers with the values necessary for the delay to be 1 second, decrements them and compares them with zero, if the register is equal 0, it is re-initialized and another register is decremented. This happens until the last layer of registers.this is ensured by calculating the number of cycles of each instruction.
One second of delay with a **27MHz clock** means that we need to pass 27 million clock cycles, the "**jpnz**" instruction takes 7 clock cycles, the "**nop**" clock cycle takes 5 cycles and the "**decr**" takes 7 clock cycles to be executed, this makes each loop take 19 clock cycles to be executed.
The inner loop, "**loop0**" is executed 255 times (FF-1 in hexadecimal by excluding 0),this loop is repeated 255 again with the loop "**loop1**" lastly this loop is executed 22 times (17-1 in hexadecimal) which represents the full 1 second. the exact calculation is:

255x255x24x19=27,180,450 clock cycles

this provides a delay of 1.006s with and error of 0.6% which is acceptable error percentage for such application. After successful compilation, the design is loaded to get a binary counter that increments every second. A good test for for the disassembler is to give it the binary file generated by the assembler and look for the generated assembly.
the corresponding mif file and disassembled code are as follows:


assembler.cpp program.txt generated.mif binary_output.txt assembler_data.h

generated.mif

```
WIDTH=8;
DEPTH=256;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;
CONTENT BEGIN
[0..255]  :   00;
    0 : 02;
    1 : 11;
    2 : 05;
    3 : 02;
    4 : 16;
    5 : 05;
    6 : 01;
    7 : FF;
    8 : 05;
    9 : 00;
    10 : FF;
    11 : 80;
    12 : 0E;
    13 : 00;
    14 : 3A;
    15 : 0B;
    16 : 80;
    17 : 0E;
    18 : 01;
    19 : 3A;
    20 : 08;
    21 : 80;
    22 : 0E;
    23 : 02;
    24 : 3A;
    25 : 05;
    26 : 0B;
    27 : 2D;
    28 : 02;
END;
```

Figure 4.5: The memory initialization file for delay program

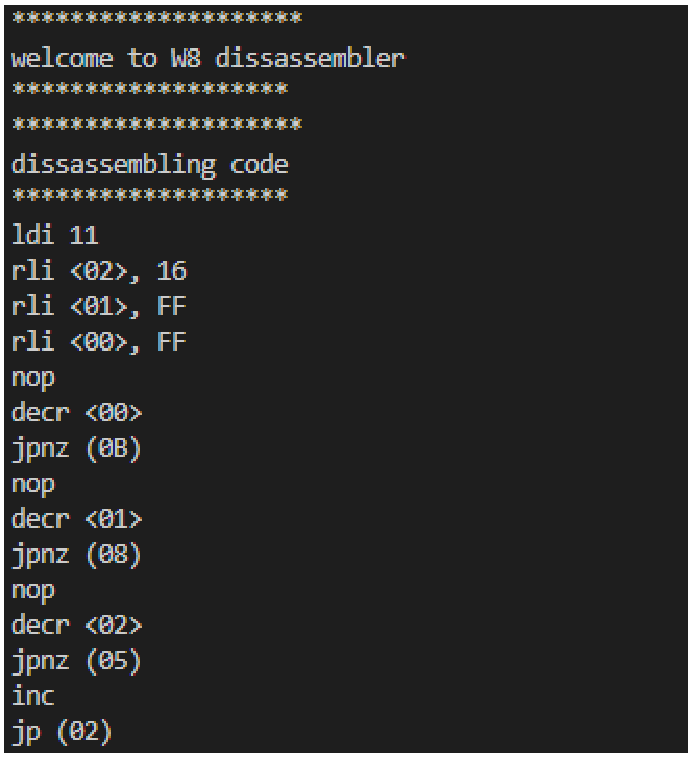

```
********************
welcome to W8 dissassembler
*******************
********************
dissassembling code
*******************
ldi 11
rli <02>, 16
rli <01>, FF
rli <00>, FF
nop
decr <00>
jpnz (0B)
nop
decr <01>
jpnz (08)
nop
decr <02>
jpnz (05)
inc
jp (02)
```

Figure 4.6: The Dissassembled code

Appendix A

The Instruction Set

If you got this far, I would like to thank you for your curiosity and dedication to learn.
In this part I will introduce the various Instructions that my design offers.
For simplicity all the following tables may have the following:
x: An 8-bit value
(x): An 8-bit address, addresses are written in decimal and the assembler takes care of conversion
R: A register reference
acc: The accumulator
I/O: An io port reference reference
F: Flag register
I: Interrupt register
PC: Program Counter
SP: Stack Pointer
IX: Index register
(PC): Memory content pointed by Program Counter
(SP): Top of the stack
(IX): Content of the address pointed by the IX
==: this means chcking for equality.

A.1 Data Flow Control Instructions

Op-code	Machine Code	Description	Clock Cycles
ldd	00	acc=(x)	8
ldr	01	acc=R	7
ldi	02	acc=x	6
rla	03	R=acc	7
rli	05	R=x	8
rlm	06	r=(x)	10
sta	07	(x)=acc	8
str	08	(x)=R	11

Table A.1: Data Flow Instructions

Important Note: Data flow instruction do not affect flag register.

A.2 Stack related Instructions

Op-code	Machine Code	Description	Clock Cycles
ldsp	09	SP=x	7
pushacc	51	(SP)=acc,SP=SP-1	6
popacc	52	SP=SP+1,acc=(SP)	7
pushreg	53	(SP)=R,SP=SP-1	8
popreg	54	SP=SP+1,R=(SP)	8
pushflag	3F	(SP)=F,SP=SP-1	6
popflag	40	SP=SP+1,F=(SP)	7
pushix	6B	(SP)=IX,SP=SP-1	13
popix	6C	SP=SP+1,IX=(SP)	9
call	3C	SP=SP+1,PC=(SP),PC=x	9
ret	3E	SP=SP+1,PC=(SP)	8
di	55	Interrupt disabled	5
ei	56	Interrupt enabled,I=x	6

Table A.2: Stack related instructions

Important Note: stack instruction do not affect flag register,except for pop flag since it retrieves an older value f flag register.

A.3 Index register Instructions

Op-code	Machine Code	Description	Clock Cycles
ldix	0A	IX=x	7
incix	20	IX=IX+1	5
decix	21	IX=IX-1	5
offsetix	6A	IX=IX+x	7
pctoix	69	IX=PC	5
adix	63	acc=acc+(IX)	6
sbix	64	acc=acc-(IX)	6
andix	65	acc=acc and (IX)	6
orix	66	acc=acc or (IX)	6
xorix	67	acc=acc xor (IX)	6
cpix	68	acc==(IX)	6
indexedld	41	acc=(IX)	6
indexedstr	42	(IX)=acc	6

Table A.3: Index register relater instructions

Important Note: "pctoix" instruction it puts the next instruction's address into Index Register, this instruction is useful for avoiding repeating same instructions with different operands.

A.4 Absolute Jump Instructions

Opcode	Machine Code	Description	Clock Cycles
jp	2D	PC=x	7
jpe	2F	PC=x if e=1	7
jpne	35	PC=x if e=0	7
jpl	30	PC=x if l=1	7
jpg	31	PC=x if g=1	7
jpc	37	PC=x if c=1	7
jpnc	38	PC=x if c=0	7
jpz	39	PC=x if z=1	7
jpnz	3A	PC=x if z=0	7
jpp	3B	PC=x if s=1	7
jpn	46	PC=x if s=0	7
jpo	47	PC=x if o=1	7
jpno	48	PC=x if o=0	7

Table A.4: Absolute Jump Instructions

A.5 Relative Jump instructions

Opcode	Machine Code	Description	Clock Cycles
jr	2E	PC=x	7
jre	32	PC=PC+x if e=1	7
jrne	36	PC=PC+x if e=0	7
jrl	33	PC=PC+x if l=1	7
jrg	34	PC=PC+x if g=1	7
jrc	49	PC=PC+x if c=1	7
jrnc	4A	PC=PC+x if c=0	7
jrz	4B	PC=PC+x if z=1	7
jrnz	4C	PC=PC+x if z=0	7
jrp	4D	PC=PC+x if s=1	7
jrn	4E	PC=PC+x if s=0	7
jro	4F	PC=PC+x if o=1	7
jrno	50	PC=PC+x if o=0	7

Table A.5: Relative Jump Instructions

A.6 Immediate Arithmetic and Logic Instructions

Opcode	Machine Code	Description	Clock Cycles
inc	0B	acc=acc+1	6
incc	5B	acc=acc+1 if c=1	6
incz	5F	acc=acc+1 if z=1	6
dec	0D	acc=acc-1	6
decc	5C	acc=acc-1 if c=1	6
decz	60	acc=acc-1 if z=1	6
adi	1A	acc=acc+x	6
sbi	1D	acc=acc-x	6
andi	24	acc=acc and x	6
ori	27	acc=acc or x	6
xori	2A	acc=acc xor x	6
neg	2C	acc= - acc	5
cpl	2B	acc= not acc	5

Table A.6: Immediate Arithmetic and Logic Instructions

A.7 Register Arithmetic and Logic Instructions

Opcode	Machine Code	Description	Clock Cycles
incr	0C	R=R+1	7
incrc	5D	R=R+1 if c=1	7
incrz	61	R=R+1 if z=1	7
decr	0E	R=R-1	7
decrc	5E	R=R-1 if c=1	7
decrz	62	R=R-1 if z=1	7
adr	18	acc=R+x	7
sbr	1B	acc=R-x	7
andr	22	acc=R and x	7
orr	25	acc=R or x	7
xorr	28	acc=R xor x	7
cp	0F	acc==acc	6
cpr	10	acc==R	7

Table A.7: register Arithmetic and Logic Instructions

A.8 Memory Arithmetic and Logic Instructions

Opcode	Machine Code	Description	Clock Cycles
adm	19	acc=acc+(x)	8
sbm	1C	acc=acc-(x)	8
andm	23	acc=acc and (x)	8
orm	26	acc=acc or (x)	8
xorm	29	acc=acc xor (x)	8
cpm	11	acc==(x)	8
sl	12	acc=acc¡¡	5
sr	13	acc=acc¿¿	5
rot	14	acc=acc,c=acc(MSB)	5
slr	15	R=R¡¡	7
srr	16	R=R¿¿	7
rotr	17	R==R(MSB)	7

Table A.8: Memory Arithmetic and Logic Instructions

Important note: The Arithmetic and Logic Instructions affect the flag register according to the output.

A.9 Input Output Instructions

Opcode	Machine Code	Description	Clock Cycles
ina	57	acc=I/O	7
inr	58	R=I/O	7
outa	59	I/O=acc	7
outr	5A	I/O=R	7
get	43	I/O=User Input	7
swp	1E	R(destination)=R(source)	8
nop	80	does nothing	5
halt	FF	halt the cpu	5

Table A.9: Input Output Instructions

Appendix B

Memory Initialization in QuartusII

Memories in QuartusII can be initialized using either HEX files or MIF files.I chose the latter for my initialization, because i had coded a C++ code that generates the file for me. those files have a simple syntax.
First:specify the width and the depth of the memory, using:

WIDTH=*word length;*
DEPTH=*number of words;*

Second:specify the radix of data and address bus:

ADDRESS_RADIX=*radix;*
DATA_RADIX=*radix;*

The radix could be unsigned(UNS), decimal (DEC), binary(BIN) etc...
Third: indicate the beginning of the content using :

CONTENT BEGIN

Fourth:Put your desired content with single locations:

address with appropriate radix:data with appropriate radix;

If there are multiple contiguous locations that have the same value use:

[Starting address..Ending address]:data with appropriate radix;

NOTE:When writing data or addresses beware for your radix to not lose data.
Last:End the File with the Key word:

END;

To initialize the ram, the MIF file could be used in the parameters of the megafunction that instantiates the memory, or it could be written in VHDL by using a synthesis attribute of type string and providing the MIF file path to your memory.

ATTRIBUTE RAM_INIT_FILE : string;
ATTRIBUTE RAM_INIT_FILE of associated type : attribute value;

The associated type is the signal/variable or type that represents the Memory. and the attribute value is the path to the MIF file.

Bibliography

J. Von Neumann, Online http://en. wikipedia. org/wiki/Von_Neumann_architecture **8** (1945).

V. A. Pedroni, *Circuit design and simulation with VHDL* (MIT press, 2010).

E. O. Hwang, *Digital logic and microprocessor design with VHDL* (2005).