

در این پروژه، پردازنده ۶ بیتی که در کلاس طراحی شده را پیاده‌سازی کرده و برای آن برنامه‌نویسی می‌کنیم.

توجه: این پروژه در صورتی قابل قبول است که برای آن گزارش هم نوشته شود. در این گزارش نحوه پیاده‌سازی پردازنده و اجرای برنامه توسط آن با استفاده از عکس‌های مناسب از خروجی شبیه‌سازی نشان داده شود.

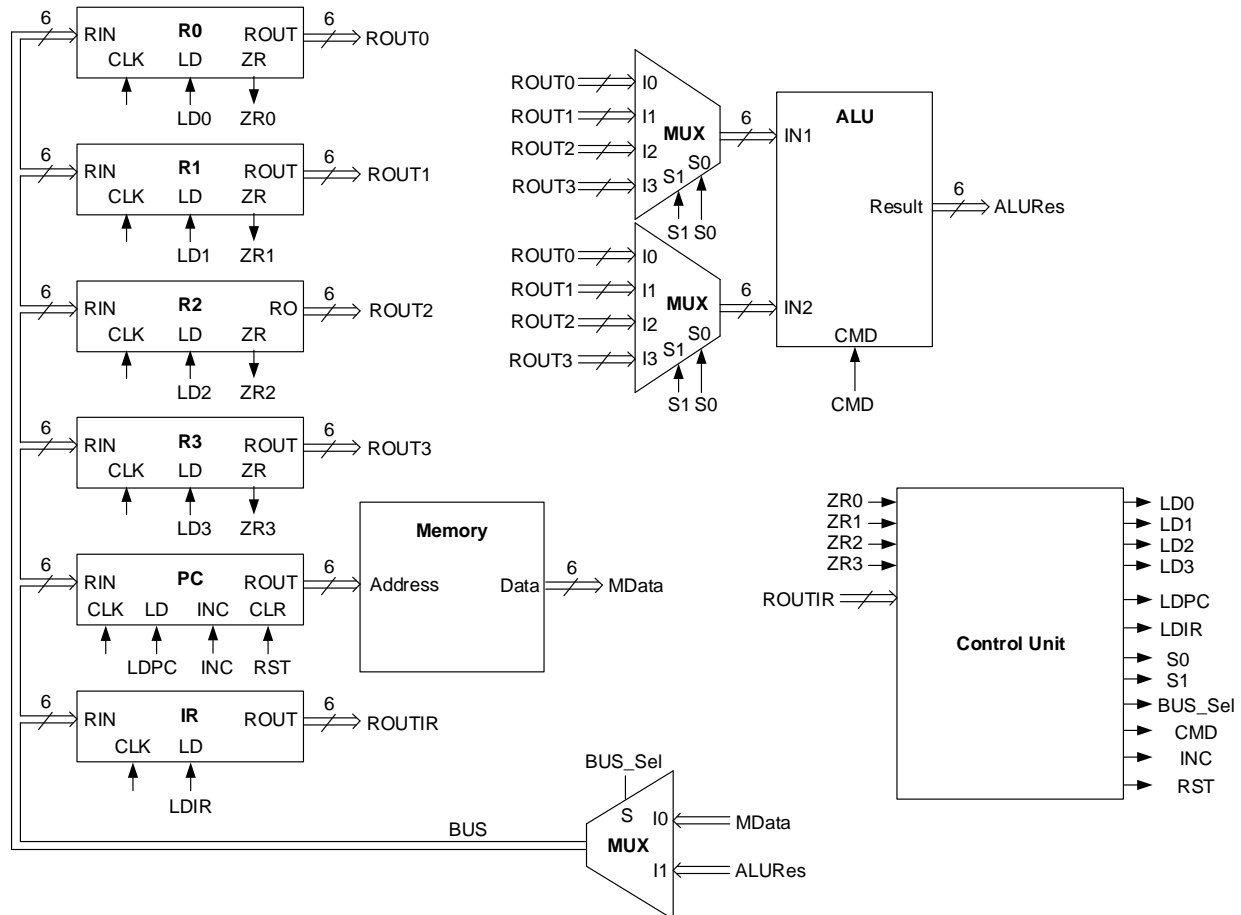
بخش اول (۴۰٪ نمره پروژه): برای انجام این پروژه ابتدا پردازنده را با استفاده از VHDL یا Verilog پیاده‌سازی کرده و صحت عملکرد آن را با اجرای کد زیر که دو عدد ۷ و ۴ را با هم جمع می‌کند بررسی کنید.

```
LOAD R0, 7
LOAD R1, 4
ADD R0, R1
```

بخش دوم (۲۰٪ نمره پروژه): با توجه به این‌که این پردازنده دستور ضرب ندارد، عمل ضرب را با استفاده از عمل جمع و به صورت نرم‌افزاری پیاده‌سازی کرده و صحت عملکرد آن را با یک مثال نشان دهید (مشابه بخش اول یک کد اسمبلی بنویسید که عمل ضرب را انجام دهد). به عنوان مثال، حاصلضرب عدد ۸ در ۶ را حساب کند.

بخش سوم (۴۰٪ نمره پروژه): دستور ضرب را با کمترین سربار سخت‌افزاری به مجموعه دستورات اضافه کرده و صحت عملکرد آن را با نوشتن یک کد که حاصلضرب ۸ در ۶ را حساب کند نشان دهید. توجه کنید که برای این کار نیاز است تغییراتی در سخت‌افزار و کد دستورات ایجاد کنید.

نمره اضافی (۱ نمره): پیاده‌سازی اسمبلر برای تبدیل کد اسمبلی به کد باینری با استفاده از زبان‌های سطح بالا مانند جاوا و پایتون. معماری پردازنده:



دستورات پردازنده:

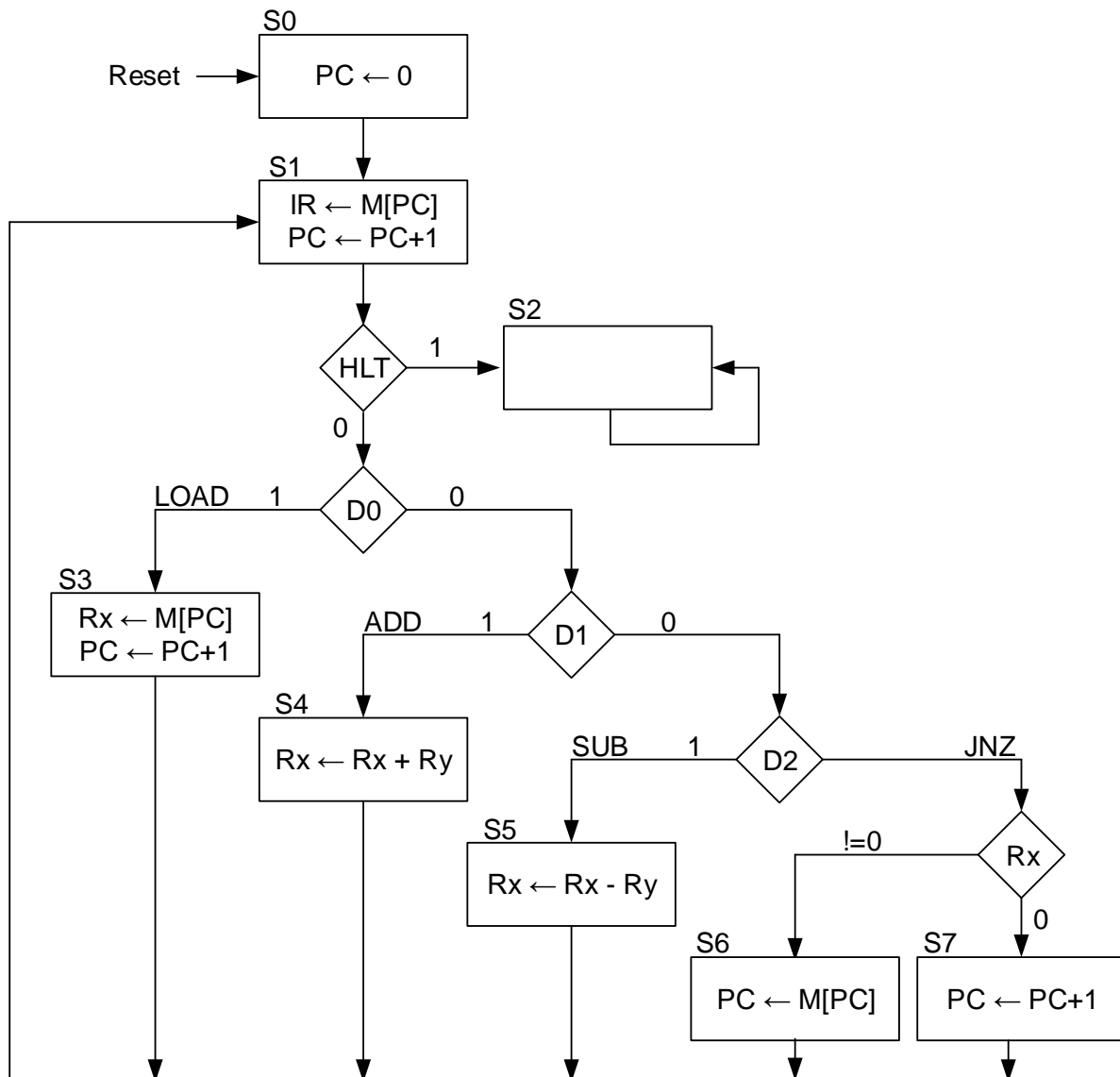
این پردازنده چهار دستور LOAD، ADD، SUB و JNZ با کد دستور (Op Code) زیر است:

کد دستور	دستور
00	LOAD
01	ADD
10	SUB
11	JNZ

قالب دستورات:

Op Code	R _{SRC}	R _{DST}
---------	------------------	------------------

چینش در حافظه	RTL	اسمبلی دستور			
<div>PC →</div> <table><tr><td>00 Rx 00</td></tr><tr><td>مقدار</td></tr><tr><td>دستور بعدی</td></tr></table>	00 Rx 00	مقدار	دستور بعدی	$Rx \leftarrow M[PC]$	LOAD Rx, VALUE
00 Rx 00					
مقدار					
دستور بعدی					
<div>PC →</div> <table><tr><td>01 Rx Ry</td></tr><tr><td>دستور بعدی</td></tr></table>	01 Rx Ry	دستور بعدی	$Rx \leftarrow Rx + Ry$	ADD Rx, Ry	
01 Rx Ry					
دستور بعدی					
<div>PC →</div> <table><tr><td>10 Rx Ry</td></tr><tr><td>دستور بعدی</td></tr></table>	10 Rx Ry	دستور بعدی	$Rx \leftarrow Rx - Ry$	SUB Rx, Ry	
10 Rx Ry					
دستور بعدی					
<div>PC →</div> <table><tr><td>11 Rx 00</td></tr><tr><td>آدرس پرش</td></tr><tr><td>دستور بعدی</td></tr></table>	11 Rx 00	آدرس پرش	دستور بعدی	$\text{If } (Rx \neq 0) \text{ PC} \leftarrow M[PC]$ $\text{else PC} \leftarrow \text{PC} + 1$	JNZ Rx, Address
11 Rx 00					
آدرس پرش					
دستور بعدی					



به نام خدا

گزارش پروژه پایانی هم‌طراحی

امین دائم دوست

۹۸۰۱۲۲۶۸۰۰۴۰

محمد حسین رحیمی

۹۸۰۱۲۲۶۸۰۰۳۶

پوریا عباسی شنبه بازاری

۹۸۰۱۲۲۶۸۱۰۱۲

توضیحات کلی:

در این پروژه دو پردازنده‌ی design1 و design2 طراحی شده‌اند که اولی برای چهار عملیات ADD، SUB، JNZ، LOAD برای بخش اول پروژه و دومی برای بخش سوم بوده که عملیات MULT را نیز انجام می‌دهد.

همچنین فایل تبدیل اسمبلی به باینری نیز در پوشه‌ی پروژه قابل مشاهده است که برای هر کی از پردازنده‌ها آماده شده است تا دستورات را به صورت مستقیم وارد فایل memory.vhd کند.

همچنین توضیحات opcode ها در بخش اسمبلر با جزئیات نوشته شده.

بخش اول: طراحی پروژه و تست جمع

ALU:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_signed.all;

entity ALU is
port(
    in1,in2 : in std_logic_vector(5 downto 0);
    cmd : in std_logic;
    result : out std_logic_vector(5 downto 0)
);
end ALU;

architecture ALU of ALU is
    signal result_sig: std_logic_vector(5 downto 0);
    begin
        process(cmd,in1,in2)
        begin
            case cmd is
                when '0' =>
                    result_sig <= in1 + in2; -- add
                when '1' =>
                    result_sig <= in1 - in2; -- sub
                when others => result_sig <= (others => 'X');
            end case;
        end process;
        result <= result_sig;
    end ALU;
```

وظیفه این واحد اجرای عمل جمع و تفریق دو رجیستر است، اگر $CMD=1$ عمل تفریق و اگر $CMD=0$ عمل جمع را انجام می‌دهد.

MRegister:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity MRegister is
    port(
        RIN : in std_logic_vector(5 downto 0);
        CLK : in std_logic;
        LD : in std_logic;
        ROUT : out std_logic_vector(5 downto 0);
        ZR : out std_logic);
end MRegister;

architecture MRegister of MRegister is
    --signal RN0,RN1,RN2,RN3,R_0,R_1,R_2,R_3: std_logic_vector(5 downto 0);
    signal Reg: std_logic_vector(5 downto 0);
begin
    ----- MAIN registers -----

    process(CLK)
    begin
        if(rising_edge(CLK)) then
            if ( LD = '1') then
                Reg <= RIN;
            end if;

        end if;
    end process;

    ROUT <= Reg;

    ZR <= '1' when (Reg="000000") else '0';

end MRegister;
```

چهار رجیستر اصلی، که مقداری را لود کرده و خروجی می‌دهند. همچنین اگر برابر ۰ باشند، مقدار ZR=1 میکنند.

Main:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Main is
    port(
        CLK : in STD_LOGIC;
        RST : in STD_LOGIC;
        BUS_Sel : out STD_LOGIC;
        INC : out std_logic ;
        CMD : out STD_LOGIC;
        IR : out STD_LOGIC_VECTOR(5 downto 0);
        PC : out STD_LOGIC_VECTOR(5 downto 0);
        LD_REG : out STD_LOGIC_VECTOR(0 to 3);
        Reg0 : out STD_LOGIC_VECTOR(5 downto 0);
        Reg1 : out STD_LOGIC_VECTOR(5 downto 0);
        Reg2 : out STD_LOGIC_VECTOR(5 downto 0);
        Reg3 : out STD_LOGIC_VECTOR(5 downto 0);
        Select0 : out STD_LOGIC_VECTOR(1 downto 0);
        Select1 : out STD_LOGIC_VECTOR(1 downto 0);
        ALU_out : out STD_LOGIC_VECTOR(5 downto 0);
        bus_data : out STD_LOGIC_VECTOR(5 downto 0);
        Memory_Data : out STD_LOGIC_VECTOR(5 downto 0)
    );
end Main;

--)) End of automatically maintained section

architecture Main of Main is

    signal LD0_sig,LD1_sig,LD2_sig,LD3_sig,LD_PC_sig, LD_IR_sig, ZR0_sig, ZR1_sig, ZR2_sig, ZR3_sig, INC_sig, CLR_sig, alu_cmd_sig,BUS_Sel_sig : std_logic;
    signal RIN0_sig, RIN1_sig, RIN2_sig, RIN3_sig, RIN_PC_sig, RIN_IR_sig, ROUT0_sig, ROUT1_sig, ROUT2_sig, ROUT3_sig, ROUT_PC_sig, ROUT_IR_sig,MData_sig,bus_input_sig : std_logic_vector(5 downto 0);
    signal alu_in1_sig, alu_in2_sig, alu_result_sig : std_logic_vector(5 downto 0);
    signal sel0_sig,sel1_sig : std_logic_vector(1 downto 0);

    component ALU is
        port(
            in1,in2 : in std_logic_vector(5 downto 0);
            cmd : in std_logic;
            result : out std_logic_vector(5 downto 0)
        );
    end component;

    component MRegister is
        port(
            RIN : in std_logic_vector(5 downto 0);
            CLK : in std_logic;
            LD : in std_logic;
            ROUT : out std_logic_vector(5 downto 0);
            ZR : out std_logic;
        );
    end component;

    component IR_Regester is
        port(
            RIN : in std_logic_vector(5 downto 0);
            CLK : in std_logic;
            LD : in std_logic;
            ROUT : out std_logic_vector(5 downto 0)
        );
    end component;

    component PC_Regester is
        port(
            RIN : in std_logic_vector(5 downto 0);
            CLK : in std_logic;
            LD, INC, CLR : in std_logic;
            ROUT : out std_logic_vector(5 downto 0));
    end component;

    component ControlUnit is
        port( clk,rst: in std_logic;
              ZR0,ZR1,ZR2,ZR3 : in std_logic;
              ROUT_IR : in std_logic_vector(5 downto 0);
              LD0,LD1,LD2,LD3 : out std_logic;
              LD_PC , LD_IR :out std_logic;
              Sel0: out std_logic_vector(1 downto 0);
              Sel1: out std_logic_vector(1 downto 0);
              BUS_Sel , ALU_CMD : out std_logic;
              INC , CLR : out std_logic);
    end component;

    component MUX4x2 is
        port(
            in0,in1,in2,in3 : in std_logic_vector(5 downto 0);
            S : in std_logic_vector(1 downto 0);
            output : out std_logic_vector(5 downto 0)
        );
    end component;

    component Memory is
        port(
            Address : in std_logic_vector(5 downto 0);
            Data : out std_logic_vector(5 downto 0)
        );
    end component;
begin
    -- Registers
    --Main
    Reg0_Cmp : MRegister      PORT MAP (RIN=>bus_input_sig, CLK=>CLK, LD=>LD0_sig, ZR=>ZR0_sig, ROUT=>ROUT0_sig);
    Reg1_Cmp : MRegister      PORT MAP (RIN=>bus_input_sig, CLK=>CLK, LD=>LD1_sig, ZR=>ZR1_sig, ROUT=>ROUT1_sig);
    Reg2_Cmp : MRegister      PORT MAP (RIN=>bus_input_sig, CLK=>CLK, LD=>LD2_sig, ZR=>ZR2_sig, ROUT=>ROUT2_sig);
    Reg3_Cmp : MRegister      PORT MAP (RIN=>bus_input_sig, CLK=>CLK, LD=>LD3_sig, ZR=>ZR3_sig, ROUT=>ROUT3_sig);
    --PC
    Reg_PC : PC_Regester      PORT MAP (RIN=>bus_input_sig, CLK=>CLK, LD=>LD_PC_sig,CLR=> CLR_sig, INC=> INC_sig , ROUT=>ROUT_PC_sig);
    --IR
    Reg_IR : IR_Regester      PORT MAP (RIN=>bus_input_sig, CLK=>CLK, LD=>LD_IR_sig , ROUT=>ROUT_IR_sig);

    -- ALU
    ALU0 : ALU                PORT MAP (in1=>alu_in1_sig, in2=>alu_in2_sig, cmd=>alu_cmd_sig,result =>alu_result_sig);
    --Control Unit
    ControlUnit_inst : ControlUnit port map(clk => CLK,rst => RST,ZR0 => ZR0_sig,ZR1 => ZR1_sig,ZR2 => ZR2_sig,ZR3 => ZR3_sig,ROUT_IR => ROUT_IR_sig,LD0 => LD0_sig,LD1 => LD1_sig,LD2 => LD2_sig,LD3 => LD3_sig,
    ROM0 : Memory             PORT MAP (Address=>ROUT_PC_sig, Data=>MData_sig);

    --ALU inputs Multiplexers
    MUX0 : MUX4x2              PORT MAP (in0=> ROUT0_sig, in1=>ROUT1_sig, in2=>ROUT2_sig, in3=>ROUT3_sig, s=>sel0_sig, output=>alu_in1_sig);
    MUX1 : MUX4x2              PORT MAP (in0=> ROUT0_sig, in1=>ROUT1_sig, in2=>ROUT2_sig, in3=>ROUT3_sig, s=>sel1_sig, output=>alu_in2_sig);
    --BUS
    bus_input_sig <= alu_result_sig when BUS_Sel_sig='0' else MData_sig when BUS_Sel_sig='1';

    BUS_Sel <= BUS_Sel_sig;
    INC <= INC_sig;
    CMD <= alu_cmd_sig;
    IR <= ROUT_IR_sig;
    PC <= Rout_PC_sig;
    LD_REG(0) <= LD0_sig;
    LD_REG(1) <= LD1_Sig;
    LD_REG(2) <= LD2_sig;
    LD_REG(3) <= LD3_sig;
    Reg0 <= Rout0_sig;
    Reg1 <= Rout1_sig;
    Reg2 <= Rout2_sig;
    Reg3 <= Rout3_sig;
    Select0 <= sel0_Sig;
    Select1 <= sel1_Sig;
    ALU_out <= alu_result_sig;
    bus_data <= bus_input_sig;
    Memory_Data <= MData_sig;
end Main;

```

وظیفه تعریف کامپوننت ها و برقراری اتصال بین آن ها و همچنین مپ کردن پورت ها را بر عهده دارد

IR_register:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity IR_Regester is
    port(
        RIN : in std_logic_vector(5 downto 0);
        CLK : in std_logic;
        LD : in std_logic;
        ROUT : out std_logic_vector(5 downto 0)
    );
end IR_Regester;

architecture IR_Regester of IR_Regester is
    --signal RN0,RN1,RN2,RN3,R_0,R_1,R_2,R_3: std_logic_vector(5 downto 0);
    signal IR: std_logic_vector(5 downto 0);
begin
    ----- MAIN registers -----

    process(CLK)
    begin
        if(rising_edge(CLK)) then
            if ( LD = '1' ) then
                IR <= RIN;
            end if;
        end if;
    end process;

    ROUT <= IR;
end IR_Regester;

```

رجیستر Instruction Register که وظیفه خواندن دستور مموری و ارسال آن به واحد کنترل را برعهده دارد

PC_Register:


```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity PC_Register is
    port(
        RIN : in std_logic_vector(5 downto 0);
        CLK : in std_logic;
        LD, INC, CLR : in std_logic;
        ROUT : out std_logic_vector(5 downto 0));
end PC_Register;

architecture PC_Register of PC_Register is
    --signal RN0,RN1,RN2,RN3,R_0,R_1,R_2,R_3: std_logic_vector(5 downto 0);
    signal PC: std_logic_vector(5 downto 0);
begin
    ----- MAIN registers -----

    process(CLK,CLR)
    begin
        if CLR = '1' then
            PC <= (others => '0');

            elsif(rising_edge(CLK)) then
                if ( LD = '1') then
                    PC <= RIN;
                end if;
                if (INC = '1') then
                    PC <= PC + 1;
                end if;

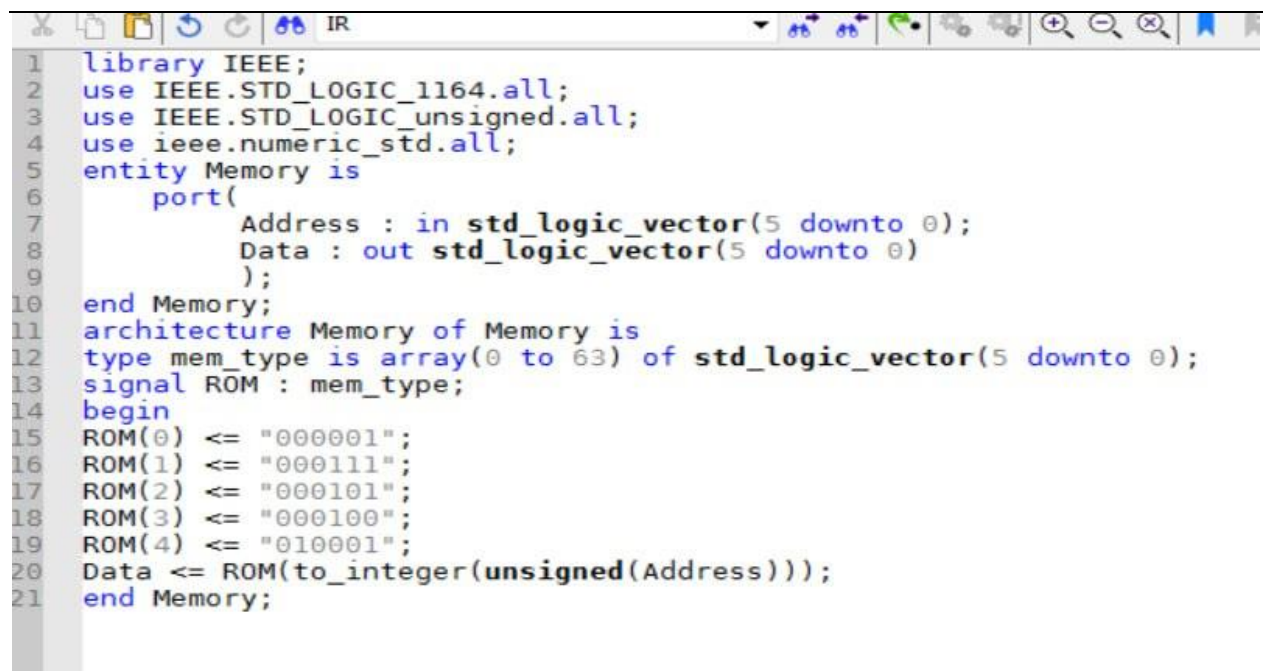
            end if;
        end process;

        ROUT <= PC;
    end PC_Register;

```

رجیستر Program Counter که وظیفه اشاره به آدرس در حافظه و ارسال آن به واحد کنترل را برعهده دارد

Memory:



```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.STD_LOGIC_unsigned.all;
4  use ieee.numeric_std.all;
5  entity Memory is
6      port(
7          Address : in std_logic_vector(5 downto 0);
8          Data : out std_logic_vector(5 downto 0)
9      );
10 end Memory;
11 architecture Memory of Memory is
12     type mem_type is array(0 to 63) of std_logic_vector(5 downto 0);
13     signal ROM : mem_type;
14     begin
15         ROM(0) <= "0000001";
16         ROM(1) <= "0001111";
17         ROM(2) <= "0001011";
18         ROM(3) <= "0001000";
19         ROM(4) <= "0100011";
20         Data <= ROM(to_integer(unsigned(Address)));
21     end Memory;
    
```

کد برنامه درون مموری ریخته میشود. وظیفه ایجاد این کد بر عهده اسمبلر است که به عنوان نمره اضافی طراحی شده. (اسمبلر در قسمت مربوط به خود توضیح داده میشود) عکس تغییر کنه!!!!

ControlUnit:

```

1
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 use IEEE.numeric_std.all;
5
6
7 entity ControlUnit is
8     port( clk,rst: in std_logic;
9           ZR0,ZR1,ZR2,ZR3 : in std_logic;
10          ROUT_IR : in std_logic_vector(5 downto 0);
11          LD0,LD1,LD2,LD3 : out std_logic;
12          LD_PC , LD_IR :out std_logic;
13          selm0_0,selm0_1,selm1_0,selm1_1: out std_logic;
14          BUS_sel , ALU_CMD : out std_logic;
15          INC , CLR : out std_logic);
16 end ControlUnit;
17
18
19 architecture ControlUnit of ControlUnit is
20     signal index : integer;
21     signal Z : std_logic_vector(3 downto 0);
22     type States is (S0, S1, S2, S3, S4, S5, S6, S7, S_HLT);
23     signal STATE : States;
24 begin
25     index <= to_integer(unsigned(ROUT_IR(3 downto 2)));
26
27     Z(0) <= ZR0;
28     Z(1) <= ZR1;
29     Z(2) <= ZR2;
30     Z(3) <= ZR3;
31
32     process( rst, CLK )
33     begin
34         if(rst = '1') then
35
36             CLR <= '1';
37             LD0 <= '0';
38             LD1 <= '0';
39             LD2 <= '0';
40             LD3 <= '0';
41             LD_IR <= '0';
42             LD_PC <= '0';
43             INC <= '0';
44             BUS_sel <= '0';
45             ALU_CMD <= '0';
46             STATE <= S0;
47

```

```

48     elsif( rising_edge(CLK) ) then
49         CLR    <= '0';
50         LD0    <= '0';
51         LD1    <= '0';
52         LD2    <= '0';
53         LD3    <= '0';
54         LD_IR   <= '0';
55         LD_PC   <= '0';
56         INC     <= '0';
57         BUS_Sel <= '0';
58         ALU_CMD <= '0';
59         case STATE is
60             when S0 =>
61                 INC <= '1';
62                 LD0 <= '0';
63                 LD1 <= '0';
64                 LD2 <= '0';
65                 LD3 <= '0';
66                 LD_IR <= '1';
67                 LD_PC <= '0';
68
69                 BUS_Sel <= '0';
70
71                 STATE <= S1;
72
73             when S1 =>
74                 CLR <= '0';
75                 INC <= '0';
76                 LD_IR <= '0';
77
78                 STATE <= S_HLT;
79
80             when S_HLT =>
81
82                 if(ROUT_IR= "000000")then
83                     STATE <= S2;
84                 elsif (ROUT_IR(5 downto 4)= "00")then
85                     STATE <= S3;
86                 elsif (ROUT_IR(5 downto 4) = "01")then
87                     STATE <= S4;
88                 elsif (ROUT_IR(5 downto 4) = "10")then
89                     STATE <= S5;
90                 elsif (ROUT_IR(5 downto 4) = "11")then
91                     if(Z(index) = '0')then
92                         STATE <= S6;
93                     else
94                         STATE <= S7;
95                     end if;
96                 end if;
97

```

```

98      when S2 =>
99          INC <= '0';
100         STATE <= S2;
101
102     when S3 =>
103         if(ROUT_IR(3 downto 2) = "00")then
104             LD0 <= '1';
105         elsif(ROUT_IR(3 downto 2) = "01")then
106             LD1 <= '1';
107         elsif(ROUT_IR(3 downto 2) = "10")then
108             LD2 <= '1';
109         elsif(ROUT_IR(3 downto 2) = "11")then
110             LD3 <= '1';
111         end if;
112
113         INC <= '1';
114         BUS_Sel <= '0';
115
116         STATE <= S0;
117
118     when S4 =>
119         if(ROUT_IR(3 downto 2) = "00")then
120             LD0 <= '1';
121         elsif(ROUT_IR(3 downto 2) = "01")then
122             LD1 <= '1';
123         elsif(ROUT_IR(3 downto 2) = "10")then
124             LD2 <= '1';
125         elsif(ROUT_IR(3 downto 2) = "11")then
126             LD3 <= '1';
127         end if;
128
129         ALU_CMD <= '0';
130         BUS_Sel <= '1';
131
132         selm0_0 <= ROUT_IR(2);
133         selm0_1 <= ROUT_IR(3);
134         selm1_0 <= ROUT_IR(0);
135         selm1_1 <= ROUT_IR(1);
136
137         STATE <= S0;
138

```

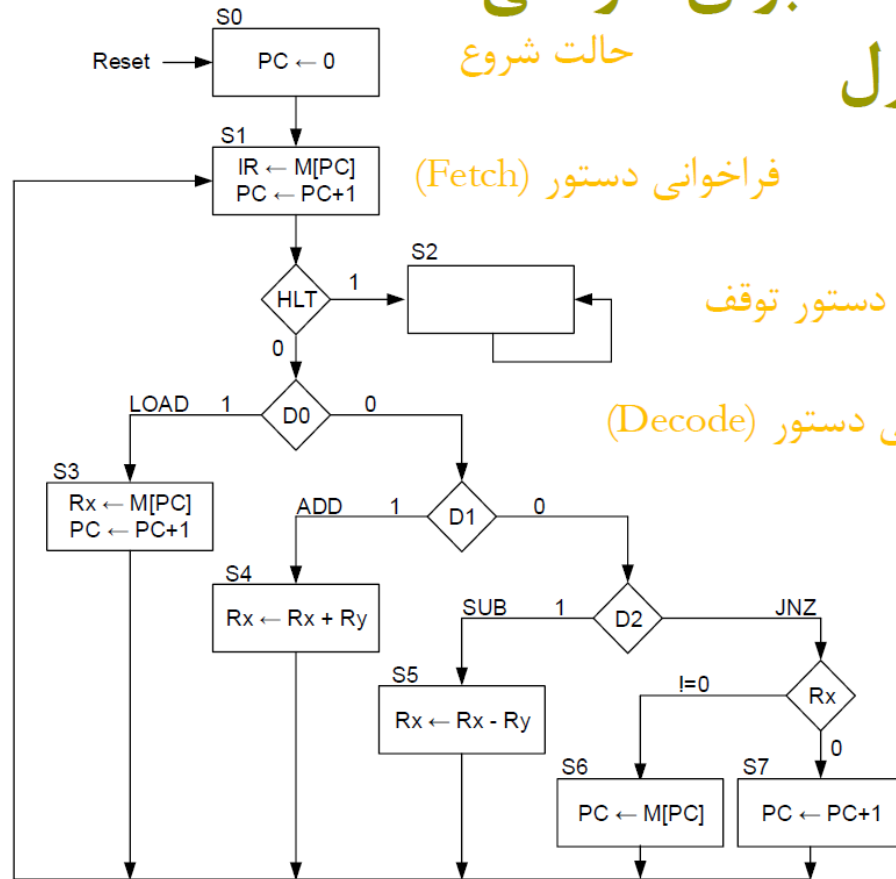
```

138
139         when S5 =>
140             if(ROUT_IR(3 downto 2) = "00")then
141                 LD0 <= '1';
142             elsif(ROUT_IR(3 downto 2) = "01")then
143                 LD1 <= '1';
144             elsif(ROUT_IR(3 downto 2) = "10")then
145                 LD2 <= '1';
146             elsif(ROUT_IR(3 downto 2) = "11")then
147                 LD3 <= '1';
148             end if;
149
150             ALU_CMD <= '1';
151             BUS_Sel <= '1';
152             selm0_0 <= ROUT_IR(2);
153             selm0_1 <= ROUT_IR(3);
154             selm1_0 <= ROUT_IR(0);
155             selm1_1 <= ROUT_IR(1);
156
157             STATE <= S0;
158
159
160         when S6 =>
161             LD_PC <= '1';
162             BUS_Sel <= '0';
163             STATE <= S0;
164
165         when S7 =>
166             INC <= '1';
167             STATE <= S0;
168         end case;
169     end if;
170 end process;
171
172
173 end ControlUnit;
174

```

واحد کنترل اصلی ترین بخش پردازنده است. طراحی آن طبق این فلوچارت انجام شده:

چارت ASM برای طراحی واحد کنترل



22

در استیت ۰، تمام مقادیر را ۰ میکنیم و مقدار PC را ریست میکنیم.

در استیت ۱، به IR دستور خواندن میدهیم و PC را یک واحد جلو میبریم

در استیت hlt، دستور را از IR میگیریم و بر اساس opcode تصمیم میگیریم چکار کنیم

در استیت ۲، دستور را متوقف میکنیم

در استیت ۳، رجیستر مبدا را لود میکنیم و PC را رو به جلو میبریم

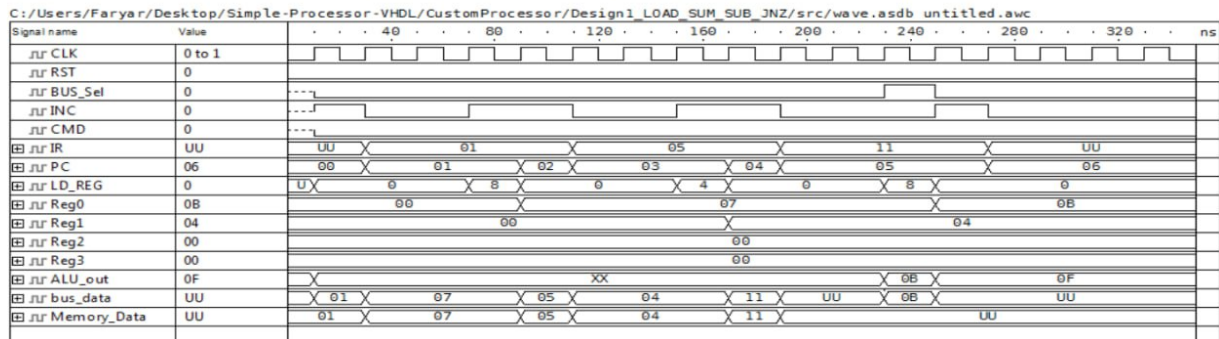
در استیت ۴، مقادیر select را برای ALU تعیین میکنیم و رجیستر مبدا را لود میکنیم و همچنین عمل جمع را مشخص میکنیم

در استیت ۵، مشابه استیت ۴ است ولی برای عمل تفریق

در استیت ۶، PC را به آدرس مقصد عمل JNZ منتقل میکنیم

در استیت ۷، PC را به جلو میبریم.

حال بررسی اجرا برای عمل جمع:



مقدار R0=0BHex را مشاهده میکنیم که برابر با 11 است. پس به درستی عمل میکند.

بخش دوم: طراحی ضرب به صورت نرم افزاری

برای این کار میبایست عمل جمع را به تعداد کافی انجام دهیم، در این صورت، 6×8 باید شش بار عدد ۸ را با خودش جمع کنیم، این کار را با دستورات ADD و JNZ انجام میدهیم:

```
Load R0, 0
Load R1, 1
Load R2, 8
Load R3, 6
Add R0, R2
Sub R3, R1
Jnz R3, 8
Hlt
```

عمکرد به این صورت است: ابتدا مقادیر ۰ و ۱ در رجیستر لود میشوند،

عمل جمع شدن در رجیستر ۰ انجام میشود و از رجیستر ۱ برای کم کردن طرف دوم ضرب، در اینجا عدد ۶، استفاده میشود. هر بار که رجیستر ۰ را با عدد ۸ جمع کردیم، یک واحد از رجیستر ۳ که عدد ۶ است کم میکنیم.

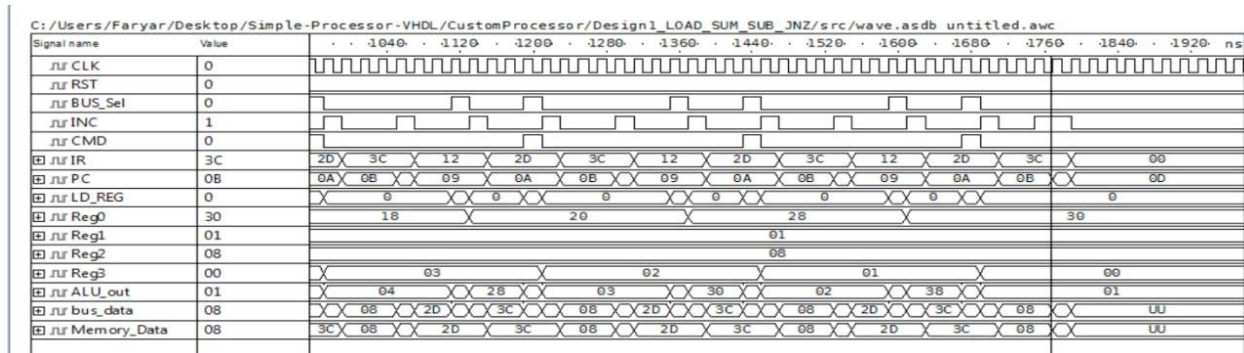
این کار را آنقدر تکرار میکنیم تا مقدار رجیستر ۳ به ۰ برسد.

در این حالت، ۶ بار جمع انجام شده و رجیستر ۰ عدد ۴۸ نشان میدهد.

با اسمبل کردن کد توسط اسمبلری که به عنوان نمره مثبت طراحی شده (خود اسمبلر در بخش مربوطه توضیح داده میشود) این مقدار در فایل رام ایجاد میشود:

```
ROM(0) <= "000001";
ROM(1) <= "000000";
ROM(2) <= "000101";
ROM(3) <= "000001";
ROM(4) <= "001001";
ROM(5) <= "001000";
ROM(6) <= "001101";
ROM(7) <= "000110";
ROM(8) <= "010010";
ROM(9) <= "101101";
ROM(10) <= "111100";
ROM(11) <= "001000";
ROM(12) <= "000000";
```

نتیجه پیاده سازی:



مشاهده میشود که مقدار R0=30Hex که برابر ۴۸ است میشود.

پس پیاده سازی موفق بود.

بخش سوم، سخت افزار ضرب:

Design2 تقریبا مشابه design1 است ولی در چند کامپوننت تفاوت دارند

ALU:

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity ALU is
7
8 port(
9     in1,in2 : in std_logic_vector(6 downto 0);
10    alu_cmd : in std_logic_vector(1 downto 0);
11    result : out std_logic_vector(6 downto 0)
12 );
13 end ALU;
14
15 architecture ALU_of ALU is
16     signal result_sig: std_logic_vector(6 downto 0);
17     begin
18         process(alu_cmd,in1,in2)
19         begin
20             case alu_cmd is
21                 when "00" =>
22                     result_sig <= in1 + in2; -- add
23                 when "01" =>
24                     result_sig <= in1 - in2; -- sub
25                 when "10" =>
26                     result_sig <= std_logic_vector(to_unsigned(to_integer(unsigned(IN1)) * to_integer(unsigned(IN2)), 7));
27                 when others => result_sig <= (others => 'Z');
28             end case;
29         end process;
30         result <= result_sig;
31     end ALU;
32
33 end ALU;
34

```

در ازای CMD=00 عمل جمع، CMD=01 هم تفریق، CMD=10 عمل ضرب را انجام میدهد.

ControlUnit:

```

78     when S8 =>
79         if(ROUT_IR(3 downto 2) = "00")then
80             LD0 <= '1';
81         elsif(ROUT_IR(3 downto 2) = "01")then
82             LD1 <= '1';
83         elsif(ROUT_IR(3 downto 2) = "10")then
84             LD2 <= '1';
85         elsif(ROUT_IR(3 downto 2) = "11")then
86             LD3 <= '1';
87         end if;
88
89         alu_CMD <= "10";
90         BUS_Sel <= '1';
91
92         selm0_0 <= ROUT_IR(2);
93         selm0_1 <= ROUT_IR(3);
94
95         -- MUX1 selector
96         selm1_0 <= ROUT_IR(0);
97         selm1_1 <= ROUT_IR(1);
98
99         STATE <= S0;

```

مشابه design1 است ولی استیت هشتم اضافه شده.

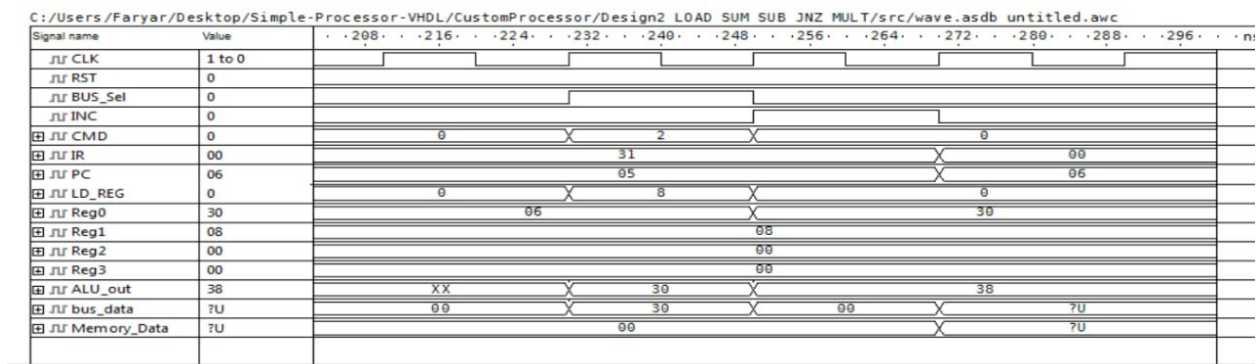
حال باید ضرب را تست کنیم:

```
Load R0, 6
Load R1, 8
MULT R0, R1
Hlt
```

کد اسمبلی آن به این صورت است. با دادن این کد به اسمبلر design2، فایل مموری به این شکل میشود:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.all;
3 use IEEE.STD_LOGIC_unsigned.all;
4 use ieee.numeric_std.all;
5 entity Memory is
6     port(
7         Address : in std_logic_vector(6 downto 0);
8         Data : out std_logic_vector(6 downto 0)
9     );
10 end Memory;
11 architecture Memory of Memory is
12     type mem_type is array(0 to 63) of std_logic_vector(5 downto 0);
13     signal ROM : mem_type;
14     begin
15         ROM(0) <= "000001";
16         ROM(1) <= "000110";
17         ROM(2) <= "000101";
18         ROM(3) <= "001000";
19         ROM(4) <= "110001";
20         ROM(5) <= "000000";
21         Data(5 downto 0) <= ROM(to_integer(unsigned(Address)));
22         data(6) <= '0';
23     end Memory;
```

حال باید تست بگیریم:



مشاهده میشود که به درستی عمل میکند.

اسمبلر، بخش نمره اضافی:


```

1 header = "library IEEE;\nuse IEEE.STD_LOGIC_1164.all;\nuse IEEE.STD_LOGIC_unsigned.all;\nuse ieee.numeric_std.all;\n\nfooter = "Data <= ROM(to_integer(unsigned(Address)));\nend Memory;"
2
3
4 reg_dic = {"R0": "00", "R1": "01", "R2": "10", "R3": "11", }
5 def getIn6bitBinaryFormat(Integer: int):
6     binary = bin(Integer)[2:]
7     while(len(binary) < 6): binary = "0" + binary
8     return binary
9 def getCodeInBinaryFormat(file_path):
10    output = []
11    try:
12        f = open(file_path, "r")
13    except:
14        print("File not found")
15    try:
16        for line in f:
17
18            parts = line.upper().replace(" ", "").split()
19            if parts[0] == "LOAD":
20                output.append( "00" + reg_dic[parts[1]] + "01")
21                output.append(getIn6bitBinaryFormat(int(parts[2])))
22            elif parts[0] == "ADD":
23                output.append("01" + reg_dic[parts[1]] + reg_dic[parts[2]])
24            elif parts[0] == "SUB":
25                output.append("10" + reg_dic[parts[1]] + reg_dic[parts[2]])
26            elif parts[0] == "JNZ":
27                output.append("11" + reg_dic[parts[1]] + "00")
28                output.append(getIn6bitBinaryFormat(int(parts[2])))
29            elif parts[0] == "HLT":
30                output.append("000000")
31            else:
32                print("Invalid OP code: " + parts[0] + "")
33                return ""
34    except:
35        print("Syntax error")
36    f.close()
37    return output
38
39
40 def getVHDLCode(binaryCode):
41     output = header
42     for i, code in enumerate(binaryCode):
43         output += "\nROM(" + str(i) + ") <= \"\" + code + "\";"
44     return output + "\n" + footer
45
46 def Main(_argv):
47     code = getVHDLCode(getCodeInBinaryFormat(_argv[0]))
48
49     f = open("../Design1_LOAD_SUM_SUB_JNZ/src/Memory.vhd", "w")
50     f.write(code)
51     f.close()
52
53 if __name__ == "__main__":
54     import sys
55     Main(sys.argv[1:])
56

```

دو بخش هدر و فوتر، نوشتن قالب ثابت فایل رام هستند که تغییر نمیکنند. یعنی بخش های مشخص شده را مینویسند:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3  use IEEE.STD_LOGIC_unsigned.all;
4  use ieee.numeric_std.all;
5  entity Memory is
6      port(
7          Address : in std_logic_vector(6 downto 0);
8          Data : out std_logic_vector(6 downto 0)
9      );
10 end Memory;
11 architecture Memory of Memory is
12     type mem_type is array(0 to 63) of std_logic_vector(6 downto 0);
13     signal ROM : mem_type;
14     begin
15         ROM(0) <= "0000001";
16         ROM(1) <= "0000110";
17         ROM(2) <= "0000101";
18         ROM(3) <= "0001000";
19         ROM(4) <= "1000001";
20         ROM(5) <= "0000000";
21         Data <= ROM(to_integer(unsigned(Address)));
22     end Memory;
    
```

اصل کار بخش میانی است که باید برنامه نویسی شود

برای دستور LOAD، ابتدا opcode=00، سپس رجیستر مبدأ، و دو رقم آخر 01 طراحی میشود. در خانه بعدی حافظه نیز، مقداری که باید لود شود نوشته میشود:

```

if parts[0] == "LOAD":
    output.append("00" + reg_dic[parts[1]] + "01")
    output.append(getIn6bitBinaryFormat(int(parts[2])))
    
```

برای دستور ADD، ابتدا opcode = 01، سپس رجیستر مبدأ و مقصد نوشته میشوند:

```

elif parts[0] == "ADD":
    output.append("01" + reg_dic[parts[1]] + reg_dic[parts[2]])
    
```

برای دستور SUB، ابتدا opcode = 10، سپس رجیستر مبدأ و مقصد نوشته میشوند:

```

elif parts[0] == "SUB":
    output.append("10" + reg_dic[parts[1]] + reg_dic[parts[2]])
    
```

برای دستور JNZ، ابتدا opcode = 11، سپس رجیستر مبدأ، و دو رقم راست 00 طراحی میشوند، در خط بعدی نیز آدرس پرش نوشته میشود:

```
elif parts[0] == "JNZ":  
    output.append("11" + reg_dic[parts[1]] + "00")  
    output.append(getIn6bitBinaryFormat(int(parts[2])))
```

دستور HLT با کد ۰۰۰۰۰۰ نمایش داده میشود:

```
elif parts[0] == "HLT":  
    output.append("000000")
```

اسمبلر بخش ۳ از design2: مشابه ۱ است ولی opcode ها تغییر کردند.

```
C: > Users > taban > Downloads > Telegram Desktop > Design2_Assembler (4).py > header
1 header = "library IEEE;\nuse IEEE.STD_LOGIC_1164.all;\nuse IEEE.STD_LOGIC_unsigned.all;\nuse ieee.numeric_
2 footer = "Data(5 downto 0) <= ROM(to_integer(unsigned(Address)));          \ndata(6) <= '0';\nend Memory;"
3
4 reg_dic = {"R0": "00", "R1": "01", "R2": "10", "R3": "11", }
5 def getIn6bitBinaryFormat(Integer: int):
6     binary = bin(Integer)[2:]
7     while(len(binary) < 6): binary = "0" + binary
8     return binary
9 def getCodeInBinaryFormat(file_path):
10    output = []
11    try:
12        f = open(file_path, "r")
13    except:
14        print("File not found")
15    try:
16        for line in f:
17
18            parts = line.upper().replace(" ", "").split()
19            if parts[0] == "LOAD":
20                output.append( "00" + reg_dic[parts[1]] + "01")
21                output.append(getIn6bitBinaryFormat(int(parts[2])))
22            elif parts[0] == "JNZ":
23                output.append("00" + reg_dic[parts[1]] + "10")
24                output.append(getIn6bitBinaryFormat(int(parts[2])))
25            elif parts[0] == "ADD":
26                output.append("01" + reg_dic[parts[1]] + reg_dic[parts[2]])
27            elif parts[0] == "SUB":
28                output.append("10" + reg_dic[parts[1]] + reg_dic[parts[2]])
29            elif parts[0] == "MULT":
30                output.append("11" + reg_dic[parts[1]] + reg_dic[parts[2]])
31            elif parts[0] == "HLT":
32                output.append("000000")
33            else:
34                print("Invalid OP code: " + parts[0] + "")
35                return ""
36    except:
37        print("Syntax error")
38    f.close()
39    return output
```



```

40
41
42 ∨ def getVHDLCode(binaryCode):
43     output = header
44     for i, code in enumerate(binaryCode):
45         output += "\nROM(" + str(i) + ") <= \" + code + "\";"
46     return output + "\n" + footer
47
48 ∨ def Main(_argv):
49     code = getVHDLCode(getCodeInBinaryFormat(_argv[0]))
50
51     f = open("../Design2_LOAD_SUM_SUB_JNZ_MULT/src/Memory.vhd", "w")
52     f.write(code)
53     f.close()
54
55 ∨ if __name__ == "__main__":
56     import sys
57     Main(sys.argv[1:])
58

```

برای دستور LOAD، ابتدا opcode=00، سپس رجیستر مبدأ، و دو رقم آخر 01 طراحی میشود. در خانه بعدی حافظه نیز، مقداری که باید لود شود نوشته میشود:

```

if parts[0] == "LOAD":
    output.append("00" + reg_dic[parts[1]] + "01")
    output.append(getIn6bitBinaryFormat(int(parts[2])))

```

برای دستور ADD، ابتدا opcode=01، سپس رجیستر مبدأ و مقصد نوشته میشوند:

```

elif parts[0] == "ADD":
    output.append("01" + reg_dic[parts[1]] + reg_dic[parts[2]])

```

برای دستور SUB، ابتدا opcode = 10، سپس رجیستر مبدأ و مقصد نوشته میشوند:

```

elif parts[0] == "SUB":
    output.append("10" + reg_dic[parts[1]] + reg_dic[parts[2]])

```

برای دستور JNZ، ابتدا opcode = 00، سپس رجیستر مبدأ، و دو رقم راست ۱۰ طراحی میشوند، در خط بعدی نیز آدرس پرش نوشته میشود:

```

elif parts[0] == "JNZ":
    output.append("00" + reg_dic[parts[1]] + "10")
    output.append(getIn6bitBinaryFormat(int(parts[2])))

```

دستور HLT با کد ۰۰۰۰۰۰ نمایش داده میشود:

```
elif parts[0] == "HLT":
    output.append("000000")
```

برای دستور MULT، ابتدا opcode = 11، سپس رجیسترهای مبدا و مقصد نوشته میشوند.

```
elif parts[0] == "MULT":
    output.append("11" + reg_dic[parts[1]] + reg_dic[parts[2]])
```