بسمه تعالی

دانشکده مهندسی برق و کامپیوتر

دانشگاه صنعتی اصفهان

زبان های توصیف سخت‌افزار و نرم‌افزار – نیمسال دوم ۹۹-۱۳۹۸

پروژه پیاده سازی معماری MIPS روی FPGA

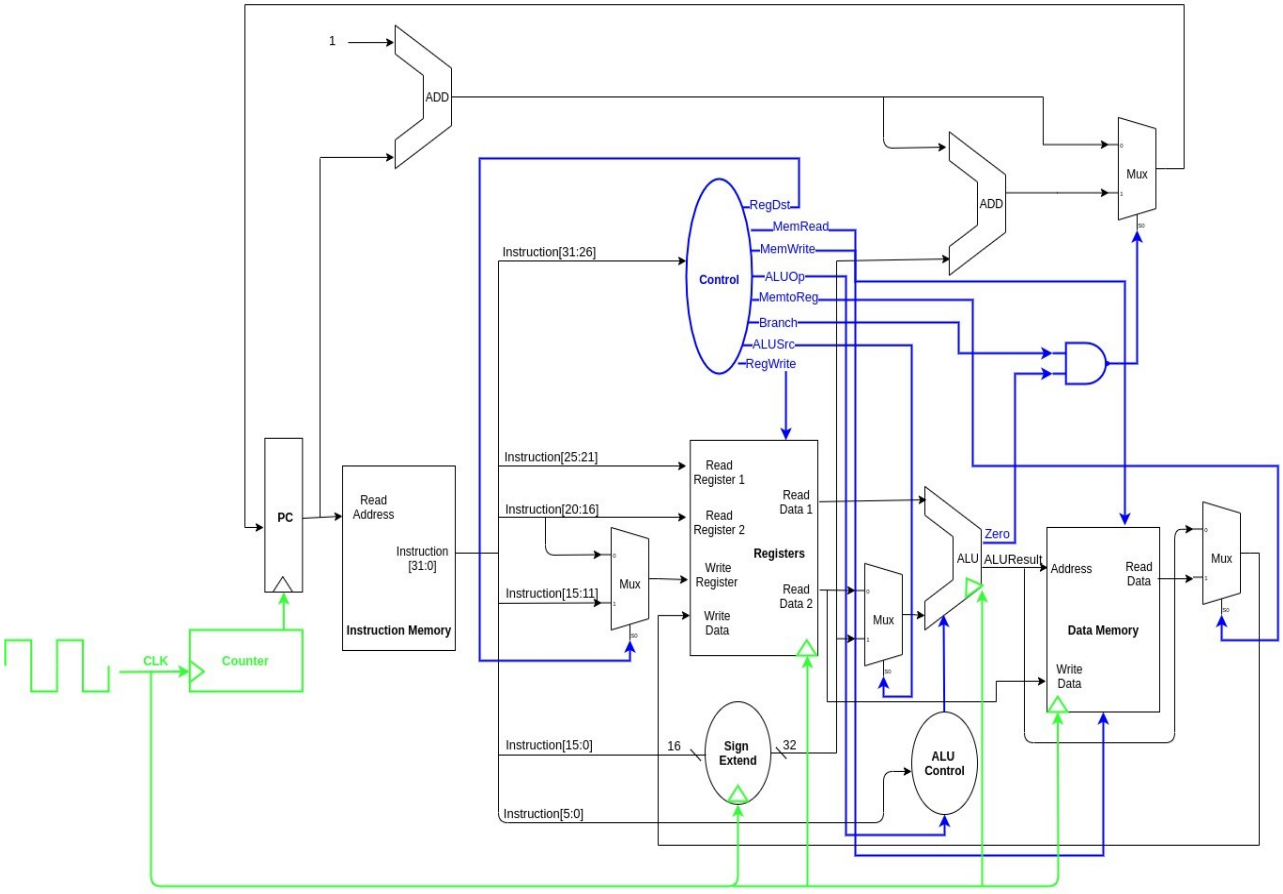
مریم سعید مهر

ش.د. : ۹۶۲۹۳۷۳

✔ فاز اول : پیاده سازی کامپیوتر MIPS

تمام ماژول های خواسته شده ، طراحی ، سیموله و سنتز شدند و تمام فایل‌ها (اعم از اسکرین‌شات‌های سیمولیشن و مدارهای RTL و گزارش سنتز و کد ماژول و کد تست‌بنچ) در پوشه‌ی آپلود شده در سامانه موجود است.

مداری مطابق شماتیک زیر طراحی و پیاده سازی شده است .(نمودار RTL هم در فایل‌ها موجود است)



توضیحات لازمه :

اولاً دستورات mips در یک کلاک قابل اجرا شدن نیستند در حالی که اگر به رجیستر pc همان کلاک اصلی را بدهیم ، معادل این است که در هر کلاک یک دستور جدید از instruction memory در حقیقت fetch می‌شود در حالی که دستور قبلی هنوز کامل نشده و این موضوع خیلی خطرناک است زیرا در واقع کامپیوتر ما اصلا درست کار نمی‌کند. برای حل این مشکل دو راه داریم : ۱- استفاده از pipeline . و یا ۲- تغییر کلاک pc به اندازه مورد نیاز.
استفاده از خط‌لوله به منظوری موازی سازی اجرای دستورات ، کمی پیچیده و پیاده‌سازی آن زمان‌بر است و مهم‌تر از آن مورد خواسته‌ی صورت پروژه نیست و پروژه درباره‌ی پیاده‌سازی mips تک سایکل است . پس تنها راه‌حل ما همان تغییر کلاک pc به اندازه‌ای که تضمین کند دستور جاری به طور کامل اجرا می‌شود و سپس دستور بعدی fetch میگردد. برای بررسی اینکه به چند کلاک دیلی نیاز داریم تا دستور جاری کامل شود ، کافیست یک بار مسیر اجرایی را طی کنیم ، به همین روش ، برای اجرای هر دستور حداقل به سه کلاک دیلی نیاز داریم. لذا یک شمارنده که هر سه کلاک یک بار یک پالس ۱ میدهد ایجاد کردم که در عمل خروجی آن ، همان کلاک pc است اینجوری ، pc هر سه کلاک یک بار دستور جدید را fetch می‌کند و در نهایت عملکرد کامپیوتر ما هم درست خواهد شد.
دوماً ، برای پیاده سازی ماژول‌های Add و Increment که خواسته‌ی صورت پروژه بود ، نیازی به ماژول های جداگانه نبود و میشد از همان ALU که قبلا پیاده کرده بودیم استفاده کنیم ولی به جهت ارضای خواسته‌های مسئله برای هر یک ، ماژول جداگانه ای طراحی ، پیاده ، سیموله و سنتز شد!! (فایل ها در پوشه های جداگانه ضمیمه شده اند)
در آخر ، اگر به مدار RTL نهایی کامپیوتر-MIPS (که در پوشه هست) مراجعه کنید ، برای کامپیوتر ، خروجی های غیرضروری تعریف شده است ! دلیل این مسئله این بود که وقتی (قبل از تعریف این پورت‌های خروجی غیرضروری) ماژول را سنتز میکردم ، ISE در نمودار RTL برخی بخش ها را به دلیل عدم اتصال به تاپ‌ماژول کلی ، ignore میکرد و در واقع سنتز نمیشدند (یا شاید هم سنتز میشدند ولی در نمودار RTL نمایش داده نمیشدند) به هر حال ، برای قابل مشاهده بودن این بخش‌های حذف شده(توسط ISE) ناچار شدم از آنها خروجی به پورت‌های تاپ‌ماژول اصلی وصل کنم. این خروجی ها در تست‌بنچ استفاده نشده اند ولی در سیمولیشن ، برخی از آنها به کار آمد :) . در کل خیلی به این پورت ها توجهی نکنید.

---

* توجه ! شماتیک فوق (که در صفحه ۱ آورده شده) را با استفاده از draw.io رسم کرده ام . خطوط سبز رنگ بیانگر کلاک و خطوط آبی رنگ ، سیگنال های کنترلی هستند.
* توجه ! فایل MIPS_Shematic.html که حاوی شماتیک ماژول طراحی شده (به همراه شماتیک فاز سوم پروژه) ضمیمه شده است. برای باز کردن کافیست به اینترنت متصل باشید. برای جابه‌جا شدن بین صفحات و مشاهده‌ی شماتیک فاز سوم ، از فلش‌های بالا سمت چپ صفحه استفاده کنید :) (کافیست موس را روی شماتیک قرار دهید تا فلش ها قابل مشاهده باشند)

✔ فاز دوم : تست کامپیوتر MIPS

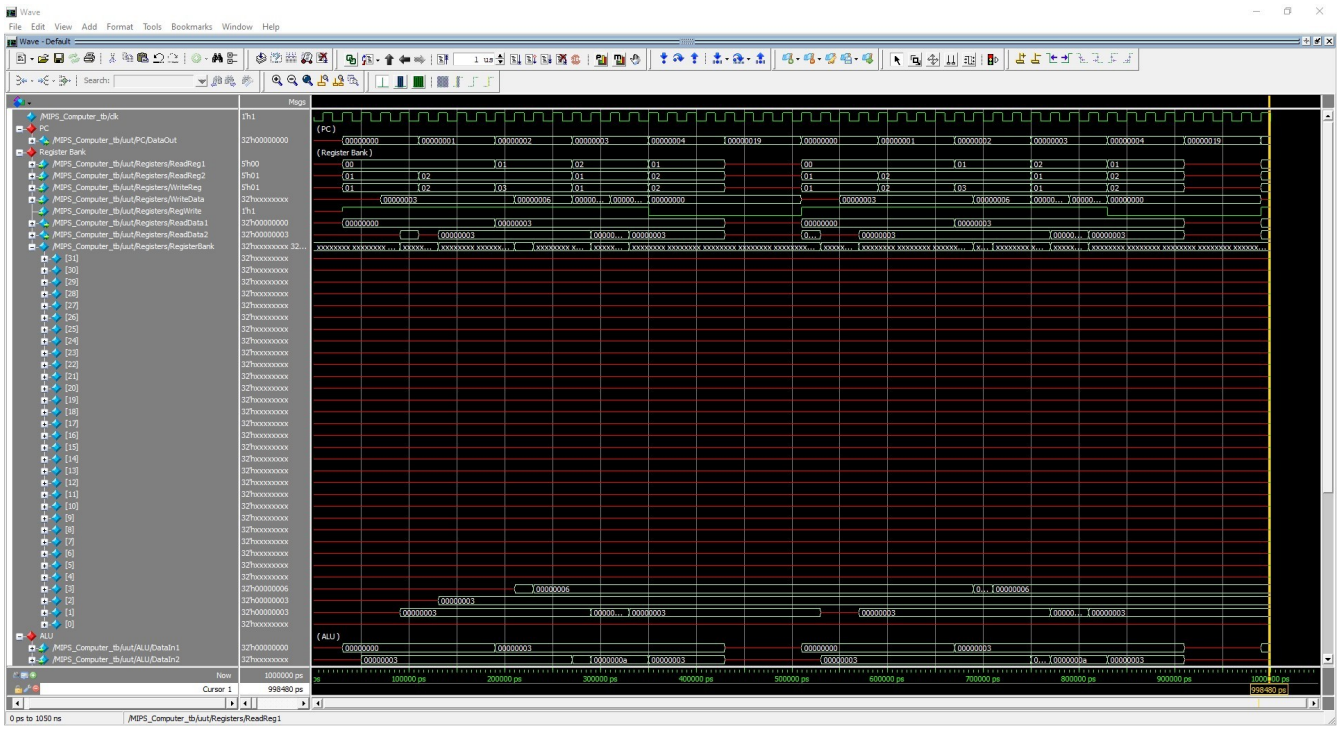
عکس و اسکرین‌شات‌های مربوط به سیمولیشن ، در فایل‌های آپلود شده ، ضمیمه شده است .
برای سیمولیشن mips ، دستوراتی که در فایل instruction.mem وجود داشت را با استفاده از قطعه کد زیر ، در Instruction Memory لود و سپس سیموله کردم.

```
$readmemb("instruction.mem",ROM,0,4);
```

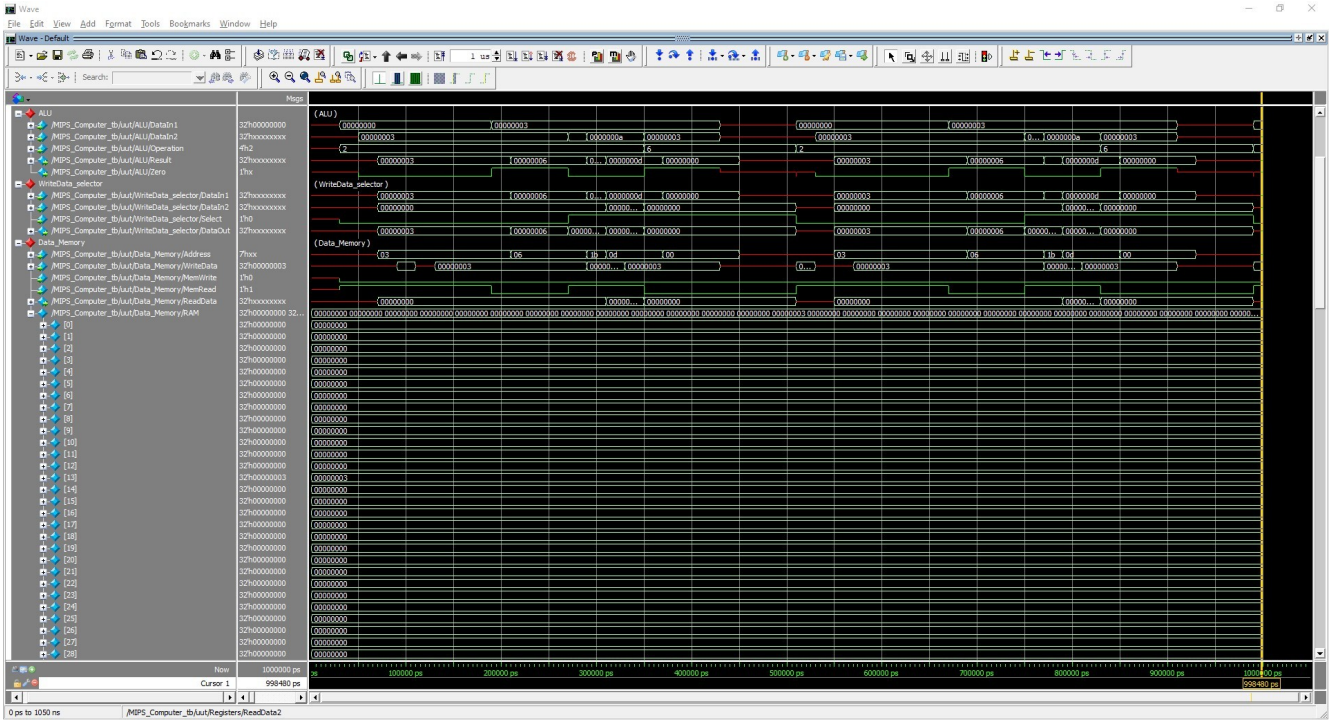
توجه : قراردادن عکس‌ها در اینجا باعث کاهش کیفیتشان میگردد :((اما با این حال بازم اینجا میاورمشان.
توجه : به دلیل اینکه بتوانم تا جای ممکن ، مواردی که کمک میکند تا درستی mips که پیاده‌سازی کردم ، چک شود را در سیمولیشن بیاورم ، آنها را گروه بندی کردم و طی ۳ اسکرین‌شات از سیمولیشن ، ضمیمه کرده ام.

توضیحات لازمه :

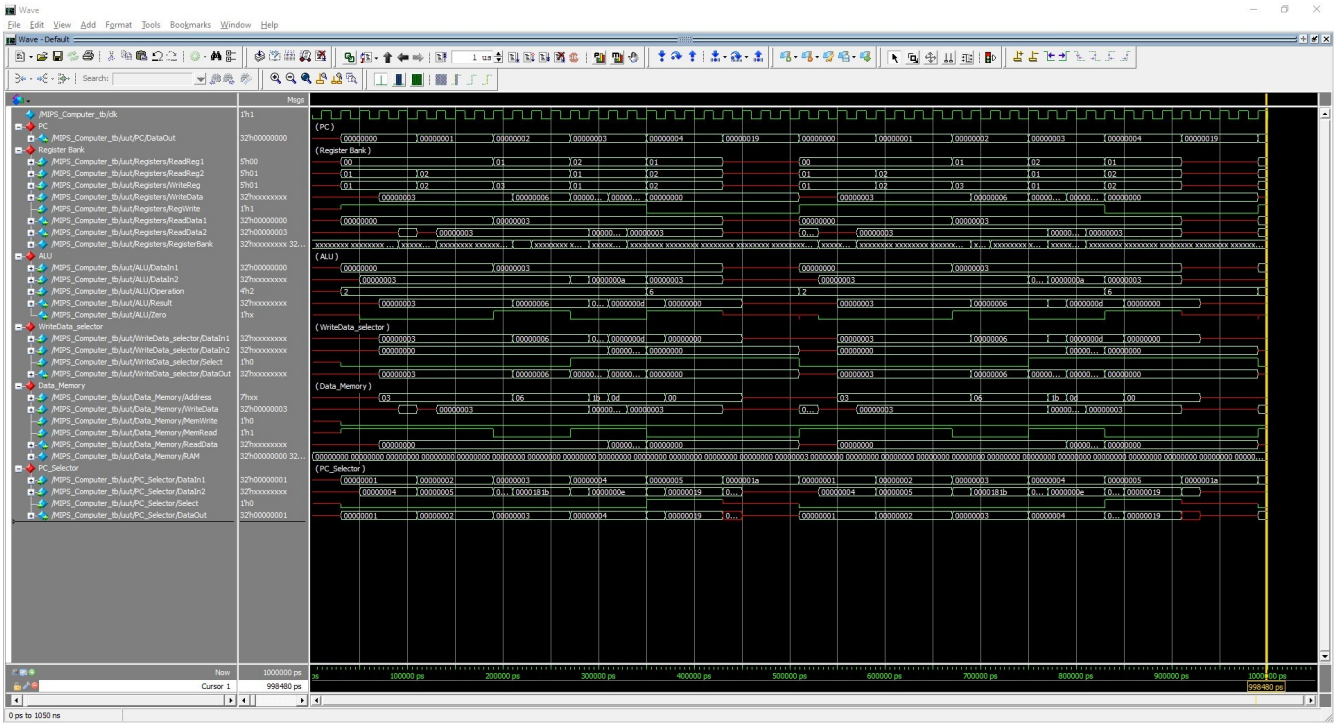
برای بررسی درستی کافیست محتوای بانک رجیستر(Register Bank) ، ALU ، PC و WriteData_Selector (همان مالتی‌پلکسر بعد واحد Data Memory برای بررسی سه instruction اول (بخش اول فاز دوم) و همچنین برای بررسی دو instruction دوم (بخش دوم فاز دوم) کافیست ALU , Data Memory , PC و PC_Selector (همان مالتی‌پلکسر بعد از واحد Add که مقدار نهایی PC را مشخص میکند) بررسی شوند . که همگی این موارد در این سه اسکرین‌شات موجود هستند.



Picture 1 : MIPS_Simulation_scr1.jpg



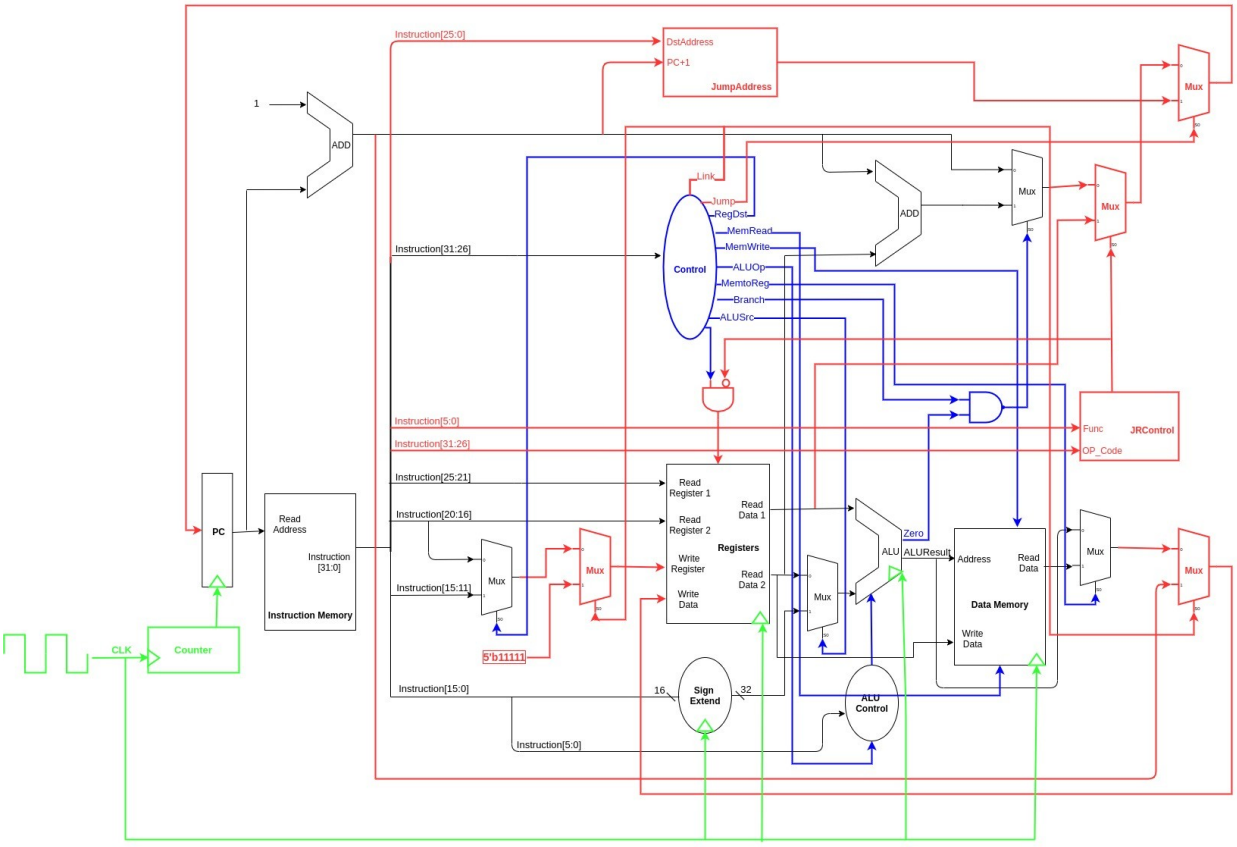
Picture 2 : MIPS_Simulation_scr2.jpg



Picture 3 : MIPS_Simulation_scr3.jpg

✔ فاز سوم : ارتقا کامپیوتر MIPS جهت پشتیبانی از دستورات Jump , Jal , Jr

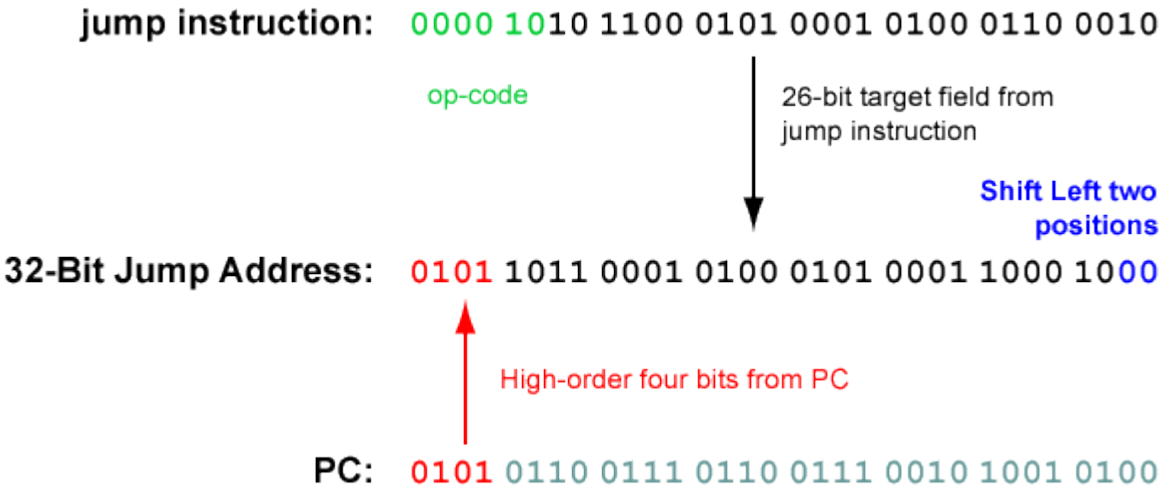
در ابتدا شماتیک سیستمی که در نهایت طراحی کرده ام :



* قسمت‌های قرمز رنگ ، همان بخش های جدیدی هستند که به منظور پشتیبانی از دستورات jump , jump and link و jump register اضافه شده اند.

در ادامه توضیح خواهم داد که چرا هر یک از این واحدها اضافه شده‌اند.

۱- Jump Address : قراره برامون آدرس جایی که jump بهش انجام می‌شود را محاسبه کند



طریقه‌ی محاسبه‌ی آدرس نهایی ، در تصویر فوق آورده شده است و عملکرد این ماژول هم دقیقا همین هست :

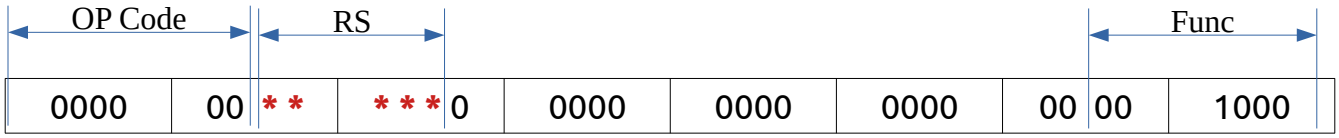
```
module JumpAddress(
        input [25:0] Address,
        input [31:0] PC2,
        output [31:0] Jump_Address
        );

      assign Jump_Address = {PC2[31:28],Address,2'b00};

endmodule
```

۲- JRControl : همانطور که میدانیم ، دستور jr یک دستور R-Type است. و عملکرد آن مطابق جدول زیر است .

Jump Instructions

Instruction	Opcode/Function	Syntax	Operation
j	000010	o label	pc += i << 2
jal	000011	o label	$31 = pc; pc += i << 2
jalr	001001	o labelR	$31 = pc; pc = $s
jr	001000	o labelR	pc = $s

بیت کد ، دستور jr به شکل زیر است :



پس این واحد صرفاً کنترل میکند که اگر دستور R-Type با OP Code و Func مطابق فوق وجود داشت ، یک سیگنال کنترلی تولید میکند.

```
module JR_Control(
        input [1:0] ALU_op,
        input [3:0] Func,
        output JRControl
        );

        assign JRControl = ({ALU_op,Funct}==6'b001000) ? 1'b1 : 1'b0;

endmodule
```

۳- ۲ مالتی‌پلکسر هم در مسیر PC قرار گرفته که با سیگنال های کنترلی Jump و JRControl (که به ترتیب توسط واحد Control و JRControl تولید می‌شوند ،) تصمیم می‌گیرند تا PC دستور بعدی (PC + 1) را fetch کند یا Jump کند (به آدرسی که واحد JumpAddress تولید می‌کند) یا محتوای یک رجیستر (که آدرس آن در دستور jr قرار دارد ،) داخل PC لود شود.

```
Mux2x1_32 PC_Selector_JR (
.DataIn1(PC_Next),
.DataIn2(ReadData1),
.Select(JRControl),
.DataOut(PC_JRNext)
);

Mux2x1_32 PC_Selector_Jump (
.DataIn1(PC_JRNext),
.DataIn2(Jump_Address),
.Select(Jump),
.DataOut(PC_FinalNext)
);
```

۴- ۲ مالتی‌پلکسر هم در مسیر Register Bank قرار دارند ، یکی WriteRegister (آدرس رجیستری که میخواهیم چیزی در آن بنویسیم) را انتخاب میکند و دیگری WriteData (دیتایی که قرار است داخل رجیستری که WriteRegister آدرسش را مشخص کرده رایت شود) را انتخاب میکند.

این مورد به دلیل پشتیبانی از دستور jump and LINK است . به دلیل لینک شدن (یعنی ذخیره ی PC + 1 در رجیستر شماره ۳۱) باید مقدار PC + 1 به رجیستر بانک منتقل شود و در آدرس $31 ذخیره گردد ، که این کار با دو مالتی‌پلکسر و یک سیگنال کنترلی Link که به واحد کنترل اضافه شده ، محقق می‌شود.

```
// Jump and Link
Mux2x1_32 RegWrite_Selector_Link (
.DataIn1(WriteReg),
.DataIn2(5'b11111), // $31
.Select(Link),
.DataOut(WriteReg_Final)
);

Mux2x1_32 DataWrite_Selector_Link (
.DataIn1(WriteData),
.DataIn2(PC2),
.Select(Link),
.DataOut(WriteData_Final)
);
```

۵- در مورد دستور jr که از نوع R-Type است ، میدانیم که واحد کنترل به رجیستر بانک امکان نوشتن در رجیستر مورد نظر (که آدرسش را WriteRegister فراهم میکند) را میدهد در حالی که در این مورد خاص (دستور jr) نباید این اتفاق بیوفتد چرا که خروجی ALU و محتوای رجیسترها در بانک رجیستر را تغییر میدهد و این خطرناک است ، لذا برای جلوگیری از این اتفاق ، سیگنال کنترلی RegWrite که در واحد کنترل آماده میشود را باNOT سیگنال JRControl در حقیقت AND میکنم و خروجی این AND به بانک رجیستر میگوید که آیا اجازه دارد در رجیستری ، چیزی بنویسد یا نه .

```
and JR_RegWrite(RegWrite_Final,RegWrite,~JRControl);
```

۶- ویرایش واحد کنترل :

دستور jr که یک دستور R-Type است و نیازی به تغییر در واحد کنترل نداشت .

برای دستورات jump , jal این دو قسمت اضافه شده است .

در مورد دستور jump ، در اصل به بقیه واحدها نیاز نداریم و تنها سیگنال Jump فعال شود کفایت میکند.

در مورد دستور jal هم علاوه بر سیگنال کنترلی Jump باید سیگنال Link و RegWrite هم فعال باشند تا آدرس PC+1 در رجیستر $31 ذخیره شود و سپس پرش به آدرس مورد نظر صورت گیرد.

```
else if(inst_in == 6'b000010)//jump
begin
RegDst = 0;
Jump = 1;
Link = 0;
ALUsrc = 0;
MemtoReg = 0;
RegWrite = 0;
MemRead = 0;
MemWrite = 0;
Branch = 0;
ALUop[1] = 0;
ALUop[0] = 0;
end

else if(inst_in == 6'b000011)//jal
begin
RegDst = 0;
Jump = 1;
Link = 1;
ALUsrc = 0;
MemtoReg = 0;
RegWrite = 1;
MemRead = 0;
MemWrite = 0;
Branch = 0;
ALUop[1] = 0;
ALUop[0] = 0;
end
```

۷- ویرایش ALU Control :

```
if(inst == 6'b001000)//jr
begin
op = 4'b0010;
end
```

در این قسمت تغییر خیلی خاصی نکرده (البته تغییر کرده ولی تغییر مفیدی نبوده) چون من با به کار بردن ماژول JRControl و مالتی‌پلکسرهایی این دستور را پشتیبانی کرده ام و نیازی به هیچ عملیاتی در ALU نبوده ولی به دلیل اینکه به هرحال این دستور از R-Type است و معمولا این نوع دستورات با ALU کار دارند ، این قسمت به واحد کنترل ALU اضافه شده.

---

* توجه ! کدهای این واحدهای جدید ، در فایل‌های آپلود شده ، ضمیمه شده است . (همچنین در کدها کاملا مشخص است که کدام قسمت ها تازه اضافه شده اند ، به این دلیل که با استفاده از کامنت مشخص شده اند)

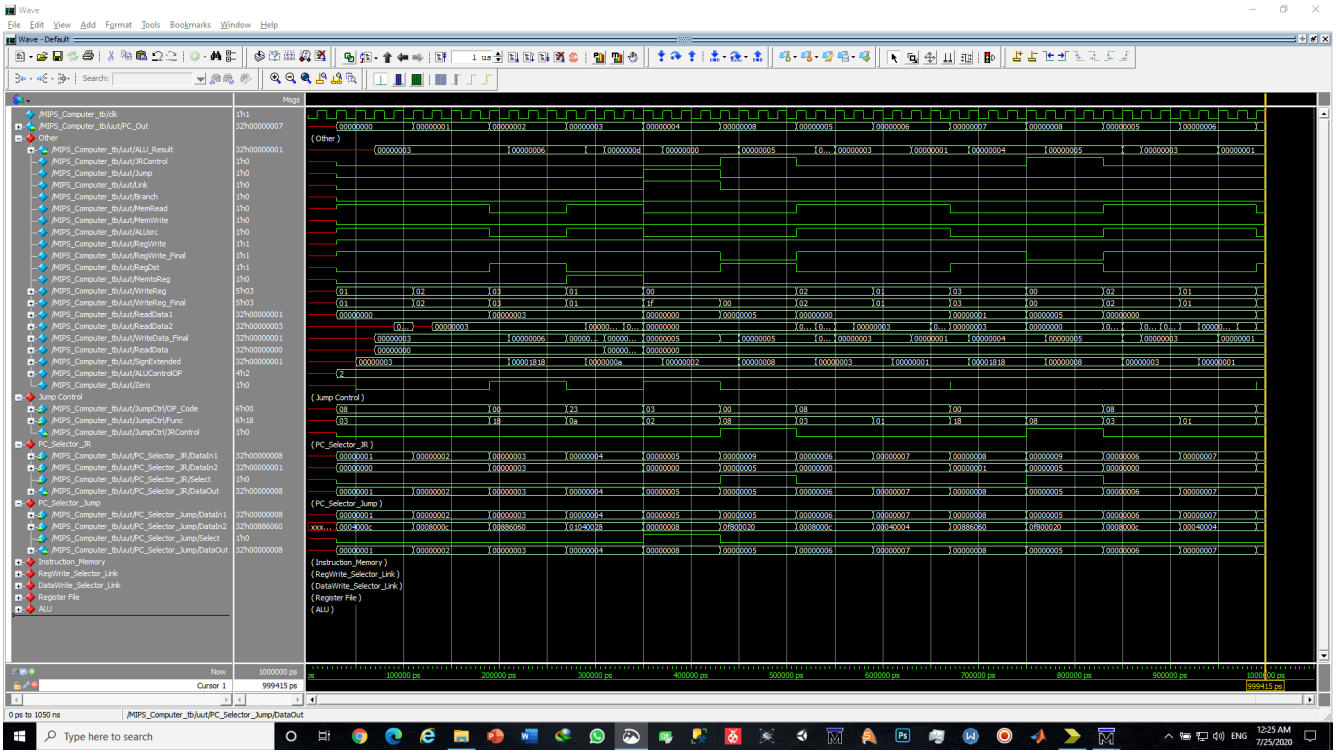
*** باز هم یادآوری میکنم ، تمام ماژول‌ها سنتز و سیموله شده اند و اسکرین‌شات های سیمولیشن و مدارات RTL آنها در پوشه ضمیمه شده است . همچنین گزارش سنتز آنها هم در کنار این اسکرین‌شات‌ها قرار دارد .**

تست فاز سوم :

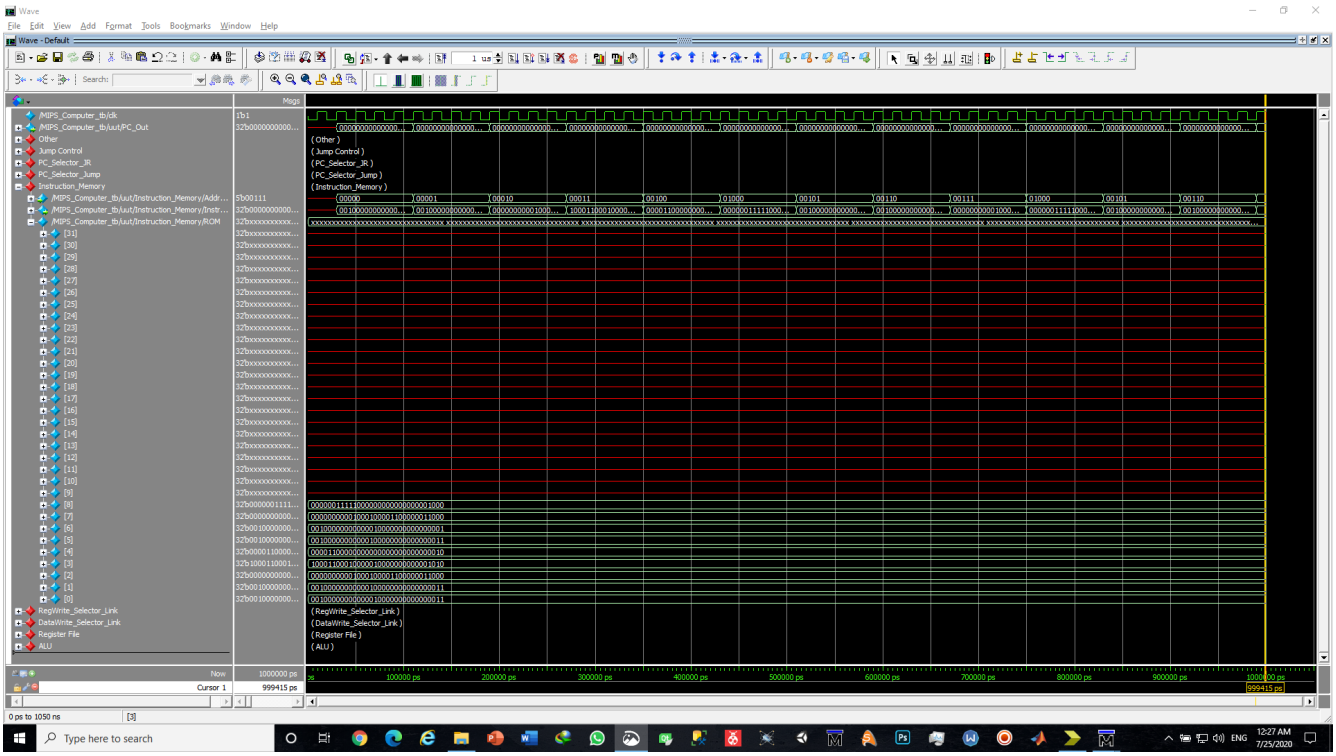
برای تست این مرحله ، دستورات jal و jump را به instruction.mem اضافه کردم ، در اصل بیت‌کدی که در instruction.mem قرار دارد، بیانگر قطعه کد زیر است :

```
0    R1 = R0 + 3;
1    R2 = R0 + 3;
2    R3 = R1 + R2;
3    R1 = Mem(R2 + 100);
4    jal  8 ; // jump to line 8 and link
5    R2 = R0 + 3;
6    R1 = R0 + 1;
7    R3 = R1 + R2;
8    jr  $31; // return to line 5
```

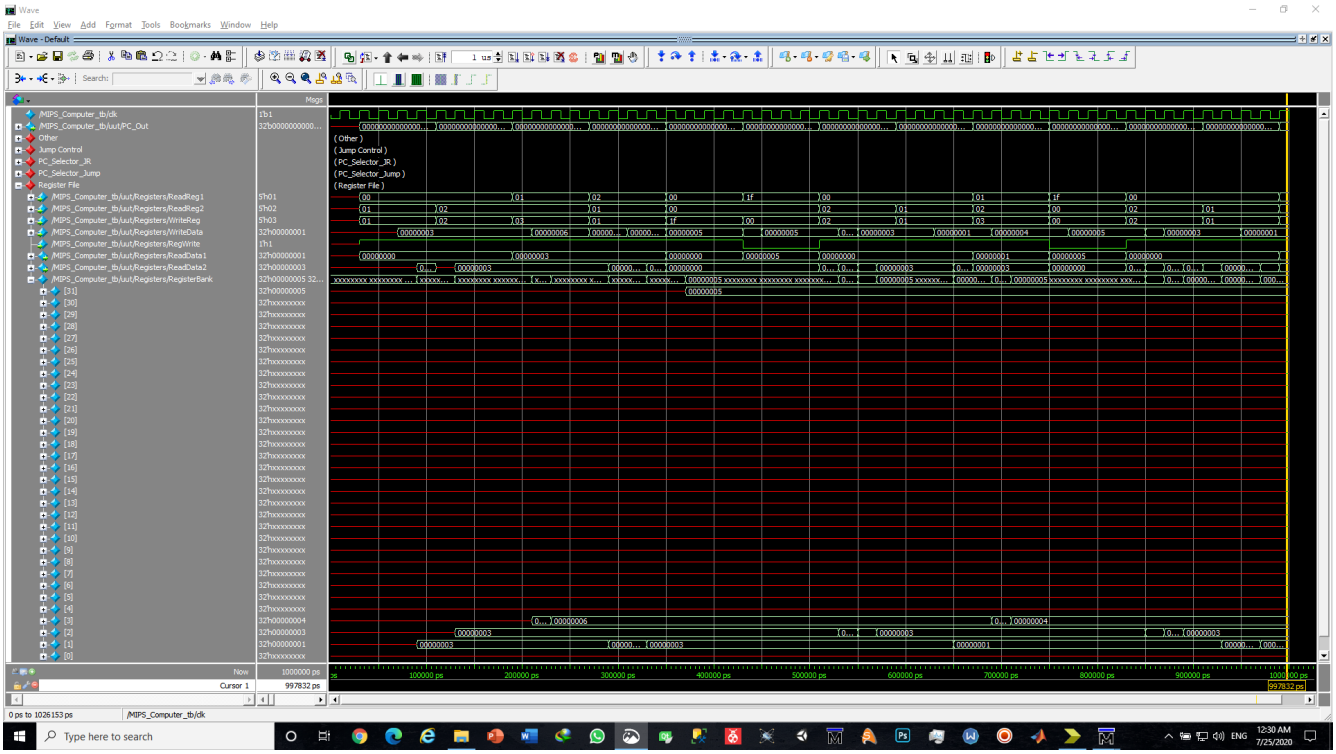
و همانطور که از کد فوق پیداست ، داخل یک لوپ که شامل دستورات خط ۵ تا ۸ است گیر میکنیم و طبیعتا در سیمولیشن باید بعد از اجرای خط ۸ ام ، مجدد ۵ داخل pc لود شود و این روند تکرار شود. که این دقیقا در اسکرین‌شات‌های سیمولیشن هم پیداست . برای بررسی درستی این بخش ، بررسی PC , Register Bank و ماژول‌های جدید کفایت میکند.
به علاوه من دستور jump خالی رو تست نکردم ، زیرا اگر کامپیوتر من بتونه دستور jal رو درست اجرا کنه خیلیییی بدیهیه که دستور jump رو هم میتونه انجام بده و این نیازی به تست نداره (میدانیم که دستور jal یک دستور jump است که قبل از آن باید مقدار PC + 1 را هم ذخیره کند (یعنی عملاً داره آدرس برگشت رو ذخیره میکنه و در زبان اسمبلی به رجیستری که آدرس برگشت رو نگه میداره $ra گویند به معنای return address . که اینجا این رجیستر همان رجیستر شماره $31 است .
عکس های سیمولیشن را در صفحه بعد ببینید (همچنین در پوشه‌ی آپلود شده ، ضمیمه شده است)



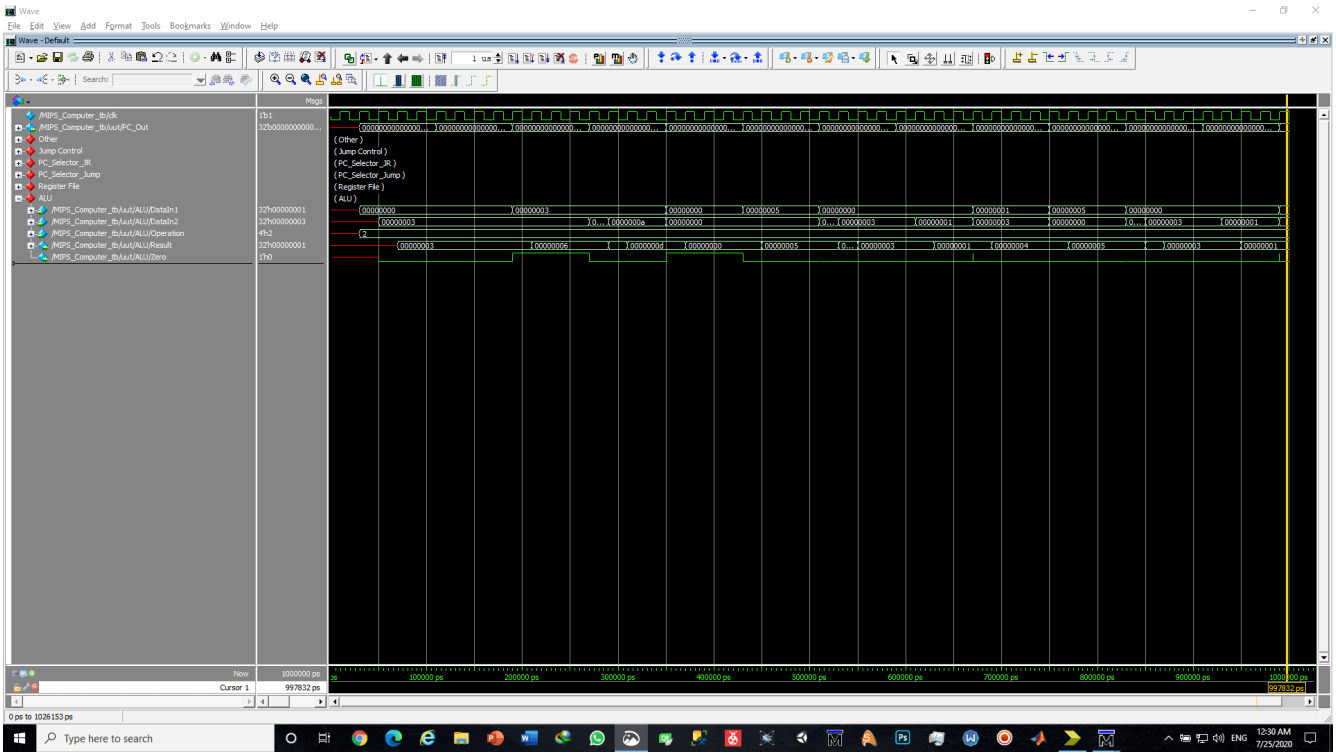
Picture 1 : 3rdPhase_Sim-scr1.png



Picture 2 : 3rdPhase_Sim-scr2.png



Picture 3 : 3rdPhase_Sim-scr3.png



Picture 4 : 3rdPhase_Sim-scr4.png