به نام خدا

گزارش کار آزمایشگاه معماری کامپیوتر

گروه ۳۲

محمد عظیم پور ۸۱۰۱۹۷۶۵۷، امیرعلی عطایی نائینی ۸۱۰۱۹۷۶۳۳

گروه درسی کد ۰۲، کلاس سه شنبه ۱۳ الی ۱۶

گزارش کار جلسات درس آزمایشگاه معماری کامپیوتر به ترتیب مشخص شده در فهرست مطالب موجود در صفحه بعد در این فایل ذکر شده است.

در هر گزارش توضیحات راجع به ماژول هایی که در آن جلسه باید پیاده سازی می شد ذکر شده و همچنین برای ماژول هایی که ساختار داشتند نمودار ساختاری رسم شده است.

در انتهای هر فاز نتایج آزمودن ماژول پیاده سازی شده با توجه به خواسته ی دستور کار آزمایش، تصویر نهایی مسیر داده پردازنده بعد از اضافه کردن قابلیت های مطلوب آن فاز و نتیج سنتز مدار طراحی شده روی برد EP4CE115F29C9L از خانواده Cyclone IV E آورده شده است.

برای فهم بهتر چگونگی اجرای PipeLine دستورات در حین آزمون، برای هر بخش یک فایل Excel ضمیمه شده است که به تفکیک سیکل ساعت نحوه ی اجرای هر بخش از هر دستور را نشان می دهد. این فایل ها در پوشه PipeLines قرار دارند.

برای اینکه امکان دیدن نمودارها با جزئیات وجود داشته باشد در پوشه Figures فایل با فرمت vsdx هرکدام از آن ها ضمیمه شده است.

فهرست مطالب

فاز اول: طراحی پردازنده ARM ۳

جلسه اول: واکشی ۳

جلسه دوم: کدگشایی و کنترل ۵

جلسه سوم: اجرا و بازنشانی ۹

جلسه چهارم: حافظه و تشخیص مخاطره ۱۲

فاز دوم: اضافه کردن تکنیک ارسال به جلو ۲۰

فاز سوم: استفاده از SRAM به عنوان حافظه داده ۲۶

فاز چهارم: استفاده از حافظه نهان ۳۱

فاز اول: طراحی پردازنده ARM

جلسه اول: واکشی

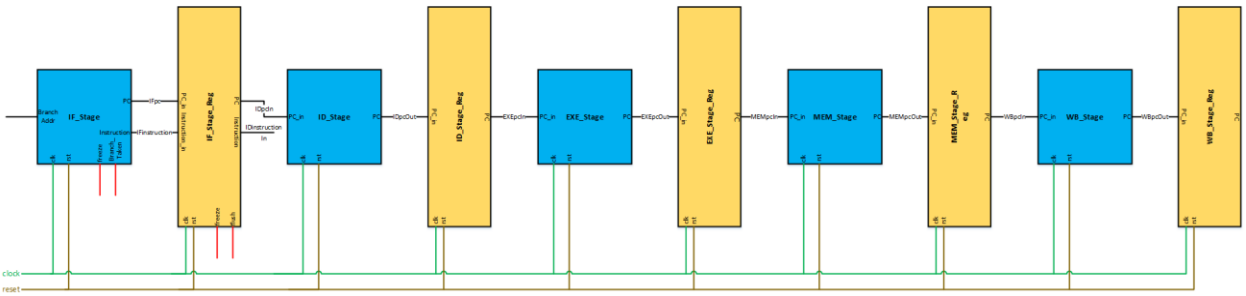
ماژول های خواسته شده در صورت آزمایش به این شکل پیاده سازی شده اند:

تمام ماژول ها به غیر از ماژول واکشی در این فاز صرفا باید ورودی خود را به خروجی منتقل می کردند. این کار به وسیله یک always در این ماژول ها انجام می شود که مقادیر خروجی را برابر ورودی متناظر قرار می دهد. در ماژول هایی که رجیسترهای میان ماژول های اصلی هستند این always به لبه بالا رونده clock حساس است ولی در ماژول های اصلی این گونه نیست (علت پیاده سازی به این شکل ترکیبی بودن ماژول های اصلی (عملیات خواندن از ماژول حافظه که تنها عملیات این ماژول است که اطلاعات به قسمت های بعدی می فرستد هم با سیگنال clock همگاگ نیست) و ترتیبی بودن رجیستر هاست).

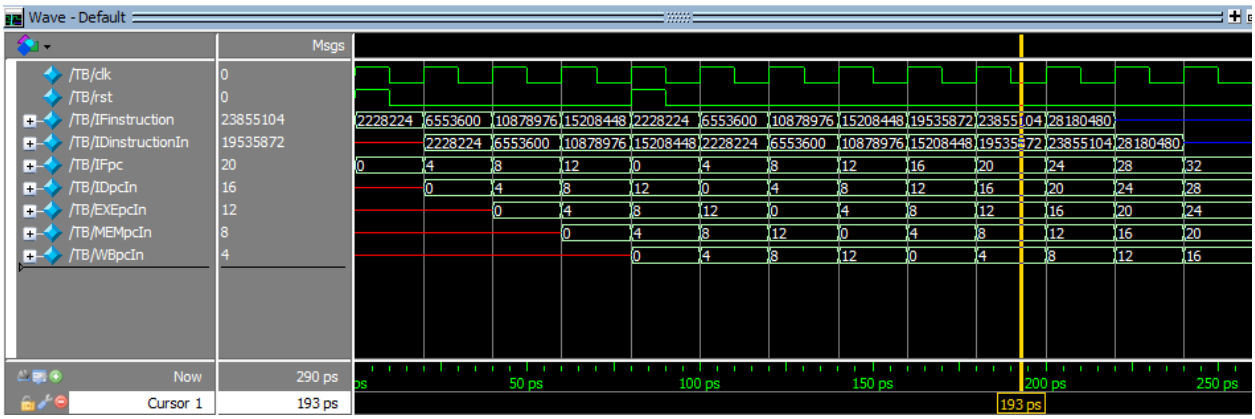
در ماژول واکشی یک آرایه با ۲۵۶ خانه ۸ بیتی به عنوان حافظه دستور تعریف و ۲۸ خانه اول آن با توجه به دستور کار آزمایشگاه مقدار دهی شده است. در هر ۴ خانه از این آرایه یک دستور ۳۲ بیتی قرار می گیرد. یک رجیستر ۳۲ بیتی نیز برای pc تعریف شده که مقدار آن در یک always تغییر می کند. خروجی Instruction ماژول چهار خانه متوالی از حافظه دستور است که مقدار pc به اولین آنها اشاره می کند (مقدار pc برحسب بایت و هر دستور نیز ۴ بایت است).

یک بلوک always در این ماژول قرار دارد که به لبه بالارونده سیگنال های clock و reset حساس است. در صورت رسیدن به لبه بالا رونده reset ماژول ریست می شود و مقدار رجیستر pc برابر صفر می شود. در غیر این صورت وقتی به لبه بالارونده clock برسیم اگر سیگنال freeze برابر یک نباشد مقدار رجیستر pc چهار تا زیاد می شود. اگر این سیگنال برابر یک باشد مقدار pc تغییری نمی کند.

برای آزمودن ماژول های پیاده سازی شده از هرکدام یک نمونه گرفتیم و به هم متصل کردیم (به همان ترتیبی که در معماری پردازنده باید کنار هم قرار می گرفتند). تصویر این مدار به صورت زیر است:



در این آزمون ابتدا پردازنده ریست می شود و سپس ۴ سیکل ساعت به خواندن دستورات می پردازد. سپس دوباره سیگنال ریست فعال می شود. نمودار سیگنال های مهم مدار آزمون به صورت زیر است:



مشاهده می شود که بعد از این اتفاق دوباره مقدار pc برابر صفر می شود و دستورات از ابتدا خوانده می شوند و این کار تا دستور آخر ادامه پیدا می کند. با توجه به جدول زیر که معادل دسیمال دستورات ۳۲ بیتی داده شده را نشان می دهد، می توان تحقیق کرد که مقداری که به عنوان دستور از ماژول واکشی خارج می شود صحیح است (این مقدار برای مشاهده بهتر روی نمودار به صورت دسیمال نشان داده شده است). همچنین می توان دید که دستور ورودی به ماژول کدگشایی (IDinstructionIn) یک سیکل عقب تر از دستور خروجی از ماژول واکشی (IFinstruction) است و این یعنی حرکت موج گونه دستور در این دو ماژول به درستی انجام می شود.

Instructions				
Binary				Decimal
00000000	00100010	00000000	00000000	2228224
00000000	01100100	00000000	00000000	6553600
00000000	10100110	00000000	00000000	10878976
00000000	11101000	00010000	00000000	15208448
00000001	00101010	00011000	00000000	19535872
00000001	01101100	00000000	00000000	23855104
00000001	10101110	00000000	00000000	28180480

همچنین مشاهده می شود که مقدار pc به صورت PipeLine در ماژول های مختلف جابجا می شود. در نمودار بالا، IFpc مقدار خارج شده از ماژول واکشی، IDpcIn مقدار وارد شده به ماژول کدگشایی، EXEpcIn مقدار وارد شده به ماژول اجرا، MEMpcIn مقدار وارد شده به ماژول حافظه و WBpcIn مقدار وارد شده به ماژول بازنشانی برای pc هستند.

جلسه دوم: کدگشایی و کنترل

ماژول های خواسته شده در صورت آزمایش به این شکل پیاده سازی شده اند:

ID_Stage_Reg: این ماژول حاوی ثبات های بعد از ماژول کدگشایی است و صرفا باید ورودی های خود را به خروجی منتقل کند. برای این هدف از یک بلوک always استفاده شده است که این کار را با لبه بالارونده clock انجام می دهد.

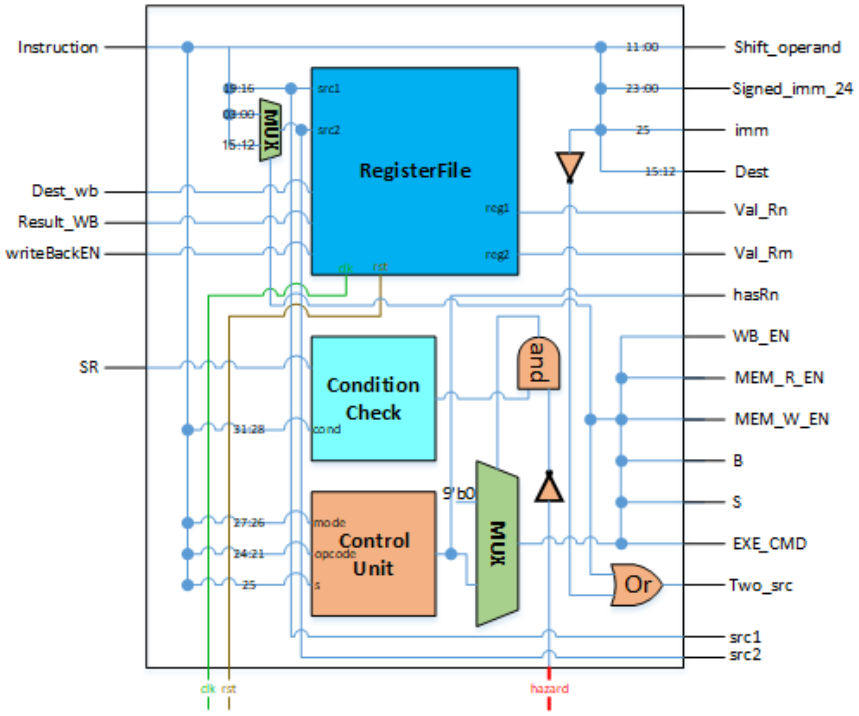
ConditionCheck: این ماژول ۴ بیت اول دستور که مربوط به شرط اجرای آن است به همراه ۴ بیت ثبات وضعیت دریافت میکند و با توجه به جدول داده شده در یک بلوک always که همگام با clock نیست خروجی را تعیین می کند. خروجی یک ماژول به معنای برقرار بودن و خروجی صفر آن به معنای برقرار نبودن شرط است.

RegisterFile: از یک آرایه ۱۵ تایی برای نگهداری ثبات های عمومی در این ماژول استفاده شده است که به وسیله یک بلوک initial به شکل خواسته شده مقدار دهی می شود (مقدار هر ثبات برابر شماره آن ثبات). ورودی ماژول دو آدرس (شماره) ۴ بیتی است که باید مقادیر داخل ثبات هایی که این دو آدرس به آن ها اشاره می کنند روی خروجی ماژول قرار بگیرد. همچنین یک ورودی یک بیتی به عنوان سیگنال بازنشانی، یک ورودی ۴ بیتی دیگر به عنوان آدرس (شماره) ثبات مقصد و یک ورودی ۳۲ بیتی به عنوان مقداری که باید بازنشانی شود به ماژول داده می شود که در صورت برابر یک بودن سیگنال بازنشانی، مقدار ورودی ۳۲ بیتی همگام با لبه پایین رونده clock در ثبات مقصد نوشته می شود.

ControlUnit: بیت های ۲۷ و ۲۶ دستور را به عنوان mode، بیت های ۲۴ تا ۲۱ دستور را به عنوان opcode و بیت ۲۵ دستور را به عنوان ورودی s دریافت می کند و سیگنال های کنترلی مربوط به دستوری که این سه ورودی تعیین می کنند ایجاد می کند. خروجی های این ماژول ورودی ۴ بیتی واحد حساب و منطق (ALU) که نوع عملیات این واحد را تعیین می کند، سیگنال خواندن از حافظه، سیگنال نوشتن در حافظه، سیگنال بازنشانی، سیگنال پرش (Branch) و سیگنال به روزرسانی ثبات وضعیت هستند. این خروجی ها در یک بلوک always با توجه به حالت های مختلف mode، opcode و s مقداردهی می شوند. سیگنال های کنترلی پردازنده به تفکیک دستورات به صورت زیر است:

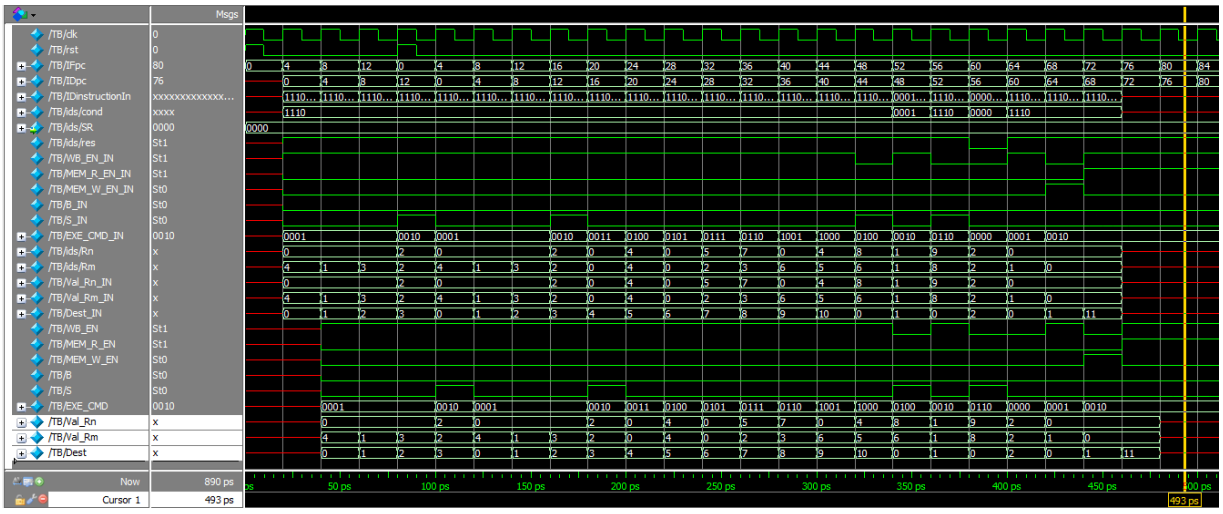
ARM Controller												
Instruction		Description	Inputs			Control Signals						
			mode	OP-Code	s	EXE_CMD	mem_read	mem_write	WB_Enable	B	S
1	MOV	Move	00	1101	s	0001	0	0	1	0	s
2	MVN	Move NOT	00	1111	s	1001	0	0	1	0	s
3	ADD	Add	00	0100	s	0010	0	0	1	0	s
4	ADC	Add with Carry	00	0101	s	0011	0	0	1	0	s
5	SUB	Subtraction	00	0010	s	0100	0	0	1	0	s
6	SBC	Subtract with Carry	00	0110	s	0101	0	0	1	0	s
7	AND	And	00	0000	s	0110	0	0	1	0	s
8	ORR	Or	00	1100	s	0111	0	0	1	0	s
9	EOR	Exclusive Or	00	0001	s	1000	0	0	1	0	s
10	CMP	Compare	00	1010	1	0100	0	0	0	0	1
11	TST	Test	00	1000	1	0110	0	0	0	0	1
12	LDR	Load Register	01	0100	1	0010	1	0	1	0	0
13	STR	Store Register	01	0100	0	0010	0	1	0	0	0
14	B	Branch	10	---	---	---	0	0	0	1	0

ID_Stage: ماژول مرحله کدگشایی پردازنده است که دستور واکشی شده را دریافت و علاوه بر ایجاد سیگنال های کنترلی مربوط به اجرای آن (که در بالا نام برده شد)، عملوند های مربوطه (مقدار رجیسترهایی که باید خوانده می شدند، مقدار shifter operand و عدد فوری مربوط به دستور پرش (signed_immed_24)) را نیز استخراج می کند. همچنین یک ورودی یک بیتی یه عنوان سیگنال بازنشانی، یک ورودی ۴ بیتی دیگر به عنوان آدرس (شماره) ثبات مقصد و یک ورودی ۳۲ بیتی به عنوان مقداری که باید بازنشانی شود به ماژول داده می شود تا در صورتی که نیاز به انجام عملیات بازنشانی باشد این کار انجام شود. در این ماژول از یک ماژول RegisterFile، یک ماژول ControlUnit و یک ماژول ConditionCheck نمونه گیری شده است (مطابق شکل زیر، فایل حاوی نمودار: 1_ARM_ID.vsdx). ورودی اول RegisterFile مستقیما برابر بیت ۱۹ تا ۱۶ دستور (Rn) است، ولی برای ورودی دوم یک عدد Multiplexer دوتایی ۴ بیت بین بیت های ۳ تا ۰ (Rm) و ۱۵ تا ۱۲ (Rd) با توجه به اینکه دستور STR هست یا خیر از روی سیگنال کنترلی نوشتن در حافظه انتخاب می کند. بعد از این انتخاب این دو مقدار ورودی به RegisterFile به خروجی src1 و src2 ماژول متصل می شوند. خروجی های RegisterFile مستقیما به خروجی ماژول وصل می شوند ولی خروجی های ControlUnit به همراه ۹ بیت صفر وارد یک Multiplexer دوتایی ۹ بیت می شوند تا خروجی ماژول ConditionCheck بینشان انتخاب کند. اگر شرط اجرای دستور برقرار بود خروجی های ControlUnit و در غیر این صورت صفر به عنوان سیگنال های کنترلی به خروجی ماژول می روند.



برای آزمودن ماژول های پیاده سازی شده از هرکدام یک نمونه گرفتیم و به هم متصل کردیم (به همان ترتیبی که در معماری پردازنده باید کنار هم قرار می گرفتند. تصویر مدار در صورت آزمایش موجود است و از تکرار آن در این قسمت خودداری می کنیم).

در این آزمون ابتدا پردازنده ریست می شود و سپس ۴ سیکل ساعت به خواندن دستورات می پردازد. سپس دوباره سیگنال ریست فعال می شود و بعد از آن دستورات تا انتها (که می شود دستور ۱۸ ام) خوانده می شوند. نمودار سیگنال های مهم مدار آزمون به صورت زیر است (توجه شود که سیگنال X_IN سیگنال خروجی از ماژول ID_Stage و ورودی به ماژول ID_Stage_Reg است و سیگنال X سیگنال خروجی از ماژول ID_Stage_Reg. به همین علت این جفت سیگنال های متناظر یک سیکل ساعت با هم اختلاف دارند):



حال درستی سیگنال های خروجی را بررسی می کنیم:

تمام دستورات به جز دو دستور ۱۳ و ۱۵ بدون شرط هستند و بین این دو دستور فقط شرط اجرای دستور ۱۵ برقرار نیست (سیگنال res خروجی ماژول ConditionCheck است). چون برای دستور ۱۳ بیاید بیت Z ثبات وضعیت صفر باشد که هست، ولی برای دستور ۱۶ باید یک باشد که نیست (ثبات وضعیت را برابر 4'b0000 تعریف کرده ایم). آن را ببینیم که تمام سیگنال های کنترلی به ازای این دستور صفر هستند. در توضیحات پایین از ذکر دلیل صفر بودن این سیگنال ها برای این دستور پرهیز می شود.

سیگنال WB_EN برای تمام دستورات به جز ۱۲ و ۱۴ و ۱۷ برابر یک است برای این سه دستور برابر صفر.

سیگنال MEM_R_EN فقط برای دستور ۱۸ که LDR است و سیگنال MEM_W_EN فقط برای دستور ۱۷ که STR است برابر یک و برای باقی دستورات برابر صفر هستند.

چون پرش نداریم سیگنال B برای تمام دستورات برابر صفر است. همچنین سیگنال به روز رسانی ثبات وضعیت در دستورات ۴ (دستور حسابی منطقی و بیت s آن برابر یک است)، ۱۲ (دستور CMP)، ۱۴ (دستور TST) برابر یک می شود.

سیگنال EXE_CMD با توجه به نوع دستور در تمامی دستورات درست تعیین می شود.

مقادیر Val_Rn و Val_Rm نیز همواره برابر مقادیر Rn و Rm ای هستند که از دستور استخراج شده اند. همچنین Dest که مقصد عملیات بازنشانیست نیز در نمودار آورده شده است.

جلسه سوم: اجرا و بازنشانی

ماژول های خواسته شده در صورت آزمایش به این شکل پیاده سازی شده اند:

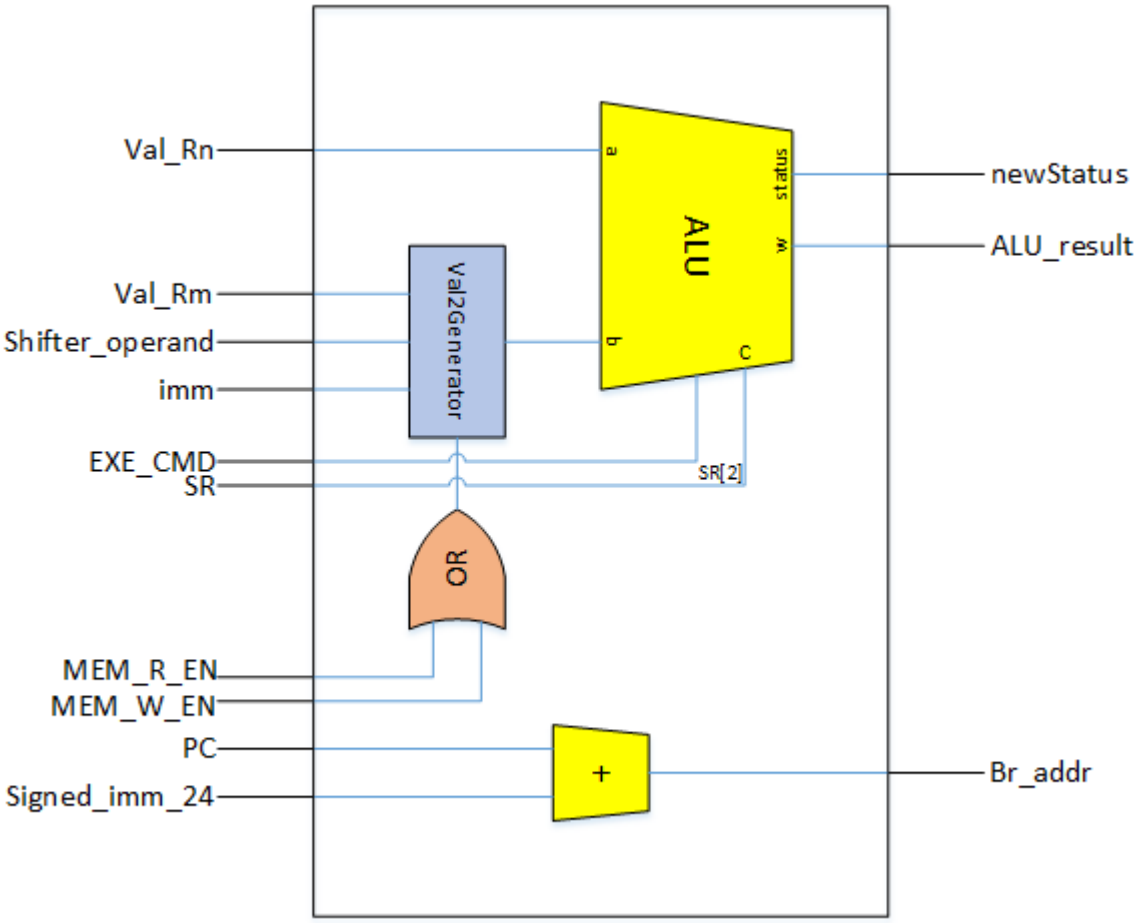
EXE_Stage_Reg: این ماژول حاوی ثبات های بعد از ماژول اجرا است و صرفا باید ورودی های خود را به خروجی منتقل کند. برای این هدف از یک بلوک always استفاده شده است که این کار را با لبه بالارونده clock انجام می دهد.

ALU: این ماژول ۴ دو عدد ۳۲ بیت، یک بیت به عنوان Carry (که بیت C ثبات وضعیت است) و ۴ بیت به عنوان مشخص کننده عملیاتی که باید انجام شود ورودی ورودی می گیرد و مطابق جدول موجود در گزارش کار عملیات را اجام می دهد. خروجی ماژول یک عدد ۳۲ بیت به عنوان حاصل عملیات است و همچنین ۴ بیت مربوط به ثبات وضعیت. خروجی ۳۲ بیتی داخل یک بلوک always که با توجه به حالات مختلف ورودی ۴ بیتی قسمت بندی شده است ناهمگام با clock تولید می شود و ۴ بیت مربوط به ثبات وضعیت با استفاده از ۴ عبارت assign در بدنه ماژول.

Val2Generator: ورودی اول ALU مستقیما همان مقدار Val_Rn است که از قسمت کدگشایی به قسمت اجرا فرستاده شده، ولی ورودی دوم حالات مختلفی می تواند داشته باشد که این ماژول با توجه به شرایط این مقدار را تولید می کند. ورودی های ماژول مقادیر Val_Rm، Shift_operand و imm هستند که توسط ماژول کدگشایی تولید شده اند، و همچنین یک ورودی دیگر که برابر حاصل OR دو سیگنال کنترلی خواندن از و نوشتن در حافظه است و مشخص می کند دستور با حاظه کار دارد یا خیر (که اگر دارد مقدار Val2 مناسب تولید شود). نحوه پیاده سازی این ماژول نیز مانند ماژول قبلی به وسیله یک بلوک always ناهمگام است که با توجه به بیت های خاص ورودی های گفته شده (مطابق توضیحات دستور کار) خروجی مناسب را تولید می کند.

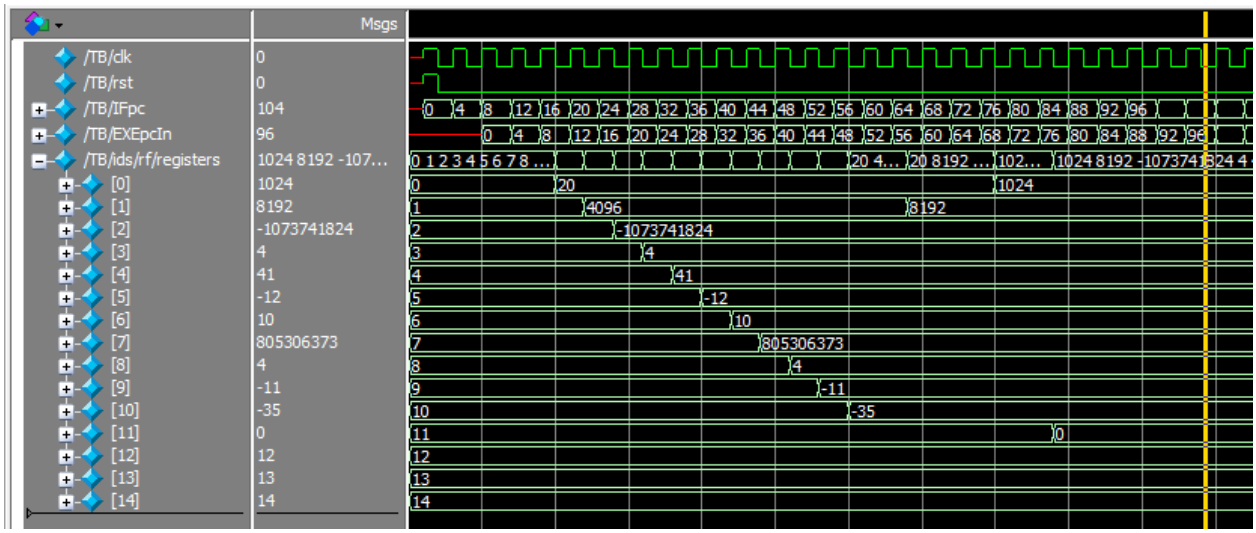
Adder: یک جمع کننده پارامتری است که پارامتر آن اندازه ورودی ها و خروجی را برحسب بیت مشخص می کند. دو ورودی و یک خروجی دارد که خروجی حاصل جمع دو ورودی است. از این ماژول برای محاسبه آدرس مقصد پرش استفاده شده است.

EXE_Stage: ماژول مرحله اجرای پردازنده است که مقادیر واکشی شده از دستورو همچنین سیگنال های کنترلی مربوط به نوشتن و خواندن از حافظه و عملیات ریاضی ای که باید انجام شود را دریافت و علاوه بر انجام عملیات حسابی توسط ALU مقدار جدید ثبات وضعیت را نیز محاسبه می کند. پیاده سازی این ماژول با نمونه گیری و متصل کردن ماژول های گفته شده در بالا و به شکل زیر انجام شده است (فایل حاوی نمودار: 1_ARM_EXE.vsdx):



برای آزمودن ماژول های پیاده سازی شده از هرکدام یک نمونه گرفتیم و به هم متصل کردیم (به همان ترتیبی که در معماری پردازنده باید کنار هم قرار می گرفتند. تصویر مدار در صورت آزمایش موجود است و از تکرار آن در این قسمت خودداری می کنیم).

در این آزمون ابتدا پردازنده ریست می شود و سپس clock زده می شود تا دستورات اجرا شود. نمودار در نمودار زیر، مقدار PC دستوری که در مرحله واکشی است (IFpc) به همراه مقدار PC دستوریکه در مرحله اجراست (EXEpcIn) به مقادیر ثبات های عمومی نشان داده شده است:



حال درستی مقادیر ثبت شده را بررسی می کنیم:

در بین ۱۸ دستور اول برنامه محک، دستورات ۱،۲،۳،۵،۷،۱۰،۱۳،۱۵،۱۶ می توانند درست اجرا شوند زیرا وابستگی داده ای برطرف نشده ندارند. دستورات دیگر یا مقدار ثبات مورد نیازشان هنوز به روز نشده و یا از ثباتی استفاده می کنند که به همین دلیل مقدار نادرست دارد. به توجه به اینکه مقدار PC برای این دستورات به ترتیب ۰،۴،۸،۱۶،۲۴،۳۶،۴۸،۵۶،۶۰ است می توان ۴/۵ سیکل ساعت پس از رسیدن PC کدشگایی به این مقدار (یا ۲/۵ سیکل ساعت پس از رسیدن PC اجرا) مشاهده کرد که مقدار درست برای این دستورات در مقصد درست نوشته می شود.

چون هنوز ماژول حافظه را پیاده سازی نکردیم به ورودی دوم Multiplexer موجود در قسمت بازنشانی ۳۲ بیت صفر را دادیم. که مشاهده می شود برای دستور ۱۸ ام که LDR است این مقدار در رجیستر شماره ۱۱ ذخیره می شود.

جلسه چهارم: حافظه و تشخیص مخاطره

ماژول های خواسته شده در صورت آزمایش به این شکل پیاده سازی شده اند:

MEM_Stage_Reg: این ماژول حاوی ثبات های بعد از ماژول اجرا است و صرفا باید ورودی های خود را به خروجی منتقل کند. برای این هدف از یک بلوک always استفاده شده است که این کار را با لبه بالارونده clock انجام می دهد.

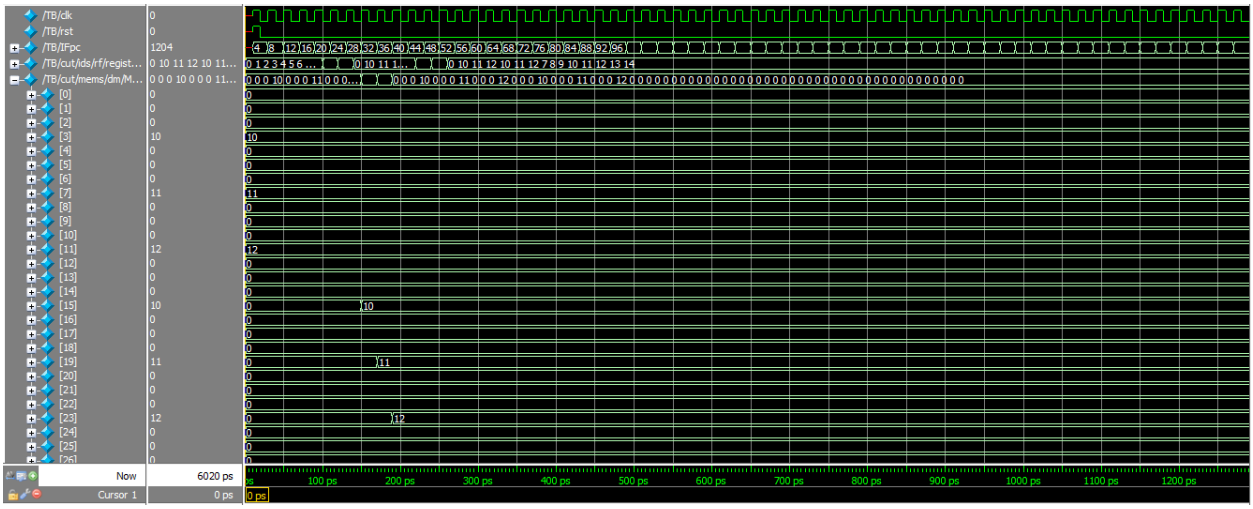
Data_Mem: ماژول حافظه مدار است. در آن آرایه ای به طول ۶۴ عنصر ۸ بیتی به عنوان حافظه تعریف شده است. این آرایه در ابتدای نمونه گیری از ماژول توسط یک بلوک initial مطابق با فایل "mem.data" که به صورت مبنای شش نوشته شده است مقداردهی می شود. عملیات خواندن از آن با توجه به سیگنال MemRead و بدون توجه به clock انجام می شود و برای عملیات نوشتن در آن از یک بلوک always همگام با له بالارونده clock استفاده شده است که در صورت یک بودن سیگنال Memwf داده ی ۳۲ بیتی ورودی را در محلی که آدرس ورودی به آن اشاره می کند یادداشت می کند.

MEM_Stage: این ماژول ۴ مخصوص قسمت حافظه پردازنده است و در آن صرفا یک نمونه از ماژئل Data_Mem نمونه گرفته شده است که توضیحات آن در بالا ارائه شد.

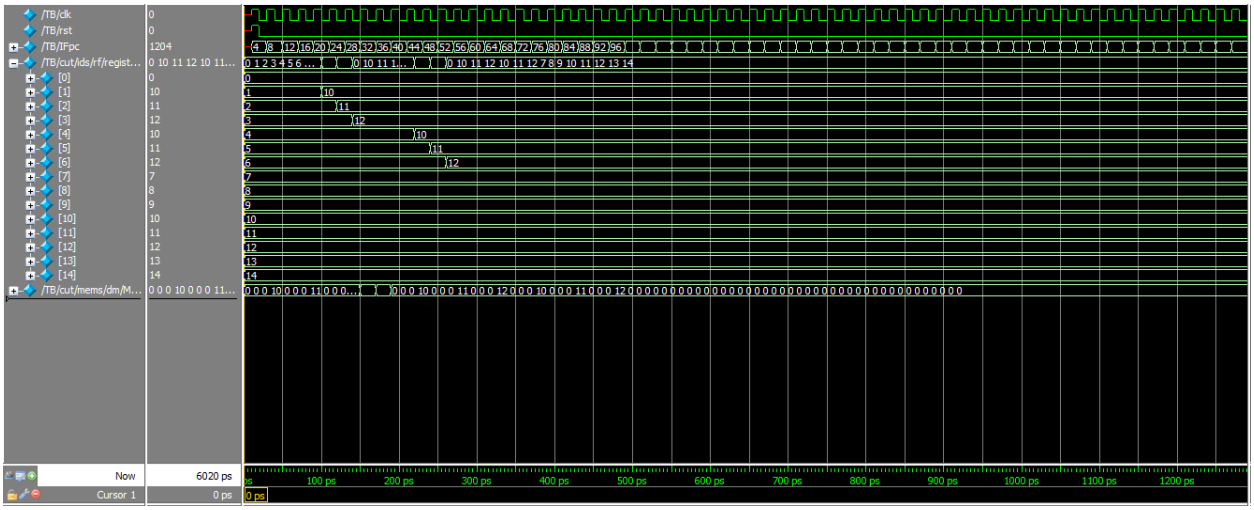
برای آزمودن حالت فعلی پردازنده (بدون hazard detection) برنامه ای نوشتیم که ابتدا سه خانه اول حافظه که قبلا آن ها را مقدار دهی کرده بودیم بخواند و مقدارشان را در ثبات های ۱ و ۲ و ۳ ذخیره کند، مقدار آن ها را به ترتیب در سه خانه بعدی حافظه بریزد و در انتها مقدار نوشته شده در سه خانه دوم حافظه را دوباره بخواند و در ثبات های ۴ و ۵ و ۶ ذخیره کند. عملکرد باقی قسمت های پردازنده در فازهای قبل مورد آزمایش قرار گرفته بود و بنابراین در این قسمت صرفا به بررسی قسمت جدید که حافظه است پرداختیم. کد این برنامه به این شکل است:

```
32'b1110_01_0_0100_1_0000_0001_010000000000    //LDR R1,[R0],1024
32'b1110_01_0_0100_1_0000_0010_010000000100    //LDR R2,[R0],1028
32'b1110_01_0_0100_1_0000_0011_010000001000    //LDR R3,[R0],1032
32'b1110_01_0_0100_0_0000_0001_010000001100    //STR R1,[R0],1036
32'b1110_01_0_0100_0_0000_0010_010000010000    //STR R2,[R0],1040
32'b1110_01_0_0100_0_0000_0011_010000010100    //STR R3,[R0],1044
32'b1110_01_0_0100_1_0000_0001_010000001100    //LDR R4,[R0],1036
32'b1110_01_0_0100_1_0000_0001_010000010000    //LDR R5,[R0],1040
32'b1110_01_0_0100_1_0000_0001_010000010100    //LDR R6,[R0],1044
```

وضعیت خانه های درگیر از حافظه در حین اجرای این برنامه به صورت زیر است:



و تصویر پایین وضعیت ثبات های عمومی پردازنده را در همین بازه زمانی نشان می دهد:



با دقت در مقادیر اولیه خانه های حافظه متوجه می شویم که عملیات های خواندن و نوشتن در حافظه به درستی انجام می شوند.

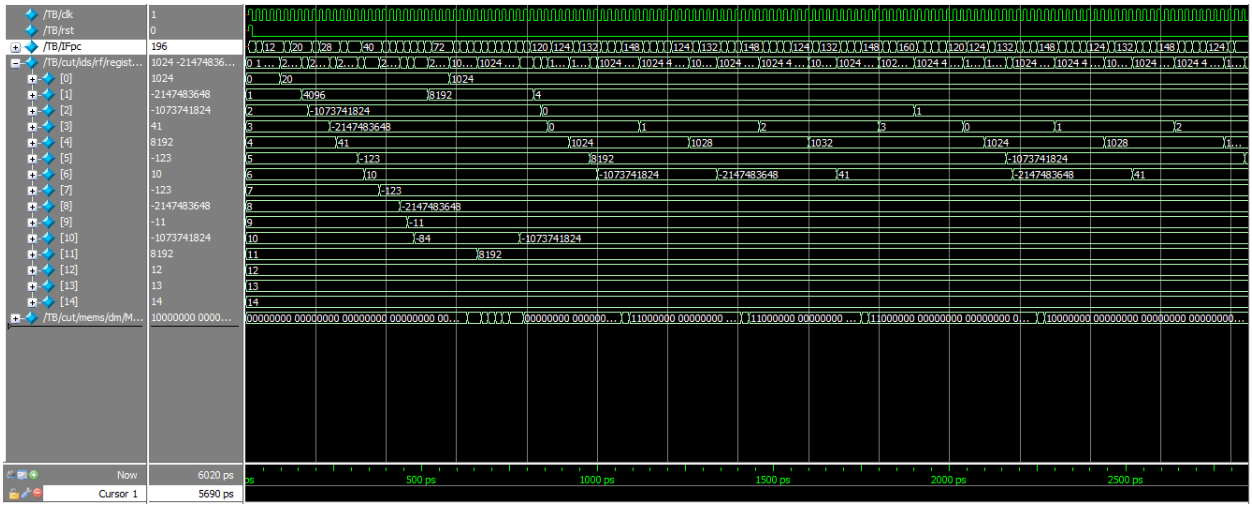
در ادامه گزارش به بررسی پیاده سازی بحث Hazard Detection می پردازیم.

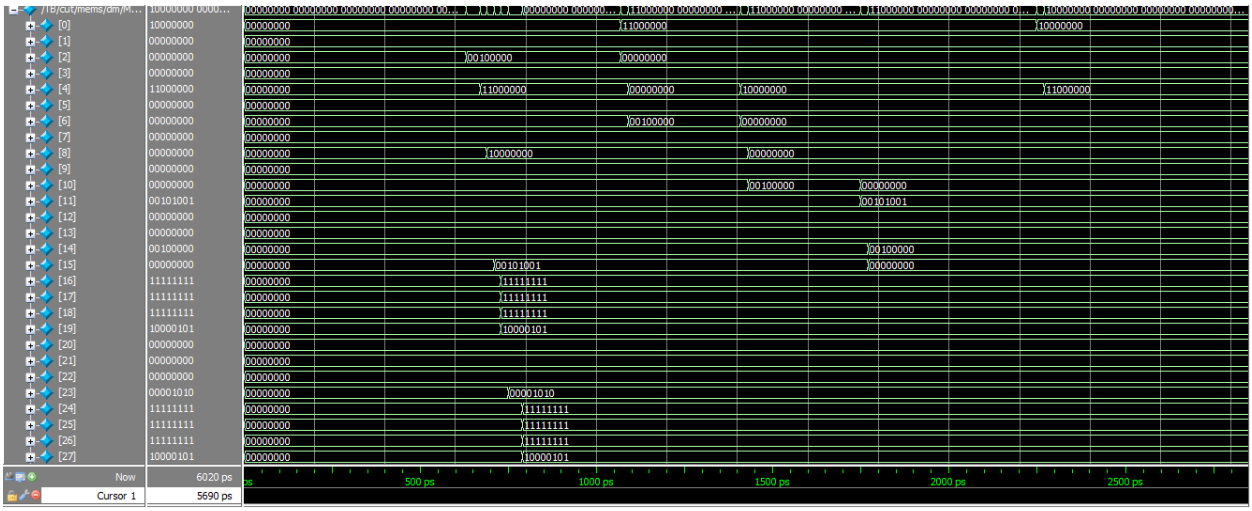
HazardDetectionUnit: در این ماژول به وسیله یک بلوک always، با توجه به دسترسی هایی که دستور در حال کدگشایی به ثبات های عمومی دارند و اینکه این ثبات ها در مراحل اجرا و حافظه در حال استفاده شدن هستند یا خیر مخاطره RAW را تشخیص می دهد.

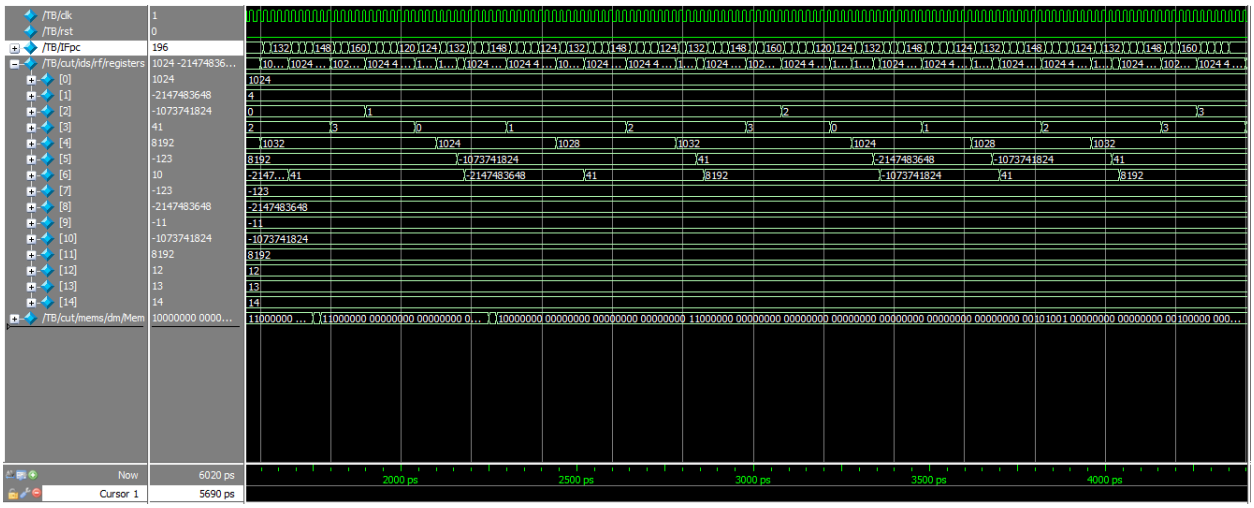
بعد از پیاده سازی این ماژول و اضافه کردن آن به مدار پردازنده به رجیسترهای بعد از ماژول کدگشایی قابلیت freeze نیز اضافه شد تا اگر مخاطره رخ داد دستورات در حال وا کشی و کدگشایی جلو نروند و همچنین دستور جدیدی واکشی نشود تا مخاطره برطرف شود.

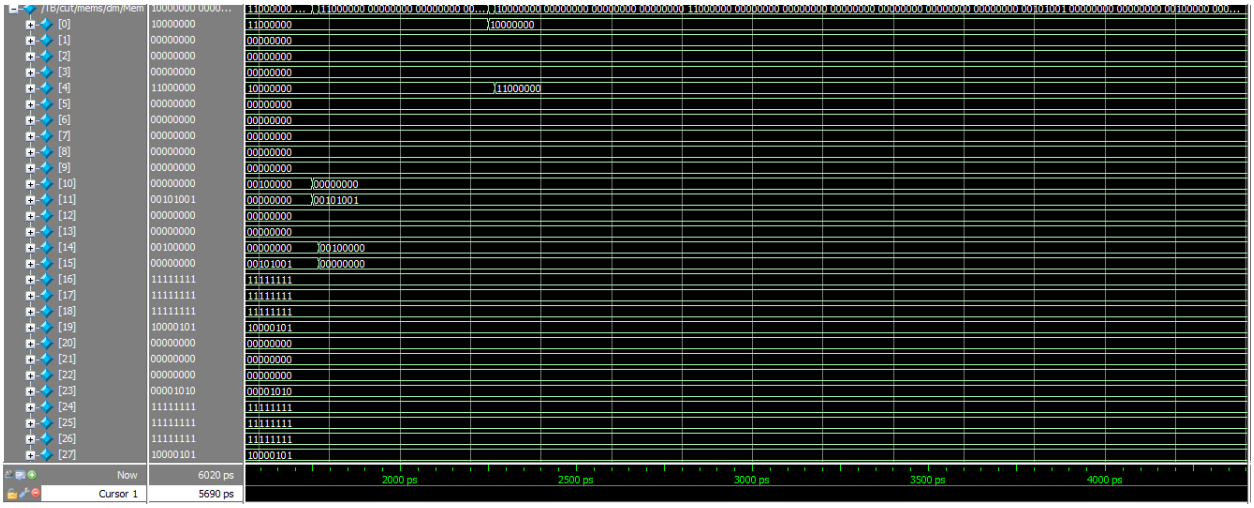
برای آزمودن ماژول های پیاده سازی شده از هرکدام یک نمونه گرفتیم و به هم متصل کردیم و عملا کار طراحی پردازنده را به اتمام رساندیم. تصویر مسیر داده پردازنده در صفحه آخر گزارش موجود است.

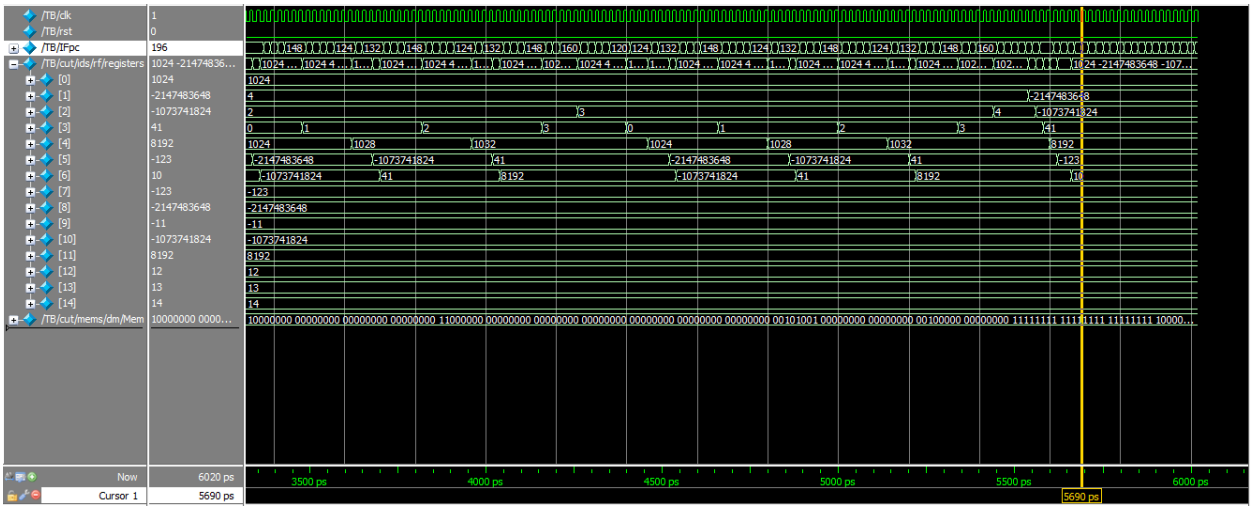
در این آزمون ابتدا پردازنده ریست می شود و سپس clock زده می شود تا دستورات برنامه محک که در RegisterFile بارگذاری شده اند خوانده و اجرا شود. در نمودارهای زیر، مقدار PC دستوری که در مرحله واکشی است (IFpc) به همراه مقادیر ثبات های عمومی و مقادیر خانه های درگیر از حافظه به ترتیب در سه بازه ی ابتدا، میانه و انتهای اجرای برنامه نمایش داده شده است.

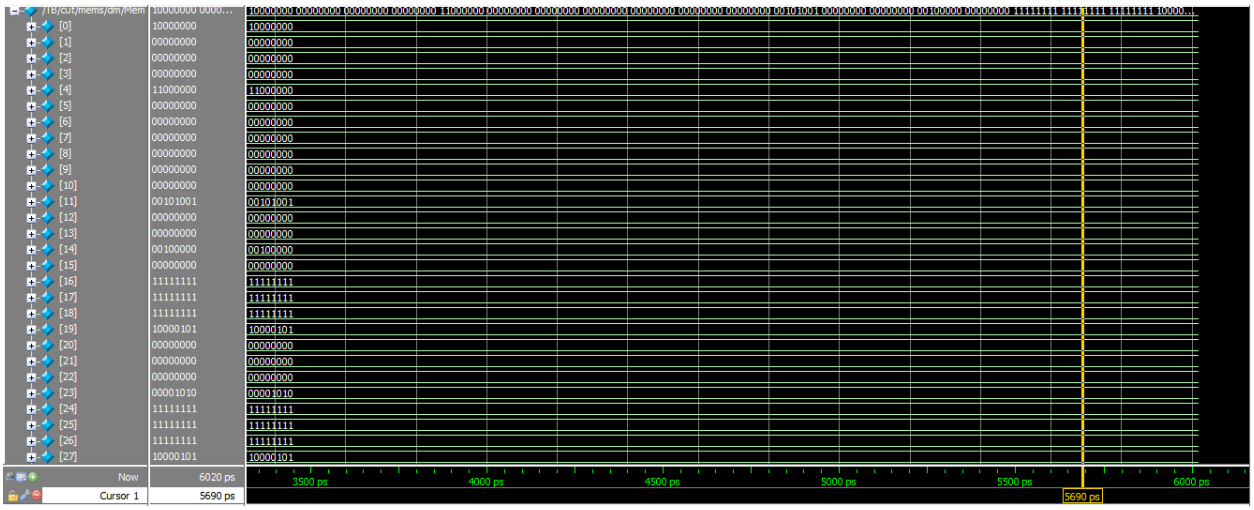





/TB/clk
/TB/rst
/TB/IFpc
/TB/cut/ds/rf/registers
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
/TB/cut/mems/dm/Mem
Now
Cursor 1
2000 ps
2500 ps
3000 ps
3500 ps
4000 ps


[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
[20]
[21]
[22]
[23]
[24]
[25]
[26]
[27]
Now
Cursor 1
2000 ps
2500 ps
3000 ps
3500 ps
4000 ps


/TB/clk
/TB/rst
/TB/IFpc
/TB/cut/ds/rf/registers
/TB/cut/mems/dm/Mem
Now
Cursor 1
3500 ps
4000 ps
4500 ps
5000 ps
5500 ps
6000 ps
5690 ps



با دقت در مقادیر ثبات ها و خانه های حافظه در هر زمان می توان متوجه شد که پردازنده به درستی تمام عملیات ها را انجام می دهد.

روال اجرای برنامه به این صورت است که تا دستور ۲۷ اجرا می شود و بعد از آن داخل یک حلقه با شمارنده R2 می افتد که این حلقه ۴ بار تکرار می شود (PC=160 دستور پرش شرطی این حلقه است که در نمودارهای بالا نیز تکرارش مشخص است). داخل این حلقه حلقه دیگری با شمارنده R3 وجود دارد که ۳ بار (در کل برنامه ۱۲ بار) اجرا می شود (PC=148 دستور پرش شرطی این حلقه است که در نمودارهای بالا نیز تکرارش مشخص است). زمان اجرای برنامه نیز برابر ۲۸۲ سیکل ساعت است؛ ۱۶۹ دستور، ۹۸ سیکل ساعت توقف و ۱۱ سیکل ساعت بعد از دستورات پرش(لحظه ای که اشاره گر به آن اشاره می کند لحظه ی اتمام اجرای دستور آخر و در لحظه ۵۶۹۰ است، ۳۰ واحد از ابتدای شروع آزمون پردازنده در حال ریست شدن است و هر سیکل ساعت ۲۰ واحد زمانی است).

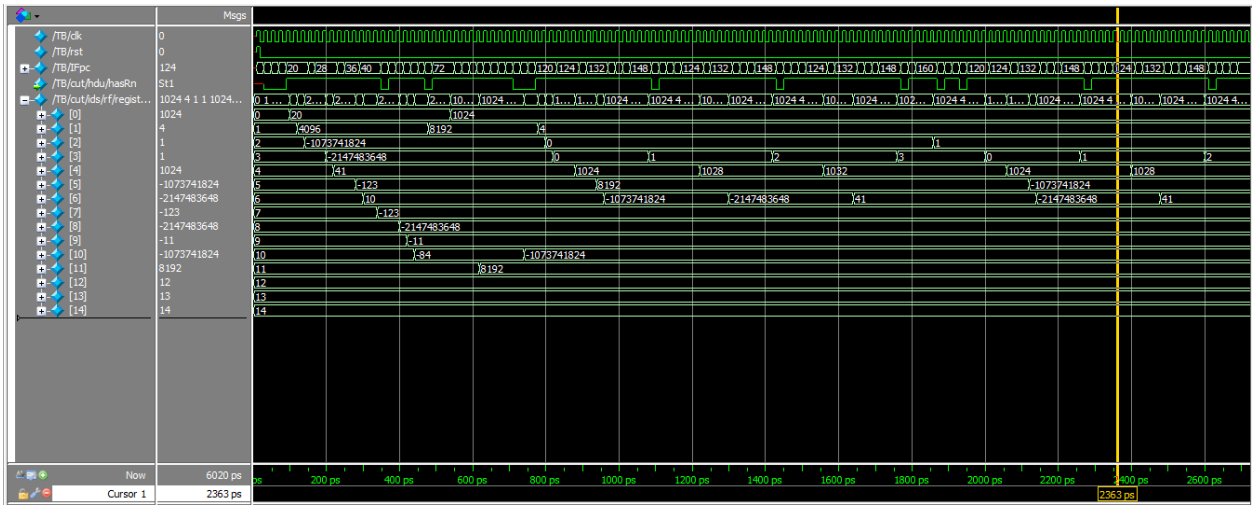
بخش امتیازی:

در دستورات MOV و MOVN چهار بیت مربوط به Rn صفر است ولی به رجیستر صفر اشاره نمی کند. به همین دلیل واحد تشخیص مخاطره ممکن است اشتباها در هنگام اجرای این دستورات اعلام مخاطره و در اجرای دستورات ایجاد وقفه کند (مانند وقفه ای که در نمودار اول بالا برای حالت IFpc=12 افتاده است).

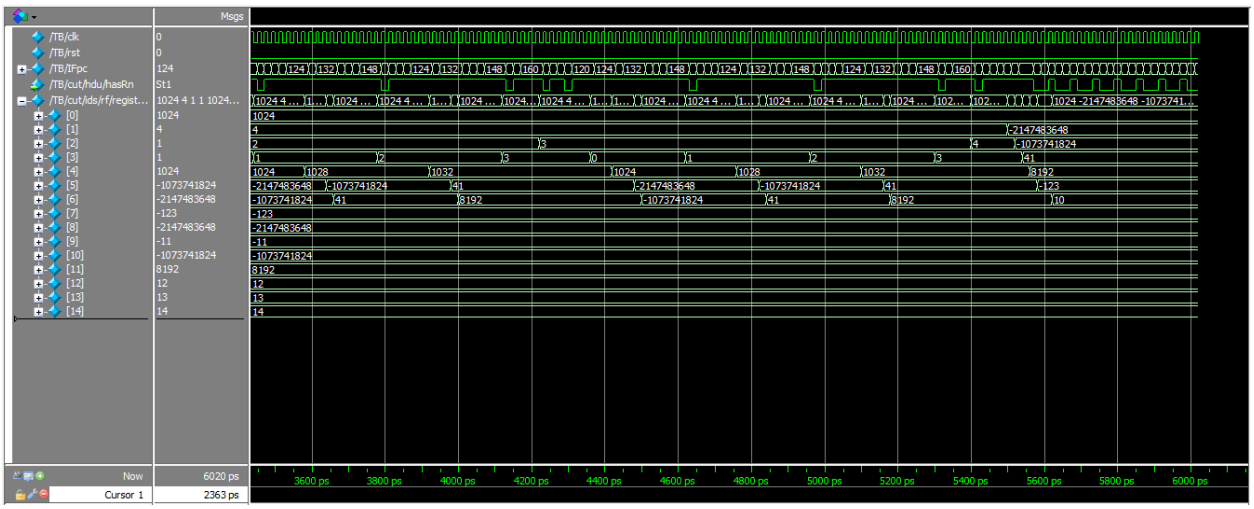
برای جلو گیری از این اتفاق، یک خروجی به خروجی های کنترلر اضافه می کنیم تا در صورتی که دستور Rn داشت برابر یک شود و در غیر این صورت (که می شود همان دو دستور گفته شده) برابر صفر. باید توجه کرد که این سیگنال نباید مانند باقی خروجی های کنترلر وارد Multiplexer شود تا در صورت وقوع مخاطره به جای آن صفر از کنترلر خارج شود، چرا که این سیگنال خودش تشخیص دهنده مخاطره است. این خروجی را نیز به عنوان یک ورودی به واحد تشخیص مخاطره می دهیم تا هنگام بررسی Rn برای این دستور ها مخاطره تشخیص ندهد. بنابراین جدول سیگنال های کنترلی پردازنده به این صورت خواهد شد:

ARM Controller													
Instruction		Description	Inputs			Control Signals							
			mode	OP-Code	s	EXE_CMD	mem_read	mem_write	WB_Enable	B	S	hasRn
1	MOV	Move	00	1101	s	0001	0	0	1	0	s	0
2	MVN	Move NOT	00	1111	s	1001	0	0	1	0	s	0
3	ADD	Add	00	0100	s	0010	0	0	1	0	s	1
4	ADC	Add with Carry	00	0101	s	0011	0	0	1	0	s	1
5	SUB	Subtraction	00	0010	s	0100	0	0	1	0	s	1
6	SBC	Subtract with Carry	00	0110	s	0101	0	0	1	0	s	1
7	AND	And	00	0000	s	0110	0	0	1	0	s	1
8	ORR	Or	00	1100	s	0111	0	0	1	0	s	1
9	EOR	Exclusive Or	00	0001	s	1000	0	0	1	0	s	1
10	CMP	Compare	00	1010	1	0100	0	0	0	0	1	1
11	TST	Test	00	1000	1	0110	0	0	0	0	1	1
12	LDR	Load Register	01	0100	1	0010	1	0	1	0	0	1
13	STR	Store Register	01	0100	0	0010	0	1	0	0	0	1
14	B	Branch	10	---	---	---	0	0	0	1	0	0

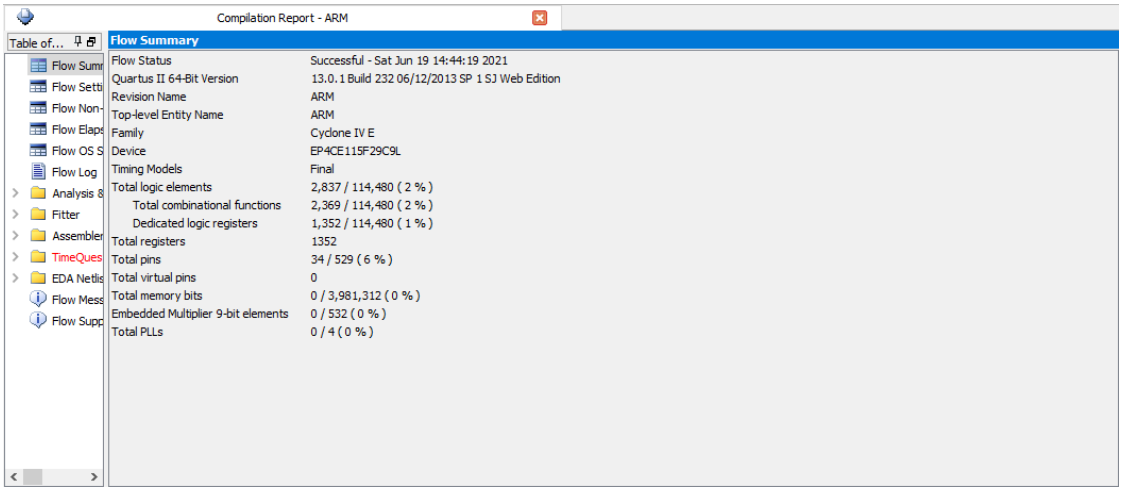
بعد از پیاده سازی و اجرا مشاهده می شود که وقفه ابتدای برنامه دیگر اتفاق نمی افتد (جاهای دیگری که دستورات MOV و MOVN استفاده می شوند قبلا هم مخاطره نداشتیم) و زمان اجرای آزمون ۲۸۱ سیکل ساعت می شود، ۱۶۹ دستور، ۹۷ سیکل ساعت توقف و ۱۱ سیکل ساعت بعد از دستورات پرش (جزئیات نحوه اجرای دستورات در فایل 1_ARM_PipeLine.xlsx موجود است):



و همچنین با توجه به مقدار نهایی ثبات های عمومی می توان تحقیق کرد که برنامه درست اجرا شده است:



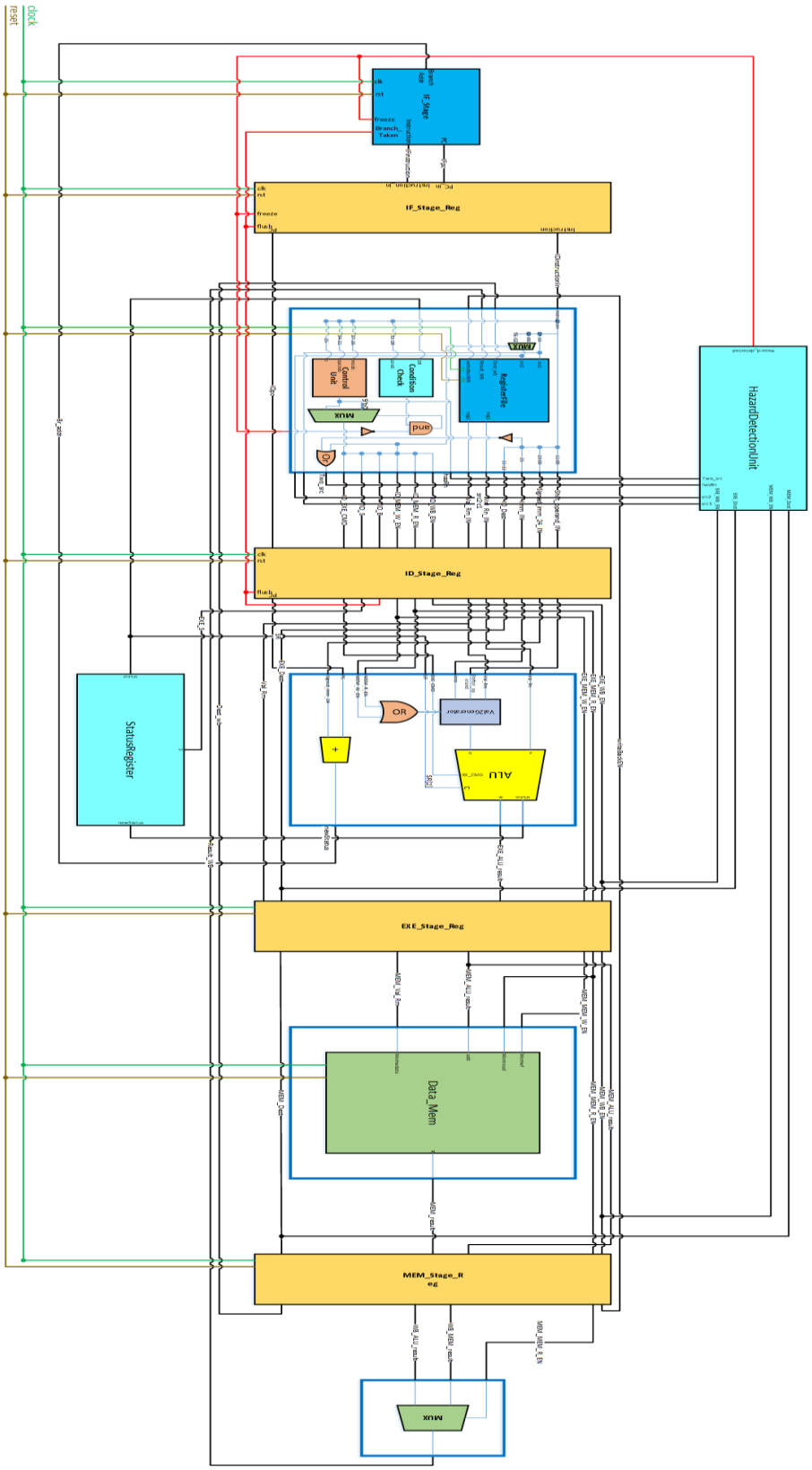
این مدار در نرم افزار Quartus سنتز شد که گزارش سنتز آن به این صورت بود:



مشاهده می شود که تعداد کل قطعات منطقی استفاده شده برای سنتز این مدار ۲۸۳۷ عدد است.

جدول نهایی گزارش فاز یک	
Total Logic Elements	2837
Total Combinational Functions	2369
Dedicated Local Registers	1352
Instructions	169
Clock Cycles	281
CPI	0.601

تصویر مسیر داده پردازنده ARM (فایل حاوی نمودار: 1_ARM_Datapath.vsdx):



فاز دوم: اضافه کردن تکنیک ارسال به جلو

در این بخش به پیاده سازی واحد ارسال به جلو می پردازیم.

در بعضی مواقعی که مخاطره رخ می دهد، مقداری که نیاز داریم تا بازنشانی شده باشد در جای دیگری از پردازنده تولید شده است ولی به مرحله بازنشانی نرسیده است. در این موارد می توانیم با ارسال این مقدار به بخش اجرا از آن استفاده کنیم و دیگی نیازی به متوقف کردن پردازنده نباشد.

برای این کار در مسیر داده پردازنده قبل از ماژول قسمت اجرا (EXE_Stage) چهار عدد Multiplexer دوتایی ۳۲ بیت قرار دادیم تا داده هایی که به این مرحله وارد می شود را انتخاب کند. برای هرکدام از Val1 و Val2، Multiplexer اول بین دو مقدار پیشفرض (حالت بدون مخاطره که برای Val1 از ثبات ها می آمد و برای Val2 از ماژول Val2Generator) و مقدار ارسال شده از پورت متناظر خروجی ALU در ثبات های بعد از مرحله اجرا (EXE_Stage_Reg) انتخاب می کند و Multiplexer دوم بین مقدار انتخاب شده Multiplexer اول و مقدار ارسال شده از Multiplexer بخش بازنشانی (Result_WB). سیگنال های انتخاب کننده این Multiplexerها توسط ماژول ForwardingUnit تولید می شود. تصویر مسیر داده پردازنده در صفحه آخر گزارش کار موجود است (برای مشاهده بهتر جزئیات نمودار فایل DataPath.vsdx نیز ضمیمه شده است).

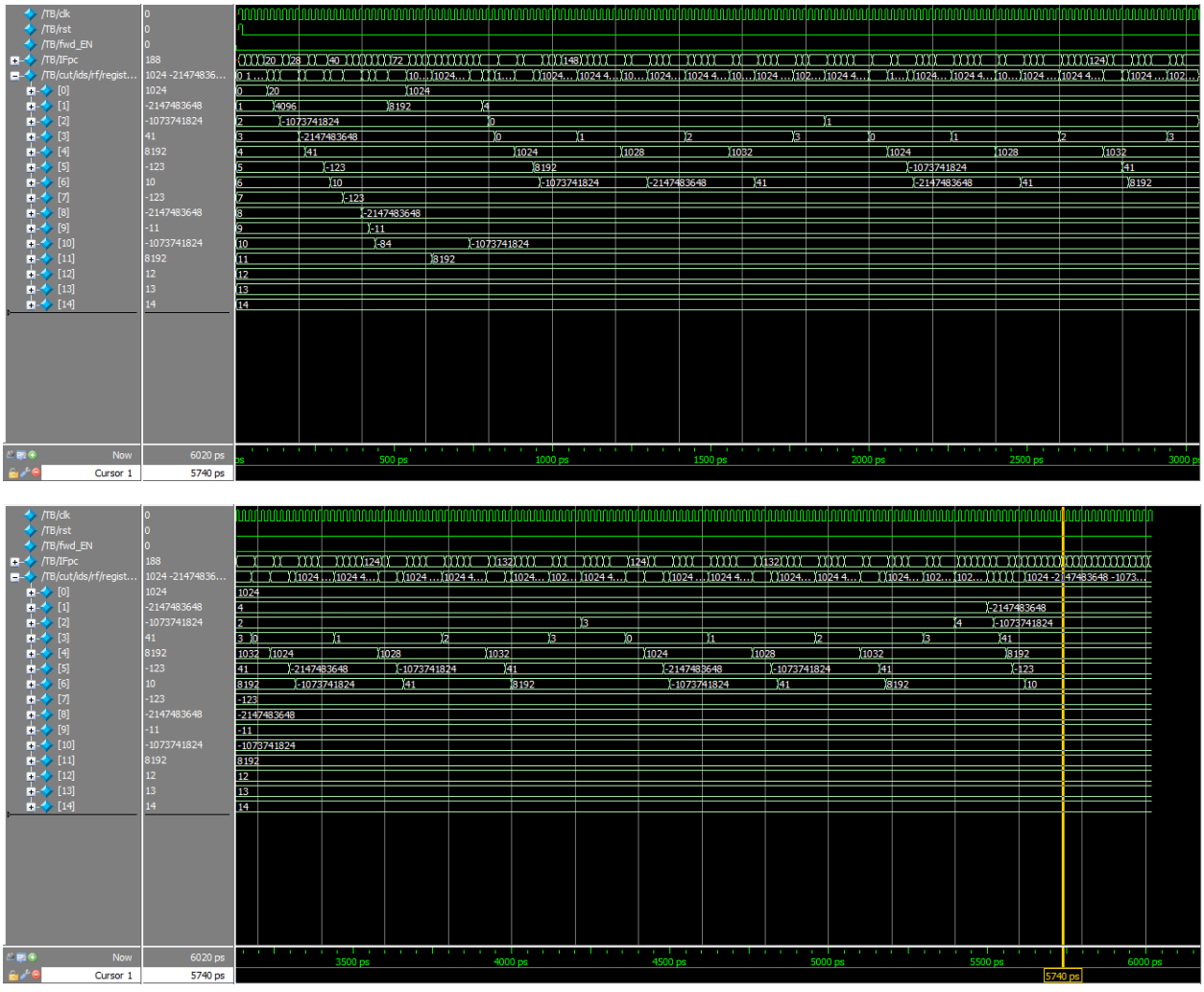
ماژول ForwardingUnit یک سیگنال ورودی به نام fwd_EN دارد که اگر برابر یک باشد اتفاقات توضیح داده شده در پایین رخ می دهد و اگر صفر باشد انگار اصلا ماژول ارسال به جلو نداریم (تمام سیگنال های خروجی برابر صفر و سیگنال stallNeaded برابر یک می شوند). این ماژول را با استفاده از یک بلوک always به صورتی پیاده سازی کردیم که آدرس ثبات های مورد نیاز در قسمت کدگشایی (src1 و src2)، آدرس ثبات مقصد در قسمت های اجرا و حافظه (EXE_Dest و MEM_Dest)، سیگنال هایی که نشان دهنده وجود داشتن src1 (hasRn) و src2 (Two_Src) هستند، سیگنال های بازنشانی دستوراتی که در مرحله اجرا و حافظه هستند (EXE_WB_EN و MEM_WB_EN) و سیگنال کنترلی خواندن از حافظه برای دستور قبلی در خط لوله یعنی دستوری که در مرحله اجراست (EXE_MEM_R_EN) را به عنوان ورودی بگیرد و با تولید کردن ۴ سیگنال انتخاب کننده بین Multiplexerها (fRnSEXE، frnSMEM، fRmSEXE و frmSMEM) تعیین کند کدام مقدار باید وارد مرحله اجرا شود.

همچنین این ماژول یک خروجی دیگر هم دارد که تعیین می کند با استفاده از تکنیک ارسال به جلو هنوز نیاز به توقف داریم یا خیر (stallNeaded). این سیگنال بعد از خروج از ماژول ForwardingUnit با خروجی Hazard_Detected ماژول HazardDetectionUnit به عنوان ورودی به یک گیت And می روند و خروجی این گیت سیگنال freeze را تولید می کند که باعث توقف پردازنده می شود (طبیعتا توقف تنها زمانی اتفاق می افتد که هم مخاطره تشخیص داده شده باشد و هم مخاطره تشخیص داده شده با تکنیک ارسال به جلو قابل برطرف کردن نباشد). سیگنال stallNeaded تنها در حالتی فعال می شود که دستور قبلی LDR باشد (که این حالت با توجه به ورودی EXE_MEM_R_EN تشخیص داده می شود) و محتوای ثبات مقصد آن را در دستور فعلی نیاز داشته باشیم. در این حالت چون دستور LDR هنوز به در

مرحله اجراست مقدار ثبات مقصد مشخص نیست. در نتیجه به یک سیکل ساعت توقف نیاز است تا دستور به مرحله حافظه برسد و مقدار مورد نظر را بخواند و بعد از آن می توانیم آن را به بخش اجرا ارسال کنیم.

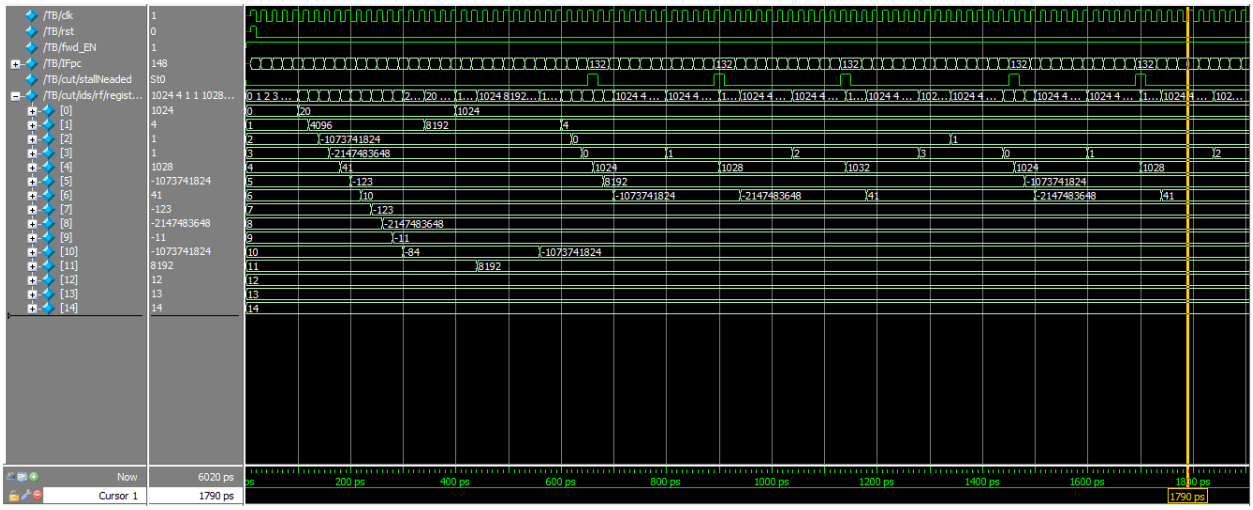
بعد از انجام کارهای گفته شده، برنامه محک را یک بار با fwd_EN برابر صفر و بار دیگر با fwd_EN برابر یک روی پردازنده اجرا کردیم که نتایج آن به صورت زیر است:

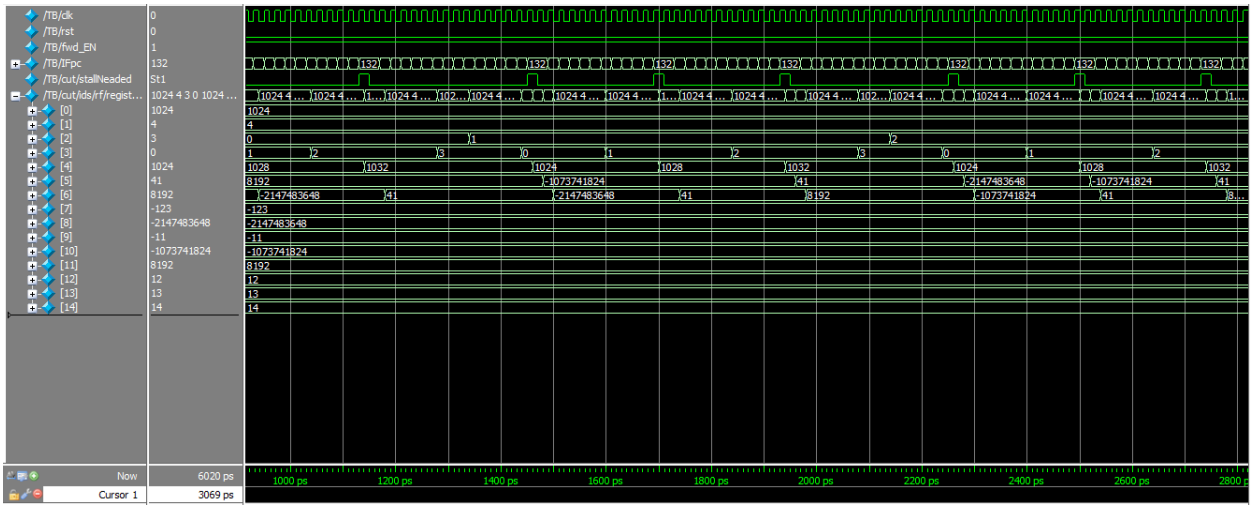
fwd_EN برابر صفر:

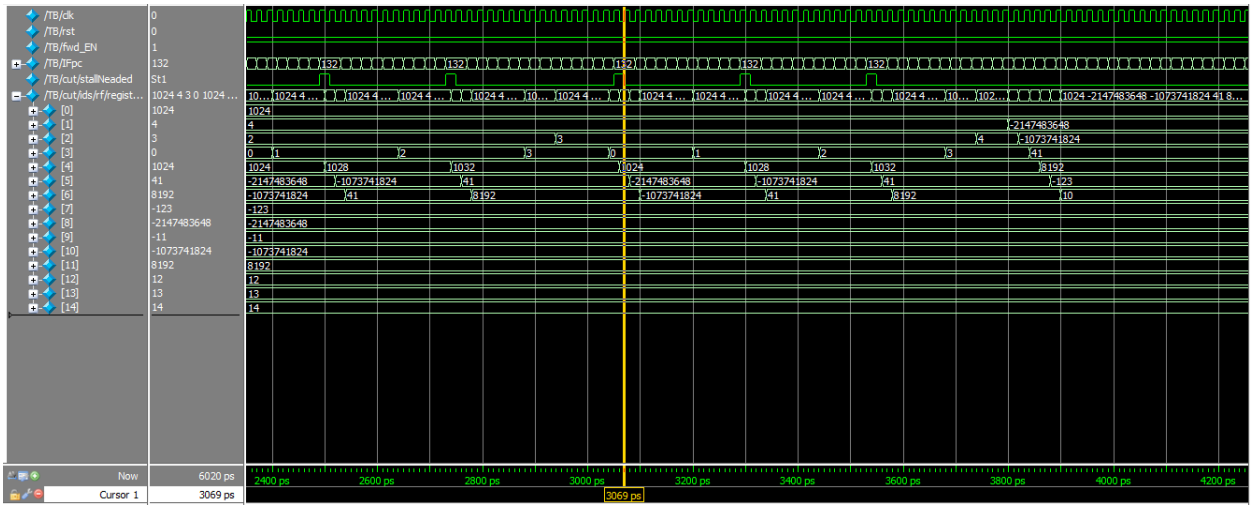


مشاهده می شود که این برنامه مانند برنامه فاز قبل در ۲۸۱ سیکل ساعت اجرا می شود. همچنین با توجه به مقادیر نهایی ثبات ها می توان فهمید که برنامه درست اجرا شده است.

fwd_EN برابر یک (جزئیات نحوه اجرای دستورات در فایل 2_FWD_PipeLine.xlsx موجود است):







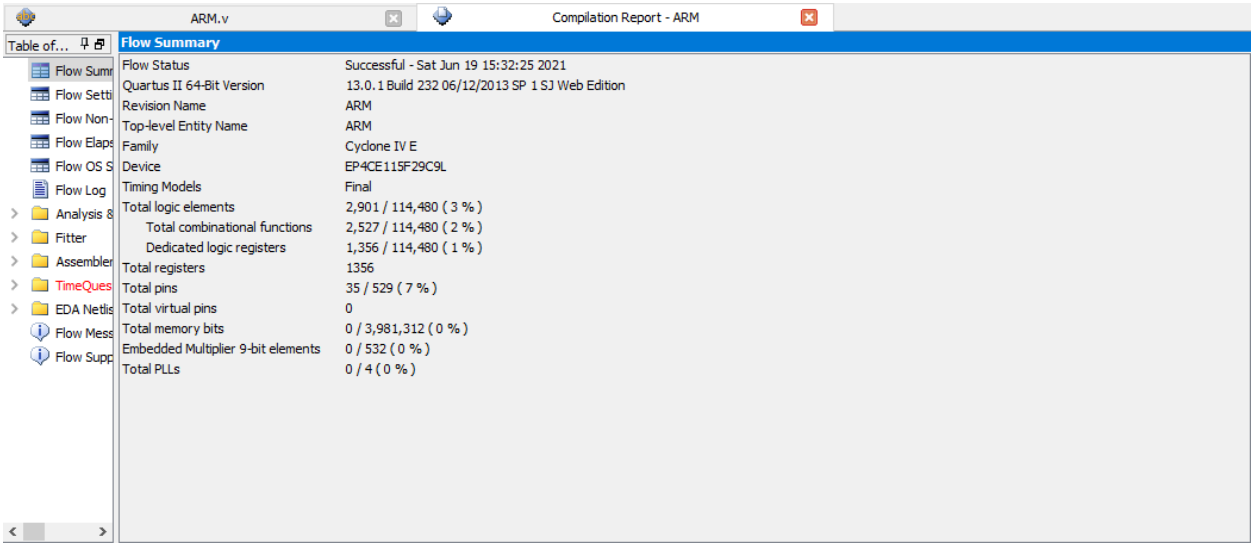
با توجه به مقادیر نهایی ثبات ها می توان فهمید که برنامه درست اجرا شده است.

اگر به مقدار IFpc در تصاویر بالا دقت کنیم متوجه می شویم که پردازنده در حین اجرای برنامه صرفا ۱۲ سیکل ساعت برای IFpc=132 توقف داشته است. این توقف ها هنگام اجرای خط ۳۲ برنامه است که باید مقادیر ثبات های R5 و R6 با هم مقایسه شوند در حالی که مقدار ثبات R6 باید در دستور قبلی که LDR است خوانده می شد. بنابراین مطابق توضیحات بالا به یک سیکل ساعت توقف در این دستور احتیاج است. همان طور که مشاهده می شود در این قسمت ها سیگنال stallNeaded برابر یک است (نحوه اجرای برنامه محک توسط پردازنده به تفکیک سیکل ساعت در فایل 2_FWD_PipeLine.xlsx رسم شده است).

اجرای برنامه با استفاده از تکنیک ارسال به جلو ۱۹۶ سیکل ساعت طول می کشد؛ ۱۶۹ دستور، ۱۲ سیکل ساعت توقف و ۱۱ سیکل ساعت بعد از دستورات پرش. تکنیک ارسال به جلو ۸۵ سیکل ساعت از توقف های به وجود آمده را کاهش می دهد. بنابراین درصد تسریع ایجاد شده توسط تکنینک ارسال به جلو برابر است با:

$$SpeedUp = \frac{281}{196} - 1 \cong 43\%$$

پردازنده بعد از اضافه کردن ماژول ارسال به جلو مجددا توسط برنامه Quartus روی برد EP4CE115F29C9L از خانواده Cyclone IV E سنتز شد که نتیجه آن به صورت زیر است:



تعداد کل قطعات منطقی استفاده شده در این حالت برابر ۲۹۰۱ قطعه است که نسبت به حالت بدون تکنیک ارسال به جلو ۶۴ قطعه بیشتر شده است (در این حالت ۲۸۳۷ قطعه به کار رفته بود). بنابراین درصد افزایش قطعات استفاده شده برابر است با:

$$Cost\ Increase = \frac{2901}{2837} - 1 \cong 2.26\%$$

نسبت افزایش کارایی به افزایش هزینه به این صورت محاسبه می شود:

$$\frac{SpeedUp}{Cost\ Increase} = \frac{\frac{281}{196} - 1}{\frac{2901}{2837} - 1} = \frac{85 \times 2837}{64 \times 196} = \frac{241145}{12544} \cong 19.22$$

پس حدودا ۱۹ برابر افزایش هزینه ای که داریم افزایش کارایی خواهیم داشت.

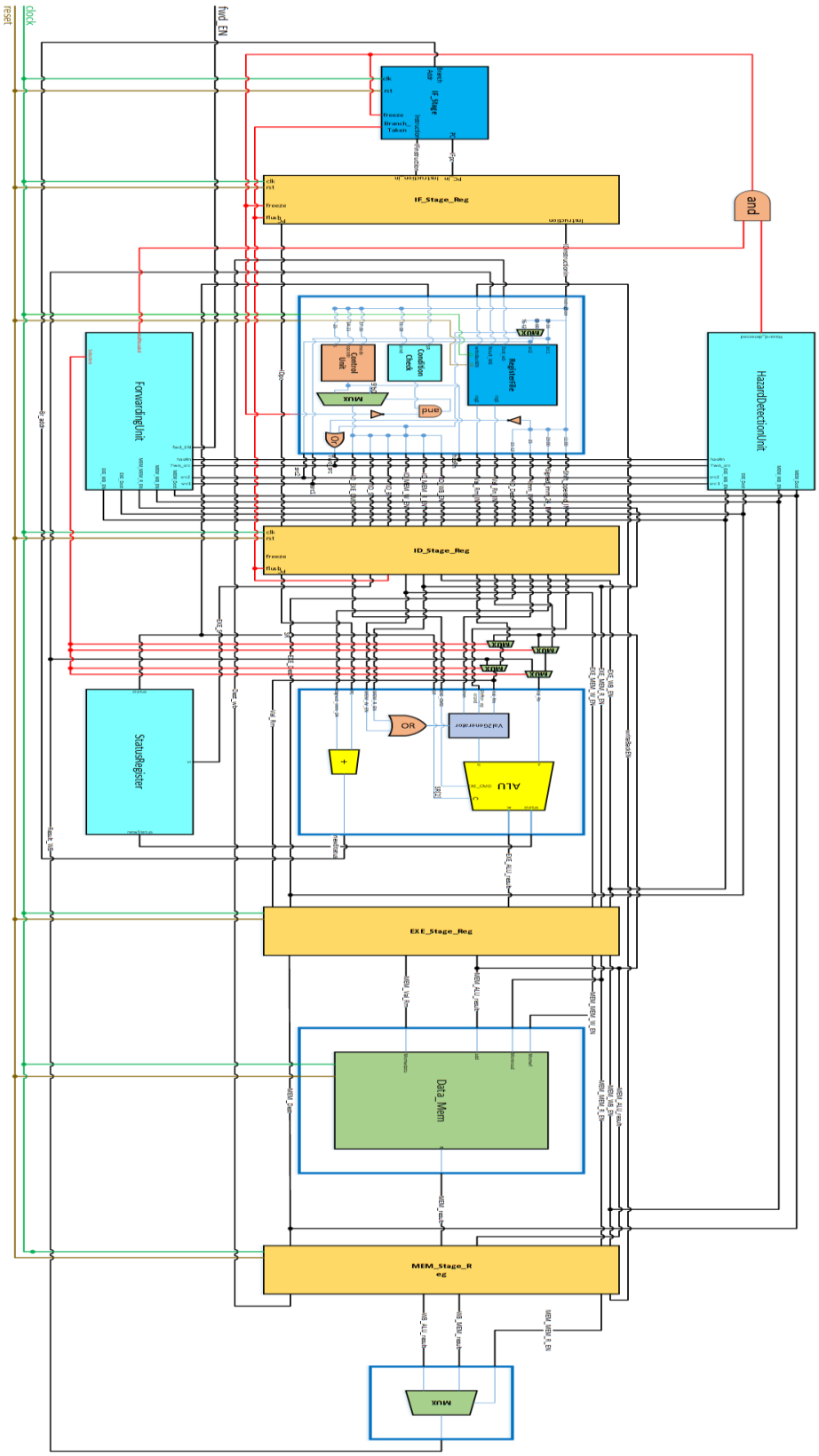
جدول نهایی گزارش فاز دو	
Total Logic Elements	2901
Total Combinational Functions	2527
Dedicated Local Registers	1356
Instructions	169
Clock Cycles	196
CPI	0.862

بخش امتیازی:

یکی از راه هایی که می توان به جای تکنیک ارسال به جلو از آن برای رفع توقف ها استفاده کرد این است که معماری پردازنده را به گونه ای تغییر دهیم که بخش کدگشایی و بخش اجرا یک مرحله فرض شوند و همچنین در صورتی که مقدار محاسبه شده در ALU مقداری بود که باید در ثبات های عمومی بازنشانی می شد همان جا این امر انجام شود. برای این کار باید ثبات هایی که میان دو ماژول کدگشایی و اجرا هستند برداشته شوند.

بخش اجرا صرفا ترکیبی اجرا می شود و با clock هماهنگ نیست، و از طرفی ثبات های عمومی و ثبات وضعیت برخلاف باقی ثبات ها و بخش حافظه هر دو با لبه پایین رونده clock مقدار خود را به روز می کنند. بنابراین با این روش مقادیر مورد نیاز برای دستورهای بعدی تا قبل از رسیدن آن دستورات به مرحله کدگشایی و اجرا (که لبه بالا رونده بعدی clock است) در ثبات ذخیره شده اند و ثبات وضعیت نیز به روز شده است و در نتیجه امکان اجرای دستور بدون توقف وجود دارد. ولی در این روش نیز برای دستوراتی که بلافاصله بعد از دستور LDR به ثبات مقصد آن احتیاج دارند به یک سیکل ساعت توقف احتیاج داریم.

مسیر داده پردازنده ARM با واحد ارسال به جلو (فایل حاوی نمودار: 2_FWD_Datapath.vsdx):

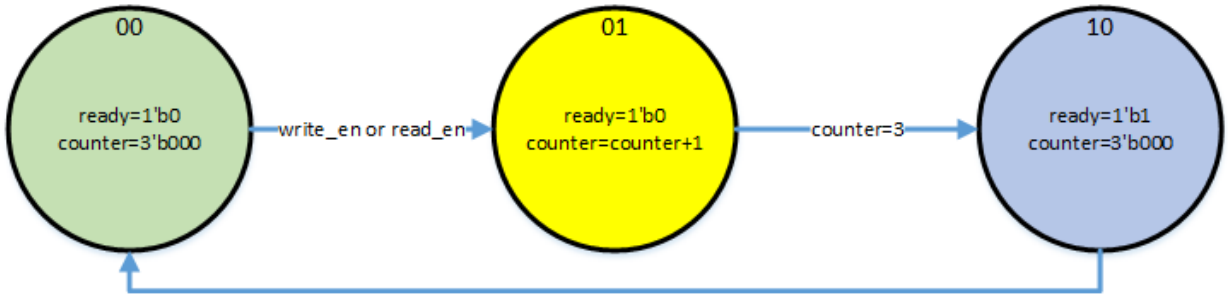


فاز سوم: استفاده از SRAM به عنوان حافظه داده

در این فاز، به جای حافظه داخلی استفاده شده در فازهای قبل از یک حافظه SRAM که از بیرون به پردازنده متصل می شود استفاده می کنیم.

برای این امر نیاز است تا این حافظه به صورت جدا به گونه ای طراحی شود که با قطعه دیگری به عنوان کنترل کننده SRAM در ارتباط باشد. کد این حافظه به صورت آماده تحویل داده شده است و ما در این آزمایش به پیاده سازی SRAM_Controller می پردازیم.

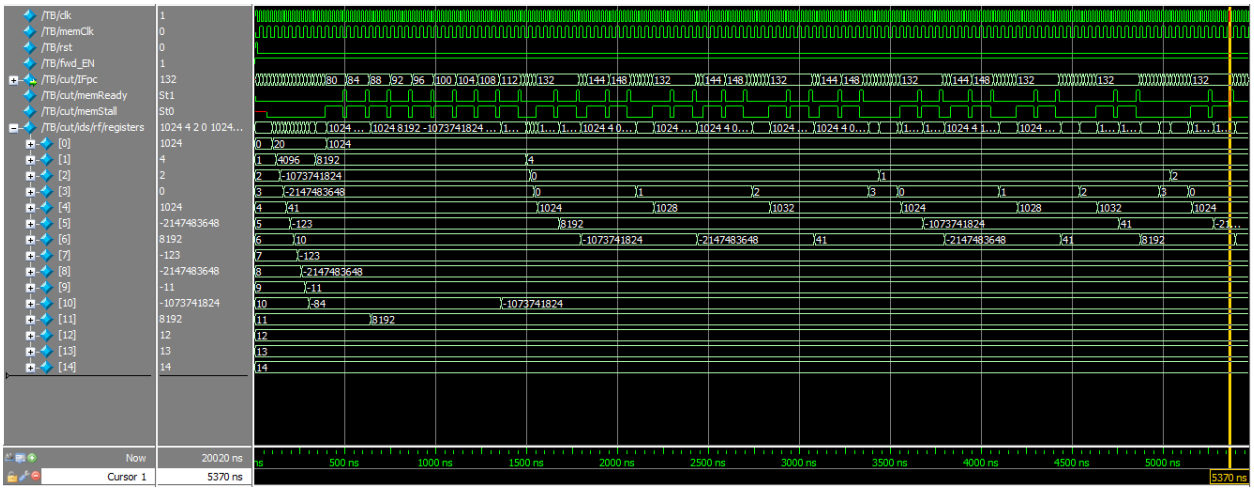
این ماژول به شکل یک ماشین حالت با سه حالت طراحی شده است. تغییر حالت با رسیدن لبه بالارونده Clock انجام می شود. حالت صفر حالت Idle است که در آن ماژول کاری انجام نمی دهد و سیگنال ready صفر است. وقتی یکی از سیگنال های read_en یا write_en فعال شود (یعنی یک دستور که با حافظه کار دارد به مرحله حافظه برسد) ماشین از حالت صفر به حالت یک می رود. در این حالت سیگنال ready صفر باقی می ماند و شمارنده ای شروع به شمردن سیکل های ساعت می کند. پس از گذشت ۴ سیکل ساعت ماشین از این حالت به حالت دو می رود (کل فرآیند حافظه ۶ سیکل ساعت زمان نیاز دارد، که SRAM_Controller در سیکل اول در حالت صفر، در سیکل های دوم تا پنجم در حالت یک و در سیکل ششم در حالت دو است) و سیگنال ready برابر یک می شود. با رسیدن لبه بالارونده Clock ماشین مجددا به حالت صفر برمی گردد و منتظر رسیدن دستور بعدی می شود. نمودار حالت های ماژول SRAM_Controller به شکل زیر است (فایل حاوی نمودار: SRAM_Ctrl.vsdx _ SRM_3):

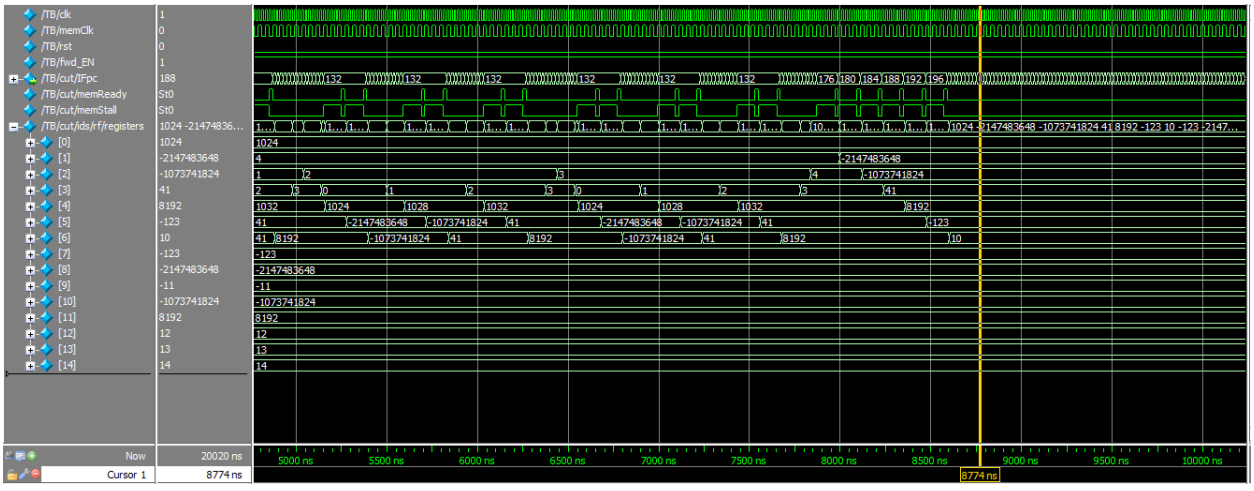


در کنار ماژول SRAM_Controller، داخل پردازنده سیگنال جدیدی با استفاده از یک گیت OR که سیگنال های MEM_MEM_W_EN و MEM_MEM_R_EN ورودی های آن هستند ساخته می شود تا نشان دهد دستوری که در مرحله حافظه است با حافظه کار دارد. سیگنال دیگری نیز تولید می شود که برابر نقیض سیگنال ready است تا در صورتی که برابر یک بود متوجه شویم که کار با حافظه تمام نشده است. این دو سیگنال با هم AND می شوند و سیگنال جدیدی به نام memStall را تولید می کنند که نشان دهنده این است که دستور موجود در مرحله حافظه با حافظه کار دارد و کار آن نیز تمام نشده است و در نتیجه نیاز به ایجاد توقف در پردازنده داریم. این سیگنال با سیگنال مربوط به توقف که توسط ماژول های تشخیص مخاطره و ارسال به جلو تولید می شد وارد یک گیت OR می شوند و خروجی آن سیگنال freeze را تولید می کند که نشان دهنده این است که باید در پردازنده وقفه ایجاد شود.

در فازهای قبلی از میان ثبات هایی که میان بخش های مختلف پردازنده بودند فقط ثبات های بعد از مراحل واکشی و کدگشایی قابلیت freeze داشتند. و سیگنال freeze موجود در پردازنده به عنوان ورودی به آن ها داده می شد. در این فاز این قابلیت به ثبات های بعد از مرحله اجرا و حافظه نیز اضافه شده است، ولی سیگنال memStall به عنوان ورودی به آن ها داده می شود. چرا که این ثبات ها نباید هنگام وقوع مخاطره متوقف شوند و باید داده ها را از خود عبور دهند.

برای آزمودن پردازنده بعد از اضافه کردن حافظه SRAM، برنامه محک روی آن اجرا شد که نتیجه اجرا به صورت زیر است (جزئیات نحوه اجرای دستورات در فایل 3_SRM_PipeLine.xlsx موجود است). در طول اجرای برنامه سیگنال fwd_EN برابر یک است و از تکنیک ارسال به جلو برای تسریع برنامه استفاده می شود.



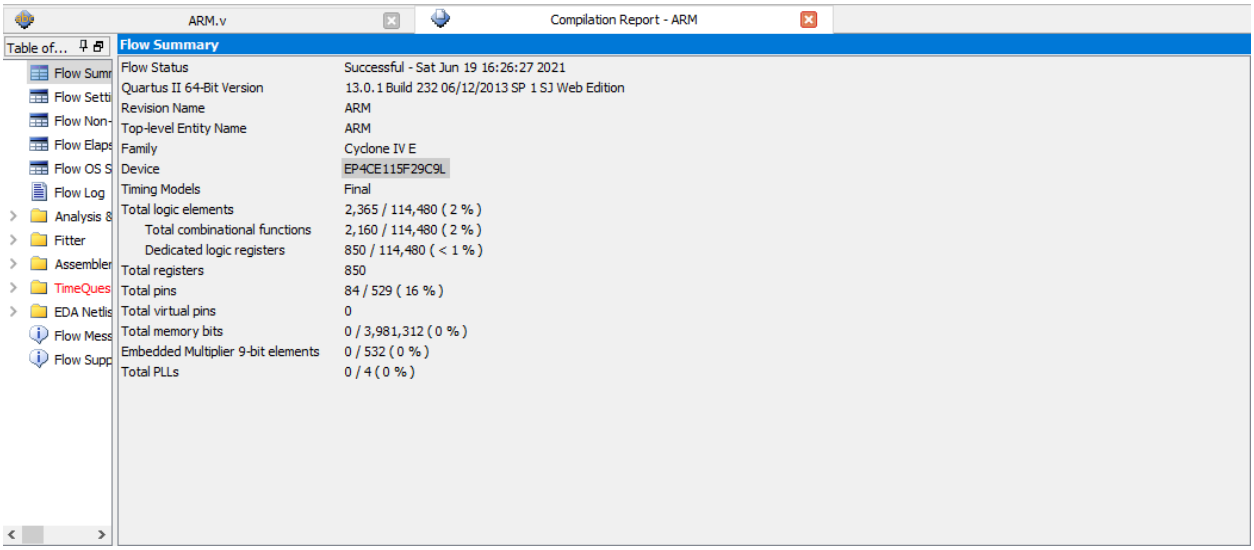


با توجه به مقادیر نهایی ثبات های عمومی می توان متوجه شد که پردازنده به درستی کار می کند.

اجرای این دستورات در این آزمون ۴۳۱ سیکل ساعت طول می کشد. تعداد کل دستورات اجرا شده همانند بخش های قبل ۱۶۹ دستور است که از این ۱۶۹ دستور، ۴۷ دستور با حافظه کار دارند. تعداد سیکل اجرای قسمت حافظه برای این دستورات ۶ سیکل است که ۵ سیکل بیشتر از حالت قبل است.

بنابراین باید ۲۳۵ سیکل ساعت اجرای برنامه بیشتر طول بکشد که همین اتفاق نیز می افتد (در حالت فعال بودن واحد ارسال به جلو برنامه در ۱۹۶ سیکل ساعت اجرا می شد که این عدد به ۴۳۱ افزایش پیدا کرده، یعنی دقیقا ۲۳۵ سیکل ساعت. جزئیات نحوه اجرای دستورات در فایل 3_SRM_PipeLine.xlsx موجود است).

پس از اطمینان از صحت عملکرد پردازنده، مدار روی برد EP4CE115F29C9L از خانواده Cyclone IV E سنتز شد که نتیج سنتز آن به صورت زیر است:



تعداد قطعات منطقی استفاده شده در سنتز مدار برابر ۲۳۶۵ قطعه است که نسبت به حالت استفاده از حافظه داخلی ۵۳۶ واحد کمتر است (در آن حالت ۲۹۰۱ قطعه استفاده می شد). یعنی بالغ بر ۱۸ درصد در تعداد قطعات صرفه جویی شده است.

$$\frac{2901 - 2365}{2901} = \frac{536}{2901} \cong 18.49\%$$

با وجود این که کاهش هزینه محسوسی در استفاده از این روش وجود دارد ولی به دلیل کند بودن حافظه خارجی سرعت پردازنده بسیار کاهش می یابد.

$$\frac{\frac{431}{169}}{\frac{196}{169}} - 1 = \frac{235}{196} \cong 119.90\%$$

جدول نهایی گزارش فاز سه	
Total Logic Elements	2365
Total Combinational Functions	2160
Dedicated Local Registers	850
Instructions	169
Clock Cycles	431
CPI	0.392

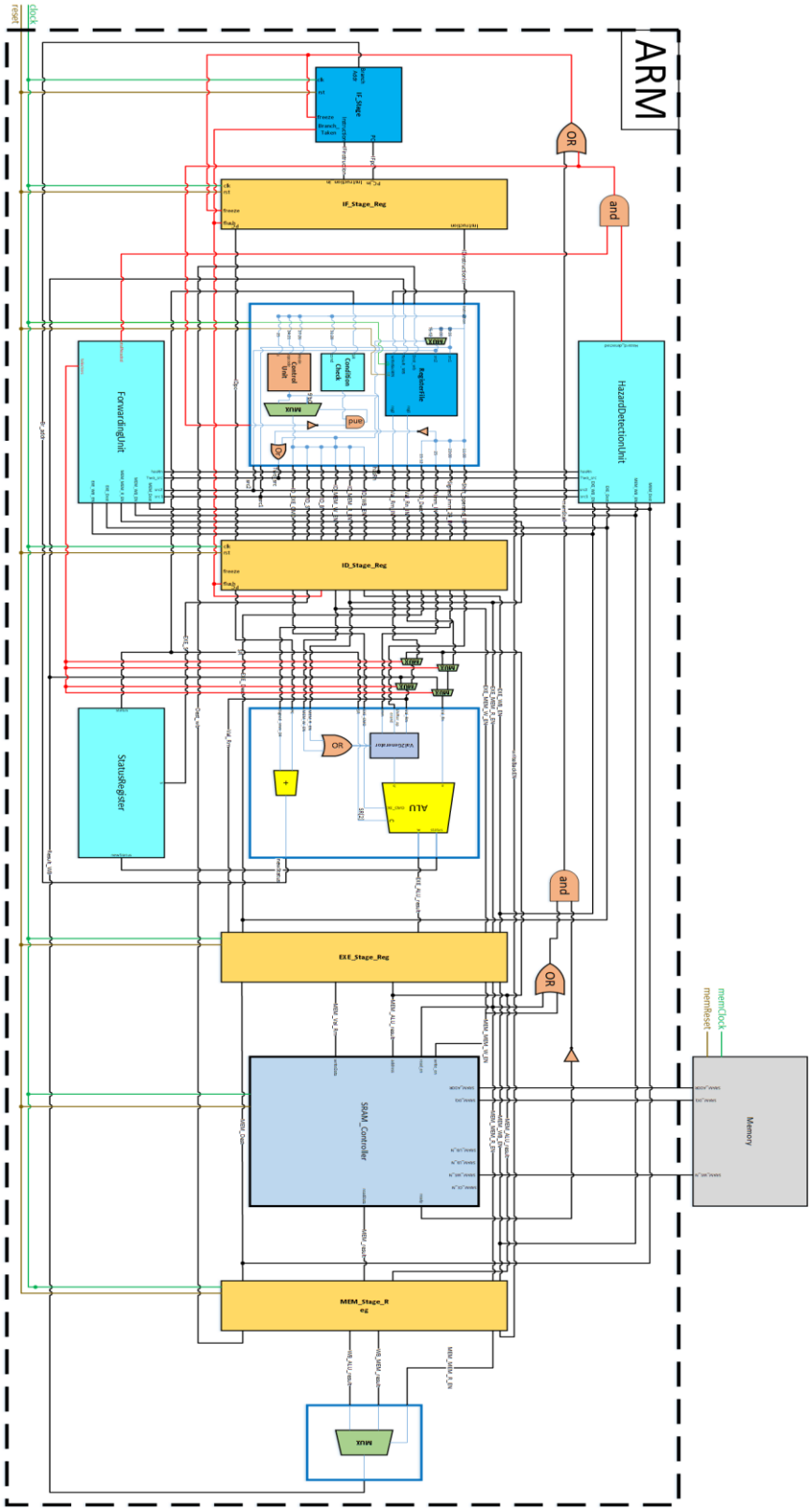
بخش امتیازی:

هرگونه بهبودی که در سرعت قسمت های مختلف پردازنده ایجاد شود طبعا کارایی آن را بالا خواهد برد ولی با توجه به موضوع این فاز که حافظه است صرفا به مواردی که سرعت را از طریق کاهش زمان دسترسی به حافظه بالا می برند می پردازیم.

به طور کل هرچه حافظه بزرگتر باشد کندتر و هرچه کوچکتر باشد سریع تر است. بنابراین برای سریع تر کردن حافظه یک روش کوچک کردن آن است؛ به این شکل که مثلا حافظه را چند قسمت کنیم و دسترسی صرفا به یکی از این بخش ها باشد. که البته این حالت مشکلاتی برای ایجاد هماهنگی بین قسمت های مختلف به وجود می آورد.

روش بهتری که به ذهن می رسد این است که توسط مکانیزمی (مثلا توسط یک ماژول) دستورات چند خط جلوتر از PC خوانده شوند و مشخص شود که به چه خانه هایی از حافظه نیاز خواهد بود. سپس در صورتی که مقادیر فعلی موجود در حافظه برای زمانی که پردازنده به این داده ها نیاز پیدا می کند نیز معتبر باشند، یعنی در آن چند خط دستور جلوتر قرار بر رونویسی آن مقادیر در حافظه نباشد (که این موارد با توجه به سیگنال های کنترلی قابل تشخیص است) حافظه عملیات مهیا کردن داده را سریع تر شروع کند تا وقتی دستور مورد نظر به مرحله حافظه رسید داده نیز آماده شده باشد. در این صورت احتمالا درصد زیادی از وقفه های مربوط به حافظه (حتی درصد بیشتری نسبت به حافظه نهان، زیرا اگر صرفا در چند دستور قبل از یک دسترسی به حافظه آن خانه تغییر نکرده باشد می توانیم از این روش به داده دسترسی پیدا کنیم) از بین می روند و نیازی به آن ها نیست. ولی مشکل دیگری که به وجود می آید این است که برای طراحی این ماژول احتمالا باید یک نمونه از ماژولی شبیه ماژول واکشی دستور (برای خواندن چند دستور آینده) و یک نمونه از ماژولی شبیه ماژول کدگشایی (برای آن که بفهمیم دستورات جلوتر با کجای حافظه کار دارند و چه کار دارند) در آن استفاده کرد که هزینه بر خواهد بود.

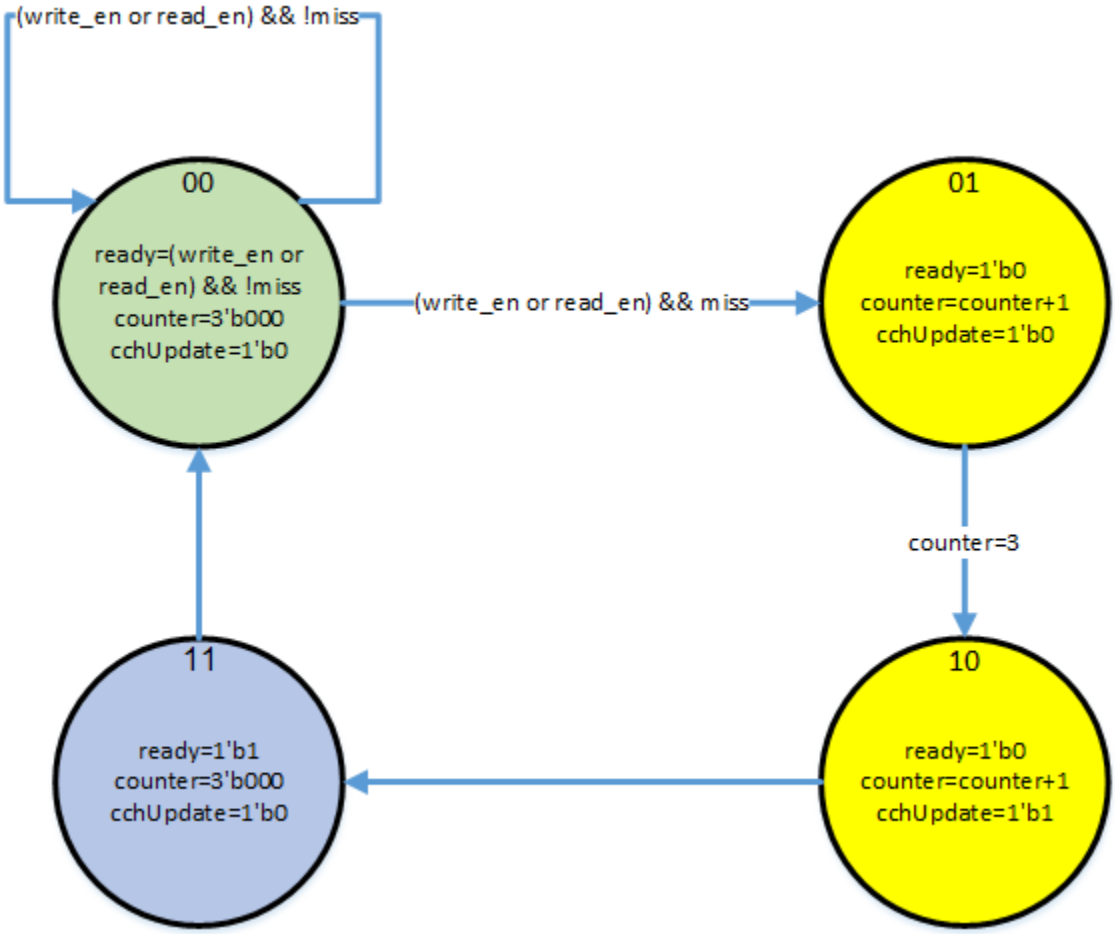
مسیر داده پردازنده ARM با حافظه خارجی (فایل حاوی نمودار: 3_SRM_Datapath.vsdx):



فاز چهارم: استفاده از حافظه نهان

در این فاز، قرار است یک حافظه نهان 2-Way Set Associative با تعداد مجموعه ۶۴ و دو کلمه ۳۲ بیتی در هر بلوک به پردازنده اضافه کنیم.

برای این منظور، ابتدا در ماژول SRAM_Controller تغییراتی ایجاد کردیم تا بتواند حافظه نهان را هم مدیریت کند. تغییرات به این صورت است که یک حالت بین حالت آخر (ready) و حالت وسط (که منتظر انجام عملیات حافظه بود) اضافه کردیم که در این حالت عملیات به روز رسانی حافظه نهان انجام شود (در این حالت سیگنال cchUpdate برابر یک می شود). همچنین در صورتی که در حالت صفر Hit اتفاق بیفتد (ماژول سیگنال miss را از حافظه نهان به عنوان ورودی دریافت می کند که صفر بودن آن به معنای Hit است) ماژول دوباره در همین حالت خواهد ماند و منتظر عملیات بعدی حافظه خواهد بود. ماشین حالت جدید این ماژول به شکل زیر است (فایل حاوی نمودار: 4_CCH_SRAM_Ctrl.vsdx):



گام بعدی طراحی حافظه نهان است. این حافظه نهان با معماری نشان داده شده در تصویر زیر در قالب ماژول Cache پیاده سازی شده است. این معماری به طور کامل در قالب آرایه cch ذخیره می شود و در

کنار آن دو آرایه ۶۴ بیتی دیگر به نام های val1 و val2 وجود دارند که مقدار سطر i ام آرایه valj مشخص می کند مقادیر موجود در بلوک j ام سطر i ام حافظه نهان معتبرند یا خیر.

Cache Architecture							
Index	Cache Line Bits						
	1	10	32	32	10	32	32
0	used	Block 1			Block 2		
		tag 1	word1	word2	tag 2	word1	word2
1	used	Block 1			Block 2		
		tag 1	word1	word2	tag 2	word1	word2
...	...	...			...		
63	used	Block 1			Block 2		
		tag 1	word1	word2	tag 2	word1	word2

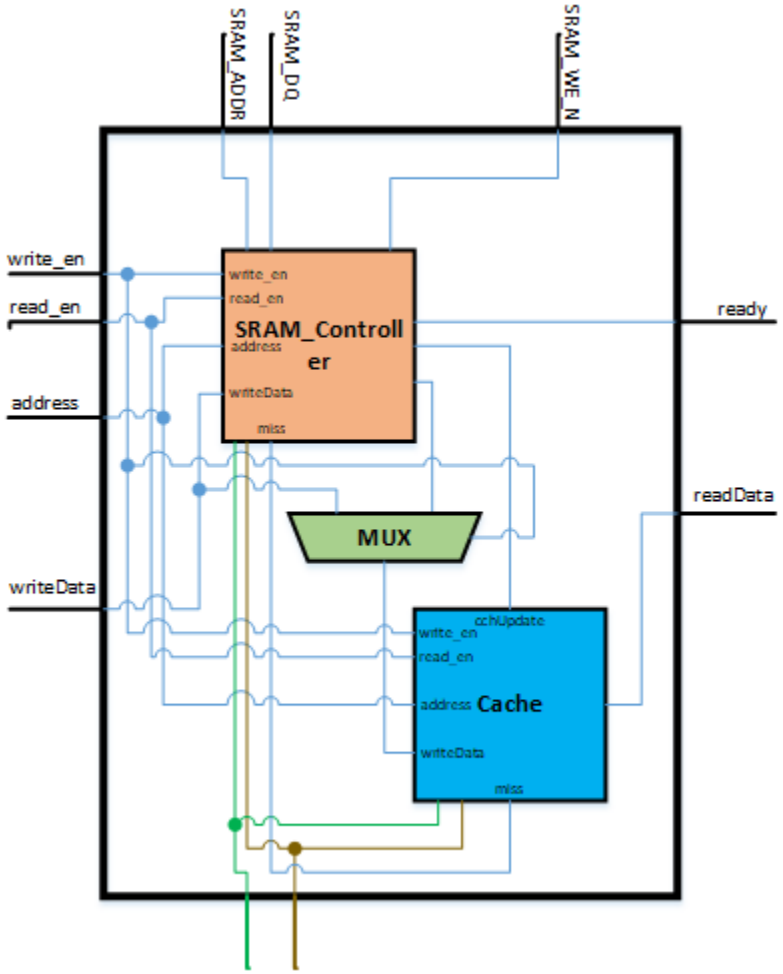
در این ماژول سه بلوک always وجود دارد. بلوک اول یک بلوک ترکیبی است که با توجه به آدرس ورودی مشخص می کنند که داده مد نظر در حافظه نهان وجود دارد یا خیر. برای این امر در ابتدای ماژول بیت های ۳ تا ۸ آدرس به عنوان index و بیت های ۹ تا ۱۸ آن به عنوان tagIn در نظر گرفته می شود. اگر tag هرکدام از بلوک های سطری که با index مشخص می شود با tagIn برابر بود و همچنین مقدار متناظر آن در آرایه val1 یا val2 برابر یک بود یعنی داده موجود در حافظه نهان معتبر است. در غیر این صورت داده معتبر نیست و نیاز است تا دسترسی به حافظه ایجاد شود. سیگنال خروجی miss به همین منظور در این ماژول فعال/غیرفعال می شود.

بلوک دوم نیز ترکیبی است و داده ای که باید به عنوان خروجی داده شود روی درگاه خروجی قرار می دهد. این بلوک ابتدا با بررسی برابر بودن tag آدرس با tag بلوک های index مشخص می کند داده مد نظر در کدام بلوک است، و سپس با توجه به بیت شماره ۲ آدرس بین دو کلمه بلوک انتخاب شده کلمه ای که مد نظر است روی درگاه خروجی قرار می دهد (دو بیت اول آدرس برای دسترسی به بایت های داخل کلمه هستند). بعد از این اتفاق با توجه به این که کدام بلوک استفاده شده است بیت ۱۴۸ همان خط حافظه نهان تغییر می کند تا بلوکی که باید با سیاست LRU جایگزین شود مشخص شود (اگر این بیت برابر صفر باشد باید بلوک اول را جیگزین کنیم و اگر برابر یک باشد بلوک دوم را). از آن جا که سیگنال ready توسط SRAM_Controller تولید می شود و تا این اتفاق نیفتاده از داده خروجی ماژول حافظه استفاده ای نمی کنیم ترکیبی بودن این ماژول مشکلی ایجاد نمی کند.

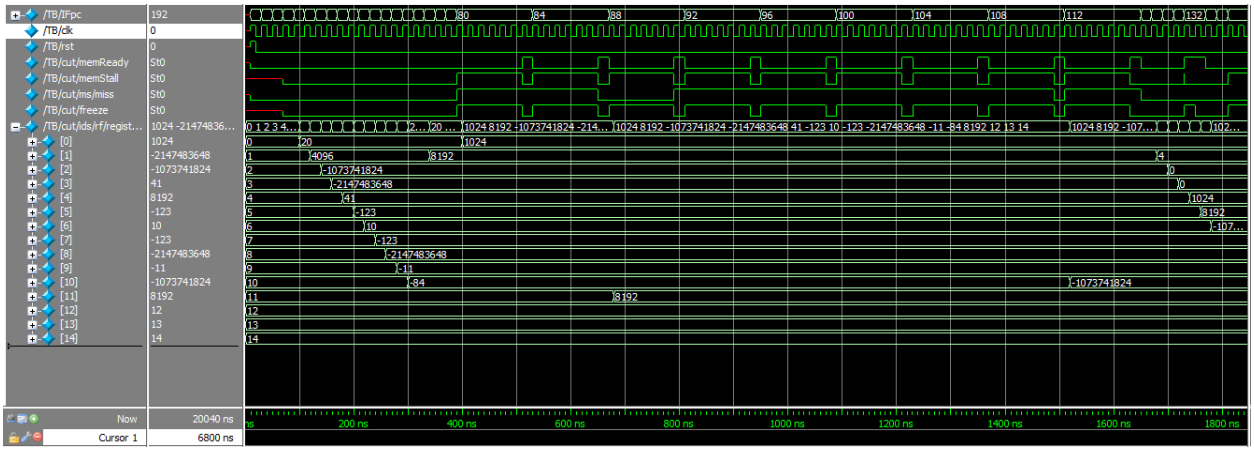
بلوک سوم یک بلوک حساس به لبه بالارونده clock و reset است. در صورت فعال بودن rst، مقدار تمام ۶۴ بیت آرایه های val1 و val2 برابر صفر می شود تا نشان دهد مقدار معتبری در حافظه نهان وجود ندارد. در صورت رسیدن لبه بالا رونده clock، اگر سیگنال ورودی cchUpdate برابر یک باشد مقادیر موجود در حافظه نهان به شکل زیر به روز می شوند:

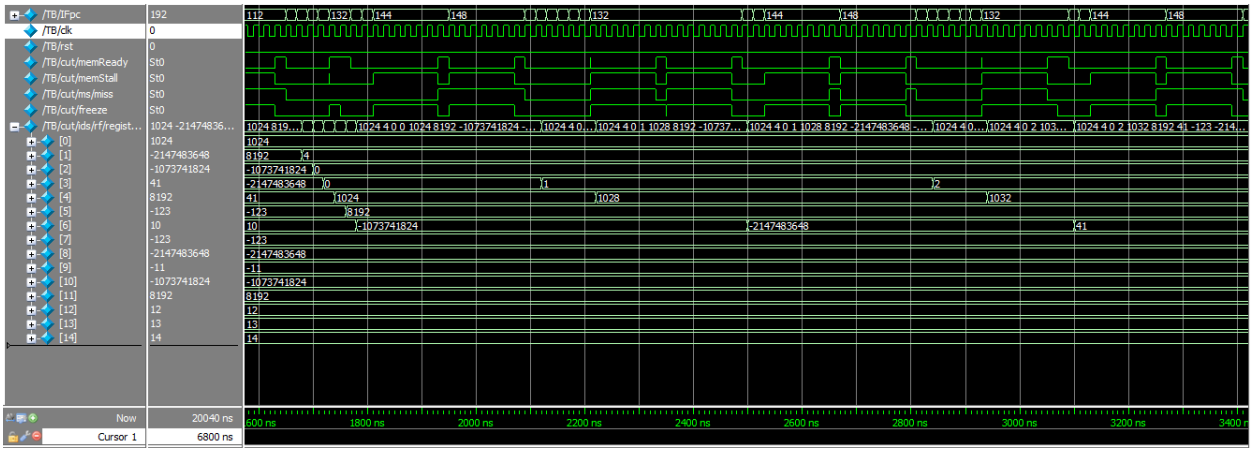
- اگر دستور خواندن از حافظه بوده باشد و miss اتفاق بیفتد، اگر آن خانه های حافظه در حافظه نهان موجود بوده باشند و صرفا مقدارشان نامعتبر باشد داده های جدید روی همان داده ها نوشته می شوند و در غیر این صورت با توجه به بیت ۱۴۸ خط حافظه نهان بلوکی که باید جایگزین شود به روز می شود. در این حالت مقدار tag بلوک مربوطه نیز به روز می شود.
- اگر دستور نوشتن در حافظه بوده باشد از آن جا که حافظه نهان No-Write Allocate است داده های حافظه نهان را به روز نمی کنیم و صرفا اگر این خانه های از قبل در حافظه نهان موجود بوده باشند بیت مربوط به آن ها را در آرایه ها val1 و val2 برابر صفر می کنیم تا از آن به بعد معتبر فرض نشوند.

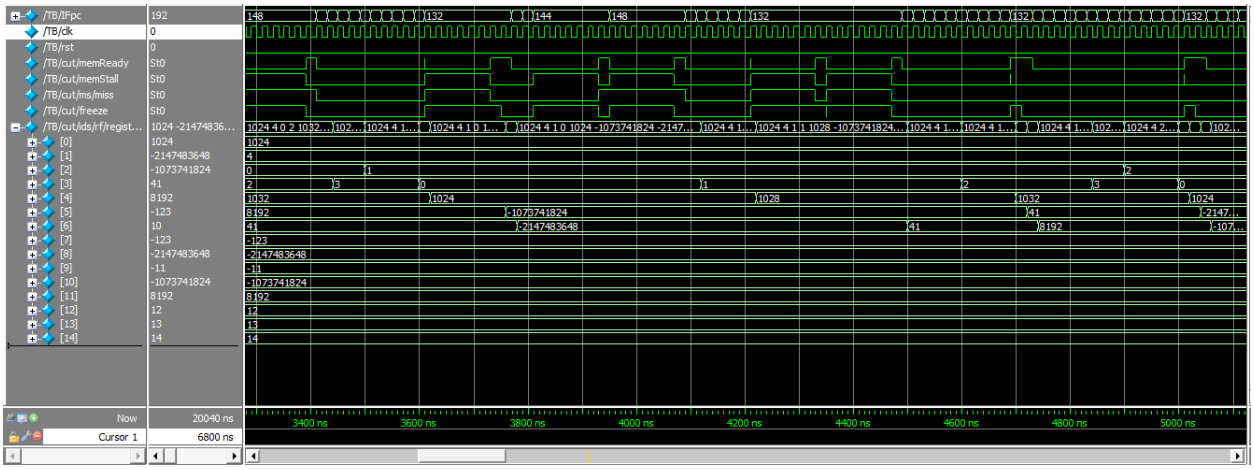
از این دو ماژول در ماژول MEM_Stage نمونه می گیریم و آن ماژول را در پردازنده قرار می دهیم. ساختار MEM_Stage چنین می شود (فایل حاوی نمودار: 4_CCH_MEM.vsdx):

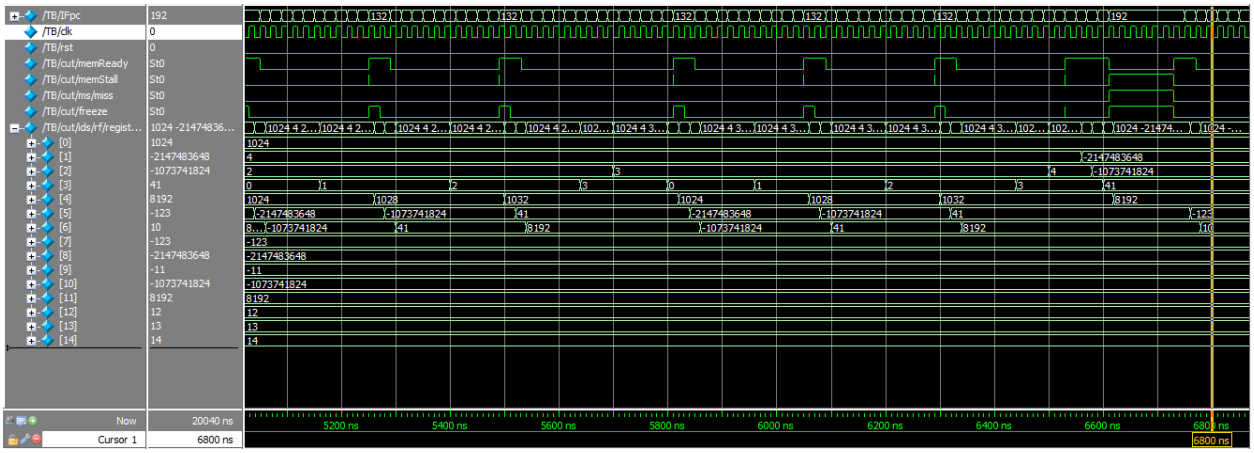


برای آزمودن برنامه ابتدا برنامه محک را روی پردازنده اجرا کردیم که نتایج آن به صورت زیر است:


/TB/IFpc 192
/TB/clk 0
/TB/rst 0
/TB/cut/memReady St0
/TB/cut/memStall St0
/TB/cut/ms/miss St0
/TB/cut/freeze St0
/TB/cut/ids/rf/regist... 1024 -21474836...
[0] 1024
[1] -2147483648
[2] -1073741824
[3] 41
[4] 8192
[5] -123
[6] 10
[7] -123
[8] -2147483648
[9] -11
[10] -1073741824
[11] 8192
[12] 12
[13] 13
[14] 14
Now 20040 ns
Cursor 1 6800 ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1400 ns
1600 ns
1800 ns


/TB/IFpc 192
/TB/clk 0
/TB/rst 0
/TB/cut/memReady St0
/TB/cut/memStall St0
/TB/cut/ms/miss St0
/TB/cut/freeze St0
/TB/cut/ids/rf/regist... 1024 -21474836...
[0] 1024
[1] -2147483648
[2] -1073741824
[3] 41
[4] 8192
[5] -123
[6] 10
[7] -123
[8] -2147483648
[9] -11
[10] -1073741824
[11] 8192
[12] 12
[13] 13
[14] 14
Now 20040 ns
Cursor 1 6800 ns
1800 ns
2000 ns
2200 ns
2400 ns
2600 ns
2800 ns
3000 ns
3200 ns


/TB/IFpc 192
/TB/clk 0
/TB/rst 0
/TB/cut/memReady St0
/TB/cut/memStall St0
/TB/cut/ms/miss St0
/TB/cut/freeze St0
/TB/cut/ids/rf/regist... 1024 -21474836...
[0] 1024
[1] -2147483648
[2] -1073741824
[3] 41
[4] 8192
[5] -123
[6] 10
[7] -123
[8] -2147483648
[9] -11
[10] -1073741824
[11] 8192
[12] 12
[13] 13
[14] 14
Now 20040 ns
Cursor 1 6800 ns
3400 ns
3600 ns
3800 ns
4000 ns
4200 ns
4400 ns
4600 ns
4800 ns
5000 ns



این اجرا ۳۴۶ سیکل ساعت طول کشید، ۱۶۹ دستور، ۱۲ سیکل ساعت توقف به دلیل مخاطره، ۱۵۰ سیکل ساعت توقف برای دسترسی به حافظه و ۱۱ سیکل ساعت بعد از دستورات پرش (جزئیات نحوه اجرای دستورات در فایل 4_CCH_PipeLine.xlsx موجود است). بنابرین تسریع ایجاد شده با اشتفاده از حافظه نهان نسبت به قسمت قبل به این صورت محاسبه می شود:

$$sppedUp = \frac{431}{346} - 1 = \frac{85}{346} \cong 24.57\%$$

از بین ۱۶۹ دستور اجرا شده ۴۷ دستور با حافظه کار دارند. ۹ دستور ابتدای این ۴۷ دستور همگی قبل از شروع حلقه ها هستند؛ ۲ تا LDR و ۷ تا STR. در خط ۱۸ که خانه ۱۰۲۴ حافظه خوانده می شود در واقع خانه ۱۰۲۸ نیز در حافظه نهان قرار می گیرد، ولی چون در خط ۱۹ روی این خانه چیزی نوشته می شود مقدار موجود در حافظه نهان نامعتبر می شود و در خط ۲۴ دوباره نیاز است تا دسترسی به حافظه ایجاد شود. برای STR ها هم که همیشه نیاز به توقف داریم و در نتیجه تمام این ۹ دستور پردازنده را متوقف می کنند.

در داخل حلقه ها، ابتدا دو دستور LDR (خطوط ۳۰ و ۳۱) مقادیر خانه هایی از حافظه را می خوانند و سپس دو دستور STR (خطوط ۳۳ و ۳۴) در صورت برقرار بودن شرطی مقادیر جدیدی در همین دو خانه می نویسند. اجرای این دستورات از حلقه اول تا ۱۲ ام به این صورت است:

حلقه اول: در خطوط ۳۰ و ۳۱ خانه های ۱۰۲۴ و ۱۰۲۸ باید خوانده شوند. مقدار این خانه ها بعد از اجرای دستور خط ۲۴ به حافظه نهان منتقل شده اند و در نتیجه Hit اتفاق می افتد و توقف نداریم.

در خطوط ۳۳ و ۳۴ شرط برقرار است و مقادیر خانه های ۱۰۲۴ و ۱۰۲۸ تغییر می کند. برای اجرای دستورات به توقف نیاز است.

حلقه دوم: در خطوط ۳۰ و ۳۱ خانه های ۱۰۲۸ و ۱۰۳۲ باید خوانده شوند. مقدار خانه ۱۰۲۸ در حافظه بعد از اجرای خطوط ۳۳ و ۳۴ حلقه قبل تغییر کرده و درنتیجه مقدار موجود در حافظه نهان نامعتبر است و باید خواندن از حافظه انجام شود. و چون خانه ۱۰۳۲ هیچگاه خوانده نشده است

مقدارش در حافظه نهان وجود ندارد و برای این دستور هم باید خواندن از حافظه انجام شود. پس برای هر دوی این دستورات نیاز به توقف داریم.

در خطوط ۳۳ و ۳۴ شرط برقرار است و مقادیر خانه های ۱۰۲۸ و ۱۰۳۲ تغییر می کند. برای اجرای دستورات به توقف نیاز است.

حلقه سوم: در خطوط ۳۰ و ۳۱ خانه های ۱۰۳۲ و ۱۰۳۶ باید خوانده شوند. مقدار خانه ۱۰۳۲ در حافظه بعد از اجرای خطوط ۳۳ و ۳۴ حلقه قبل تغییر کرده و درنتیجه مقدار موجود در حافظه نهان نامعتبر است و باید خواندن از حافظه انجام شود. ولی بعد از اجرای این دستور مقدار خانه ۱۰۳۶ هم به حافظه نهان می آید و برای دستور خط ۳۱ Hit اتفاق می افتد. پس فقط برای دستور خط ۳۰ نیاز به توقف داریم.

در خطوط ۳۳ و ۳۴ شرط برقرار است و مقادیر خانه های ۱۰۳۲ و ۱۰۳۶ تغییر می کند. برای اجرای دستورات به توقف نیاز است.

حلقه چهارم: در خطوط ۳۰ و ۳۱ خانه های ۱۰۲۴ و ۱۰۲۸ باید خوانده شوند. مقدار خانه ۱۰۲۴ در حافظه بعد از اجرای خطوط ۳۳ و ۳۴ حلقه اول تغییر کرده و درنتیجه مقدار موجود در حافظه نهان نامعتبر است و باید خواندن از حافظه انجام شود. ولی بعد از اجرای این دستور مقدار خانه ۱۰۲۸ هم به حافظه نهان می آید و برای دستور خط ۳۱ Hit اتفاق می افتد. پس فقط برای دستور خط ۳۰ نیاز به توقف داریم.

در خطوط ۳۳ و ۳۴ شرط برقرار است و مقادیر خانه های ۱۰۲۴ و ۱۰۲۸ تغییر می کند. برای اجرای دستورات به توقف نیاز است.

حلقه پنجم: در خطوط ۳۰ و ۳۱ خانه های ۱۰۲۸ و ۱۰۳۲ باید خوانده شوند. مقدار خانه های ۱۰۲۸ و ۱۰۳۲ در حافظه بعد از اجرای خطوط ۳۳ و ۳۴ حلقه دوم و سوم تغییر کرده و درنتیجه مقدار موجود در حافظه نهان نامعتبر است و باید خواندن از حافظه انجام شود. پس برای هر دوی این دستورات نیاز به توقف داریم.

در خطوط ۳۳ و ۳۴ شرط برقرار نیست، دستور اجرا نمی شود و مقادیر خانه های ۱۰۲۸ و ۱۰۳۲ در حافظه تغییر نمی کند.

حلقه ششم: در خطوط ۳۰ و ۳۱ خانه های ۱۰۳۲ و ۱۰۳۶ باید خوانده شوند. بعد از اینکه برای اجرای دستور خط ۳۱ حلقه قبل برای خانه ۱۳۲ دسترسی به حافظه پیدا شد هر دو خانه ۱۰۳۲ و ۱۰۳۶ به حافظه نهان منتقل شدند و مقدارشان نیز تغییر نکرده است. پس Hit اتفاق می افتد و نیاز به توقف نیست.

در خطوط ۳۳ و ۳۴ شرط برقرار نیست، دستور اجرا نمی شود و مقادیر خانه های ۱۰۳۲ و ۱۰۳۶ در حافظه تغییر نمی کند.

در ۶ حلقه باقی مانده نیز دسترسی به همین خانه های حافظه (و با همین ترتیب) نیاز است. تا انتهای اجرای این حلقه ها مقدار این خانه های حافظه تغییر نمی کند و در نتیجه مقادیر موجود در حافظه نهان معتبر می مانند. بنابراین تمام دستورات LDR با Hit مواجه می شوند و نیازی به توقف پردازنده و دسترسی به حافظه نیست.

بعد از اتمام این حلقه ها، در خطوط ۴۱ تا ۴۴ برنامه دستورات LDR مقادیر خانه های ۱۰۲۴ تا ۱۰۳۶ حافظه را در ثبات ها ذخیره می کنند. مقادیر تمام این ۴ خانه هنگام اجرای حلقه ها به حافظه نهان منتقل شده است و بعد از انتقال تغییری در آن ها ایجاد نشده است. بنابراین مقادیر موجود در حافظه نهان معتبرند و Hit اتفاق می افتد و نیاز به توقف پردازنده نیست. سپس در خطوط ۴۶ و ۴۷ مقادیر خانه های ۱۰۴۰ و ۱۰۴۴ خوانده می شود که چون به این خانه ها دسترسی نداشته ایم در حافظه نهان نیستند. بنابراین در این دو دستور پردازنده متوقف می شود تا دسترسی به حافظه انجام شود.

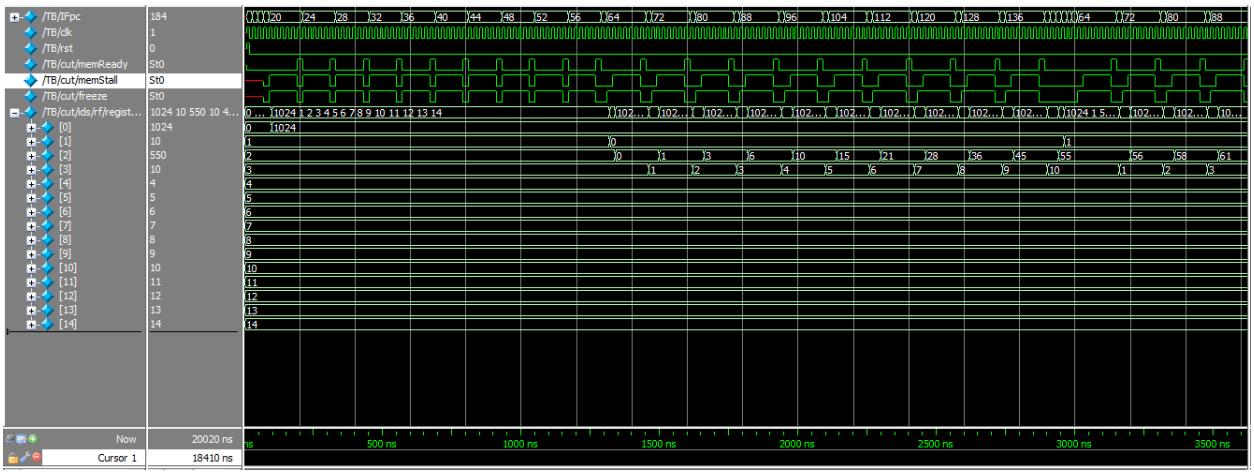
از مجموع ۴۷ دستوری که دسترسی به حافظه نیاز دارند، ۳۲ دستور LDR هستند. با توجه به توضیحات داده شده از این ۳۲ دستور، ۲۲ دستور با Hit و ۱۰ دستور با Miss مواجه می شوند.

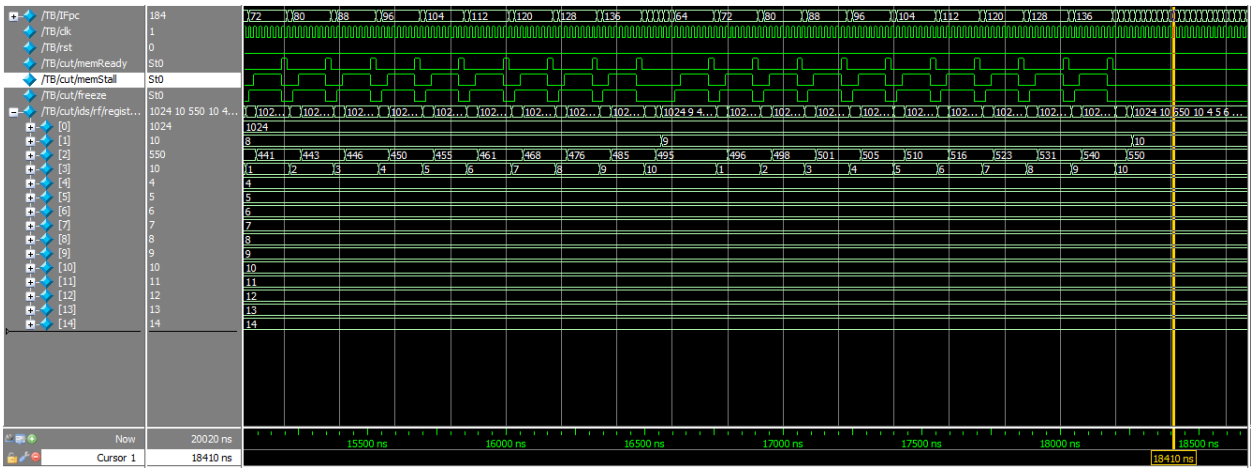
بعد از این آزمون، برنامه محک دیگری با ۱۱۰ دسترسی به حافظه برای بررسی بهتر عملکرد حافظه نهان به صورت زیر نوشته شد. در این برنامه ابتدا خانه های یک تا ۱۰ حافظه توسط ثبات های عمومی مقدار دهی می شوند و سپس ده مرتبه جمع مقادیر همین خانه های حافظه محاسبه و انباشته می شود.

```
32'b1110_00_1_1101_0_0000_0000_101100000001     MOV  R0,#1024
32'b1110_01_0_0100_0_0000_0001_000000000000     STR  R1,[R0],#0
32'b1110_01_0_0100_0_0000_0010_000000000100     STR  R2,[R0],#4
32'b1110_01_0_0100_0_0000_0011_000000001000     STR  R3,[R0],#8
32'b1110_01_0_0100_0_0000_0100_000000001100     STR  R4,[R0],#12
32'b1110_01_0_0100_0_0000_0101_000000010000     STR  R5,[R0],#16
32'b1110_01_0_0100_0_0000_0110_000000010100     STR  R6,[R0],#20
32'b1110_01_0_0100_0_0000_0111_000000011000     STR  R7,[R0],#24
32'b1110_01_0_0100_0_0000_1000_000000011100     STR  R8,[R0],#28
32'b1110_01_0_0100_0_0000_1001_000000100000     STR  R9,[R0],#32
32'b1110_01_0_0100_0_0000_1010_000000100100     STR  R10,[R0],36
32'b1110_00_1_1101_0_0000_0001_000000000000     MOV  R1,#0
32'b1110_00_1_1101_0_0000_0010_000000000000     MOV  R2,#0
```

```
32'b1110_01_0_0100_1_0000_0011_000000000000    LDR  R3,[R0],#0
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000000100    LDR  R3,[R0],#4
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000001000    LDR  R3,[R0],#8
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000001100    LDR  R3,[R0],#12
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000010000    LDR  R3,[R0],#16
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000010100    LDR  R3,[R0],#20
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000011000    LDR  R3,[R0],#24
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000011100    LDR  R3,[R0],#28
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000100000    LDR  R3,[R0],#32
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_01_0_0100_1_0000_0011_000000100100    LDR  R3,[R0],#36
32'b1110_00_0_0100_1_0010_0010_000000000011    ADD  R2,R2,R3
32'b1110_00_1_0100_0_0001_0001_000000000001    ADD  R1,R1,#1
32'b1110_00_1_1010_1_0001_0000_000000001010    CMP  R1,#10
32'b1011_10_1_0100_1_1111_1111_111111010001    BLT  #-23
```

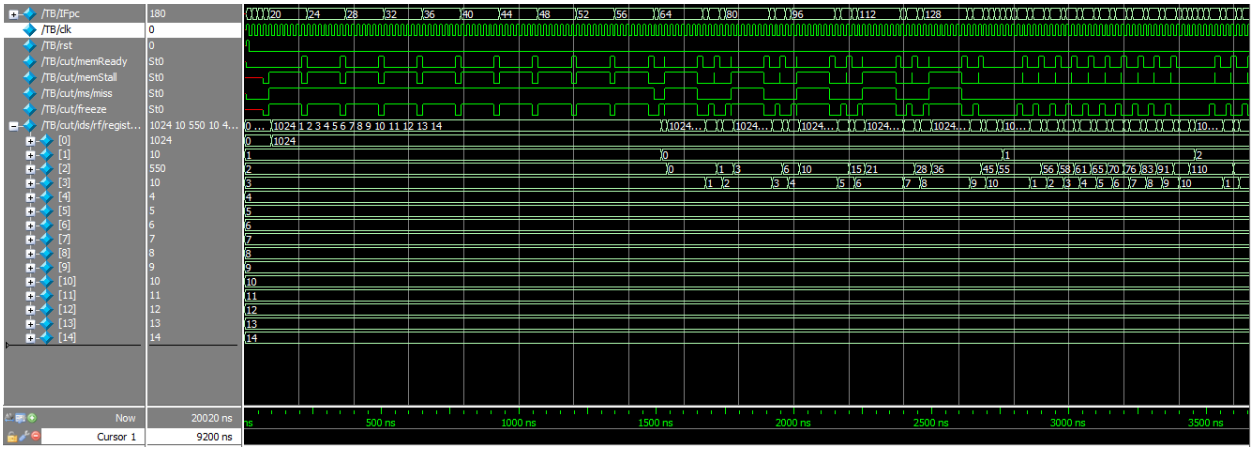
ابتدا این کد را توسط برنامه فاز قبل (SRAM) اجرا کردیم که نتیجه آن به صورت زیر است:

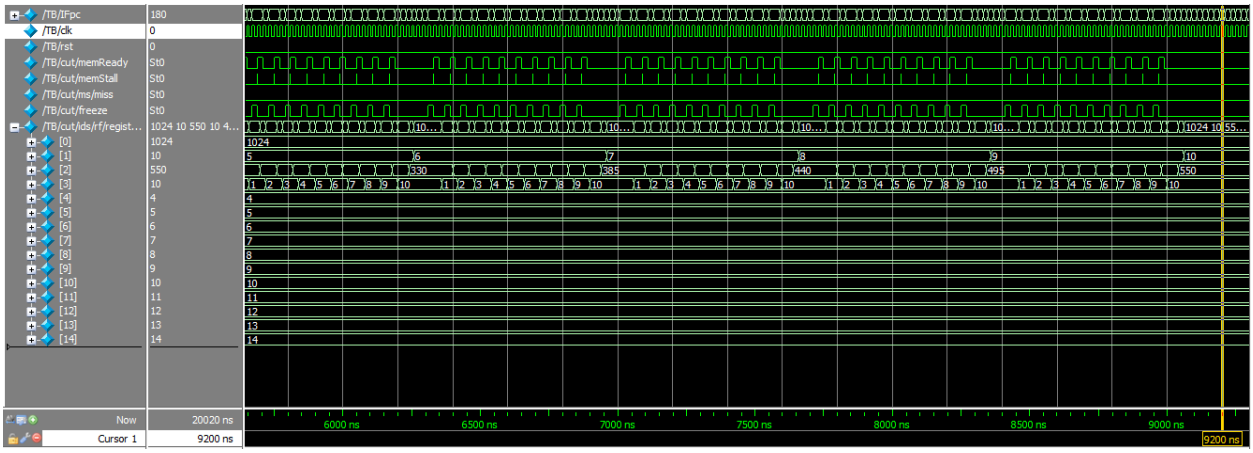




پاسخ به درستی برابر ۵۵۰ است و اجرای برنامه ۹۰۶ سیکل ساعت طول کشیده است؛ ۲۴۳ دستور، ۱۰۰ سیکل ساعت توقف به دلیل مخاطره، ۵۵۰ سیکل ساعت توقف برای دسترسی به حافظه و ۹ سیکل ساعت بعد از دستورات پرش.

سپس برنامه را روی پردازده ای که حافظه نهان به آن الحاق شده اجرا کردیم که نتیجه آن به صورت زیر است:



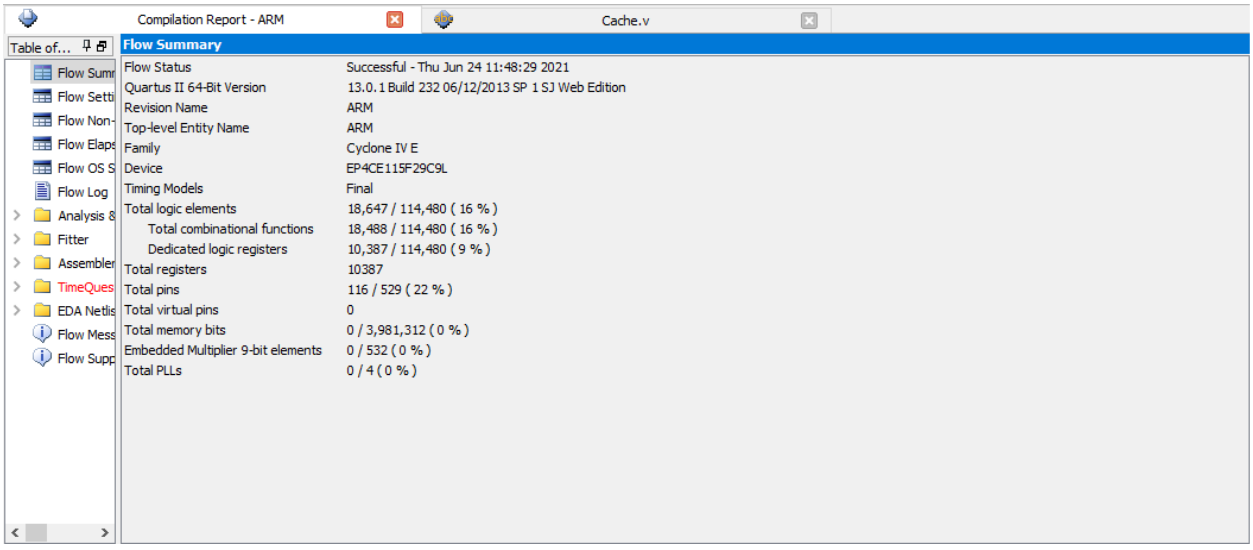


در این آزمون نیز پاسخ به درستی برابر ۵۵۰ شده و همچنین اجرای برنامه ۴۴۶ سیکل ساعت طول کشیده است. در این اجرا، از بین ۱۰۰ مرتبه ای که داده از حافظه خوانده می شود فقط ۵ مرتبه miss اتفاق می افتد و ۹۵ مرتبه داده در حافظه نهان وجود دارد (در حلقه اول برای دسترسی به خانه های ۱۰۲۴، ۱۰۳۲، ۱۰۴۰، ۱۰۴۸ و ۱۰۵۶ miss اتفاق می افتد که پس از دسترسی به حافظه مقادیر خانه های ۱۰۲۸، ۱۰۳۶، ۱۰۴۴، ۱۰۵۲ و ۱۰۶۰ نیز داخل حافظه نهان قرار می گیرند. در حلقه های بعدی هم از آنجا که داده ها تغییری نمی کنند داده های حافظه نهان معتبرند و برای تمام دسترسی ها Hit اتفاق می افتد) و بنابراین باید نسبت به اجرای قبلی ۴۶۰ سیکل ساعت کمتر زمان صرف شود (۴۷۵ سیکل توقف های حافظه را نداریم و در عوض ۱۵ مرتبه توقفی که داریم یک سیکل ساعت بیشتر طول می کشند).

بنابراین میزان تسریع حاصل شده با استفاده از حافظه نهان به این صورت محاسبه می شود:

$$speedUp = \frac{902}{442} \cong 204\%$$

پس از اطمینان از صحت عملکرد پردازنده آن را روی همان برد مراحل قبل سنتز کردیم که نتایج سنتز به صورت زیر است:



مشاهده می شود که استفاده از حافظه نهان هزینه سنتز را حدودا ۷ برابر می کند.

$$costInreasse = \frac{18647}{2365} - 1 \cong 688\%$$

بنابراین اگر بخواهیم نسبت افزایش کارایی به افزایش هزینه را محاسبه کنیم خواهیم داشت:

برای برنامه محک: $\frac{\frac{431}{346}-1}{\frac{18647}{2635}-1} = \frac{85\times2635}{346\times16282} \cong 3.98\%$

برای برنامه دوم: $\frac{\frac{906}{446}-1}{\frac{18647}{2635}-1} = \frac{460\times2635}{446\times16282} \cong 16.69\%$

جدول نهایی گزارش فاز چهار	
Total Logic Elements	18647
Total Combinational Functions	18448
Dedicated Local Registers	10387
Instructions	169
Clock Cycles	346
CPI	0.488

در فاز اول، زمانی که تمام حافظه را با استفاده از حافظه داخلی ساختیم این حافظه تنها ۶۴ بایت فضا داشت. در واقع زمانی که حافظه بزرگ طراحی شده بود تعداد قطعات منطقی سنتز چیزی در مرتبه ۴۵۰۰۰۰ می شد که از ظرفیت حدودا ۱۴۴۰۰۰ تایی ابزار سنتز بیشتر بود و مدار روی برد جا نمی گرفت. بنابراین تعداد قطعات منطقی محاسبه شده مثلا در سنتز فاز دو برای جدول مقایسه ای پایین قابل استفاده نیست.

جدول نهایی گزارش فاز چهار		
Phase	CPI	Cost
Internal Memory	0.862	450,000
SRAM	0.392	2365
Cache	0.488	18647

بخش امتیازی:

مزیت معماری 2_Way Set Associative نسبت به معماری Direct Map برای حافظه نهان در این است که در آن واحد به جای یک بلوک، دو بلوک از حافظه نهان به طور موازی بررسی می شوند. این امر تاثیری در سرعت حافظه نهان ندارد ولی با زیاد کردن گنجایش هر سطر فضای بیشتری برای ذخیره داده ها فراهم می کند. همچنین از آن جا که عملیات بررسی بلوک های هر سطر به صورت موازی و هم زمان انجام می شود، اگر مثلا به معماری Direct Map با ۱۲۸ سطر از یک معماری 2_Way Set Associative با ۶۴ سطر استفاده کنیم حافظه نهان سریع تر خواهد شد، چرا که تنها تغییری که در سرعت تاثیر می گذارد

نصف شدن تعداد سطور حافظه نهان است. بنابر استدلال مشابه، اگر حافظه نهان 2_Way Set Associative طراحی شده در این بخش نیز به حافظه 4_Way Set Associative تبدیل شود سرعت آن بیشتر خواهد شد.

تغییر دیگری که می توان برای افزایش کارایی حافظه نهان پیشنهاد داد قرار دادن مثلا ۴ کلمه به جای ۲ کلمه در هر بلوک است. در این حالت با استفاده از اصل محلی بودن مکانی در هر بار دسترسی به حافظه داده های بیشتری به حافظه نهان منتقل می شوند و احتمال اینکه دسترسی های بعدی به این خانه ها با Hit همراه باشد بیشتر می شود و درنتیجه کارایی بالاتر می رود.

با توجه به این که چه دسترسی هایی و با چه ترتیبی به حافظه انجام شود الگوریتم جایگزینی بهینه متفاوت می شود. معمولا همین سیاست LRU سیاست بهینه ای تلقی می شود چرا که غالبا در کدنویسی ها دسترسی هایی که به یک بخش مشخص حافظه انجام می شود کنار هم است و پس از اتمام کار با آن بخش سراغ بخش های دیگر می روند. ولی در صورتی که احتمال دسترسی مجدد به داده ای که اخیرا استفاده شده کمتر از احتمال دسترسی مجدد به داده ای باشد که قبلتر استفاده شده است مثلا سیاست MRU بهتر خواهد بود.

در نهایت هیچ تضمینی نیست که سیاست جایگزینی انتخاب شده همیشه بهترین سیاست باشد. در صورتی که برنامه نویس معماری حافظه نهان را بداند می تواند برنامه و دسترسی های حافظه را طوری بنویسد که به ازای تمام آن ها نیز نیاز به توقف پردازنده و دسترسی به حافظه اصلی وجود داشته باشد.

مسیر داده پردازنده با استفاده از حافظه نهان (فایل حاوی نمودار: 4_CCH_Datapath.vsdx):