

Digital Systems 2 – Report

Anita Ghandehari 810195533

Soheil Shirvani 810195416



فهرست مطالب:

**توضیح آزمایش2**

مسیر داده در پردازنده3

مشخصات پردازنده4

مجموعه دستورات پردازنده5

انواع ثبات ها در پردازنده7

**مرحله وا کشی دستور Instruction Fetch8**

**مرحله کد گشایی Instruction Decode9**

کنترلر10

Condition Control11

**مرحله اجرا Execute13**

**مرحله حافظه Memory16**

**مرحله باز نشانی Write Back17**

**واحد Status Register 18**

**واحد تشخیص هازارد Hazard Detection 19**

**واحد انجام رو به جلو Forwarding 21**

**نتایج آزمایش22**

شبیه سازی در نرم افزار Modelsim23

شبیه سازی در نرم افزار Quartus25

مقایسه دو حالت مدار27

**برخی از مشکلات پیش آمده28**

در این آزمایش ما سعی داریم تا یک پردازنده ی آرم را با کد وریلاک پیاده سازی کنیم و آن را تست و سیمولیت کنیم.

این پردازنده یک پایپلاین 5 استیج ای است که انها به ترتیب شامل:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute Unit (EXE)
4. Memory Unit (MEM)
5. Write Back Unit (WB)

است که هر کدام از این استیج ها دارای یک قسمت شامل ثبات های بعد از آن نیز هست و دارای 3 ماژول مجزا شامل:

1. Hazard Unit
2. Status Register Unit
3. Forwarding Unit

و همین طور دارای یک Controller که در استیج ID قرار دارد و یک Datapath کامل است.

در شکل های زیر می توانیم Datapath کلی مدار را نیز مشاهده کنیم:

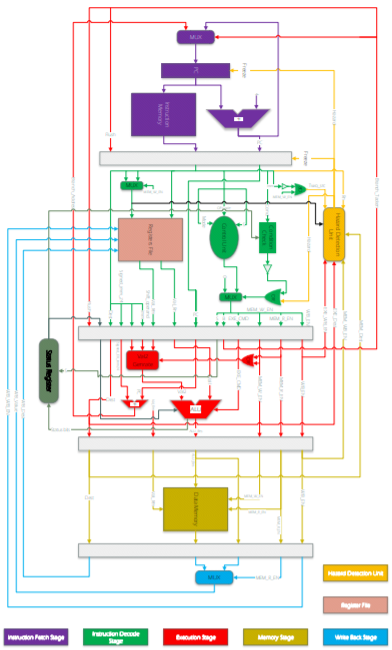


Figure 1: شکل Datapath مدار بدون forwarding unit

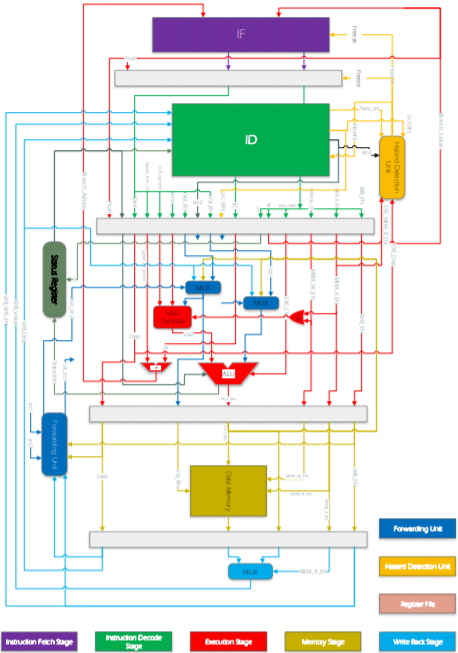


Figure 2: شکل Datapath مدار با forwarding unit

حال برای هر کدام از واحد ها به اختصاص توضیح مربوطه را می دهیم.

و از مشخصات دیگر پردازنده میتوان:



نام برد. و مجموعه دستورات آن:

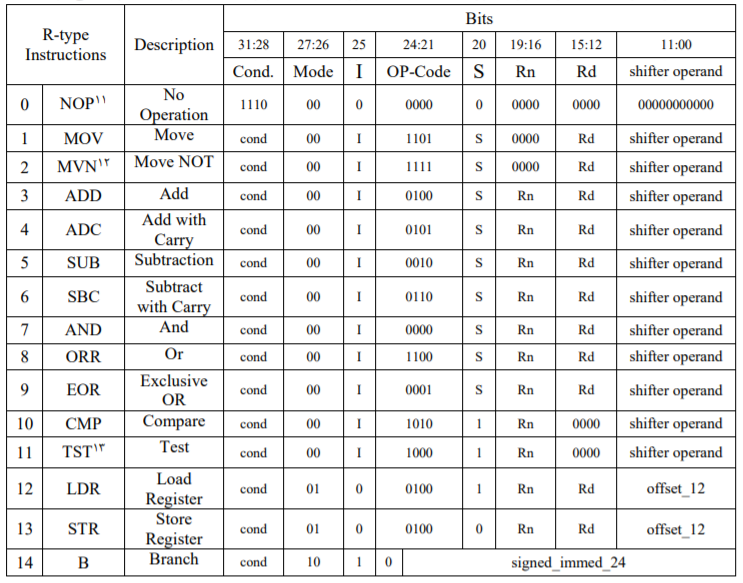
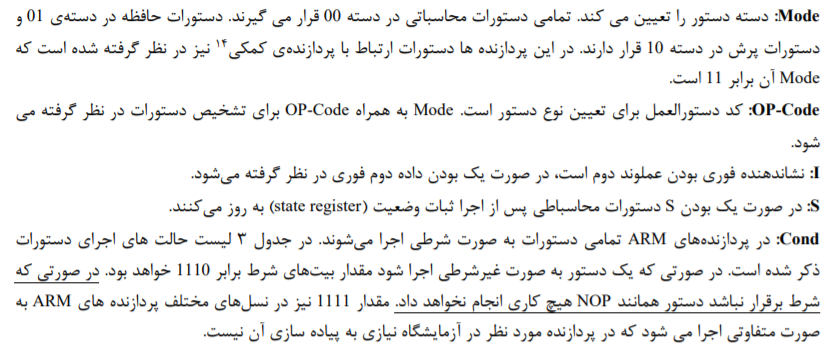


Figure 3: لیست دستورات

که در آن :



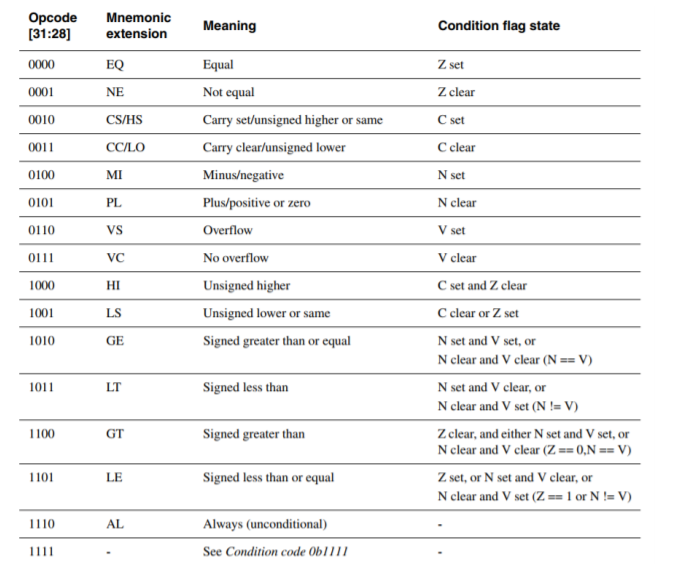
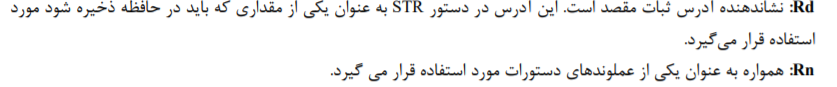


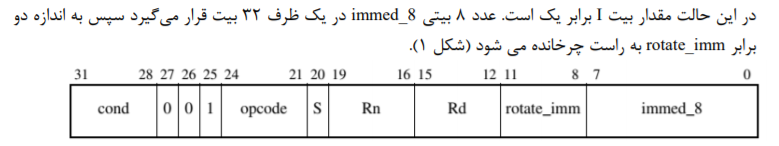
Figure 4: کد دستورات شرطی

که در آن:

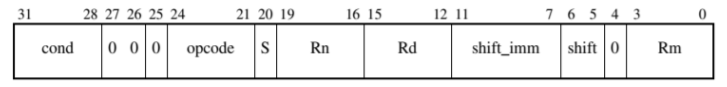
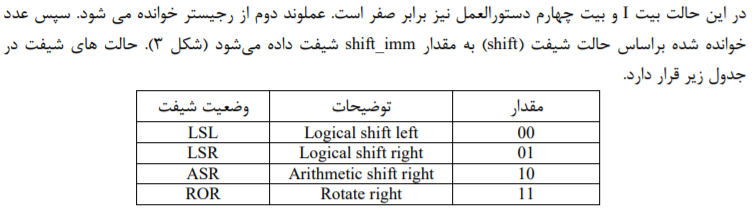


و برای shifter operand 3 مدل زیر را داریم:

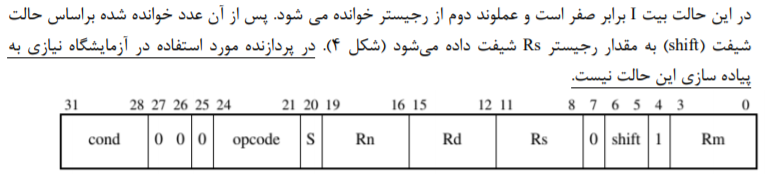
1. 32 بیت شیفت عدد فوری (32-bit immediate):



1. شیفت فوری (immediate shift):

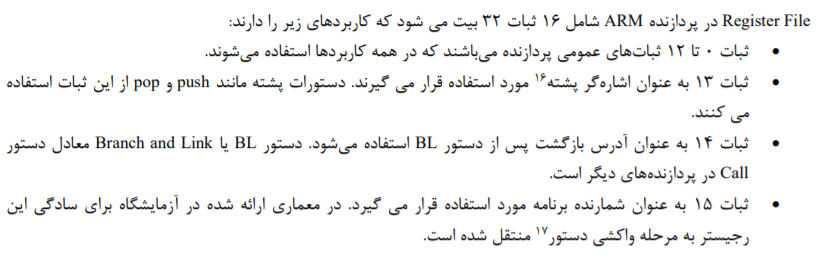


1. شیفت ثباتی (Register Shift):

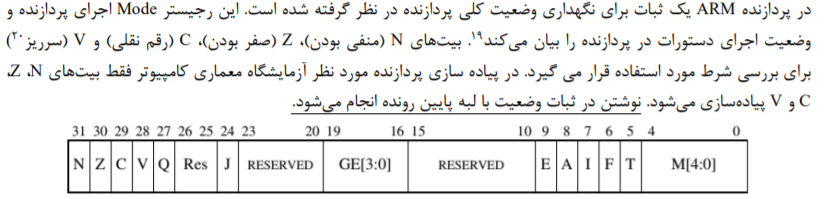


و همین طور پردازنده ی آرم دارای دو نوع ثبات است که شامل:

1. ثبات های عمومی:



1. ثبات وضعیت:

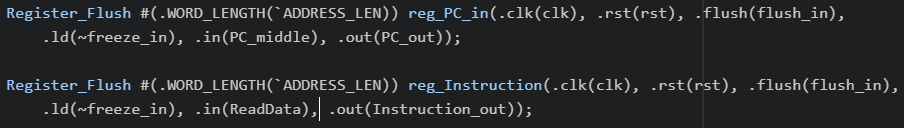


حال برای هر قسمت از مدار توضیحات مربوطه به همراه کد آن داده می شود.

این واحد وظیفه ی تولید PC که نشان دهنده ی شماره ی دستور ما و دادن دستور به بعد به مدار را دارد. این واحد دارای یک ثبات برای PC است که بعد از اجرای هر دستور در صورت نبود پرش، PC = PC + 4 می شود تا بتواند دستور بعد را اجرا کند و اگر دستور پرش در حال اجرا باشد مقدار Branch Address درون آن قرار می گیرد تا به دستور مربوطه پرش کند. سپس این مقدار وارد یک حافظه دستور می شود که تمامی دستورات در آن قرار می گیرند. آدرس این حافظه در واقع همان مقدار PC ما است. برای ایجاد PC + 4 نیاز به یک جمع کننده بدون کلاک و برای تعیین پرش بودن یا نبودن ( برای انتخاب مقدار PC برای هر دستور) نیاز به یک مولتی پلکسر داریم که مقدار انتخاب آن از Controller می آید. همچنین ثبات PC ما یک دستور Freeze نیز دارد تا در صورت نیاز بتواند PC را تغییر ندهد و در نتیجه مدار را Stall کند تا دستورات درون پایپ با اتمام برسند و مقادیر مورد نیاز در دستور بعد از دستور قبل محاسبه شوند. در حالت اول می توانیم مقادیر انتخاب مولتی پلکسر (Branch\_Taken)، Freeze را صفر قرار دهیم و دستوراتی را از حافظه دستورات خوانده و اجرا کنیم. در این صورت توانسته ایم این واحد را تست کنیم. این قسمت مقدار PC و همین طور 32 بیت دستور خوانده شده از حافظه دستورات را به استیچ بعد خروجی می دهد و درحالی که ورودی های آن سیگنال های Branch\_Taken، Freeze و Branch\_Address است که از مرحله EXE می آیند و بعدا توضیح داده خواهند شد. کد های مربوط به آن:

|  |  |
| --- | --- |
| Figure 5: کد مربوط به جمع کننده برای PC | Figure 6: کد مربوط به ثبات PC |
| Figure 7: کد مربوط به انتخاب PC یا آدرس پرش | **Figure 8: کد مربوط به حافظه دستورات ها** |

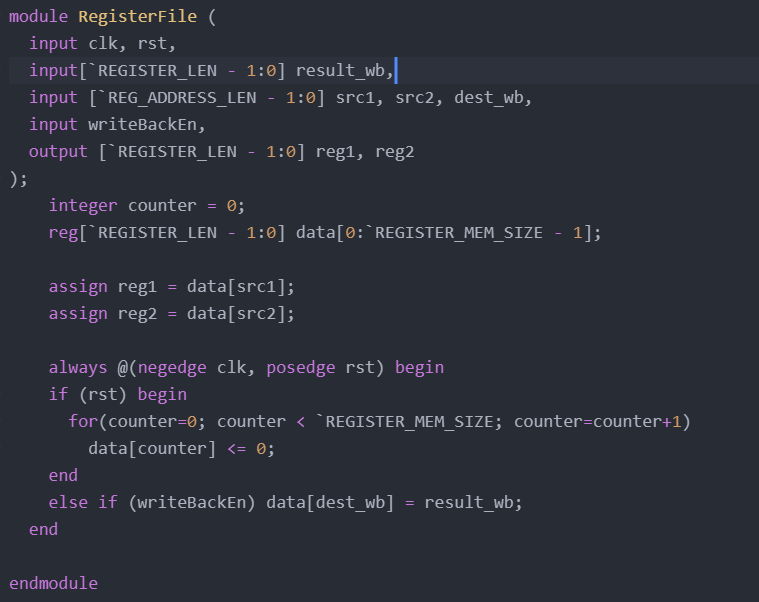
که این قسمت شامل ثبات های بعد از آن یعنی IF\_Reg نیز می شود که در واقع خروجی همین استیج یعنی دستور اولیه Instruction و مقدار PC است (یک سیگنال Flush) نیز از این استیج عبور می کند که در قسمت بعدی توضیح داده خواهید شد که کد آن به شکل زیر می شود:



**Figure 9: کد مربوط به قسمت IF\_REG**

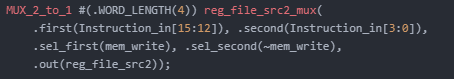
این واحد وظیفه دریافت دستور از مرحله IF دیکود کردن دستور را دارد. در این مرحله سیگنال های کنترلی تولید و مقادیر رجیستر ها خوانده می شوند. لازم به ذکر است در این مرحله دستور به طور کامل کدگشایی می شود و دیگری نیاز به Op-code نیست.

در بخش اول پیاده سازی این بخش ابتدا یک Register File را پیاده سازی کردیم. در این Register File 15 Register 32 بیتی وجود دارند. این Register File دارای دو پورت خواندن ناهمگام و یک پورت نوشتن همگام با لبه پایین رونده clock می باشد. شیوه پیاده سازی Register File در شکل زیر آورده شده است.

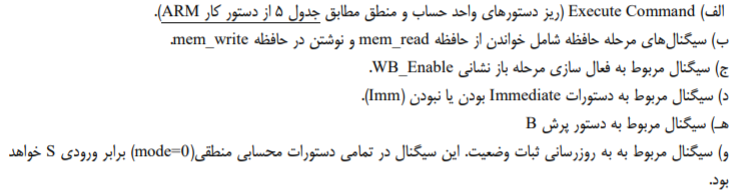


**Figure10: کد مربوط به قسمت Register File**

همچنین یک مالتی پلکسر در ورودی Register File وجود دارد که از بین بیت های (0-4) یا (12-15) ورودی با توجه به سیگنال mem\_write یکی را به عنوان آدرس دوم به Register File می دهد. کد این بخش در زیر آمده است:

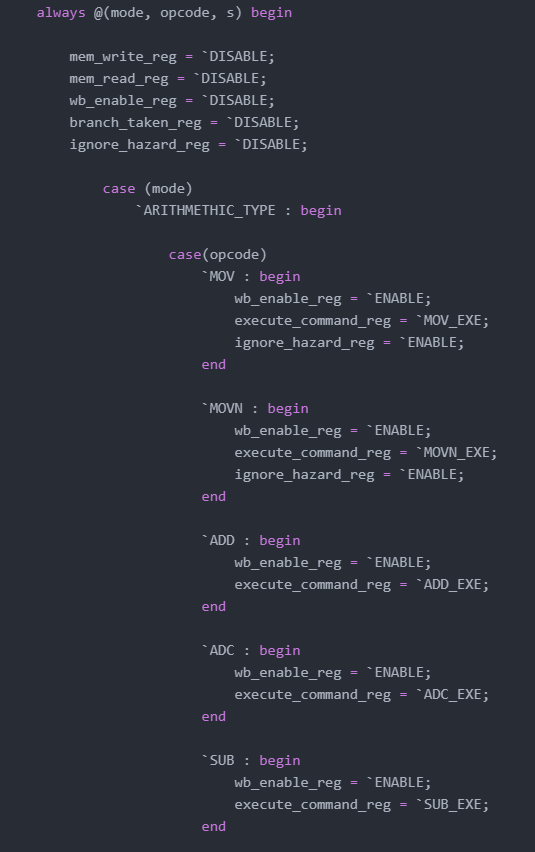


در بخش بعدی واحد کنترلر را پیاده سازی کردیم. این کنترلر با توجه به Opcode, Mode, S سیگنال های کنترلی مورد نیاز برای همه بخش ها پردازنده را تولید می کند. سیگنال های تولیدی به شرح زیر است:



با توجه به موارد بالا، مشخص است که دستور به طور کامل دیکود می شود و در قسمت های بعدی از سیگنال های کنترلی تولید شده استفاده می شود و دستور در بخش های بعدی وارد نشده است.

در شکل زیر بخشی از پیاده سازی Control Unit را مشاهده می کنید:

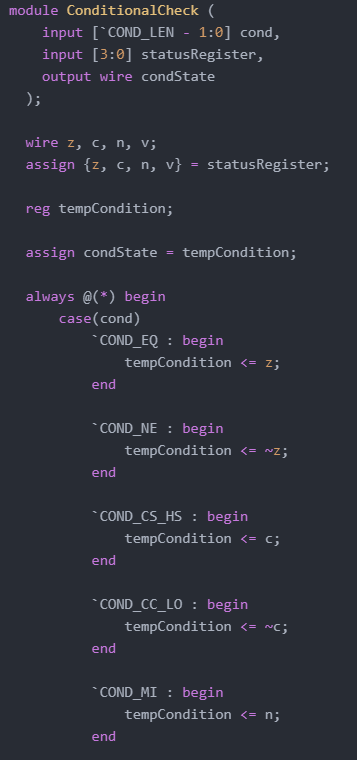
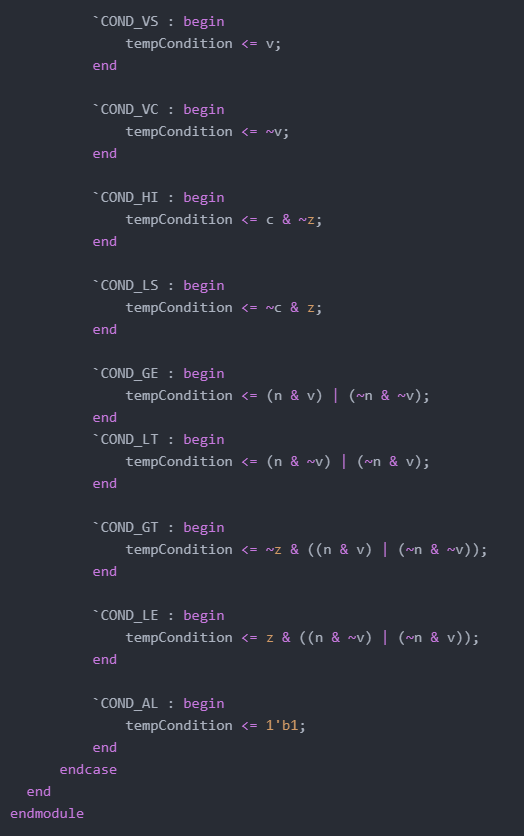


**Figure11: بخشی از کد مربوط به قسمت Control Unit**

همان طور که در پیاده سازی نیز مشخص است که با توجه به opcode و mode هرکدام از سیگنال های کنترلی مربوط به همان دستور به خصوص تولید می شود.

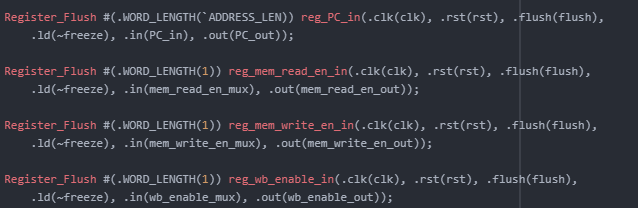
در بخش بعدی به پیاده سازی بخش Condition check پرداخته شده است. در این بخش ابتدا بیت های بخش cond از دستور ورودی بررسی می شود و با استفاده از Status Register برقراری شرط مورد نظر بررسی می شود. در شکل زیر بخشی از پیاده سازی مربوط به این بخش را مشاهده می کنید.

همچنین یک مالتی پلکسر با توجه به مقدار OR دو سیگنال Hazard و notخروجی condition check یا 10 داده خروجی را به رجیستر پایپ لاین منتقل می کند یا 10 بیت صفر.



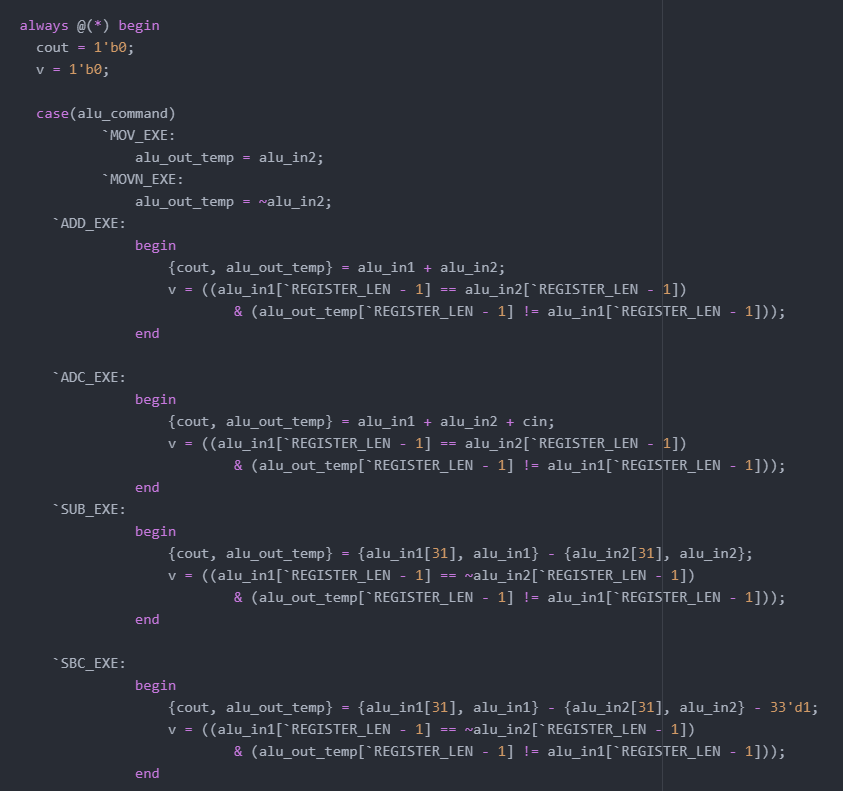
**Figure12: بخشی از کد مربوط به قسمتCondition Check**

تمامی سیگنال ها و داده های خروجی از استیج ID مطابق با Figure1 وارد رجیستر های مربوط به پایپ لاین بین دو استیج ID و EXE می شوند. بخش از کد این بخش در زیر آمده است:



همچنین لازم به ذکر است در این بخش در صورت فعال بودن سیگنال Flush دستورات در رجیستر ورودی به مرحله بعد flush می شوند. این سیگنال توسط کنترلر تولید می شود و برای جلوگیری از اجرای ترتیب نادرست از دستورات با توجه به دستور Branch، رجیستر های خروجی استیج های IFو ID به طور کامل flush می شوند.

اجرای تمامی دستورات منطقی در این بخش از پردازنده اجرا می شود. به علاوه آدرس درست پرش، آدرس ذخیره یا خواندن دستورات از حافظه داده نیز در بخش انجام می شود. یکی از مهم ترین بخش ها در این قسمت واحد ALU میباشد. این واحد وظیفه اصلی محاسباتی در این بخش را بر عهده دارد. این ماژول دارای دو ورودی داده و یک خروجی داده می باشد. همچنین 4 بیت(Execute Command) تولید شده در کنترلر در بخش ID برای تعیین کردن عملیات ALU وارد آن می شوند. همچنین بیت C از status register نیز به عنوان ورودی به ALU داده می شود. در شکل زیر بخشی از پیاده سازی این ماژول را میبینید:



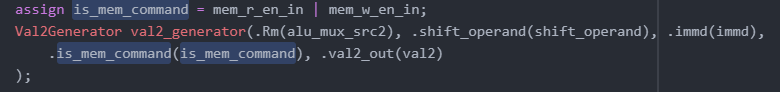
**Figure13: بخشی از کد مربوط به قسمتALU**

علاوه بر ALU نقش ماژول Val2 Generator نیز بسیار حائز اهمیت می باشد. این ماژول وظیفه تولید ورودی دوم ALU را دارد. این ماژول با توجه به سیگنال های تولیدی در استیج ID ورودی دوم مورد نیاز ALU را تولید می کند. در این ماژول در دستورات مختلف عملکردی به شرح زیر دارد:

* در دستورات LDR و STR برابر بت 12 بیت offset می باشد، که از واحد ID وارد واحد EXE می شود
* در دستورات 32 بیت عدد فوری بر اساس براساس shift-operand داده immediate را تولید می کند.
* در دستورات شیفت فوری داده ورودی Rm را شیفت می دهد.

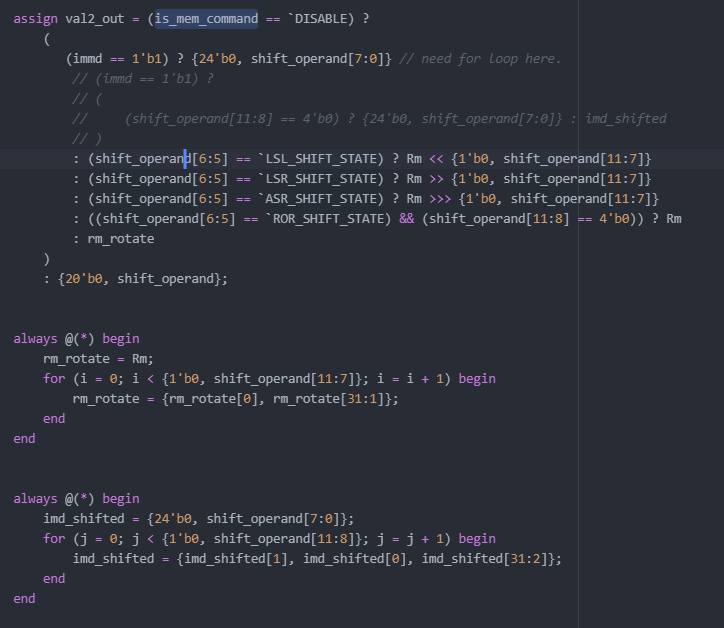
این ماژول 3 ورودی: Rm, shift-operand , immd, is\_mem\_command را دریافت میکند و با توجه به مقدار is\_mem\_command که از OR شدن دو سیگنال mem\_w \_en\_in و mem\_r \_en\_in که همان سیگنال های mem\_w\_en و mem\_r\_en وارد شده به واحد EXE از واحد ID میباشند، خروجی متناظر را با توجه به دستورات ذکر شده در بالا تولید میکند.

کد این بخش از واحد EXE را در شکل زیر مشاهده می کنید:



**Figure14: کد مربوط به قسمت تولید ورودی دوم ALU**

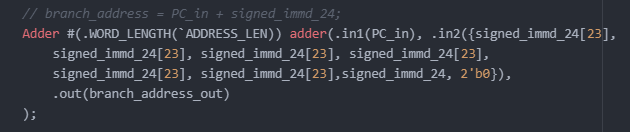
همچنین در شکل زیر کد ماژول Val2 Generator را مشاهده می کنید:



**Figure15: کد مربوط Val2 Generator**

همچنین در این قسمت یک Adder پیاده سازی شده است که وظیفه آن جمع مقدار PC با مقدار signed\_immd\_24 برای تولید آدرس پرش در دستورات Branch می باشد.

کد مربوط به این بخش را در شکل زیر مشاهده می کنید:



**Figure16: کد مربوط به Adderدر بخش EXE**

علاوه بر موارد ذکر شده دو مالتی پلکسر برای بخش forwarding نیز پیاده شده است که توضیحات مربوط به آن ها در قسمت forwarding موجود می باشد.

همچنین همانند قسمت های گذشته تمامی خروجی های این قسمت مطابق با Figure 1 در رجیسترهای بین دو استیج دخیره می شوند و با لبه بالا روند کلاک وارد استیج بعدی می شوند.

در این قسمت حافظه داده ها در پردازنده پیاده سازی شده است. این حافظه با دریافت سیگنال های mem\_r\_en\_in و mem\_w\_en\_in از حافظه داده می خواند یا می نویسد. این حافظه از آدرس 1024 شروع شده و در هر مرحله 32 بیت داده یا 4 بایت داده خوانده یا نوشته می شود. همچنین در این حافظه امکان دسترسی به تک بایت وجود ندارد. حجم این حافظه 256 بایت و امکان خواندن و نوشتن تنها در آدرس ها با مضارب 4 وجود دارد. در این حافظه 4 ورودی دارد:

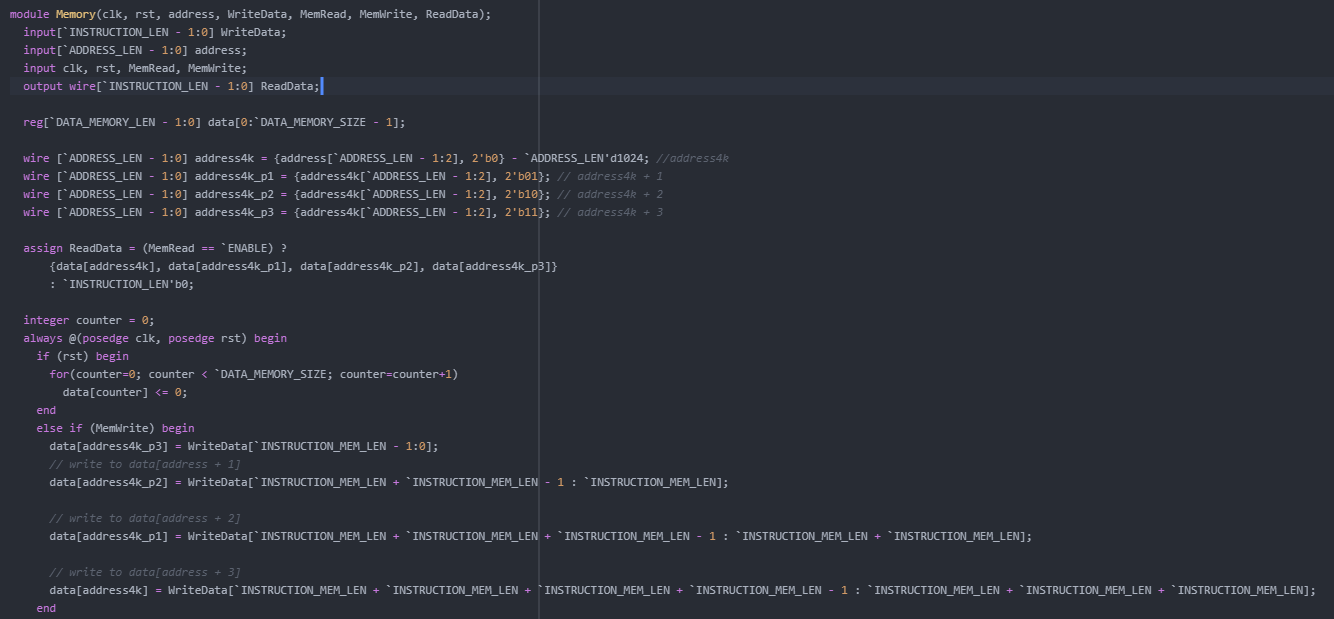
alu\_res\_in: که برای تعیین آدرس خواندن و یا نوشتن می باشد.

Val\_Rm: این ورودی مقداری است که باید در حافظه نوشته شود.

mem\_r\_en\_in: سیگنال برای خواندن از حافظه.

mem\_w\_en\_in: سیگنال برای نوشتن در حافظه.

در شکل زیر بخش از کد پیاده سازی حافظه را مشاهده می کنید:

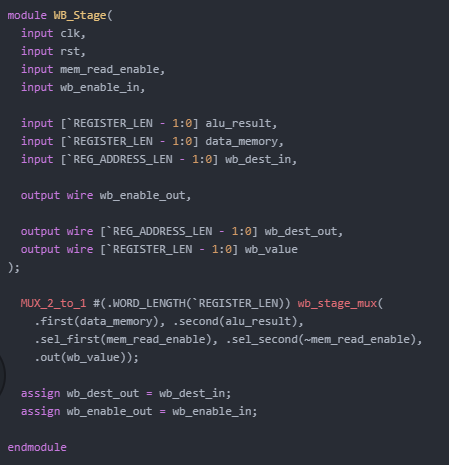


**Figure17: کد مربوط به Memory**

همان طور که در شکل 17 مشخص است، پیاده سازی به شکلی می باشد که امکان دسترسی به تک بایت در حافظه وجود ندارد و در هر مرحله 4 بایت در حافظه نوشته یا از آن خوانده می شود.

همچنین خروجی حافظه ، des و مقدار alu\_Res وارد رجیستر پایپ لاین در بین دو استیج Memory و WB میشود.

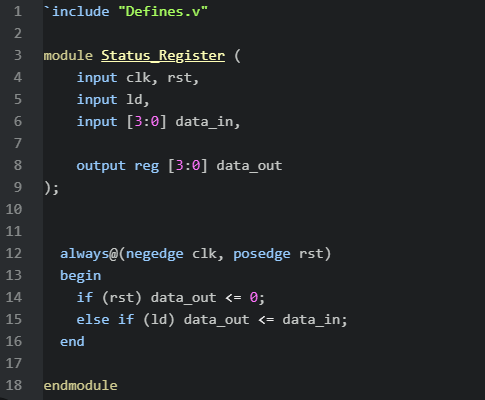
در این بخش یک مالتی پلکسر پیاده سازی شده است که با توجه به مقدار mem\_read\_enable انتخاب می کند از بین مقادیر alu\_result و data\_memory کدام یک به استیج ID منتقل شود. پیاده سازی این بخش در شکل زیر مشاهده می کنید:



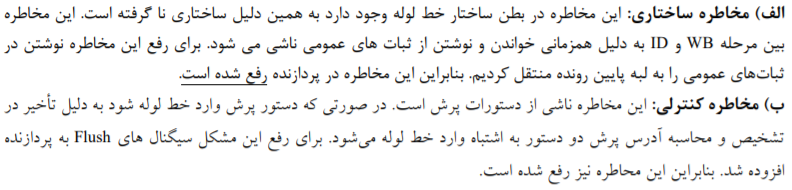
**Figure18: کد مربوط به WB**

این ماژول مسئول مگه داری Status های ما است. به دلیل اینکه تمام دستورات نمی توانند Status را تغییر دهند یک سیگنال لود برای آن قرار داده شده است که هر وقت S و یا همان STATUS در دستور فعال بود بتوان مقادیر Status جدید که از مرحله EXE به آن می رسد را تغییر دهد. تمامی حالات Status در توضیح آزمایش و دستورات گفته شده است. در این آزمایش ما فقط از 4 Status استفاده می کنیم. این موارد در دستورات شرطی که انجام شدن یا نشدن انها با توجه به مقدار یکی از این Status ها است نیاز است و تغییر آن فقط در زمان هایی که دستور های خاصی که S ان ها 1 است انجام می پذیرد. بقیه مدار مانند یک ثبات عادی کار می کند که در صورت وجود لود مقدار ورودی را در خروجی می برد.

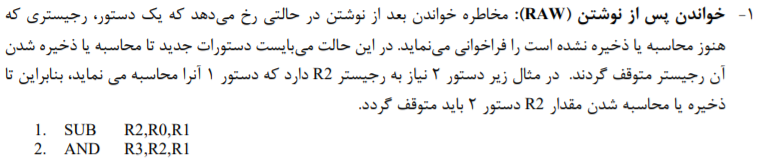
کر مربوط به این قسمت در شکل زیر آمده است که چون کد ساده ای است توضیحات بالا برای آن امده است.

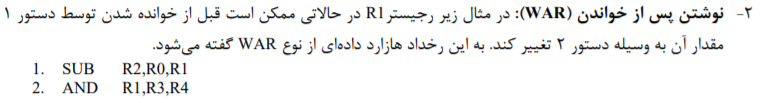


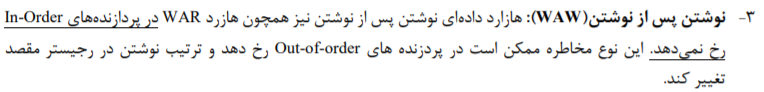
ما در پردازنده ها در کل 3 نوع مخاطره (Hazard) داریم.

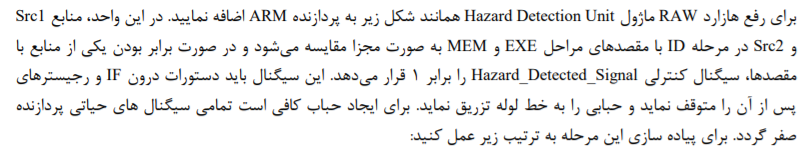




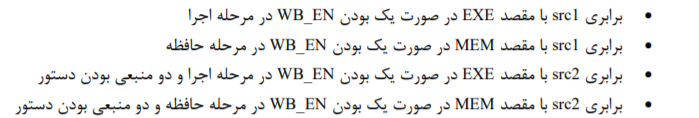








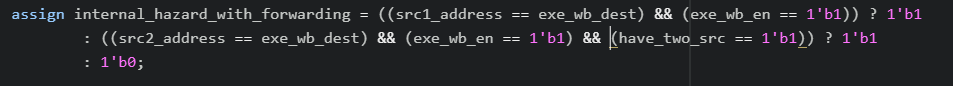
حالت هایی که هازارد RAW رخ می دهد به شرح زیر است:



برای ایجاد این ماژول همانند بالا دستورات را کنترل می کنیم.

ورودی های این ماژول ثبات مقصد در استیج EXE، سیگنال نوشتن در استیج EXE، ثبات مقصد ار استیج MEM و سیگنال نوشتن در استیج MEM است و همین طور منبع های ثبات های ما که همان scr1 و src2 هستند.

همان طور که گفته شده ما می خواهیم RAW را بر طرف می کنیم، می دانیم این hazard زمانی رخ می دهد که دستور ما Store باشد و یا بیت Immediate دستور ما 1 باشد (شامل دستوراتی که تنها دارای یک منبع هستند). برای زمان store بیت نوشتن در مموری یا MEM\_W و برای زمان Immediate بیت I 0 است که نات آن را استفاده می کنیم. با OR کردن این 2 بیت می فهمیم دستور ما احتمال hazard دارد و آن را با حالت های بالا چک می کنیم. کد مربوط به آن:



است که تمامی حالت های hazard را با هم چک می کند.

در این حالت، زمانی که هر کدام از اتفاقات گفته شده رخ دهد این واحد می تواند پایپ کلی پردازنده را متوقف کند. این کار توسط خروجی همین واحد که در واقع همان سیگنال Freeze که در استیج IF و ID گفته شد است. در صورت یک بودن این سیگنال ثبات ها تغییر نمی کنند و کنترلر تمامی سیگنال های کنترلی را 0 خروجی می دهد.

پردازنده می تواند در حالت Hazard که همین حالت است و یا حالت Forwarding که در قسمت بعد توضیح داده می شود کار کند. در حالت Hazard سیستم کند تر کار می کند چرا که در صورت وقوع RAW، پایپ ها می ایستند و منتظر می مانند تا محاسبات قبلی انجام شوند ولی در حالت Forwarding این اتفاق نمی افتد که در حالت بعد توضیح داده می شود.

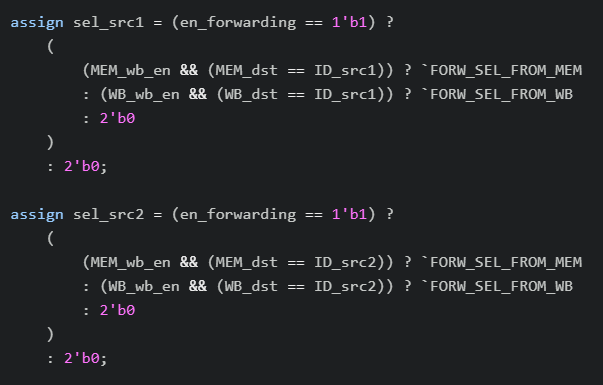
در این قسمت می خواهیم همان hazard هایی که در قسمت قبل با Hazard Detection بر طرق کردیم با واحد Forwarding انجام دهیم. فرق این قسمت با قبلی آن است که در حالت قبل ما در صورت بروز Hazard ما تمام پایپ را متوقف می کنیم تا محاسبات انجام شوند و منبع های مورد نیاز نوشته شوند ولی در این قسمت می خواهیم بدون توقف پایپ ها منبع مورد نیاز را از استیج بعدی به استیج های قبلی انتقال دهیم و بتوانیم سریع تر به ان ها دسترسی پیدا کنیم.

برای این قسمت ما از داده های رو به جلو برای دو قسمت می توانیم استفاده کنیم، یکی زمانی که store اتفاق می افتد ما به جای توقف پایپ با استفاده از Hazard Detection، می توانیم از نتیجه واحد ALU در استیج مموری و ثبات مقصد در استیج باز نشانی استفاده کنیم به این صورت که یک مولتی پلکسر به استیج EXE اضافه می کنیم و سیگنال انتخاب آن را با استفاده از همین واحد درست می کنیم. این مولتی پلکسر آدرس حافظه در استیج MEM را می سازد.

برای قسمت بعد زمانی که حالت Immediate داریم و RAW رخ داده است می توانیم به جای توقف پایپ، یک مولتی پلکسر دیگر به استیج EXE اضافه می کنیم تا بتوانیم داده ی آن را از استیج بعد به آن منتقل کنیم. در این مولتی پلکسر داده یا از نتیجه واحد ALU در استیج مموری یا ثبات مقصد در استیج باز نشانی و یا در صورت عادی و همان منبع 2 دستور است. که انتخاب آن نیز از همین واحد ساخته می شود. این مولتی پلکسر منبع دوم واحد ALU را می سازد و در صورت برابر بودن با ثبات مقصد در استیج Write Back باید ثبات مقصد Write Back انتخاب شود

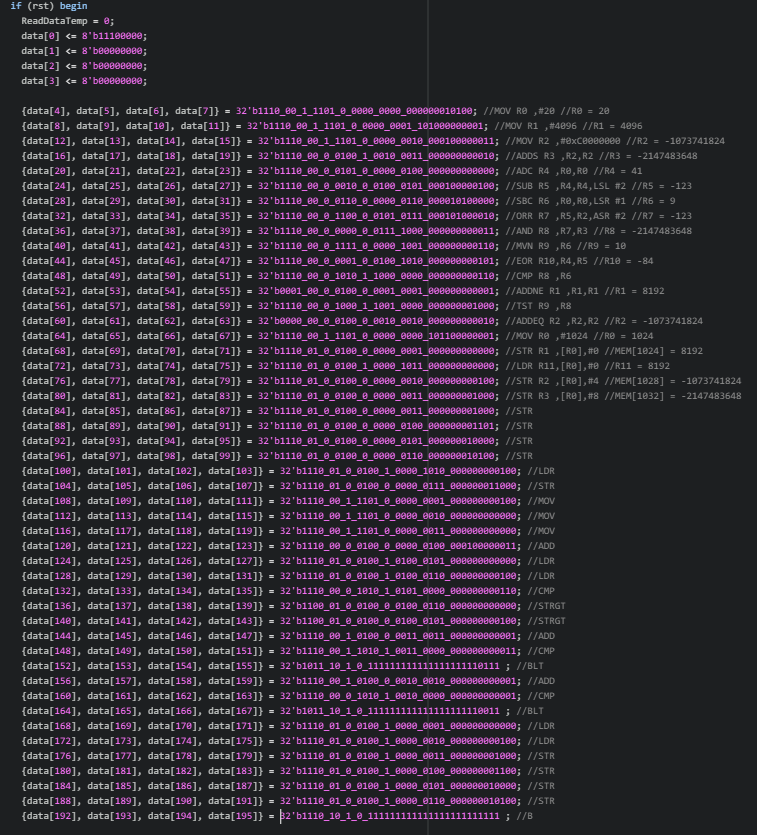
در این قسمت در صورتی که در استیج مموری سیگنال نوشتن فعال باشد (MEM\_WB\_EN) و ثبات مقصد در استیج مموری با ثبات منبع 1 که همان منبع 1 خروجی از فایل ثبات ها در استیج EXE است برابر باشد باید خروجی واحد ALU را که از استیج MEM آمده است بر داریم. در این جالت سیگنال انتخاب مولتی پلکسر اول را انتخاب می کنیم.

برای مولتی پلکسر دوم، همانند قبل است و زمانی که شروط بالا برابر با منبع 2 از ثبات فایل در استیج EXE به جای منبع 1 بود باید داده واحد ALU از مموری برداشته شود و در صورت برابر بودن با ثبات مقصد در استیج Write Back باید ثبات مقصد Write Back انتخاب شود. کد مربوط به این قسمت:



در این قسمت می خواهیم به نتایج این آزمایش بپردازیم.

برای این کار یک تست بنچ می نویسیم و ورودی های آن ریست مدار و سیگنال داشتن forwarding و یا hazard است. دستورات تست را درون حافظه دستورات در مرحله واکشی می نویسیم. دستورات مانند زیر می شوند:

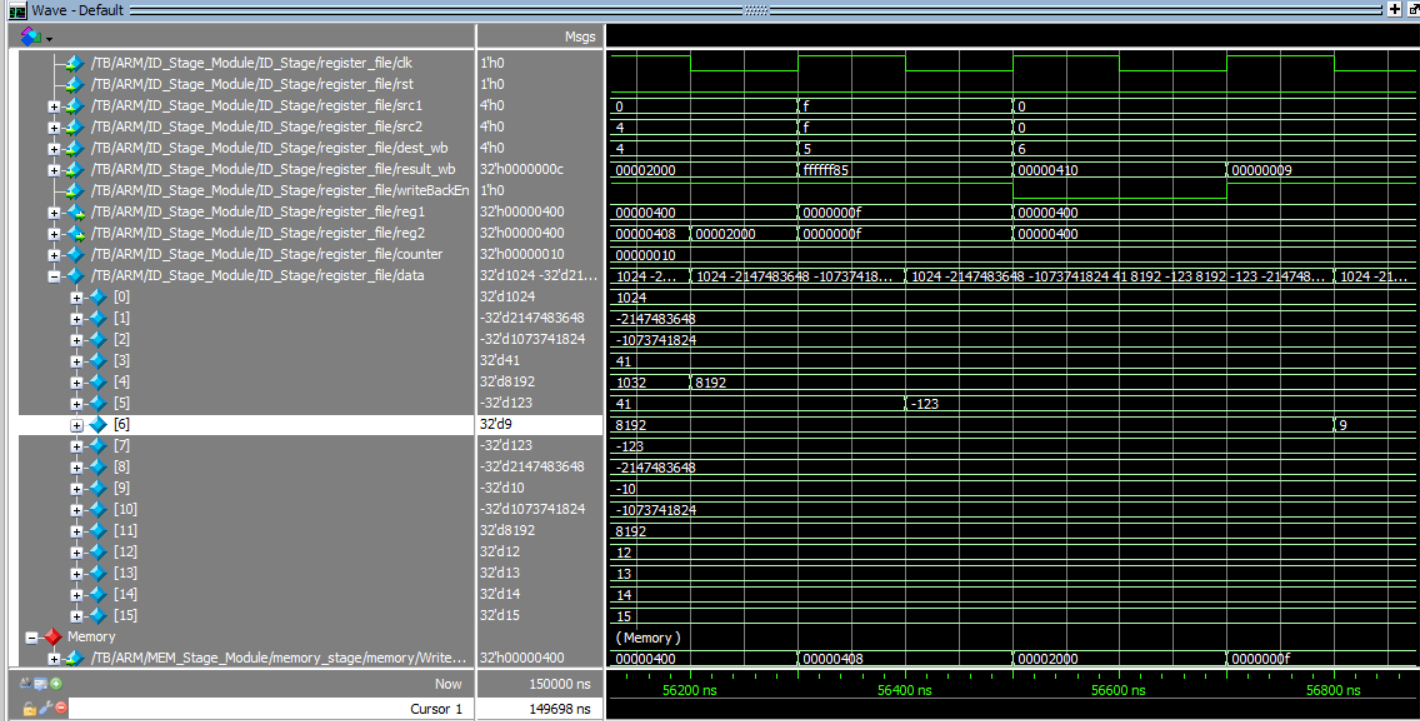


حال این مدار را یک بار در Modelsim و یک بار در Quartus اجرا می کنیم و نتایج را مشاهده می کنیم. برای مشاهده نتایج در Modelsim نتیجه آخری در ثبات ها و حافظه را نشان می دهیم و تعداد clock هایی که طول کشیده تا دستورات کامل شوند را برای هر دو حالت نمایش می دهیم. و در Quartus، مدار RTL تولید شده به همراه تعداد المان های منطقی و resource استفاده شده توسط مدار را نشان می دهیم.

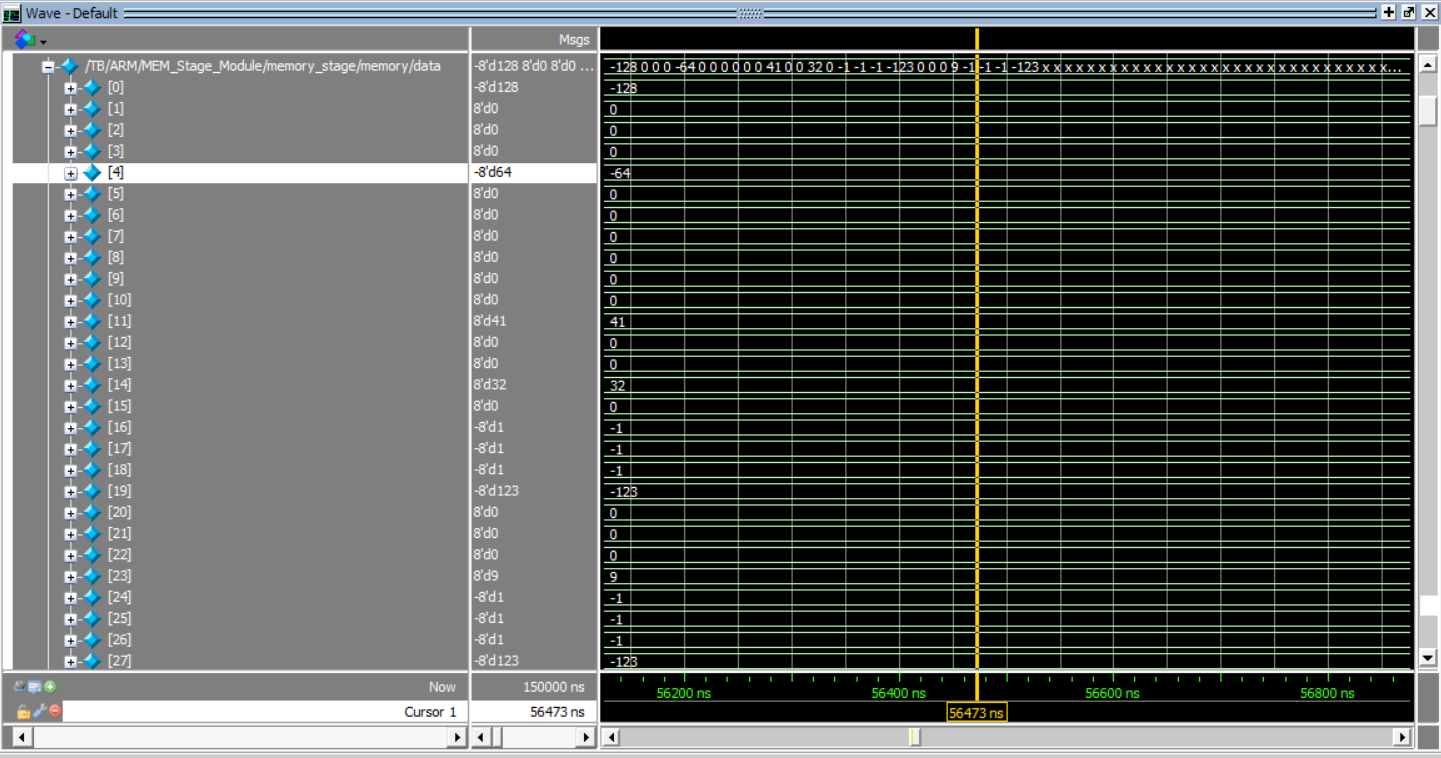
* نتایج شبیه سازی آزمایش در نرم افزار Modelsim: همان طور که میدانیم باید جواب هر دو حالت با هم یکی باشد که در عکس ها هم آمده و درست است.

در حالت داشتن Hazard Detection Unit:

مقادیر ثبات ها در انتها و تغییرات آخر انها به شکل:



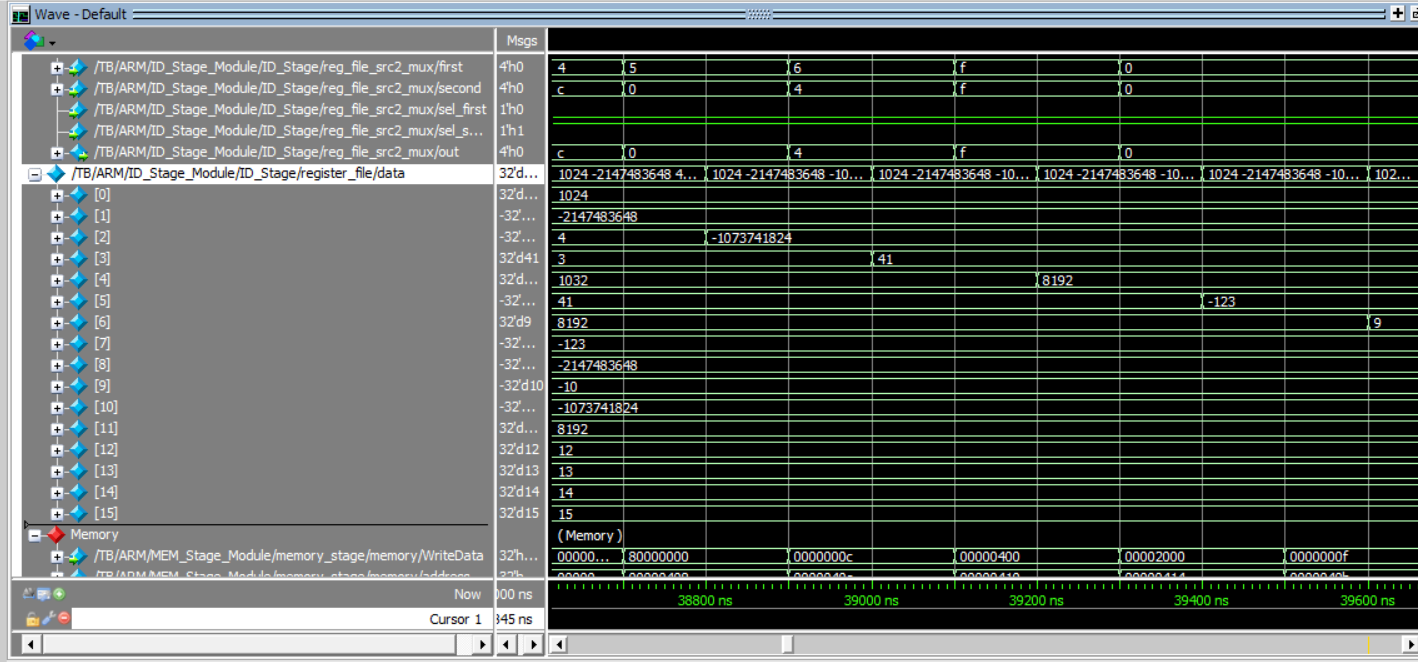
مقادیر نهایی در حافظه:



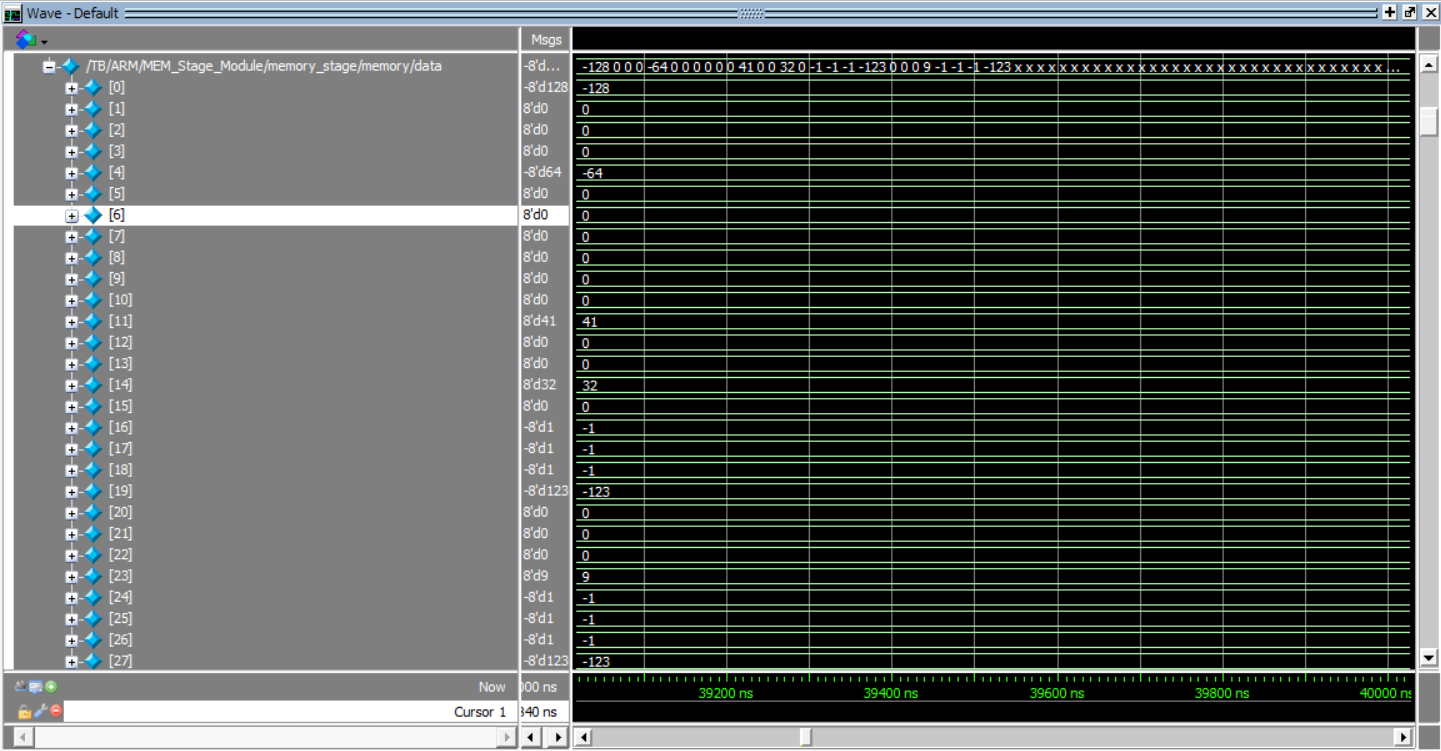
همان طور که در شکل هم دیده می شود مدار بعد از 56800 ns تمام دستورات را به اتمام می رساند. زمان کلاک ما در این مدار 100\*2 بوده است که در نتیجه: زمان می برد تا کار مدار تمام شود. این زمان رای حالت Hazard Detection است.

در حالت داشتن Forwarding Unit:

مقادیر ثبات ها در انتها و تغییرات آخر انها به شکل:



و مقادیر ذخیره شده در حافظه به شکل:

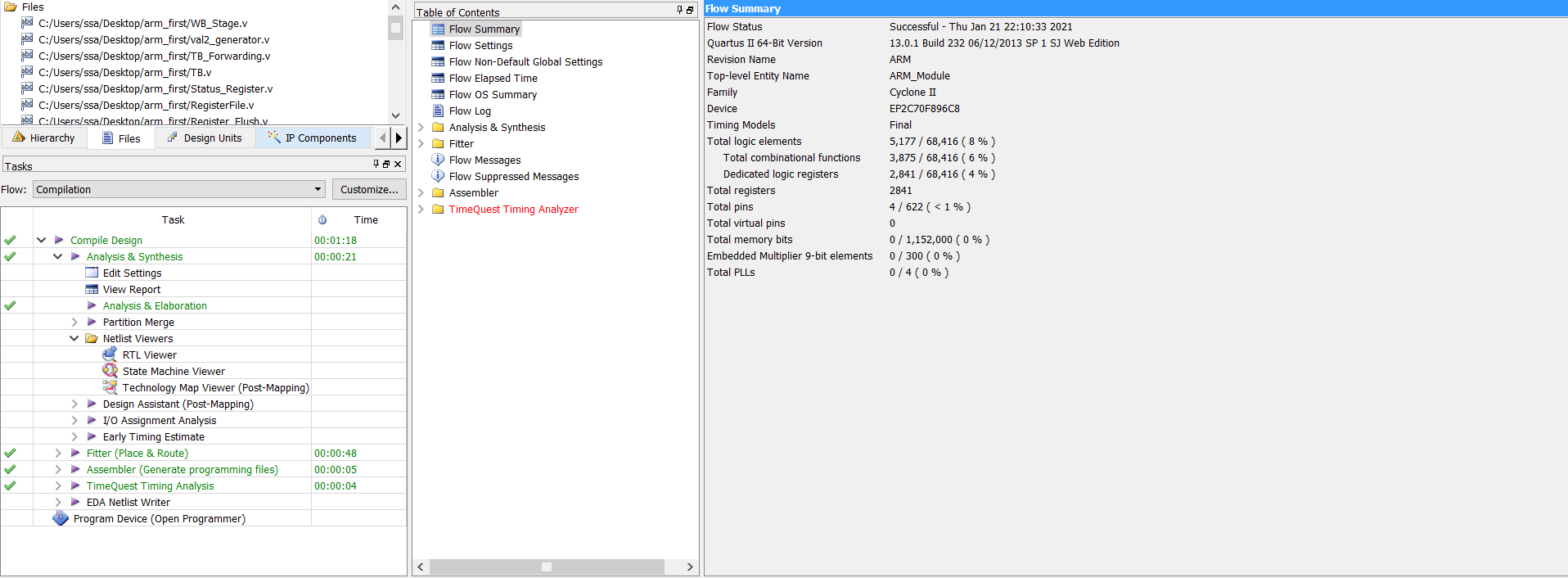


همان طور که در شکل هم دیده می شود مدار بعد از 39600 ns تمام دستورات را به اتمام می رساند. زمان کلاک ما در این مدار 100\*2 بوده است که در نتیجه: زمان می برد تا کار مدار تمام شود. این زمان رای حالت forwarding است.

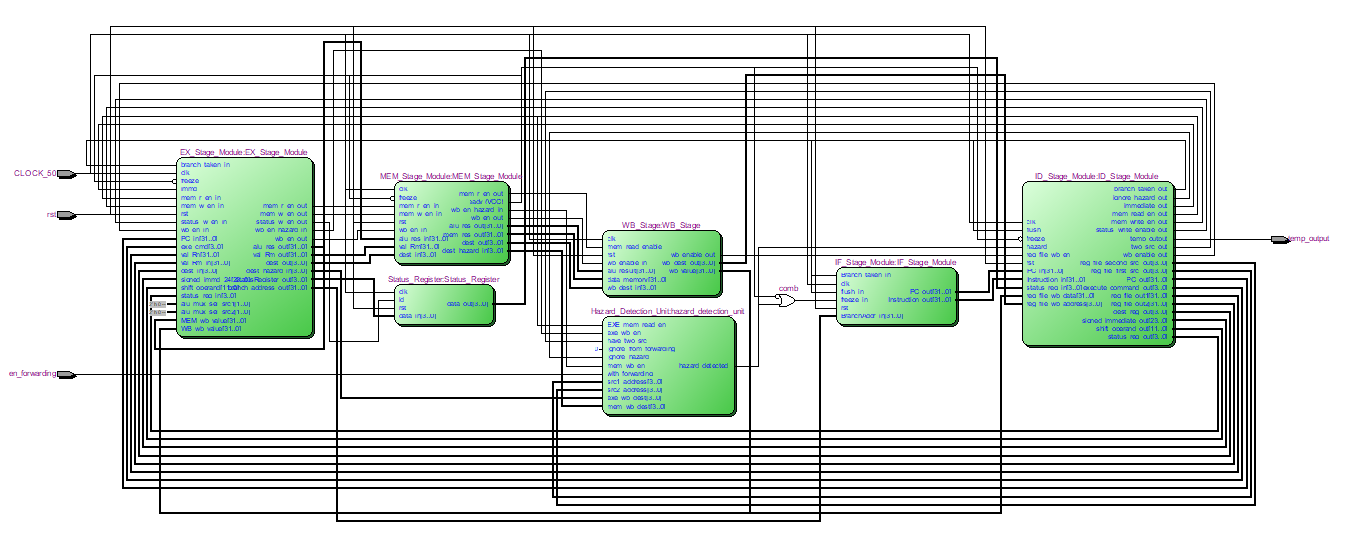
* حال همین مدار ها را در Quartus شبیه سازی می کنیم و داریم:

*در حالت اول برای مدار با Hazard Detection داریم:*

*نتایج بعد از اجرای کد درون نرم افزار:*



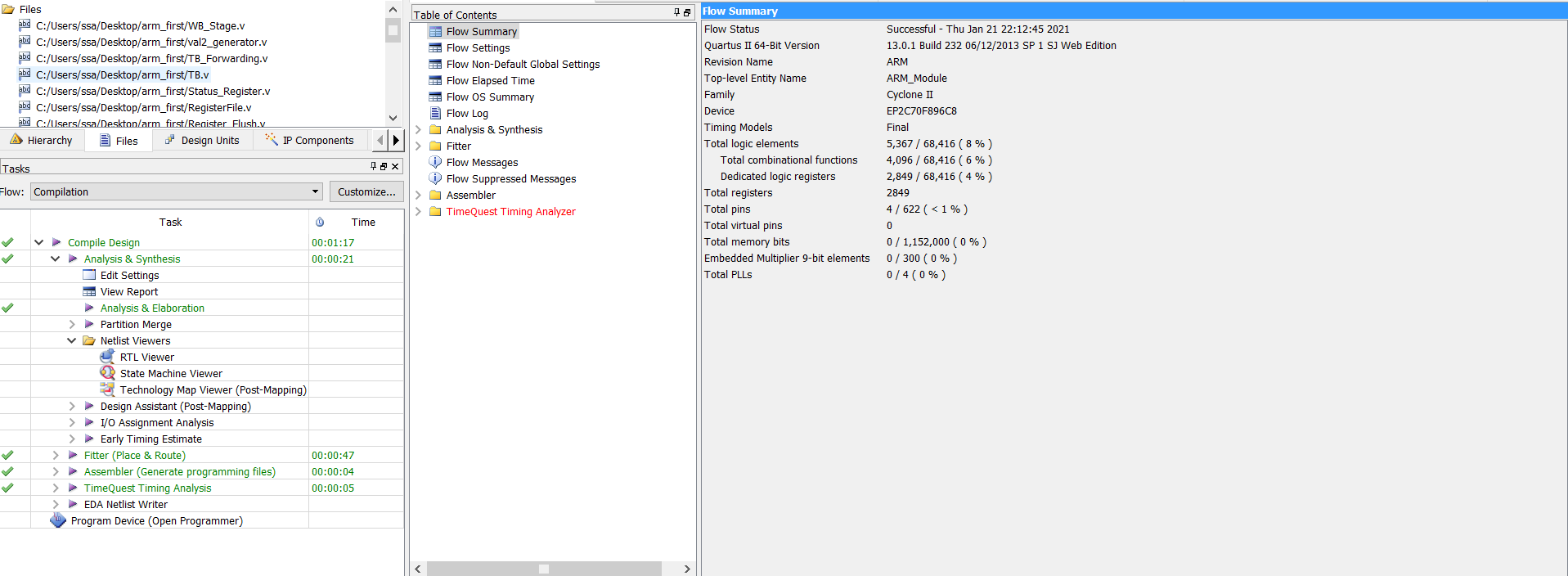
*مدار دیده شده توسط نرم افزار:*



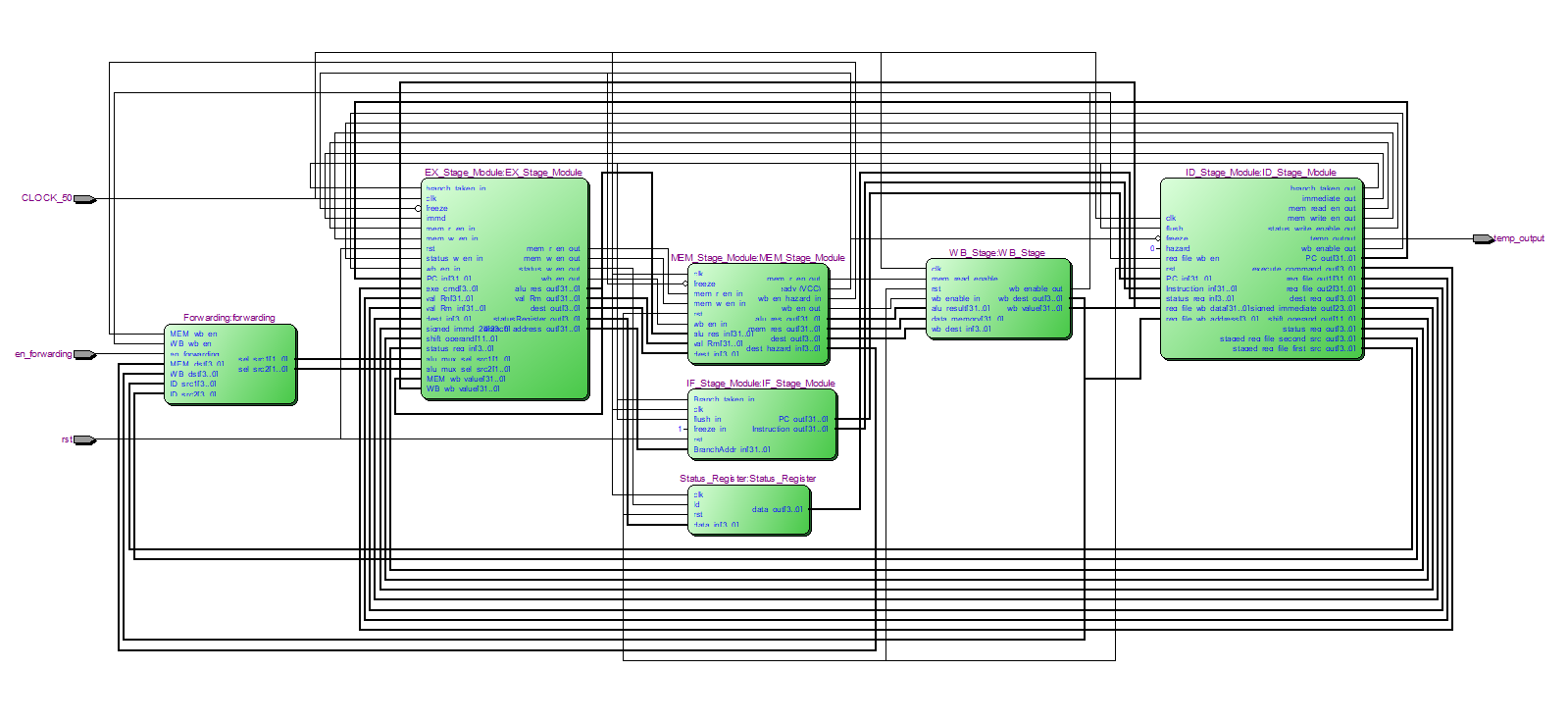
*به علت اینکه خروجی برای مدار نهایی از مرحله ID گذاشتیم مدار ID جلو تر کشیده شده است. ولی همان طور که می دانستیم اول IF و ID و بقیه مدار بود که درست است. علاوه بر آن همان طور که می دانستیم Hazard Unit خروجی ای به IF و ID می دهد که درست است. بقیه مدار و سیگنال ها نیز با مقایسه با حالت اولیه می بینیم که همگی همان و درست است.*

*و در حالت فورواردینگ داریم:*

*نتایج بعد از اجرای کد درون نرم افزار:*



*مدار دیده شده توسط نرم افزار:*



*به علت اینکه خروجی برای مدار نهایی از مرحله ID گذاشتیم مدار ID جلو تر کشیده شده است. ولی همان طور که می دانستیم اول IF و ID و بقیه مدار بود که درست است. علاوه بر آن همان طور که می دانستیم Forwarding خروجی های به EXE می دهد که در واقع انتخاب مولتی پلکسر ها است که درست است. ورودی های این واحد نیز از قسمت های MEM, EXE آمده است که درست است. بقیه مدار و سیگنال ها نیز با مقایسه با حالت اولیه می بینیم که همگی همان و درست است.*

*برای مقایسه دو حالت جدول زیر را داریم:*

|  |  |  |  |
| --- | --- | --- | --- |
| *Total Register* | *Total Logic Element* | *Total Clock* |  |
| *2841* | *5177 (8%)* | *284* | *Hazard Detection Unit* |
| *2649* | *5367 (8%)* | *194* | *Forwarding Unit* |

*همان طور که انتظار داشتیم در حالت دو به جلو سرعت مدار تقریبا 1.5 برابر حالت هازارد بود. ولی این سرعت باعث گرفتن مقدار بیشتری از المان های منطقی نیز بوده است چرا که همان طور که دیده می شود مقدار المان های منطقی در حالت رو به جلو بیشتر است که نشان از بیشتر گرفتن ریسورس کل مدار می شود. در اینجا ما مقداری ریسورس بیشتر رو فدای سرعت بالاتر کرده ایم که می زان تسریع نشان می دهد این trade-off به خوبی انجام شده است چرا که سرعت مناسبی دریافت کرده ایم.*

*برخی از مشکلات پیش آمده به شرح زیر است:*

1. *زمانی که می خواستیم قسمت ID را به قسمت قبل وصل کنیم و تست کنیم با مشکلات زیادی رو به رو شدیم. این قسمت برای گروه ما بیشترین ایراد را داشت که بیشترین قسمت آن قسمت Controller آن بوده است. چرا که در حالت اول برخی از دستورات را کاملا آشنا نبودیم و کمی طول کشید تا سیگنال های کنترلی همه را بدون ایراد درست کنیم.*
2. *مشکل دیگر زمانی بود که می خواستیم مدار نهایی را تست کنیم. این مدار که در هر مرحله تست شده بود زمانی که با کل دستورات تست شد غلط های زیادی از جمله رجیستر نشده بودن 1 سیگنال کنترلی، تغییر نکردن درست آدرس و ... بود. بیشترین زمان در انجام پروژه برای این قسمت بود چرا که باید مرحله به مرحله کا کد را چک می کردیم و پیدا می کردیم هر کدام از دستورات در کدام مرحله دچار مشکل می شوند. همین طور برخی از سیگنال ها در مرحله آخر مقدار X می گرفتند که به معنی وصل نبودن بودند. زمان زیادی در این قسمت صرف شد تا مدار بتواند برای بار اول کار کند.*
3. *مشکل دیگر زمانی بود که می خواستیم بعد از نوشتن مدار رو به جلو آن را با کل مدار تست کنیم. در ابتدا فکر کردیم شاید مشکل از قسمت قبل هم بوده باشد در حالی که با تست کردن دوباره مرحله به مرحله برای پیدا کردن مشکل فهمیدیم که مولتی پلکسر هایی که در این مرحله به مدار اضافه شده بودند مشکل داشتند. زمانی نسبتا خوبی برای درست کردن این مشکل گرفت که در آخر موفق به درست کردن آن شدیم.*
4. *یکی از مشکلات جالب زمان Compile با Quartus داشتیم. در این زمان تمام resource های ما 0 گزارش می شدند و نمی دانستیم دلیل آن چیست. بعد از تحقیق متوجه شدیم به این دلیل است که نرم افزار خود Optimization هایی روی مدار انجام می دهد و چون مدار ما هیچ خروجی ای از مدار نداشت مدار را درست نمی کرد. به همین دلیل، یک خروجی از رجیستر فایل برای مدار نهایی در نظر گرفتیم که آن خروجی SRC2 برای این ماژول بود. بعد از وصل کردن آن به یکی از پین های FPGA مدار ساخته شد و توانستیم خروجی را مقایسه کنیم.*
5. *مشکلات دیگری نیز در مراحل وجود داشت ولی سخت ترین آنها با شرح بالاست*