تمرین کامپیوتری شماره 3 معماری کامپیوتر

مصطفی کرمانی نیا 810101575

امیر نداف فهمیده 810101540

1- مجموعه دستورات مدنظر :

- **R-Type: add, sub, and, or, slt, sltu**
- **I-Type: lw, addi, xori, ori, slti, sltiu, jalr**
- **S-Type: sw**
- **J-Type: jal**
- **B-Type: beq, bne, blt, bge**
- **U-Type: lui**

2- شکل کلی دستورات را طبق طراحی رسمی ریسک داریم:

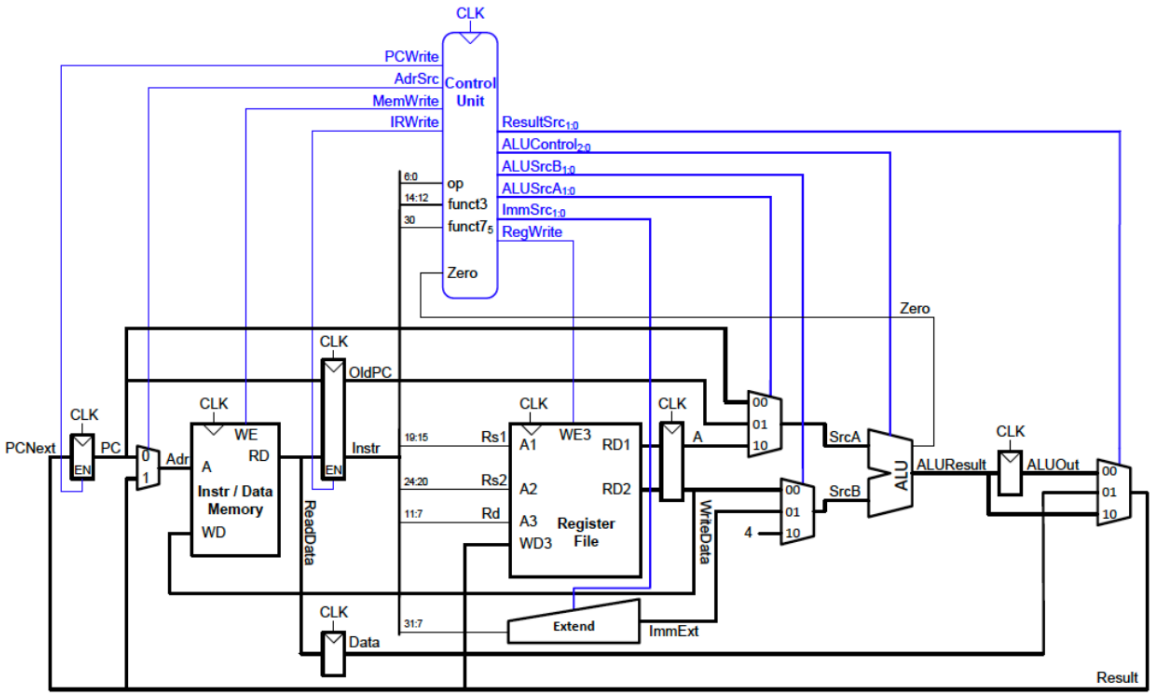
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

3- آپکد ها را طبق طراحی رسمی ریسک قرار می دهیم:(دستورات ساخته شده هایلایت شدند)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011 51	0x0 0	0x00 0	rd = rs1 + rs2	
sub	SUB	R	0110011 51	0x0 0	0x20 32	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011 51	0x6 6	0x00 0	rd = rs1 | rs2	
and	AND	R	0110011 51	0x7 7	0x00 0	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011 51	0x2 2	0x00 0	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011 51	0x3 3	0x00 0	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011 19	0x0 0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011 19	0x4 4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011 19	0x6 6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011 19	0x2 2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011 19	0x3 3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011 3	0x2 2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011 35	0x2 2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011 99	0x0 0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011 99	0x1 1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011 99	0x4 4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011 99	0x5 5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111 111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111 103	0x0 0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111 55			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

4- شکل مسیرداده ی مدنظر را رسم می کنیم:

حالت اولیه (که در درس به آن رسیده بودیم)



دستوراتی که باید اضافه کنیم به این مسیر داده:

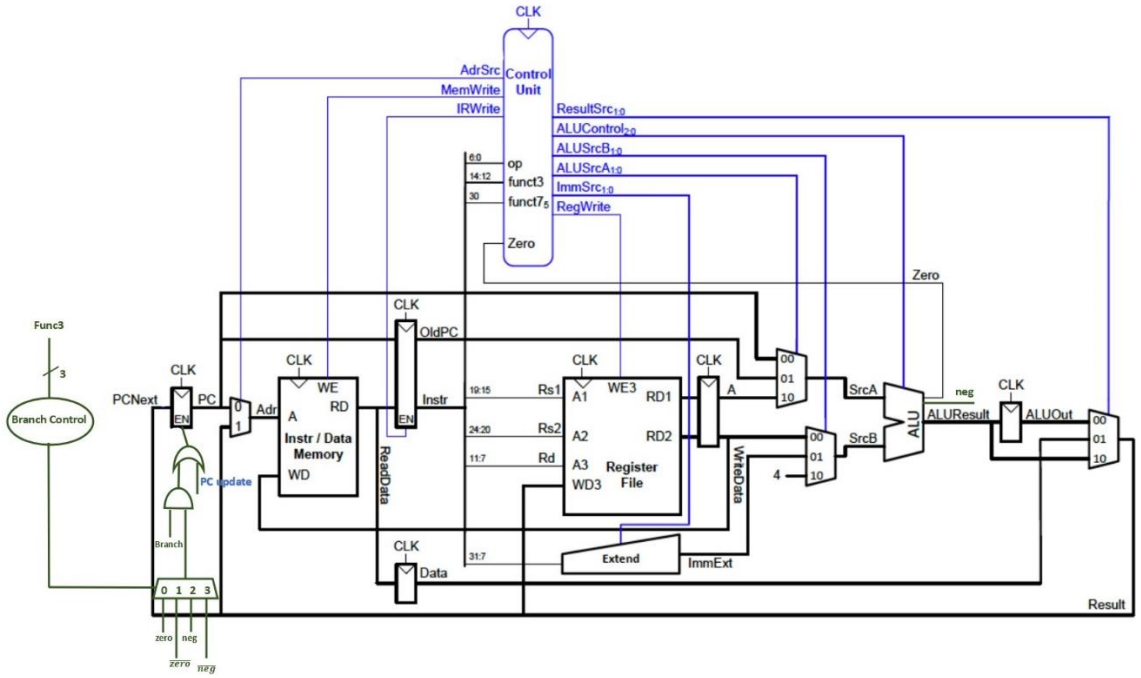
- **R-Type: sltu**
- **I-Type: xori, sltiu, jalr**
- **B-Type: bne, blt, bge**
- **U-Type: lui**

Sltu و xori : برای افزودن این دستورات نیاز به تغییر مسیر داده نیست، چون ALU همین الان هم طوری طراحی شده که دو دستور دیگر جا دارد. پس ALU را اینگونه نهایی می کنیم:

ALU CONTROL [2:0]	ALU RESULT
000	SRC A + SRC B
001	SRC A – SRC B
010	SRC A & SRC B
011	SRC A | SRC B
100	SRC A XOR SRC B
101	SRC A SLT SRC B
110	SRC A SLTU SRC B

البته این دستورات برای ALU است و از یک واحد کمکی به اسم ALU Control استفاده کردیم تا وقتی کنترلر مقدار ALUOp را مشخص کرد این واحد بر اساس ALUOp و func3 و func7 ، سیگنالALU CONTROL را (که در بالا جدولش هست) مشخص کند.

bne, blt, bge : برای این بخش یک سیگنال neg از ALU بیرون می آوریم که برای برنچ استفاده شود. لود رجیستر PC را به صورت زیر تغییر می دهیم. PC update برای زمانی است که می خواهیم به دستور بعدی برویم. برای زمانی که می خواهیم برنچ انجام دهیم ابتدا یک سیگنال Branch از کنترلر می آید. یک واحد Branch control داریم که تصمیم می گیرد کدام یک از سیگنال های خروجی ALU باید با Branch، and شود. به این صورت که اگر دستور beq بود سیگنال zero، اگر bne بود سیگنال zero~، اگر blt بود سیگنال neg و اگر bge بود سیگنال neg~ با Branch، and می شود و خروجی آن نشان می دهد که PC باید به مقدار جدید ست شود یا نه.



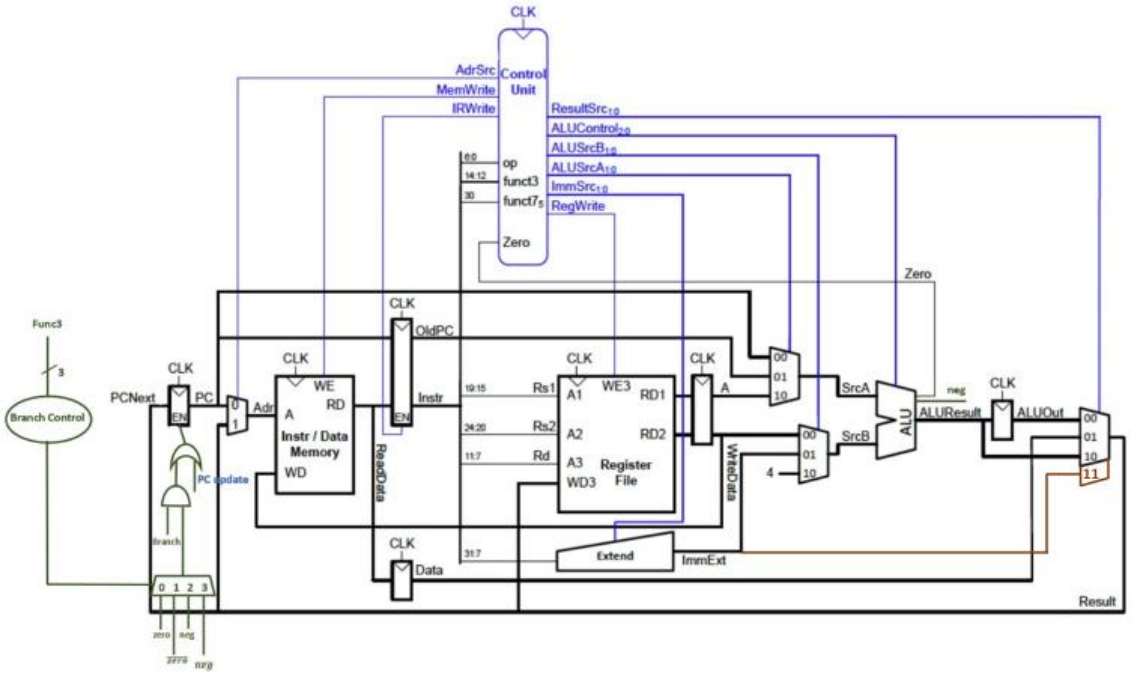
Jalr : برای این بخش هم نیازی به تغییر در مسیر داده نداریم و فقط کنترلر را عوض می کنیم.

طبق توضیحات اصلی ریسک داریم:

jal	Jump And Link	J	1101111	111		rd = PC+4; PC += imm
jalr	Jump And Link Reg	I	1100111	103 0x0	0	rd = PC+4; PC = rs1 + imm

Lui: طبق توضیحات ریسک داریم: برای این بخش دیتای **ImmExt** باید در رجیستر ذخیره شود به این منظور این دیتا را به مالتی پلکسر **Result** میدهیم و این مالتی پلکسر را چهار ورودی می کنیم.

lui	Load Upper Imm	U	0110111		rd = imm << 12



نهایتا برای **ImmSrc** هم داریم:

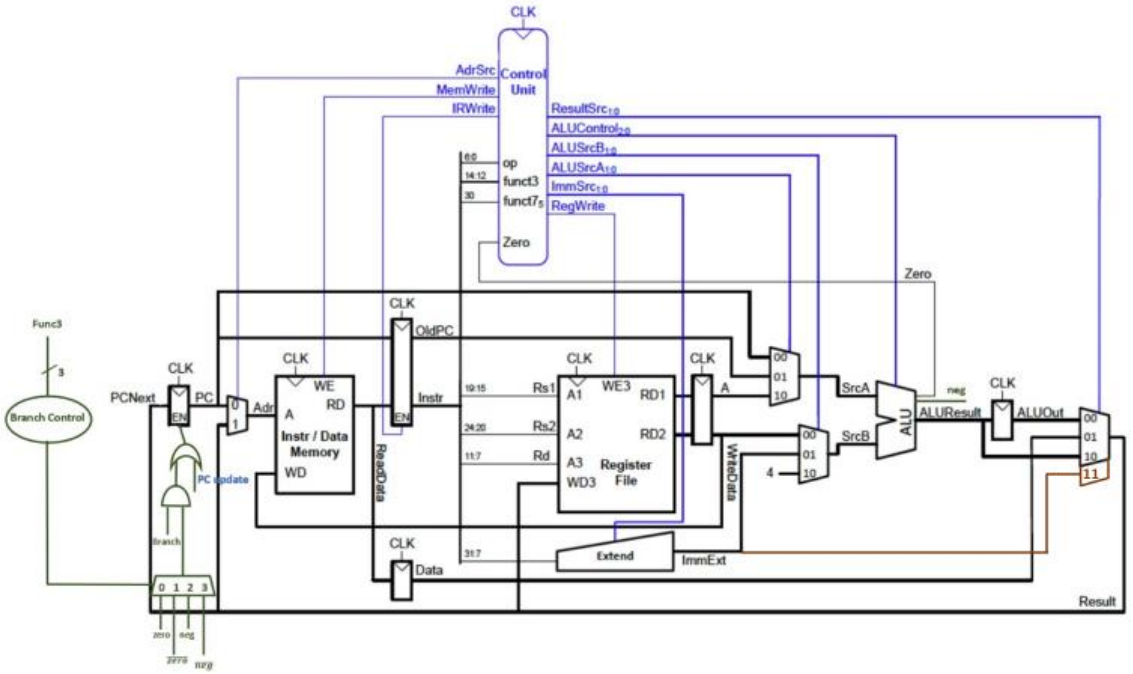
اولا دوباره به طرح رسمی ریسک رجوع میکنیم:

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

پس داریم:

ImmSrc	ImmExt	Input Type
000	**{ 20{ Ins[31] } , Ins[31:20] }**	I-Type
001	**{ 20{ Ins[31] } , Ins[31:25] , Ins[11:7] }**	S-Type
010	**{ 19{ Ins[31] } , Ins[31] , Ins[7] , Ins[30:25] , Ins[11:8] , 1'b0 }**	B-Type
011	**{ 11{ Ins[31] } , Ins[31] , Ins[19:12] , Ins[20] , Ins[30:21] , 1'b0 }**	J-Type
100	**{ Ins[31:12] , 12'b0 }**	U-Type

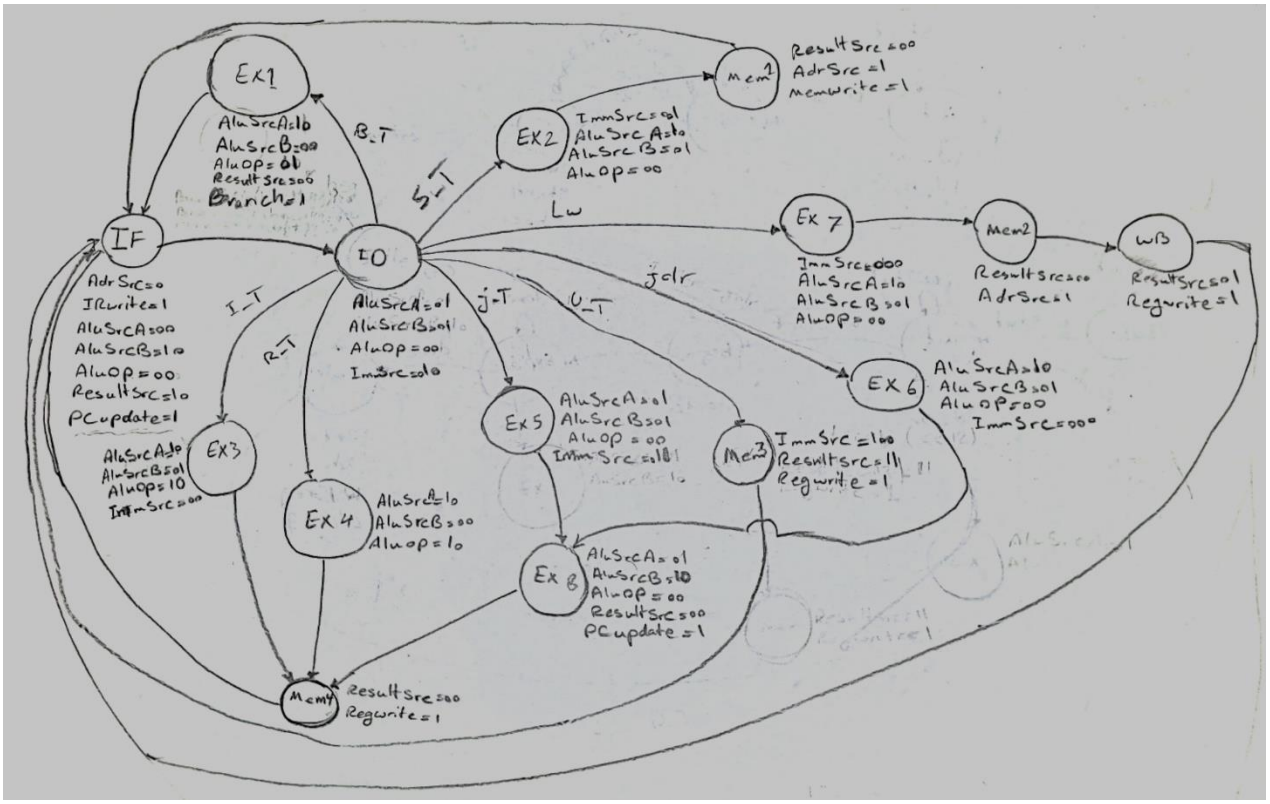
در نهایت داریم:



ImmSrc	ImmExt	Input Type
000	**{ 20{ Ins[31] } , Ins[31:20] }**	I-Type
001	**{ 20{ Ins[31] } , Ins[31:25] , Ins[11:7] }**	S-Type
010	**{ 19{ Ins[31] } , Ins[31] , Ins[7] , Ins[30:25] , Ins[11:8] , 1'b0 }**	B-Type
011	**{ 11{ Ins[31] } , Ins[31] , Ins[19:12] , Ins[20] , Ins[30:21] , 1'b0 }**	J-Type
100	**{ Ins[31:12] , 12'b0 }**	U-Type

ALU CONTROL [2:0]	ALU RESULT
000	SRC A + SRC B
001	SRC A – SRC B
010	SRC A & SRC B
011	SRC A | SRC B
100	SRC A XOR SRC B
101	SRC A SLT SRC B
110	SRC A SLTU SRC B

4- حالا وقت اعمال تغییرات و طراحی کنترلر است: طراحی نهایی به شکل زیر است.



این کنترلر در واقع تکامل یافته کنترلر جزوه است. تغییرات آن به شرح زیر می باشد.

اولین تغییر در دستورات B_T است که با توجه به تغییرات مسیر داده و اضافه کردن Branch Control تغییرات زیادی نیاز نداشت و با انتخاب کردن A , B و انجام عملیات تفریق سیگنال های لازم را برای اتفاق افتادن branch آماده می کنیم و بر اساس آن ها عدد جدید PC در رجیستر لود می شود یا نمی شود.

تغییر بعدی در اضافه کردن دستور jalr است که در واقع در استیت EX6 ما مقدار جدید pc را که برابر RS1+imm است را حساب می کنیم سپس به EX8 می رویم تا هم رجیستر PC لود شود هم PC بعدی را حساب کنیم و این مقدار را در رجیستر مقصد ذخیره کنیم.

برای دستور J_T هم همین کار را کردیم با این تفاوت که در EX5 مقدار PC + Imm را حساب می کنیم و در مرحله به EX8 می رویم و کار های بالا را انجام می دهیم. تفاوت این دو دستور آخر در اپرند های ALU و نحوه اکستند کردن هستند.

برای I_T ها هم**EX3** را اضافه کردیم که در آن دیتای **imm** اکستند شده را با مقدار رجیستر **A** روی **ALU** می بریم و **ALU control** با توجه به **func3** و **func7** عملیات **ALU** را مشخص می کند. سپس دوباره به **Mem4** می رویم تا نتیجه را در رجیستر مقصد ذخیره کنیم.

برای U_T یا دقیق تر بگم **lui** به **Mem3** می رویم که مقدارصحیح اکستند شده را روی ریزالت قرار می دهد و سیگنال نوشتن در رجیستر فایل را فعال می کند تا دیتا توی رجیستر مقصد لود شود.

5- برای تست برنامه باید بزرگترین عنصر یک آرایه 10 عنصری از اعداد صحیح بی علامت 32 بیتی را بیابیم. ابتدا الگوریتم را نوشته و اسمبلی و سپس زبان ماشین آنرا می نویسیم. فرض کردم که دیتا ما در خانه 1000 قرار دارد.

الگوریتم:

```
a[10] ;
result = 0 ;
for (i=0 ; i<10 ; i++)
    if (result <= a[i])
        result = a[i];
```

```
a ⟶ x1
result ⟶ x2

x3 = (x1 < 1040) ? 1 : 0
a[i] ⟶ x4
x5 = (x4 < x2) ? 1 : 0
```


اسمبلی:

```
addi x1, zero, 1000
add x2, zero, zero
Loop: slti x3, x1, 1040
beq x3, zero, EndLoop
lw x4, 0(x1)
sltu x5, x4, x2
```

```
bne x5, zero, EndIF
add x2, x4, zero
EndIF: addi x1, x1, 4
Jalr X10 ,Loop
EndLoop:
```

زبان ماشین به فرمت هگز(به فرمتی که با کدمان خوانده می شود):

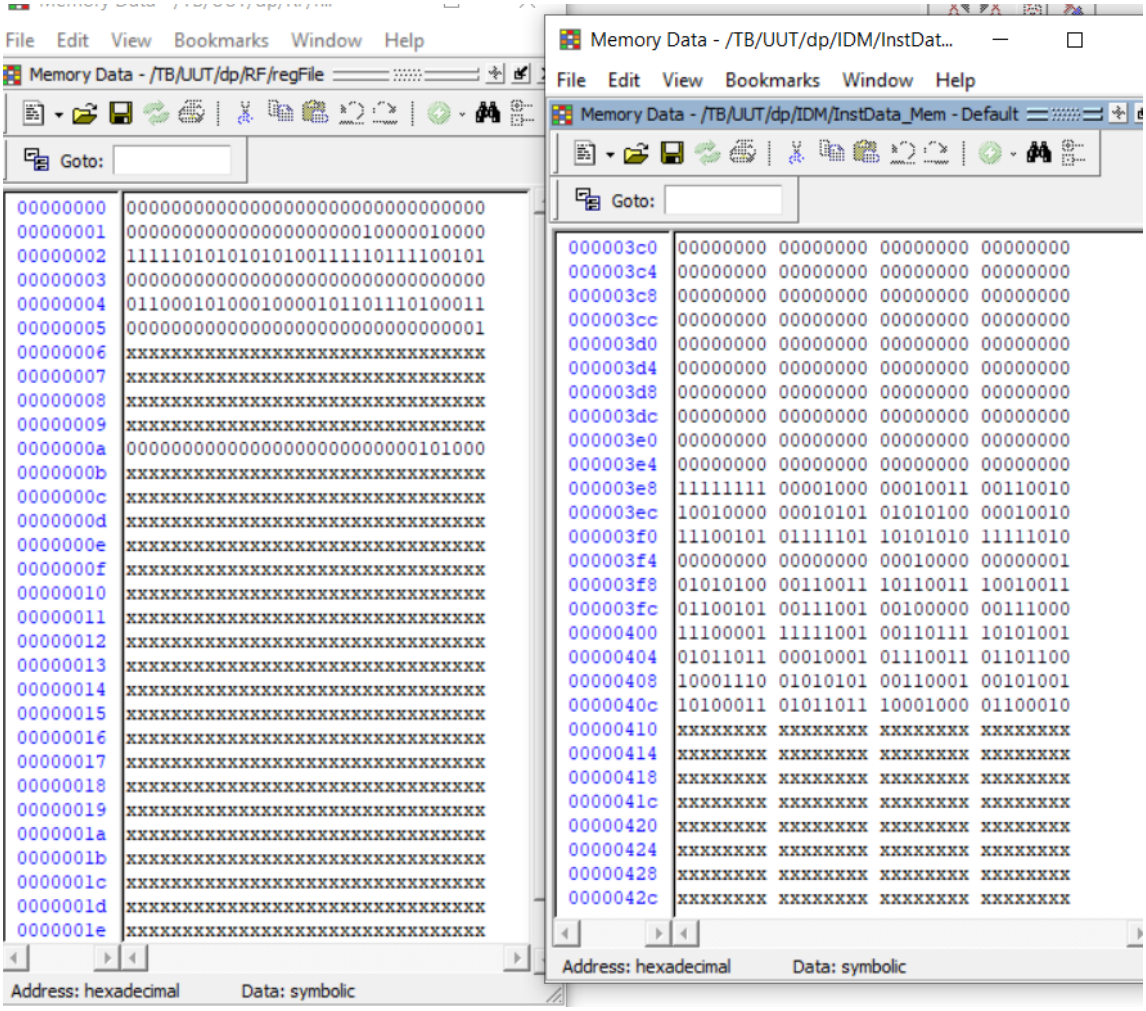
```
93
00
80
e3
33
01
00
00
93
a1
00
41
63
e8
01
00
03
a2
00
00
b3
32
22
00
63
94
02
00
33
01
02
00
93
80
40
00
f6
f5
```

f5
fe

دیتا هم از خانه هزار شروع به لود کردن کردیم:

ff
08
13
32
90
15
54
12
e5
d7
aa
fa
00
00
10
01
54
33
b3
93
65
39
20
38
e1
f9
37
a9
b5
11
73
c6
e8
55
31
29
a3
b5
88
62

6- نهایتا بعد از شبیه سازی برنامه ی خود، داریم:



توضیح: در حافظه سمت راست ما ده تا دیتا 32 بیتی از خانه 3e8 هگز که خانه هزار می شود تا خانه 410 هگز که خانه 1040 می شود پر کردیم. همان طور که مشاهده می شود دیتای رجیستر دوم قرار بود ماکسیمم این اعداد بدون علامت باشد که برابر با دیتای خانه 3f0 است. همین طور در x4 نیز دیتای خوانده شده از حافظه است که بعد از اتمام برنامه برابر آخرین عنصر است. در x10 هم مقدار pc بعد از jal است. X3 و x5 هم متغیر هایی برای دستورات slti و sltu هستند.