در اين پروژه، پردازنده 6 بيتی كه در كلاس طراحي شده را پيادەسازي كرده و براي آن برنامەنويسي مي‌كنيم.
توجه: اين پروژه در صورتي قابل قبول است كه براي آن گزارش هم نوشته شود. در اين گزارش نحوه پيادەسازي پردازنده و اجراي برنامه توسط آن با استفاده از عكس‌هاي مناسب از خروجي شبيه‌سازي نشان داده شود.

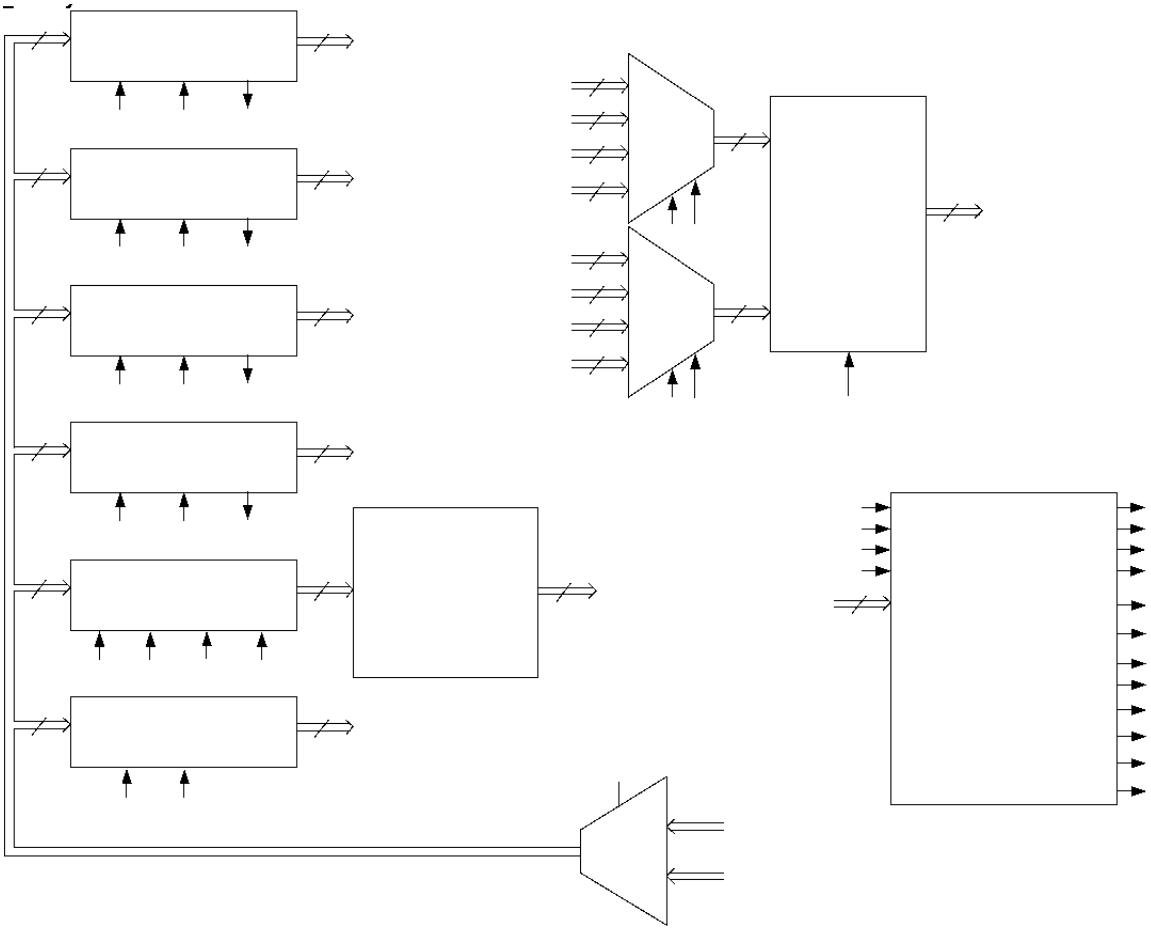
بخش اول (40% نمره پروژه): براي انجام اين پروژه ابتدا پردازنده را با استفاده از VHDL يا Verilog پيادەسازي كرده و صحت عملكرد آن را با اجراي كد زير كه دو عدد 7 و 4 را با هم جمع مي‌كند بررسي كنيد.

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

بخش دوم (20% نمره پروژه): با توجه به اين‌كه اين پردازنده دستور ضرب ندارد، عمل ضرب را با استفاده از عمل جمع و به صورت نرم‌افزاري پيادەسازي 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 8 در 6 را حساب كند.

بخش سوم (40% نمره پروژه): دستور ضرب را با كمترين سربار سخت‌افزاری به مجموعه دستورات اضافه كرده و صحت عملكرد آن را با نوشتن يک كد كه حاصلضرب 8 در 6 را حساب كند نشان دهيد. توجه كنيد كه برای اين كار نياز است تغييراتی در سخت‌افزار و كد دستورات ايجاد كنيد.

معماري پردازنده:



دستورات پردازنده:

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

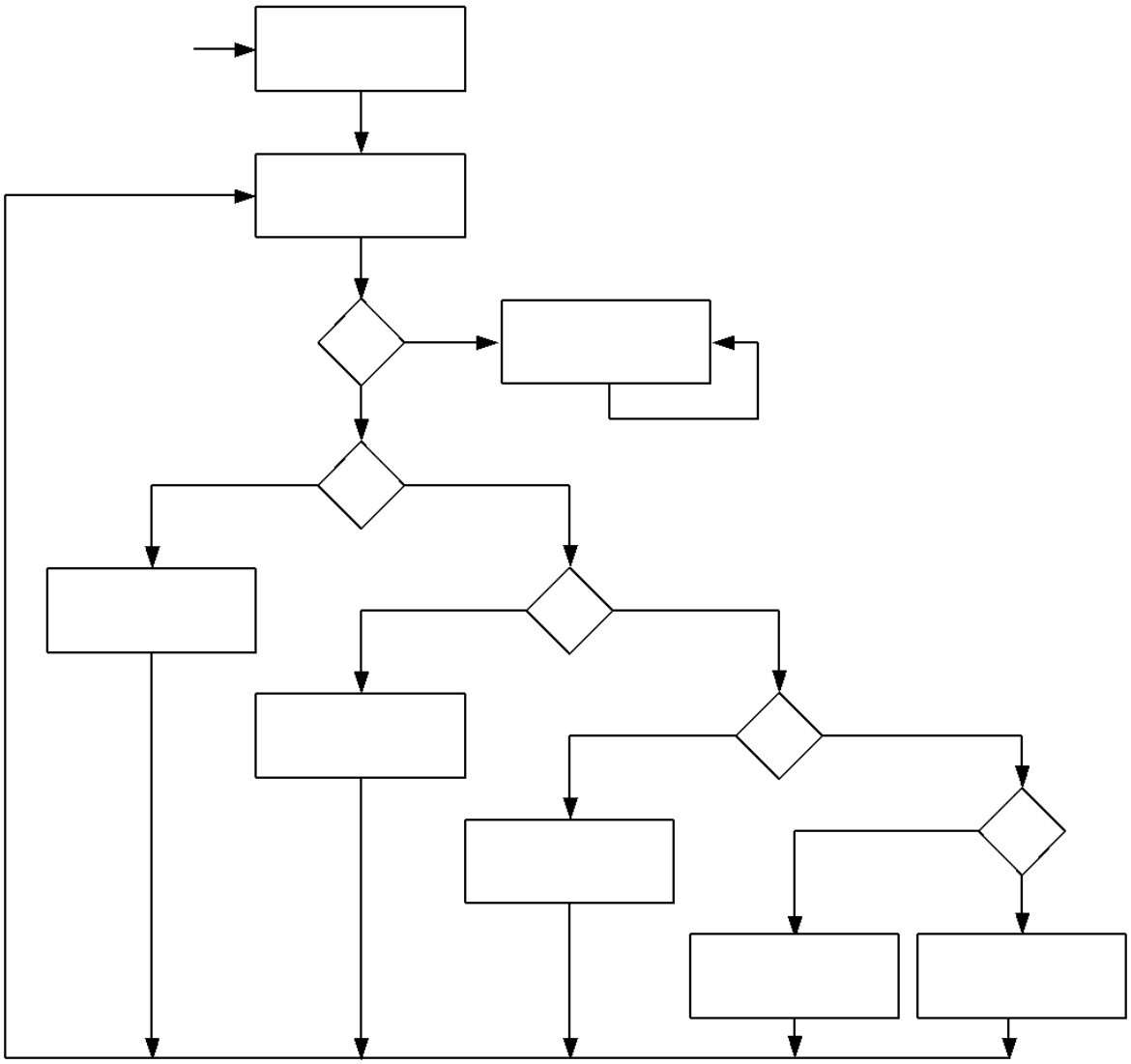
دستور	کد دستور
LOAD	00
ADD	01
SUB	10
JNZ	11

قالب دستورات:

Op Code	R_{SRC}	R_{DST}

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

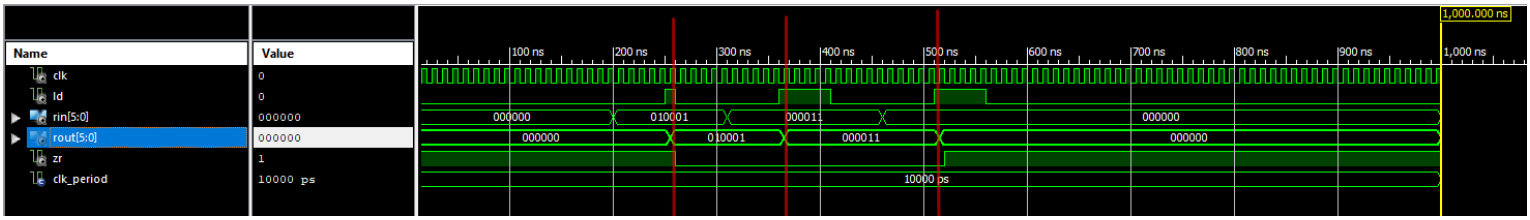
چارت ASM براي طراحي واحد كنترل:



رجیسترهای اصلی - REG

اگر به رفتار شبیه‌ساز دقت کنیم میبینیم که رجیستر دقیقا همان رفتاری را دارد که انتظار داریم. یعنی:

- سر لبه‌ی بالارونده‌ی کلاک اگر لود یک باشد، مقدار ورودی را ذخیره میکند.
- زمانی که مقدار ذخیره شده صفر است، مقدار ZR یک و زمانی که یک است، صفر می‌شود.

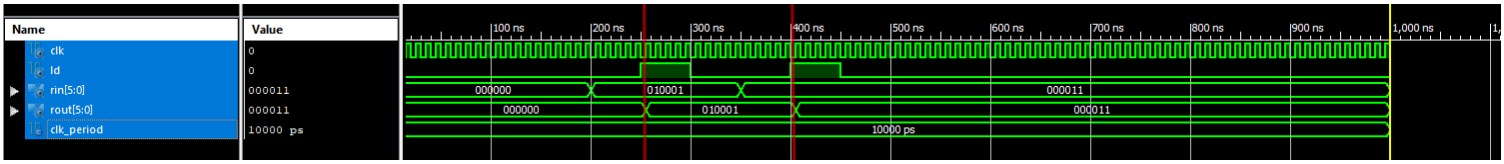



```
RIN <= "010001";
WAIT FOR Clk_period * 5;
LD <= '1';
WAIT FOR Clk_period;
LD <= '0';
WAIT FOR Clk_period * 5;

RIN <= "000011";
WAIT FOR Clk_period * 5;
LD <= '1';
WAIT FOR Clk_period * 5;
LD <= '0';
WAIT FOR Clk_period * 5;

RIN <= "000000";
WAIT FOR Clk_period * 5;
LD <= '1';
WAIT FOR Clk_period * 5;
LD <= '0';
WAIT FOR Clk_period * 5;

WAIT;
```

رجیستر دستور - IR

این رجیستر هم طبق انتظارمان عمل می‌کند. یعنی:

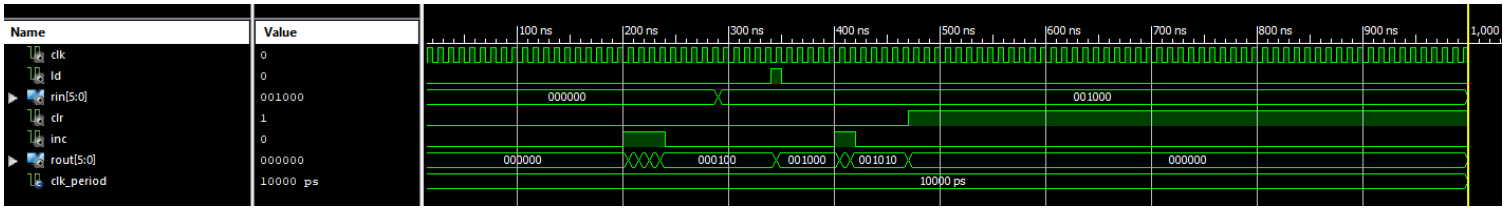
- سر لبه‌ی بالارونده‌ی کلاک اگر لود یک باشد، مقدار ورودی را ذخیره میکند.




```
RIN <= "010001";
WAIT FOR Clk_period * 5;
LD <= '1';
WAIT FOR Clk_period * 5;
LD <= '0';
WAIT FOR Clk_period * 5;

RIN <= "000011";
WAIT FOR Clk_period * 5;
LD <= '1';
WAIT FOR Clk_period * 5;
LD <= '0';
WAIT FOR Clk_period * 5;
```

رجیستر شمارنده برنامه - PC

اگر به رفتار خروجی در شبیه‌ساز نگاه کنیم می‌بینیم که کاملا به درستی عمل میکند. یعنی طبق این تست‌بنچ انتظار داشتیم که ابتدا با اعمال INC = 1 در چهار کلاک، مقدار را به ۴ برسانیم. سپس با دادن LD = 1 و عدد ۸، مقدار ۸ را لود کنیم. سپس مقدار لود شده را با ۲ کلاک اعمال INC = 1 به ۱۰ برسانیم و در نهایت رجیستر را با دادن CLR = 1 ریست کنیم. که میبینیم همه‌ی این اتفاقات به ترتیب رخ می‌دهند.




```
-- increase to 4 (000100)
INC <= '1';
WAIT FOR Clk_period;
WAIT FOR Clk_period;
WAIT FOR Clk_period;
WAIT FOR Clk_period;
INC <= '0';
WAIT FOR Clk_period * 5;

-- load 8 (001000)
RIN <= "001000";
WAIT FOR Clk_period * 5;
LD <= '1';
WAIT FOR Clk_period;
LD <= '0';
WAIT FOR Clk_period * 5;

-- increase to 10 (001010)
INC <= '1';
WAIT FOR Clk_period;
WAIT FOR Clk_period;
INC <= '0';
WAIT FOR Clk_period * 5;

-- clear
CLR <= '1';
WAIT FOR Clk_period * 5;

WAIT;
```

حافظه MEM - ROM

این بخش که یک حافظه Read Only می‌باشد هم طبق تست‌بنچ، مطابق انتظار ما عمل می‌کند.

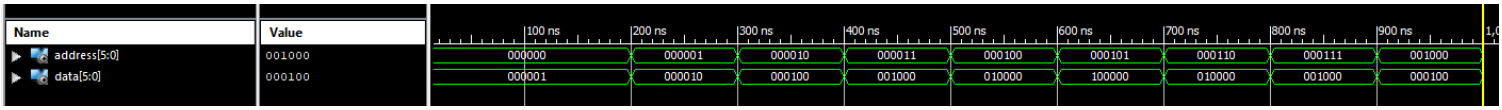
```
CONSTANT ROM : mem_array := (
    "000001",
    "000010",
    "000100",
    "001000",
    "010000",
    "100000",
    "010000",
    "001000",
    "000100",
    "000010",
    "000001",
    "000000",
    "000000",
    "000000",
    "000000"
```

یعنی اگر در حافظه مقادیر تصویر چپ را قرار دهیم،

```
ADDRESS <= "000000";
WAIT FOR 100 ns;
ADDRESS <= "000001";
WAIT FOR 100 ns;
ADDRESS <= "000010";
WAIT FOR 100 ns;
ADDRESS <= "000011";
WAIT FOR 100 ns;
ADDRESS <= "000100";
WAIT FOR 100 ns;
ADDRESS <= "000101";
WAIT FOR 100 ns;
ADDRESS <= "000110";
WAIT FOR 100 ns;
ADDRESS <= "000111";
WAIT FOR 100 ns;
ADDRESS <= "001000";
WAIT FOR 100 ns;
```

سپس مانند تصویر راست، در شبیه‌سازی به ترتیب خانه‌های حافظه را وارد کنیم،

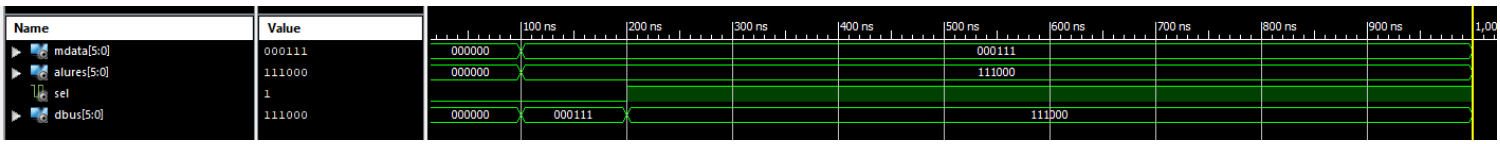
دقیقا در دیتا همان چیزی نوشته می‌شود که در خانه‌ی حافظه‌ی مورد نظرمان وجود دارد:



کنترل‌کننده گذرگاه - BUSC

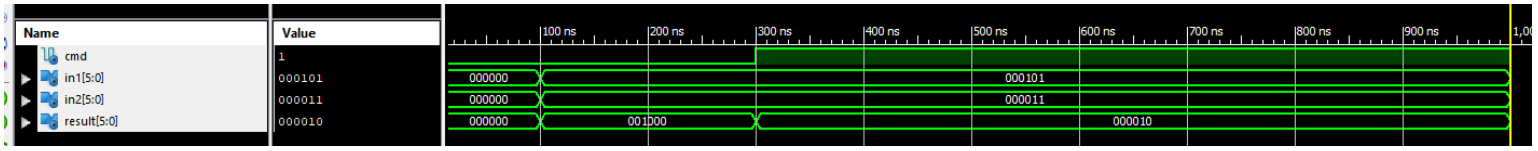
این بخش یک مالتی‌پلکسر دو به یک است که اگر سلکت صفر باشد باید مقدار MDATA و اگر یک باشد باید مقدار ALURES را در خروجی که DBUS یا گذرگاه داده است قرار دهد. طبق تست‌بنچ می‌بینیم که این بخش هم به درستی کار میکند.

```
MDATA <= "000111";
ALURES <= "111000";

SEL <= '0';
WAIT FOR 100 ns;
SEL <= '1';
WAIT FOR 100 ns;
```



واحد محاسبه و منطق - ALU

از این بخش انتظار داریم که در صورت صفر بودن CMD دو ورودی را با هم جمع کند و در صورت یک بودن، وردی اول منهای ورودی دوم را حساب کند. طبق تست‌بنچ میبینیم این بخش هم درست کار میکند.

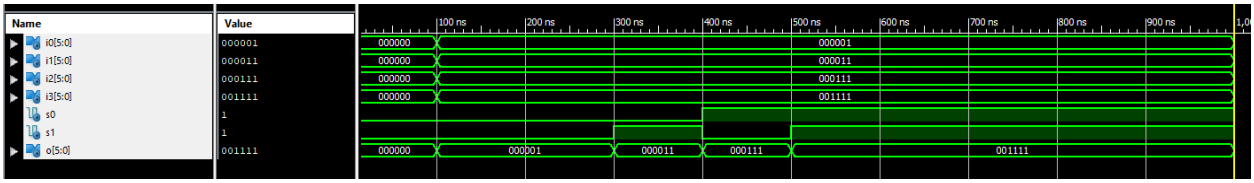



```
IN1 <= "000101"; --5
IN2 <= "000011"; --3
WAIT FOR 100 ns;
CMD <= '0'; -- RES: "001000" --8
WAIT FOR 100 ns;
CMD <= '1'; -- RES: "000010" --2
WAIT FOR 100 ns;
```

تعیین‌کننده ورودی واحد محاسبه و منطق - MUX4

اینجا یک مالتی‌پلکسر ۴ به ۱ داریم که ورودی و خروجی آن شش‌بیتی است.

طبق تست‌بنچ میبینم که درست کار میکند.

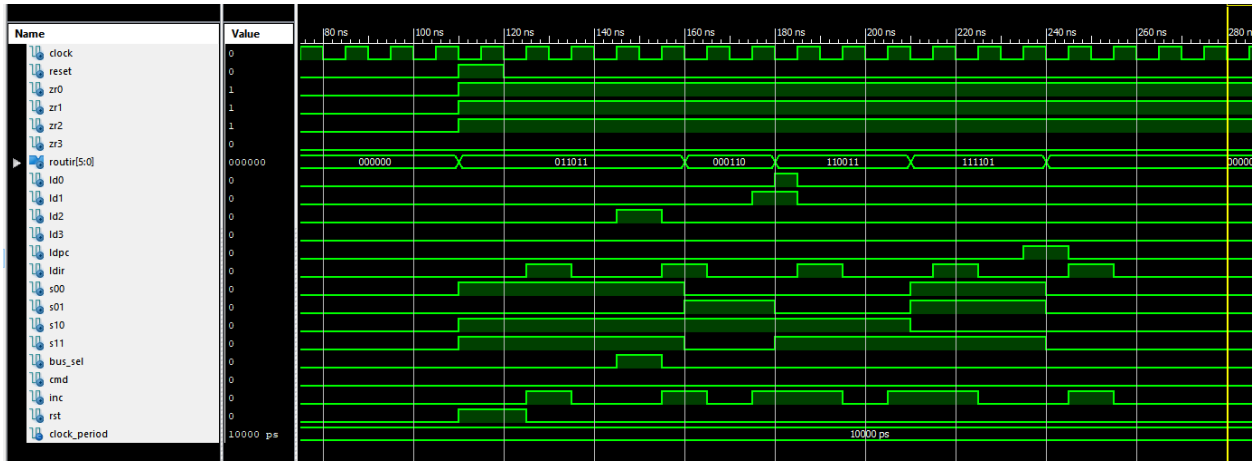



```
I0 <= "000001";
I1 <= "000011";
I2 <= "000111";
I3 <= "001111";
WAIT FOR 100 ns;
S0 <= '0';
S1 <= '0';
WAIT FOR 100 ns;
S0 <= '0';
S1 <= '1';
WAIT FOR 100 ns;
S0 <= '1';
S1 <= '0';
WAIT FOR 100 ns;
S0 <= '1';
S1 <= '1';
```

واحد کنترل - ControlUnit

این واحد با توجه به ورودی‌های زیروی رجیسترهای اصلی و خروجی رجیستر دستور، خروجی مورد نیاز تقریبا تمام بقیه‌ی برنامه را تعیین می‌کند که شامل: لود رجیسترهای اصلی و شمارنده برنامه و رجیستر دستور، سلکتورهای مالتی‌پلکسرهای ورودی ALU و کنترل‌کننده گذرگاه و همچنین کامند و اینکریز و ریست می‌شود.

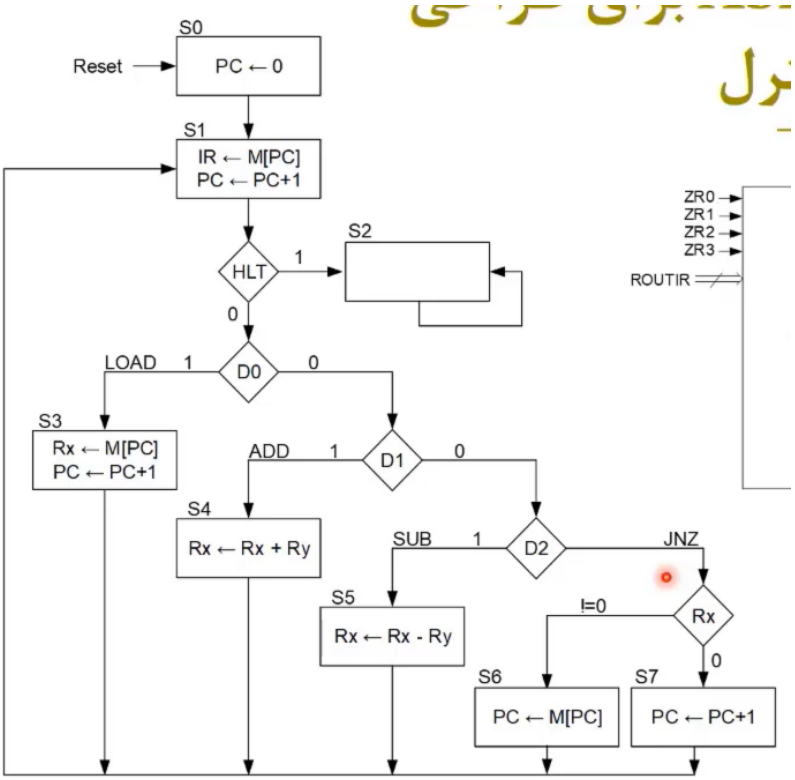
این بخش هم با توجه تست‌بنچ درست کار می‌کند.

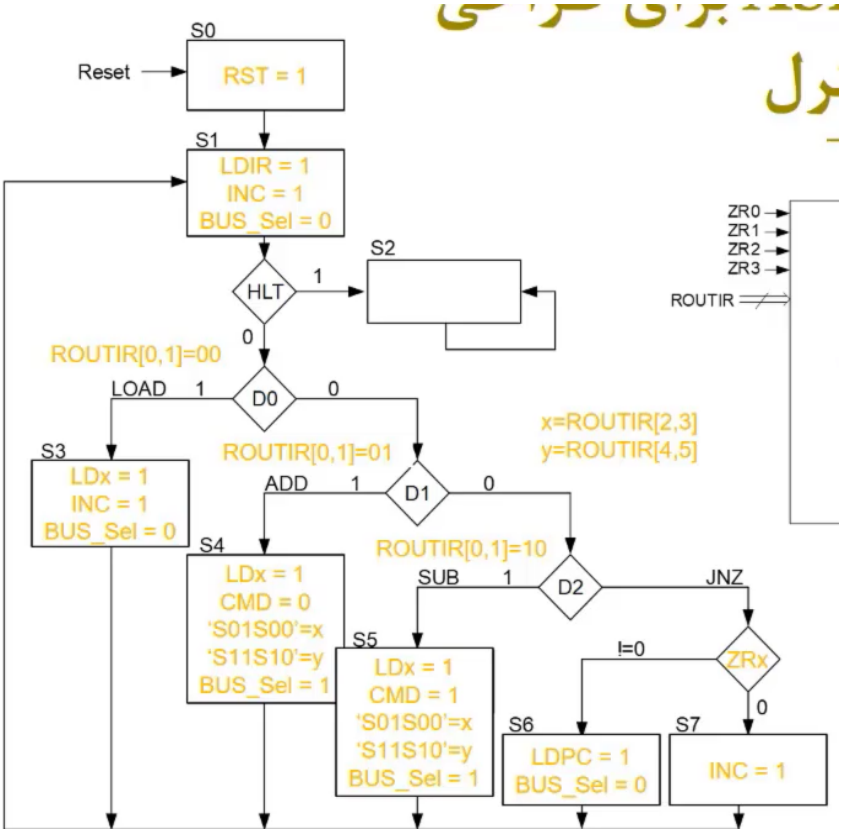



```
WAIT FOR clock_period;

Reset <= '1';
ROUTIR <= "011011";
ZR0 <= '1';
ZR1 <= '1';
ZR2 <= '1';
ZR3 <= '0';

WAIT FOR clock_period;

Reset <= '0';

WAIT FOR clock_period * 4;

ROUTIR <= "000110";

WAIT FOR clock_period * 2;

ROUTIR <= "110011";

WAIT FOR clock_period * 3;

ROUTIR <= "111101";

WAIT FOR clock_period * 3;

ROUTIR <= "000000";

WAIT FOR clock_period * 4;
```

این واحد دقیقا مطابق ویدیوی آموزشی پیاده‌سازی شده یعنی چارت ASM آن دقیقا به همان شکل می‌باشد.

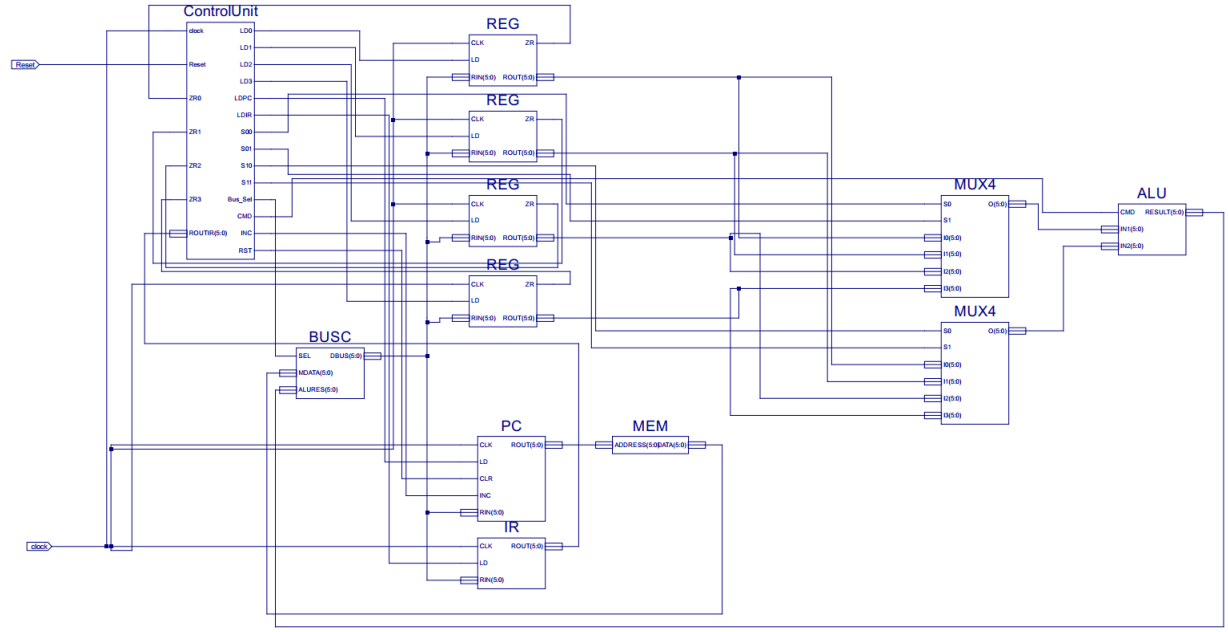
تصویر ASM واحد کنترل





تست کامل برنامه - CPU

ابتدا شماتیک را به این شکل، طبق طراحی انجام شده، ساختیم:



سپس با استفاده از اسمبلر که با زبان سی‌شارپ نوشتیم، خواسته‌های پروژه را به کد باینری تبدیل کردیم.

برای تست این پروژه ما هر دو بخش اول و دوم را پشت هم در مموری آوردیم.

ابتدا دستورات اسمبلی بخش دوم را نوشتیم که عمل ضرب بود که با پیاده‌سازی با استفاده از دستوراتی که داشتیم انجام دادیم. یعنی ۶ بار ۸ را با خودش جمع کردیم.

سپس به صورت دستی رجیسترهای اصلی را صفر کردیم.

در نهایت برای خواسته بخش اول که جمع دو عدد ۷ و ۴ بود، کد اسمبلی را به اسمبلر نوشته شده دادیم و خروجی را در ادامه‌ی مموری نوشتیم.

یعنی سی پی یو باید ابتدا عمل ضرب ۶ در ۸ را انجام دهد و نتیجه را در رجیستر صفر بریزد.

سپس رجیسترهای اصلی را صفر کند.

سپس جمع ۷ و ۴ را انجام دهد و نتیجه را در رجیستر صفر بریزد.

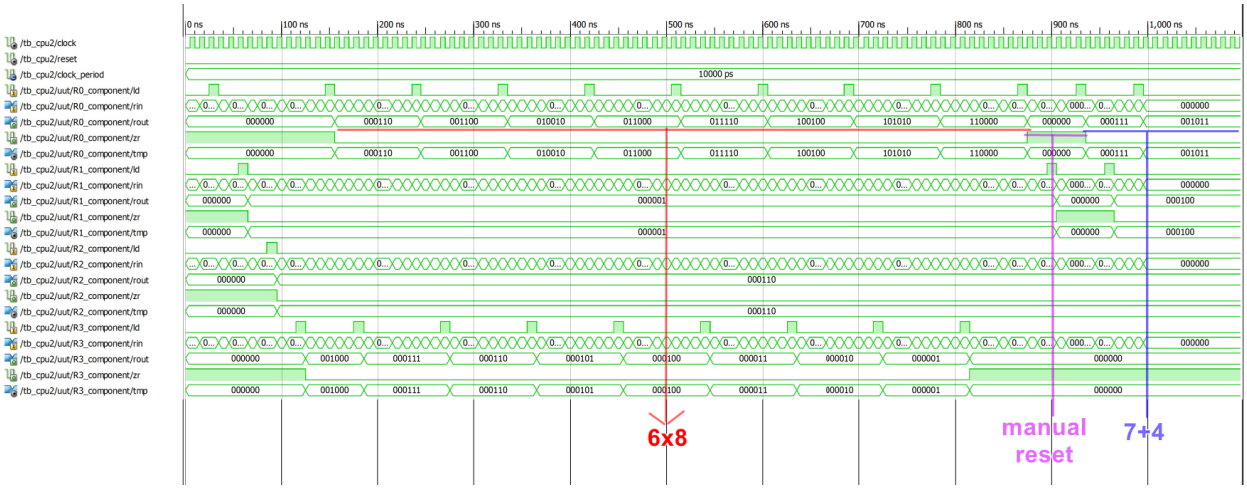
این که هر دو بخش را در یک بار مموری نوشتیم، برای این بود که صحت عملکرد برنامه را خیلی بهتر بتوانیم اثبات کنیم.

به این ترتیب خواسته‌های پروژه انجام شد.

تست‌بنچ کامل سی‌پی‌یو (پر کردن خانه‌های مموری)

```
ARCHITECTURE Behavioral OF MEM IS
    ----------------------------------------------------------------------------------
    TYPE mem_array IS ARRAY(0 TO 63) OF STD_LOGIC_VECTOR(5 DOWNTO 0);
    CONSTANT ROM : mem_array := (
        --------------TEST Loop (second part of project) ----------------------
        -- 8x6 = 6 times add 8 to 8
        "000011", -- Load R0
        "000000", -- 0
        "000111", -- Load R1
        "000001", -- 1
        "001011", -- Load R2 (counter)
        "000110", -- 6
        "001111", -- Load R3
        "001000", -- 8
        "010010", -- Add R0, R2 (mem[8])
        "101101", -- Sub R3, R1
        "111011", -- JNZ R3
        "001000", -- 8 (go to mem[8])
        -- "000000", -- HLT (commented this so cpu can continue working)
        --------------reseting registers ----------------------
        "000011", -- Load R0
        "000000", -- 0
        "000100", -- Load R1
        "000000", -- 0
        -- "000000", -- HLT (commented this so cpu can continue working)
        --------------TEST Add (first part of project) ----------------------
        "000011", -- load R0
        "000111", -- 7
        "000111", -- load R1
        "000100", -- 4
        "010001", -- Add R0, R1
        "000000", -- HLT
        "000000",
        "000000",
        "000000",
        "000000",
        "000000",
```

نتیجه شبیه‌سازی کل سی‌پی‌یو (برای تست‌های برنامه)



کد عمل ضرب ۶ در ۸

```
Load R0, 0
Load R1, 1
Load R2, 6
Load R3, 8
L0: Add R0, R2
Sub R3, R1
Jnz R3, L0
Hlt
```

کد باینری این بخش هم در تصویری که برای تست‌بنچ کامل CPU ارائه شد وجود دارد.

بخش اضافی - اسمبلر

برای این سی‌پی‌یو یک اسمبلر با زبان سی‌شارپ نوشتیم.

برای استفاده از این برنامه ابتدا کدی که می‌خواهیم تبدیل کنیم را در فایل codes.asb قرار می‌دهیم.

سپس با اجرای این فایل با دستور

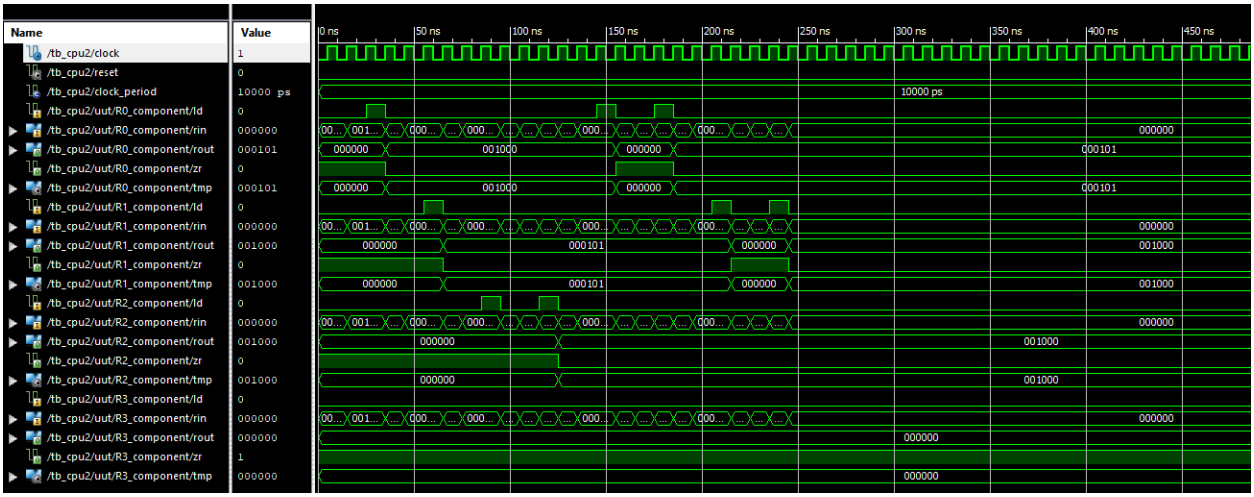
dotnet run ./codes.asb

کد تبدیل‌شده را در فایل codes.bytecode تحویل میگیریم و می‌توانیم در مموری paste کنیم.

بخش اضافی - Exchange

باید مقدار رجیستر R0 و R1 را با هم عوض کنیم.

طبق شبیه‌سازی می‌بینیم که این کار به درستی انجام شده




```
ARCHITECTURE Behavioral OF MEM IS
    ----------------------------------------------------------------------------------
    TYPE mem_array IS ARRAY(0 TO 63) OF STD_LOGIC_VECTOR(5 DOWNTO 0);
    CONSTANT ROM : mem_array := (
        --------------TEST Loop (second part of project) ----------------------
        "000011", --Load R0
        "001000", --8
        "000111", --Load R1
        "000101", --5
        "001000", -- Load R2
        "000000", -- 0
        "011000", -- Add R2 R0
        "000011", -- Load R0
        "000000", -- 0
        "010001", -- Add R0 R1
        "000111", -- Load R1
        "000000", -- 0
        "010110", -- Add R1 R2
        "000000", --HLT
        "000000",
        "000000",
```

بخش اضافی - برنامه‌نویسی Exchange بدون رجیستر کمکی

این کار ممکن است. و منطق کد آن به شکل زیر است:

```
// Code to swap 'x' and 'y'
x = x + y; // x now becomes 15
y = x - y; // y becomes 10
x = x - y; // x becomes 5
```

یعنی جابجایی دو مقدار در حافظه بدون رجیستر کمکی امکان دارد. به شرط اینکه می‌توانستیم تعیین کنیم که نتیجه‌ی تفریق به جای رجیستر اول به رجیستر دوم برود. اما سخت‌افزار و طراحی ما این محدودیت را دارد که برای عمل جمع و تفریق همیشه حاصل را در رجیستری که ابتدا نوشتیم میریزد و حق انتخاب نداریم.

بخش سوم پروژه

برای این بخش که صورت آن اضافه کردن دستور جدید ضرب به سخت‌افزار است، نیاز است تا تغییراتی در حجم سخت‌افزار ایجاد کنیم و آن را توسعه دهیم.

تغییراتی که باید ایجاد کنیم:

- آپ‌کد از دو بیت به سه بیت تغییر می‌کند چون نیاز داریم که به جای ۴ دستور، ۵ دستور را ساپورت کنیم که این کار با کمتر از ۳ بیت ممکن نیست.
- یک بیت زیاد کردن آپ کد باعث می‌شود تا کل گذرگاه سیستم، از ۶ بیت به ۷ بیت تغییر پیدا کند، چرا که دو بیت اول آن قبلا آپ‌کد بود که با زیاد شدن تعدادبیت‌های آن، حالا باید سه بیت اول را آپکد در نظر بگیریم.
- سیگنال CMD در ALU که قبلا تک‌بیتی بود و با آن تعیین می‌کردیم که میخواهیم ورودی را تفریق کنیم یا جمع کنیم، حالا باید به دوبیت افزایش پیدا کند تا بتوانیم عمل ضرب هم به آن اضافه کنیم. یعنی از دو عمل می‌شود سه عمل که برای انتخاب بین سه عمل به حداقل دو بیت نیاز داریم.
- تمام رجیسترهای اصلی هم باید یکی بهشان اضافه کنیم. یعنی به این شکل:

R0 ----> R00, R01

R1 ----> R10, R11

R2 ----> R20, R21

R3 ----> R30, R31

این به این خاطر است که حاصل ضرب nبیت در mبیت به حداکثر n+m بیت نیاز دارد یعنی برای ضرب ۶بیت در ۶بیت به حداکثر ۱۲ بیت نیاز داریم که میتوانیم بخش اول بیت‌هارا در رجیستر صفرم و بخش دوم را در رجیستر یکم از هر رجیستر ذخیره کنیم.

- به همین نسبت ZRهای واحد کنترل و لودهای رجیسترها که توسط واحد کنترل تعیین می‌شوند هم دوبرابر می‌شود.

مراحل انجام کار به همین شکلی است که مطرح شد و با دنبال کردن آن و افزایش حجم سخت‌افزارها می‌توان به نتیجه رسید.