

۱. توضیح ماژول ها

```
module inst_mem(inst,addr,instruction);
```

این ماژول در بر دارنده حافظه دستورات پردازنده می باشد.

درون ماژول ما به وسیله ی دستور زیر ، فایل instruction.txt را که در پوشه پروژه قرار دارد را می خوانیم :

```
initial begin
```

```
    $readme ("instruction.txt", instruction);
```

```
end
```

ورودی addr ، آدرس ۳۲ بیتی است که وارد حافظه می شود و سپس محتوای آن خط آدرس توسط خروجی ۳۲ بیتی inst از ماژول خارج خواهد شد.

Instruction کل حافظه دستورات ما هست، ما برای تست کردن صحت اجرای برنامه در تست بنچ کل حافظه را نیز در خروجی ماژول قرار دادیم!

به دلیل **محدودیت حافظه** مدلسیم ، ما حافظه دستورات را **۱۶ کیلوبایت معادل ۶۵۵۳۶ کلمه** تعریف کرده ایم یعنی :

```
output logic [31:0]instruction[0:65536];
```

```
module ALU(z,c,a,b,alusel);
```

این ماژول محاسباتی ما هست، a و b دو ورودی ۳۲ بیت آن هستند و c خروجی ۳۲ بیت آن.

اگر c برابر صفر بشود ، پرچم z به عنوان خروجی یک خواهد شد.

نوع عملیات محاسباتی بر اساس ورودی ۳ بیت alusel تعیین خواهد شد :

000 : جمع - 001 : تفاضل - 010 : اندمنطقی - 011 : اورمنطقی - 100 : slt (اگر a کوچکتر از b باشد c یک میشود)

```
module PC(parout,parin,clk,rst);
```

این ماژول رجیستر ۳۲ بیتی شمارنده برنامه می باشد که سینکرون می باشد.

در صورت فعال بودن rst خروجی parout صفرخواهد شد.

در هر لبه مثبت کلاک ، ۳۲ بیت parin در ۳۲ بیت parout قرار داده می شود.

```
module MUX(c,a,b,sel);
```

این ماژول مالتیپلکسر ۳۲ بیت دو به یک است. دو ورودی ۳۲ بیت a و b را می گیرد اگر مقدار sel صفر باشد ، a را روی خروجی c قرار خواهد داد و اگر sel یک باشد ، b را روی خروجی c قرار می دهد.

علاوه بر این مالتیپلکسر ۳۲ بیتی ، از مالتیپلکسرهای ۵ بیت و ۱ بیت هم در برنامه استفاده شده است.

```
module data_mem(clk,readdata,addr,writedata,memread,memwrite,memory);
```

این ماژول حافظه داده را نگه داری می کند. به دلیل **محدودیت حافظه** مدلسیم ، ما حافظه دستورات را **۱۶ کیلوبایت معادل ۶۵۵۳۶ کلمه** تعریف کرده ایم یعنی :

```
output logic [31:0]memory[0:65536];
```

ما داده ارا از فایل memory.txt که در محل پوشه پروژه قرار دارد فراخوانی کرده و در حافظه وارد می کنیم :

```
initial begin
```

```
    $readmemb ("memory.txt",memory);
```

```
End
```

ورودی ۳۲ بیت addr ، آدرس محل حافظه را مشخص می کند ، سپس در صورت فعال بودن سیگنال کنترلی memread ، محتوای آن آدرس در خروجی ۳۲ بیت readdata نمایش داده خواهد شد.

برای نوشتن در حافظه به صورت سنکرون عمل میکنیم ، به این صورت که اگر سیگنال ورودی memwrite فعال باشد ، با اولین لبه مثبت کلاک ، مقدار ورودی ۳۲ بیت writedata در خانه حافظه به آدرس addr نوشته خواهد شد.

ما برای نشان دادن مقادیر حافظه در تست بنچ ، کل حافظه memory را از ماژول خروجی گرفتیم.

```
module reg_file(register,readdata1,readdata2,readreg1,readreg2,writereg,writedata,regwrite,clk);
```

این ماژول دربردارنده ی رجیستر فایل می باشد. متغیر register یک آرایه ۳۲ در ۳۲ بیت است ، که کلمه نخست آن یا همان R0 همواره صفر می باشد. یعنی :

```

initial begin
    register[0]=32'b0;
end

```

می توان همزمان دو رجیستر را خواند ، برای خواندن دو ورودی ۵ بیت readreg1 و readreg2 آدرس محل رجیسترها را مشخص کنند، سپس مقادیر آنها در دو خروجی ۳۲ بیت readdata1 و readdata2 نمایش داده خواهد شد.

عمل نوشتن در رجیسترفایل به صورت سنکرون می باشد ، به این صورت که در صورت فعال بودن سیگنال کنترلی regwrite ، با اولین لبه مثبت کلاک ، مقدار ورودی ۳۲ بیت writedata در آدرس ورودی writereg نوشته خواهد شد.

ما برای تست کردن و دیدن محتویات رجیستر فایل ، حافظه register که دربردارنده تمام رجیسترفایل می باشد را از ماژول خروجی گرفته ایم.

```

module sum(c,a,b);

```

این ماژول یک جمع کننده combitonal می باشد که دو ورودی ۳۲ بیت a و b را گرفته و حاصل جمع را در خروجی ۳۲ بیت c می گذارد.

```

module alu_control(alusel,func,aluop);

```

این ماژول کنترل کننده عملیات واحد محاسباتی پردازنده می باشد.

خروجی ۳ بیت alusel به ALU وارد خواهد شد و نوع دستور محاسباتی را برای ALU مشخص خواهد کرد بدین صورت که اگر ۰۰۰ : جمع - ۰۰۱ : تفاضل - ۰۱۰ : اند - ۰۱۱ : اور - ۱۰۰ : slt خواهد بود.

ورودی ۲ بیت aluop یک سیگنال کنترلی است که از سمت کنترلر اصلی می آید اگر aluop :

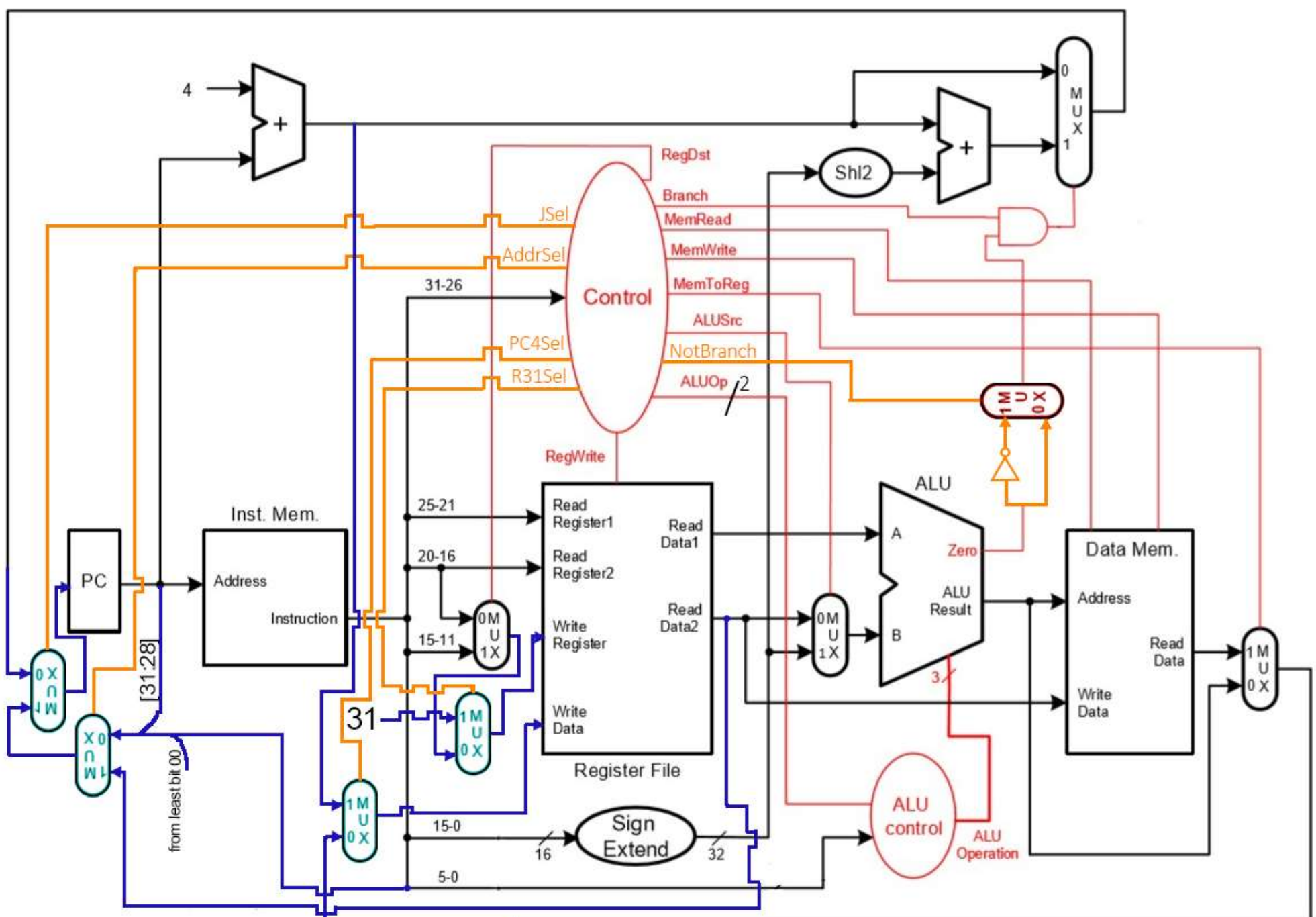
۰۰ : alusel را ۰۰۰ خواهد کرد. این حالت برای دستور addi استفاده می شود.(حالت default)

۰۱: نوع دستور R-type است پس به مقدار ۶ بیت ورودی func که درواقع ۶ بیت کم ارزش insta می باشد نگاه می کند. باتوجه به آن جمع یا تفاضل یا ... را مشخص می کند.

۱۰: alusel را ۰۰۱ خواهد کرد. این حالت برای دستورات beq و bne استفاده می شود.

۱۱: alusel را ۰۱۰ خواهد کرد. این حالت برای دستور andi استفاده می شود.

۲. مسیر داده



دیتا پث ما همان دیتا پث اصلی MIPS است که دکتر تدریس کرده اند ، سیم های آبی و نارنجی (کنترلی) و mux ها جدید نشان دهنده قسمت های جدیدی هستند که به مسیر داده اضافه شده اند.

توضیح مسیر داده دستورات جدید اضافه شده :

bne : برای این دستور که عکس دستور beq عمل می کند کافیسیت از سیگنال Zero که از ALU خارج می شود یک قرینه بگیریم تا درست اجرا شود ، برای تشخیص اینکه چه زمانی باید Zero یا قرینه اش را بفرستیم ، یک MUX سر راهشان قرار دادیم. این MUX توسط سیگنال کنترلی NotBranch کنترل خواهد شد، بدین صورت که اگر دستور ma bne باشد ، NotBranch برابر یک خواهد شد و قرینه Zero را عبور می دهد.

addi ,andi : در این دستورات مسیر داده برای عبور داده ها به alu و انجام عملیات و ریختن در رجیستر مقصد کاملاً آماده است ، اما نمی توان نوع عملیات محاسباتی را برای ALU مشخص کرد ، به همین دلیل ما سیگنال کنترلی یک بیت ALUOp را که از قبل داشتیم تبدیل به دو بیت کردیم ، تا بتوانیم با این دوبیت دستور جمع برای addi و اند برای andi را مستقیماً به ALUCONTROL بدیم تا ALUCONTROL به ALU دستورات را برساند.

jr ,j : برای اجرای این دستورات ، ورودی PC باید تغییر کند ، به همین دلیل لازم است یک MUX سر راه ورودی PC باشد ، که توسط سیگنال کنترلی JSel کنترل می شود، اگر JSel یک باشد ، آدرس PC جدید وارد می شود. برای دستور j بایستی ۲۶ بیت کم ارزش خروجی INST MEM که حاوی آدرس جدید است جدا شود ، سپس ۲ بیت صفر به سمت راست آن اضافه می شود ، در ادامه ۴ بیت پرارزش خروجی PC که نشان دهنده شماره page حافظه دستورات است را از سمت چپ اضافه می کنیم. اکنون یک آدرس ۳۲ بیتی درست داریم. در کنار دستور jr دستور j نیز یک آدرس ۳۲ بیتی را از readdata1 برایمان می آورد ، بنابراین برای انتخاب اینکه آیا آدرس دستور j یا آدرس دستور jr وارد pc شود ، یک MUX دیگر نیز قبل از PC قرار می دهیم که توسط سیگنال کنترلی AddrSel انتخاب می شود ، اگر یک باشد آدرس jr را رد می کند.

jal : این دستور همانند دستور j است با این تفاوت که بایستی آدرس آخرین دستور قبل از پرش را در رجیستر ۳۱ ذخیره کند. برای اینکار ابتدا باید عدد ۳۱ را به WriteReg بدیم ، بنابراین قبل از ورودی WriteReg یک MUX قرار می دهیم که توسط سیگنال کنترلی R31Sel کنترل می شود ، اگر یک باشد عدد ۳۱ وارد آن می شود. در ادامه کار بایستی مقدار PC+4 را در رجیستر ۳۱ ذخیره کنیم پس قبل از ورودی WriteData یک MUX قرار می دهیم که اگر سیگنال کنترلی PC4Sel یک باشد ،آنگاه مقدار PC+4 را وارد WriteData خواهد کرد.

۳. کنترلر و سیگنال های کنترلی :

	RegDst	RegWrite	ALUSrc	MemRead	MemWrite	MemToReg	Branch	ALUOp	NotBranch	JSel	AddrSel	R31Sel	PC4Sel
RT	1	1	0	0	0	0	0	01	X	0	X	0	0
lw	0	1	1	1	0	1	0	00	X	0	X	0	0
sw	X	0	1	0	1	X	0	00	X	0	X	0	0
beq	X	0	0	0	0	X	1	10	0	0	X	0	0
bne	X	0	0	0	0	X	1	10	1	0	X	0	0
addi	0	1	1	0	0	0	0	00	X	0	X	0	0
andi	0	1	1	0	0	0	0	11	X	0	X	0	0
J	X	X	X	X	X	X	X	X	X	1	0	0	0
Jr	X	0	X	X	0	X	0	00	X	1	1	0	0
Jal	X	1	X	X	X	X	X	X	X	1	0	1	1

OPC	ALUOp	func	ALU Action	ALUOperation
RT	01	100000	add	000
	01	100010	sub	001
	01	100100	and	010
	01	100101	or	011
	01	101010	slt	100
lw	00	XXXXXX	add	000
sw	00	XXXXXX	add	000
beq	10	XXXXXX	sub	001
bne	10	XXXXXX	sub	001
addi	00	XXXXXX	add	000
andi	11	XXXXXX	and	010
Jr	00	XXXXXX	add	000

۴. طراحی برنامه های تست و سیمولیشن آنها :

در ابتدا یک فایل memory.txt در پوشه پروژه ایجاد کردیم. سپس ۵۰۱ خط عدد ۳۲ بیتی (معادل ۲۰۰۴ آدرس) در آن نوشتیم ، مقادیر آدرس های ۱۰۰۰ تا ۱۰۷۶ این فایل حافظه به ترتیب دسیمال برابر است با :

مقدار	آدرس	خانه
1	1000	250
2	1004	251
3	1008	252
4	1012	253
5	1016	254
6	1020	255
7	1024	256
8	1028	257
9	1032	258
10	1036	259
32	1040	260
16	1044	261
8	1048	262
4	1052	263
24	1056	264
8	1060	265
256	1064	266
64	1068	267
32	1072	268
1	1076	269

توجه داشته باشید داده های هردو برنامه همین مقادیر هستند و همچنین این دو برنامه در یک سیمولیشن اجرا شده یعنی ابتدا دستورات برنامه اول ، سپس دستورات برنامه دوم اجرا خواهد شد.

برنامه اول (مجموعه اعضای یک آرایه ۱۰ بیتی از ۱۰۰۰ تا ۱۰۳۶) :

ما مقادیر آدرس های ۱۰۰۰ تا ۱۰۳۶ (معادل خانه ۲۵۰ تا ۲۵۹) را میخوانیم و حاصل جمع را در آدرس ۲۰۰۰ (معادل خانه ۵۰۰) خواهیم ریخت.

```
add R1, R0, R0
addi R2, R0, 40
add R4, R0, R0
LOOP: slt R3, R1, R2
beq R3, R0, END-LOOP
lw R5, 1000(R1)
add R4, R4, R5
addi R1, R1, 4
J LOOP
END-LOOP: sw R4, 2000(R0)
```

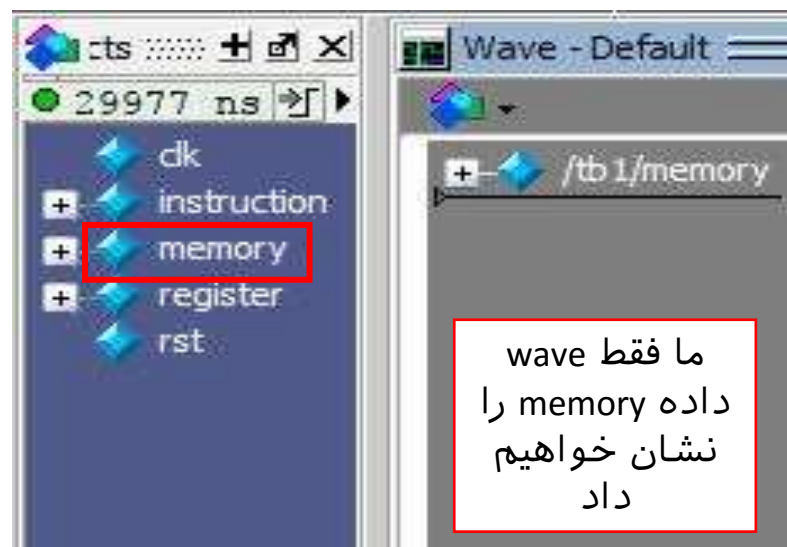
در فایل Testbenches.sv یک ماژول تست بنچ به نام tb1() وجود دارد که ماژول زیر را سیمولیت می کند :

```
datapath uut0(register,clk,rst,instruction,memory);
```

register خروجی تمام حافظه regfile می باشد ،

instruction خروجی تمام حافظه Inst-mem می باشد،

memory خروجی تمام حافظه Data-mem می باشد ما در سیمولیشن فقط به مقادیر Data-Mem نگاه می کنیم.





برنامه دوم (پیدا کردن بزرگترین عنصر یک آرایه ۲۰ بیتی از ۱۰۰۰ تا ۱۰۷۶) :

ما مقادیر آدرس های ۱۰۰۰ تا ۱۰۷۶ (معادل خانه ۲۵۰ تا ۲۶۹) را میخوانیم و مقدار بزرگترین عنصر و اندیس آنرا به ترتیب در آدرسهای ۲۰۰۰ و ۲۰۰۴ (معادل خانه ۵۰۰ و ۵۰۱) حافظه میریزیم/

```
add R1, R0, R0
addi R2, R0, 80
add R8, R0, R0
lw R4, 1000(R1)
LOOP: slt R3, R1, R2
beq R3, R0, END-LOOP
lw R5, 1000(R1)
slt R6, R4, R5
beq R6, R0, CONTINUE
lw R4, 1000(R1)
add R7, R8, R0
CONTINUE: addi R8, R8, 1
addi R1, R1, 4
J LOOP
END-LOOP: sw R4, 2000(R0)
sw R7, 2004(R0)
```