8085 processor design and VHDL code

Negin Safari
neginsafari@ut.ac.ir sneginsafari@gmail.com

***Abstract*—This is the report of VHDL implementation of 8085 processor and the design of it. In this project one of the Intel processors named 8085 is designed and implemented using VHDL. RTL design by using VHDL which is a hardware description language is practiced.**

***Keywords*— Intel processor, VHDL.**

I. Introduction

The Intel 8085 ("eighty-eighty-five") is an 8-bit microprocessor produced by Intel and introduced in March 1976. It is software-binary compatible with the more-famous Intel 8080 with only two minor instructions added to support its added interrupt and serial input/output features. However, it requires less support circuitry, allowing simpler and less expensive microcomputer systems to be built. The "5" in the part number highlighted the fact that the 8085 uses a single +5-volt (V) power supply by using depletion-mode transistors, rather than requiring the +5 V, −5 V and +12 V supplies needed by the 8080. This capability matched that of the competing Z80, a popular 8080-derived CPU introduced the year before. These processors could be used in computers running the CP/M operating system. The 8085 is supplied in a 40-pin DIP package. To maximise the functions on the available pins, the 8085 uses a multiplexed address/data (AD0-AD7) bus. However, an 8085 circuit requires an 8-bit address latch, so Intel manufactured several support chips with an address latch built in. These include the 8755, with an address latch, 2 KB of EPROM and 16 I/O pins, and the 8155 with 256 bytes of RAM, 22 I/O pins and a 14-bit programmable timer/counter. The multiplexed address/data bus reduced the number of PCB tracks between the 8085 and such memory and I/O chips.

A. Registers

The processor has seven 8-bit registers accessible to the programmer, named A, B, C, D, E, H, and L, where A is also known as the accumulator. The other six registers can be used as independent byte-registers or as three 16-bit register pairs, BC, DE, and HL (or B, D, H, as referred to in Intel documents), depending on the particular instruction. Some instructions use HL as a (limited) 16-bit accumulator. As in the 8080, the contents of the memory address pointed to by HL can be accessed as pseudo register M. It also has a 16-bit program counter and a 16-bit stack pointer to memory (replacing the 8008's internal stack). Instructions such as PUSH PSW, POP PSW affect the Program Status Word (accumulator and flags). The accumulator stores the results of arithmetic and logical operations, and the flags register bits (sign, zero, auxiliary carry, parity, and carry flags) are set or cleared according to the results of these operations. The sign flag is set if the result has a negative sign (i.e. it is set if bit 7 of the accumulator is set). The auxiliary or half carry flag is set if a carryover from bit 3 to bit 4 occurred. The parity flag is set to 1 if the parity (number of 1-bits) of the accumulator is even; if odd, it is cleared. The zero flag is set if the result of the operation was 0. Lastly, the carry flag is set if a carryover from bit 7 of the accumulator (the MSB) occurred.

Figure 1: 8085's Flag register

In Figure 1 there is a picture of this processor's flags. The order of flags in programmer's view is the same as what is displayed in this Figure.

B. Instructions

As in many other 8-bit processors, all instructions are encoded in a single byte (including register-numbers, but excluding immediate data), for simplicity. Some of them are followed by one or two bytes of data, which can be an immediate operand, a memory address, or a port number. A NOP "no operation" instruction exists, but does not modify any of the registers or flags. Like larger processors, it has CALL and RET instructions for multi-level procedure calls and returns (which can be conditionally executed, like jumps) and instructions to save and restore any 16-bit register-pair on the machine stack. These are intended to be supplied by external hardware in order to invoke a corresponding interrupt-service routine, but are also often employed as fast system calls.

In this processor some of the instructions are for working with 8-bit registers, while some others are for executing 16-bit operations. Here these two types are discussed in general. All two-operand 8-bit arithmetic and logical (ALU) operations work on the 8-bit accumulator (the A register). For two-operand 8-bit operations, the other operand can be either an immediate value, another 8-bit register, or a memory cell addressed by the 16-bit register pair HL. The only 8-bit ALU operations that can have a destination other than the accumulator are the unary incrementation or decrementation instructions, which can operate on any 8-bit register or on memory addressed by HL, as for two-operand 8-bit operations. Direct copying is supported between any two 8-bit registers and between any 8-bit register and an HL-addressed memory cell, using the MOV instruction. An immediate value can also be moved into any of the foregoing destinations, using the MVI instruction. Due to the regular encoding of the MOV instruction (using nearly a quarter of the entire opcode space) there are redundant codes to copy a register into itself (MOV B, B, for instance), which are of little use, except for delays. However, what would have been a copy from the HL-addressed cell into itself (i.e., MOV M, M) instead encodes the HLT instruction, halting execution until an external reset or unmasked interrupt occurs. Although the 8085 is an 8-

bit processor, it has some 16-bit operations. Any of the three 16-bit register pairs (BC, DE, HL or SP) can be loaded with an immediate 16-bit value (using LXI), incremented or decremented (using INX and DCX), or added to HL (using DAD). LHLD loads HL from directly addressed memory and SHLD stores HL likewise. The XCHG operation exchanges the values of HL and DE. Adding HL to itself performs a 16-bit arithmetical left shift with one instruction. The only 16-bit instruction that affects any flag is DAD (adding BC, DE, HL, or SP, to HL), which updates the carry flag to facilitate 24-bit or larger additions and left shifts. Adding the stack pointer to HL is useful for indexing variables in (recursive) stack frames. A stack frame can be allocated using DAD SP and SPHL, and a branch to a computed pointer can be done with PCHL. These abilities make it feasible to compile languages such as PL/M, Pascal, or C with 16-bit variables and produce 8085 machine code. Subtraction and bitwise logical operations on 16 bits is done in 8-bit steps. Operations that have to be implemented by program code (subroutine libraries) include comparisons of signed integers as well as multiplication and division.

II. Design Phase

In this part the proposed design will be discussed. This design is made of two parts, datapath and controller. These two main parts would work as 8085 after connecting together. In Figure 2 the schematic of this datapath is illustrated.

In this datapath, there are some main parts. One of them is Register array which has a complex structure made of registers and multiplexers, and an incrementer.

This part contains all the registers namely B, C, D, E, H, L, SP, and PC. This module can write a specific register or even put the data of it on internal bus or put it on the address output of this module.

There is an internal bus that all of the modules can put data on it through a tristate. It drives most of the registers in here. There are two buffers in this datapath. One of them only holds the address and the other one has the ability to hold data or address according to the machine cycle that it is working in. There is a register for saving the opcode of instruction in order to extract the important parts from it in controller. The data of instruction register is provided through the 8-bit internal bus which is connected to the input data of the processor by using some tri states.

There is a temporary register which is derived by the internal bus. This register is different from the other register, because it works with the negative edge of the clock. This means that if the load of this register was issued, on the negative edge of the clock, the input data will appear on the output of this register. This feature was needed for implementing the arithmetic operations, as their operation had to be done in one cycle. In this cycle we need to put the data of source on the internal bus while the clock has a high value and on the negative edge of clock, this data will be saved in temp register. After that while the clock has a low value the arithmetic operation will be done and the data will appear on the output of ALU and we need to put this data on internal bus. By issuing the load signal of its destination (assuming it was a register) we can save it in the proper place.

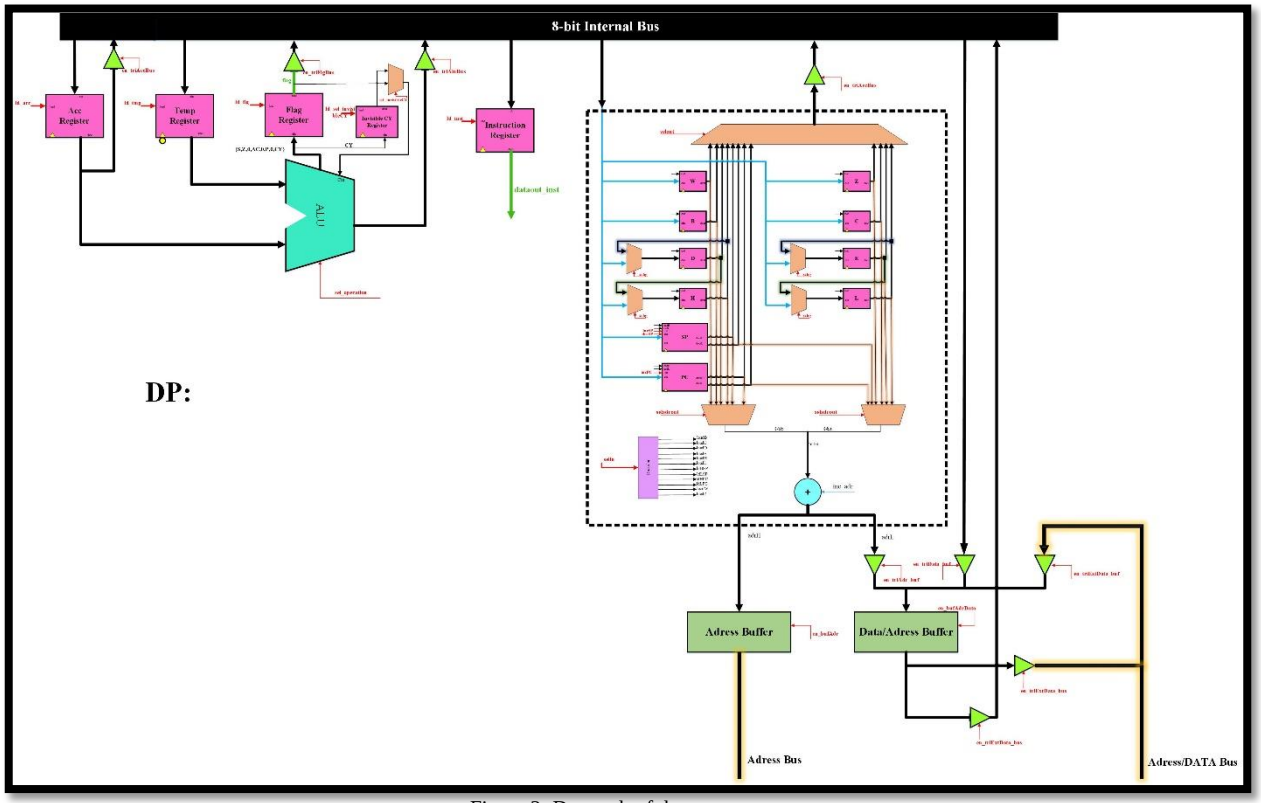


Figure 2: Datapath of the processor

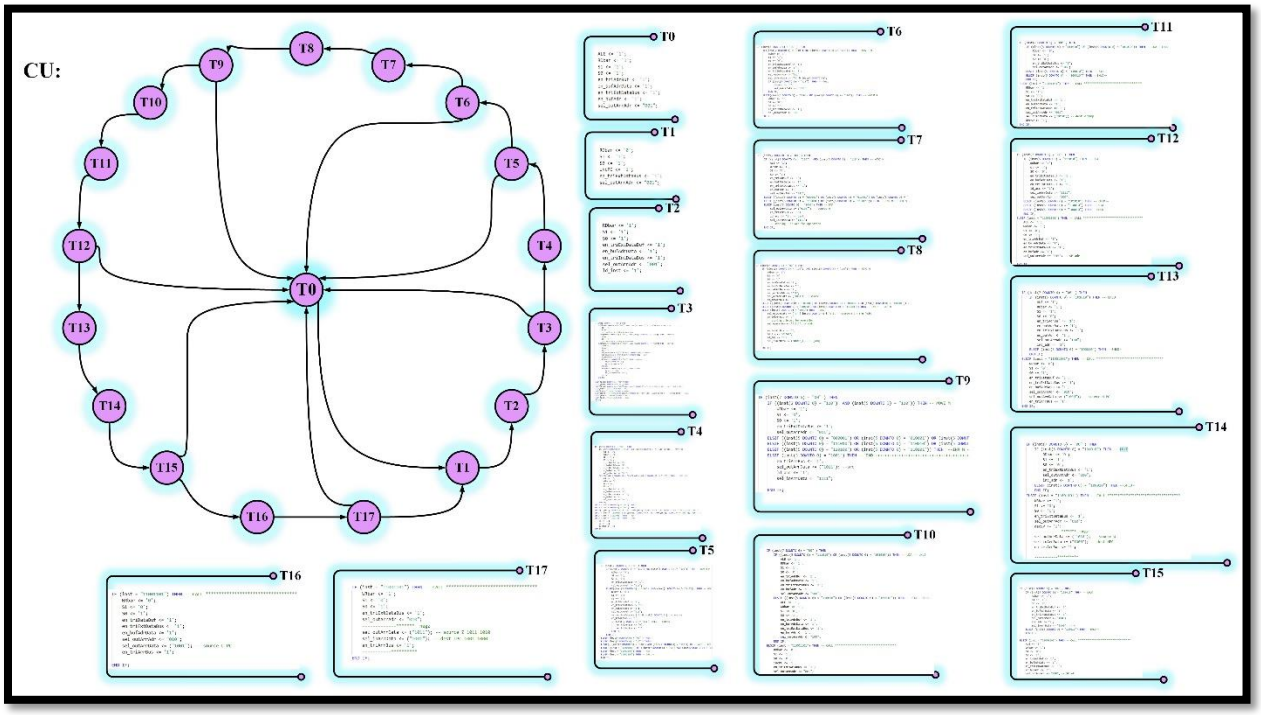


Figure 3: Controller of the processor

The Accmulator register is mainly for storing the result of the arithmetic operation in some instructions. We can also act use this register as a normal one and load it or save it to memory.

Previously, Flag register was discussed in this report. We can see that an 8-bit register is provided for it in datapath. After the execution of some instruction that has something to do with the ALU, this register can be updated. The load signal of each flag is issued in the controller according to the instruction that is being executed and the cycle of execution. One important point is the separated register for invisible Carry flag. In instructions like DAD, INX, and DCX we need to have access to this flag and update it but the point is that the Carry flag register doesn't have to be updated in these instructions. In these instructions the operation must be executed on 16-bit data. This means that we have to do it once for the 8 least significant bits and then according to the carry flag do another operation on the 8 most significant bits.

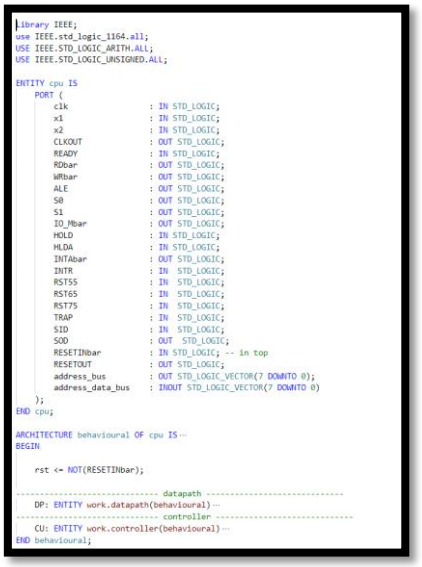
The ALU is the other part which is fully combinational and is used for executing arithmetic operations. It has a sel_operation input signal which indicates what to do with the input of this module.

There is another important part in this design that issues all the control signals. The name of this module is control unit. In Figure 3 there is a brief schematic of the diagram of this finite state machine. The flower shaped schematic is the state machine and it shows the flow of execution. On the right side of this picture there is a brief of signals that should be issued in each state.

As we know this processor is a CISC processor. This means that the size of the instructions are not the same. Due to this reason the states show the number of cycles for each instruction. In each state according to the opcode of instruction one of the else-if statements will be true and right signals will be issued. For more information about each instruction and the method of execution, the main controller code is the best source. According having so many instructions with so many signal issuing we are not able to discuss them all.

The corresponding VHDL codes of both controller and datapath is provided. After this long coding journey. These two modules are connected to each other in a top module named cpu. In Figure 4 the code of this top module is attached.As you can see in Figure 4, we were not able to have all of the code as it was too much.

Another important thing which is needed for simulating our code is the memory. The code of this memory is very similar to the provided memory code for AFTAB processor. In this code we should read an input file to initialize the memory. This initializing file contains the instructions and the data for this processor. We also need to know what will be saved in the memory afterwards. For this purpose in the end of simulation another text file named mem_res.txt will be made and by checking this file we can know if the action was accomplished correctly or not.



```
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY cpu IS
    PORT (
        clk             : IN STD_LOGIC;
        x1              : IN STD_LOGIC;
        x2              : IN STD_LOGIC;
        CLKOUT          : OUT STD_LOGIC;
        READY           : IN STD_LOGIC;
        RDbar           : OUT STD_LOGIC;
        WRbar           : OUT STD_LOGIC;
        ALE             : OUT STD_LOGIC;
        S0              : OUT STD_LOGIC;
        S1              : OUT STD_LOGIC;
        IO_Mbar         : OUT STD_LOGIC;
        HOLD            : IN STD_LOGIC;
        HLDA            : IN STD_LOGIC;
        INTAbar         : OUT STD_LOGIC;
        INTR            : IN  STD_LOGIC;
        RST55           : IN  STD_LOGIC;
        RST65           : IN  STD_LOGIC;
        RST75           : IN  STD_LOGIC;
        TRAP            : IN  STD_LOGIC;
        SID             : IN  STD_LOGIC;
        SOD             : OUT STD_LOGIC;
        RESETINbar      : IN STD_LOGIC; -- in top
        RESETOUT        : OUT STD_LOGIC;
        address_bus     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        address_data_bus: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END cpu;

ARCHITECTURE behavioural OF cpu IS
BEGIN

    rst <= NOT(RESETINbar);

---------------------------- datapath ----------------------------
    DP: ENTITY work.datapath(behavioural)
---------------------------- controller ----------------------------
    CU: ENTITY work.controller(behavioural)
END behavioural;
```

Figure 4: VHDL code of CPU top module

III. Simulation Phase

After the time that the coding was completed, testing and simulation begun. The whole processor was checked instruction by instruction and after that I made sure that it works well. The file that contains all the instructions that have been checked is a text file that can be found among test files.

Following this, another program should have been organized for the question in this project. The question can be described as below.

Write an assembly code for the following program. Start from the location 0100H in the memory, read its data and multiply the data by 2, and write the result to the same location (0100H). Using loops, repeat this procedure for the next 9 locations of the memory (0100H to 0109H). Then translate your assembly to the machine code. Fill the instruction segment of the memory with the machine code and run the program on the processor designed and justify the obtained results.

What we should do in the first step is providing assembly code. For running this program, we need to have a register as an index for our loop. Register B contains this index and initially it will be loaded with 0x0A and in the end of each iteration this value should be decremented. In the end of the loop we have to check if the value of B is zero or not. If it wasn't zero, the Zero flag would have not been issued and after that we will jump based on the condition of not zero to the beginning of the so called loop. Otherwise, the condition of jump would not be available and we get out of the loop. Before entering the loop the HL register will be initialized with the beginning address which is 0x0100. In this loop, what we do is basically reading a data from the saved address in HL and increment this address in the end of each iteration. Due to not having multiply instruction, we use addition. First we have to load accumulator with zero and then add the read data to it and after that we should add the value of accumulator with itself and in the last step we should send this value back to memory to the corresponding address.

In Figure 5 this assembly code is provided. It is the same as what we have discussed in the previous paragraph.

Program(assembly)

```
                    LXI   HL, 0x0100
                    MVI   B, 0x0A
"BEGIN_LOOP" :      MVI   A, 0x00
                    ADD   M[HL]
                    ADD   A
                    MOV   M[HL], A
                    DCR   B
                    INX   HL
                    JNZ   "BEGIN_LOOP"
```

Figure 5: assembly code of the program

Program(machine code)

```
0x21, 0x00, 0x01
0x06, 0x0A
0x3E, 0x00
0x86
0x87
0x77
0x05
0x23
0xC2, 0x05, 0x00
```

Figure 6: machine code of the program

Now, we are going to take a look at the instructions and the waveforms issued for this program. We should state that address 0x0100 to 0x0109 of memory is initialized with the numbers 0x01 to 0x0A.

In Figure 7 we can see the execution of LXI instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we start reading another byte from the memory which is the second byte of our instruction and this process is the same as what has been said. Finally in T6 the second byte will be on the data bus and by issuing the load signal of L register and allowing the value to come on the internal bus, in the next cycle which is T7 the 8 least significant bits of the immediate data will be stored in register L. Also in T7 we start reading another byte from memory and finally in T9 the second byte will be on the data bus and by issuing the load signal of H register and allowing the value to come on the internal bus, in the next cycle the 8 most significant bits of the immediate data will be stored in register H. So, we can see that the value of HL register pair would be 0x0100 after this instruction.

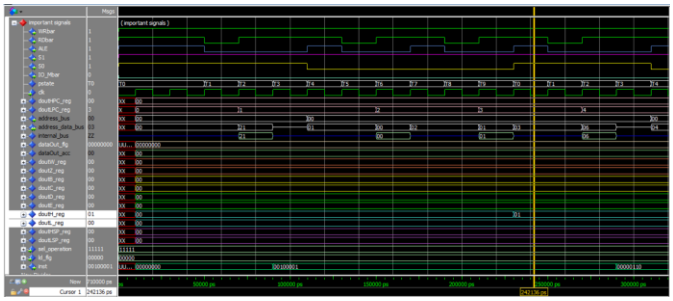


Figure 7: LXI waveforms

In Figure 8 we can see the execution of MVI instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we start reading another byte from the memory which is the second byte of our instruction and this process is the same as what has been said. Finally in T6 the second byte will be on the data bus and by issuing the load signal of B register and allowing the value to come on the internal bus, in the next cycle the immediate data will be stored in register B. So, we can see that the value of register B would be 0x0A after this instruction.

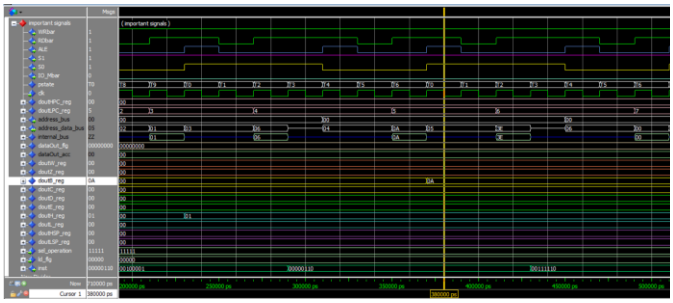


Figure 8: MVI waveforms

In Figure 9 we can see the instruction MVI is being executed and it changes the value of register A as we wanted. After this execution, register A or the accumulator would have 0x00 value stored in it.

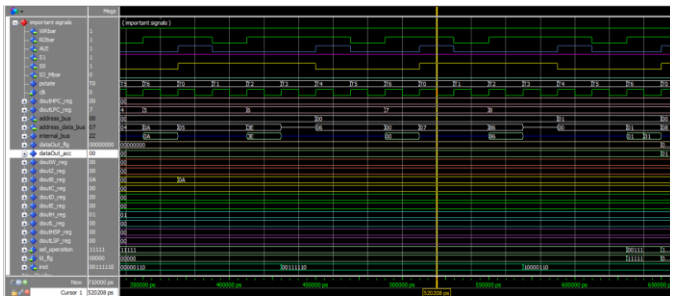


Figure 9: MVI waveforms for A

In Figure 10 we can see the execution of ADD M instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we start reading the data stored in HL address, from the memory. In T4 the address that we want which is HL in here will appear on the address bus and ALE will be issued to show that it is an address. In the next cycle, which is T5, the readbar signal will be issued and finally in T6 the data will be on the data bus and by issuing the load signal of temp register in the first half of the clock cycle and also by allowing the value to come on the internal bus, on the negative edge of the clock this value will be stored in temp register. On the second half of the clock (clk = 0) the addition will be completed and the result data will be appeared on the internal bus. At this moment by deactivating other enable signals, ALU will be the only deriver of the internal bus. In this state the load signal of Accumulator is issued and on the rising edge of the clock, the result will be stored in the Accumulator. In T6 the right sel_operation is selected for executing the addition. So, we can see that the value of register A would be 0x01 after this instruction and the initial data in the so called memory location was 0x01.

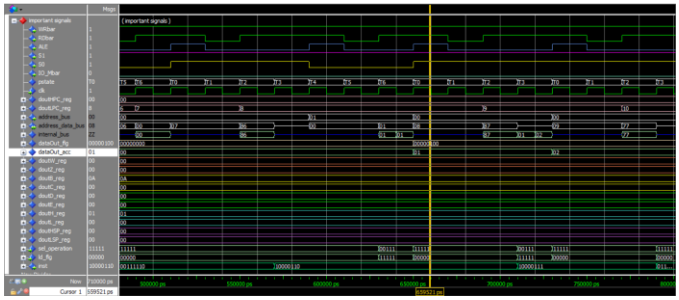


Figure 10: ADD M waveforms

In Figure 11 we can see the execution of ADD A instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we get that the source of this operation is register A and we issue the signals in a way that the only driver of internal bus is register A. By issuing the load signal of temp register in the first half of the clock cycle, on the negative edge of the clock the value of register A will be stored in temp register. On the second half of the clock (clk = 0) the addition will be completed and the result data will be appeared on the internal bus. At this moment by deactivating other enable signals, ALU will be the only deriver of the internal bus. In this

state the load signal of Accumulator is issued and on the rising edge of the clock, the result will be stored in the Accumulator. In T4 the right sel_operation is selected for executing the addition. So, we can see that the value of register A would be 0x02 after this instruction.

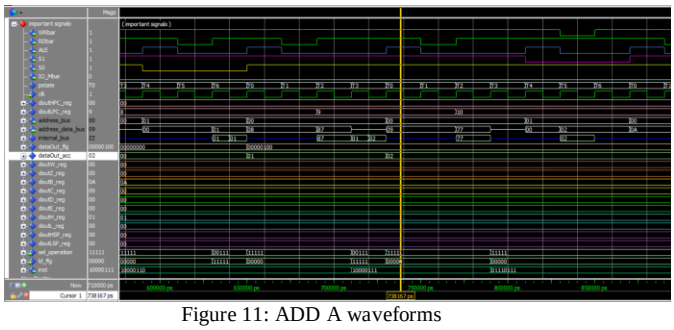


Figure 11: ADD A waveforms

In Figure 12 we can see the execution of MOV M,r instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we know that it is a move instruction and we put the value of HL register as an address on the address bus and we also issue ALE signal to indicate that this is an address. In T5 we get that the source of move instruction is register A and we issue the signals in a way that the only driver of internal bus is register A and then by enabling the corresponding tri state and buffer, we put the value of the data on the address/data bus. In T6 we keep the value of that data and finally in the next cycle the write operation will be completed. So, we can say that the value of Accumulator is now written in the memory and in the address that HL register has stored it.

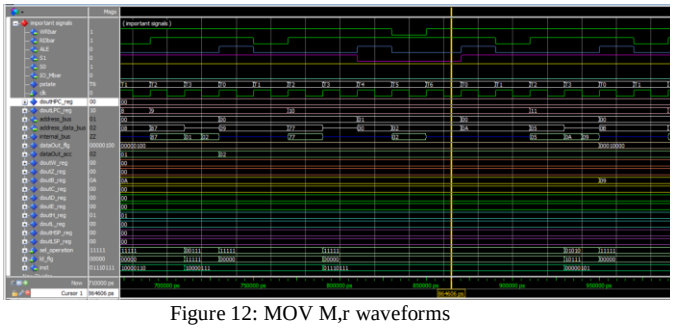


Figure 12: MOV M,r waveforms

In Figure 13 we can see the execution of DCR B instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we get that the source of this operation is register B and we issue the signals in a way that the only driver of internal bus is register B. By issuing the load signal of temp register in the first half of the clock cycle, on the negative edge of the clock the value of register B will be stored in temp register. On the second half of the clock (clk = 0) the decrementation will be completed and the result data will be appeared on the internal bus. At this moment by deactivating other enable signals, ALU will be the only deriver of the internal bus. In this state the load signal of register B is issued and on the rising edge of the clock, the result will be stored in the register B. In T4 the right sel_operation is selected for executing the decrementation. So, we can see that the value of register B would be 0x09 after this instruction.

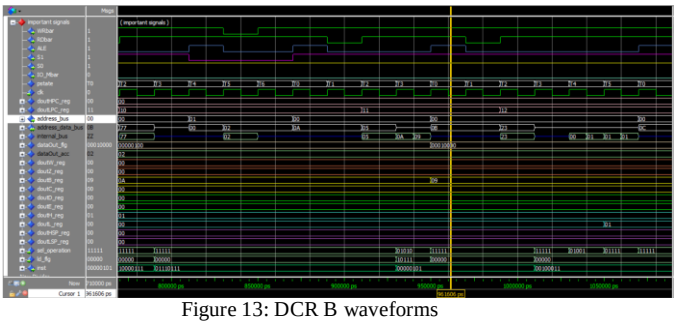


Figure 13: DCR B waveforms

In Figure 14 we can see the execution of INX HL instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we get that for now the source of this operation is register L and we issue the signals in a way that the only driver of internal bus is register L. By issuing the load signal of temp register in the first half of the clock cycle, on the negative edge of the clock the value of register L will be stored in temp register. On the second half of the clock (clk = 0) the decrementation will be completed and the result data will be appeared on the internal bus. At this moment by deactivating other enable signals, ALU will be the only deriver of the internal bus. In this state the load signal of register L is issued and on the rising edge of the clock, the result will be stored in the register L. In T4 the right sel_operation is selected for executing the incrementation and also for this instruction we need to store the carry flag temporary in order to be able to complete this operation on the most significant bits too, but, for this instruction we are not alloed to update the carry flag. That's why we have to issue the load signal of invisible carry register.

In T6 we get that for now the source of this operation is register H and we issue the signals in a way that the only driver of internal bus is register H. By issuing the load signal of temp register in the first half of the clock cycle, on the negative edge

of the clock the value of register H will be stored in temp register. On the second half of the clock (clk = 0) the decrementation based on invisible carry will be completed and the result data will be appeared on the internal bus. At this moment by deactivating other enable signals, ALU will be the only deriver of the internal bus. In this state the load signal of register H is issued and on the rising edge of the clock, the result will be stored in the register H. In T6 the right sel_operation is selected for executing the decrementation based on invisible carry. This means that if the invisible carry register was one, then we will have an incrementation on H register too. So, we can see that the value of register pair HL would be 0x0101 after this instruction.

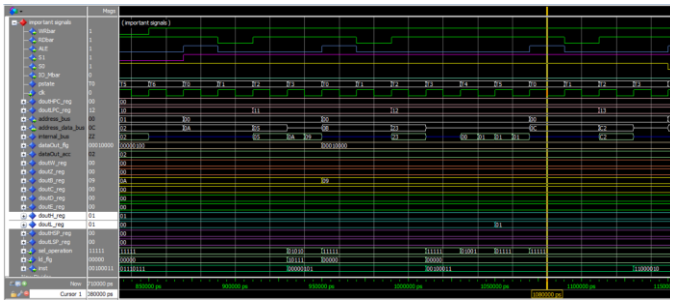


Figure 14: INX HL waveforms

In Figure 15 we can see the execution of JNZ instruction. As you can see in T0 the address saved in PC register will appear on the address bus and the ALE signal will be issued to show it's a read. In T1 the readbar signal will be issued and in next cycle which is T2 the data which is an instruction will appear on the databus and for saving it we will issue the correct signals to put this value on internal bus and issue the load of instruction register. In T3 instruction opcode will be saved in its register. In T4 we start reading another byte from the memory which is the second byte of our instruction and this process is the same as what has been said. Finally in T6 the second byte will be on the data bus and by issuing the load signal of Z register and allowing the value to come on the internal bus, in the next cycle which is T7 the 8 least significant bits of the immediate data will be stored in register Z (this is temporary). Also in T7 we start reading another byte from memory. In T8 as well as reading the memory and issuing the related signals, because now the needed PC for getting the third byte is latched, we can check the needed flag (here Z flag), if it was logical zero, we can load the value of Z register in the 8 least significant bits of the PC register, if not, we won't do that and nothing will be loaded and PC saves its own value. Finally in T9 the second byte will be on the data bus and if the Z flag was logical zero, we can load the read value which is currently on internal bus in the 8 most significant bits of the PC register, if not, we won't do that and nothing will be loaded and PC saves its own value. If PC was loaded with this new immediate value, it means that we have jumped to this new address. As you can see Z flag was not logical one and the jump to address 0x0005 happened.

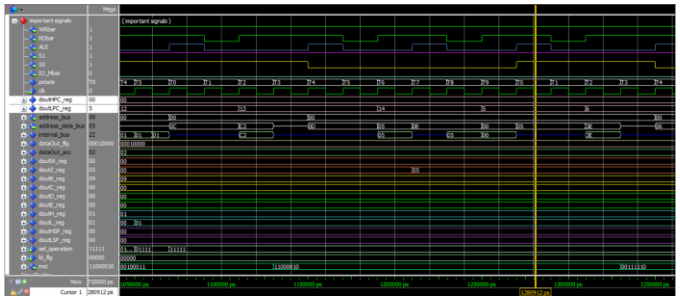


Figure 15: JNZ waveform (jumped)

This whole process will be continued for 10 times in general and the last instruction would be JNZ, but, this time the jump won't happen because the Z flag is logical one. This can be seen in Figure 16.

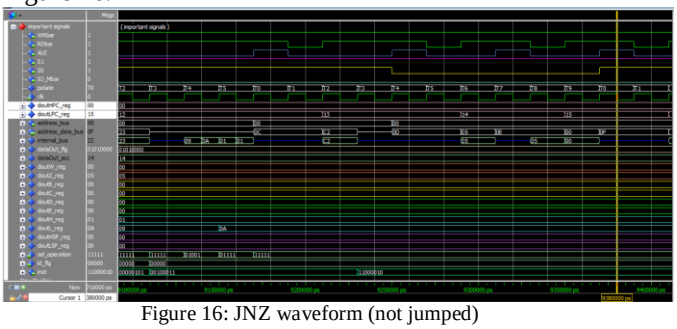


Figure 16: JNZ waveform (not jumped)

In the end, we can check the memory file that has been generated, named mem_res.txt and we get that the data located in 0x0100 to 0x01009 has been doubled.

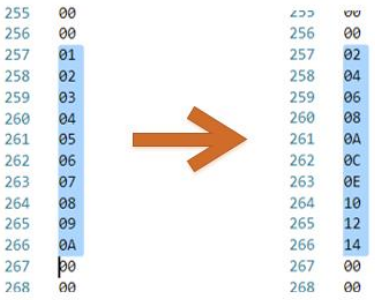


Figure 17: before and after execution

IV. Implementation Phase

The main purpose of this part of the report is just to indicate that this VHDL code is synthesizable. Using Quartus II Synthesis tool the verified codes are synthesized and the selected FPGA for this aim is Cyclone IV GX EP4CGX15BF14A7. In Figure 18 the flow summery of this implementation is provided.

Flow Summary	
Flow Status	Successful - Fri Apr 07 12:13:12 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	8085
Top-level Entity Name	cpu
Family	Cyclone IV GX
Device	EP4CGX15BF14A7
Timing Models	Final
Total logic elements	969 / 14,400 (7 %)
Total combinational functions	959 / 14,400 (7 %)
Dedicated logic registers	143 / 14,400 (< 1 %)
Total registers	143
Total pins	39 / 81 (48 %)
Total virtual pins	0
Total memory bits	0 / 552,960 (0 %)
Embedded Multiplier 9-bit elements	0
Total GXB Receiver Channel PCS	0 / 2 (0 %)
Total GXB Receiver Channel PMA	0 / 2 (0 %)
Total GXB Transmitter Channel PCS	0 / 2 (0 %)
Total GXB Transmitter Channel PMA	0 / 2 (0 %)
Total PLLs	0 / 3 (0 %)

Figure 18. Flow summery of implementation

The maximum frequency of this circuit is 41.38 MHz which is one of the results provided by synthesis tool and its timing analyser. In Figure 19 this result is shown.

Slow 1200mV -40C Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	41.38 MHz	41.38 MHz	clk	

Figure 19. Maximum frequency of Circuit

By using RTL viewer we can get the schematic provided by synthesis tool after analysing our VHDL code. Figure 20 shows the interface between controller and datapath.

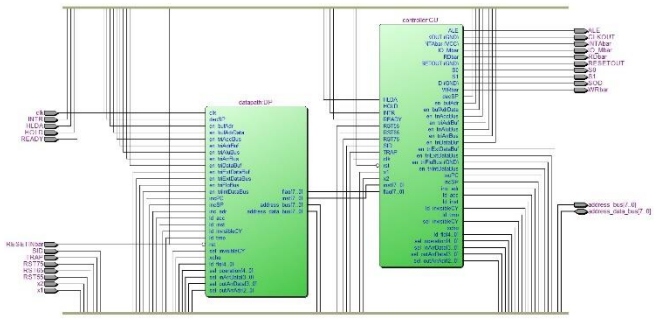


Figure 20. Connection schematic of DP and CU in Quartus

Figure 21 shows the schematic of datapath which is drawn by RTL viewer of synthesis tool after compiling and analysing our VHDL code.

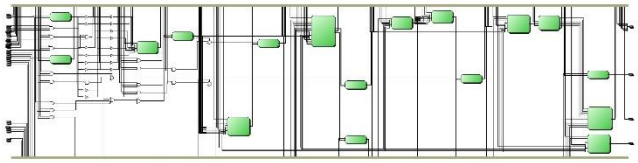


Figure 21. Datapath schematic generated in RTL viewer of Quartus

Figure 22 shows the schematic of control unit which is drawn by RTL viewer of synthesis tool after compiling and analysing our VHDL code.

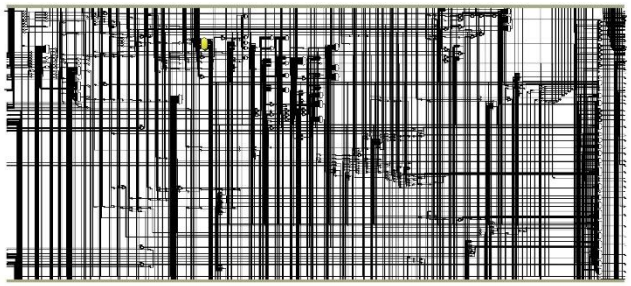


Figure 22. Controller schematic generated in RTL viewer of Quartus

V. Conclusion

In this computer assignment a processor named 8085 is implemented. This processor is modelled using VHDL and it is verified using VHDL test bench. After testing we concluded that this design can handle its input programs and it works without any problems.

References

[1] Introduction au language VHDL (developpez.com)
[2] Data Types in VHDL (technobyte.org)
[3] Intel 8085 - Wikipedia