CA #3 report

محمدعلی زارع 810197626
حمیدرضا خدادادی 810197499

- **آپکود ها استاندارد هستند به جز jr که 000001 است.**

- **مسیرداده :**

ساختار کلی مسیرداده همانند مسیر داده نشان داده شده در درس است و دستورات گفته در درس همانطور پیاده سازی شده اند.
برای pc بعدی یک mux چهار ورودی قرار داده شده که به جز ورودی pc + 4 ، یک ورودی آن مستقیم از خروجی رجیستر A برای دستور jr می آید. ورودی دیگر آن برای دستور j / jal است که 4 بیت اول pc [31:28] را به اول آدرس آن و 2 صفر به آخر آن اضافه شده است و ورودی آخر برای دستور beq / bne است که آفست داده شده در دستور با pc + 4 جمع میشود و وارد آن میشود. این mux با سیگنال pc_src کنترل میشود.
همینطور برای mux پشت write_register نیز یک ورودی عدد ثابت 31 برای دستور jal در نظر گرفته شده است.
به mux ورودی write_data رجیستر هم یک ورودی اضافه شده که pc + 4 در آن قرار میگیرد. این ورودی برای دستور jal است تا آدرس دستور بعدی را در رجیستر 31 ذخیره کرد.

- **کنترلر :**

کنترلر با توجه به آپکود (6 بیت اول دستور) سیگنال های مورد نیاز را ارسال میکند. کنترلر یک زیرماژول هم برای سیگنال مورد نیاز ALU دارد که در زیرماژول ها توضیح داده شده است.
سیگنال zero از ALU نیز وارد کنترلر میشود تا در تصمیم گیری pc_src برای دستورهای beq / bne استفاده شود.
6 بیت آخر هم به عنوان func وارد کنترلر میشود تا در صورتی که دستور از نوع R-Type بود در زیرماژول Alu_controller استفاده شود.
سیگنال ها نیز در جدول نشان داده شده اند.
استیت ها همانند استیت نشان داده شده در ویدیو درس است. برای دستور bne یک استیت همانند استیت beq اضافه شده با این تفاوت که در صورت صفر بودن سیگنالzero ، سیگنال pc_write فعال می‌شود. برای addi و andi نیز دو هرکدام یک استیت اضافه شده که ورودی mux A را 1 میکند تا مقدار رجیستر وارد شود و ورودی B را 10 میکند تا 16 بیت sign extend شده آخر دستور وارد شود. عمل Alu هم با توجه به دستور + یا & می‌شود. یک استیت مشابه هم بعد این دو استیت قرار دارد تا خروجی رجیستر Alu در رجیستر داده شده نوشته شود.
برای jr هم یک استیت اضافه شده تا mux pc_src مقدار 11 داشته باشد که خروجی رجیستر A که مقدار رجیستر است وارد شود و با سیگنال pc_write دستور بعدی مشخص شود .
دستور jal نیز همانند روشی که در ویدیو گفته شد اضافه شده است.

زیرماژول ها :

- DataMem : از reg دو بعدی برای ذخیره دستورها و دیتاها استفاده شده (512 تا 32 reg بیتی) .دستورها و دیتاها از فایل memroy.data خوانده میشوند و در این ساختمان داده

CA #3 report

ریخته میشوند. ورودی ماژول آدرس 32 بیتی است که با تقسیم به 4 کردن آن، ایندکس در آرایه به درست می آید و آن را در خروجی قرار میدهد. با خوردن کلاک در صورت بودن سیگنال mem_write ورودی write_data در آدرس ریخته میشود.
در صورت وجود سیگنال mem_read نیز داده موجود در آدرس بر روی خروجی read_data قرار میگیرد.

- RegFile : از reg دو بعدی (32 تا 32 بیتی) برای نگه داری استفاده شده است.
indexingدر این ماژول همان عدد ورودی است . با خوردن کلاک در صورت وجود سیگنال reg_write، دیتای write_data در رجیستر شماره write_reg_address ریخته میشود.
خروجی های read_data نیز همیشه محتویات آدرس های read_reg 1 / 2 را نشان میدهند.
مقدار R0 نیز همیشه صفر باقی میماند.

- PC : خروجی آن نشان دهنده آدرس دستور بعدی است. در صورت وجود سیگنال rst مقدار آن صفر میشود و با هر بار اجرای دستور و عملی نیز مقدار next_pc را از mux قبل خود دریافت و در خروجی خود قرار میدهد.

- ALU : عمل مورد نظر که با alu_op مشخص شده و از ALU_controller دریافت میکند را روی دو ورودی 32 بیتی خود اعمال میکند و در خروجی قرار میدهد. همینطور در صورت صفر بودن خروجی سیگنال zero را به بیرون میدهد تا در کنترلر استفاده شود . عمل های ALU در جدول مشخص شده است.

- ALU_Controller : با دریافت 6 بیت func و دو بیت alu_case عمل موردنیاز ALU را مشخص میکند . حالت های آن در جدول آمده است.

- Shift2 : ورودی خود را دو بیت به سمت چپ شیفت میدهد.

- SignExtend : ورودی 16 بیت میگیرد و با رعایت علامت آن را به 32 بیت تبدیل میکند. (برای استفاده آفست دستور beq / bne در ALU)

- MUX : چهار نوع mux استفاده شده است.

1. 4 ورودی 32 بیت برای ورودی PC و ورودی دوم ALU
2. 2 ورودی 32 بیت برای خروجی PC و ورودی اول ALU
3. 3 ورودی 5 بیتی برای ورودی آدرس write_reg
4. 3 ورودی 32 بیتی برای ورودی write_data ی رجیستر

مسیرداده و کنترلر در ماژول Mips32 به هم متصل شده اند.

CA #3 report

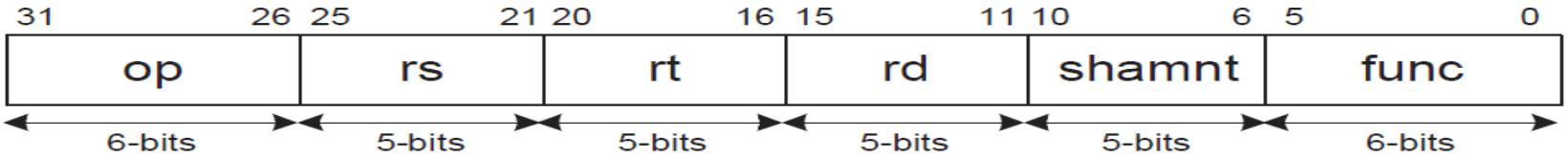
برای تست برنامه :

می توانید نام فایل مموری را که در ماژول DataMem که در حال حاضر memory.data است و به صورت دیفالت محتوای برنامه ی اول را اجرا میکند به memory_Q2.data (برای برنامه ی دوم) یا تست خودتان تغییر دهید و نتیجه ی برنامه را چک کنید .

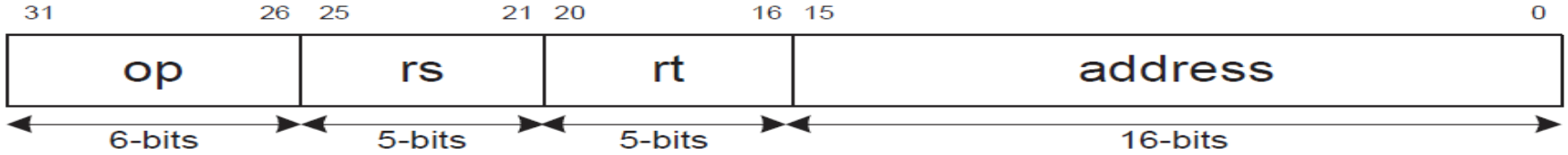
یا میتوانید نام فایل را در ماژول DataMem ثابت نگه دارید و نام اصلی برنامه ی دوم یا تست خودتان را به memory.data تغییر بدهید و برنامه را تست کرده و نتیجه را مشاهده فرمایید .

Arithmetic/Logical Instructions: add, sub, and, or, slt, addi, andi
Memory Reference Instruction: lw, sw
Control Flow Instructions: j, jal, jr, beq, bne

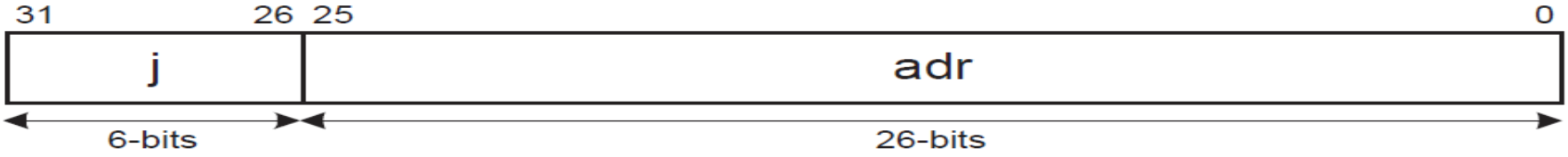
- **R type format : add-sub-and-or-slt-jr**

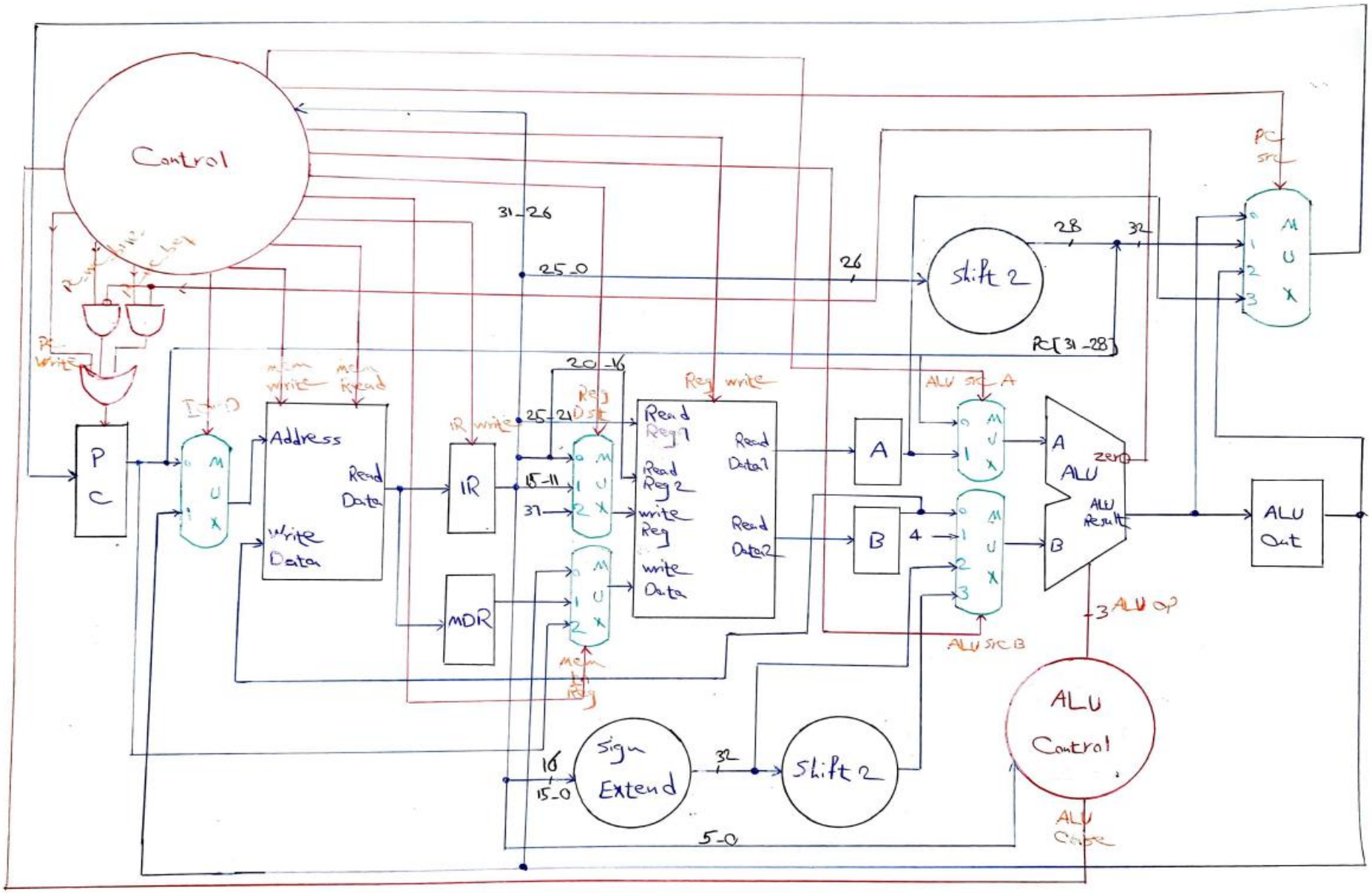


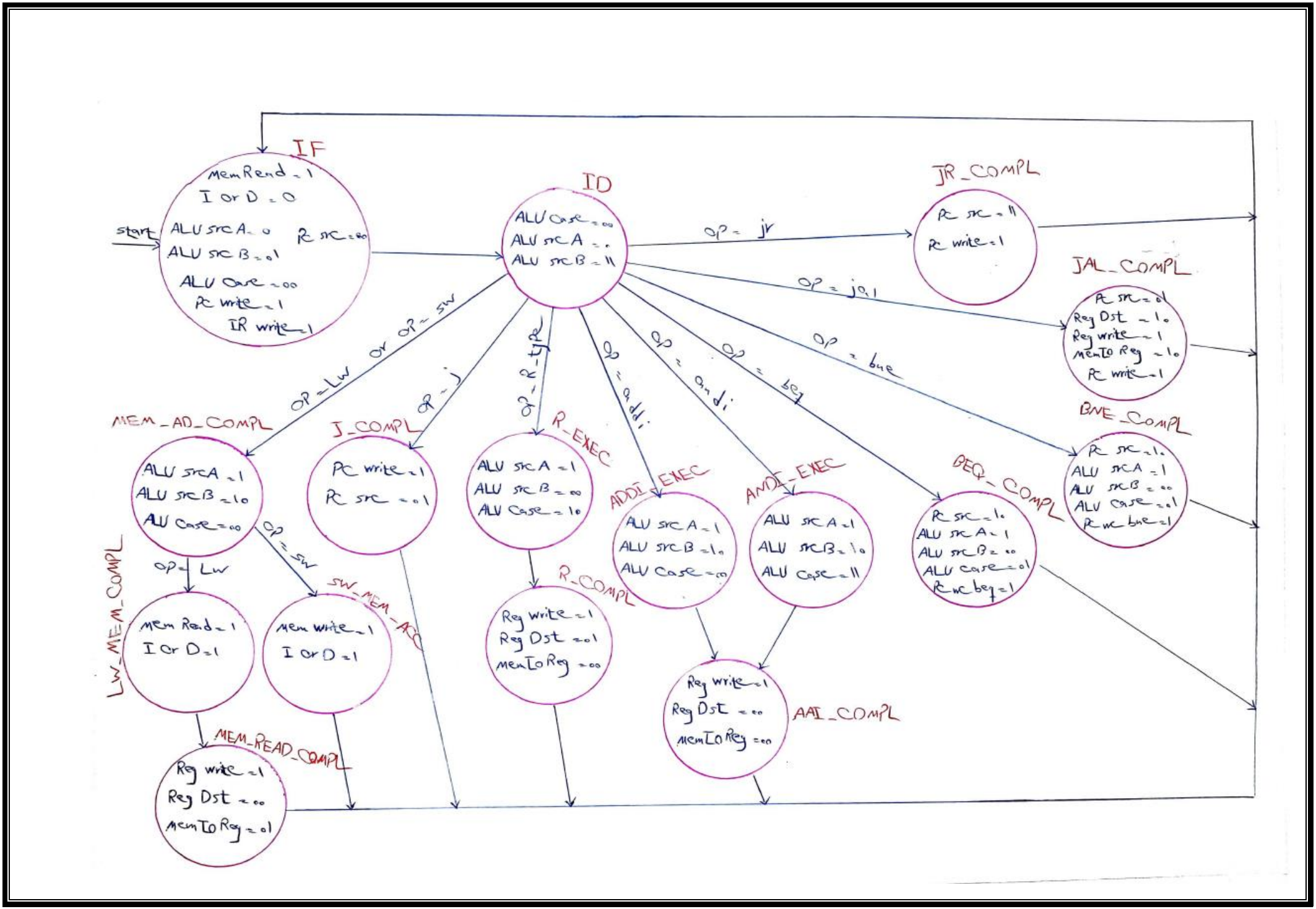
- **I type format : lw-sw-beq-bne-addi-andi**



- **J type format : j-jal**




Control
31-26
25_0
26
Shift 2
28
32
PC src
PC write
Mem write
Mem Read
IR write
Reg Dst
Reg write
ALU src A
20-16
25-21
15-11
3:1
Address
Read Data
Write Data
PC
IR
MDR
Read Reg1
Read Reg2
write Reg
write Data
Read Data1
Read Data2
A
B
4
ALU
zero
ALU Result
ALU Out
ALU src B
3 ALU op
Mem to Reg
16
15_0
Sign Extend
32
Shift 2
ALU Control
ALU Code
5_0


IF
MemRead = 1
I or D = 0
ALU srcA = 0
ALU srcB = 01
PC src = 00
ALU Case = 00
PC write = 1
IR write = 1
start
ID
ALU Case = 00
ALU srcA = 0
ALU srcB = 11
JR_COMPL
PC src = 11
PC write = 1
op = jr
JAL_COMPL
PC src = 01
Reg Dst = 10
Reg write = 1
MemTo Reg = 10
PC write = 1
op = jal
BNE_COMPL
PC src = 10
ALU srcA = 1
ALU srcB = 00
ALU Case = 01
PC wc bne = 1
op = bne
BEQ_COMPL
PC src = 10
ALU srcA = 1
ALU srcB = 00
ALU case = 01
PC wc beq = 1
op = beq
MEM_AD_COMPL
ALU srcA = 1
ALU srcB = 10
ALU Case = 00
op = Lw or op = sw
J_COMPL
PC write = 1
PC src = 01
op = j
R_EXEC
ALU srcA = 1
ALU srcB = 00
ALU Case = 10
op = R_type
ADDI_EXEC
ALU srcA = 1
ALU srcB = 10
ALU Case = 00
op = addi
ANDI_EXEC
ALU srcA = 1
ALU srcB = 10
ALU Case = 11
op = andi
Lw_MEM_COMPL
MemRead = 1
I or D = 1
op = Lw
SW_MEM_ACC
MemWrite = 1
I or D = 1
op = sw
MEM_READ_COMPL
Reg write = 1
Reg Dst = 00
MemTo Reg = 01
R_COMPL
Reg write = 1
Reg Dst = 01
MemTo Reg = 00
AAI_COMPL
Reg write = 1
Reg Dst = 00
MemTo Reg = 00

Control Signals After Detecting The Opcode

Instruction	Type	Opcode	Func	ALU Op	I or D	ALU Src A	ALU Src B	Reg Dst	Reg Write	Mem To Reg	Mem Read	Mem Write	PC src	PC write	PC WC beq	PC WC bne
add	R	000000	100000	10	-	1	00	01	1	00	-	-	-	-	-	-
sub	R	000000	100010	10	-	1	00	01	1	00	-	-	-	-	-	-
and	R	000000	100100	10	-	1	00	01	1	00	-	-	-	-	-	-
or	R	000000	100101	10	-	1	00	01	1	00	-	-	-	-	-	-
slt	R	000000	101010	10	-	1	00	01	1	00	-	-	-	-	-	-
jr	R	000001 (000000)	001000	-	-	-	-	-	-	-	-	-	11	1	-	-
lw	I	100011	NA	00	1	1	10	00	1	01	1	-	-	-	-	-
sw	I	101011	NA	00	1	1	10	-	-	-	-	1	-	-	-	-
addi	I	001000	NA	00	-	1	10	00	1	00	-	-	-	-	-	-
andi	I	001100	NA	11	-	1	10	00	1	00	-	-	-	-	-	-
beq	I	000100	NA	01	-	1	00	-	-	-	-	-	10	-	1	-
bne	I	000101	NA	01	-	1	00	-	-	-	-	-	10	-	-	1
j	J	000010	NA	-	-	-	-	-	-	-	-	-	01	1	-	-
Jal	J	000011	NA	-	-	-	-	10	1	10	-	-	01	1	-	-

ALU OPERATION

000	and
001	or
010	add
110	sub
111	slt

ALU CASE

00	lw - sw - addi
01	beq-bne
10	and - or - add - sub - slt
11	andi

PC SRC SEL

00	PC + 4
01	j - jal
10	beq - bne
11	jr

I OR D

0	Pc
1	Alu out

MEM TO REG SEL

00	ALU out
01	MDR out
10	PC + 4

REG DST SEL

00	[20:16] instruction
01	[15:11] instruction
10	5'b31

ALU SRC A

0	Pc
1	A out

ALU SRC B

00	B out
01	32'b4
10	Sign extend out
11	Shift out

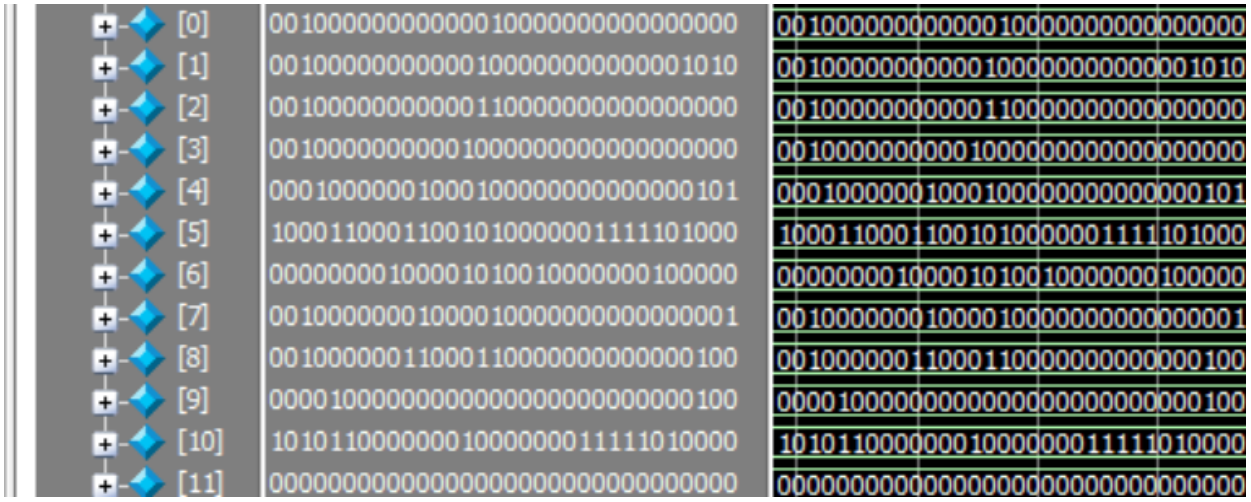
برنامه ی اول : دستورات و آرایه ی اعداد در مموری

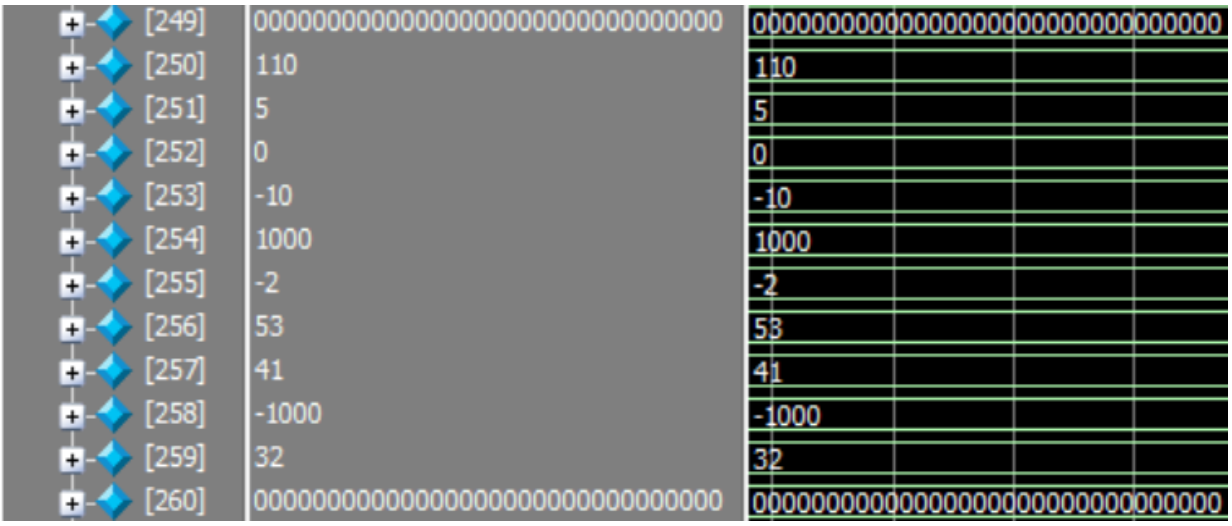
- 110 + 5 + 0 + (-10) + 1000 + (-2) + 53 +41 + (-1000) + 32 = 229

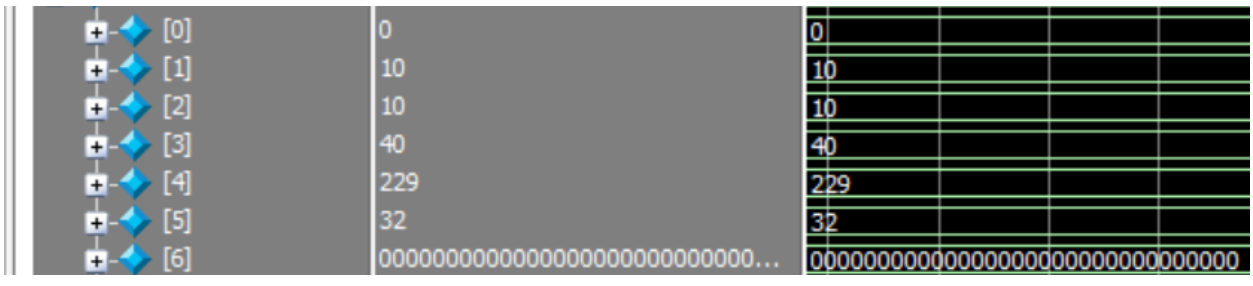
```
1   001000_00000_00001_0000000000000000    // addi R1, R0, 0 #loop_counter
2   001000_00000_00010_0000000000001010    // addi R2, R0, 10 #loop_end
3   001000_00000_00011_0000000000000000    // addi R3, R0, 0 #address_counter
4   001000_00000_00100_0000000000000000    // addi R4, R0, 0 #sum
5   000100_00001_00010_0000000000000101    // LOOP : beq R1, R2, END_LOOP
6   100011_00011_00101_0000001111101000    //        lw R5, 1000(R3)
7   000000_00100_00101_00100_00000_100000  //        addi R4, R4, R5
8   001000_00001_00001_0000000000000001    //        addi R1, R1, 1
9   001000_00011_00011_0000000000000100    //        addi R3, R3, 4
10  000010_00000000000000000000000100      //        j LOOP
11  101011_00000_00100_0000011111010000    // END_LOOP : sw R4, 2000(R0)
12  00000000000000000000000000000000
```

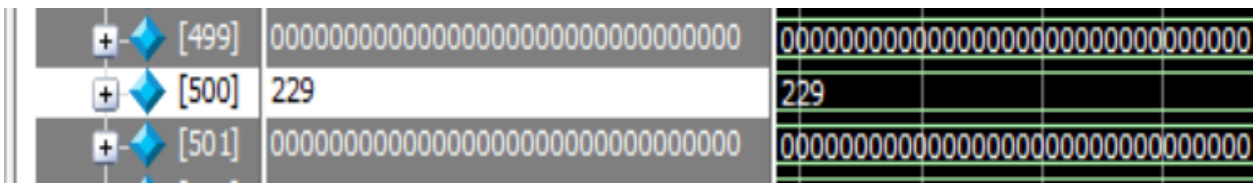
```
249   00000000000000000000000000000000
250   00000000000000000000000000000000
251   00000000000000000000000001101110    //110
252   00000000000000000000000000000101    //5
253   00000000000000000000000000000000    //0
254   11111111111111111111111111110110    //-10
255   00000000000000000000001111101000    //1000
256   11111111111111111111111111111110    //-2
257   00000000000000000000000000110101    //53
258   00000000000000000000000000101001    //41
259   11111111111111111111110000011000    //-1000
260   00000000000000000000000000100000    //32
261   00000000000000000000000000000000
```

در عکس اول محتوای حافظه برای دستورات و در عکس دوم محتوای حافظه برای آرایه ی اعداد (آرایه ی 10 تایی و با شروع از خانه ی 1000 حافظه) و عکس سوم محتوای رجیستر های استفاده شده در دستورات در پایان اجرا و عکس چهارم محتوای خانه ی 2000 حافظه و خروجی برنامه :









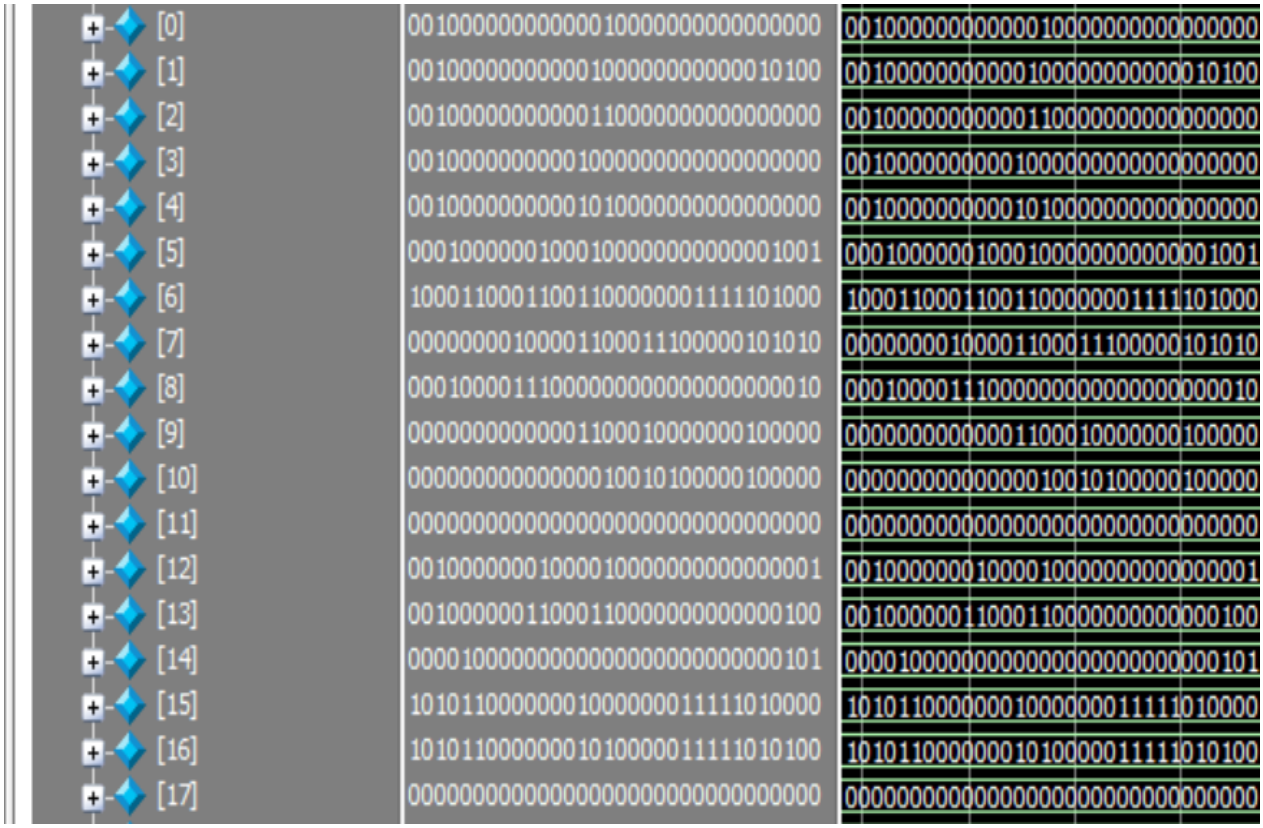
برنامه ی دوم : دستورات و آرایه ی اعداد در ممووری

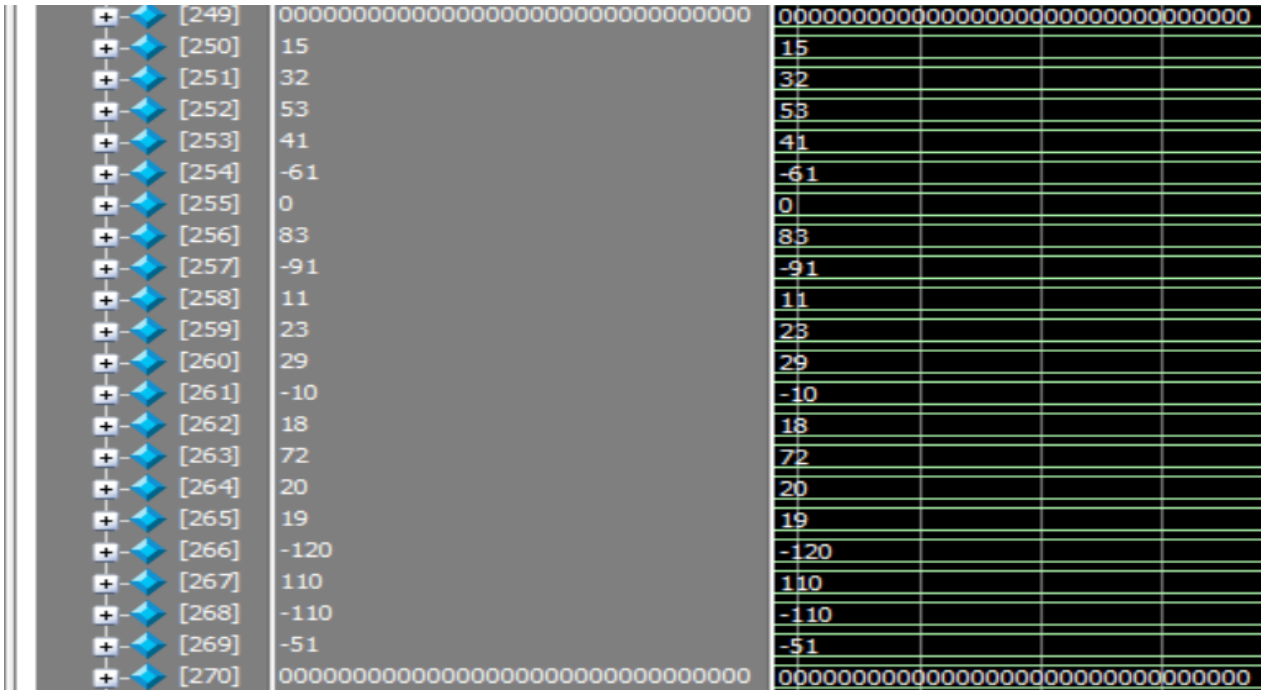
- **MAX = 110 and index of MAX = 17**

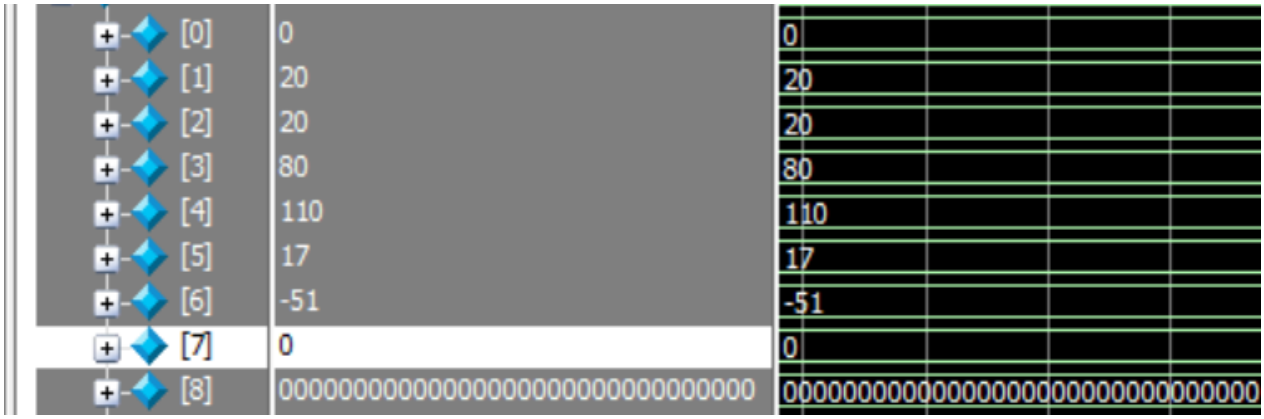
```
1   001000_00000_00001_0000000000000000    // addi R1, R0, 0 #loop_counter
2   001000_00000_00010_0000000000010100    // addi R2, R0, 20 #loop_end
3   001000_00000_00011_0000000000000000    // addi R3, R0, 0 #address_counter
4   001000_00000_00100_0000000000000000    // addi R4, R0, 0 #max
5   001000_00000_00101_0000000000000000    // addi R5, R0, 0 #max_index
6   000100_00001_00010_0000000000001001    // Loop : beq R1, R2, END_LOOP
7   100011_00011_00110_0000001111101000    //        lw R6, 1000(R3) #data
8   000000_00100_00110_00111_00000_101010  //        slt R7, R4, R6 #max < current
9   000100_00111_00000_0000000000000010    //        bqe R7, R0, END_DO_UPDATE
10  000000_00000_00110_00100_00000_100000  // DO_UPDATE : add R4, R0, R6
11  000000_00000_00001_00101_00000_100000  //             add R5, R0, R1
12  00000000000000000000000000000000       // END_DO_UPDATE
13  001000_00001_00001_0000000000000001    //        addi R1, R1, 1
14  001000_00011_00011_0000000000000100    //        addi R3, R3, 4
15  000010_00000000000000000000000101      //        j LOOP
16  101011_00000_00100_0000011111010000    // END_LOOP : sw R4, 2000(R0)
17  101011_00000_00101_0000011111010100    //            sw R5, 2004(R0)
18  00000000000000000000000000000000
```

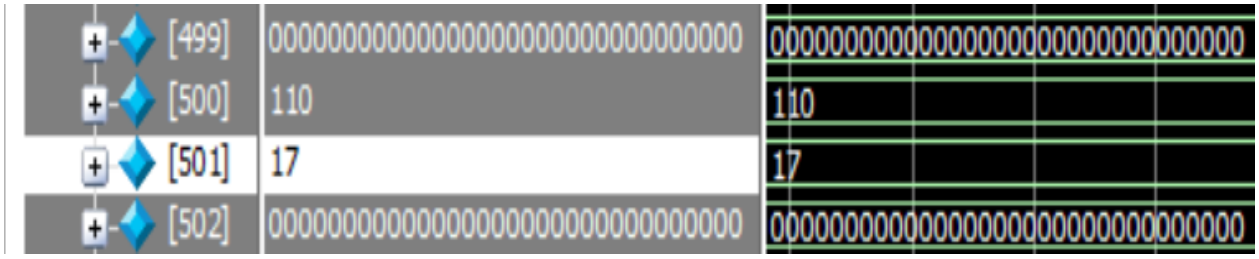
```
250  00000000000000000000000000000000
251  00000000000000000000000000001111    //15
252  00000000000000000000000000100000    //32
253  00000000000000000000000000110101    //53
254  00000000000000000000000000101001    //41
255  11111111111111111111111111000011    //-61
256  00000000000000000000000000000000    //0
257  00000000000000000000000001010011    //83
258  11111111111111111111111110100101    //-90
259  00000000000000000000000000001011    //11
260  00000000000000000000000000010111    //23
261  00000000000000000000000000011101    //29
262  11111111111111111111111111110110    //-10
263  00000000000000000000000000010010    //18
264  00000000000000000000000001001000    //72
265  00000000000000000000000000010100    //20
266  00000000000000000000000000010011    //19
267  11111111111111111111111110001000    //-120
268  00000000000000000000000001101110    //110
269  11111111111111111111111110010010    //-11
270  11111111111111111111111111001101    //-51
271  00000000000000000000000000000000
```

در عکس اول محتوای حافظه برای دستورات و در عکس دوم محتوای حافظه برای آرایه ی اعداد (آرایه ی 20 تایی و با شروع از خانه ی 1000 حافظه) و عکس سوم محتوای رجیستر های استفاده شده در دستورات در پایان اجرا و عکس چهارم محتوای خانه های 2000 و 2004 حافظه و خروجی برنامه :






[0] 0
[1] 20
[2] 20
[3] 80
[4] 110
[5] 17
[6] -51
[7] 0
[8] 00000000000000000000000000000000
0
20
20
80
110
17
-51
0
00000000000000000000000000000000


[499] 00000000000000000000000000000000
[500] 110
[501] 17
[502] 00000000000000000000000000000000
00000000000000000000000000000000
110
17
00000000000000000000000000000000