

2021

Central Processing Unit

CoDesign Final-Project

Mahshid Alizade - Ashkan Mousazade



Guilan university

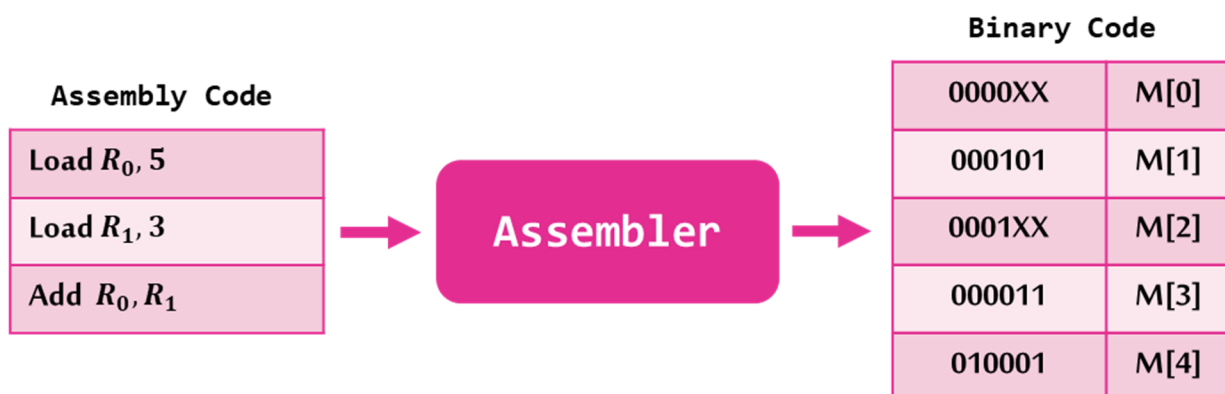
مقدمه

واحد پردازش مرکزی (CPU) که از آن با عنوان مغز رایانه یاد می‌شود، تنها واحد پردازشی رایانه نیست؛ اما مهم‌ترین آن‌ها به شمار می‌رود. سی‌پی‌یو در واقع آن بخش از رایانه است که اقدامات، محاسبات و اجرای برنامه‌ها را بر عهده دارد. عملکرد پایه‌ی سی‌پی‌یو شامل سه گام واکشی (Fetch)، رمزگشایی (Decode) و اجرا (Execute) است. درواقع سی‌پی‌یو داده‌های دستوری را از RAM دریافت، کدگشایی و پردازش می‌کند و تحویل می‌دهد.

هدف از این پروژه

در این پروژه قصد داریم :

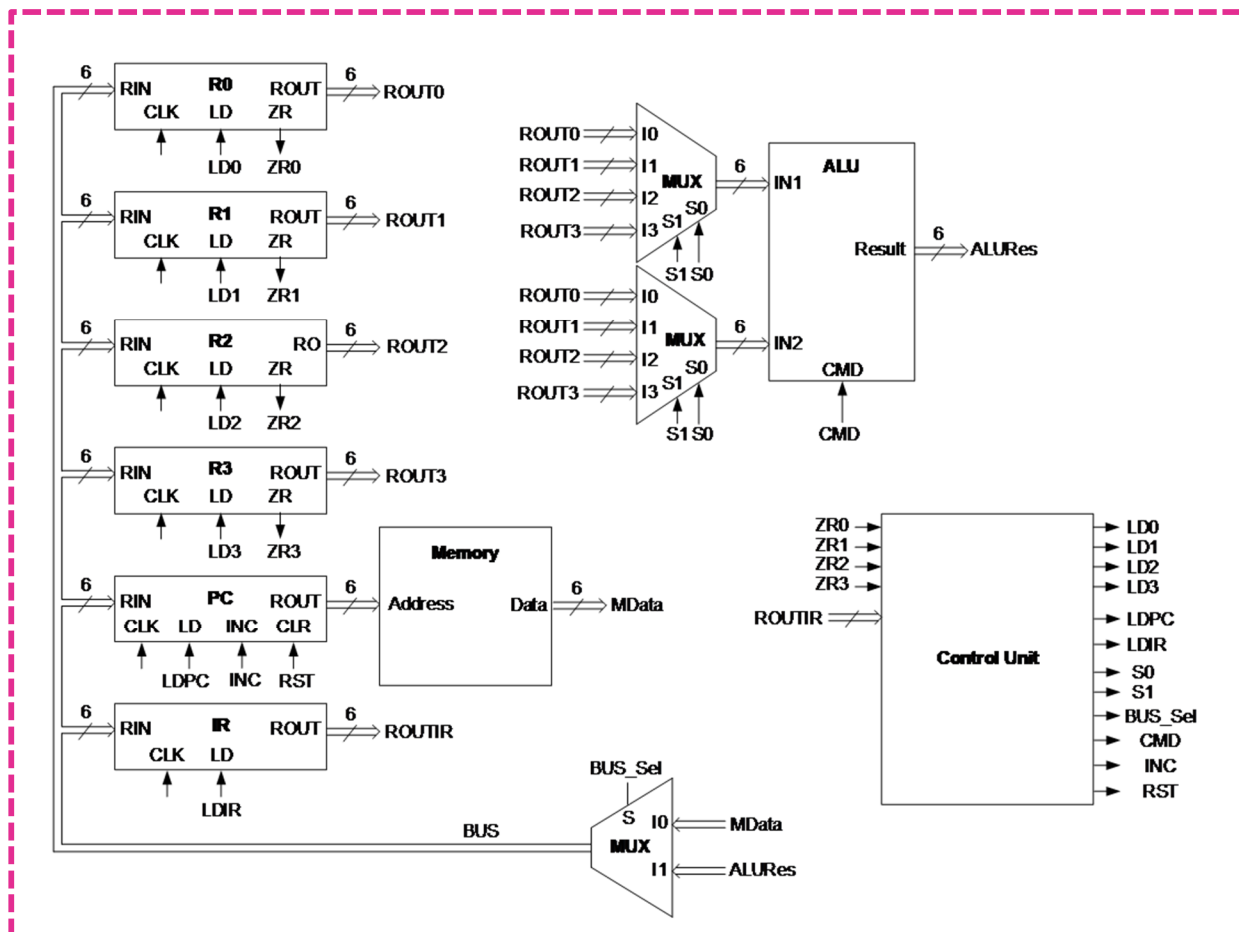
۱. یک اسمبلر با زبان برنامه نویسی دلخواه (ما در اینجا از java استفاده کردیم) طراحی کنیم، که بعنوان ورودی یک کد اسمبلی دریافت کرده و بعنوان خروجی معادل کد باینری آن را تحویل دهد.



** کد باینری تولید شده، در واقع خانه های حافظه ROM هستند.

۲. پردازنده 6 بیتی ، شامل بخش های زیر را با VHDL پیاده سازی می کنیم.

1. ثبات های اصلی ، 2. شمارنده ، 3. ثبات دستور ، 4. حافظه ROM ، 5. واحد کنترل و 6. واحد محاسبه و منطق 7. BUS.



سپس کد خروجی اسمبلر را در حافظه ROM قرار می دهیم و با استفاده از TestBench درستی آن را امتحان می کنیم.

بخش اسمبلر :

در بخش اسمبلر ، که با زبان جاوا طراحی شده ، برنامه یک فایل تکست را بعنوان ورودی می گیرد و بعنوان خروجی یک کد باینری بر می گرداند.

۱- خواندن فایل تکست :

```
// ----- Read file -----
try {
    FileReader file_reader = new FileReader("Assebl yCode. txt");
    Scanner sc = new Scanner(file_reader );

    int i = 0;
    while (sc.hasNextLine()) {
        data_str += sc.nextLine() + "; ";
    }

    data = data_str.split(";");
} catch (IOException e) {
    e.printStackTrace();
}
```

در ادامه یکی یکی به شناسایی دستورات میپردازیم ، قالب دستورات به طور کلی به صورت زیر است :

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

اطلاعات هر دستور بصورت زیر هستند :

چینش در حافظه	کد دستور (Code Op)	RTL	اسمبلی دستور
<div>PC → 00 R_x 00</div> <div>مقدار</div> <div>دستور بعدی</div>	00	$R_x \leftarrow M[PC]$	Load R_x, Val
<div>PC → 01 R_x R_y</div> <div>دستور بعدی</div>	01	$R_x \leftarrow R_x + R_y$	Add R_x, R_y
<div>PC → 10 R_x R_y</div> <div>دستور بعدی</div>	10	$R_x \leftarrow R_x - R_y$	Sub R_x, R_y
<div>PC → 11 R_x 00</div> <div>آدرس پرش</div> <div>دستور بعدی</div>	11	$if (R_x \neq 0) PC \leftarrow M[PC]$ $else PC \leftarrow PC + 1$	JNZ $R_x, Addr$

۲- شناسایی دستور Load :

```
// ----- LOAD -----
if(segment[0].equals("Load")){

    oprator = "00"; // OPcode

    switch (segment[1]) {
        case "R0, ":
            operand1 = "00";
            break;
        case "R1, ":
            operand1 = "01";
            break;
        case "R2, ":
            operand1 = "10";
            break;
        case "R3, ":
            operand1 = "11";
            break;

        default:
            break;
    }

// ----- PrintOut -----
System.out.println("m(" + index + ") <= \"" + oprator + operand1 + "11" + "\" ;" );
index++;
String value = "000000" + (Integer.toString(Integer.parseInt(segment[2]))).toString();
System.out.println
("m(" + index + ") <= \"" + value.substring(value.length() - 6, value.length() ) + "\" ;" );
```

۳- شناسایی دستور Add :

```
// ----- ADD -----
else if(segment[0].equals("Add") ){

    oprator = "01"; // OPcode

    switch (segment[1]) {
        case "R0, ":
            operand1 = "00";
            break;
        case "R1, ":
            operand1 = "01";
            break;
        case "R2, ":
            operand1 = "10";
            break;
        case "R3, ":
            operand1 = "11";
            break;

        default:
            break;
    }

}
```

به همین ترتیب عملوند دوم را هم شناسایی میکنیم و :

```
// ----- PrintOut -----
System.out.println("m(" + index + ") <= \"" + oprator + operand1 + operand2 + "\" ;" );
```

۴- شناسایی دستور Sub:

```
// ----- SUB -----  
else if(segment[0].equals("Sub")) {  
    oprator = "10"; // OPcode  
  
    switch (segment[1]) {  
        case "R0,":  
            operand1 = "00";  
            break;  
        case "R1,":  
            operand1 = "01";  
            break;  
        case "R2,":  
            operand1 = "10";  
            break;  
        case "R3,":  
            operand1 = "11";  
            break;  
  
        default:  
            break;  
    }  
}
```

به همین ترتیب عملوند دوم را هم شناسایی میکنیم و:

```
// ----- PrintOut -----  
System.out.println("m(" + index + ") <= \"" + oprator + operand1 + operand2 + "\" ;" );
```

۵- شناسایی دستور JNZ:

```
// ----- JNZ -----  
else if(segment[0].equals("Jnz")){  
    oprator = "11"; // OPcode  
  
    switch (segment[1]) {  
        case "R0,":  
            operand1 = "00";  
            break;  
        case "R1,":  
            operand1 = "01";  
            break;  
        case "R2,":  
            operand1 = "10";  
            break;  
        case "R3,":  
            operand1 = "11";  
            break;  
  
        default:  
            break;  
    }  
}
```

```
// ----- PrintOut -----  
System.out.println("m(" + index + ") <= \"" + oprator + operand1 + "11" + "\" ;" );  
index++;  
String value = "000000" + (Integer.toBinaryString(Integer.parseInt(segment[2]))).toString();  
System.out.println  
("m(" + index + ") <= \"" + value.substring(value.length() - 6, value.length() ) + "\" ;" );
```

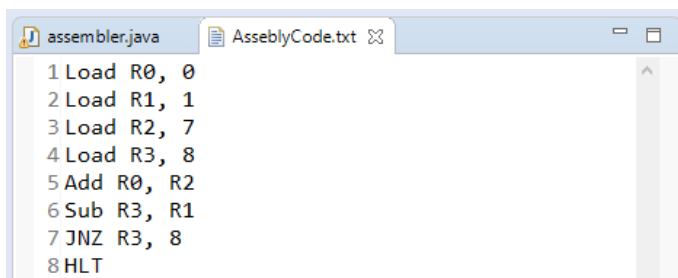
۶- شناسایی Halt :

دستور Halt نشان دهنده ی پایان کار است .

```
else if(segment[0].equals("Hl t")){  
    System.out.println("m(" + index + ") <= \"\" + \"000000\" + \"\\\" ;\" );  
    break;  
}
```

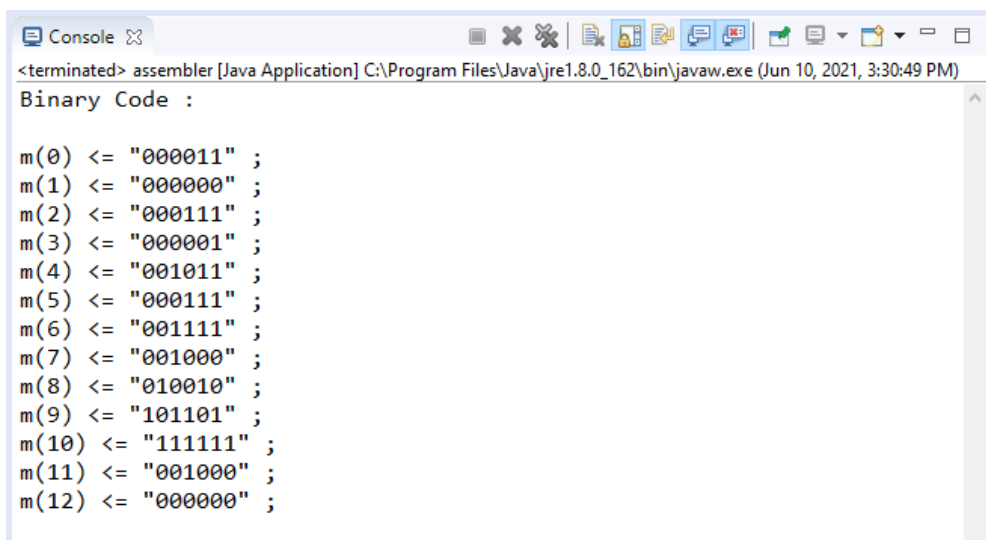
۷- یک مثال از ورودی و خروجی اسمبلر :

ورودی :



```
1 Load R0, 0  
2 Load R1, 1  
3 Load R2, 7  
4 Load R3, 8  
5 Add R0, R2  
6 Sub R3, R1  
7 JNZ R3, 8  
8 HLT
```

خروجی :



```
<terminated> assembler [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe (Jun 10, 2021, 3:30:49 PM)  
Binary Code :  
  
m(0) <= "000011" ;  
m(1) <= "000000" ;  
m(2) <= "000111" ;  
m(3) <= "000001" ;  
m(4) <= "001011" ;  
m(5) <= "000111" ;  
m(6) <= "001111" ;  
m(7) <= "001000" ;  
m(8) <= "010010" ;  
m(9) <= "101101" ;  
m(10) <= "111111" ;  
m(11) <= "001000" ;  
m(12) <= "000000" ;
```

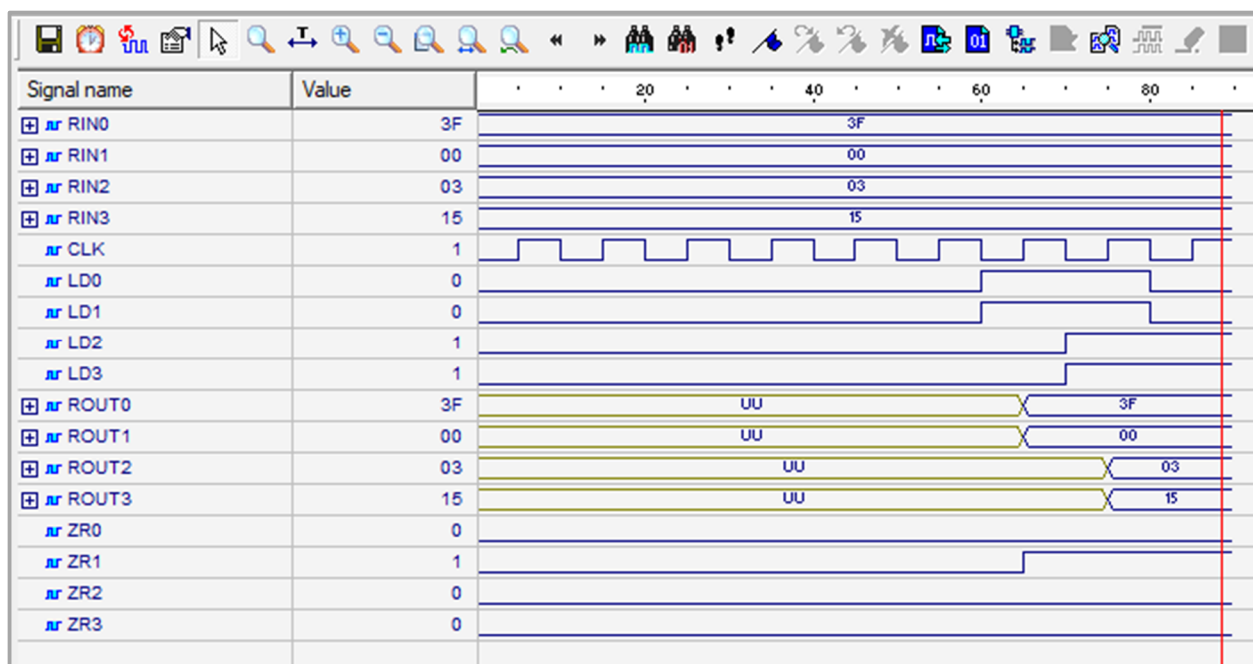
بخش پردازنده:

در بخش پردازنده که کارکرد هر بخش پردازنده به زبان VHDL پیاده سازی شده است را توسط دادن ورودی و گرفتن خروجی مناسب بررسی می کنیم.

۱- ثبات های اصلی :

در این بخش 4 ثبات اصلی داریم که هر کدام دارای 1 ورودی 6 بیتی RIN، clk و load و خروجی های 6 بیتی ROUT و ZR هستند.

با توجه به تعریف، اگر load یک باشد و لبه ی بالا رونده کلاک آمده باشد، ورودی را به خروجی انتقال می دهیم و اگر مقدار هر ثبات صفر باشد، خروجی ZR ثبات مربوطه صفر میشود.



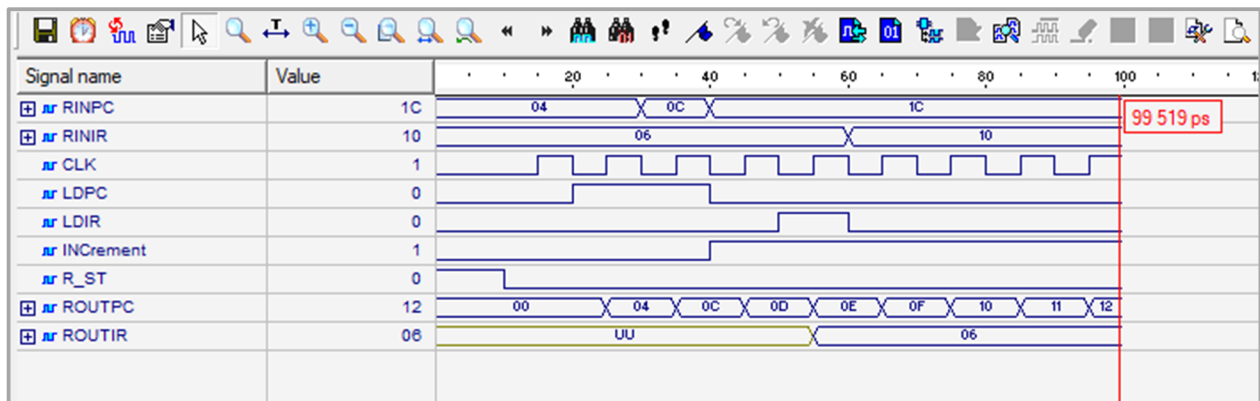
۲- شمارنده برنامه و ثبات دستور :

بخش شمارنده شامل یک ورودی 6 بیتی **RINPC** و ورودی های **clk** , **loadPC** , **increment** و **reset** و یک خروجی 6 بیتی **ROUTPC** است.

با توجه به تعریف اگر **reset** یک باشد خروجی صفر میشود در غیر این صورت هر زمان که لبه ی بالا رونده **clk** رسید، اگر **load** آن فعال بود ورودی را به خروجی انتقال می دهد و اگر **increment** فعال بود به خروجی یک واحد اضافه میکند.

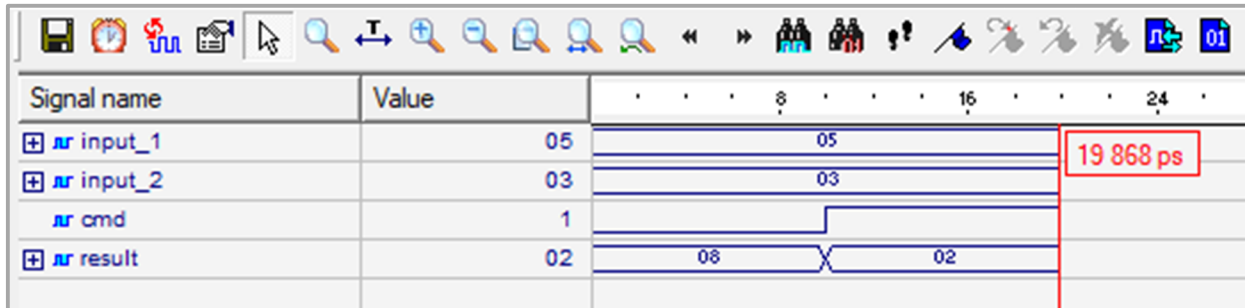
بخش ثبات دستور دارای یک ورودی 6 بیتی **RINIR** و ورودی های **clk** و **loadIR** و یک خروجی 6 بیتی **ROUTIR** است.

با توجه به تعریف اگر **reset** یک باشد ورودی صفر به خروجی میدهیم در غیر این صورت هر زمان که لبه ی بالا رونده **clk** رسید و **load** آن فعال بود ورودی را به خروجی انتقال می دهد.



۳- واحد محاسبه و منطق :

این بخش شامل 2 ورودی 6 بیتی و 1 بیت خط سلکت CMD و یک خروجی 6 بیتی است. با توجه به تعریف هر زمان که cmd صفر باشد ALU ورودی ها را با هم جمع میکند و هر وقت یک باشد از هم کم میکند. در این بخش بعنوان مثال 2 عدد 5 و 3 را در نظر گرفتیم و با هم جمع و تفریق میکنیم.

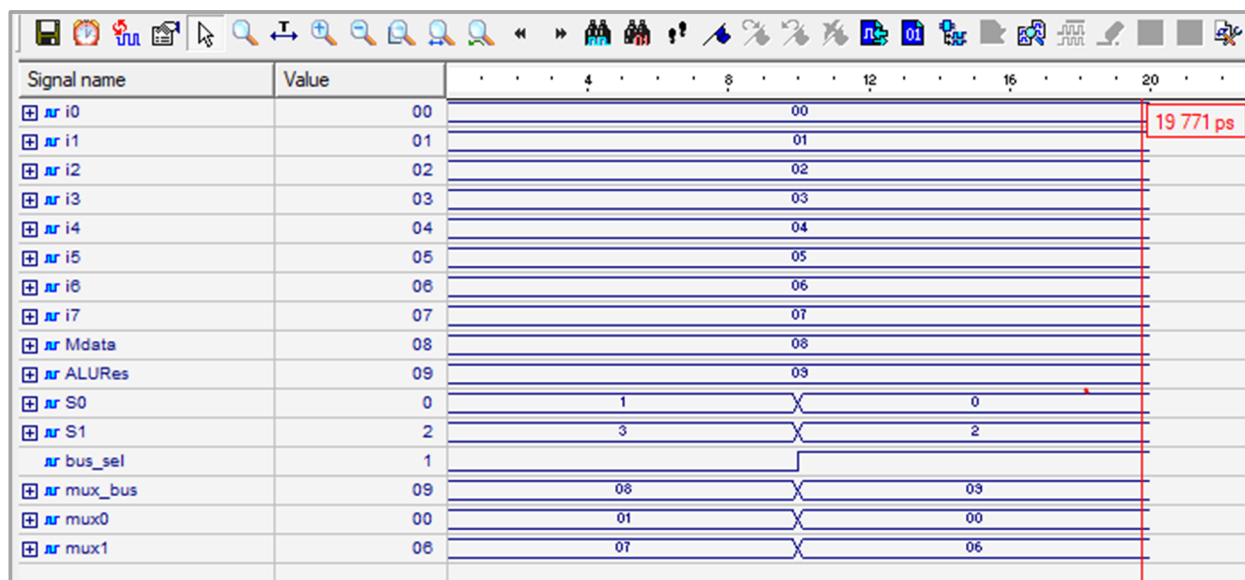


۴- MUX ها :

2 بخش MUX داریم در پردازنده، یکی پشت واحد محاسبه و منطق و دیگری پشت data bus.



MUX بخش data bus، دارای 2 ورودی Mdata و ALUres و یک خروجی bus می باشد که با توجه به مقدار خط سلکت bus_sel یکی از دو ورودی انتخاب میشود و به خروجی منتقل می شود.

MUX های بخش ALU، هر کدام دارای 4 ورودی 6 بیتی و یک خروجی 6 بیتی (mux0 و mux1) و یک خط سلکت 2 بیتی (s0 و s1) هستند.



٥- حافظه ROM :

این بخش دارای یک ورودی 6 بیتی **Address** و یک خروجی 6 بیتی **Data** می باشد، که بر اساس آدرس ورودی ، مقدار همان خانه از حافظه را میخواند و بعنوان خروجی **Data** بر میگرداند.

Signal name	Value				80			160	
 nr Address	05	00	01	02	03	04	05	149 652 ps	
 nr Data	07	03	00	07	01	0B	07		

۶- واحد کنترل :

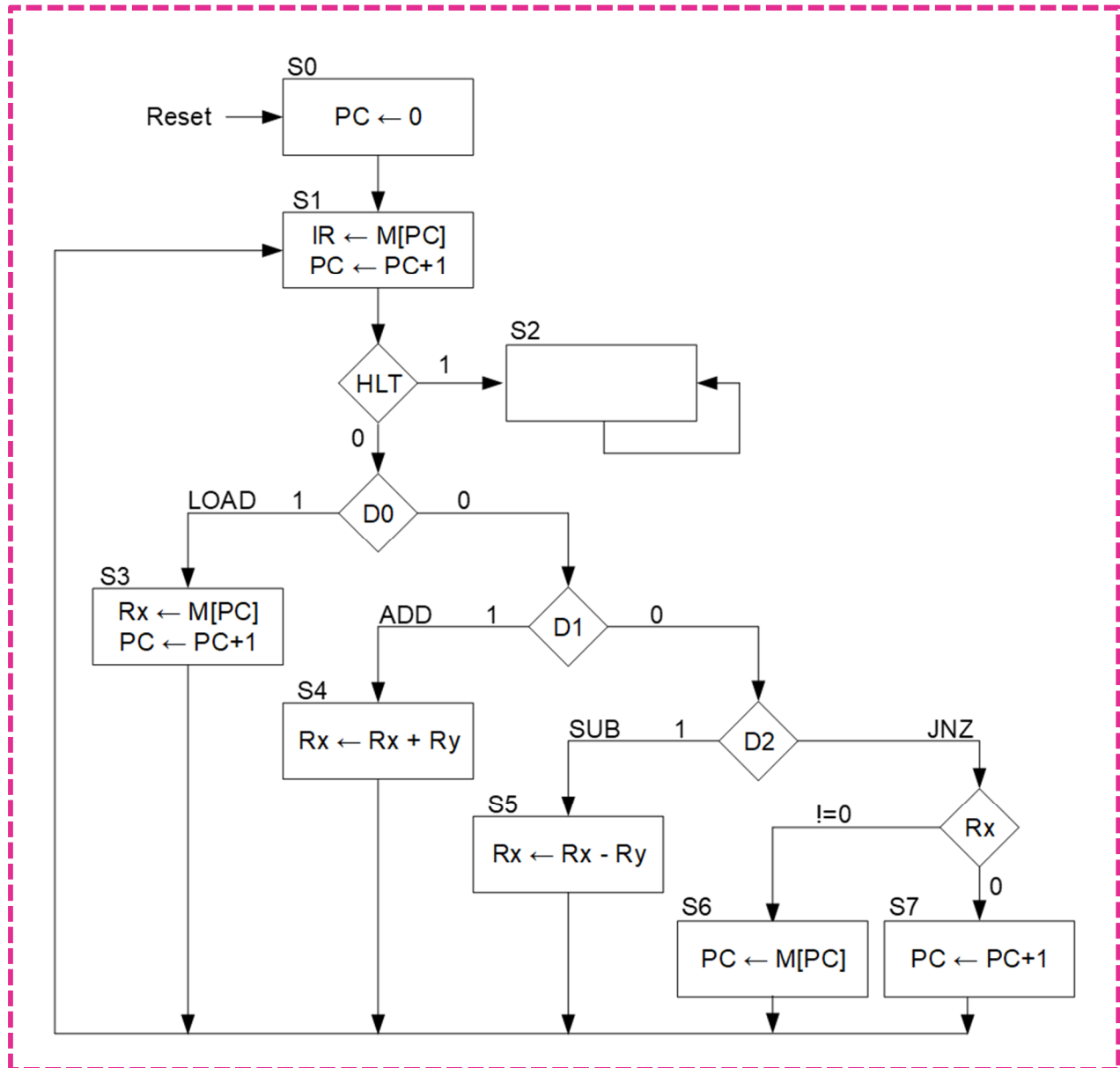
بخش **control unit** دارای 4 ورودی (**clk** ، **ZR** ، **reset** و **ROUTIR 6** بیتی) و 6 خروجی (**load** ثبات ها و خروجی خط سلکت **MUX** ها ، **increment** ، **cmd** و **rst**) است. همانطور که از اسم این بخش مشخص است ، وظیفه ی کنترل سایر بخش های ذکر شده را بر عهده دارد و تصمیم میگیرد در هر زمان کدام خروجی فعال یا غیر فعال شود.

****** برای اطلاع بیشتر از نحوه ی عملکرد این بخش ، کافی ست چارتی که در صفحه بعد آمده مطالعه شود.

با توجه به تعریف و چارت **ASM** تعریف شده، اگر ریست فعال شود استیت فعلی ما **S0** می شود و با آمدن لبه کلاک به استیت های بعدی میرویم.

The timing diagram displays various digital signals over an 80 ns period. A red box highlights a specific duration of 71 655 ps for the ROUT_IR signal. The signals shown include clock, rst, ZR0-ZR3, ROUT_IR, LD0-LD3, LD_PC, LD_IR, Sel0, Sel1, BusSelect, ALU_CMD, INC, CLR, PSTATE, and NSTATE. The ROUT_IR signal is shown with a value of 00 and a duration of 71 655 ps. The Sel0 and Sel1 signals are shown with values 0 and 2, and 0 and 3 respectively. The ALU_CMD signal is shown with values 0 and 1. The INC signal is shown with values 0 and 1. The CLR signal is shown with values 0 and 1. The PSTATE and NSTATE signals are shown with values s2 and s1 respectively.

چارت مربوط به واحد کنترل :



بخش اول پروژه :

صحت عملکرد برنامه را با اجرای کد زیر که دو عدد 5 و 3 را با هم جمع می کند بررسی کنید.

```
Load R0, 5
Load R1, 3
Add R0, R1
Hi t
```

پاسخ :

بخش اول.

در ابتدا کد اسمبلی بالا را به اسمبلر (که در بخش اول پروژه پیاده سازی کردیم) می دهیم و خروجی زیر را دریافت میکنیم .

```

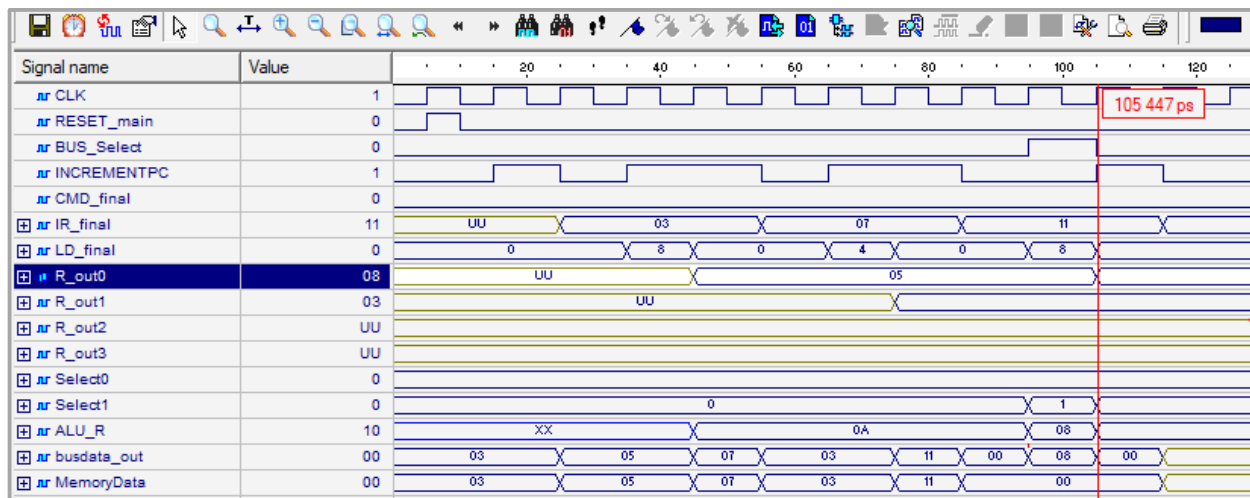
<terminated> assembler [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe (Jun 10, 2021, 10:16:06 PM)
Binary Code :

m(0) <= "000011" ;
m(1) <= "000101" ;
m(2) <= "000111" ;
m(3) <= "000011" ;
m(4) <= "010111" ;
m(5) <= "000000" ;

```

بخش دوم.

سپس خروجی بالا را به حافظه ی ROM می دهیم و خروجی تست بنچ را مشاهده میکنیم :



بخش دوم پروژه :

عمل ضرب را با استفاده از عمل جمع و به صورت نرم‌افزاری پیاده‌سازی کرده و صحت عملکرد آن را با یک مثال نشان دهید. برای مثال ضرب 7 در 8 .

پاسخ :

بخش اول.

ابتدا کد ضرب را با استفاده از دستور جمع ، بصورت زیر پیاده سازی میکنیم:

```
Load R0, 0
Load R1, 1
Load R2, 7
Load R3, 8
Add R0, R2
Sub R3, R1
Jnz R3, 8
Hlt
```

بخش دوم.

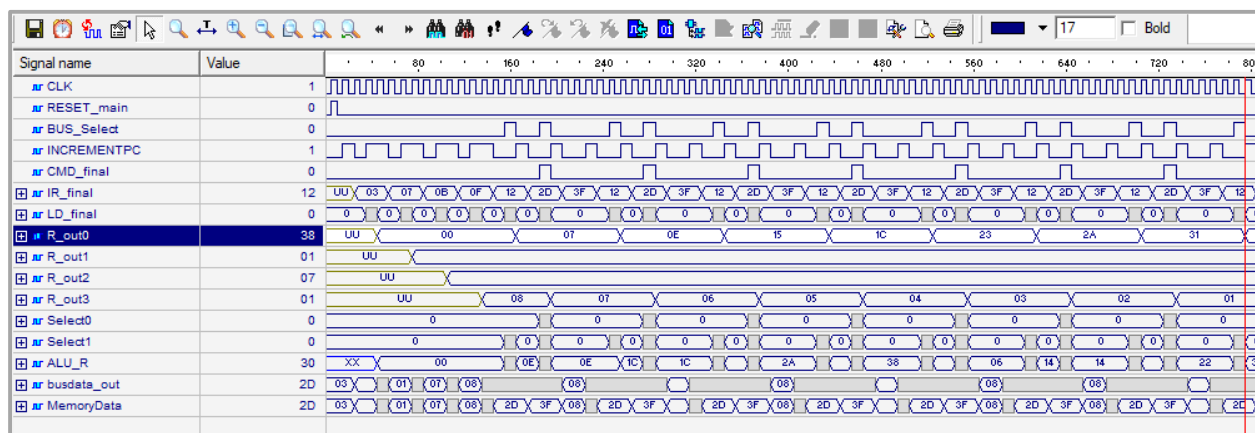
حالا کد اسمبلی بالا را به اسمبلر (که در بخش اول پروژه پیاده سازی کردیم) می دهیم و خروجی زیر را دریافت میکنیم :

```
Console
<terminated> assembler [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe (Jun 10, 2021, 11:51:39 PM)
Binary Code :

m(0) <= "000011" ;
m(1) <= "000000" ;
m(2) <= "000111" ;
m(3) <= "000001" ;
m(4) <= "001011" ;
m(5) <= "000111" ;
m(6) <= "001111" ;
m(7) <= "001000" ;
m(8) <= "010010" ;
m(9) <= "101101" ;
m(10) <= "111111" ;
m(11) <= "001000" ;
m(12) <= "000000" ;
```

بخش سوم.

سپس خروجی بالا را به حافظه ی ROM می‌دهیم و خروجی تست بنچ را مشاهده میکنیم :



* همانطور که میبینید در خروجی نهایی عدد 38 بصورت هگزادسیمال نمایش داده شده است که معادل 56 دسیمال است.