تمرین کامپیوتری شماره 2 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:

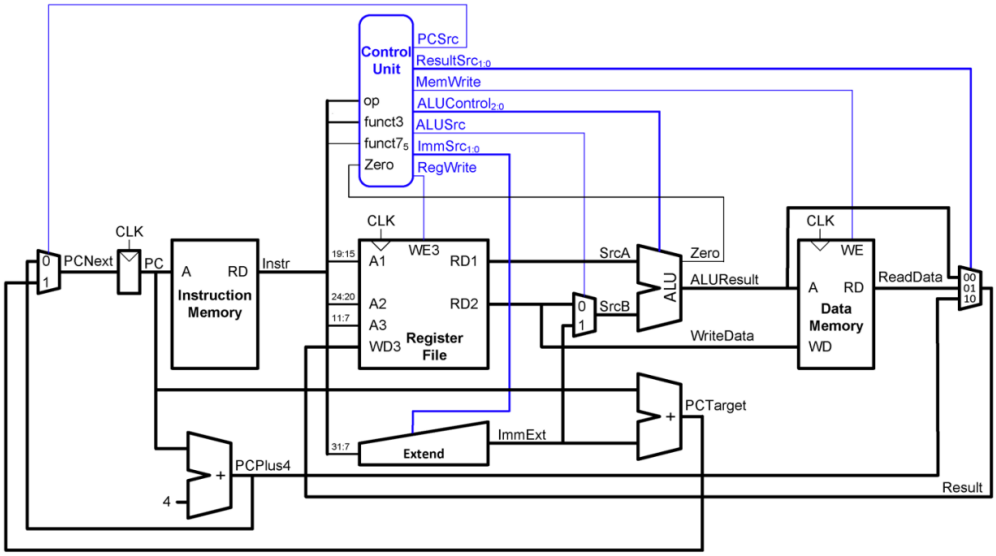
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

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)



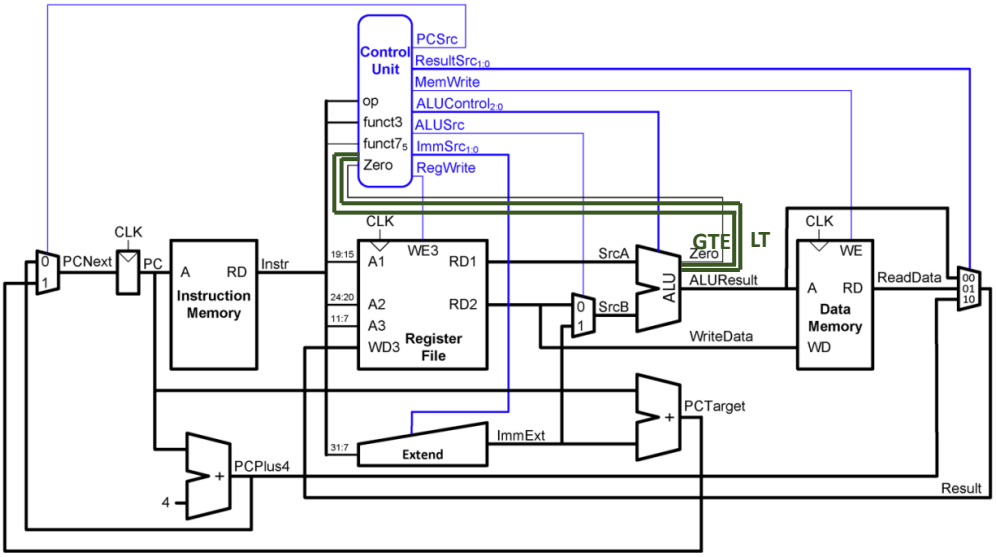
دستوراتی که باید اضافه کنیم به این مسیر داده:

- **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

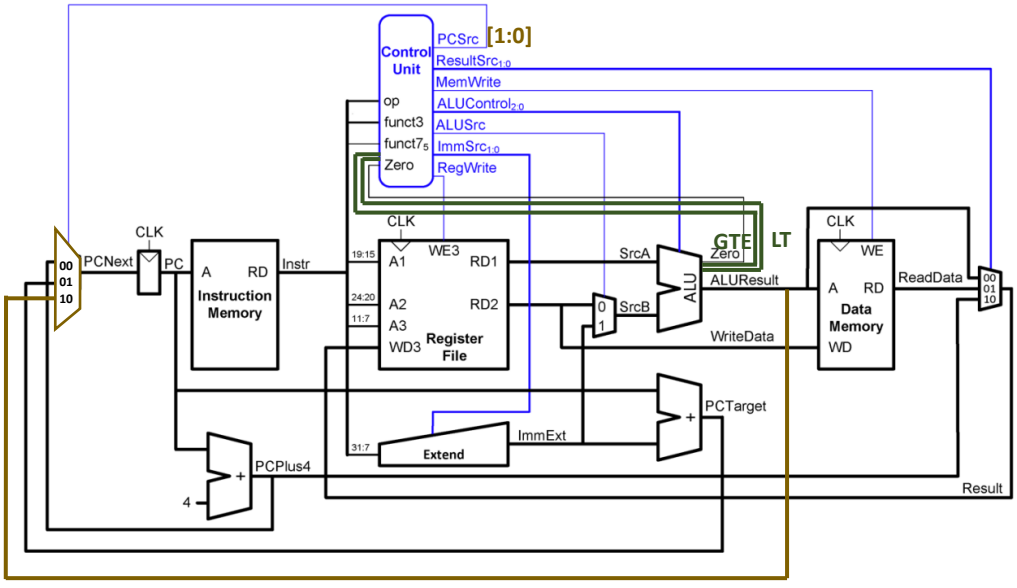
bne, blt, bge : تغییرات لازم برای این دستورا قبلا با افزودن **beq** در مسیر داده انجام شده است پس حالا تنها لازم است که دو سیگنال دیگر همانند **zero** از **ALU** بگیریم تا بتوانیم توسط آنها عملیات های **blt** و **bge** را انجام دهیم(عملیات **bne** توسط همان سیگنال **zero** هندل میشود) پس تغییری که در **ALU** میدهیم اینگونه میشود:



Jalr : 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

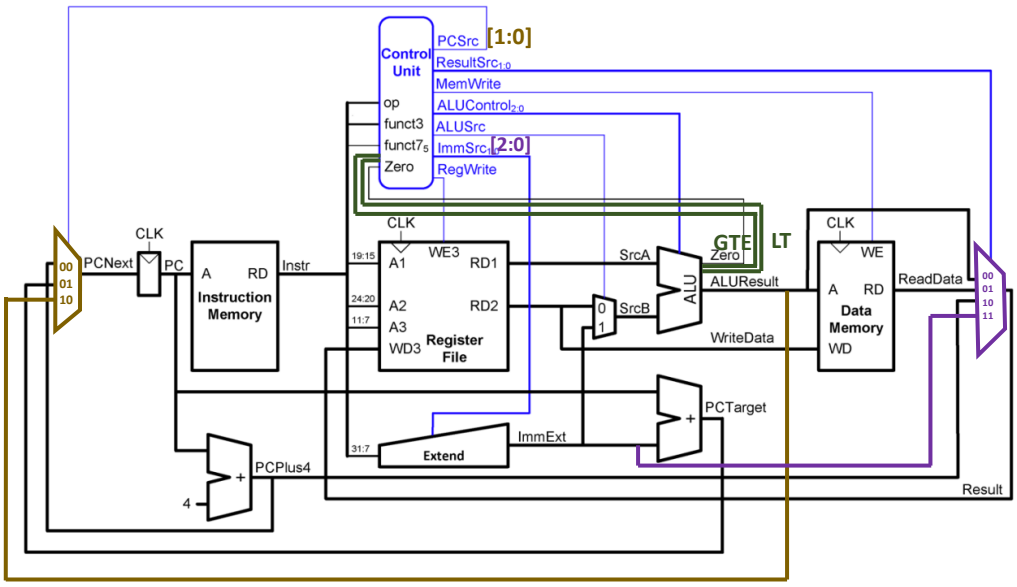
حالا ما برای اعمال **jal** مسیری داریم که **pc** را با **imm** جمع کرده و در **pc** بگذارد اما مسیری نداریم که **jalr** را انجام دهد، برای اینکار اولا مسیری داریم که **rs1** را با **imm** جمع کند، حالا کافیست مسیری اضافه کنیم که نتیجه را در **pc** بگذارد.برای اینکار **pcsrc** دو بیتی میشود یعنی :



Lui: طبق توضیحات ریسک داریم:

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

اولا در این بخش لازم است مسیری از **Extend** به داخل **WD3** داشته باشیم که اینکار را با افزودن یک بخش به **ALU** سازنده ی **result**میکنیم. حالا باید به بخش **extend** یک خانه ی جدید اضافه کنیم که چون **4** خانه ی آن پر است باید یک بیت به **ImmSrc** هم اضافه شود:



نهایتا برای **ImmSrc** هم داریم:

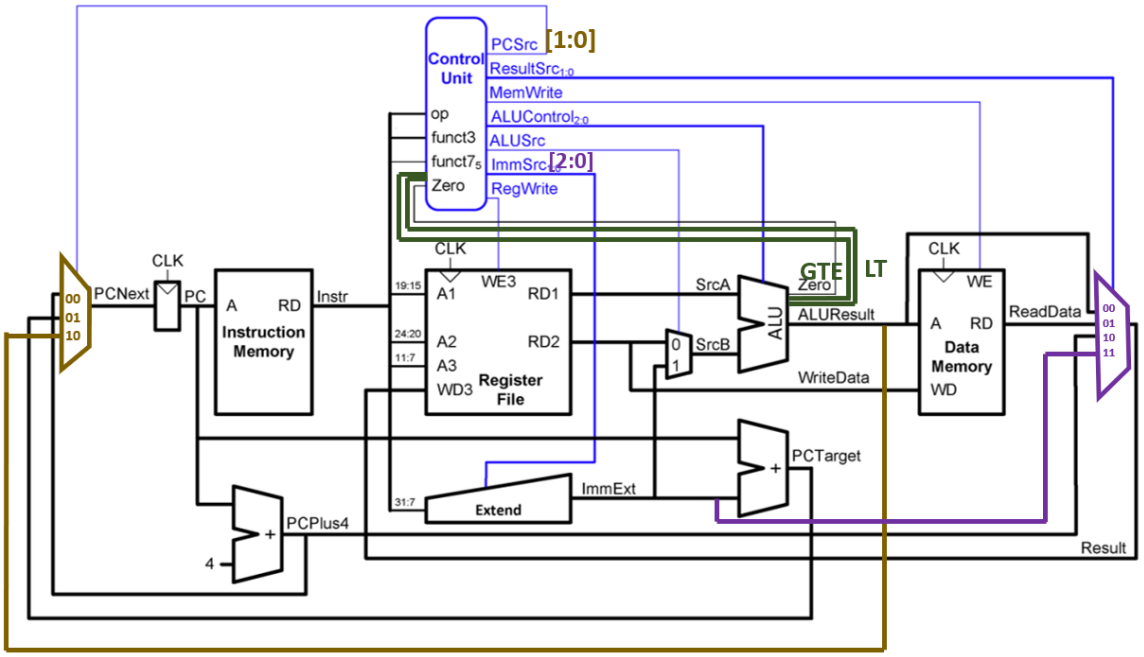
اولا دوباره به طرح رسمی ریسک رجوع میکنیم:

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:

اولا طبق طراحی اصلی ریسک، ستون های op و Func3 و Func7 و INSt را پر می کنیم:

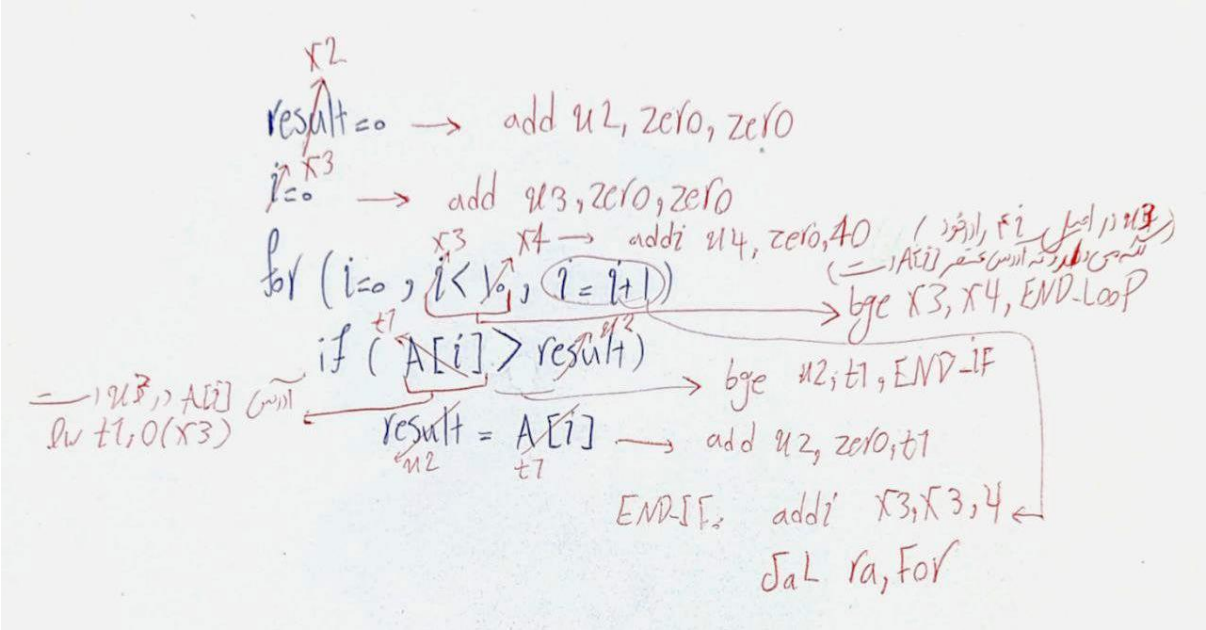
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011 51	0x0 0	0x00 0	rd = rs1 + rs2	
sub	SUB	R	0110011 51	0x0 0	0x20 32	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011 51	0x6 6	0x00 0	rd = rs1 | rs2	
and	AND	R	0110011 51	0x7 7	0x00 0	rd = rs1 & rs2	
sll	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	

سپس به کمک نگاه کردن به طراحی مسیر داده، بقیه جدول را پر میکنیم:

op	Func3 [2:0]	Func7 [6:0]	Inst	Zero	Lt	GtE	PCSrc [1:0]	ResultSrc [1:0]	MemWrite	ALUControl [2:0]	ALUSrc	IMMSrc [2:0]	RegWrite
51	0	0	add	-	-	-	00	00	0	000	0	--	1
51	0	32	sub	-	-	-	00	00	0	001	0	--	1
51	6	0	or	-	-	-	00	00	0	011	0	--	1
51	7	0	and	-	-	-	00	00	0	010	0	--	1
51	2	0	slt	-	-	-	00	00	0	101	0	--	1
51	3	0	sltu	-	-	-	00	00	0	110	0	--	1
19	0	-	addi	-	-	-	00	00	0	000	1	000	1
19	4	-	xori	-	-	-	00	00	0	100	1	000	1
19	6	-	ori	-	-	-	00	00	0	011	1	000	1
19	2	-	slti	-	-	-	00	00	0	101	1	000	1
19	3	-	sltiu	-	-	-	00	00	0	110	1	000	1
3	2	-	lw	-	-	-	00	01	0	---	1	000	1
35	2	-	sw	-	-	-	00	--	1	000	1	001	0
99	0	-	Beq		-	-	Zero?01:00	--	0	001	0	010	0
99	1	-	Bne		-	-	Zero?00:01	--	0	001	0	010	0
99	4	-	Blt	-		-	Lt?01:00	--	0	---	0	010	0
99	5	-	Bge	-	-		Gte?01:00	--	0	---	0	010	0
111	-	-	Jal	-	-	-	01	10	0	---	-	011	1
103	0	-	Jalr	-	-	-	10	10	0	000	1	000	1
55	-	-	lui	-	-	-	00	11	0	---	-	100	1

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.

الگوریتم:



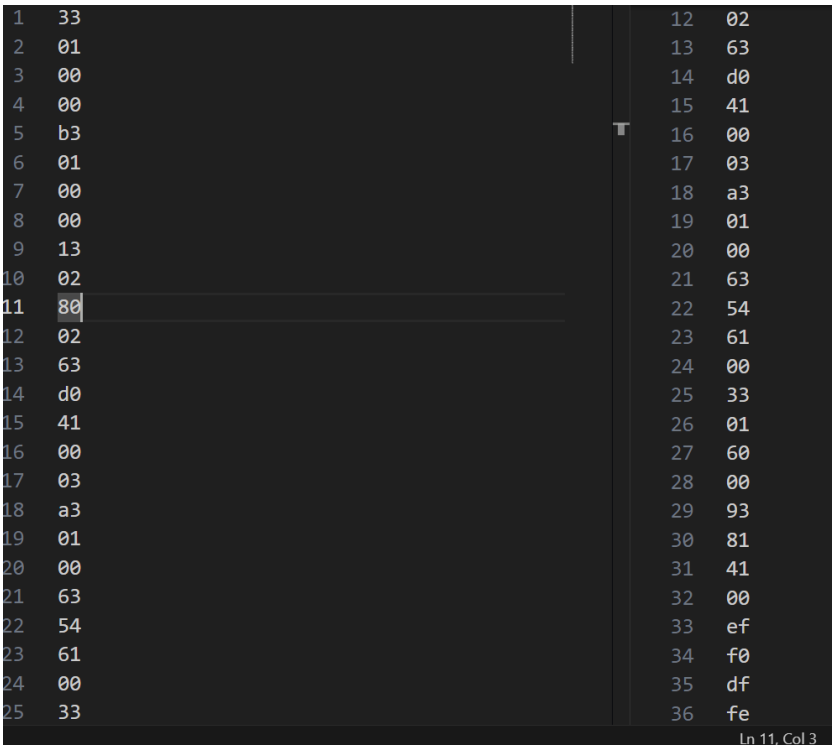
اسمبلی:

```
1   add X2, Zero, Zero
2   add X3, Zero, Zero
3   addi X4, Zero, 40
4   For: bge X3, X4, End_Loop
5   lw t1, 0(X3)
6   bge X2, t1, End_IF
7   add X2, Zero, t1
8   End_IF: addi X3, X3, 4
9   Jal ra, For
10  End_Loop:
```

زبان ماشین:

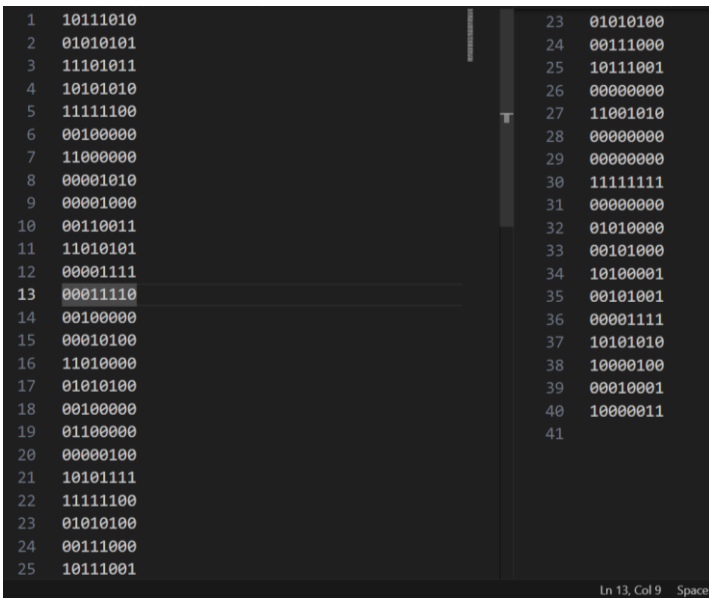
```
1   0x00000133 // add X2, Zero, Zero
2   0x000001b3 // add X3, Zero, Zero
3   0x02800213 // addi X4, Zero, 40
4   0x0041d063 // For: bge X3, X4, End_Loop / End_Loop = PC + 24
5   0x0001a303 // lw t1, 0(X3)
6   0x00615463 // bge X2, t1, End_IF / End_IF = PC +8
7   0x00600133 // add X2, Zero, t1
8   0x00418193 // End_IF: addi X3, X3, 4
9   0xfedff0ef // Jal ra, For / For = PC - 20
10  // End_Loop:
```

هگز به فرمتی که با کدمان خوانده می شود:

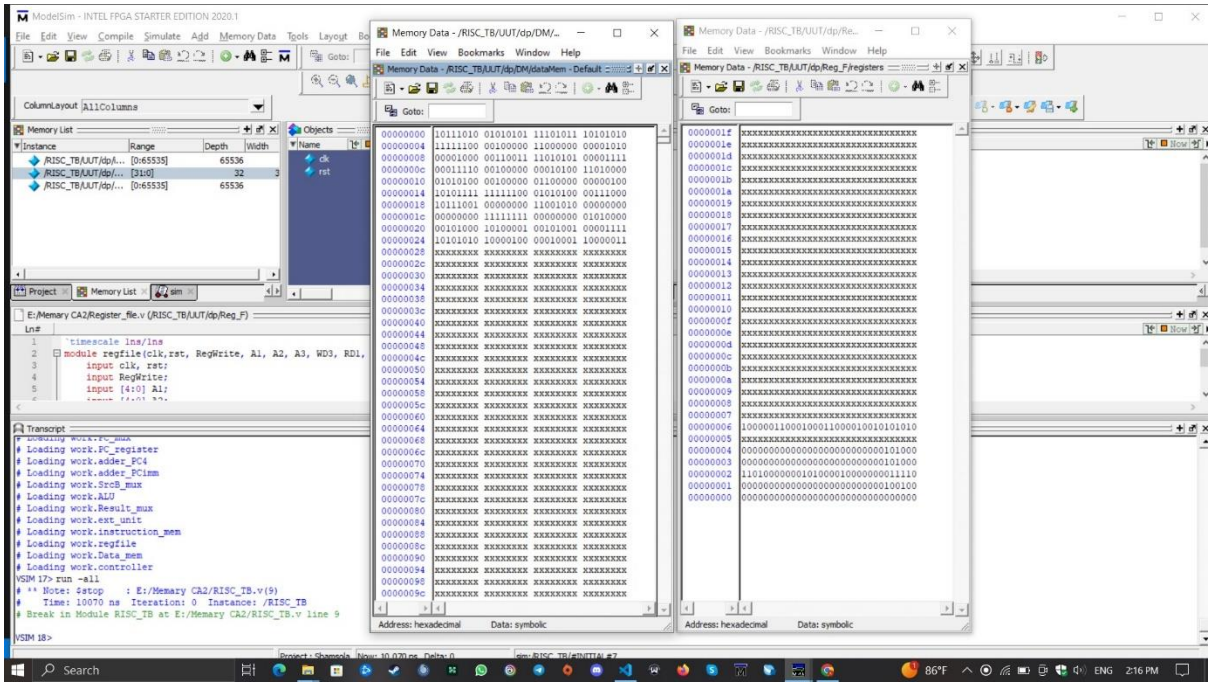


(فایل نهایی Inst.mem است)

و DataMem را هم بصورت زیر پر کردیم:



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



همانطور که مشخص است، DataMem خوانده شده و سپس InstMem هم خوانده شده و دستورات آن اعمال شده است و نهایتا بزرگترین عدد موجود در آرایه 10 عنصری ما پیدا شده و در رجیستری که در فایل اسمبلی مشخص کرده بودیم یعنی (X2) ریخته شده است:(سمت چپ DataMem است و سمت راست رجیستر های ما را مشخص کرده است)