

# تمرین پنجم معماری کامپیوتر

## بخش عملی

دکتر اسدی

دانشجویان: امیرکسری احمدی  
روژین تقیزادگان

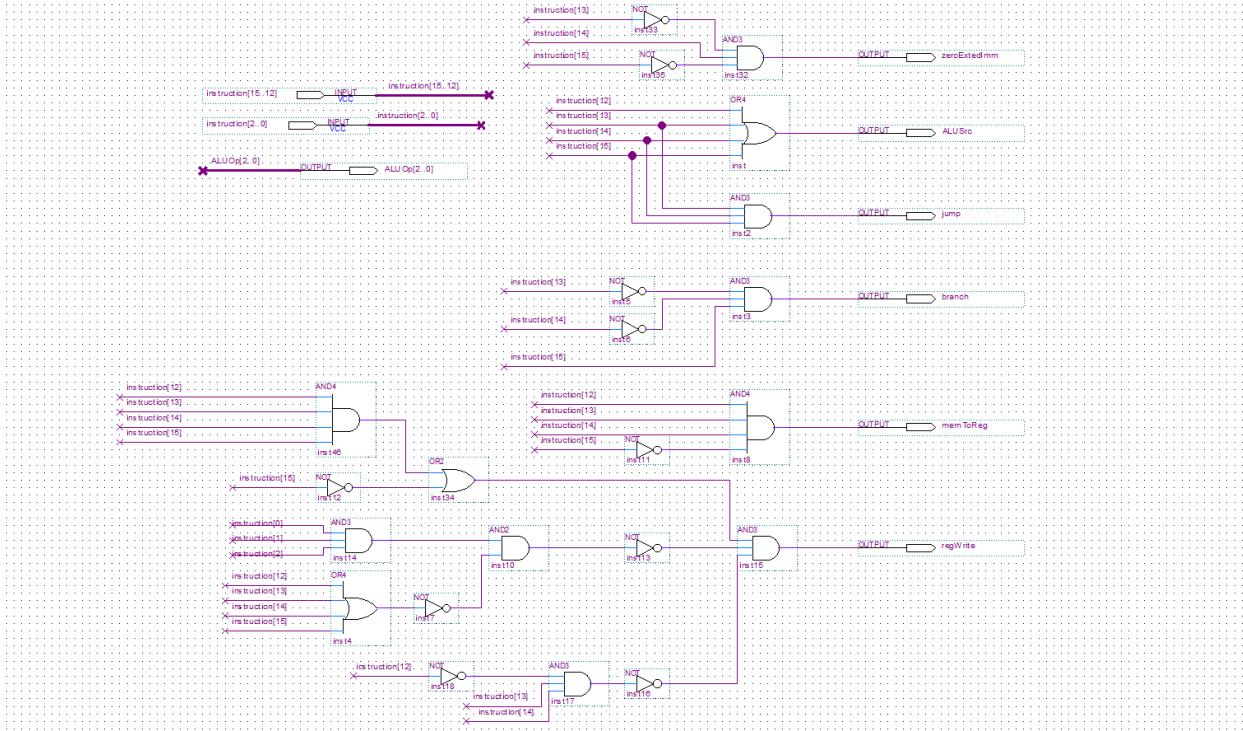
شماره دانشجویی: ۴۰۱۱۷۰۵۰۷  
۴۰۱۱۰۵۷۷۵

## طراحی کنترل یونیت :

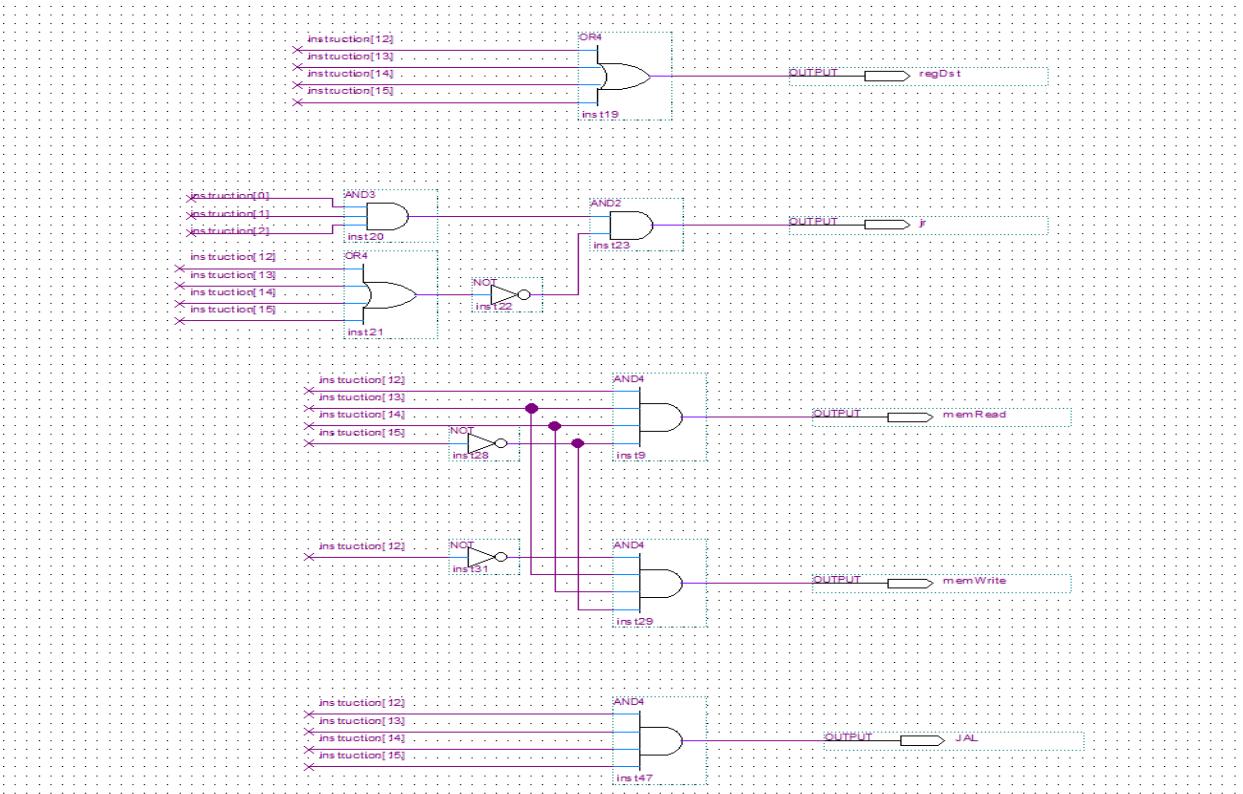
در این تمرین ابتدا به طراحی کنترل یونیت پرداخته‌ایم. ورودی‌های آن ۴ بیت Opcode و ۳ بیت دستورالعمل است. سیگنال‌های تولید شده آن در جدول زیر به همراه نحوه ساخت آنها توسط بیت‌های ورودی آمده است :

سیگنال تولیدی	عبارت منطقی
zeroExtendImm	$\sim\text{Op3}.\text{Op2}.\sim\text{Op1}$
ALUSrc	$\text{Op3}.\text{Op2}.\text{Op1}.\text{Op0}$
Jump	$\text{Op3}.\text{Op2}.\text{Op1}$
Branch	$\text{Op3}.\sim\text{Op2}.\sim\text{Op1}$
memToReg	$\sim\text{Op3}.\text{Op2}.\text{Op1}.\text{Op0}$
regWrite	$((\text{Op3}.\text{Op2}.\text{Op1}.\text{Op0}) \text{Op3}).(\sim(\text{Op3} \text{Op2} \text{Op1} \text{Op0}).(\text{F2}.\text{F1}.\text{F0})).(\sim(\text{Op2}.\text{Op1}.\sim\text{Op0}))$
regDst	$\text{Op3} \text{Op2} \text{Op1} \text{Op0}$
jr	$(\sim(\text{Op3} \text{Op2} \text{Op1} \text{Op0}).(\text{F2}.\text{F1}.\text{F0}))$
memRead	$\sim\text{Op3}.\text{Op2}.\text{Op1}.\text{Op0}$
memWrite	$\text{Op3}.\text{Op2}.\text{Op1}.\sim\text{Op0}$
JAL	$\text{Op3}.\text{Op2}.\text{Op1}.\text{Op0}$
ALUOp2	$((\sim\text{Op2}.\text{Op0}) (\sim\text{Op1}.\text{Op0}))$
ALUOp1	$((\sim\text{Op1}.\text{Op0}) (\text{Op3}.\text{Op2}.\text{Op1}))$
ALUOp0	$((\sim\text{Op0}).(\text{Op2} \oplus \text{Op1})) \text{Op3}$

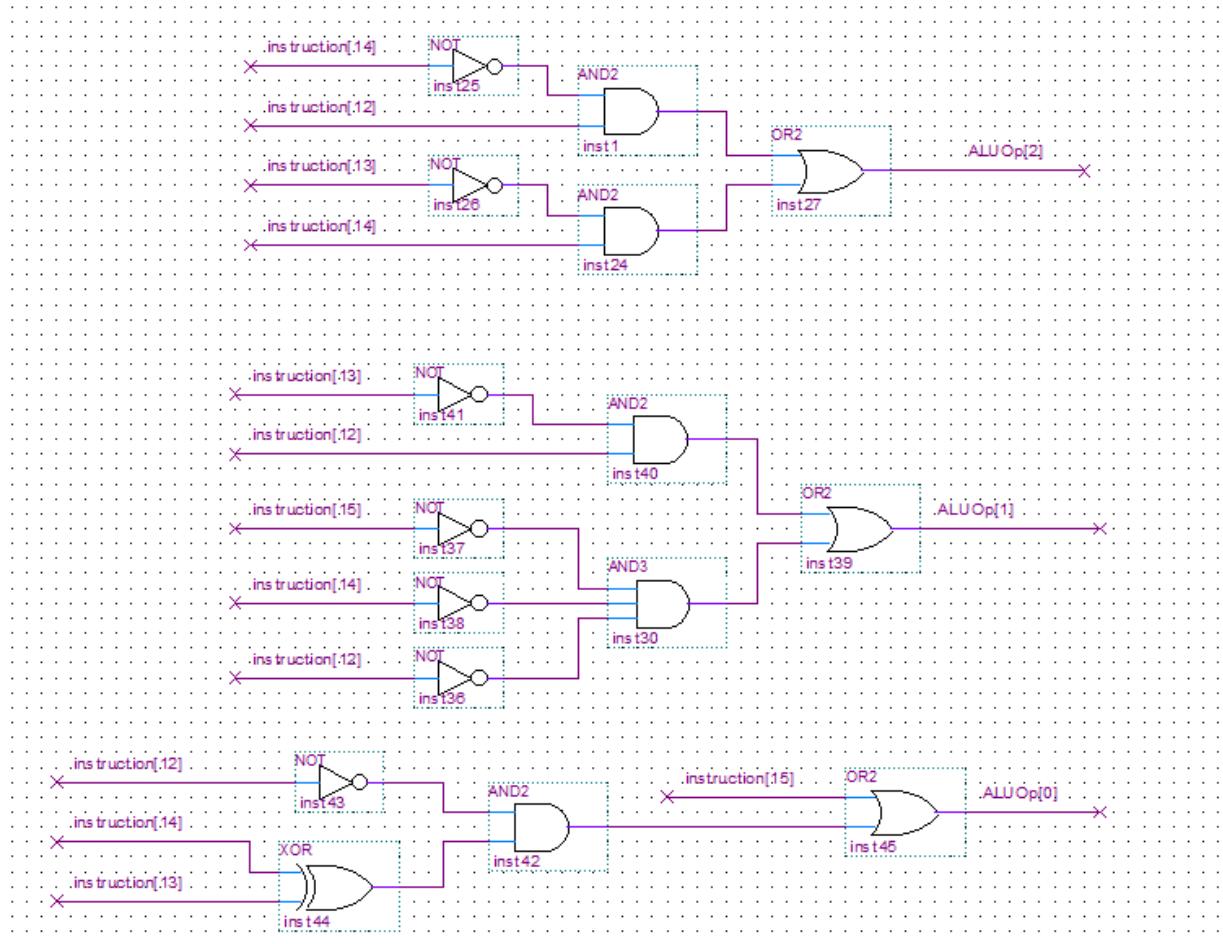
در تصاویر صفحه بعد می‌توان نحوه پیاده سازی این قطعه را مشاهده کنید :



شكل 1 - بخش اول کنترل یونیت



شكل 2 - بخش دوم کنترل یونیت



شكل 3 - بخش سوم کنترل یونیت

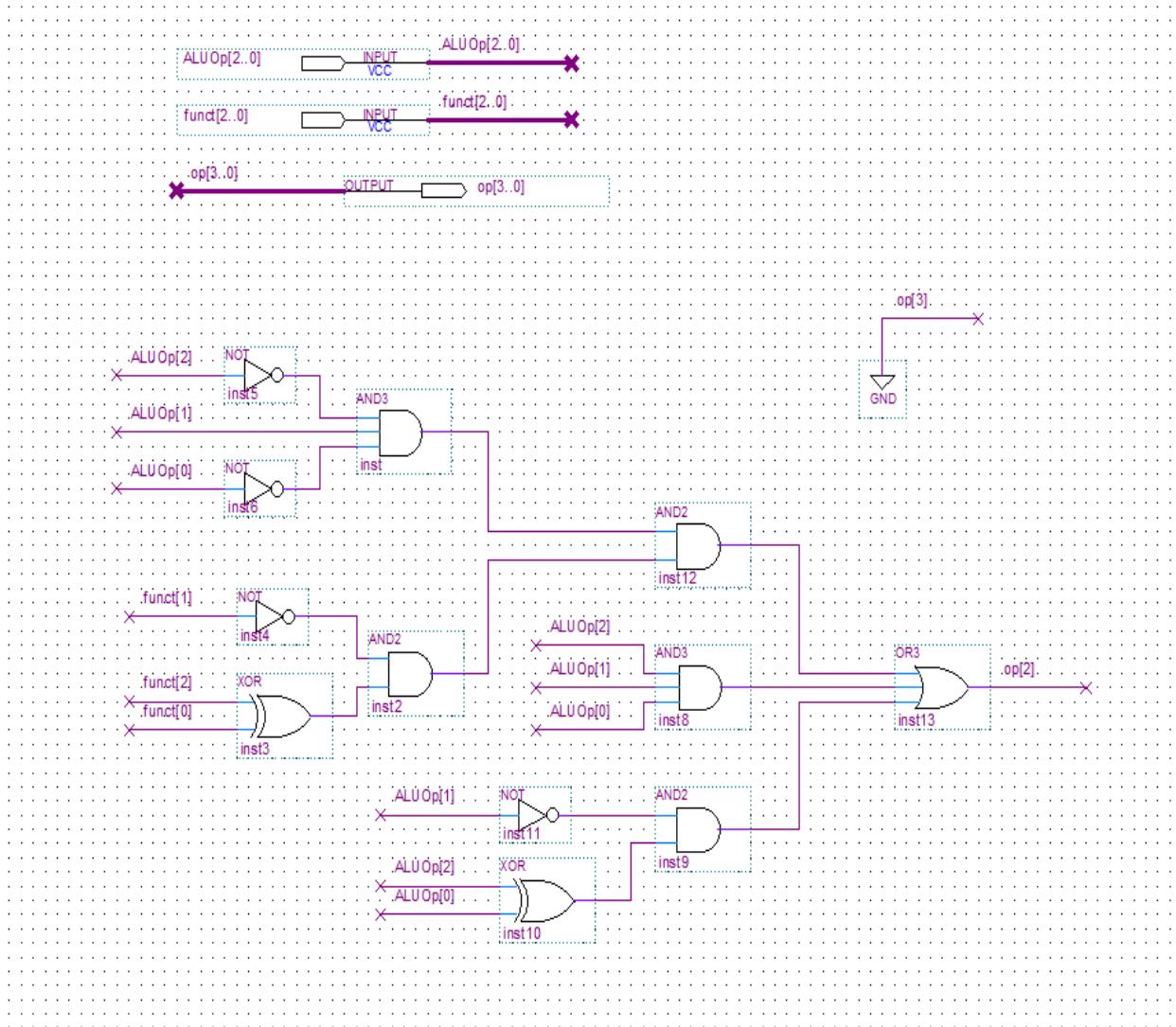
Instruction	Opcode3	Opcode2	Opcode1	Opcode0	ALUOp2	ALUOp1	ALUOp0
LB	0	1	1	1	0	0	0
SB	0	1	1	0	0	0	0
BEQ	1	0	0	0	0	0	1
R-TYPE	0	0	0	0	0	1	0
ADDI	0	0	1	0	0	1	1
SUBI	0	0	1	1	1	0	0
ANDI	0	1	0	0	1	0	1
ORI	0	1	0	1	1	1	0
BNQ	1	0	0	1	1	1	1

$ALUOp_2 = (op_2 \cdot \overline{op_1}) + (\overline{op_2} \cdot op_0)$   
 $ALUOp_1 = (\overline{op_1} \cdot op_0) + (\overline{op_3} \cdot \overline{op_2} \cdot \overline{op_0})$   
 $ALUOp_0 = op_3 + \overline{op_0} \cdot (op_2 \oplus op_1)$

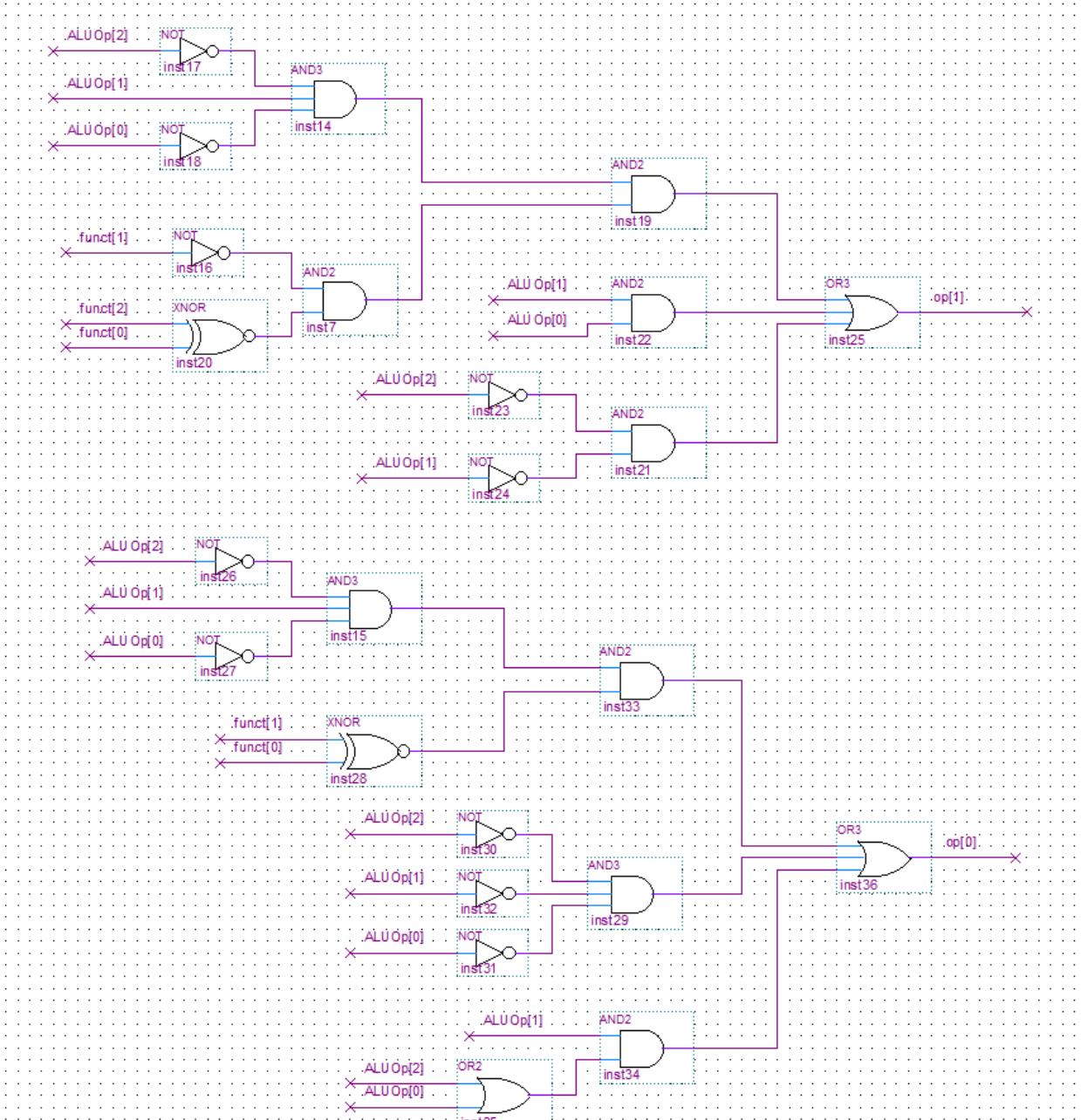
شكل 4 - جدول ALUOp بر اساس Opcode

## طراحی ALU Control

پس از طراحی کنترل یونیت سراغ طراحی ALU Control رفتیم. ورودی‌های این واحد ۳ بیت ALUOp که از Control Unit خارج می‌شوند و ۳ بیت funct دستورالعمل است. خروجی آن نیز ۴ بیت ورودی ALU است که نوع دستور در ALU را مشخص می‌کند.  
در تصاویر پایین نیز می‌توان نحوه پیاده سازی این واحد را مشاهده نمایید:



شکل ۵ - بخش اول ALU Control



شكل 6 - بخش دوم ALU Control

در جدول زیر می‌توان عبارات منطقی این خروجی‌ها را ببینید:

Instruction	ALUOp2	ALUOp1	ALUOp0	F2	F1	F0	Op3	Op2	Op1	Op0	action
LB	0	0	0	X	X	X	0	0	1	1	add
SB	0	0	0	X	X	X	0	0	1	1	add
BEQ	0	0	1	X	X	X	0	1	1	0	equal
ADD	0	1	0	0	0	0	0	0	1	1	add
SUB	0	1	0	0	0	1	0	1	0	0	subtract
AND	0	1	0	0	1	0	0	0	0	0	and
OR	0	1	0	0	1	1	0	0	0	1	or
MULT	0	1	0	1	0	0	0	1	0	1	mult
XOR	0	1	0	1	0	1	0	0	1	0	xor
ADDI	0	1	1	X	X	X	0	0	1	1	add
SUBI	1	0	0	X	X	X	0	1	0	0	subtract
ANDI	1	0	1	X	X	X	0	0	0	0	and
ORI	1	1	0	X	X	X	0	0	0	1	or
BNQ	1	1	1	X	X	X	0	1	1	1	n-equal

$$Op3 = 0$$

$$Op2 = \overline{ALUOp_2} \cdot ALUOp_1 \cdot \overline{ALUOp_0} \cdot \overline{F1}(F2 \oplus F0) + \overline{ALUOp_1}(ALUOp_2 \oplus ALUOp_0) + ALUOp_2 \cdot ALUOp_1 \cdot ALUOp_0$$

$$Op1 = \overline{ALUOp_2} \cdot ALUOp_1 \cdot \overline{ALUOp_0} \cdot \overline{F1}(F2 XNOR F0) + (\overline{ALUOp_2} \cdot \overline{ALUOp_1}) + (ALUOp_1 \cdot ALUOp_0)$$

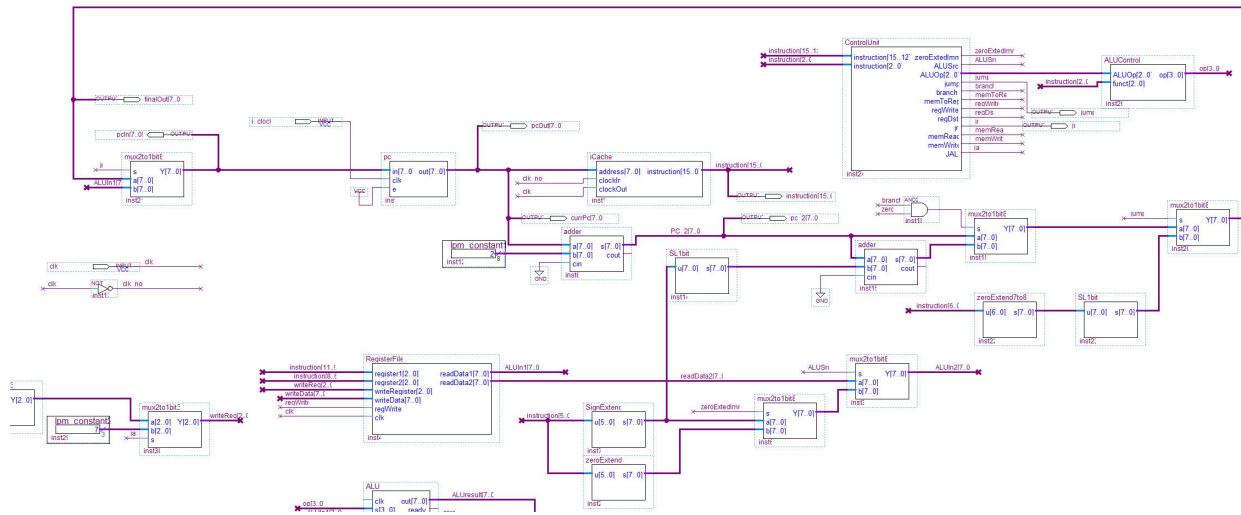
$$Op0 = (\overline{ALUOp_2} \cdot \overline{ALUOp_1} \cdot \overline{ALUOp_0}) + \overline{ALUOp_2} \cdot ALUOp_1 \cdot \overline{ALUOp_0} \cdot (F1 XNOR F0) + ALUOp_1(ALUOp_2 + ALUOp_0)$$

شکل 7 - جدول سیگنال‌های ورودی ALU بر اساس ALUOp

## طراحی مسیر داده :

در نهایت پس از طراحی تمامی اجزای مورد نیاز برای تولید سیگنال‌های کنترلی به سراغ طراحی مسیر داده آمده‌ایم تا تمامی واحدهای مورد نیاز در پردازنده mips را پیاده سازی کرده باشیم و بتوانیم از آن استفاده کنیم. در بخش‌های مختلف این بخش را بررسی می‌کنیم :

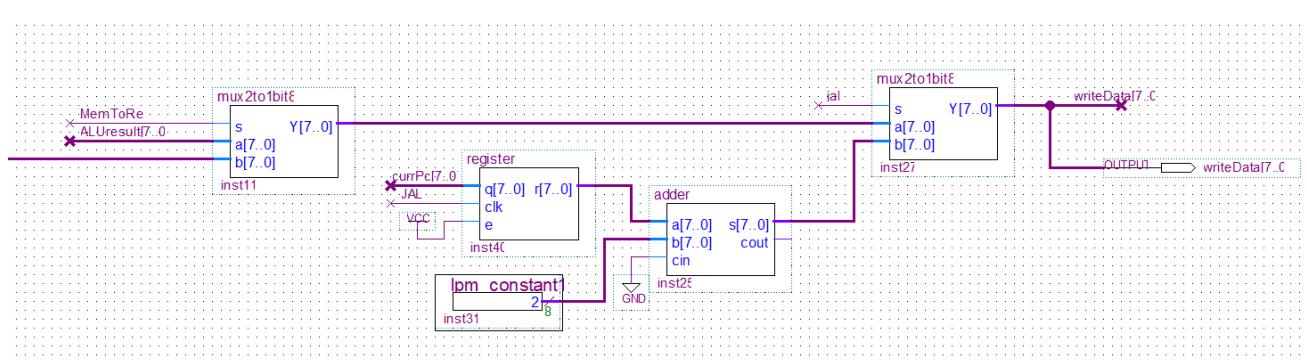
### ۱- واکشی دستور و آپدیت PC :



در این بخش هر بار با استفاده از PC که محتویات آن به عنوان ورودی به I-Cache داده می‌شود یک دستور را می‌خوانیم و سپس با استفاده از اددر محتویات آن را به علاوه ۲ کرده تا به دستور بعدی اشاره کنیم. حال در این مرحله این مقدار  $PC_{old} + 2$  با relative address ( $SIGN\_EXTEND(immediate \ll 1)$ ) مقدار  $PC_{old} + 2$  جمع شده و مقدار PC در صورت برنج محاسبه می‌شود، در این حالت اگر  $Branch = 1$  باشد این مقدار توسط ماکس اول انتخاب شده و در این حالت سیگنال Jump Jr صفر هستند و همین مقدار در لبه کلاک درون PC ریخته می‌شود. اگر دستور ما نه Jump Branch و نه Jump کند نیز همان مقدار  $PC_{old} + 2$  درون این رجیستر ریخته می‌شود. اگر از دستور JR استفاده کنیم مقدار این سیگنال ۱ می‌شود و اهمیت ندارد که ورودی صفر ماکس قبل PC چه باشد و تنها مهم این است که مقدار آن رجیستر را از رجیستر فایل خوانده شده را در ورودی اول قرار داده و به درون PC منتقل می‌کند. حال اگر دستور ما Jump مقدار مورد نیاز برای جامپ از طریق ۱ بیت شیفت آدرس به دست آمده و درون PC ریخته می‌شود برای به این شکل که سیگنال  $1 = Jump$  و سیگنال  $0 = JR$  خواهد بود. در مورد انتقال مقدار کنونی PC به جامپ رجیستر نیز در قسمت بعدی توضیح داده شده.

## ۲- پیاده سازی دستور JAL :

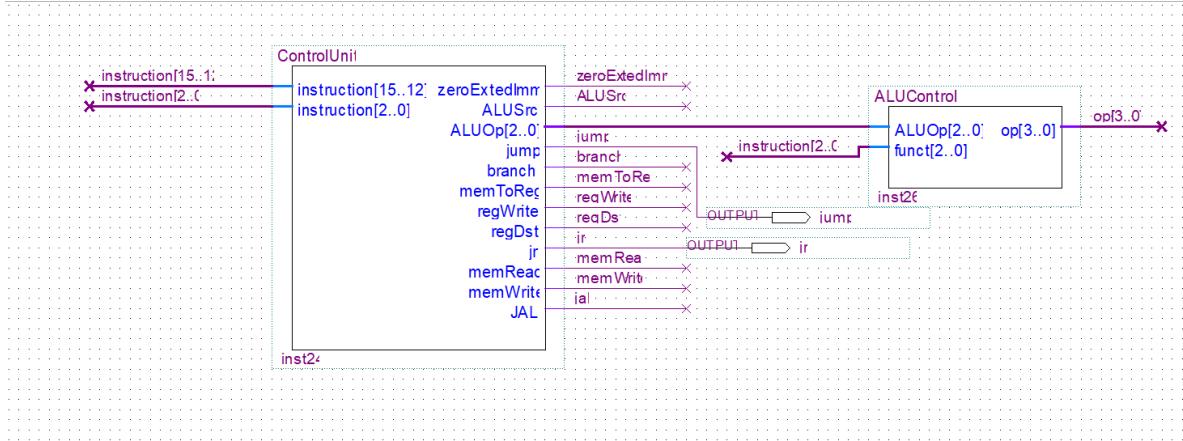
بخش مورد نظر برای جامپ این دستور که مانند حالت قبل است تنها تفاوت این است که باید مقدار  $PC_{old} + 2$  را در یک رجیستر ذخیره کنیم. بدین منظور به قبل ورودی دیتا رجیستر فایل که قرار است آن را در یک ثبات بریزد یک مالتی پلکسراضافه می کنیم که با سیگنال jal سلکت می کند به این شکل که ورودی صفر آن همان حالت قبلی را دارد و ورودی یک آن مانند شکل زیر است :



در این شکل می توانید ببینید که همان  $PC_{old}$  درون یک رجیستر قبل adder ریخته شده که با سیگنال JAL کلاک می خورد تا مقدار  $PC_{curr}$  در زمان محاسبات دچار تغییر نشود و سپس با دو جمع شده و در صورت یک بودن سیگنال JAL به رجیستر فایل برای ذخیره روی ثبات مورد نظر می رود.

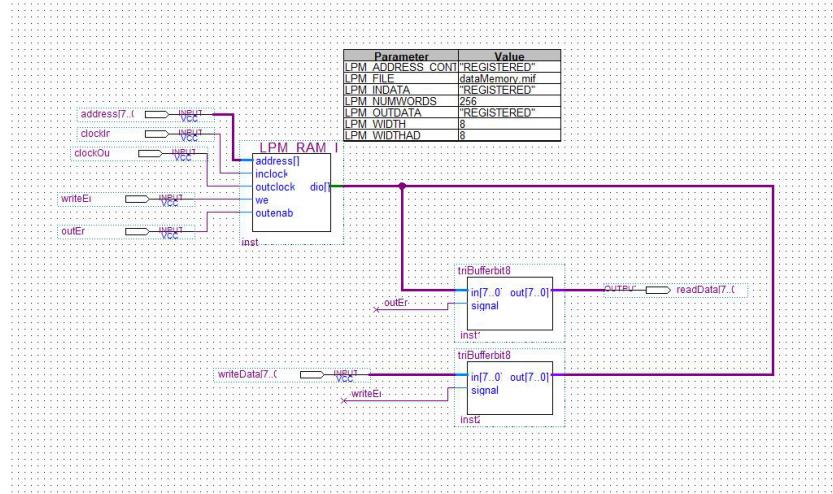
یک نکته وجود دارد که برای اجرای دستورات لود دو دفعه نیاز به نوشتن ماشین کد آن در I-Cache داریم.

### ۳- کنترل سیگنال‌ها :



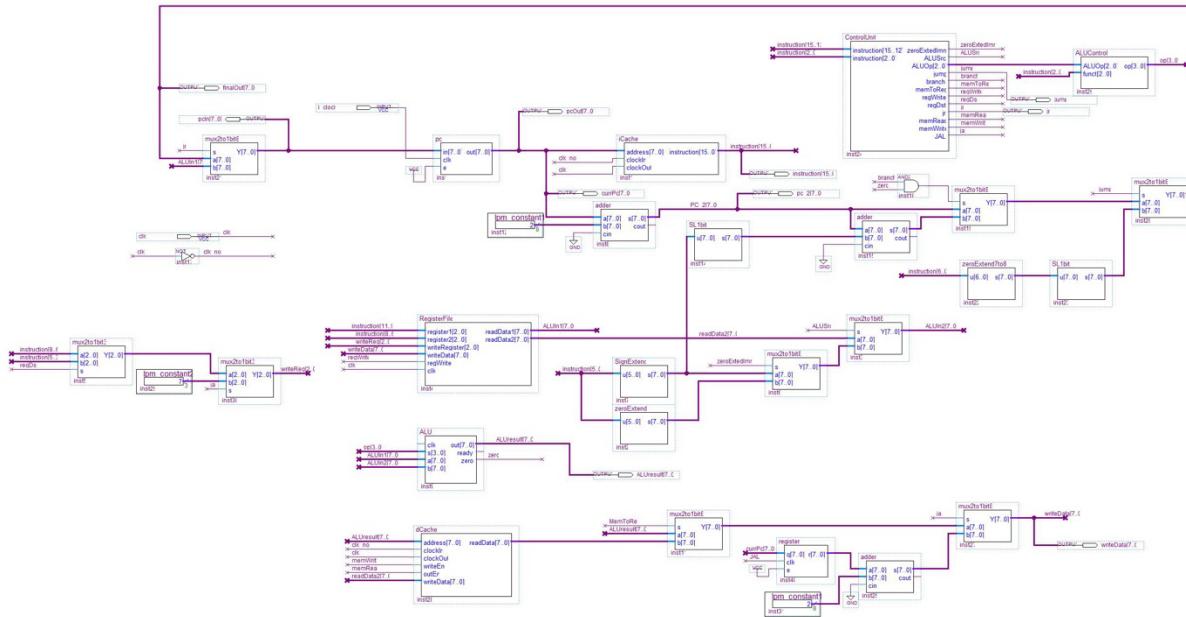
این بخش بدین شکل است که زمانی که دستور از I-Cache خوانده شود به Control Unit که طراحی آن را در قسمت اول دیدیم رفته و سیگنال‌های کنترلی مورد نیاز را به دست می‌آوریم و ALUOp ها به همراه شده که دستور واحد ALU Control وارد function bits می‌شوند. اکنون تمام سیگنال‌ها آماده است.

### ۴- طراحی data memory



پس از تست datapath، تصمیم گرفتیم که ساختار data memory ساخته شده در تمرین قبل را کمی تغییر دهیم. پین دوطرفه data را به دو tri state buffer هشت بیتی وصل می‌کنیم و سیگنال کنترلی آن‌ها را برابر read enable و write enable قرار می‌دهیم. سپس مانند شکل مقابل باس‌ها را به هم وصل می‌کنیم تا بتوانیم به پین دوطرفه، پین ورودی و خروجی وصل کنیم. در بلوك control unit مشخص کردہ‌ایم که دو سیگنال کنترلی هیچ وقت با هم فعال نشوند

در نهایت در شکل پایین نیز می‌توانید این واحد را به طور کامل مشاهده کنید:



تست‌ها:

تست ۱: تابع فیبوناچی

```
#include <stdio.h>

int fibonacci(int a) {
    if(a < 2)
        return a;
    return fibonacci(a-1) + fibonacci(a-2);
}

int main() {
    for(int i = 0; i <= 5; i++)
        printf("%d ", fibonacci(i));

    return 0;
}
```

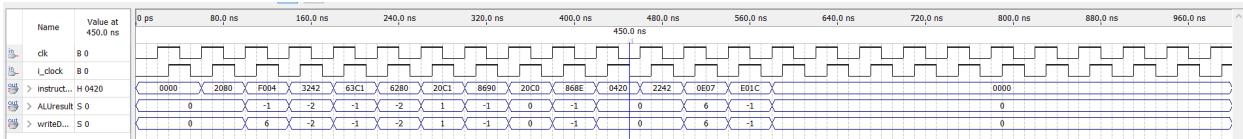
کد بازگشتی فیبوناچی

0 1 1 2 3 5 8 13 21 34 55

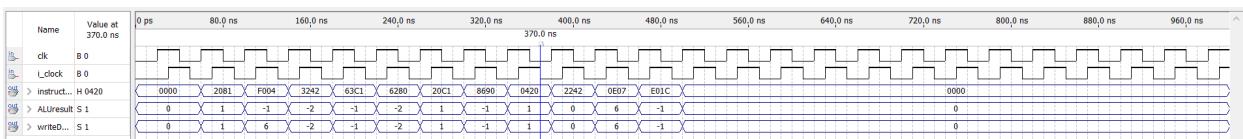
نتیجه اجرای کد فیبوناچی

ADDRESS	CODE(B)	CODE(H)	ASSEMBLY LINE		
00					
02	0010 000 010 000101	2085	addi	r2, r0, 5	
04	1111 XXXXX 0000100	F004	jal	fibonacci	
06	1110 XXXXX 0011100	E01C	j	done	
08	0011 001 001 000010	3242	fibonacci:		r0 = 0
0A	0110 001 111 000001	63C1	subi	r1, r1, 2	r1 = sp
0C	0110 001 010 000000	6280	sb	r7, 1(r1)	r2 = a0
			sb	r2, 0(r1)	r3 = t0
					r4 = v0
0E	0010 000 011 000001	20C1	addi	r3, r0, 1	r5 = s1
10	1000 011 010 010000	8690	beq	r2, r3, base_case	r6
12	0010 000 011 000000	20C0	addi	r3, r0, 0	r7 = ra
14	1000 011 010 001110	868E	beq	r2, r3, base_case	
16	0011 010 010 000001	3481	subi	r2, r2, 1	
18	1111 XXXXX 0000100	F004	jal	fibonacci	
1A	0000 100 000 101 000	0828	add	r5, r4, r0	
1C	0111 001 010 000000	7280	lb	r2, 0(r1)	
1E	0111 001 010 000000	7280	lb	r2, 0(r1)	
20	0011 010 010 000010	3482	subi	r2, r2, 2	
22	1111 XXXXX 0000100	F004	jal	fibonacci	
24	0000 101 100 100 000	0B20	add	r4, r5, r4	
26	0111 001 111 000001	73C1	lb	r7, 1(r1)	
28	0111 001 111 000001	73C1	lb	r7, 1(r1)	
2A	0111 001 010 000000	7280	lb	r2, 0(r1)	
2C	0111 001 010 000000	7280	lb	r2, 0(r1)	
2E	0010 001 001 000010	2242	addi	r1, r1, 2	
30	0000 111 000 000 111	0E07	jr	r7	
32	0000 010 000 100 000	0420	base_case:		
34	0010 001 001 000010	2242	add	r4, r2, r0	
36	0000 111 000 000 111	0E07	addi	r1, r1, 2	
			jr	r7	
38			done:		

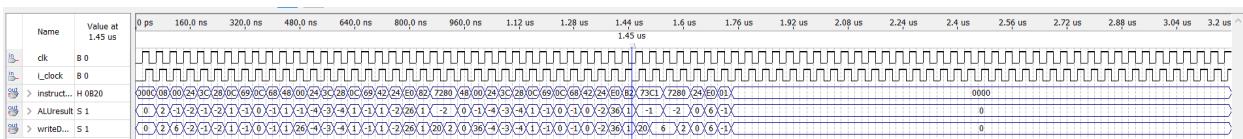
ماشین کد فیبوناچی



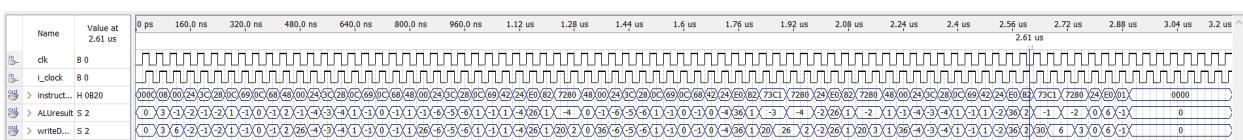
تست ورودی = ٠



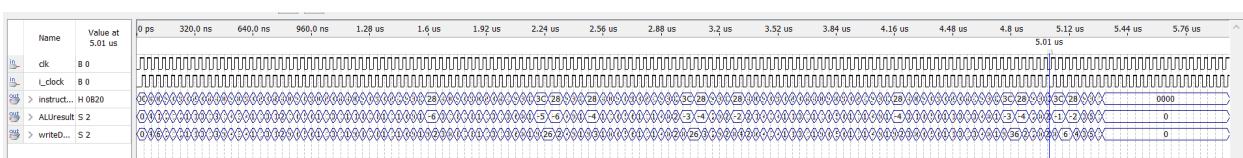
تست ورودی = ١



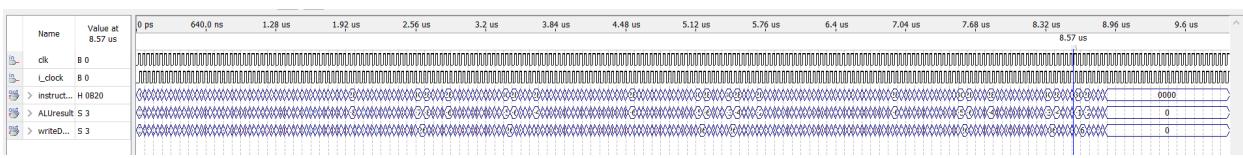
تست ورودی = ٢



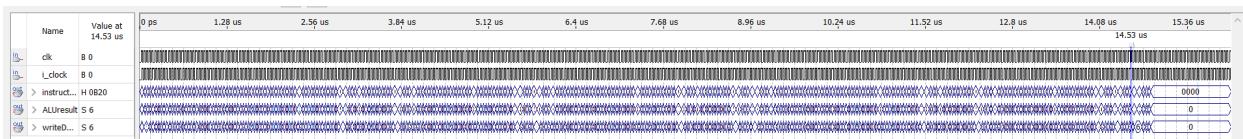
تست ورودی = ٣



تست ورودی = ٤



تست ورودی = ٥



تست ورودی = ٦

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f
000	0000	0000	2085	0000	F004	0000	E01C	0000	3242	0000	63C1	0000	6280	0000	20C1	0000
010	8690	0000	20C0	0000	868E	0000	3481	0000	F004	0000	0828	0000	7280	0000	7280	0000
020	3482	0000	F004	0000	0B20	0000	73C1	0000	73C1	0000	7280	0000	7280	0000	2242	0000
030	0E07	0000	0420	0000	2242	0000	0E07	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
090	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

تصویری از حافظه ستور العمل

در این بخش ابتدا می‌توانید کد C برنامه فیبوناچی را مشاهده کنید. سپس ما این کد را به اسمبلی مورد نیاز برنامه تبدیل کردیم. یک نکته وجود دارد که برای اجرای دستورات لود دو دفعه نیاز به نوشتن ماشین کد آن در I-Cache داریم. استفاده از ثبات هایمان نیز به این شکل است که ثبات صفر همواره مقدار صفر دارد، ثبات ۱ است که پونیتر است، ثبات ۲ برای همان عددی است که فیبوناچی آن را می‌خواهیم حساب کنیم. با توجه به اینکه برنامه ما از فیبوناچی صفر آغاز به کار می‌کند، باید برای بدست آوردن  $(fib(x))$  مقدار  $x+1$  را در ثبات ۲ بریزیم بنابراین در اول برنامه قبل شروع در آن مقدار ۶ را می‌ریزیم.

حال که ماشین کد دستورات این برنامه در پردازنده خود را به دست آورده اند آنها را در I-Cache به شکل یکی در میان ریخته و برای فهمیدن اتمام برنامه نیز به اینستراکشن صفر می‌رسیم که متوجه می‌شویم که برنامه ما درست کار می‌کند و مقدار درست در رجیستر  $\$R4$  ذخیره می‌شود. تست‌های این برنامه از فیبوناچی ۰ تا ۵ در صفحه قبل قرار گرفته شده است. که مقدار به دست آمده در آن با استفاده از خروجی ALUResult مشخص شده که در هر تست در آن نقطه که یک خط آبی از بالای صفحه پایین ۱۰۰۰ مشخص است که در تست اعداد ۰ و ۱ همان مقادیر پایه است که خودشان می‌شود. در ورودی ۲ عدد ۱ و در ورودی ۳ عدد ۲ و در ورودی ۴ عدد ۳ را برمی‌گرداند و در نهایت نیز با ورودی ۵ عدد ۵ که مقدار درست است را برمی‌گرداند.

## تست ۲: تابع ساختگی برای تست همه دستورات

```
#include <stdio.h>

int8_t function(int8_t a) {
    if (a == 0 || a == 1 || a == 2)
        return (a+8)^7;
    else
        return (function(a-2) & 5) + (function(a-3) | 6);
}

int main() {
    for(int8_t i = 0; i < 10; i++)
        printf("%i ", function(i));

    printf("\n");
    printf("%d %d", function(4) & 5, function(3) | 6);

    return 0;
}
```

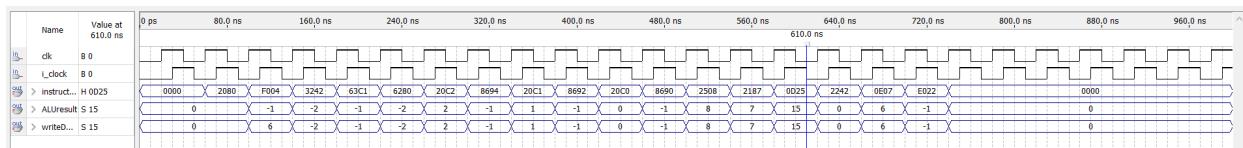
کد تابع ساختگی

15 14 13 19 19 16 24 23 22 35 27

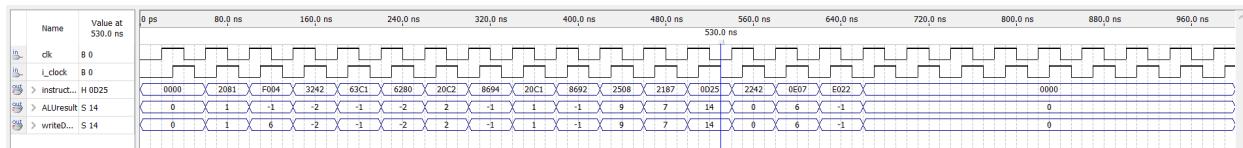
نتیجه اجرای کد ساختگی

ADDRESS	CODE(B)	CODE(H)	ASSEMBLY LINE	
00				
02	0010 000 010 000101	2085	addi r2, r0, 5	
04	1111 XXXXX 0000100	F004	jal function	
06	1110 XXXXX 0100010	E022	j done	
			function:	r0 = 0
08	0011 001 001 000010	3242	subi r1, r1, 2	r1 = sp
0A	0110 001 111 000001	63C1	sb r7, 1(r1)	r2 = a0
0C	0110 001 010 000000	6280	sb r2, 0(r1)	r3 = t0 r5 = s1
0E	0010 000 011 000010	20C2	addi r3, r0, 2	r6
10	1000 011 010 010100	8694	beq r2, r3, base_case	r7 = ra
12	0010 000 011 000001	20C1	addi r3, r0, 1	
14	1000 011 010 010010	8692	beq r2, r3, base_case	
16	0010 000 011 000000	20C0	addi r3, r0, 0	
18	1000 011 010 010000	8690	beq r2, r3, base_case	
1A	0011 010 010 000010	3482	subi r2, r2, 2	
1C	1111 XXXXX 0000100	F004	jal function	
1E	0100 100 100 000101	4905	andi r4, r4, 5	
20	0000 100 000 101 000	0828	add r5, r4, r0	
22	0111 001 010 000000	7280	lb r2, 0(r1)	
24	0111 001 010 000000	7280	lb r2, 0(r1)	
26	0011 010 010 000011	3483	subi r2, r2, 3	
28	1111 XXXXX 0000100	F004	jal function	
2A	0101 100 100 000110	5906	ori r4, r4, 6	
2C	0000 101 100 100 000	0B20	add r4, r5, r4	
2E	0111 001 111 000001	73C1	lb r7, 1(r1)	
30	0111 011 111 000001	73C1	lb r7, 1(r1)	
32	0111 001 010 000000	7280	lb r2, 0(r1)	
34	0111 001 010 000000	7280	lb r2, 0(r1)	
36	0010 001 001 000010	2242	addi r1, r1, 2	
38	0000 111 000 000 111	0E07	jr r7	
			base_case:	
3A	0010 010 100 001000	2508	addi r4, r2, 8	
3C	0010 000 110 000111	2187	addi r6, r0, 7	
3E	0000 110 100 100 101	0D25	xor r4, r6, r4	
40	0010 001 001 000010	2242	addi r1, r1, 2	
42	0000 111 000 000 111	0E07	jr r7	
44			done:	

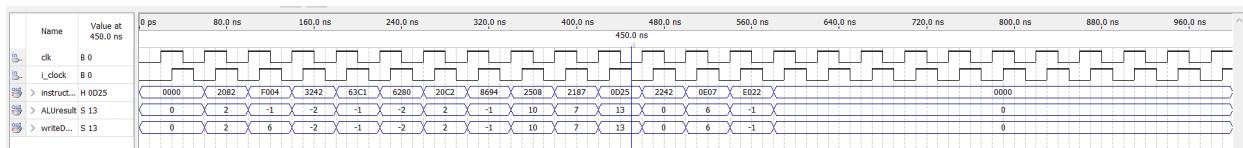
ماشین کد تابع ساختگی



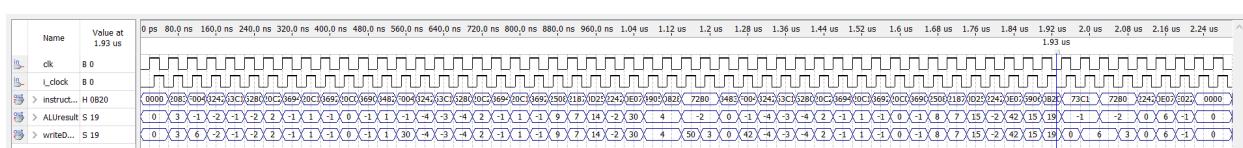
تست ورودی = ٠



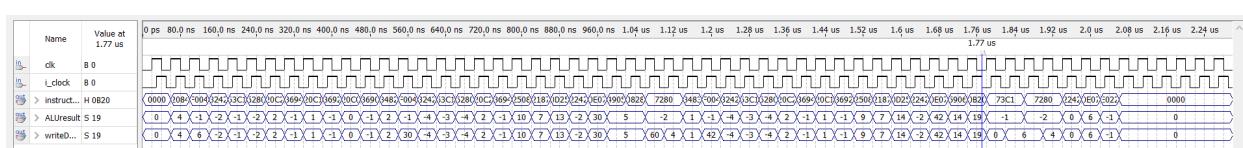
تست ورودی = ١



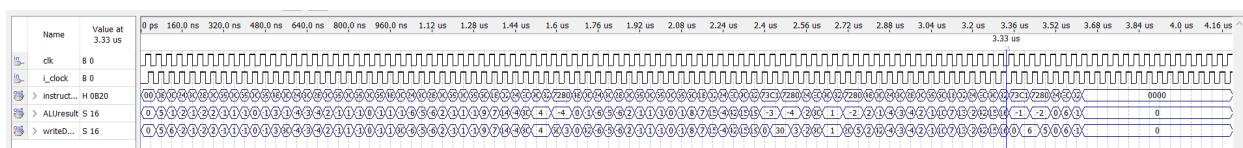
تست ورودی = ٢



تست ورودی = ٣



تست ورودی = ٤



تست ورودی = ٥

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f
000	0000	0000	2085	0000	F004	0000	E022	0000	3242	0000	63C1	0000	6280	0000	20C2	0000
010	8694	0000	20C1	0000	8692	0000	20C0	0000	8690	0000	3482	0000	F004	0000	4905	0000
020	0828	0000	7280	0000	7280	0000	3483	0000	F004	0000	5906	0000	0B20	0000	73C1	0000
030	73C1	0000	7280	0000	7280	0000	2242	0000	0E07	0000	2508	0000	2187	0000	0D25	0000
040	2242	0000	0E07	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
090	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

این برنامه نیز مانند برنامه ابتدایی ما کار می‌کند فقط محاسبات ریاضیاتی بیشتری دارد که در کد C آن می‌توانید کارایی آن را ببینید. در اینجا نیز ثبات‌ها همان وظیفه قبلی را دارند و ما با تبدیل برنامه اسمبلی به ماشین کد تونستیم که این برنامه را نیز اجرا کنیم. تست‌های این برنامه نیز برای اعداد ۰ تا ۵ انجام شده و در صفحه قبل می‌توانید مشاهده کنید که نتیجه نهایی در نقطه‌ای که خط آبی از بالا صفحه پایین آمده در قرار دارد که با مقادیر به دست آمده از کد C تطابق دارد.

تست ۳: باقیمانده برابر ۳

```
#include <stdio.h>

int mod(int x) {
    if(x < 3)
        return x;
    return mod(x-3);
}

int main() {
    for(int i = 0; i < 18; i++)
        printf("%d ", mod(i));
    return 0;
}
```

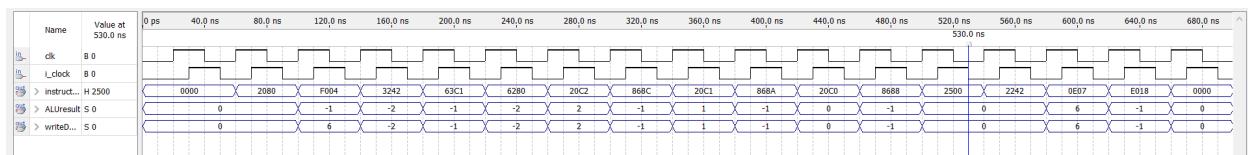
کد باقیمانده برابر ۳

0 1 2 0 1 2 0 1 2 0 1 2 0 1 2

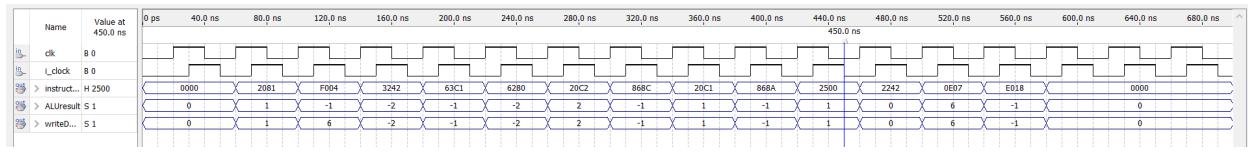
نتیجه اجرای کد باقیمانده برابر ۳

ADDRESS	CODE(B)	CODE(H)	ASSEMBLY LINE		
00					
02	0010 000 010 000101	2085	addi	r2, r0, 5	
04	1111 XXXXX 0000100	F004	jal	mod	
06	1110 XXXXX 0011000	E018	j	done	
			mod:		
08	0011 001 001 000010	3242	subi	r1, r1, 2	r0 = 0
0A	0110 001 111 000001	63C1	sb	r7, 1(r1)	r1 = sp
0C	0110 001 010 000000	6280	sb	r2, 0(r1)	r2 = a0
					r3 = t0
					r4 = v0
0E	0010 000 011 000010	20C2	addi	r3, r0, 2	r5 = s1
10	1000 011 010 001100	868C	beq	r2, r3, base_case	r6
12	0010 000 011 000001	20C1	addi	r3, r0, 1	r7 = ra
14	1000 011 010 001010	868A	beq	r2, r3, base_case	
16	0010 000 011 000000	20C0	addi	r3, r0, 0	
18	1000 011 010 001000	8688	beq	r2, r3, base_case	
1A	0011 010 010 000011	3483	subi	r2, r2, 3	
1C	1111 XXXXX 0000100	F004	jal	mod	
1E	0111 001 111 000001	73C1	lb	r7, 1(r1)	
20	0111 011 111 000001	73C1	lb	r7, 1(r1)	
22	0111 001 010 000000	7280	lb	r2, 0(r1)	
24	0111 001 010 000000	7280	lb	r2, 0(r1)	
26	0010 001 001 000010	2242	addi	r1, r1, 2	
28	0000 111 000 000 111	0E07	jr	r7	
			base_case:		
2A	0010 010 100 000000	2500	addi	r4, r2, 0	
2C	0010 001 001 000010	2242	addi	r1, r1, 2	
2E	0000 111 000 000 111	0E07	jr	r7	

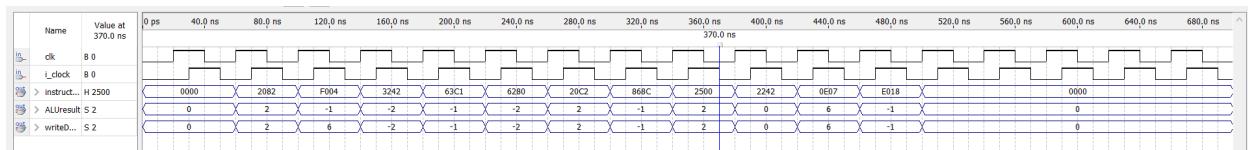
ماشین کد تابع باقیمانده برابر ۳



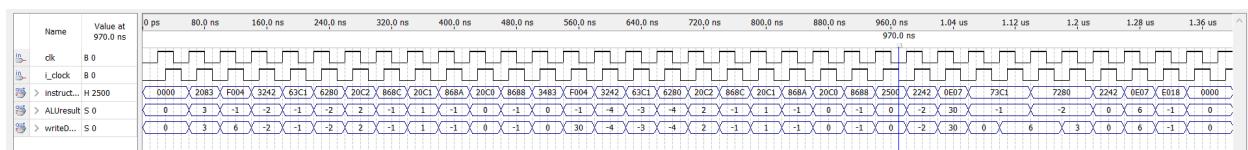
تست ورودی ۰



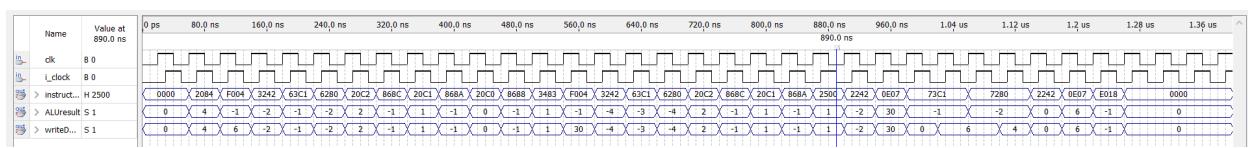
تست ورودی ۱



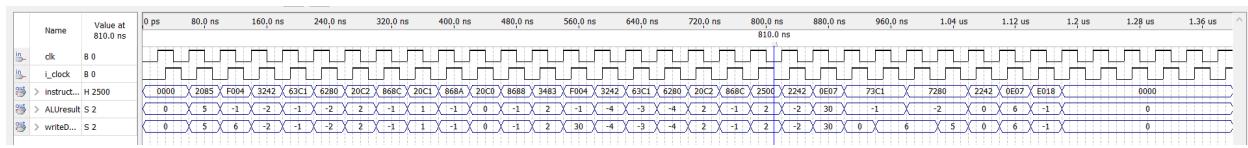
تست ورودی ۲



تست ورودی ۳



تست ورودی ۴



تست ورودی ۵

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f
000	0000	0000	2085	0000	F004	0000	E018	0000	3242	0000	63C1	0000	6280	0000	20C2	0000
010	868C	0000	20C1	0000	868A	0000	20C0	0000	8688	0000	3483	0000	F004	0000	73C1	0000
020	73C1	0000	7280	0000	7280	0000	2242	0000	0E07	0000	2500	0000	2242	0000	0E07	0000
030	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
090	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

در این برنامه به سراغ محاسبه باقیمانده اعداد بر ۳ رفتیم که کد C آن را می‌توانید در صفحه اول مشاهده کنید، سپس این کد را به کد اسمبلی تبدیل کرده (با استفاده از کارایی ثبات‌ها مانند مثال‌های قبل). حال با تبدیل این برنامه‌ها به ماشین کد توانایی ذخیره آن روی حافظه دستورالعمل را داریم. تصویر این صفحه همان حافظه دستورالعمل است که با دستورات این برنامه تکمیل شده است. حال با اجرای این برنامه آن را تست کرده که تست آن برای اعداد ۰ تا ۵ را در صفحه قبل می‌توانید ببینید که همان طور که در مثال‌های قبل گفته شد مقدار آن را با خروجی ALUResult در نقطه‌ای که خط آبی جدا کننده وجود دارد مشخص می‌کند که حالات پایه ۰ تا ۲ به ترتیب همان مقدار خود را خروجی داده‌اند و در ورودی‌های ۳ تا ۵ نیز آن مقادیر دوباره به دست آمده که کاملاً درست است.

```
#include <stdio.h>
```

```
int divide(int x) {
    if(x < 3)
        return 0;
    return divide(x-3) + 1;
}
```

```
int main() {
    for(int i = 0; i <= 21; i++)
        printf("%d ", divide(i));
    return 0;
}
```

تست ۴: خارج قسمت بر ۳

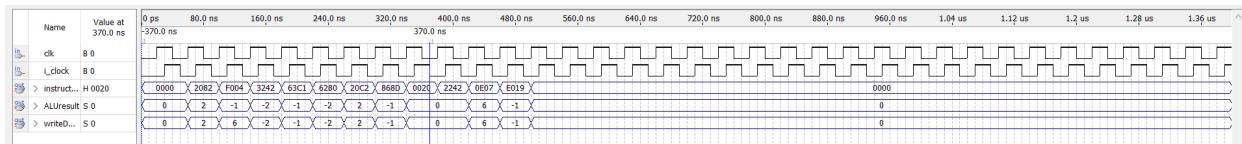
```
0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6
```

نتیجه اجرای کد خارج قسمت بر ۳

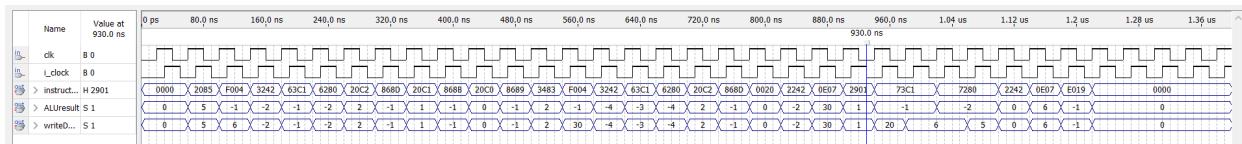
کد خارج قسمت بر ۳

ADDRESS	CODE(B)	CODE(H)	ASSEMBLY LINE		
00					
02	0010 000 010 000101	2085	addi	r2, r0, 5	
04	1111 XXXXX 000100	F004	jal	divide	
06	1110 XXXXX 0011001	E019	j	done	
08	0011 001 001 000010	3242			divide: r0 = 0
0A	0110 001 111 000001	63C1	subi	r1, r1, 2	r1 = sp
0C	0110 001 010 000000	6280	sb	r7, 1(r1)	r2 = a0
			sb	r2, 0(r1)	r3 = t0
					r4 = v0
0E	0010 000 011 000010	20C2	addi	r3, r0, 2	r5 = s1
10	1000 011 010 001101	868D	beq	r2, r3, base_case	r6
12	0010 000 011 000001	20C1	addi	r3, r0, 1	r7 = ra
14	1000 011 010 001011	868B	beq	r2, r3, base_case	
16	0010 000 011 000000	20C0	addi	r3, r0, 0	
18	1000 011 010 001001	8689	beq	r2, r3, base_case	
1A	0011 010 010 000011	3483	subi	r2, r2, 3	
1C	1111 XXXXX 000100	F004	jal	divide	
1E	0010 100 100 000001	2901	addi	r4, r4, 1	
20	0111 001 111 000001	73C1	lb	r7, 1(r1)	
22	0111 011 111 000001	73C1	lb	r7, 1(r1)	
24	0111 001 010 000000	7280	lb	r2, 0(r1)	
26	0111 001 010 000000	7280	lb	r2, 0(r1)	
28	0010 001 001 000010	2242	addi	r1, r1, 2	
2A	0000 111 000 000 111	0E07	jr	r7	
				base_case:	
2C	0000 000 000 100 000	0020	add	r4, r0, r0	
2E	0010 001 001 000010	2242	addi	r1, r1, 2	
30	0000 111 000 000 111	0E07	jr	r7	
32				done:	

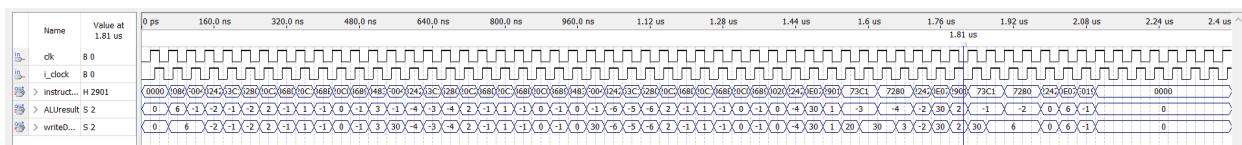
ماشین کد تابع خارج قسمت بر ۳



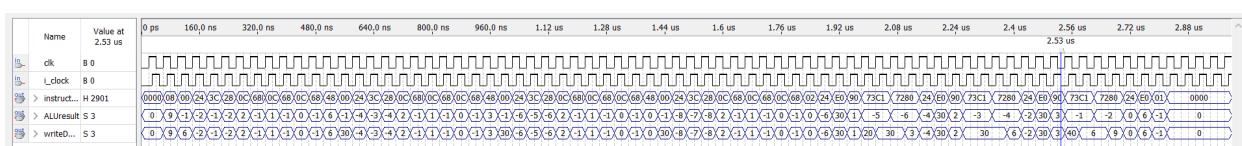
نست ورودی = ۲



تست ورودی = ۵



تست ۱۹ (۵۵)



9 - 162010 (cont'd)

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f
000	0000	0000	2089	0000	F004	0000	E019	0000	3242	0000	63C1	0000	6280	0000	20C2	0000
010	868D	0000	20C1	0000	868B	0000	20C0	0000	8689	0000	3483	0000	F004	0000	2901	0000
020	73C1	0000	73C1	0000	7280	0000	7280	0000	2242	0000	0E07	0000	0020	0000	2242	0000
030	0E07	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
090	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

این برنامه نیز به شدت شبیه برنامه قبلی است تنها با این تفاوت که در این برنامه به سراغ محاسبه خارج قسمت اعداد در تقسیم بر ۳ رفتیم که کد C آن را می‌توانید در صفحه اول مشاهده کنید، سپس این کد را به کد اسمنبلی تبدیل کرده (با استفاده از کارایی ثبات‌ها مانند مثال‌های قبل). حال با تبدیل این برنامه‌ها به ماشین کد توانایی ذخیره آن روی حافظه دستورالعمل را داریم. تصویر این صفحه همان حافظه دستورالعمل است که با دستورات این برنامه تکمیل شده است. حال با اجرای این برنامه آن را تست کرده که تست آن برای اعداد ۲ ، ۵ ، ۶ و ۹ را در صفحه قبل می‌توانید ببینید که همان طور که در مثال‌های قبل گفته شد مقدار آن را با خروجی ALUResult در نقطه‌ای که خط آبی جدا کننده وجود دارد مشخص می‌کند. برای ورودی ۲ خروجی صفر برای ورودی ۵ خروجی ۱ برای ورودی ۶ خروجی ۲ و برای ورودی ۹ خروجی ۳ به دست می‌آید.