پروژه درس معماری کامپیوتر

Name	Student ID
Arman Zarei	97110444
Mobina PourNemat	97105833
Mahsa Amani	97105769

ماژول ALU

- **ورودی ها**

همان طور که در داک نیز گفته شد ۵ تا ورودی به شرح زیر داریم که به این ماژول داده می شود

1. in1 که ۳۲ بیت می باشد و ورودی اول ماژول ما می باشد
2. in2 که ۳۲ بیت می باشد و ورودی دوم ماژول ما می باشد
3. Shiftamt که ۵ بیت بوده و به منظور مقدار ورودی اول در ۲ دستور شیفت استفاده می شود
4. Opcode که ۳ بیت و بوده و همان Opcode اصلی منهای بیت سمت چپ (MSB) می باشد و از آن برای تشخیص نوع عملیات استفاده می شود و مطابق با این ورودی ، خروجی مورد نظر تعیین می شود
5. cin که carry in در عملیات جمع می باشد که در آن دستور استفاده می شود

- **خروجی ها**

همان طور که در داک گفته شد خروجی های ماژول ALU به شرح زیر می باشند

1. Out که همان خروجی ALU ما می باشد
2. Eq که در صورت تساوی ۲ ورودی ۱ می شود (in1 == in2)
3. Zero که در صورت ۰ شدن خروجی این سیگنال ۱ می شود
4. Sign که مثبت یا منفی بودن خروجی را مشخص میکند . در صورت منفی بودن ۱ و در غیر اینصورت ۰ می باشد
5. Overflow که برای عملیات جمع استفاده می شود و در صورت رخ دادن overflow در جمع این سیگنال ۱ میشود

- **عملیات ها و پیاده سازی**

در ALU ، ۷ تا عملیات باید هندل شود که هر کدام و نحوه پیاده سازی در زیر ذکر شده است

1. عملیات جمع

برای پیاده سازی این عملیات از ماژول lpm_add_sub استفاده شده (از قسمت add آن) و ورودی های in1 و in2 و cin را میگیرد و out_add[31..0] و سیگنال add_overflow در خروجی بیرون می دهد . out_add[31..0] که نتیجه عملیات می باشد و به یک MUX می رود که در صورتی که دستور ما جمع بود خروجی به همان out اصلی برود . از سیگنال add_overflow و Opcode برای اینکه بفهمیم Overflow داریم یا نه استفاده میشود . چون گفته شده Overflow تنها برای دستورات جمع می باشد

2. عملیات تفریق

برای پیاده سازی این عملیات نیز از ماژول lpm_add_sub استفاده شده (از قسمت sub آن) و همانند ماژول بالا کار میکند ولی تنها در خروجی out_sub[31..0] می باشد که به MUX ما میرود که در صورت اینکه Opcode ما برای این عملیات باشد ، این خروجی انتخاب میشود

3. عملیات های شیفت به راست و چپ

برای پیاده سازی این عملیات ها از ماژول lpm_clshift استفاده شده (در یکی right shift و دیگری left shift) و ورودی های in1 و shiftamt را میگیرند و in1 را به مقدار shiftamt شیفت می دهد و در خروجی out_shiftright[31..0] و out_shiftleft[31..0] را میدهند

4. عملیات NAND

برای پیاده سازی این عملیات از ۲ ماژول lpm_and و lpm_inv استفاده شده که در ابتدا بیت ها را متناظر AND میکنیم و سپس NOT میکنیم . پس خروجی lpm_and را به ورودی lpm_inv میدهیم . ورودی lpm_and ، in1 و in2 می باشند و خروجی lmp_inv نتیجه این عملیات ها می باشد با اسم out_nand[31..0]

5. عملیات set on less than

برای پیاده سازی این عملیات از ماژول lpm_compare استفاده شده که ۲ ورودی in1 و in2 را میگیرد و خروجی های زیر را تولید میکند

- Eq که همان سیگنال خروجی ماژول ALU ما می باشد و تساوی in1 و in2 را مشخص میکند
- out_slt0 که همان خروجی مورد نظر برای این دستور می باشد . (دقت کنید که این ماژول ۱ بیت خروجی میدهد ولی ما در خروجی ALU باید ۳۲ بیت خروجی دهیم برای همین بیت های [31..1] out_slt را به GND وصل کرده ایم که مقدار ۰ بگیرند
- le_signal که این سیگنال مشخص می کند که آیا $in1 \leq in2$ هست یا خیر و برای پیاده سازی دستور min استفاده می شود

6. عملیات min

برای پیاده سازی این عملیات از یک MUX استفاده کردیم و ورودی های دیتای آن همان in1 و in2 هستند و برای ورودی آدرس از سیگنال le_signal که در ماژول slt بالا توضیح داده شد استفاده میکنیم . در صورتی که $in1 \leq in2$ ، in1 به خروجی می رود و در غیر این صورت in2 به خروجی می رود

- **سایر پیاده سازی ها**

- برای پیاده سازی سیگنال Zero صرفا خروجی نهایی را با ۰ مقایسه می کنیم و در صورت تساوی این سیگنال ۱ میشود .
- برای پیاده سازی سیگنال Sign نیز صرفا بیت سمت چپ (پر ارزش out) را به آن میدهیم (در صورت منفی بودن ou31 ، ۱ میشود که Sign نیز ۱ میشود و در غیر اینصورت هر دو ۰ می شوند)

در نهایت همه ی out های ماژول های عملیات های بالا را به ورودی های دیتای یک MUX وصل میکنیم و به ورودی های آدرس آن Opcode[2..0] را میدهیم که متناسب با دستور ، خروجی درست به بیرون (همان out[31..0] برود)

- **Waveform**

برای تست ماژول به صورت مستقل از CPU و جداگانه برای آن یک waveform طراحی شده که در پیوست فایل موجود می باشد با نام ALU_Waveform که در آن به صورت کامل عملکرد صحیح اجزا بررسی شده

ماژول Instruction Cache

در این ماژول همانطور که در داک پروژه گفته شد طراحی انجام شد . ولی به جای ۱۶ بیت آدرس از ۱۵ بیت آدرس استفاده شد چرا که هنگامی که ۱۶ بیت برای آدرس در نظر گرفته می شد و ورودی کلاک نیز به آن داده میشد ماژول کامپایل نمیشد و ارور میداد ! ولی تفاوتی هم در اجرای دستورات ایجاد نمی شود چرا که دستور به اندازه ای که چنین حافظه ای را پر کند وجود ندارد ! برای افزایش سایز I_Cache نیز میتوانستیم از کنار هم قرار دادن ماژول های کوچک تر و استفاده از MUX این کار را انجام دهیم ولی با توجه به گفته TA در تالار درس نیازی به این کار نبود .

برای پیاده سازی نیز صرفا ۱۵ بیت پایینی PC را به ماژول مورد نظر داده و با توجه با آدرس یکی از خانه های ROM که در آن دستور قرار دارد (۲۰ بیتی می باشد) به ما داده می شود . (لیست دستورات در پایین)

برای تست کلی CPU نیز دستوراتی طراحی شده که در جدول زیر مشاهده میکنید که همه ی خروجی های ماژول CPU تست شود .

```
1  load R0 , 1
2  load R1 , 2
3  load R2 , 1
4  load R3 , 2
5  load R4 , 1
6  add R0 , R1 , R2  ( cin = 0 )
7  sub R3 , R1 , R2
8  srl R4 , R2 , 2
9  sll R5 , R3 , 3
10 nand R5 , R3 , R4
11 min R3 , R1 , R2
12 slt R2 , R0 , R1
13 load R6 , 1
14 load R7 , 1
15 slt R8 , R6 , R7 (eq check)
16 sub R8 , R6 , R7 (zero check)
17 load R9 , 2^9
18 load R10 , 2^9
19 sll R9 , R9 , 22
20 sll R10 , R10 , 22
21 add R11 , R10 , R9 (overflow check)
22 sub R12 , R8 , R7 (sgn check)
23 add R0 , R1 , R2 ( cin = 1 )
```

ماژول Control Unit

برای طراحی این بخش ابتدا تمام سیگنال های کنترلی مورد نیاز برای بخش های مختلف cpuمان را بدست می آوریم.

سیگنال های مورد نیاز عبارتند از:

۱) MemRead :

با یک شدن این سیگنال مقداری که در خانه با آدرس ورودی قرار دارد از حافظه دستور خوانده می شود.

۲) RegWrite :

با یک شدن این سیگنال مقدار موجود در ثباتی که بیت های ۱۰ تا ۱۴ (ReadReg1)و نیز ثباتی که بیت های ۵ تا ۹ (ReadReg2) نشان می دهند از رجیستر فایل خوانده می شوند و به ترتیب در ReadData1 و ReadData2 قرار می گیرد.

۳) CinSrc :

این سیگنال مربوط به سیگنال کنترلی مالتی پلکسری است که در مسیر ورودی Cin ، ALU قرار دارد که در صورت ۱ شدن بیت Cin از دستور وارد بخش Cin مالتی پلکسر می شود و در صورت ۰ شدن مقدار ۰ در Cin در ALU قرار می گیرد.

۴) AluSrcA :

سیگنال کنترلی ماکسی است که در ورودی اول ALU قرار دارد که در صورت ۱ شدن آن مقدار ثبات Aکه همان ReadData1 است وارد ALUمی شود و در صورت ۰ شدن مقدار PC در آن قرار می گیرد.

۵) AluSrcB :

سیگنال کنترلی ماکسی است که در ورودی دوم ALU قرار دارد و در صورت ۱ شدن مقدار ۴ در ALU قرار میگیرد(برای جمع شدن مقدار فعلی PC با ۴ و تولید مقدار PCجدید) و در صورت ۰ شدن مقدار ثبات Bکه همان ReadData2 است وارد ALU می شود.

۶) AluOp :

این سیگنال نیز ۳ بیتی است و به ALU می رود تا نوع عملی را که قرار است بر روی عملوند ها قرار بگیرد را تعیین کند و چون ۷ نوع عمل در ISA ما موجود است پس ۷ عمل نیاز به ۳ بیت برای تفکیک دارند.

۷) PCLoad:

سیگنال کنترلی PCبرای لود شدن مقدار جدید PCیعنی PCقبلی به اضافه ۴ در آن.

۸) IRWrite :

سیگنال کترلی IRبرای نوشتن در آن باید سیگنال ۱ شود.

۹) LoadImm :

این سیگنال کنترلی برای ماکسی است که یکی از ورودی هایش مقداری است که از کنار هم قرار دادن بیت های ۵ تا ۱۴ و ۲۲ بیت ۰ درکنار هم ایجاد می شود و ورودی دیگر خروجی ALUOut است.

این پردازنده که طراحی آن به صورت مالتی سایکل است هر دستور آن برای اجرا به ۴ کلاک نیاز دارد:

کلاک اول(IF):

این کلاک در تمامی دستورات مشترک است و ۲ عمل در این بخش انجام میگیرند :

1) IR <= Mem[PC]

2) PC <= PC + 1

کلاک دوم (ID & RR):

این کلاک نیز در تمامی دستورات مشترک است و ۳ عمل در این بخش انجام میگیرد:

1) A <= GPR[IR[14-10]

2) B <= GPR[IR[9-5]

3) AluOut <= 22{0} + IR[14-5]

کلاک سوم:

در این کلاک با توجه به نوع دستورات عملیات خاصی انجام میگیرد:

۱) دستوراتی که به in1و in2 نیاز دارند:

AluOut <= in1 op in2

۲)دستوراتی که به in1و shamt نیاز دارند:

AluOut <= in1 (>> or <<) shamt

۳)دستور ذخیره مقدار shamtدر بانک ثبات :

GPR[IR[4-0]] <= AluOut

این نوع از دستورات در کلاک سوم تمام می شوند.

کلاک چهارم:

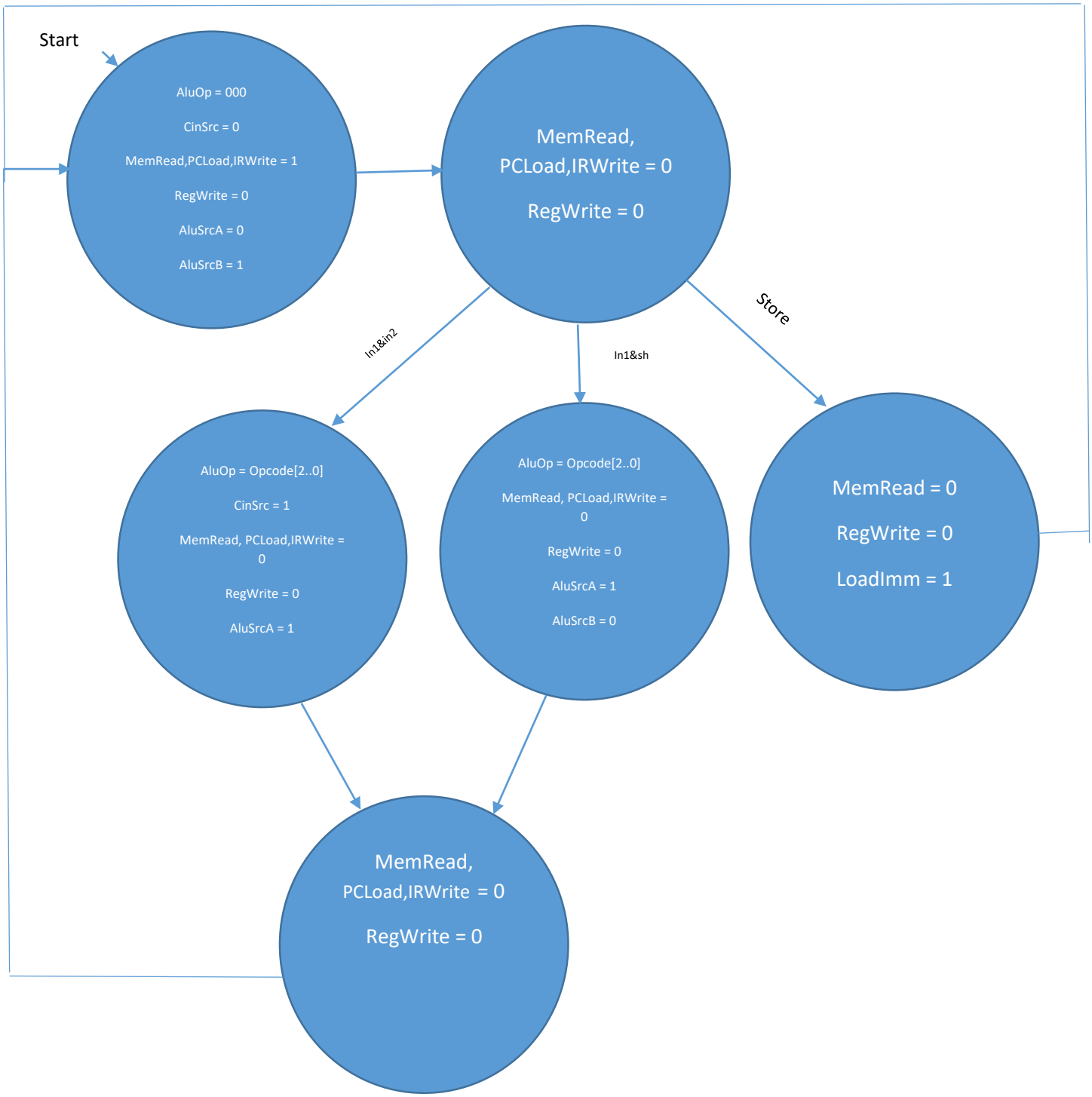
این کلاک نیز در دستورات بخش ۱ و ۲ کلاک سوم مشترک است و عمل زیر انجام میگیرد:

GPR[IR[4-0]] <= AluOut

شکل FSM آن به صورت زیر خواهد بود و چون ۵ حالت موجود است پس به ۳ بیت برای state نیاز خواهیم داشت و در نهایت یک ROM طراحی میکنیم که opcode و state را ورودی میگیرد (۴+۳ = ۷)و سیگنال های کنترلی و next state را خروجی می دهد (11+۳ = 14).

این ROM ۷ بیت ورودی و 14 بیت خروجی دارد که حجم آن برابر است با:

$14 * 2^7 =$ ۴/۶۶ KB


Start
AluOp = 000
CinSrc = 0
MemRead,PCLoad,IRWrite = 1
RegWrite = 0
AluSrcA = 0
AluSrcB = 1
MemRead, PCLoad,IRWrite = 0
RegWrite = 0
In1&in2
In1&sh
Store
AluOp = Opcode[2..0]
CinSrc = 1
MemRead, PCLoad,IRWrite = 0
RegWrite = 0
AluSrcA = 1
AluOp = Opcode[2..0]
MemRead, PCLoad,IRWrite = 0
RegWrite = 0
AluSrcA = 1
AluSrcB = 0
MemRead = 0
RegWrite = 0
LoadImm = 1
MemRead, PCLoad,IRWrite = 0
RegWrite = 0

طراحی CPU

با توجه به گزارش های ALU و Control Unit و Instruction Cache و همچنین Register File که در تمارین زده شده بود در ۴ ماژول برای طراحی در اختیار داریم که نحوه ی کارکرد کامل آن ها در گزارش ها ذکر شده است .

* نکته : برای طراحی PC برای اینکه به طراحی پردازنده Multi-Cycle که در درس داشتیم بیشتر شبیه باشد آن را جدا طراحی و پیاده سازی کرده و از تمرین استفاده نشده .

طراحی PC :

برای طراحی یک lpm_shiftreg قرار داده شده که مقدار PC را در خود نگه می دارد . یکی از مشکلات lpm_shiftreg برای این طراحی این است که وقتی کلاک آن را به کلاک سیستم وصل میکنیم و اگر load آن ۰ باشد مقدار درون آن Shift میخورد که چیزی نیست که ما مد نظر داریم . برای همین مورد ما به طور کلی همه load های lpm_shiftreg ها را به Vcc وصل کرده ایم . برای این PC نیز کلاک آن را PCLoad که از CU می آید وصل کردیم که در جای مناسب PC مقدار دهی شود (در اصل PC+1 در آن قرار گیرد) ورودی آن نیز خروجی ALU می باشد .

سایر طراحی :

برای توضیح سایر طراحی CPU از سمت چپ به راست شکل را توضیح می دهیم . بعد از واحد PC واحد I_Cache را مشاهده می کنیم که ورودی آن PC و Clk می باشند که منطقی نیز می باشد و خروجی آن را به Instruction Register وصل کرده ایم که در اصل Invisible Register در طراحی ما می باشد . در بالای I_Cache نیز مشاهده میکنید که مقدار Immediate از خروجی IR بدست آمده که در صورتی که دستور ما Load بود از آن استفاده میکنیم .

مقدار Immediate و همچنین خروجی رجیستر ALUOut به یک ماکس رفته که خروجی این MUX ورودی Write Data در Register File را مشخص میکند . همچنین سیگنالی که این مقدار را تعیین میکند و Load_Immediate نام دارد از CU می آید . سپس رجیستر فابل را داریم که همانطور مشاهده میکنید مقادیری که باید به ورودی آن داده شده (مقدار Read_Reg_1 و Read_Reg_0 و Write_Reg از بخش های IR می آیند . سیگنال Write نیز از CU . خروجی های این Register File نیز به دو رجیستر (که این ها نیز Invisible Registers می باشند) می روند .

قبل از ALU ۲ عدد MUX داریم که MUX اولی برای انتخاب بین RegA و PC می باشد و MUX دومی برای انتخاب بین Constant ای که مقدار ۱ دارد و RegB . سیگنال های ALUSrcA و ALUSrcB نیز از CU به این MUX ها می آیند .

* نکته : توجه کنید در اینجا به جای PC+4 از PC+1 استفاده شده چرا که در ROM ای که در Instrcution Cache استفاده شده Word Addressable بوده و آدرس ها یکی یکی زیاد می شوند . میتوانستیم با ۴ نیز طراحی کنیم ، اینطور که ۲ بیت پایینی PC را ۰ همیشه در نظر گرفته و از بیت ۲ به بعد به Instruction Cache آدرس دهیم و ...

در ALU نیز ورودی های A و B که ۲ ورودی عددی آن می باشند به آن داده شده و مقدار Shift Amount که از Instruction آمده و Opcode که از سمت CU می آید و مقدار Cin که از دستور می آید (در صورتی که دستور جمع بوده و ۱ وارد شده . این توسط CU کنترل شده و سیگنال CinSrc به ماکس آن داده شده)

بعد از ALU نیز رجیستر ALUOut می باشد که خروجی ALU به آن رفته که آن نیز Invisible Register می باشد .

تست Waveform ماژول اصلی (CPU) و زیر ماژول ها :

برای تست نیز همانطور که در گزارش Instruction Cache گفته شد ، مجموعه دستوراتی درنظر گرفته شده که همه ویژگی های CPU و سیگنال های ماژول ها تست شود . از سیم های زیادی نیز خروجی گرفته شده که در Waveform آن هارا مشاهده میکنید که عملکرد صحیح را مشاهده کنید .

برای اطمینان از کارایی زیر ماژول هایی همچون ALU و Control Unit و Register File برای آن ها Waveform هایی تشکیل شده که در فایل مربوطه آن ها مشاهده میکنید .