به نام خدا

نام استاد : دکتر محمد صالحی

موضوع : گزارش پروژه نهایی درس هم طراحی

نویسنده : سحر فخریه کاشان

در این پروژه، پردازنده 6 بیتی که در کلاس طراحي شده را پیادهسازي کرده و براي آن برنامهنويسي مي‌کنيم.

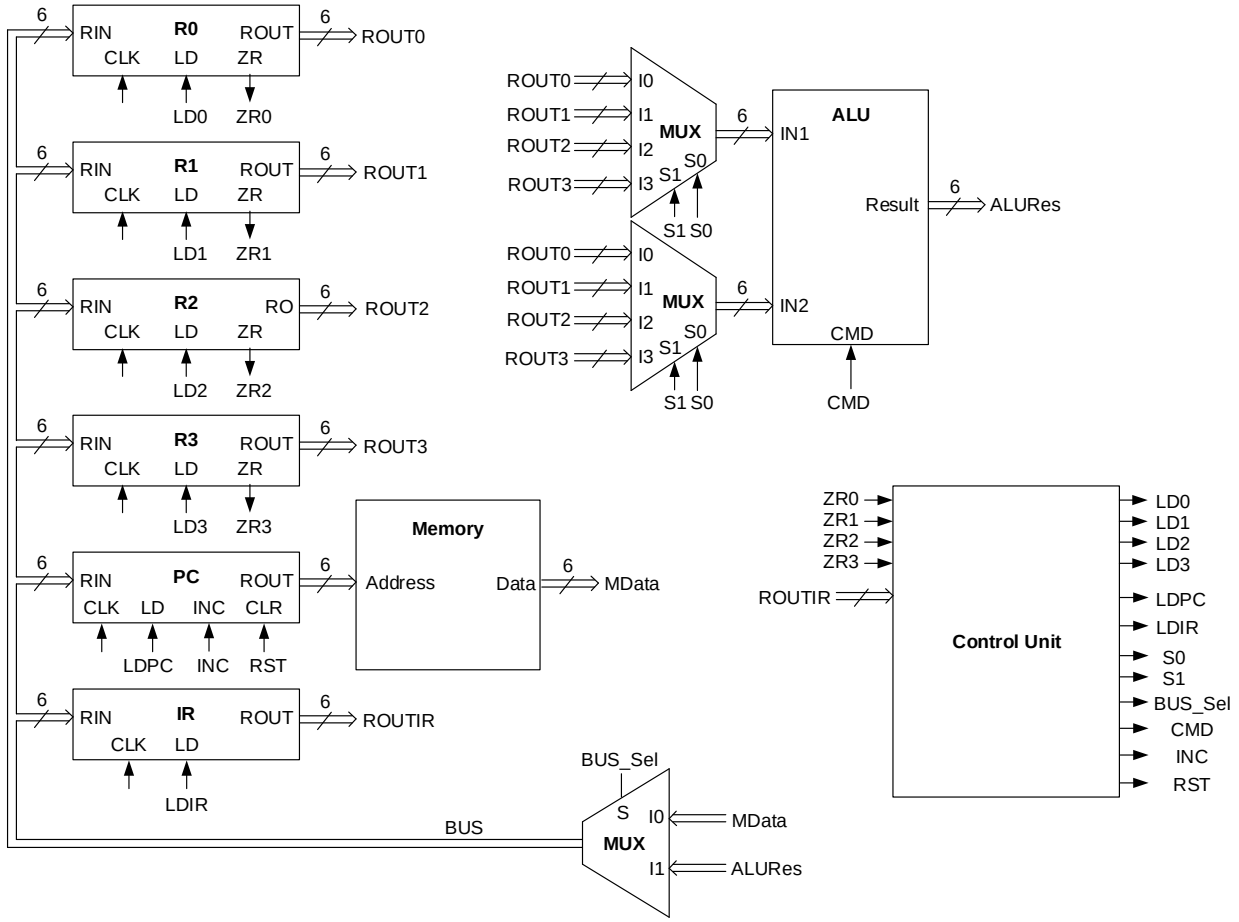
توجه: اين پروژه در صورتي قابل قبول است که براي آن گزارش نوشته شود. در اين گزارش نحوه پيادهسازي پردازنده و اجراي برنامه توسط آن با استفاده از عکس‌هاي مناسب از خروجي شبيه‌سازي نشان داده شود.

بخش اول: براي انجام اين پروژه ابتدا پردازنده را با استفاده از VHDL يا Verilog پيادهسازي كرده و صحت عملكرد آن را با اجراي كد زير كه دو عدد 5 و 3 را با هم جمع مي‌كند بررسي كنيد (**40% نمره پروژه**).

```
LOAD R0, 5
LOAD R1, 3
ADD R0, R1
```

بخش دوم: با توجه به اين‌كه اين پردازنده دستور ضرب ندارد، عمل ضرب را با استفاده از عمل جمع و به صورت نرم‌افزاري پيادهسازي كرده و صحت عملكرد آن را با يك مثال نشان دهيد (مشابه بخش اول يک كد اسمبلی بنويسيد كه عمل ضرب را انجام دهد). به عنوان مثال، حاصلضرب عدد 7 در 8 را حساب كند. (**60% نمره پروژه**).

معماري پردازنده:



دستورات پردازنده:

اين پردازنده چهار دستور LOAD، ADD، SUB و JNZ با كد دستور (Op Code) زير است:

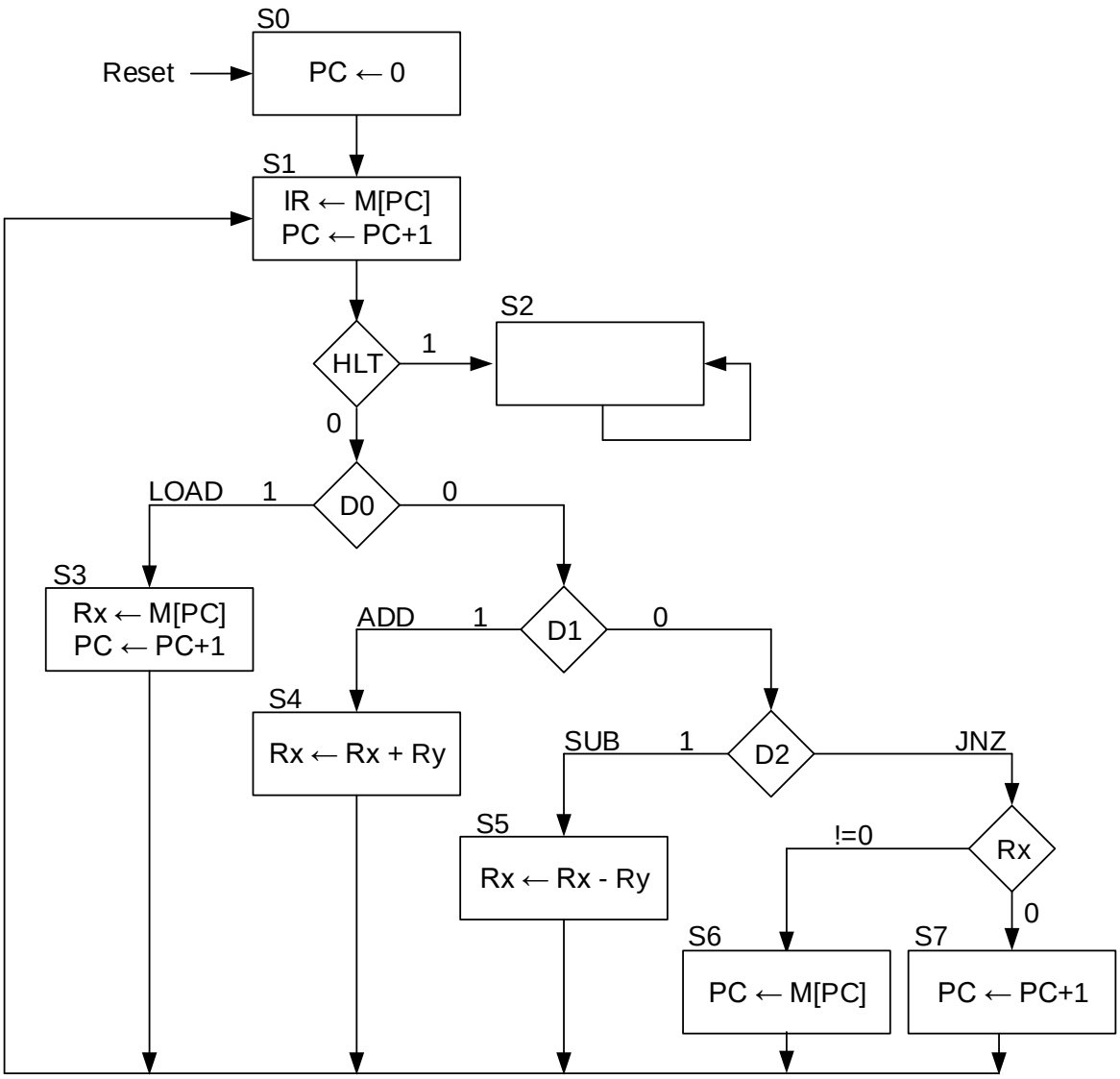
دستور	كد دستور
LOAD	00
ADD	01
SUB	10
JNZ	11

قالب دستورات:

Op Code	R_{SRC}	R_{DST}

چینش در حافظه	RTL	اسمبلی دستور
PC → 00 Rx 00 مقدار دستور بعدی	Rx ← M[PC]	**LOAD Rx, VALUE**
PC → 01 Rx Ry دستور بعدی	Rx ← Rx + Ry	**ADD Rx, Ry**
PC → 10 Rx Ry دستور بعدی	Rx ← Rx - Ry	**SUB Rx, Ry**
PC → 11 Rx 00 آدرس پرش دستور بعدی	If (Rx != 0) PC ← M[PC] else PC ← PC + 1	**JNZ Rx, Address**

چارت ASM براي طراحي واحد كنترل:



كد پايتون اسمبلر:

كد:

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QFileDialog,
```

```
QMessageBox


class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        path = QFileDialog.getOpenFileName(self, 'Open a file', '',
                                           'All Files (*.*)')
        if path != ('', ''):
            global inputFile
            inputFile = path[0]


app = QApplication(sys.argv)
window = MainWindow()

file1 = open(inputFile, "r")
commands = file1.readlines()
line_len = len(commands)

LineSplit = []

for i in range(line_len):
    LineSplit = commands[i].split(' ')


    if LineSplit[0] == "add":
        print("01" + '{0:02b}'.format(int(LineSplit[1][1])) +
'{0:02b}'.format(int(LineSplit[2][1])))

    elif LineSplit[0] == "jnz":
        print("11" + '{0:02b}'.format(int(LineSplit[1][1])) +
'{0:02b}'.format(0))
        print('{0:06b}'.format(int(LineSplit[2])))

    elif LineSplit[0] == "sub":
        print("10" + '{0:02b}'.format(int(LineSplit[1][1])) +
'{0:02b}'.format(int(LineSplit[2][1])))

    elif LineSplit[0] == "load":
        print("00", end='')
        if LineSplit[1] == "R0,":
            print("0000")
            print('{0:06b}'.format(int(LineSplit[2])))
```

```
        elif LineSplit[1] == "R1,":
            print("0100")
            print('{0:06b}'.format(int(LineSplit[2])))
        elif LineSplit[1] == "R2,":
            print("1000")
            print('{0:06b}'.format(int(LineSplit[2])))
        elif LineSplit[1] == "R3,":
            print("1100")
            print('{0:06b}'.format(int(LineSplit[2])))
    else:
        print("111111")


file1.close()
```

ورودی، یک فایل txt. به صورت زیر است :

load R0, 5

load R1, 3

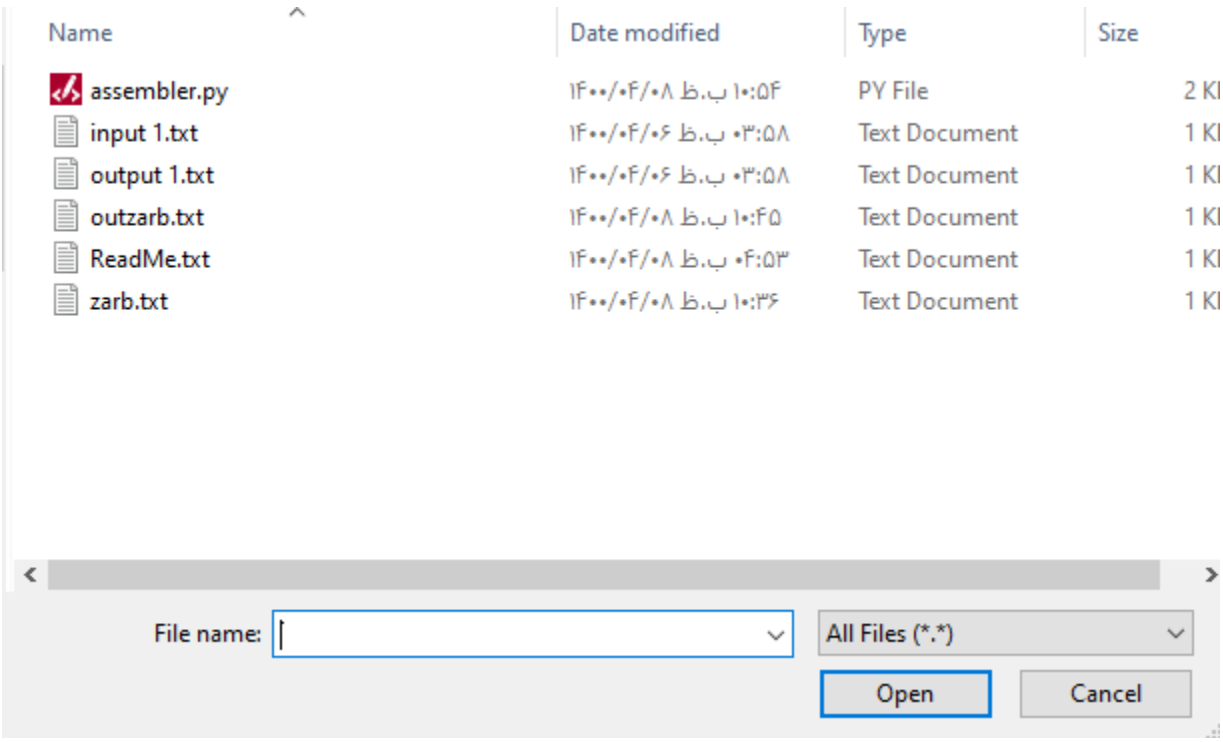
add R0, R1

خروجی :

```
000000
000101
000100
000011
010001

Process finished with exit code 0
```

توضیحات کد :

ابتدا یک file browser به صورت زیر باز میشود و از ما میخواهد تا فایل ورودی txt. که در آن کد های اسمبلی نوشته شده است را انتخاب کنیم و سپس آن را تبدیل به کد باینری میکنیم که میتوان آن را درون حافظه ROM قرار داد



سپس یک حلقه به اندازه تعداد خطوط فایل txt. میزنیم

بعد هر دستور را از جایی که فاصله دارند جدا میکنیم در آرایه ایجاد شده خانه اول که شامل دستور مورد نظر است را بررسی میکنیم و با توجه به دستور عملیات لازم را انجام میدهیم

دستورات پردازنده

- قالب دستورات

اسمبلی دستور	RTL	چینش در حافظه
LOAD Rx, VALUE	Rx ← M[PC]	PC → 00 Rx 00 / مقدار / دستور بعدی
ADD Rx, Ry	Rx ← Rx + Ry	PC → 01 Rx Ry / دستور بعدی
SUB Rx, Ry	Rx ← Rx - Ry	PC → 10 Rx Ry / دستور بعدی
JNZ Rx, Address	If (Rx != 0) PC ← M[PC] else PC ← PC + 1	PC → 11 Rx 00 / آدرس پرش / دستور بعدی

اگر دستور برابر با **add** شروع شده بود $01+R_x+R_y$ چاپ خواهد شد که R_x و R_y دو عدد دو رقمی باینری هستند برای همین از کد :

```
'{0:02b}'.format(int(LineSplit[1][1]))
```

استفاده میکنیم و متغیر دوم آرایه جدا شده را به آن پاس میدهیم.

اگر دستور برابر با **jnz** شروع شده بود ، $11+R_x+00$ چاپ خواهد شد و سپس در ادامه آدرس گرفته میشود که باید تبدیل به یک عدد شش رقمی باینری شود برای همین از کد :

```
print('{0:06b}'.format(int(LineSplit[2])))
```

استفاده میکنیم و متغیر دوم آرایه جدا شده را به آن پاس میدهیم.

اگر دستور برابر با **sub** شروع شده بود ، $10+R_x+R_y$ چاپ خواهد شد که R_x و R_y دو عدد دو رقمی باینری هستند برای همین از کد :

```
'{0:02b}'.format(int(LineSplit[1][1]))
```

استفاده میکنیم و متغیر دوم آرایه جدا شده را به آن پاس میدهیم.

اگر دستور برابر با **load** شروع شده بود ، $00+R_x+00$ چاپ خواهد شد که در اینجا کنترل خواهد شد که مقدار R_x برابر با کدام رجسیتر است

اگر این متغیر برابر با R0 بود مقدار 00 ، اگر R1 بود مقدار 01 ، اگر R2 بود مقدار 10 ، اگر R3 بود مقدار 11 به جای R_x گذاشته میشود.

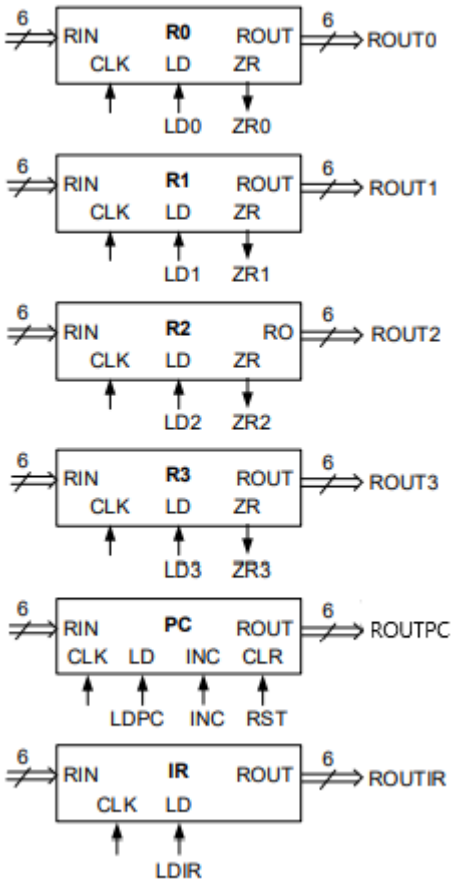
سپس در خط بعدی مقدار value گرفته شده از متغیر دوم آرایه جدا شده تبدیل به یک عدد شش رقمی باینری میشود با استفاده از این کد :

```
print('{0:06b}'.format(int(LineSplit[2])))
```

اگر دستور هیچ کدام از موارد بالا نبود مقدار ۱۱۱۱۱۱ به ازای halt چاپ خواهد شد

کد VHDL :

ثبات های پردازنده:



کد:

ثبات اصلی:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
entity REGS is
    PORT(   RIN : in STD_LOGIC_VECTOR(5 downto 0);
            ROUT0,ROUT1,ROUT2,ROUT3 : out STD_LOGIC_VECTOR(5 downto 0);
            LD0,LD1,LD2,LD3, CLOCK : in std_logic;
            ZR0,ZR1,ZR2,ZR3 : out std_logic);
end REGS;
architecture REGS of REGS is
signal reg0,reg1,reg2,reg3 : std_logic_vector(5 downto 0) := "000000" ;
begin
Process(CLOCK)
     begin
        if Rising_edge(CLOCK) then
            if(LD0 = '1')then reg0 <= RIN;
            end if;
        end if;
end process;
Process(CLOCK)
     begin
        if Rising_edge(CLOCK) then
            if(LD1 = '1')then reg1 <= RIN;
            end if;
        end if;
end process;
Process(CLOCK)
     begin
        if Rising_edge(CLOCK) then
            if(LD2 = '1')then reg2 <= RIN;
            end if;
        end if;
end process;
Process(CLOCK)
     begin
        if Rising_edge(CLOCK) then
            if(LD3 = '1')then reg3 <= RIN;
            end if;
        end if;
end process;
ROUT0 <= reg0;
ROUT1 <= reg1;
ROUT2 <= reg2;
ROUT3 <= reg3;
ZR0 <= '1' when reg0 = "000000" else '0' ;
```

```
ZR1 <= '1' when reg1 = "000000" else '0' ;
ZR2 <= '1' when reg2 = "000000" else '0' ;
ZR3 <= '1' when reg3 = "000000" else '0' ;
end REGS;
```

Pc:

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity PC is
    PORT( RIN : in std_logic_vector(5 downto 0);
          ROUT : out std_Logic_vector( 5 downto 0);
          CLOCK, LD, INC, CLR : in std_logic);
end PC;
architecture Behavioral of PC is
signal regs : std_logic_vector( 5 downto 0);
begin
    Process( CLOCK, CLR)
    variable reg : std_logic_vector( 5 downto 0) := "000000";
    begin
        if( CLR = '1') then reg := "000000";
        elsif( Rising_edge(CLOCK)) then
            if( LD = '1') then reg := RIN;
            end if;
            if ( INC = '1') then reg := reg + 1;
            end if;
        end if;
        regs <= reg;
    end process;
ROUT <= regs;
end Behavioral;
```

IR:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity IR is
    PORT(    RIN : in STD_LOGIC_VECTOR(5 downto 0);
             ROUT : out STD_LOGIC_VECTOR(5 downto 0);
             LD, CLOCK : in std_logic);
end IR;
architecture Behavioral of IR is
signal reg : std_logic_vector(5 downto 0);
begin
    Process(CLOCK)
     begin
        if Rising_edge(CLOCK) then
            if(LD = '1')then reg <= RIN;
            end if;
        end if;
    end process;
ROUT <= reg;
end Behavioral;
```

٤ ثبات اصلی ٦ بیتی :

یک ورودی ۶ بیتی پردازنده دارد به نام RIN و یک خروجی ۶ بیتی دیتا به نام ROUT دارد یک کلاک دارد، یک ورودی کنترلی به نام LD دارد که وقتی مقدارش برابر ۱ باشد مقدار ورودی را در سر بالارونده کلاک ذخیره میکند

یک مقدار ZR داریم که وقتی مقدار رجیستر صفر باشد در خروجی zero عدد یک را مینویسد

ثبات دستور ٦ بیتی :

یک ورودی ۶ بیتی پردازنده دارد به نام RIN و یک خروجی ۶ بیتی دیتا به نام ROUT دارد یک کلاک دارد، یک ورودی کنترلی به نام LD دارد که وقتی مقدارش برابر ۱ باشد مقدار ورودی را در سر بالارونده کلاک ذخیره میکند

ثبات شمارنده برنامه :

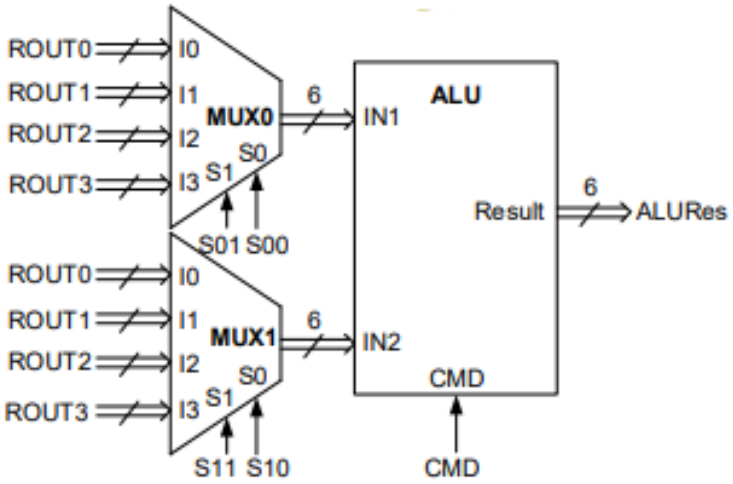
یک ورودی ۶ بیتی پردازنده دارد به نام RIN و یک خروجی ۶ بیتی دیتا به نام ROUT دارد یک کلاک دارد، یک ورودی کنترلی به نام LD دارد که وقتی ما میخواهیم program couter را با یک مقدار جدیدی پر کنیم از این متغیر استفاده میکنیم

ورودی INC داریم که هر بار دستور اجرا میشود به صورت پیش فرض program couter یک دانه افزایش پیدا میکند تا به دستور بعدی اشاره کند

و در نهایت ورودی CLR دارد که وقتی سیستم reset میشود ورودی CLR فعال میشود و مقدار program couter صفر میشد تا به خانه اول حافظه اشاره کند و برنامه از اول اجرا شود

در شکل بالا به ترتیب ۴ ثبات اول، ثبات های اصلی ، ثبات پنجم ثبات شمارنده برنامه و ثبات اخر ثبات دستور است

واحد محاسبه و منطق (ALU):



کد:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ALU is
    PORT( IN1, IN2 : in std_logic_vector(5 downto 0);
          Result : out std_logic_vector(5 downto 0);
          CMD : in std_logic);
end ALU;
architecture Behavioral of ALU is
begin
process( CMD, IN1, IN2)
begin
    if( CMD = '0')then Result <= (IN1 - IN2);
    else Result <= (IN1 + IN2);
    end if;
end process;
end Behavioral;
```

توضیحات:

دو دستور محاسباتی جمع و تفریق را انجام میدهد و اگر دستورات دیگری هم وجود داشت داخل ALU پیاده سازی میشود.

دارای دو ورودی ۶ بیتی IN1 و IN2 هست و یک خروجی ۶ بیتی Result دارد که به bus contoroller وصل میشود که مشخص میکند این خروجی را کدام یکی از رجیستر استفاده کند

در نهایت یک ورودی تک بیتی با نام CMD دارد که اگر صفر باشد ALU ، دستور جمع و اگر ۱ باشد دستور تفریق را انجام دهد و اگر بخواهیم تعداد دستورات ALU افزایش یابد باید تعداد بیت های CMD را افزایش دهیم

هر کدام از ورودی های ALU میتواند از هر کدام از رجیستر ها بیاید

MUX ها :

کد:

```
Mux1 : process( sel1,sel0,Rout0,Rout1,Rout2,Rout3 )
begin
    if(sel1 = '0' and sel0 = '0')then IN1 <= Rout0;
    elsif(sel1 = '0' and sel0 = '1')then IN1 <= Rout1;
    elsif(sel1 = '1' and sel0 = '0')then IN1 <= Rout2;
    else IN1 <= Rout3;
    end if;
end process;
Mux2 : process( sel2,sel3,Rout0,Rout1,Rout2,Rout3 )
begin
    if(sel3 = '0' and sel2 = '0')then IN2 <= Rout0;
    elsif(sel3 = '0' and sel2 = '1')then IN2 <= Rout1;
    elsif(sel3 = '1' and sel2 = '0')then IN2 <= Rout2;
    else IN2 <= Rout3;
    end if;
end process;
end Behavioral;
```

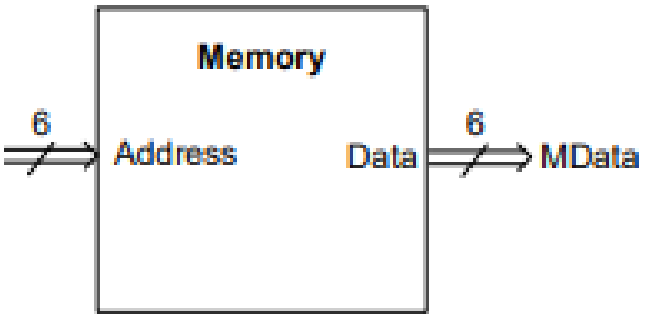
توضیحات:

ورودی اول و دوم ALU از یک مالتیپلکسر ۴ به ۱ ، ۶ بیتی که با استفاده از دو بیت select میتواند بگوید که کدام یک از ورودی های MUX به ورودی ALU وصل بشوند

معنی S_{00} این است که select 0 از مالتیپلکسر صفر و معنی S_{01} این است که select 1 از مالتیپلکسر صفر و معنی S_{10} این است که select 0 از مالتیپلکسر ۱ و معنی S_{11} این است که select 1 از مالتیپلکسر ۱

که این ۴ سیگنال select از control unit می‌آید که ما بسته به این که در این دو دستور از چه رجسیتر هایی استفاده کردیم این سیگنال ها مقداردهی میشوند.

حافظه ROM :



کد:

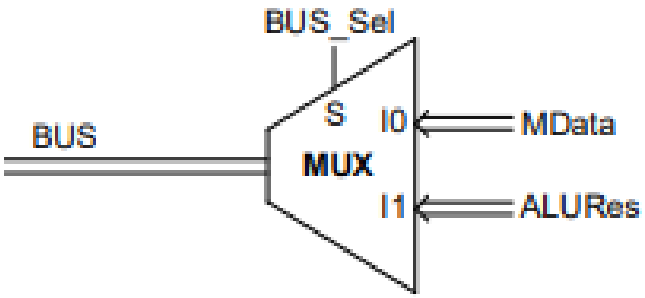
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
entity Memory is
    PORT( Addr : in std_logic_vector( 5 downto 0);
          Dout : out std_logic_vector( 5 downto 0);
          CLOCK : in std_logic);
end Memory;
architecture Behavioral of Memory is
TYPE Memory is array ( 63 downto 0) of std_logic_vector(5 downto 0);
Signal msignal : Memory;
begin
--msignal(0) <= "000000" ;
--msignal(1) <= "000101" ;
--msignal(2) <= "000100" ;
--msignal(3) <= "000011" ;
--msignal(4) <= "010001" ;
--msignal(5) <= "111111" ;
msignal(0)<="000000";
msignal(1)<="000000";
msignal(2)<="000100";
msignal(3)<="000111";
msignal(4)<="001000";

msignal(5)<="000001";
msignal(6)<="001100";
msignal(7)<="001000";
msignal(8)<="111100";
msignal(9)<="001011";
msignal(10)<="111111";
msignal(11)<="101110";
msignal(12)<="010001";
msignal(13)<="111100";
msignal(14)<="001001";
msignal(15)<="111111";
Dout <= msignal( to_integer(unsigned(Addr)));
end Behavioral;
```

توضیحات:

در این حافظه ما فقط از آن دیتا میخوانیم و هیچ دیتایی داخل آن نمیریزیم این حافظه ۶ بیت ورودی آدرس دارد بنابراین ما میتوانیم 2^6 خانه برای مموری داشته باشیم ۶ بیت خروجی دیتا دارد پس هر خانه از این مموری ۶ بیت است که برای خواندن دستورات و مقادیر از حافظه استفاده میشود

در این کد مقادیر اولیه کامنت شده کد باینری مربوط به جمع دو عدد ۳ و ۵ میباشد و ادامه مقادیر ، مقادیر دودویی مربوط به ضرب دو عدد ۷ و ۸ میباشد.

کنترل کننده گذرگاه (BUS CONTROLLER) :



کد:

```
Bus_selion : process( Bus_Sel,ALUres, MData )
begin
    if(Bus_Sel = '1')then Bus6 <= ALUres;
    else Bus6 <= MData;
    end if;
end process;
```

توضیحات :

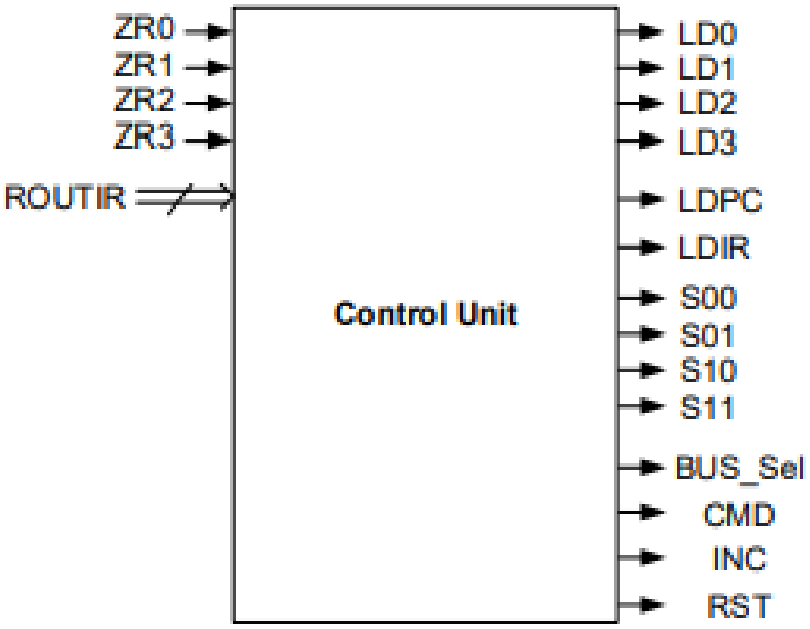
وظیفه اش این است که تعیین کند در هر لحظه چه مقداری بر روی باس نوشته شودکه خروجی باس به ورودی همه ی رجیستر های وصل است . در مموری و در دستور load مقداری از حافظه از طریق سیگنال MData به هرکدام از رجیستر های R0 تا R3 منتقل میشود

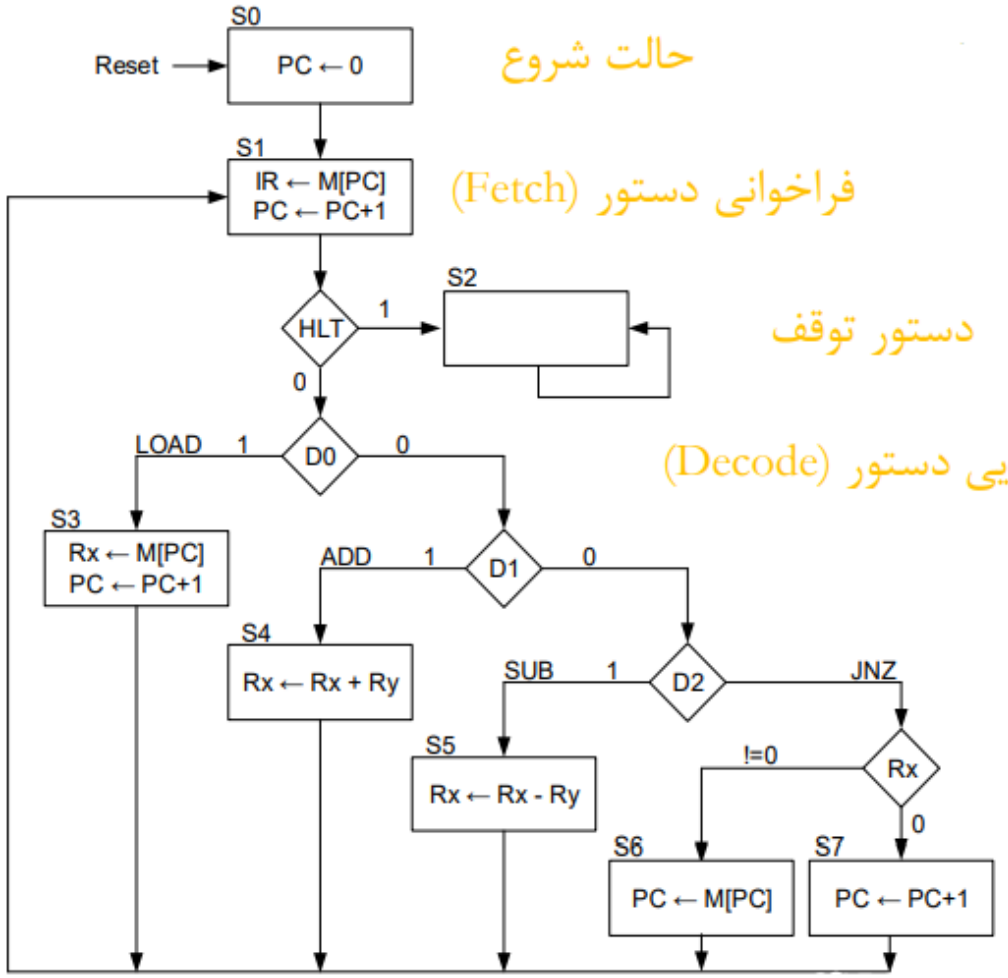
در دستور جمع یا تفریق خروجی ALU از طریق سیگنال ALURes به هرکدام از رجیستر های R0 تا R3 منتقل میشود

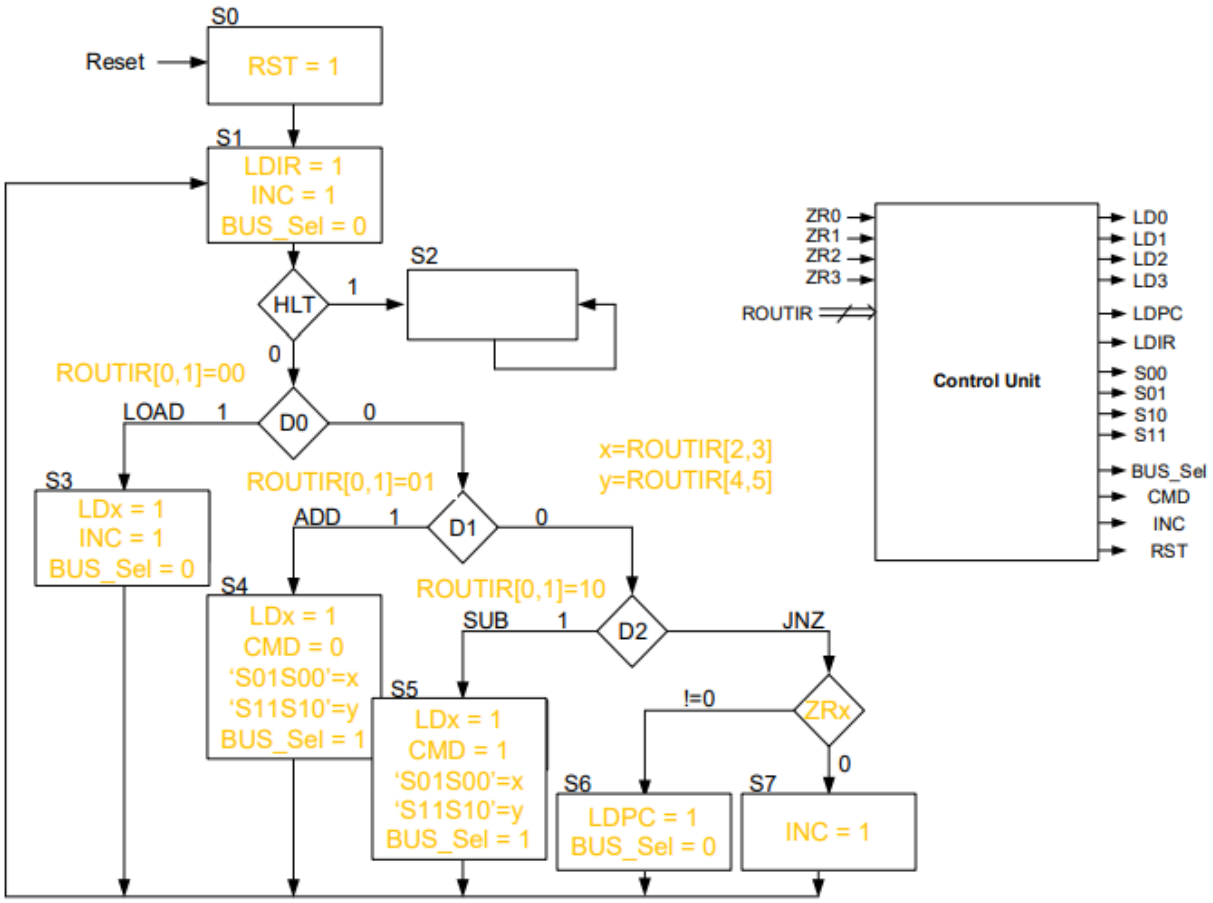
در دستور jump not zero اگر مقدار موجود در رجیستر صفر نبود برنامه به آدرس داده شده در دستور پرش میکند و از آن جا ادامه می دهد و هر بار دستور جدیدی خوانده میشود آن دستور از مموری به instruction reg منتقل میشود

برای تولید باس کنترلر از یک مالتیپلکسر ۲ به ۱، ۶ بیتی که با استفاده یک خط select میتواند انتخاب کند که MData یا ALURes به رجسیتر ها منتقل شود

واحد کنترل (control unit):




S0
Reset
PC ← 0
حالت شروع
S1
IR ← M[PC]
PC ← PC+1
فراخوانی دستور (Fetch)
HLT
1
S2
دستور توقف
0
LOAD
1
D0
0
رمزگشایی دستور (Decode)
S3
Rx ← M[PC]
PC ← PC+1
ADD
1
D1
0
S4
Rx ← Rx + Ry
SUB
1
D2
JNZ
S5
Rx ← Rx - Ry
!=0
Rx
0
S6
PC ← M[PC]
S7
PC ← PC+1



کد:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
entity CPU_With_Control is
PORT(  CLOCK , RST: in std_logic ;
       mem_content,out0, out1, out2, out3: out std_logic_vector(5 downto 0)) ;
end CPU_With_Control;
architecture Behavioral of CPU_With_Control is
signal LD0, LD1, LD2, LD3, LDPC, LDIR, sel0, sel1, sel2, sel3, Bus_sel, CMD, INC, CLR, ZR0, ZR1, ZR2, ZR3: std_logic := '0';
Signal index : integer;
Signal  addr, IR, Bus6, MData, Rout0, rout1, ROut2, Rout3, IN1, IN2, ALURes : std_logic_vector(5 downto 0):="000000";
TYPE States is (S0, S1,decision, S2, S3, S4, S5, S6, S7);
Signal PresentState: States;
Signal Z : std_logic_vector(3 downto 0);
begin
index <= to_integer(unsigned(IR(3 downto 2)));
Z(0) <= ZR0;Z(1) <= ZR1;Z(2) <= ZR2;Z(3) <= ZR3;
    process(CLOCK, RST)
    begin
       if( RST = '1') then
            PresentState <= S0;
            CLR <= '1';
```

```
        Bus_Sel <= '0';
        LDIR <='0';
        INC <= '0';
        LD0 <= '0';LD1 <= '0';LD2 <= '0';LD3 <= '0';
    elsif(Rising_edge(CLOCK))then
        Case PresentState is
        When S0 =>
            INC <= '1';
            PresentState <= S1;
            LDIR <= '1';
            Bus_Sel <= '0';
            LDPC <='0';
            LD0 <= '0';LD1 <= '0';LD2 <= '0';LD3 <= '0';
        When S1 =>
            CLR <= '0';
            PresentState <= decision;
            INC <= '0';
            LDIR <= '0';
        when decision =>
            if( IR = "111111")then PresentState <= S2;
            else
                if(IR(5 downto 4) = "00")then PresentState <= S3;

                elsif(IR(5 downto 4) = "01")then PresentState <= S4;
                elsif(IR(5 downto 4) = "10")then PresentState <= S5;
                else
                    if(Z(index) = '0')then PresentState <= S6;
                    else PresentState <= S7;
                    end if;
                end if;
            end if;
        when S2 =>
            PresentState <= S2;
        when S3 =>
            if(IR(3 downto 2) = "00")then LD0 <= '1';
            elsif(IR(3 downto 2) = "01")then LD1 <= '1';
            elsif(IR(3 downto 2) = "10")then LD2 <= '1';
            else LD3 <= '1';
            end if;
            INC <= '1';PresentState <= S0;
        when S4 =>
            sel1 <= IR(3);sel0 <= IR(2);sel3 <= IR(1);sel2 <= IR(0);
            if(IR(3 downto 2) = "00")then LD0 <= '1';
            elsif(IR(3 downto 2) = "01")then LD1 <= '1';
            elsif(IR(3 downto 2) = "10")then LD2 <= '1';
```

```
            elsif(IR(3 downto 2) = "11")then LD3 <= '1';
            end if;
            CMD <= '1';
            Bus_Sel <= '1';
            PresentState <= S0;
        when S5 =>
            sel1 <= IR(3);sel0 <= IR(2);sel3 <= IR(1);sel2 <= IR(0);
            if(IR(3 downto 2) = "00")then LD0 <= '1';
            elsif(IR(3 downto 2) = "01")then LD1 <= '1';
            elsif(IR(3 downto 2) = "10")then LD2 <= '1';
            elsif(IR(3 downto 2) = "11")then LD3 <= '1';
            end if;
            CMD <= '0';
            Bus_Sel <= '1';
            PresentState <= S0;
        when S6 =>
            LDPC <= '1';
            PresentState <= S0;
        when S7 =>
            INC <= '1';
```

```
            PresentState <= S0;
        end case;
    end if;
end process;
Memory : Entity work.Memory(Behavioral) port map( Addr => Addr,CLOCK => CLOCK,Dout => MData);
PC : Entity work.PC(Behavioral) port map( RIN => Bus6,ROUT => Addr,CLOCK => CLOCK,LD => LDPC,INC => INC,CLR => RST);
IR_inst : Entity work.IR(Behavioral) port map( RIN => Bus6,Rout => IR,LD => LDIR,CLOCK => CLOCK);
ALU : Entity work.ALU(Behavioral)port map( IN1 => IN1,IN2 => IN2,CMD => CMD,Result => ALURes);
Regs : Entity work.Regs(Regs) port map( RIN => Bus6,ROUT0 => Rout0,ROUT1 => Rout1,ROUT2 => Rout2,ROUT3 => Rout3,
    ZR0 => ZR0,ZR1 => ZR1,ZR2 => ZR2,ZR3 => ZR3,LD0 => LD0,LD1 => LD1,LD2 => LD2,LD3 => LD3,CLOCK => CLOCK);
out0<= Rout0;
out1<= Rout1;
out2<= Rout2;
out3<= Rout3;
mem_content <= MData;
```

توضیحات:

تمام عملیات پردازنده را مدیریت میکند و وظیفه آن خواندن مقادیر مورد نیاز از سیگنال ها و رجیستر ها و تولید سیگنال های کنترلی بخش های مختلف است.

ورودی های آن تمام ZR های ثبات های اصلی ما هستند که برای jump not zero استفاده میشوند تا ببینیم اگر مقدار پردازنده صفر نیست jump کنیم به یک بخش دیگر از حافظه و دستورات دیگری را اجرا کنیم

خروجی IR از طریق ROUTIR به واحد کنترل به عنوان ورودی وصل میشود و به کمک آن مقادیر تصمیم گیری انجام میشود که منجر به تولید سیگنال های مختلف برای بخش های مختلف پردازنده است

سیگنال های LD0 تا LD3 ورودی های لود سیگنال رجیسترهای اصلی ما است. سیگنال LDPC در jump not zero استفاده میشود. LDIR هر بار قرار باشد دستور جدیدی خوانده شود استفاده میشود

S00 تا S11، select های دو مالتیپلکسر متصل به ALU هستند

BUS_Sel که تعیین میکند مقدار MData یا ALURes به رجسیتر ها منتقل شود

CMD برای ALU که مشخص میکند اگر cmd برابر با صفر باشد عملیات جمع و اگر مقدارش با یک برابر بود تفریق را انجام دهد

سیگنال INC برای PC که تعیین میکند PC افزایش پیدا کند یا نه و در نهایت سیگنال RST که به عنوان ورودی CLR برای PC که هر وقت فعال شد PC از خانه صفر حافظه شروع به خواندن کند

عمل ضرب:

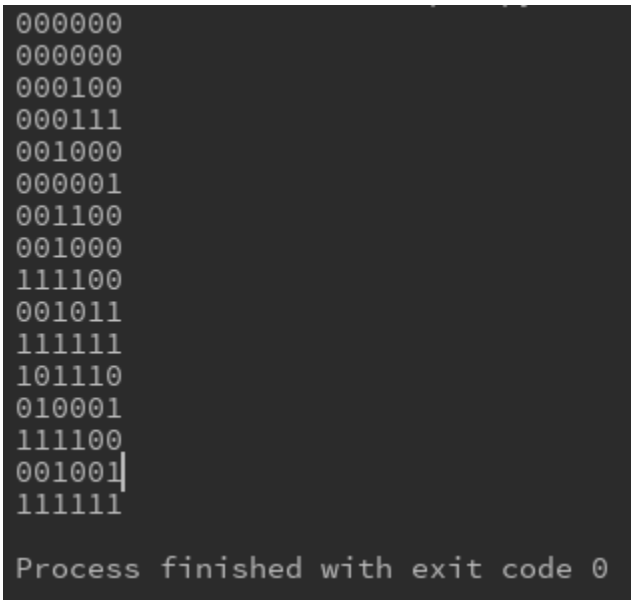
برای انجام عمل ضرب در ابتدای پروژه یک کد اسمبلی برای ضرب نوشته و آن را به عنوان ورودی به کد پایتونمان میدهیم و کد دودویی دریافتی را در خانه های حافظه ی پردازنده ی پیاده سازی شده قرار میدهیم:

کد اسمبلی ضرب :

```
load R0, 0
load R1, 7
load R2, 1
```

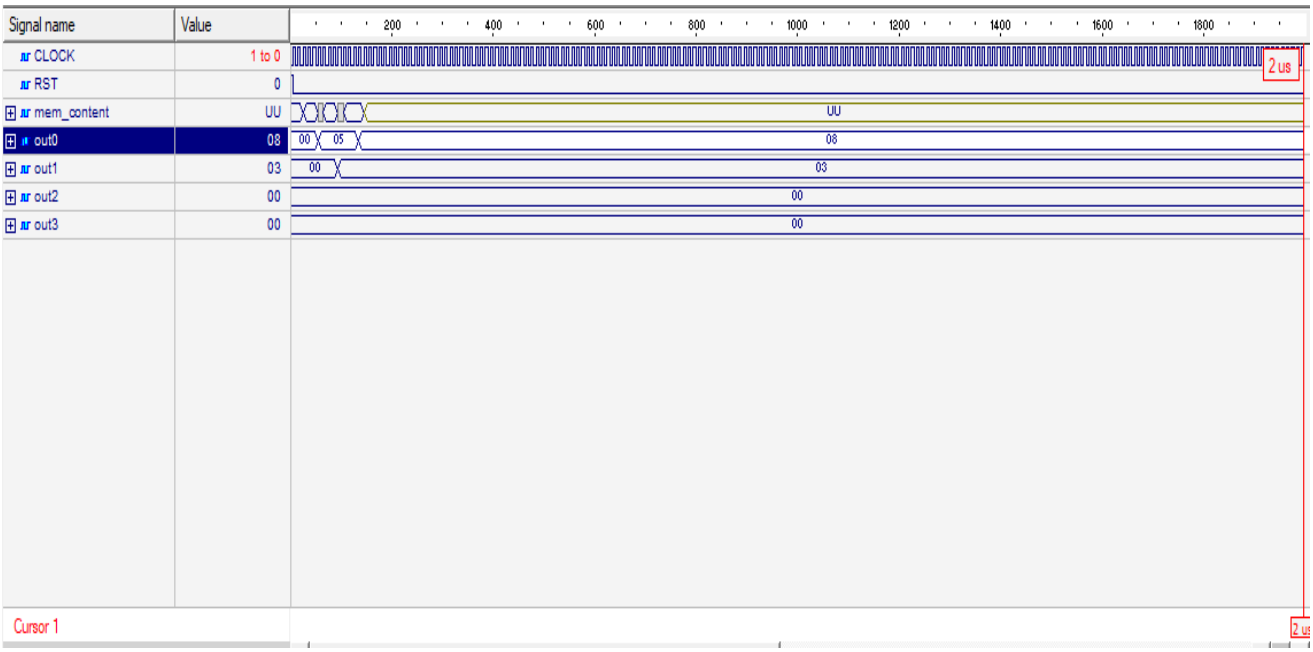
```
load R3, 8
jnz R3, 11
halt
sub R3, R2
add R0, R1
jnz R3, 9
halt
```

کد دو دویی ضرب :

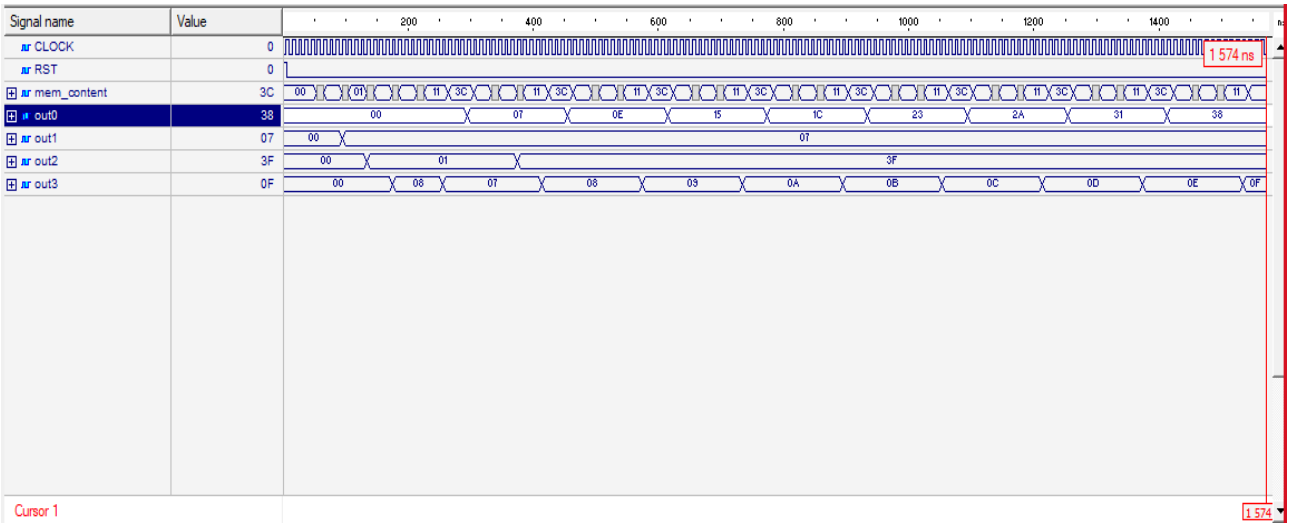


این کد باید به ما جواب ضرب دو عدد ۷ و ۸ را بدهد.

خروجی های کد :



در این عکس test bench عملیات جمع دو عدد ۳ و ۵ را میبینیم که به صورت عدد ۸ در رجیستر R0 ذخیره شده است.



در این عکس خروجی test bench عملیات ضرب دو عدد ۷ و ۸ را میبینیم که به صورت عدد ۳۸ (که مقدار هگز است و باینری آن ۵۶ میشود)

با تشکر