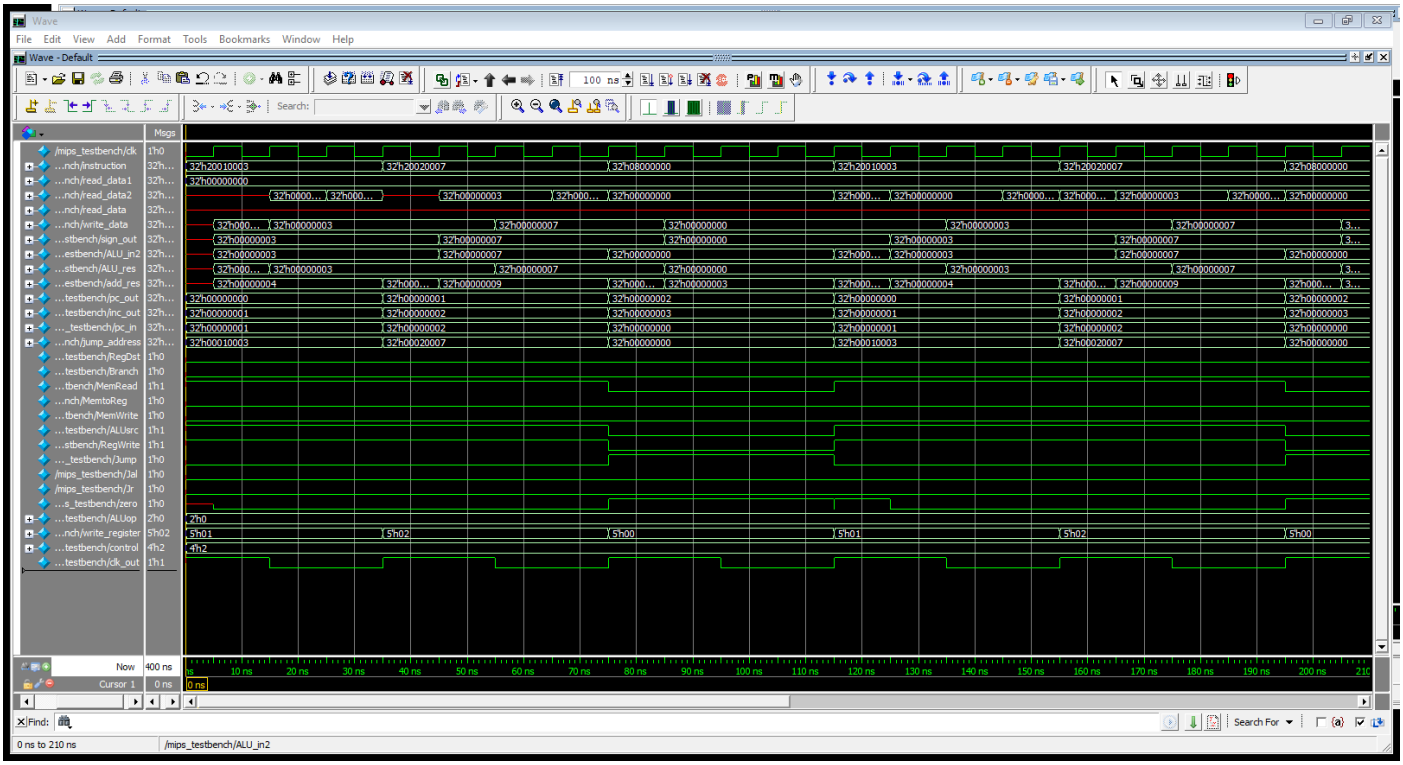
تصویر زیر نتیجه اجرای مجموعه دستورات مقابل می باشد :

assign memory[0] = 32'b001000_00000_00001_0000000000000011 // R1 <- 3

assign memory[1] = 32'b001000_00000_00010_0000000000000111 // R2 <- 7

assign memory[2] = 32'b000010_00000000000000000000000000 // jump to ins[0]



اولین دستور معادل addi R1, 3 دومین دستور معادل addi R2, 7 و سومین دستور معادل j 0 می باشد.

واضح است در اولین دستور حاصل ALU_res پس از ۳ کلاک برابر با ۳ شده لذا در همان کلاک write_data نیز ۳ می شود مقدار write_register نیز از همان ابتدای ورودی دستور برابر ۰۰۰۰۱ یعنی ۱ قرار گرفته لذا در یک کلاک بعد ازاماده شدن حاصل ALU مقدار ۳ بر روی رجیستر ۱ نوشته می شود.

دومین دستور (addi R2, 7) نیز به همین ترتیب اجرا می شود با این تفاوت که حاصل ALU مقدار ۷ شده و write_register نیز مقدار ۰۰۰۱۰ یعنی ۲ را اتخاذ می کند لذا بعد از ۴ کلاک مقدار ۷ بر روی رجیستر ۲ نوشته می شود.

سومین دستور (j 0) است که موجب ورودی مقدار ۰ به pc_in خواهد شد لذا در کلاک بعدی رجیستر pc ، مقدار ۰ وارد آن شده و مجددا دستور اول اجرا می گردد و همین روال تا بی نهایت تکرار خواهد شد.(به این منظور یک مالتی پلکسر بر سر راه pc_in قرار می دهیم که از میان حاصل مالتی پلکسر مقابل ماژول add و آدرس پرش یکی را انتخاب کند select این مالتی پلکسر در این مرحله کنترلjump خواهد بود و هر زمان دستور از نوع jump باشد آدرس پرش به عنوان ورودی بعدی رجیستر pc اتخاذ می شود. و آدرس پرش حاصل کانکت شده ۲۶ بیت آدرس پرش موجود در دستور با ۶ بیت پر ارزش pc + 1 است. نکته حائز اهمیت این است که در مرحله های بعد تغییراتی در این mux صورت می گیرد و select آن jump | jal می شود زیرا برای هندل کردن دستور jal نیز به این mux نیاز داریم)

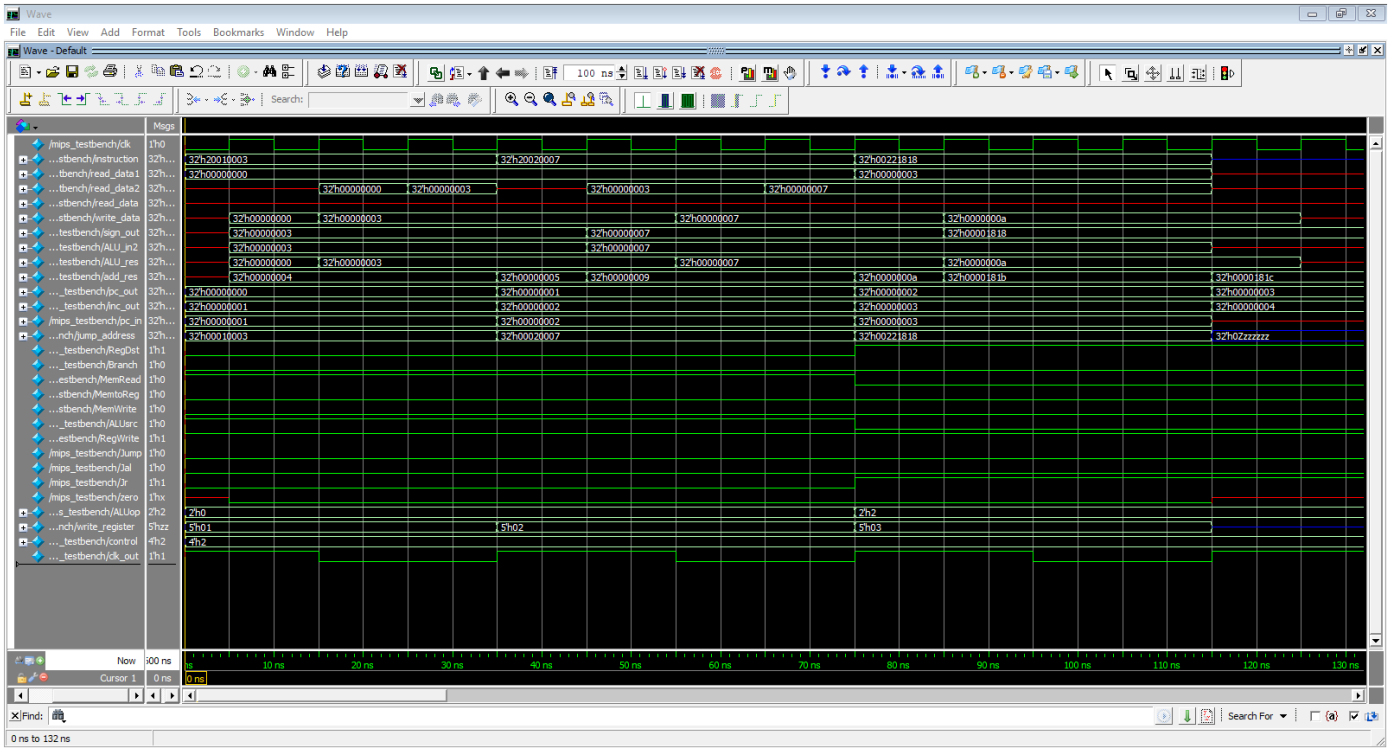
لذا دستورات نوع l_type, jump به درستی عمل می کنند.

تصویر زیر نتیجه اجرای مجموعه دستورات مقابل می باشد :

assign memory[0] = 32'b001000_00000_00001_0000000000000011 // R1 <- 3

assign memory[1] = 32'b001000_00000_00010_0000000000000111 // R2 <- 7

assign memory[2] = 32'b000000_00001_00010_00011_00000_011000 // R3 <- R1+ R2



دستور ۱و ۲ همان دستورات ۱و۲ مثال فوق هستند اما دستور ۳ معادل add R3, R2, R1 می باشد لذا ضمن رسیدن خط اجرای برنامه به دستور سوم مقدار رجیستر های ۱و ۲ از بانک رجیستر خوانده شده و و روی read_data1, read_data2 قرار می گیرند در کلاک بعدی حاصل ALU یعنی ALU_res مقدار R1 + R2 یعنی عدد ۱۰ را تولید می کند و در همان کلاک این مقدار بر روی write_data قرار گرفته و با توجه به اینکه از همان ابتدای ورود به دستور سوم write_register مقدار ۰۰۰۱۱ معادل ۳ را اتخاذ کرده بود ضمن کلاک بعدی مقدار ۱۰ بر روی رجیستر ۳ قرار می گیرد.

لذا دستورات نوع R-type ما نیز به درستی عمل می کند.

تصویر زیر نتیجه اجرای مجموعه دستورات مقابل می باشد :

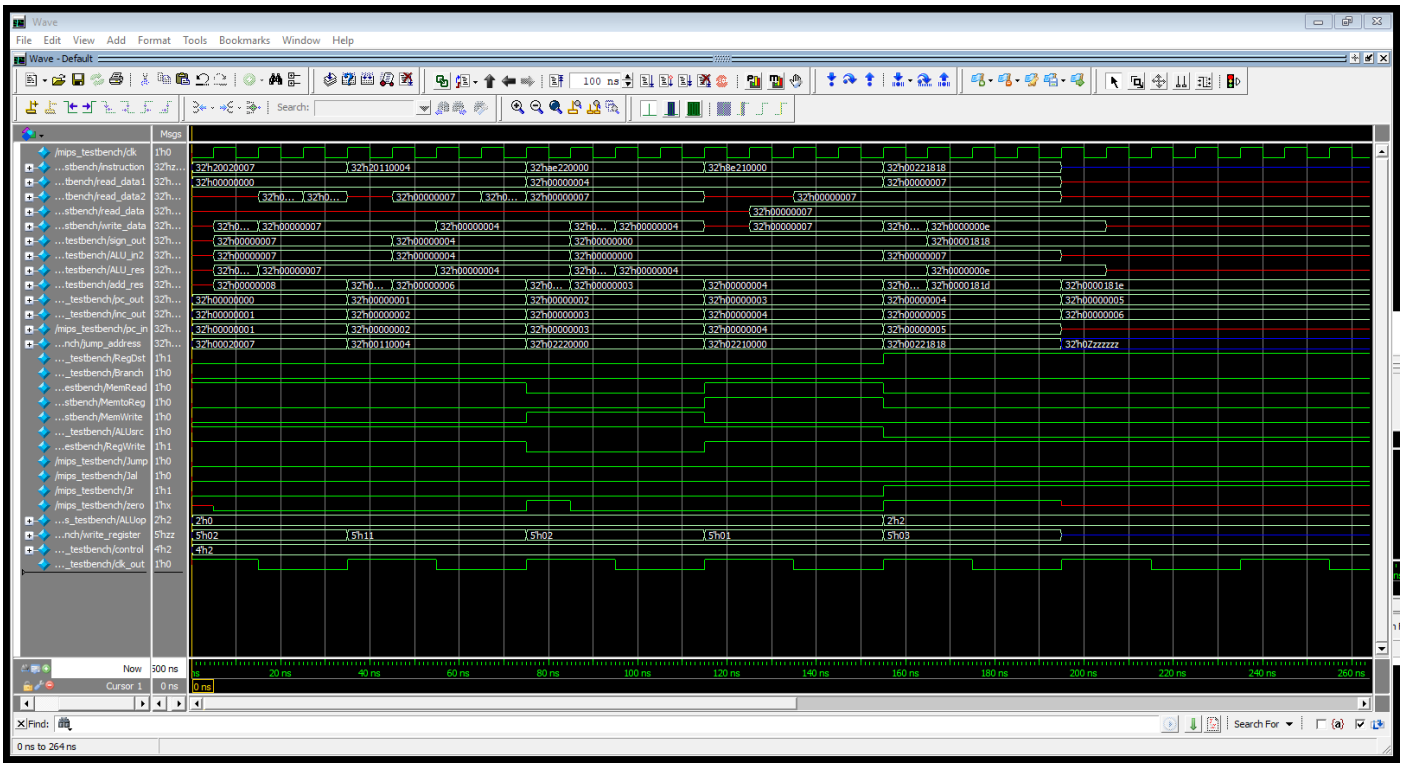
```
assign memory[0] = 32'b001000_00000_00010_0000000000000111; //R2 <- 7

assign memory[1] = 32'b001000_00000_10001_0000000000000100; //R17 <- 4

assign memory[2] = 32'b101011_10001_00010_0000000000000000; //7-> mem[4]

assign memory[3] = 32'b100011_10001_00001_–0000000000000000; //R1 <- mem[4]

assign memory[4] = 32'b000000_00001_00010_00011_00000_011000; //R3 <- R1 + R2
```



دو دستور اول دستورات معادل addi R2, 7 , addi R17, 4 می باشند که در مثال های قبل گام های اجرا این دستورات بررسی شد دستور سوم معادل sw R2, 0(R17) می باشد این دستور موظف است محتویات رجیستر ۲ را در خانه شماره ۴ حافظه ذخیره کند. در کلاک سوم این دستور حاصل ALU محاسبه می شود که درحقیقت همان offset + R17 یعنی ۴ است و این ادرس خانه ای از حافظه است که مقدار رجیستر ۲ باید در آن ذخیره شود به علاوه همانطور که در سخت افزار مشخص است داده ای که باید در حافظه نوشته شود درخروجی read_data2 از بانک رجیستر ظاهر شده و کنترل MemWrite نیز فعال می شود. به این ترتیب دستور save word اجرا می گردد.

در دستور چهارم قصد بررسی یک دستور load word به شکل lw R1, 0(R17) را داریم در این دستور نیز ALU ادرس خانه ای از حافظه که داده باید از آن خوانده شود را محاسبه می کند و سپس دیتا ممووری محتوای آدرس را خوانده و مقدار آن را برروی خروجی read_data قرار می دهد و کنترل MemToReg نیز فعال می گردد این مقدار read_data به ورودی write_data بانک رجیستر متصل شده و لذا در کلاک بعدی این مقدار بر روی رجیستری که آدرس آن در write_register آمده است (همان رجیستر R1) نوشته می شود. (البته در قسمت های بعد سخت افزار برای پیاده سازی بخش های اضافی اطلاح شده و mux های دیگری نیز در این میان قرار می گیرند.)

دستور پنجم نیز همان add R3, R1, R2 است که مقادیر رجیستر های R1, R2 یعنی ۷و ۷ را جمع زده در حاصل ALU به همان شکلی که قبلا اشاره شد در داخل رجیستر ۳ ذخیره می شود.

لذا دستورات sw و lw نیز به درستی عمل می کنند.(واحد کنترل به بررسی دستور sw نپرداخته بود من این دستور را به واحد کنترل اضافه کرده ام.)

تصویر زیر نتیجه اجرای مجموعه دستورات مقابل می باشد :

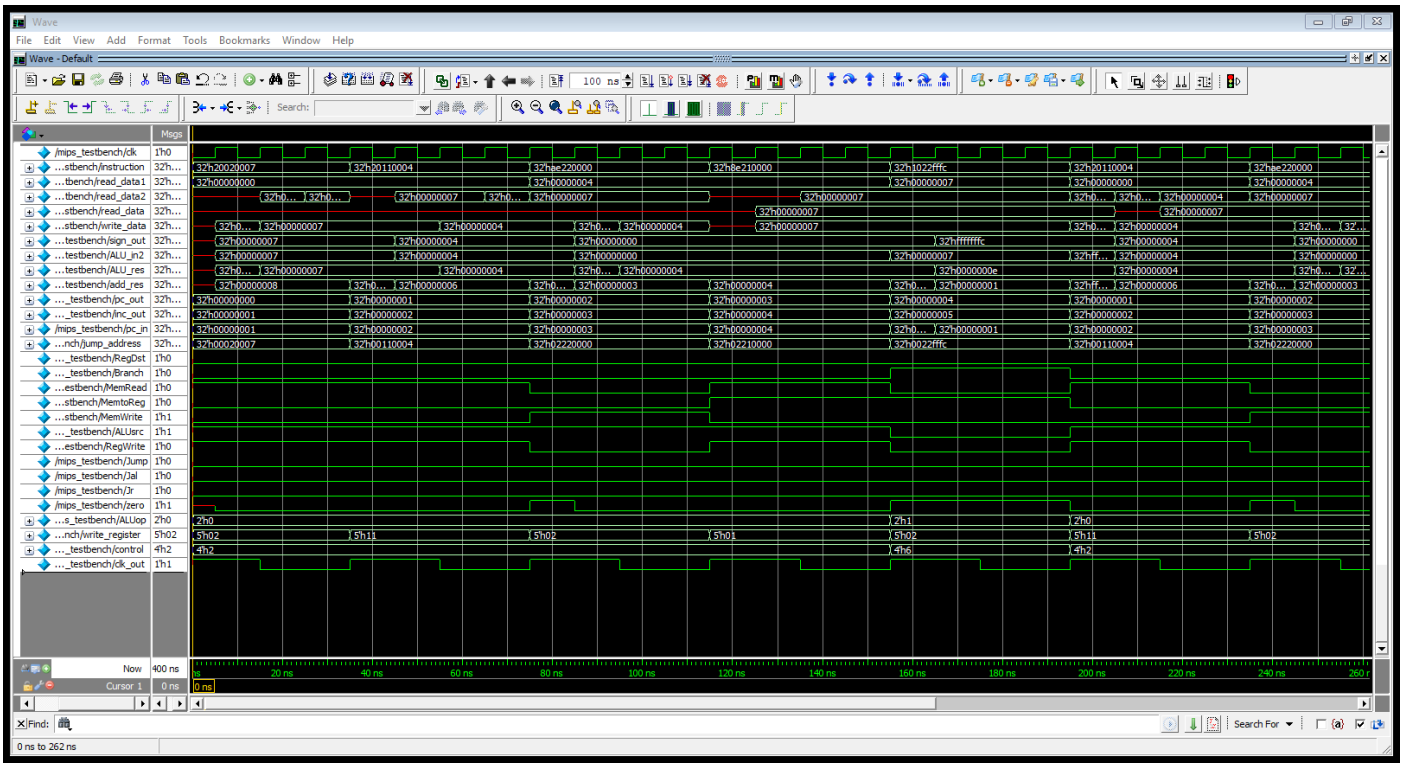
assign memory[0] = 32'b001000_00000_00010_0000000000000111; //R2 <- 7

assign memory[1] = 32'b001000_00000_10001_0000000000000100; //R17 <- 4

assign memory[2] = 32'b101011_10001_00010_0000000000000000; //7-> mem[4]

assign memory[3] = 32'b100011_10001_00001_0000000000000000; //R1 <- mem[4]

assign memory[4] = 32'b000100_00001_00010_1111111111111100; // if R1 = R2 → jump to ins[1]



دستورات اول تا چهارم کاملا مشابه سوال قبل هستند اما دستور پنجم دستور beq R1, R2, -4 می باشد که این دستور مقادیر رجیستر های ۱و ۲ را خوانده و در read_data1, read_data2 که خروجی های بانک رجیستر هستند قرار می دهد و سپس در صورتی که خروجی zero ی ALU برابر ۱ شد و opcode دستور جاری مربوط به انشعاب باشد یعنی کنترل branch فعال باشد آن گاه آدرس pc + 1 جاری یعنی 5 با حاصل sign extend شده آدرس انشعاب یعنی 4- جمع شده و آدرس پرش یعنی 1 تولید می شود که در گام بعدی این آدرس در pc_in قرار می گیرد و منجر به بازگشت به دستور۱ می گردد. و این روال تا ابد تکرار می شود.

لذا دستور branch نیز به درستی عمل می کند.

تصویر زیر نتیجه اجرای مجموعه دستورات مقابل می باشد :

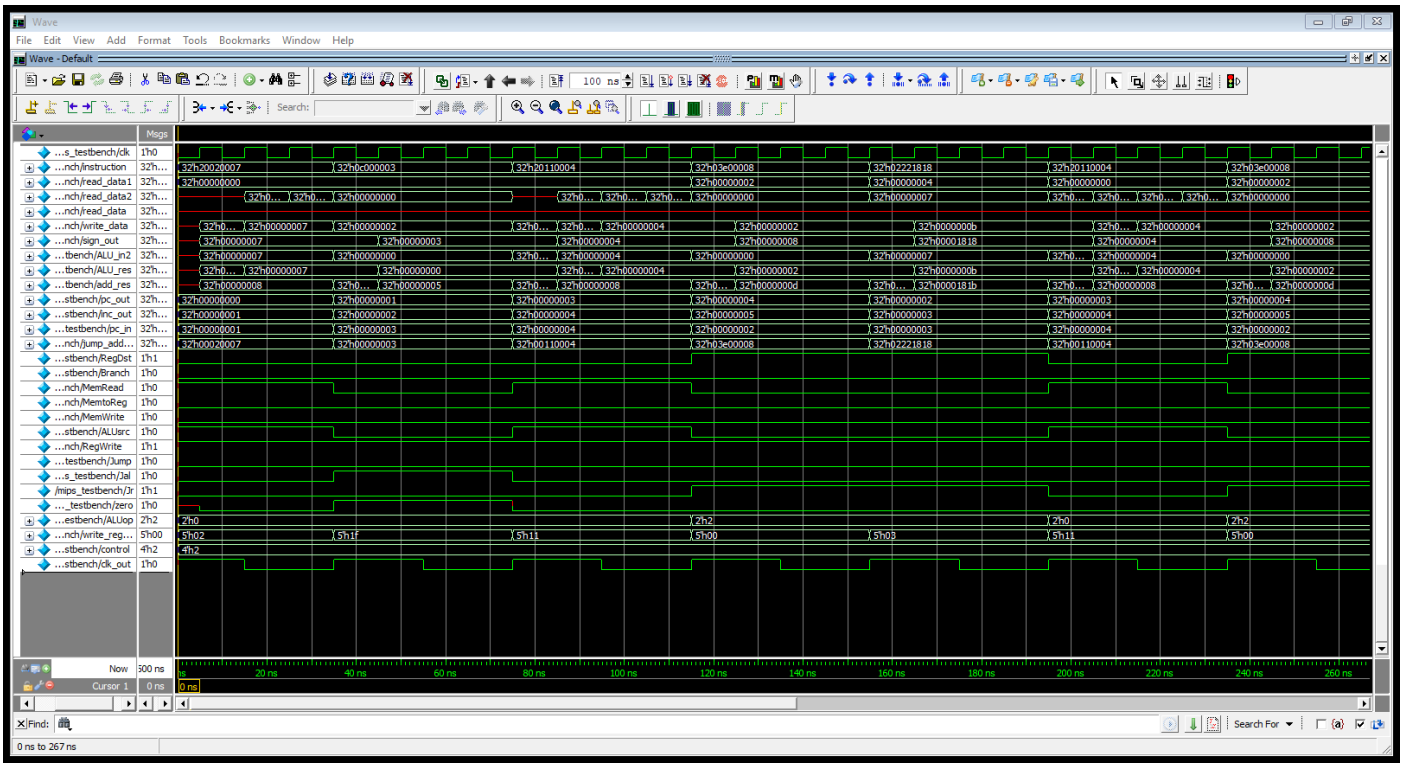
```
assign memory[0] = 32'b001000_00000_00010_0000000000000111; //R2 <- 7

assign memory[1] = 32'b000011_00000000000000000000000011; //jal 3

assign memory[2] = 32'b000000_10001_00010_00011_00000_011000; //R3 <- R17 + R2

assign memory[3] = 32'b001000_00000_10001_0000000000000100; //R17 <- 4

assign memory[4] = 32'b000000_11111_00000_00000_00000_001000; //jr $ra
```

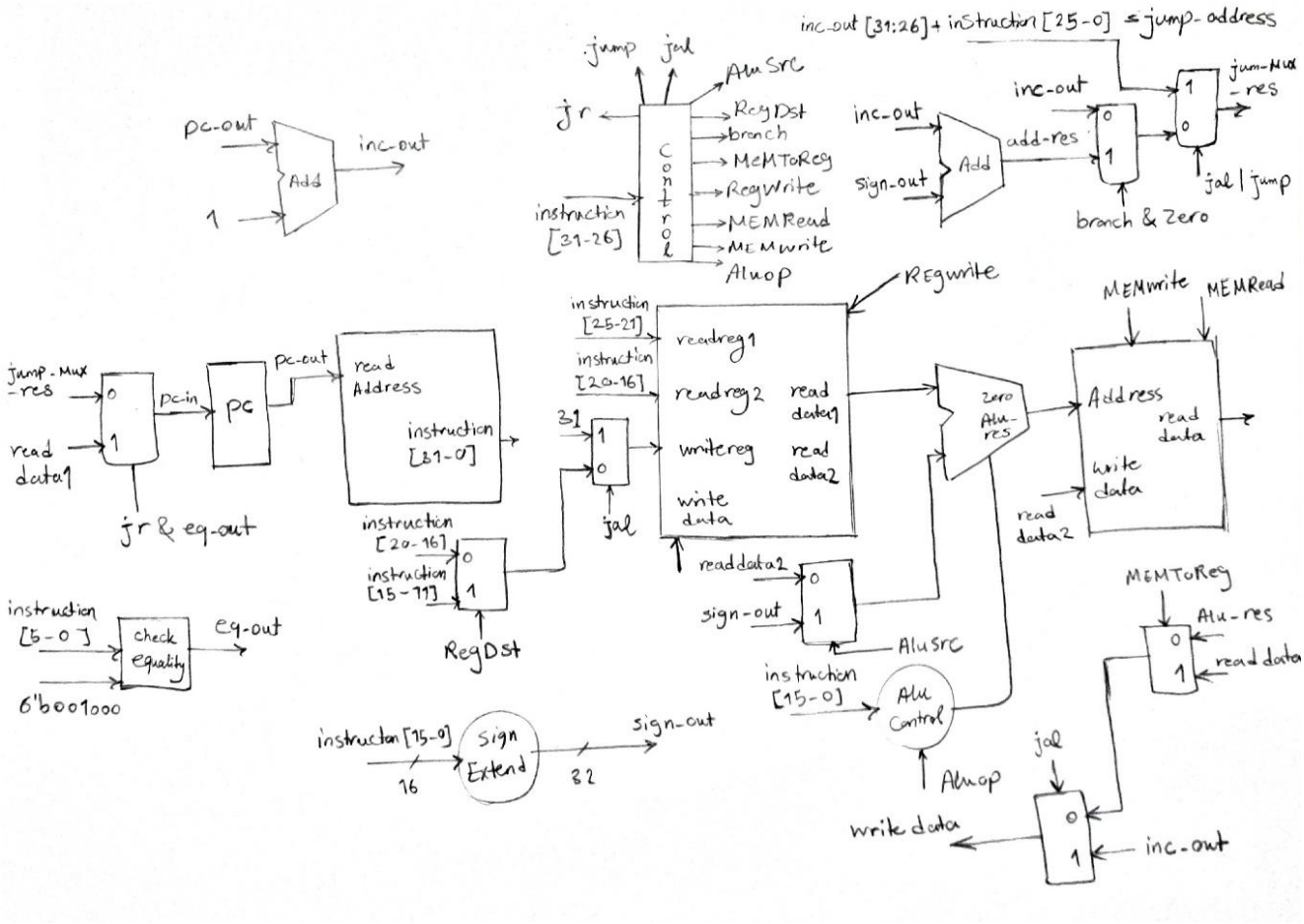


دستور اول مشابه دستورات مثال های قبل است اما دستور دوم دستور 3 jal می باشد که این دستور موجب فعال شدن کنترل jal می شود و مالتی پلکسر قرار گرفته شده بر سر pc_in که از میان آدرس پرش و pc + 1 یکی را به عنوان ورودی pc_in انتخاب می کند اینبار آدرس پرش را برمی گزیند زیرا select این مالتی پلکسر Jal | Jump می باشد و چنانچه دستور از این دو نوع باشد ۲۶ بیت آدرس پرش دستور ، با 6 بیت پرارزش، PC + 1 جاری کانکت شده و به عنوان ورودی pc در کلاک بعدی قرار می گیرد تا به دستور مورد نظر پرش کنیم البته نکته مهم این است که ضمن اجرای این دستور لازم است آدرس بازگشت یعنی Pc + 1 یا همان inc_out در رجیستر $ra که رجیستر با شماره ۳۱ است ذخیره شود به این منظور یک مالتی پلکسر بر سر write_register موجود در رجیستر بانک و یک مالتی پلکسر بر سر راه write_data موجود در رجیستر بانک قرار می گیرد که select هر دو کنترل jal می باشد و اولی از میان رجیستر ۳۱ یا همان $ra و نتیجه مالتی پلکسری که قبلا موجود بود (از میان rd یا rt بنابر نوع دستور یکی را برمیگزید) یکی را بر میگزیند و دومی از میان inc_out و حاصل مالتی پلکسر موجود در خروجی data_memory یکی را بر می گزیند.

پس از اجرای jal 3 دستور سوم (دستور شماره ۲) دیگر اجرا نمی شود و مستقیما به دستور چهارم (دستور شماره ۳) پرش می کنیم این دستور مقدار ۴ را دررجیستر ۱۷ قرار می دهد و مراحل آن قبلا بازگو شده پس به آن نمیپردازیم در کلاک بعدی به دستور شماره ۴ می رویم این دستور Jr $ra است یعنی آدرس بازگشتی که موقع jal در رجیستر شماره ۳۱ قرار داده شده بود را از آن میخوانیم و ورودی pc را برای کلاک بعدی مساوی این مقدار بازگشت قرار می دهیم. ضمن فراخوانی jal 3 مقدار inc_out که همان ۲ بود در ra نوشته شده بود اینبار مقدار ۲ از ra خوانده شده و در pc قرار می گیرد لذا در کلاک بعدی به دستور شماره ۲ پرش می کنیم که این دستور مقادیر رجیستر های ۲ و ۱۷ را جمع کرده و در رجیستر شماره ۳ قرار می دهد. سپس مجددا به خط برنامه بعدی می آید و زمانی که دوباره به jr $ra می رسد مجددا به دستور شماره ۲ پرش می کند و این روال تکرار دستورات شماره ۲ تا شماره ۴ دائما تکرار می شود زیرا آدرس موجود در ra تغییری نکرده است. پس دستورات jal, jr نیز به درستی عمل می کنند.

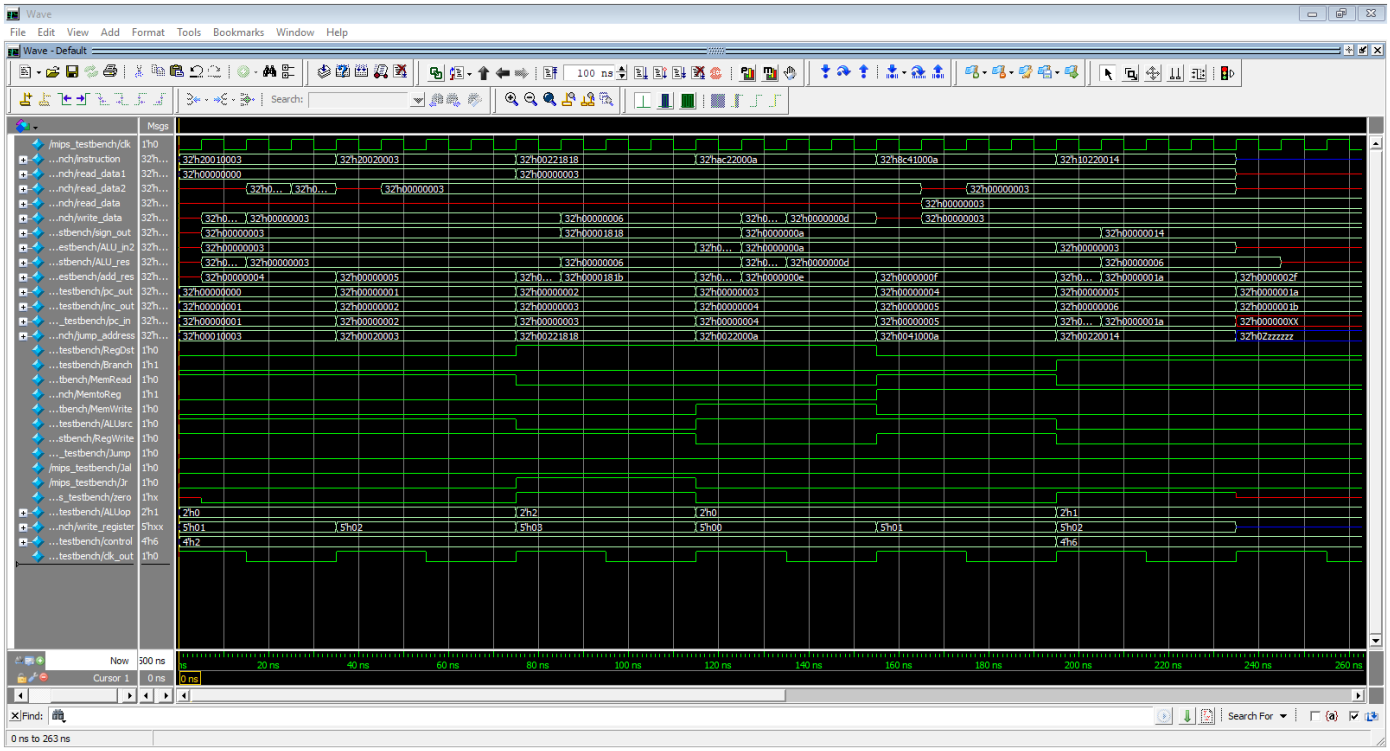
نکات انتهایی : در این پروژه دستورات sw, j, jal, jr نیز افزوده شده و متناسب با آن ها بخش هایی در واحد control نیز تغییر یافته است.

سخت افزار نهایی پیاده سازی شده مطابق زیر می باشد:



تصاویر فاز دوم پروژه :

```
assign memory[0] = 32'b001000_00000_00001_0000000000000011 // addi R1, R0, 3
assign memory[1] = 32'b001000_00000_00010_0000000000000011 // addi R2, R0, 3
assign memory[2] = 32'b000000_00001_00010_00011_00000_011000 // add R3, R1, R2
assign memory[3] = 32'b101011_00001_00010_0000000000001010 // sw R2, 10(R1)
assign memory[4] = 32'b100011_00010_00001_0000000000001010 // lw R1, 10(R2)
assign memory[5] = 32'b000100_00001_00010_0000000000010100 // beq R1, R2, 20
```



ابتدا دستور شماره 0 وارد شده این دستور از نوع i-type است که قصد دارد مقدار ۳ را بر روی رجیستر شماره ۱ بریزد. محاسبات لازم توسط ALU در ۳ کلاک بعد از ورود instruction نهایی می شود و ALU_res مقدار داده ای که قرار است بر روی write-register ریخته شود بر روی write_data قرار می دهد و در یک کلاک بعد از قرار گیری داده در write_data این مقدار بر روی رجیستری که آدرس آن در write-register قرار دارد یعنی همان رجیستر ۱ ریخته می شود. به علاوه در این دستور کنترل RegWrite نیز فعال می شود زیرا قصد دارید داده ای را در یک رجیستر بنویسیم. دستور شماره ۱ نیز به طور مشابه مقدار ۳ را بر روی رجیستر شماره ۲ قرار می دهد. دستتور شماره ۲ قصد دارد مقدار رجیستر های ۱ و ۲ را جمع زده و حاصل را در رجیستر ۳ ذخیره کند به این منظور بلافاصله پس از load شدن instruction آدرس ۱ و ۲ در read-register1, read-register2 قرار می گیرد و بانک رجیسر مقادیر رجیستر ۱ و۲ را بر روی خروجی های read-data1 , read-data2 قرار می دهد. این مقادیر وارد ALU شده و در کلاک بالا رونده بعدی در خروجی ALU-res نتیجه جمع یعنی مقدار ۶ ظاهر می شود. به علاوه خروجی zero مربوط به ALU نیز فعال می شود زیرا مقادیر ۲ ورودی باهم برابر است. بلافاصله پس از آماده شدن ALU-res این مقدار بر روی write-data قرار می گیرد و مقدار write-register نیز ۳ است یعنی انتظار داریم در کلاک بعدی مقدار ۶ بر روی رجیستر شماره ۳ نوشته شود.

دستور شماره ۳ دستور sw است که این مورد جداگانه به سخت افزار اضافه شده این دستور قصد دارد محتوای رجیستر 2 را در خانه شماره ۱۳ حافظه قرار دهد به عبارتی دستور بدین شکل است sw R2, 10(R1) که خانه حافظه همان R1 + 10 یعنی ۱۳ است این آدرس توسط ALU محاسبه شده و در ALU_res قرار می گیرد و سپس وارد address دیتا مموری می شود به علاوه read-data2 که محتویات رجیستر شماره 2 است در ورودی write-data دیتاموری قرار می گیرد و کنترل MemWrite نیز فعال می شود به این ترتیب در کلاک بعد مقدار ۳ در خانه شماره ۱۳ حافظه نوشته می شود. دستور شماره ۴ قصد دارد مقدار خانه شماره ۱۳ را از حافظه بخواند و آن را در رجیستر شماره 1 قرار دهد به عبارتی دستور بدین شکل است lw R1, 10(R2) مشابه بالا آدرس حافظه توسط ALU محاسبه شده و سپس در address قرار می گیرد و داده مربوطه از حافظه خوانده شده و از آن جایی که کنترل MemToReg فعال است

مالتی پلکسر آن را به ورودی write-data بانک رجیستر انتخاب می کند و در write-register نیز ادرس رجیستر مقصد یعنی R1 قرار گرفته به این ترتیب ضمن رسیدن کلاک بعدی مقدار ۳ از حافظه در داخل رجیستر شماره۱ قرار میگیرد.

دستور شماره ۵ مقادیر دو رجیستر R1, R2 را از بانک رجیستر خوانده وارد ALU کرده و مقایسه می کند و در صورت برابری کنترل zero مربوط به ALU فعال شده و از انجایی که دستور beq است کنترل branch فعال می شود پس مالتی پلکسر حاصل sign extend شده آدرس پرش + PC + 1 را به عنوان ورودی بعدی pc گزینش می کند آدرس پرش دستور ۲۰ و مقدار pc+1 همان ۶ است پس انتظار می رود در کلاک بعدی pc به خانه شماره ۲۶ پرش کنیم. (دستور شماره ۳ در فاز دوم نبوده و خودم آن را برای ذخیره داده در مموری به جایی دستی ذخیره کردن اضافه کرده ام به همین سبب تعداد دستورات یکی بیشتر شده و pc مرحله آخر ۲۶ می شود اگر این دستور نبود pc در مرحله اخر مقدار ۲۵ را اتخاذ می کرد)