Designing a CISC CPU

Goina Radu
group: 30432

Contents

Chapter 1

Introduction

1.1 Proposal

The goal of this project is to design and implement a 16 bit CISC processor capable of executing a simple program that is stored within the CPU's memory. This CPU processor will be able to run 20 different instructions and use three addressing modes (register, immediate, indirect).

1.2 Specifications

The CPU will be simulated in Vivado and will run a program that is stored in the instruction memory. The program execution will be tracked using the waveform generated by Vivado and the result of the program will be visible in the Vivado Objects window.

Chapter 2

Bibliographic study

CISC stands for complex instruction set architecture is an computer architecture in which a single instruction is capable of executing several low level operations or capable of multi-step operations or addressing modes. [4]

My CPU will be inspired by the 8086 processor architecture, at least from the instruction execution point of view. The internal architecture of the 8086 processor is divided into two 2 units: The Bus Interface Unit (BIU) and The Execution Unit (EU).

The Bus Interface Unit (BIU) it's a component that provides the interface to memory and I/O operations. It has a couple of useful functions: it generates physical addresses for memory access, it fetches instructions from memory and transfers data between the EU and memory and I/O, maintains 6-byte pre-fetch instruction queue (for pipelining). But my CPU will not have I/O components or have to deal with pipelining, therefore this component will be completely changed to better align with the requirements of my system.

The Execution Unit (EU) it's the component that decodes and executes instructions. It contains the general purpose registers, the ALU, Special purpose registers, Instruction register and Instruction Decoder, and the Flag Register. The changes made to this unit and the components will be described in detail in the Design chapter. [1]

Micro-programmed Control Unit is a control unit that uses a microcode to execute instructions. The microcode is a set of instructions that can be modified or updated, allowing for greater flexibility and ease of modification. The control signals for each instruction are generated by a microprogram that is stored in memory.

Control Word: A control word is a word whose individual bits represent various control signals.
Micro-routine: A sequence of control words corresponding to the control sequence of a machine instruction constitutes the micro-routine for that instruction Micro-instruction: Individual control words in this micro-routine are referred to as microinstructions[2].

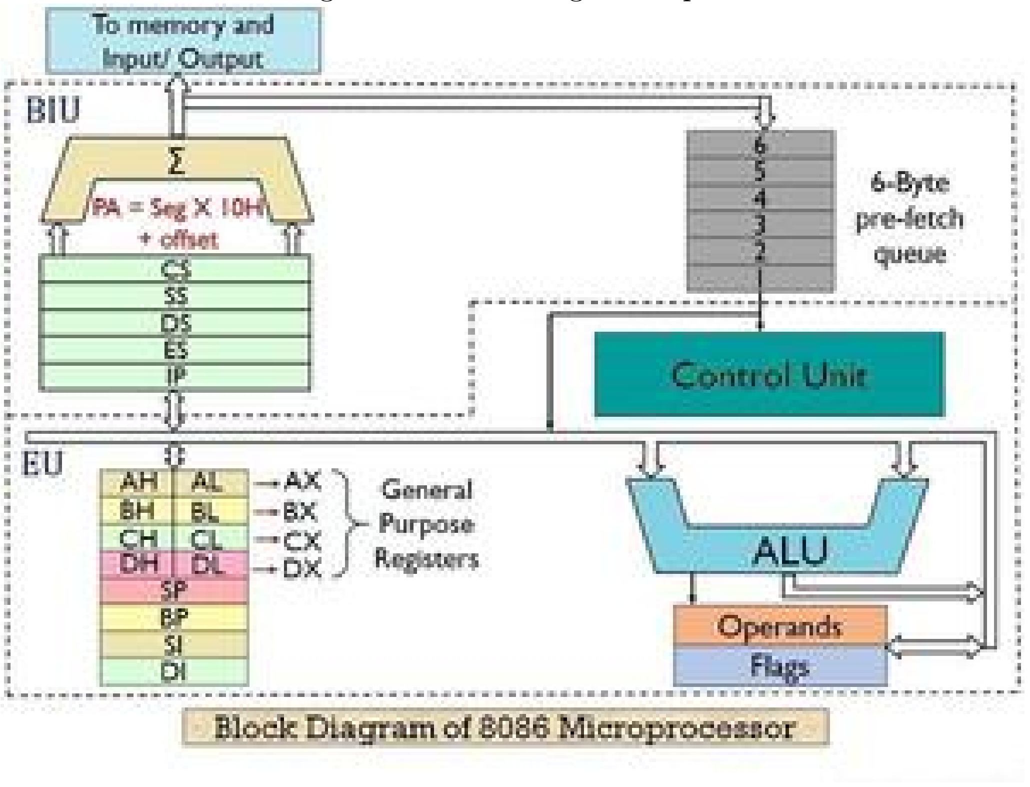
Chapter 3

Design

3.1 Data path

First lets take a look at the 8086 data path

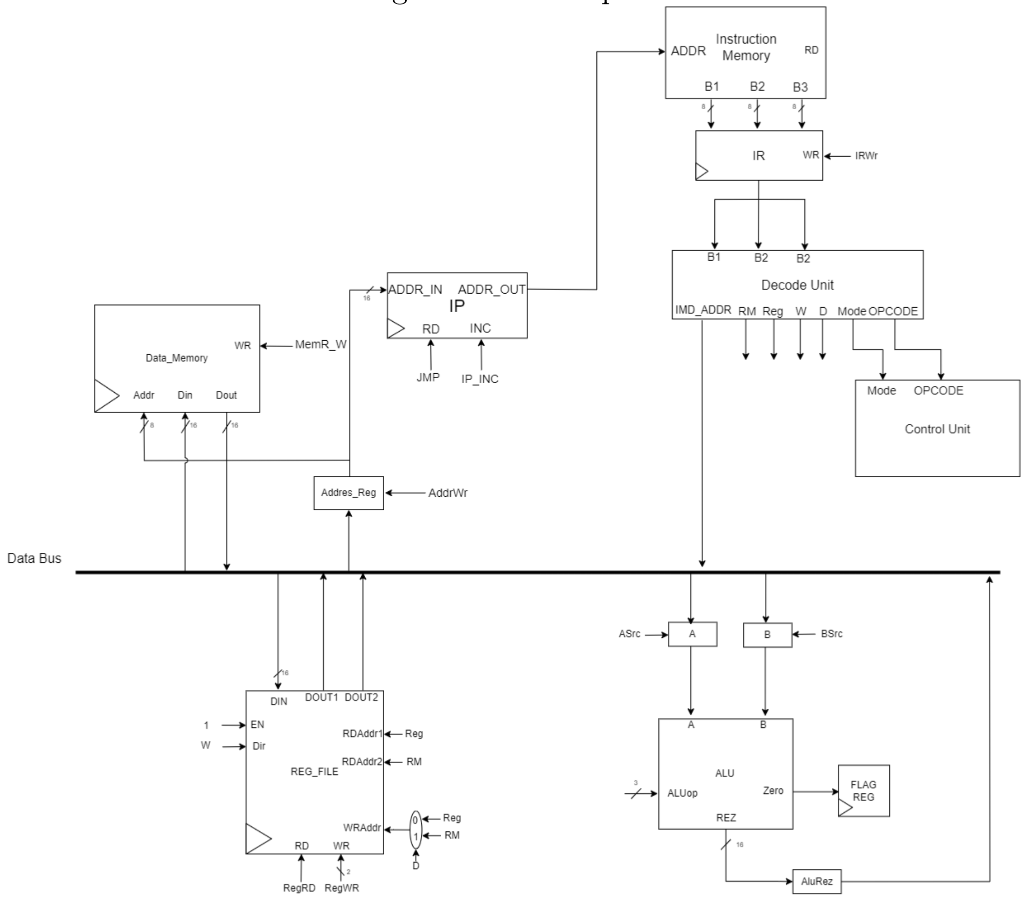
Figure 3.1: 8086 rough data path



We can see that the 8086 processor is processor is composed of two units: the BIU and the EU.
My processor is not pipelined, won't communicate with input/output component and the memory will be handled differently from the 8086 processor.

Therefore in my design I removed the entire BIU and and made slight modifications to the EU to also include the memory elements.

Figure 3.2: Data path

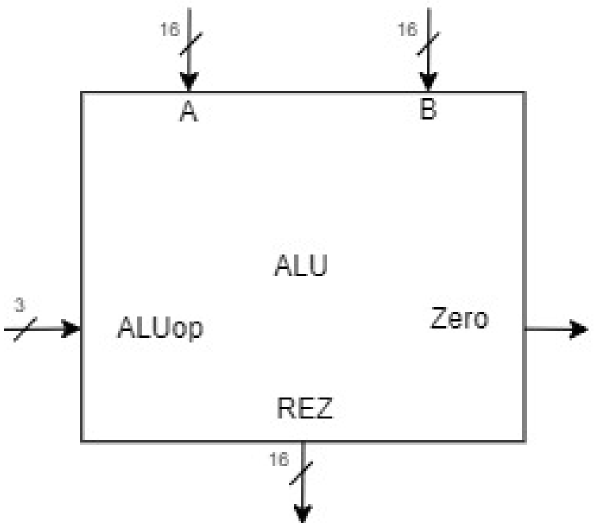


3.2 Data path components

3.2.1 ALU

Executes arithmetic and logic operations and sets the flags accordingly.
The operations that the alu is able to perform are the following: inc, dec, add, sub, and, or, xor, pass A

Figure 3.3: ALU



A,B - input data ALUop - control signal that chooses the operation
REZ - the result of the operation
Zero - '1' if the result is 0

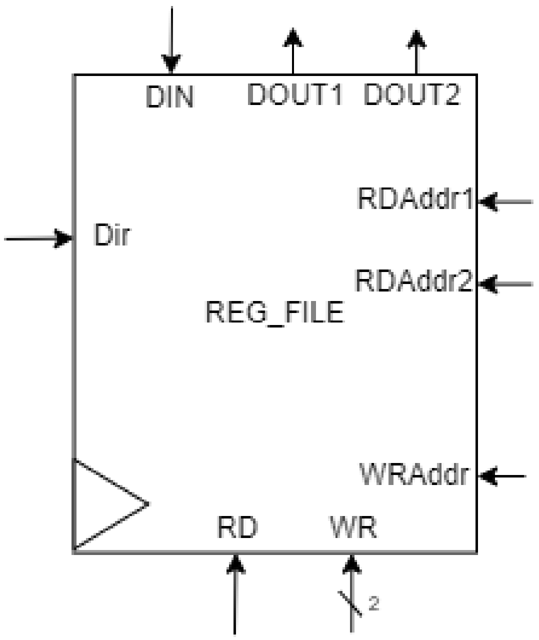
3.2.2 Register File

My processor's equivalent of the 8086 general purpose registers. In my CPU I have no need for special registers therefore each register behaves the same and can be used in any context.

It has asynchronous reads and synchronous writes

It is important to keep in mind that my processor, unlike the 8086, does not support byte addressing although the register file supports byte addressing.

Figure 3.4: Register File

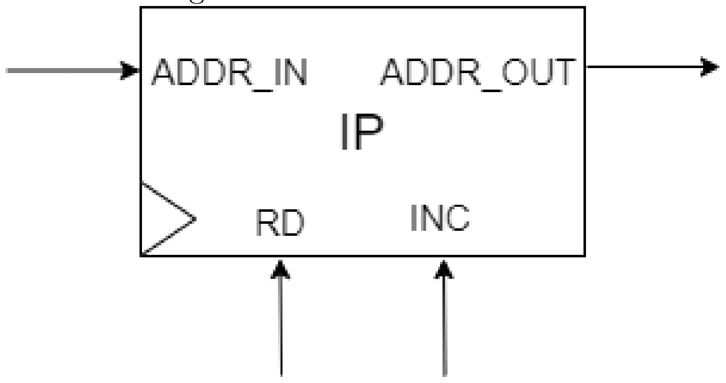


DIN - data to be written in the register file
WRAddr1 - select register where to write data
RDAddr1, RDAddr2 - select the registers where to read data
DOUT1, DOUT2 - data read from the registers
WR - selects between writeing a byte or a word
RD - selects between reading a byte or reading a word
Dir - selects the high or low byte

3.2.3 Instruction pointer

Register/counter that points to the next instruction in the instruction memory that will be executed

Figure 3.5: Instruction Pointer

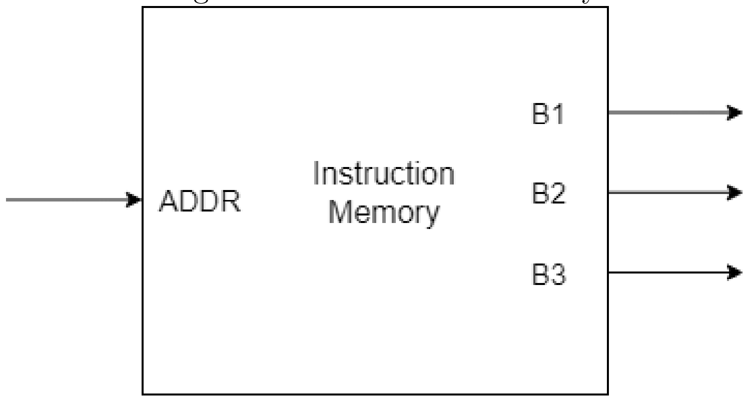


ADDR_IN - address that will be written if RD is '1'
ADDR_OUT- address that it the IP will point to
RD - signal for writing an address
INC - signal that increments the address

3.2.4 Instruction memory

512x24 ROM that holds the instructions of the program that the CPU will execute. It has 23 data bits because an instruction be up to 3 bytes in size.

Figure 3.6: Instruction Memory



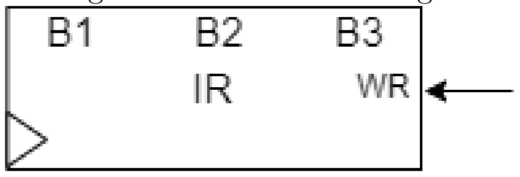
ADDR - address of the instruction

B1, B2, B3 - instruction bytes, B1 is the most significant byte

3.2.5 Instruction register

Register that hold the instruction that is currently being executed

Figure 3.7: Instruction Register

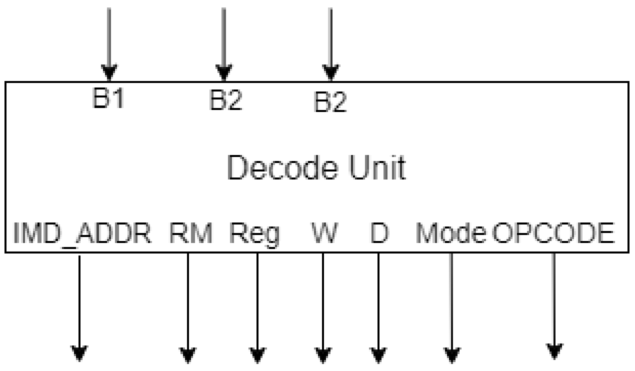


B1, B2, B3 - that that will be written in the register
WR - signal that writes the data into the register

3.2.6 Decode Unit

Unit that decodes relevant information from the instruction data

Figure 3.8: Instruction Register

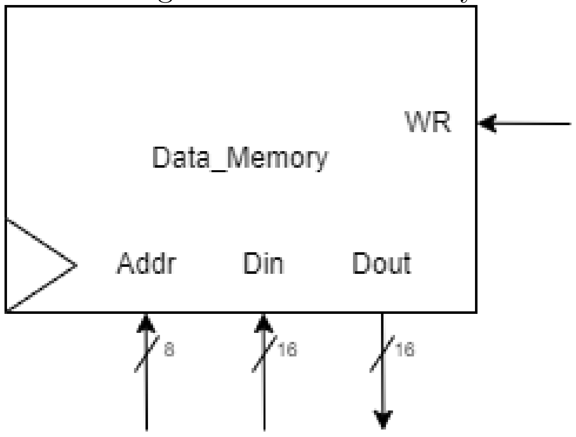


B1, B2, B3 - instruction data
The outputs will be described in detail in the analysis part

3.2.7 Data Memory

The main memory of my CPU has a capacity of 1KB and operates on a word-addressable basis.

Figure 3.9: Data memory

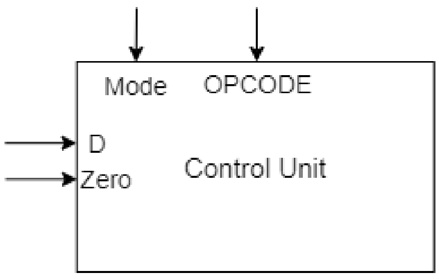


DIN - data input
DOUT - data output
ADDR - address of the word to be read or written
WR - control signal for writing in the data memory

3.2.8 Control Unit

Component that controls the flow of instruction and coordinating the other functional units within the CPU.

Figure 3.10: Control unit



D, MODE, OPCODE - inputs from the instruction

Zero - inputs from the ALU flag register
The outputs of the control unit are the control signals. These will be detailed in the analysis part of the documentation.

3.2.9 Bus controller

This component is not depicted on the data path
Usually in this type of system one would use tri-state buffers to write data on the bus. Despite this I choose to use a multiplexer controlled by the BUS_CTR signal to write on the bus.
I made this design decision because Vivado can't synthesise "true" tri-state buffers as it converts them to multiplexers anyway.
BUS_CTR value:

- 0000 - RF_DOUT1
- 0001 - RF_DOUT2
- 1001 - RF_DOUT1
- 0111 - DM_DOUT
- 0011 - ALU_REZ_REG
- 0101 - IMD_ADDR

Chapter 4

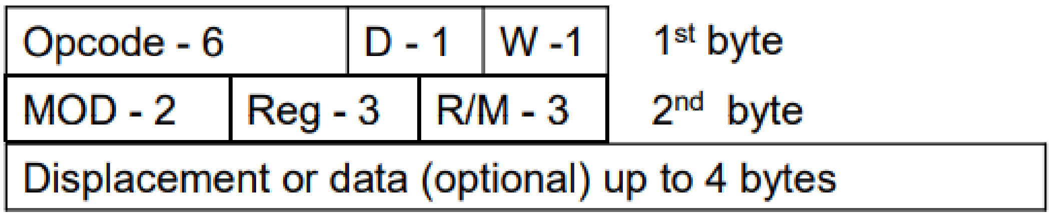
Analysis

4.1 Instruction format

4.1.1 8086 processor

Lets first take a look at the way the 8086 processor encodes data[3]

Figure 4.1: Data encoding



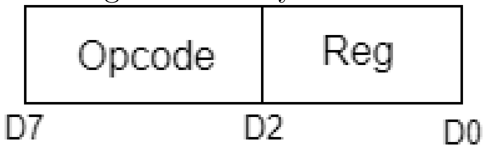
As we can see the the instruction in the 8086 architecture have varying length, between 1 and 4 bytes. Lets see what each byte means
Byte 1
Opcode (6 bit) – specifies the operation that will be done
D (1 bit) – states if the Reg field in byte 2 is destination (‘1’) or source (‘0’)
W (1 bit) – states if the operation will be done on 8-bit data or 16 bit data
Byte 2
MOD (2 bits) - Memory(3 displacement types)/Register mode
Reg (3 bits) - the register field
R/M (3) – the register/memory field
Byte 3-6
These bytes hold immediate data or a displacement for an address

4.1.2 My processor

In my processor I will have a instruction on 1,2 and 3 bytes that will be inspired by the ones in the 8086 processor.

One byte format
Register Addressing

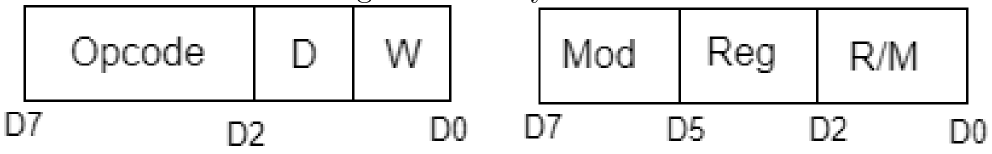
Figure 4.2: 1 byte format



Addresses only one register specified in the REG(3bits) field

Two Byte format
Register Addressing
Register to/from Memory

Figure 4.3: 2 byte format



Mode (2bits) - selects the mode

- 00 Memory addressing mode
- 11 Register addressing mode

D (1bit) - states if the R/M field in byte 2 is destination ('0') or source ('1')
W (1bit) - ignored, my processor does not have byte addressing
Reg (3bits) - register
R/M (3its) - register/memory field

Three Byte format
Immediate Operand to Register
Jump type

Figure 4.4: Immediate to Register format

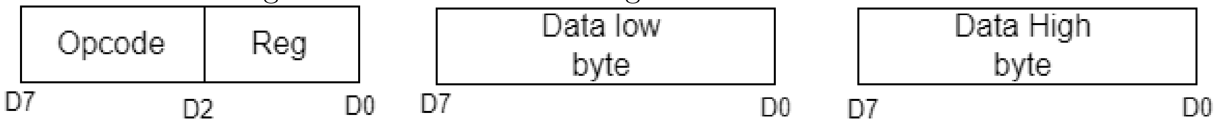
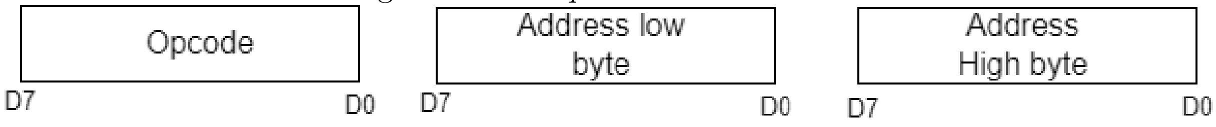


Figure 4.5: Jump format



4.2 Supported instruction

These are the instruction that are implemented in my CPU and the addressing modes they support

- INC
    1. Register Addressing
- DEC
    1. Register Addressing
- ADD
    1. Register to Register
    2. Immediate Operand to Register
- SUB
    1. Register to Register

    2. Immediate Operand to Register

- AND
    1. Register to Register
    2. Immediate Operand to Register
- OR
    1. Register to Register
    2. Immediate Operand to Register
- TEST
    1. Register to Register
    2. Immediate Operand to Register
- XOR
    1. Register to Register
    2. Immediate Operand to Register
- MOV
    1. Register to Register
    2. Register to/from Memory
    3. Immediate Operand to Register

These are the opcodes for each instruction
00000 - inc
00001 - decrement
00010 - mov REG, REG / mov REG, [REG]
00011 - add REG, REG
00100 - sub REG, REG
00101 - and REG, REG
00110 - or REG, REG
00111 - xor REG, REG
01000 - test REG, REG
01001 - JZ imd
01010 - JNZ imd
01011 - jump imd
01100 - add REG, imd

01101 - mov REG, imd
01110 - mov REG , [imd]
01111 - sub REG, imd
10000 - and REG, imd
10001 - or REG, imd
10010 - xor REG, imd
10011 - test REG, imd

4.3 Control unit

4.3.1 Control unit signals

In this section i will detail what each control signal does

JMP - instruction pointer(IP)gets the value held by the Address_Register
PCwr - unused*
IP_INC - increments the value in the instruction pointer(IP)
MemR_W - write signal of the Data_Memory
RFDate - unused*
RegRD - Reg_File RD signal
RegWR - Reg_File WR signal
Asrc - write signal for the A reg (Alu reg)
BSrc - write signal for the B reg (Alu reg)
AluOP - operation select for the ALU
Bus_Ctr - control signal that determines which component writes on the buss
AddrWr - write data to Address_Register
flagWR - write flags in the flag register
*values that at some point were used but were replaced with other signals

4.3.2 Control unit details

The control unit emulates a micro-programmed control unit. Emulates, because it is made using one big process where its functionality is described.

Much like a regular micro-programmed control unit it stores the control signals in Control Words and generates the next micro-instruction based on the opcode, address in the control word and other signals coming from external units.

The Control Words are described in the following table

Addr	NextAddr	JMp	PCwr	IP_INC	MemR_W	RFDate	RegRD	RegWR	Asrc	BSrc	AluOP	Bus_Ctr	AddrWr	flagWR
0	11110	0	0	0	0	0	0	00	1	0	000	0000	0	1
1	11110	0	0	0	0	0	0	00	1	0	001	0000	0	1
2	mode	0	0	0	0	0	0	00	1	0	111	0000	0	0
3	11110	0	0	0	0	0	0	00	0	1	010	0001	0	1
4	11110	0	0	0	0	0	0	00	0	1	011	0001	0	1
5	11110	0	0	0	0	0	0	00	0	1	100	0001	0	1
6	11110	0	0	0	0	0	0	00	0	1	101	0001	0	1
7	11110	0	0	0	0	0	0	00	0	1	110	0001	0	1
8	11111	0	0	0	0	0	0	00	0	1	100	0001	0	1
9	zero	0	0	0	0	0	0	00	0	0	000	0000	0	0
10	zero	0	0	0	0	0	0	00	0	0	000	0000	0	0
11	11001	0	0	0	0	0	0	00	0	0	000	0101	1	0
12	11110	0	0	0	0	0	0	00	0	1	010	0001	0	1
13	11110	0	0	0	0	0	0	00	1	0	111	0101	0	0
14	11101	0	0	0	0	0	0	00	0	0	000	0101	1	0
15	11110	0	0	0	0	0	0	00	0	1	011	0001	0	1
16	11110	0	0	0	0	0	0	00	0	1	100	0001	0	1
17	11110	0	0	0	0	0	0	00	0	1	101	0001	0	1
18	11110	0	0	0	0	0	0	00	0	1	110	0001	0	1
19	11111	0	0	0	0	0	0	00	0	1	100	0001	0	1
24	opcode	0	0	0	0	0	0	00	1	0	000	0101	0	0
25	11111	1	1	0	0	0	0	00	0	0	000	0000	0	0
26	opcode	0	0	0	0	0	0	00	1	0	000	1001	0	0
27	D bit	0	0	0	0	0	0	00	0	0	000	0001	1	0
28	11111	0	0	0	1	0	0	00	0	0	000	0000	0	0
29	11111	0	0	0	0	0	0	10	0	0	000	0111	0	0
30	11111	0	0	0	0	0	0	10	0	0	000	0011	0	0
31	opcode	0	1	1	0	0	0	00	0	0	000	0111	0	0

Table 4.1: Microcodes

Addr - the address of the control word in the memory
NextAddr - the address of the next control word in the micro-routine, the locations where the address is not a binary number mean that the next address depends on the control signal in that cell

Control words descripion:

- 0 - execute increment
- 1 - execute decrement
- 2 - mov register/memory begining
- 3 - execute add
- 4 - execute sub
- 5 - execute and
- 6 - execute or
- 7 - execute xor
- 8 - execute test
- 9 - JZ test zero
- 10 - JZN test zero
- 11 - prepare jump address
- 12 - execute add imd
- 13 - execute mov imd
- 14 - execute mov imd address
- 15 - execute sub imd
- 16 - execute and imd
- 17 - execute or imd
- 18 - execute xor imd
- 19 - execute test imd
- 24 - get first operand then go to immediate micro instructions

- 25 - execute jump
- 26 - get first operand then go to regiter register micro instructions
- 27 - go to data_memory destination or to reg_file destination
- 28 - write data_memory
- 29 - write in reg_file from data_memory
- 30 - write in reg_file from ALU
- 31 - instruction fetch (all micro-routines begin with this one)

Chapter 5

Implementation

The CPU was implemented in Vivado using VHDL
In this chapter I will present some of the important components of my CPU

5.1 Control Unit implementation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity control_unit is
    Port ( opcode : in STD_LOGIC_VECTOR (4 downto 0);
           clk: in std_logic;
           rst: in std_logic;
           mode : in STD_LOGIC_VECTOR (1 downto 0);
           D: in STD_LOGIC;
           zero: in STD_LOGIC;
           jmp : out STD_LOGIC;
           PCWr : out STD_LOGIC;
           IP_INC : out STD_LOGIC;
           MemR_W : out STD_LOGIC;
           RFDate : out STD_LOGIC;
           RegRD : out STD_LOGIC;
           RegWR : out STD_LOGIC_VECTOR(1 downto 0);
           ASrc : out STD_LOGIC;
           BSrc : out STD_LOGIC;
           AluOP : out STD_LOGIC_VECTOR(2 downto 0);
           bus_ctr: out STD_LOGIC_VECTOR(3 downto 0);
           AddrWr: out STD_LOGIC;
           flagWR:out STD_LOGIC
           );
end control_unit;

architecture Behavioral of control_unit is

type microprogram is array (0 to 31) of std_logic_vector (23 downto 0);
signal microcode: microprogram :=(
     0  => "111100000000010000000001",     -- INC
     1  => "111100000000010001000001",     -- DEC
     2  => "111100000000010111000000",     -- MOV REG, REG / MOV REG,[REG]
     3  => "111100000000001010000101",     -- ADD REG,REG
     4  => "111100000000001011000101",     -- SUB REG,REG
     5  => "111100000000001100000101",     -- AND REG,REG x
     6  => "111100000000001101000101",     -- OR REG, REG x
     7  => "111100000000001110000101",     -- XOR REG, REG x
     8  => "111110000000001100000101",     -- TEST REG, REG x
     9 =>  "111110000000000000000000",     -- JZ
     10 =>  "111110000000000000000000",    -- JNZ
     11 => "110010000000000000010110",     -- JUMP IMD
     12 => "111100000000001010000001",     -- ADD REG, IMD
     13 => "111100000000010111010100",     -- MOV REG, IMD
     14 => "111010000000000000010110",     -- MOV REG, [IMD]
     15 => "111100000000001011000101",     -- SUB REG, IMD
     16 => "111100000000001100000101",     -- AND REG, IMD x
     17 => "111100000000001101000101",     -- OR REG, IMD x
     18 => "111100000000001110000101",     -- XOR REG, IMD x
     19 => "111110000000001100000101",     -- TEST REG, IMD x
     24 => "111110000000010000010100",     -- GET_OPEERAND_A_IMD
     25 => "111111100000000000000000",     -- JUMP
     26 => "111110000000010000100100",     -- GET_OPEERAND_A
     27 => "111010000000000000000110",     -- MOV REG, [REG]
     28 => "111110001000000000000000",     -- DATA_MEMORY_WR
     29 => "111110000001000000011100",     -- REG_FILE_WR_DM
     30 => "111110000001000000001100",     -- REG_FILE_WR_ALU
     31 => "100000110000000000011100",     -- IF
     others => "100000110000000000011100"
     --others => "10000_0110000000000"
```

```
   );

signal next_addr: std_logic_vector(4 downto 0);
signal state : integer range 0 to 31 := 31;
begin


next_addr <= microcode(state)(23 downto 19);
jmp      <= microcode(state)(18);
PCWr     <= microcode(state)(17);
IP_INC   <= microcode(state)(16);
MemR_W   <= microcode(state)(15);
RFDate   <= microcode(state)(14);
RegRD    <= microcode(state)(13);
RegWR    <= microcode(state)(12 downto 11);
ASrc     <= microcode(state)(10);
BSrc     <= microcode(state)(9);
ALUop    <= microcode(state)(8 downto 6);
bus_ctr  <= microcode(state)(5 downto 2);
AddrWr  <= microcode(state)(1);
flagWr  <= microcode(state)(0);

process(clk)
begin
    if rst ='1' then
        state <= 31; -- IF
    elsif rising_edge(clk) then
        if state = 31
        and (to_integer(unsigned(opcode))<=2
        or to_integer(unsigned(opcode)) = 11
        or to_integer(unsigned(opcode)) = 13
        or to_integer(unsigned(opcode)) = 14
        or to_integer(unsigned(opcode)) = 31)

        then
            state <=to_integer(unsigned(opcode));
        else
            case to_integer(unsigned(opcode))is
                when 2 => -- When MOV instruction
                    if mode = "11" then
                        state <= to_integer(unsigned(next_addr));
                        --MOV reg, reg
                    else

                        if state = 28 or state = 29 then
                            state <= to_integer(unsigned(next_addr));
                            -- take next instruction
                        elsif state = 27 then
                            if D ='1' then
                                state <= 28; --data memory destination
                            else
                                state <=29;  -- reg file destination
                            end if;
                        else
                            state <= 27;
                        end if;
                    end if;

                when 9 =>
                    if state = 25 or state = 11 then
                        state <= to_integer(unsigned(next_addr));
                    elsif state = 9 then
                        if zero ='1' then
                            state <= 11;
                        else
                            state <=31;
                        end if;
                    else
                        state <=9;
                    end if;

                when 10 =>
                    if state = 25 or state = 11 then
                        state <= to_integer(unsigned(next_addr));
                    elsif state = 10 then
                        if zero ='0' then
                            state <= 11;
                        else
                            state <=31;
                        end if;
```

```
                    else
                        state <=10;
                    end if;

                when others => --When other instructions
                   if to_integer(unsigned(opcode)) < 9
                   and to_integer(unsigned(opcode))>2 then
                        if state = 26 then
                            state <= to_integer(unsigned(opcode));
                        elsif state /= to_integer(unsigned(opcode))
                        and state /= 30  then
                            state <= 26;
                        else
                            state <= to_integer(unsigned(next_addr));
                        end if;

                   elsif to_integer(unsigned(opcode))=12
                   or (to_integer(unsigned(opcode)) < 20
                   and to_integer(unsigned(opcode))>14) then
                        if state = 24 then
                            state <= to_integer(unsigned(opcode));
                        elsif state /= to_integer(unsigned(opcode))
                        and state /= 30  then
                            state <= 24;
                        else
                            state <= to_integer(unsigned(next_addr));
                        end if;
                   else
                        state <= to_integer(unsigned(next_addr));
                   end if;
            end case;
        end if;
    end if;
end process;

end Behavioral;
```

Its important to note that the control unit will run on the falling edge of the clock. This is done in order to fix some timeing problems and speed up execution.

5.2 Data Memory Implementation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity data_memory is
    Port (
        din : in STD_LOGIC_VECTOR(15 downto 0);
        clk : in std_logic;
        dout : out STD_LOGIC_VECTOR (15 downto 0);
        addr : in STD_LOGIC_VECTOR (8 downto 0);
        WR : in STD_LOGIC);
end data_memory;

architecture Behavioral of data_memory is

type ram_mem is array (0 to 1023 ) of std_logic_vector (7 downto 0);
signal data_mem: ram_mem :=(
     x"06",
     x"00",

     x"01",
     x"00",

     x"02",
     x"00",

     x"03",
     x"00",

     x"04",
     x"00",

     x"05",
     x"00",

     x"07",
     x"00",
    others => x"FF"
   );

signal computed_addr:std_logic_vector(9 downto 0);
begin

computed_addr <= addr & '0';

RAM_PROCESS:process(clk)
begin
    if rising_edge(clk) then
        if WR = '1' then
            data_mem(to_integer(unsigned(computed_addr))) <= din(7 downto 0);
            data_mem(to_integer(unsigned(computed_addr) + 1)) <= din(15 downto 8);
        end if ;
    end if;
end process;

dout(7 downto 0) <= data_mem(to_integer(unsigned(computed_addr)));
dout(15 downto 8) <= data_mem(to_integer(unsigned(computed_addr) + 1));

end Behavioral;
```

5.3 Register implementation

This is a component that is defined using generics. Because of this I am able to use this component for every register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reg_generic is
    generic (
        DATA_WIDTH : integer := 16
    );
    Port (
        din  : in STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
        rst  : in STD_LOGIC;
        wr   : in std_logic;
        clk  : in STD_LOGIC;
        dout : out STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0)
    );
end reg_generic;

architecture Behavioral of reg_generic is
    signal inter : std_logic_vector(DATA_WIDTH - 1 downto 0) := (others => '0');
begin
    dout <= inter;

    process(clk, rst)
    begin
        if rst = '1' then
            inter <= (others => '0');
        elsif rising_edge(clk) then
            if wr = '1' then
                inter <= din;
            end if;
        end if;
    end process;

end Behavioral;
```

Chapter 6

Testing

In order to test my CPU I will implement the following assembly program, that calculates the number of odd numbers in an array, in my CPU.

```
START:
        MOV CX, H       #Array length
        MOV BX, ARR     #Array address
LOOP:
        MOV AX, [BX]
        TEST AX,1

        JNZ INC_COUNT
        DEC CX
        jz END_PROGRAM
        INC BX
        JMP LOOP

INC_COUNT:
        inc DX
        DEC CX
        jz END_PROGRAM
        INC BX
        JMP LOOP

END_PROGRAM:
```

Converted into my instruction set it will loook like this

```
START:
    mov R2, 0000h
    mov R1, 0001h
LOOP:
    mov R0, [R1]
    test R0, 0001h
    jnz INC_COUNT(9)
    dec R2
    jz END_PROGRAM(14)
    inc R1
    jmp LOOP(2)
INC_COUNT:
    inc R3
    dec R2
    jz END_PROGRAM(14))
    inc R1
    jmp LOOP(2)
END_PROGRAM:
```

This is how it will look in the instruction memory

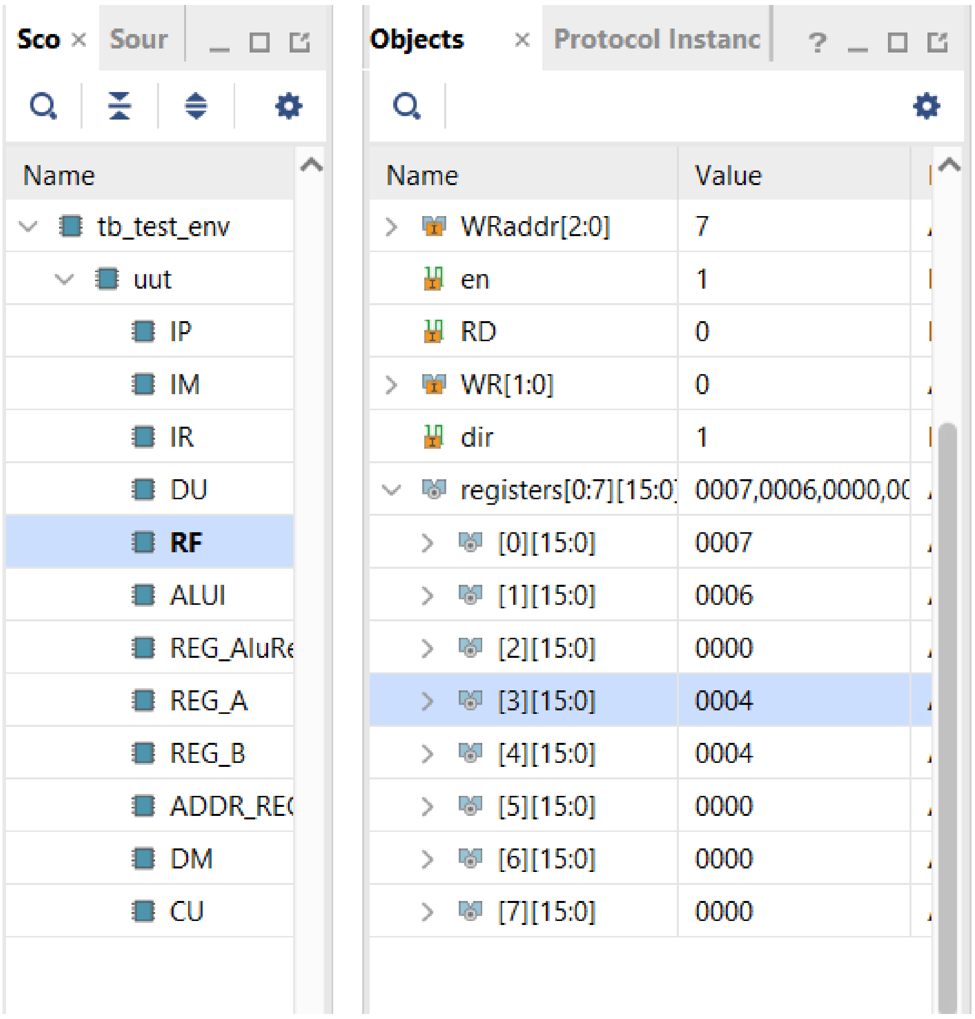
```
constant instr_mem : rom_mem := (
                "011100100000000000000000", -- (mov R2, 0000h)
                "011100010000000000000001", -- (mov R1, 0001h)
                --LOOP
                "000100000000000100000000", -- (mov R0, [R1])
                "100110000000000000000001", -- (test R0, 0001h)
                "010100000000000000001001", -- (jnz INC_COUNT(9))
                "000010100000000000000000", -- (dec R2)
                "010010000000000000001110", -- (jz END_PROGRAM(14))
                "000000010000000000000000", -- (inc R1)
                "010110000000000000000010", -- (jmp LOOP(2))
                -- INC_COUNT
                "000000110000000000000000", -- (inc R3)
                "000010100000000000000000", -- (dec R2)
                "010010000000000000001110", -- (jz END_PROGRAM(14))
                "000000010000000000000000", -- (inc R1)
                "010110000000000000000010", -- (jmp LOOP(2))
                -- END_PROGRAM
                "000100000111000110000000", -- (mov R4, R3)
                others => x"ffffff"
        );
```

And this is the array in the data memory

```
signal data_mem: ram_mem :=(
     x"06",
     x"00",

     x"01",
     x"00",

     x"02",
     x"00",

     x"03",
     x"00",

     x"04",
     x"00",

     x"05",
     x"00",

     x"07",
     x"00",
    others => x"FF"
   );
```

If we run the simulation we can see the execution of the program in the waveform, and the result in the Objects tab

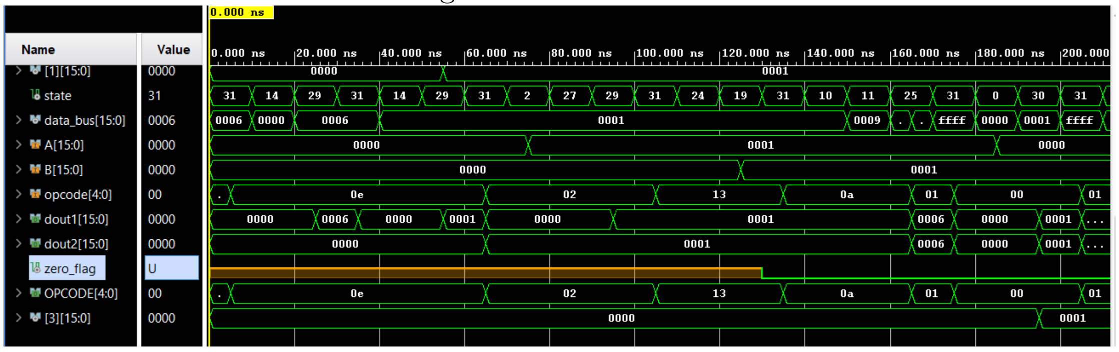
Figure 6.1: Objects



Name
- tb_test_env
  - uut
    - IP
    - IM
    - IR
    - DU
    - **RF**
    - ALUI
    - REG_AluRe
    - REG_A
    - REG_B
    - ADDR_REC
    - DM
    - CU

Name	Value
WRaddr[2:0]	7
en	1
RD	0
WR[1:0]	0
dir	1
registers[0:7][15:0]	0007,0006,0000,0C
[0][15:0]	0007
[1][15:0]	0006
[2][15:0]	0000
[3][15:0]	0004
[4][15:0]	0004
[5][15:0]	0000
[6][15:0]	0000
[7][15:0]	0000

Figure 6.2: Waveform



Name	Value
[1][15:0]	0000
state	31
data_bus[15:0]	0006
A[15:0]	0000
B[15:0]	0000
opcode[4:0]	00
dout1[15:0]	0000
dout2[15:0]	0000
zero_flag	U
OPCODE[4:0]	00
[3][15:0]	0000

Chapter 7

Conclusion

To conclude I managed to design and implement a working CISC processor with 3 addressing modes, that is capable of running simple programs.

The processor can be easily expanded further by adding new microinstructions in the control unit. It would also be possible to add another addressing mode but that would probably require the data path to be modified.

Bibliography

[1] Geeksforgeeks. Architecture of 8086. `https://www.geeksforgeeks.org/architecture-of-8086/`. [ONLINE].

[2] Geeksforgeeks. omputer organization | hardwired v/s micro-programmed control unit. `https://www.geeksforgeeks.org/computer-organization-hardwired-vs-micro-programmed-control-unit/`. [ONLINE].

[3] R. N. Rajotiya. 8086 instruction format-ii. `https://care4you.in/8086-instruction-format-ii/`. [ONLINE].

[4] Wikipedia. Complex instruction set computer. `https://en.wikipedia.org/wiki/Complex_instruction_set_computer`. [ONLINE].