فهرست

قسمت ARM معمولی :

توصیف عملکردی:

پردازنده آرم طراحی شده دارای کلاک با لبه بالا رونده و ریست است(status register و register file با لبه پایین رونده کار میکنند). همچنین شامل ۵ استیج و ۴ رجیستر میانی می باشد (غیر از ماژول های hazard detection و status register). در استیج اول (IF) دستورات از حافظه دستور واکشی می شوند و شمارنده برنامه به دستور بعد می رود. در استیج دوم (ID) دستور واکشی شده دیکود میشود. عملگرهای آن از رجیسترفایل خوانده می شوند و سیگنال های کنترلی آن تولید میشوند و به مرحله بعد فرستاده میشوند. در استیج ۳ (EXE) ورودی ها دستور لازم الاجرا به ALU داده میشوند. همچنین آدرس پرش محاسبه میشود و به عنوان کاندیدا به استیج اول فرستاده میشود. نتیجه ALU به مرحله بعد فرستاده میشوند و Status register در صورت لزوم آپدیت میشود. در استیج ۴ (MEM) در صورت لزوم از مموری خوانده یا در آن نوشته میشود و داده خوانده شده به مرحله بعد میرود. در استیج ۵ (WB) بین خروجی فرستاده شده از ALU و حافظه داده با توجه به نوع دستور یکی خوانده میشود و در صورت نیاز در رجیستر فایل نوشته میشود. واحد hazarad detection با توجه به نوع دستور و عملگرها و شماره رجیستر مقصد دستور فعلی (درون ID) و دستور های درون استیج های MEM و EXE مخاطره را تشخیص میدهد و با تولید سیگنال hazard باعث وقفه در پردازنده و ایجاد حباب میشود. status register هم وضعیت محاسبات قبلی ALU را نشان می دهد و وضعیت آن برای تصمیم راجع به اجرای شرطی دستورات وارد استیج ID (ماژول Condition Check) میشود.

**ارتباط بین ماژول ها و استیج ها را در سطح RTL توضیح میدهیم (کد هم همین گونه است). سپس در توضیح سطح کد فقط به توضیح کد هر ماژول می پردازیم.

توصیف در سطح RTL:

استیج اول: یک رجیسترPC برای ذخیره شمارنده برنامه داریم. ورودی آن بر اساس سیگنال Branch Taken از بین آدرس پرش (که در استیج اجرا محاسبه شده) و آدرس دستور عادی بعدی (pc+4) که از خروجی واحد جمع کننده در همین استیج می آید، تعیین میشود. همچنین خروجی رجیستر PC علاوه بر فرستاده شدن به واحد جمع کننده به عنوان آدرس به حافظه دستور فرستاده میشود تا دستور متناظر خوانده شود. همچنین pc(جمع شده با۴) و دستور در رجیستر میانی ذخیره میشوند.

استیج دوم: با توجه به سیگنال MEM_W_EN از بین Rm و Rd یکی به عنوان شماره عملگر دوم به همراه Rn به عنوان شماره عملگر اول به رجیستر فایل داده میشود و مقادیر آن ها خوانده میشود. opcode و mode و s_in به واحد کنترل فرستاده میشوند و این واحد سیگنال های دستور را تولید میکند. بیت های Cond مربوط به دستور به همراه وضعیتstatus register به واحد Condition Check فرستاده میشوند، مطابقت آنها سنجیده میشود و به عنوان سیگنال یک بیتی خروجی داده می شود. حال اگر این سیگنال صفر باشد(شرط با وضعیت تطابق نداشته باشد) یا واحد تشخیص مخاطره، مخاطره تشخیص داده باشد، سیگنال های کنترلی تولید شده توسط واحد کنترل صفر میشوند. سپس

این سیگنال ها به مرحله بعد فرستاده میشوند. تمام سیگنال های کنترلی، عملگرهای خوانده شده از رجیسترفایل، شماره رجیستر مقصد، بیت imm، pc و بیت های مربوط به shifter operand و signed_imm24(برای دستور پرش) در رجیستر میانی دوم ذخیره میشوند و وارد استیج بعد میشوند.

استیج سوم: یک جمع کننده داریم که pc آمده از استیج قبل را با sign extend شده ی signed_imm24 جمع میکند و آدرس پرش را تولید میکند. ماژول Val2Generate ورودی های Val_Rm (عملگر دوم خوانده شده از رجیستر فایل)، shifter operand، imm و خروجی or (که نشان دهنده ی کار داشتن دستور با حافظه داده است) را دریافت میکند و ورودی دوم ALU را تولید می کند. ماژول ALU دو ورودی داده را به همراه بیت carry از رجیستر وضعیت و دستور اجرایی می گیرد که ورودی اول و دستور اجرایی و بیت های وضعیت از استیج قبل آمده اند. این ماژول محاسبه اش را انجام می دهد و نتیجه و بیت های وضعیت جدید را تولید میکند. بیت های وضعیت جدید به رجیستر وضعیت میروند و در صورت یک بودن بیت s (در لبه پایین رونده) در این رجیستر ذخیره می شوند. بیت های وضعیت خروجی این رجیستر را به استیج ID (به عنوان ورودی Condition Check) و رجیستر میانی دوم (برای ذخیره و فرستادن آن به استیج EXE) می فرستیم. سیگنال های مموری و WB_EN و Val_Rm و شماره رجیستر مقصد(Dest) که از استیج قبل آمده اند به همراه خروجی ALU در رجیستر میانی سوم ذخیره و به استیج بعد فرستاده میشوند.

استیج چهارم: در این استیج حافظه داده قرار دارد. سیگنال های خواندن و نوشتن در حافظه به همراه آدرسی که توسط ALU تولید شده است و مقدار قابل نوشتن Val_Rm که دومین مقدار خوانده شده از رجیستر فایل است، وارد حافظه می شوند و مقدار خوانده شده از حافظه خروجی داده می شود. تمام wire های این مرحله به جز MEM_W_EN که دیگر به آن نیاز نداریم در رجیستر میانی چهرم ذخیره و به استیج بعد فرستاده میشوند.

استیج پنجم: یک مالتی پلکسر داریم که با توجه به سیگنال MEM_R_EN از بین خروجی ALU و داده ی خوانده شده از مموری یکی را انتخاب میکند و به عنوان مقداری که می خواهیم در رجیستر مقصد ذخیره کنیم، خروجی میدهد. این مقدار در صورت فعال بودن WB_EN در رجیستر با شماره Dest در رجیسترفایل(در لبه پایین رونده) نوشته میشود.

ماژول تشخیص مخاطره: این ماژول شماره رجیستر مقصد و سیگنال نوشتن در رجیسترفایل مربوط به استیج های MEM و WB را به همراه شماره عملگرهای اول و دوم(که مقدارشان از رجیسترفایل خوانده میشود) و سیگنال های imm و MEM_W_EN (برای تشخیص دو عملگری بودن دستور) و EXE_CMD (برای قسمت امتیازی یعنی تشخیص اینکه دستور عملگر اول دارد یا خیر) دریافت می کند و سیگنال خروجی بنام hazard تولید میکند. در صورت یک بودن این سیگنال رجیستر PC و رجیستر میانی اولfreeze میشوند یعنی خروجیشان در لبه بالارونده مقدار ورودی را نمیگیرد و بدین وسیله دستورات موجود در IF وID صبر میکنند تا دستورات قبلیشان جلو بروند و در یک یا دو سیکل بعد دیگر شرط رخداد هازارد برقرار نباشد.

*همچنین لازم به ذکر است چنانچه سیگنال B (نشان دهنده دستور پرش) که به استیج اجرا وارد میشود یک باشد دستور پرش می باشد و آدرس پرش توسط مالتی پلکسر pc انتخال میشود و برای برطرف کردن مخاطره کنترلی دو رجیستر میانی اول و دوم flush میشوند (خروجیشان صفر میشود و به حباب تبدیل میشوند). در رجیستر میانی اول flush نسبت به freeze اولویت دارد.

توضیح ماژول های کد:

*ماژول های یک استیج را همگی کنار هم در فایل جدا نسبت به آن استیج قرار داده ایم که نام فایل به Modules ختم میشود. حال به توضیح این ماژول ها می پردازیم.

استیج اول:

PC_REG: pc_in ۳۲ بیتی و freeze را دریافت میکند. در posedge rst یا posedge clk اگر rst فعال باشد خروجی ۳۲ بیتی pc_out صفر میشود، اگر freeze فعال باشد تغییری نمیکند و در غیر اینصورت برابر pc_in میشود.

MUX_PC: اگر ورودی branch_taken یک باشد خروجی pc_in برابر ورودی branch_address و در غیر اینصورت pc4 میشود. ورودی ها و خروجی ۳۲ بیتی اند.

ADD4: ورودی ۳۲ بیتی pc_out را دریافت میگیرد و خروجی pc4 را برابر pc_out+4 قرار میدهد.

INST_MEM: حافظه دستور با ظرفیت ۱۰۲۴ دستور ۳۲ بیتی است که pc_out را به عنوان آدرس بایت میگیرد، ۲ واحد به راست شیفت میدهد تا آدرس دستور بدست آید و دستور را خوانده و در خروجی inst قرار میدهد. حافظه در ابتدا توسط فایل "instruction.txt" مقداردهی اولیه میشود.

استیج دوم:

REG_FILE: رجیستر فایل شامل ۱۵ رجیستر ۳۲ بیتی است که در ابتدا یا هنگام ریست هر رجیستر مقدارش برابر شماره اش میشود. مقدار رجیستر با شماره src1 در خروجی val_rn و مقدار رجیستر با شماره src2 در خروجی val_rm قرار میگیرد. در لبه پایین رونده و در صورت فعال بودن ورودی wb_wb_en(مربوط به استیج WB)، wb_value در رجیستر با شماره wb_dest قرار میگیرد.

CONTROL: برای سادگی signal را با تایپ reg تعریف میکنیم که همان کانکت شده‌ی خروجی های mem_read, mem_write, wb_en, b, s می باشد. حال اگر or_out یک باشد(هازارد رخ داده باشد یا شرط اجرای دستور برقرار نباشد) signal برابر 5'b0 و exe_cmd برابر 4'b0 میشود. در غیراینصورت اگر mode == 2'b10 دستور پرش است، signal برابر 5'b00010 میشود(فقط b یک میشود) و exe_cmd مهم نیست. در غیر اینصورت با توجه به mode و opcode دستور تعیین میشود و signal و exe_cmd برایش تعیین میشود. بیت s برای تمام دستورات برابر s_in میشود به جز دستورات tst و cmp که s همیشه ۱ است و دستورات ldr و str که s صفر است. بیت b فقط برای دستور پرش ۱ میشود. بیت wb_en برای تمام دستورات محاسباتی و mov و mvn ۱ میشود، برای دستور پرش و tst و cmp ۰ میشود و برای ldr و str برابر s_in میشود چون در دستور ldr بیت وضعیت ۱ است و باید wb_en و mem_read فعال باشند و mem_write غیرفعال؛ در دستور str قضیه برعکس است. پس بیت های mem_read و mem_write برای ldr و str به ترتیب برابر s_in و ~s_in میشوند و در تمام دستورات دیگر صفر میشوند. خروجی exe_cmd هم برای هر دستور محاسباتی یکتا تعیین میشود، برای tst مشابه and، برای cmp مشابه sub و برای ldr و str مشابه add تولید میشود.

CONDITION_CHECK: این ماژول ۴بیت cond را به عنوان شرط اجرای دستور و ۴ بیت status را به عنوان وضعیت فعلی میگیرد. اگر cond برابر 4’b1110 باشد خروجیcond_out یک میشود. در حالات دیگر cond، بیت مربوطه ی وضعیت چک میشود و اگر شرط برقرار بود cond_out یک میشود. برای سادگی، ۴ بیت status را به ۴ بیت n,z,c,v تفکیک کرده ایم.

MUX_SRC2: مالتی پلکسر با خروجی ۴بیتی src2 برای تعیین شماره اپرند دوم رجیسترفایل. اگر mem_write یک باشد rd و درغیراینصورت rm روی خروجی میرود.

استیج سوم:

ALU: واحد محاسبات با ورودی های ۳۲ بیتی in_1 و in_2 (علامتدار) و ۴ بیتی exe_cmd و تک بیت c_in و خروجی های ۳۲ بیتی result و ۴ بیتی status. دو wire signed با نام های c_in2 و c_in3 تعریف میکنیم که به ترتیب به {31’b0, c_in} و {31’b0,~c_in} assign میکنیم. اولی در دستور adc برای جمع با in1+in2 و دومی در subc برای تفریق از in1-in2 به کار میروند. خروجی status برای سادگی به چهار بیت n,z,c,v تفکیک شده است. در تمام دستورات به جز ۴ دستور جمع و تفریق، بیت c صفر است و فقط result از محاسبه مربوطه به دست می آید اما در دستورات جمع و تفریق حاصل، ۵ بیت است که بیت پرارزش در c و ۴بیت دیگر در result ذخیره میشوند. بیت n که نشان دهنده علامت است بیت ۳۱ result و بیت z که نشان دهنده صفر بودن نتیجه است از nor کردن تمام بیت های result بدست می آید. بیت v نشان دهنده سرریز، در دو دستور جمع زمانی یک میشود که بیت علامت دو ورودی ۱ باشد و بیت علامت خروجی ۰ (چون حاصل جمع دو عدد منفی مثبت شده است) یا برعکس و در دو دستور تفریق زمانی یک میشود که بیت علامت ورودی اول ۰ و ورودی دوم ۱ ولی خروجی ۱ باشد یا اینکه بیت علامت ورودی اول ۱ و ورودی دوم ۰ ولی خروجی ۰ باشد.

VAL2_GENERATE: ورودی های ۳۲ بیتی val_rm و ۱۲ بیتی shift_operand و تک بیتی imm و تک بیتی mem_flag (حاصل mem_read | mem_write) را دریافت میکند و خروجی ۳۲بیتی val2 را به عنوان عملگر دوم ALU تولید میکند. اسامی دیگری که در این توضیح به کار میرود به صورت wire تعریف میشوند. shift_operand را به rotate_imm ۴ بیتی و immed_8 ۸ بیتی تفکیک میکنیم. immed_8 را با عنوان immed_8_extended به ۳۲ بیت extend میکنیم. rotate_imm_val حاصل یک واحد شیفت به چپ rotate_imm هست(چون می خواهیم ۵ بیت شود و بتواند تا ۳۲ بیت شیفت کند). shift_mode برابر بیت ۶و۵ و shift_imm برابر بیت های ۷ تا ۱۱ shift_operand است. الان ۳ حالت داریم: ۱)mem_flag یک باشد: val2 برابر sign extend شده ی shift_operand میشود. ۲) در غیراین صورت اگر imm یک باشد: immed_8_extended را در کنار خودش قرار میدهیم تا ۶۴ بیت حاصل شود. حال به اندازه rotate_imm_val به راست شیفت میدهیم. به این ترتیب بیت هایی که هنگام شیفت از سمت راست نمونه راستی خارج میشوند، از نمونه چپی به نمونه راستی به عنوان بیت های پرارزش وارد میشوند. حال ۳۲ بیت کم ارزش حاصل در val2 قرار میگیرد. ۳) در غیر اینصورت در حالت شیفت قرار داریم (برای اطمینان بررسی می کنیم بیت shift_operand[4] صفر باشد) : حال اگر shift_mode برابر 2’b00یا 2’b01 باشد به ترتیب شیفت منطقی به چپ(>>) و راست(<<) می دهیم اگر 2’b10 باشد شیفت ریاضی به راست می دهیم(از سمت

چپ علامت وارد میشود) (<<<). اگر 2’b11 باشد مانند حالت(۲)(rotate)، دو نمونه از val_rm را کنار هم میگذاریم و به اندازه shift_imm به راست شیفت می دهیم و ۳۲ بیت کم ارزش در val2 قرار میگیرد.

ADDER32: یک جمع کننده ۳۲ بیتی که ورودی اولش pc و ورودی دومش signed_imm_24_extended است و خروجی اش آدرس پرش می باشد.

استیج چهارم:

این استیج تنها ماژول Memory را دارد که در فایل Memory.v قراردارد. حافظه داده با ظرفیت ۶۴ کلمه ۳۲ بیتی است که address را به عنوان آدرس بایت میگیرد، ۲ واحد به راست شیفت میدهد تا آدرس کلمه بدست آید. در صورت یک بودن سیگنال ورودی MEMread، کلمه متناظر را خوانده و در خروجی MEM_result قرار میدهد. همچنین در لبه بالارونده کلاک اگر سیگنال ورودی MEMwrite یک باشد ورودی ۳۲ بیتیdata را در آدرس متناظر مینویسد.

استیج پنجم:

فقط یک مالتی پلکسر ۳۲ بیتی دارد که در فایل WB_Stage.v قرار دارد. اگر MEM_R_en یک باشد خروجی out برابر ورودی MEM_result و در غیر این صورت برابر ALU_result میشود.

واحد تشخیص مخاطره:

*ابتدای سیگنال های مربوط به استیج اجرا Exe و استیج مموریMem آورده ایم. در چهار حالت خروجی hazard یک میشود:

۱) Exe_WB_EN (سیگنال نوشتن در رجیسترفایل که از استیج اجرا می آید) یک باشد و src1 برابر با Exe_Dest باشد و EXE_CMD مربوط به دستور mov و mvn نباشد(قسمت امتیازی: چون این دستورات src1 ندارند). ۲) مثل حالت (۱) با این تفاوت که سیگنال ها از استیج مموری می آیند (Mem_WB_EN و Mem_Dest). ۳) دستور دو منبعی باشد(برای چک کردن کافیست imm صفر باشد و دستور STR نباشد پس باید MEM_W_EN هم صفر باشد) و Exe_WB_EN یک باشد و src2 برابر با Exe_Dest باشد. ۴) مثل حالت (۳) با این تفاوت که سیگنال ها از استیج مموری می آیند (Mem_WB_EN و Mem_Dest).

*رجیستر های میانی در فایل های جدا قرار دارند که نامشان به Reg ختم میشود و وظیفه هدایت سیگنال ها از یک استیج به استیج بعدی را به صورت سنکرون بر عهده دارند. نیازی به توضیح بیشتر آنها دیده نمیشود.(flush و freeze را در سطح RTL توضیح دادیم)

* برای سنتز از خانواده Cyclone IV E و دیوایس EP4CE6E22C6 استفاده شده است.

گزارش کامپایل و سنتز

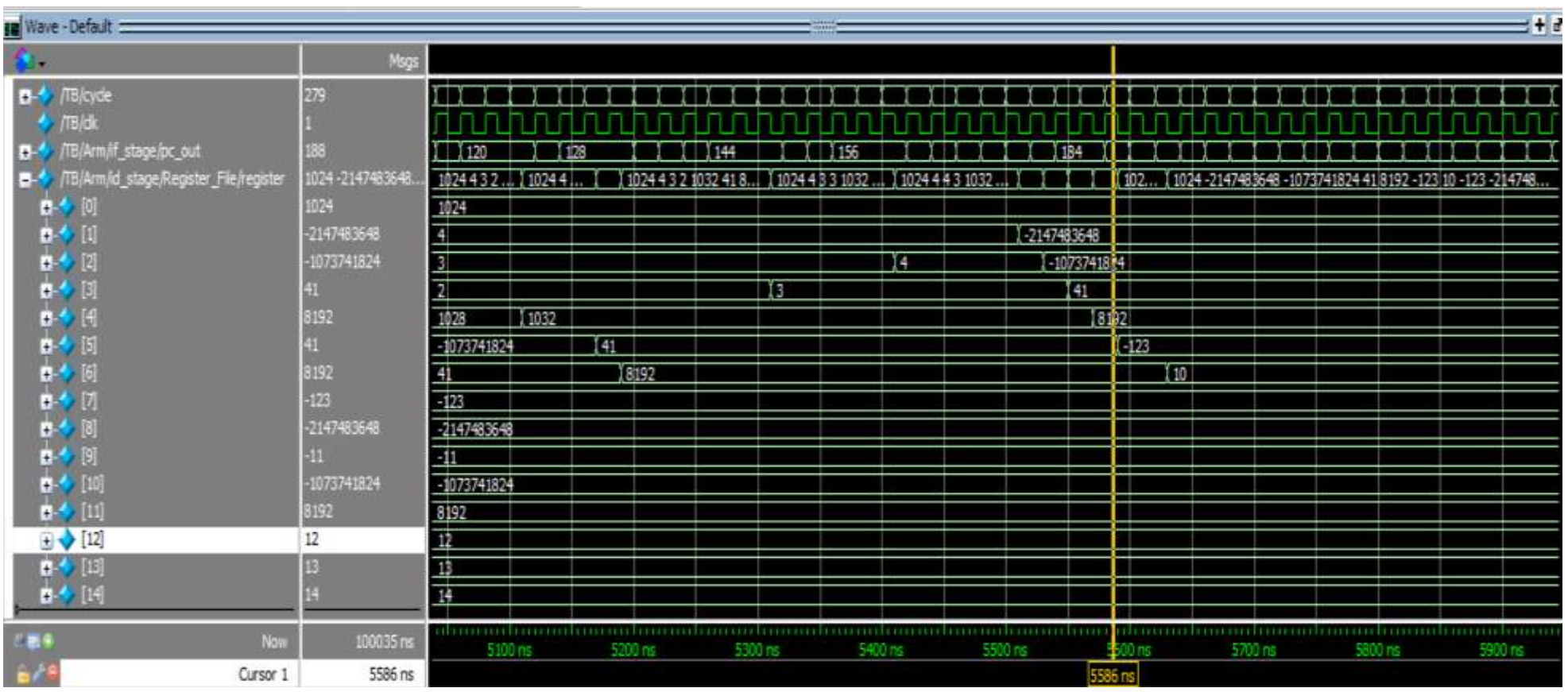
```
Type   ID   Message
  0 204019 Generated file ARM_min_1200mv_0c_fast.vo in folder "C:/Users/ACER/Desktop/Lab/ARM/Quartus/simulation/modelsim/" for EDA simulation tool
  0 204019 Generated file ARM.vo in folder "C:/Users/ACER/Desktop/Lab/ARM/Quartus/simulation/modelsim/" for EDA simulation tool
  0 204019 Generated file ARM_6_1200mv_85c_v_slow.sdo in folder "C:/Users/ACER/Desktop/Lab/ARM/Quartus/simulation/modelsim/" for EDA simulation tool
  0 204019 Generated file ARM_6_1200mv_0c_v_slow.sdo in folder "C:/Users/ACER/Desktop/Lab/ARM/Quartus/simulation/modelsim/" for EDA simulation tool
  0 204019 Generated file ARM_min_1200mv_0c_v_fast.sdo in folder "C:/Users/ACER/Desktop/Lab/ARM/Quartus/simulation/modelsim/" for EDA simulation tool
  0 204019 Generated file ARM_v.sdo in folder "C:/Users/ACER/Desktop/Lab/ARM/Quartus/simulation/modelsim/" for EDA simulation tool
> 0        Quartus Prime EDA Netlist Writer was successful. 0 errors, 1 warning
  0 293000 Quartus Prime Full Compilation was successful. 0 errors, 223 warnings
```

```
Transcript
# Compile of IF_Stage_Reg.v was successful.
# Compile of MEM_reg.v was successful.
# Compile of Memory.v was successful.
# Compile of StatusRegister.v was successful.
# Compile of TB.v was successful.
# Compile of WB_Stage.v was successful.
# 16 compiles, 0 failed with no errors.

ModelSim>
```

CPI	Execution Time	Dedicated logic registers	Total combinational functions	Total logic elements
279/170 = 1.64	279	1525/6272 (24%)	2380/6272 (38%)	2856/6272 (46%)

تصویر حاصل از شبیه سازی نشان دهنده درستی کد



مشکلات هنگام برنامه نویسی:

۱-در محاسبه دستورات adc و sbc مشکلی در استفاده از carry_in وجود داشت که با تعریف دو وایر میانی به شکل مقابل مشکل حل شد. C_in2 در adc و c_in3 در sbc استفاده میشود.

```
assign c_in2={31'b0,c_in};
assign c_in3={31'b0,~c_in};
```

۲-هنگام کامپایل با مشکلات متعددی در زمینه اتصال سیم ها در ماژول اصلی ARM روبه رو شدیم که با دقت بیشتر این مشکلات حل شدند.

۳-هنگام سنتز در فایل حافظه دستور روی ’_‘ بین دستورات ارور گرفته میشود که آنها را برداشتیم و درست شد. همینطور مشکل کوچکی در رجیستر فایل هنگام چک کردن شرط نوشتن بود که بعلت استفاده از if به جای else if در صورت برقرار نبودن شرط if(rst) بود و حل شد.

۴-علیرغم سنتز موفق به علت نداشتن هیچ گونه خروجی در ماژول سطح بالا، کوارتس کل مدار را به علت بهینه سازی حذف کرد و استفاده از المان های منطقی صفر بود. این مشکل را با تعریف یک خروجی برای ماژول سطح بالا (خروجی دادن دستور خوانده شده از حافظه دستور) حل کردیم.

قسمت Forwarding :

خواسته اول : کد Verilog معادل با RTL طراحی شده توضیح داده شود و نتایج شبیه سازی برای نشان دادن درستی کد آورده شود:

برای طراحی قسمت فورواردینگ ما باید تغییراتی در مسیر داده ها و شیوه کنترل hazard می دادیم و یک ماژول کنترل کننده این فرآیند باید طراحی می شد. این مراحل در ادامه توضیح داده خواهدشد.

```
94  module MUX_ALU1(alu_src1, val_rn, alu_res, wb_value, sel_src1);
95          input [31:0]val_rn, alu_res, wb_value;
96          output [31:0] alu_src1;
97          input [1:0] sel_src1;
98
99          assign alu_src1=(sel_src1==2'b00)? val_rn : (sel_src1==2'b01)? alu_res :(sel_src1==2'b10)? wb_value : val_rn;
100 endmodule
101 ////////////
102 module MUX_ALU2(val2_src, val_rm, alu_res, wb_value, sel_src2);
103         input [31:0]val_rm, alu_res, wb_value;
104         output [31:0] val2_src;
105         input [1:0] sel_src2;
106
107         assign val2_src=(sel_src2==2'b00)? val_rm : (sel_src2==2'b01)? alu_res : (sel_src2==2'b10)? wb_value : val_rm;
108 endmodule
109
```

کد بالا توصیف دو MUX سه ورودی هست. MUX_ALU1 ، ورودی نخست alu را تعیین خواهد کرد و MUX_ALU2 نیز ورودی داده ۳۲ بیت val_generator را تعیین می کند.

در mux اول هنگامی که سلکت ماکس ۰ باشد، val_rn آمده از استیج ID را درایو می کند، هنگامی سلکت ۰۱ باشد، خروجی alu که از استیج Mem آمده را درایو می کند و در نهایت اگر سلکت ۱۰ باشد، داده ۳۲ بیت برگشت داده شده از استیج WB (که همان مقداری هست که باید در رجیسترفایل نوشته شود) را درایو خواهدکرد.

در mux دوم نیز هنگامی که سلکت ماکس ۰ باشد، val_rm آمده از استیج IF را درایو می کند، هنگامی سلکت ۰۱ باشد، خروجی alu که از استیج Mem آمده را درایو می کند و در نهایت اگر سلکت ۱۰ باشد، داده ۳۲ بیت برگشت داده شده از استیج WB (که همان مقداری هست که باید در رجیسترفایل نوشته شود) را درایو خواهدکرد.

```
23          MUX_ALU2 ma2(val2_src, Val_Rm, alu_res, wb_value, sel_src2);
24          VAL2_GENERATE Val2_Generate(val2_src, Shift_operand, imm, mem_flag, val2);
25  //check for val2 src
26
27          wire c_in;
28          assign c_in = SR[1];
29
30          wire [31:0] alu_src1;
31          MUX_ALU1  ma1(alu_src1, Val_Rn, alu_res, wb_value, sel_src1);
32
33          ALU ALU(alu_src1, val2, EXE_CMD, c_in, ALU_result, status);
34
```

در نهایت این دو مالتیپلکسر در استیج EX نمونه گیری شده اند و مطابق کد بالا سیم هایی که در بخش قبل توضیحشان داده شد به آنها وصل شده. توجه داشته باشید که که دو سیم sel_src1 و sel_src2 که همان سلکتور های ماکس ها هستند، توسط ماژول forwarding درایو خواهندشد.

اکنون به توضیح ماژول فورواردینگ می پردازیم:

```
module Forwarding_Unit (src1, src2, MEM_Dest, MEM_WB_EN, WB_Dest, WB_WB_EN, Sel_src1, Sel_src2);
input [3:0] src1;
input [3:0] src2;
input [3:0] MEM_Dest;
input MEM_WB_EN;
input [3:0] WB_Dest;
input WB_WB_EN;
output [1:0] Sel_src1;
output [1:0] Sel_src2;

        assign Sel_src1 = ((MEM_WB_EN == 1'b1) && (src1 == MEM_Dest))? 2'b01:
                          ((WB_WB_EN == 1'b1) && (src1 == WB_Dest))? 2'b10:
                           2'b00;

        assign Sel_src2 = ((MEM_WB_EN == 1'b1) && (src2 == MEM_Dest))? 2'b01:
                          ((WB_WB_EN == 1'b1) && (src2 == WB_Dest))? 2'b10:
                           2'b00;

endmodule
```

این واحد، ۶ ورودی می گیرد و در ازای آن ، وضعیت سلکت ماکس ها را مشخص می کند؛ دو ورودی src1 و src2 که همان آدرس رجیسترها هستند، از استیج ID رجیستر شده و وارد می شوند. mem_dest، آدرس مقصدی هست که در استیج mem مستقیم وارد می شود. wb_dest نیز همان آدرس مقصد هست و مستقیم از استیج wb آورده می شود. دو سیگنال mem_wb_en و wb_wb_en نیز که مشخص کننده وضعیت نوشتن داده در رجیسترفایل هستند، از استیج های متناظرشان ، مستقیما ، آورده خواهندشد.

شروطی که برای تعیین مقدار sel ها بررسی می شود ، یکسان هستند، در وهله اول می بینیم که آیا سورسی که در دستور فعلی در استیج ex هست با مقصد دستور قبلی که الان در استیج mem هست باهم برابرند یا خیر؟ درصورتی برابری sel برابر ۰۱ خواهدشد.

در شرط دوم می بینیم که آیا سورسی که در دستور فعلی در استیج ex هست با مقصد دو دستور قبل تر که الان در استیج wb هست باهم برابرند یا خیر؟ درصورتی برابری sel برابر ۱۰ خواهدشد.

توجه شود که در همه حالات فعال بودن سیگنال نوشتن نیز چک می شود. هم چنین اولویت برای forwarding با دستور نزدیک تر می باشد.

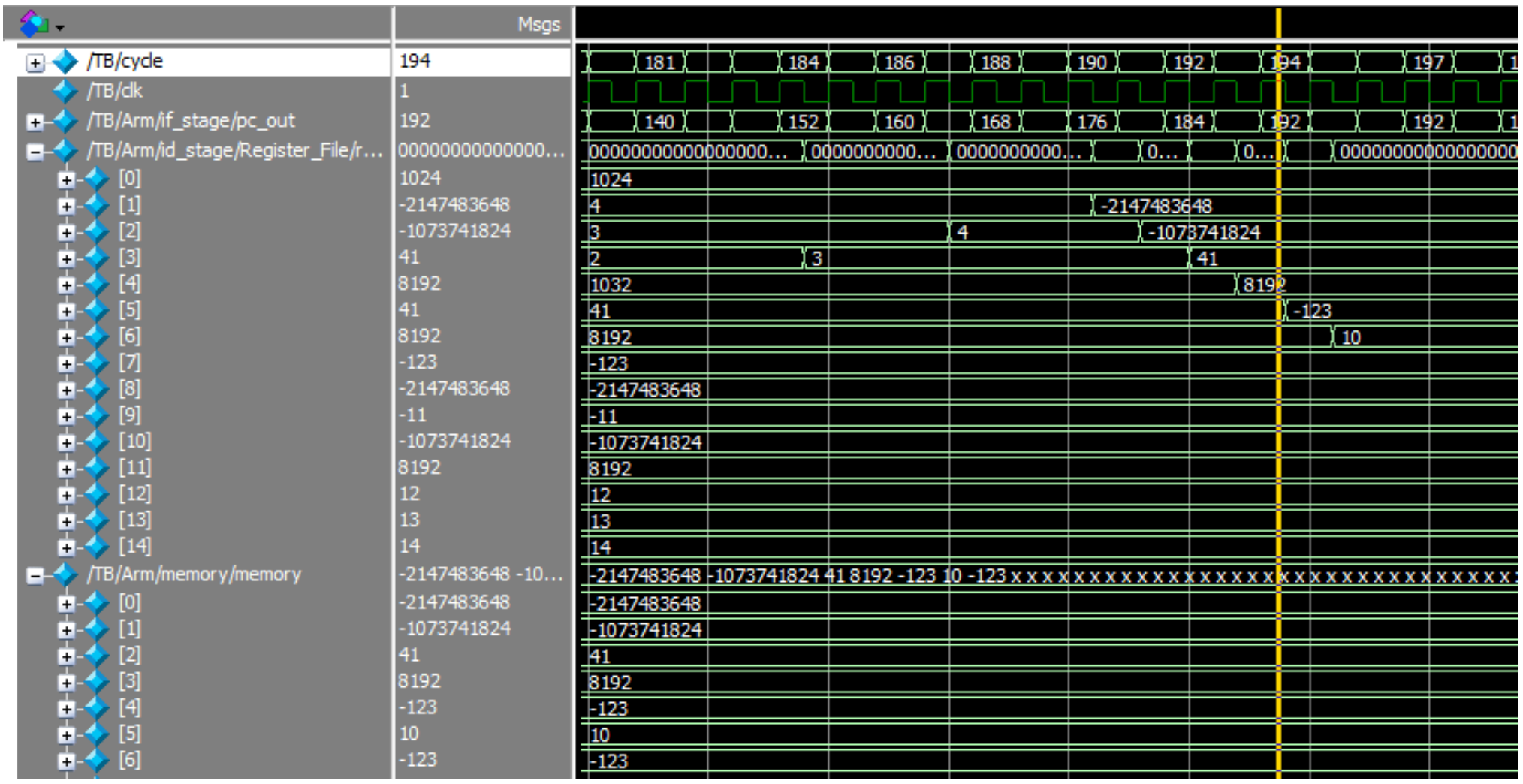
ما تغییراتی در ماژول هازارد نیز دادیم:

```
if(!FW) begin
        if(Exe_WB_EN & (src1 == Exe_Dest) & (EXE_CMD != 4'b1001) & (EXE_CMD != 4'b0001)) hazard_Detected = 1'b1;
        else if(Mem_WB_EN & (src1 == Mem_Dest) & (EXE_CMD != 4'b1001) & (EXE_CMD != 4'b0001)) hazard_Detected = 1'b1;
        else if(Exe_WB_EN & (src2 == Exe_Dest) & (~imm) & (~MEM_W_EN)) hazard_Detected = 1'b1;
        else if(Mem_WB_EN & (src2 == Mem_Dest) & (~imm) & (~MEM_W_EN)) hazard_Detected = 1'b1;
        else hazard_Detected = 1'b0;
        end
else if(FW)
        begin
        if ( (ID_EX_MEM_R_EN && Exe_Dest == src1) | ( ((~imm) && (~MEM_W_EN)) && Exe_Dest == src2)))  hazard_Detected = 1'b1;
        else hazard_Detected = 1'b0;
        end
end
```

سیگنال FW یک ورودی هست که توسط کاربر تعیین می شود، اگر یک باشد پردازنده در حالت forwarding کار می کند در غیر این صورت فورواردینگ غیرفعال هست.

با استفاده از تکنیک فورواردینگ پیاده سازی شده ، اکثر حالات مداخله برطرف شده اند اما مداخله در دستورات خواندن یا همان load برطرف نخواهدشد! به همین علت ، زمانی که آدرس مقصد استیج ex (در مود خواندن) با آدرس سورس۱ یا آدرس سورس۲ استیج دیکدینگ برابر باشد، سیگنال هازارد فعال می شود، توجه شود که برای src2 حتما باید دو منبعی بودن دستور چک شود. این حالت معادل این هست که یک دستور خواندن داشته باشیم و منبع دستور بعدی با مقصد دستور خواندن یکی شده باشد.

در ادامه نتایج شبیه سازی در مودلسیم آورده می شود:



همان طورکه مشاهده می شود در پایان برنامه مقادیر رجیسترها و خانه های حافظه کاملا صحیح می باشند، همچینین باتوجه به شکل ، برنامه در طول 194 سیکل انجام شده است.

خواسته ۲و۳و۴ : نتایج سنتز آورده شود و میزان افزایش کارایی را با آزمایش دوم) بدون ارسال به جلو(مقایسه کنید)میزان بهبود کارایی را محاسبه کنید.)

میزان هزینه سختافزاری را محاسبه کنید)درصد افزایش استفاده از المان های منطقی.)

میزان کارایی بر هزینه (Cost per Performance)را محاسبه کنید.

فورورادینگ دارد؟	بله	خیر
تعداد سیکل اجرا برنامه	۱۹۴	۲۷۹

مطابق با جدول بالا تعداد سیکل های پردازنده ، 30% بهبود داشته است.

در جدول بعد نتایج حالت با فورواردینگ و بدون پس از سنتز آورده شده است:

حالت	CPI	Execution Time	Dedicated logic registers	Total combinational functions	Total logic elements
معمولی	279/170 = 1.64	279	1525/6272 (24%)	2380/6272 (38%)	2856/6272 (46%)
forwarding	194/170= 1.14	194	750/6272 (12%)	2042/6272 (32%)	2509/6272 (40%)

	Resource	Usage
1	Estimated Total logic elements	2,509
2		
3	Total combinational functions	2042
4	> Logic element usag...mber of LUT inputs	
5		
6	v Logic elements by mode	
1	-- normal mode	1927
2	-- arithmetic mode	115
7		
8	v Total registers	750
1	-- Dedicated logic registers	750
2	-- I/O registers	0
9		
10	I/O pins	35
11	Total memory bits	2048

همان طورکه میبینید، المان های منطقی به اندازه ۶ درصد کمتر استفاده شده اند. مقدار CPI نسبت به حالت قبل حدود ۳۰ درصد بهترشده است. برای محسابه میزان پرفورمنس بر هزینه ، ابتدا ماکسیمم فرکانس کاری مدار را مشاهده می کنیم:

	Fmax	Restricted Fmax	Clock Name
1	30.29 MHz	30.29 MHz	clk

پس فرکانس مدار ۳۰ مگاهرتز هست، بنابراین ، 30000/2509=11900 برابر با مقدار کارایی بر هزینه هست.

مشکلات و خطاهای رفع شده:

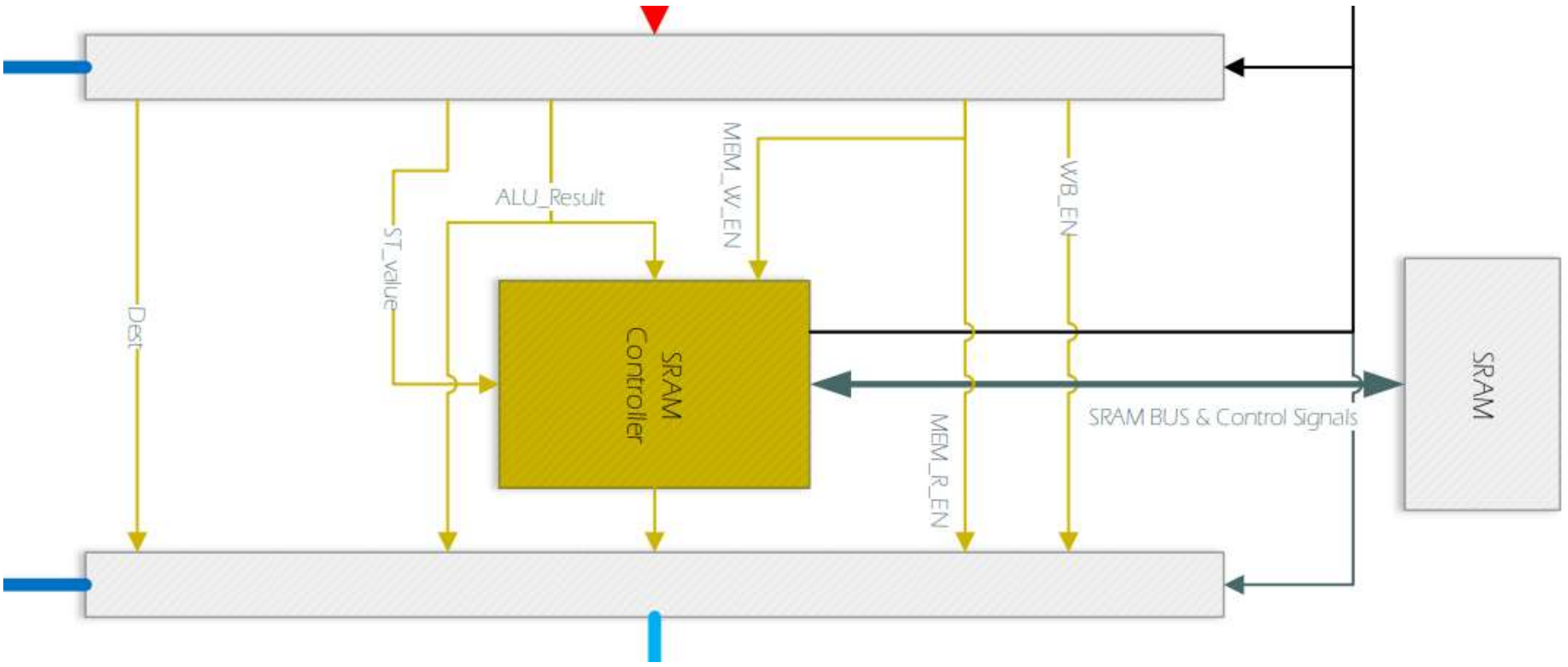
ما در کد اولیه اشتباها src1 و src2 را مستقیم متصل کرده بودیم ، در حالی که می بایست ، از رجیستر استیج ID رد می شد.

در قسمت نمونه گیری ماژول ها در top level ، سیم mem_r_en مخصوص استیج ex که باید وارد هازارد یونیت شود ، جابجا وصل شده بود و این موضوع باعث فعال شدن اشتباه هازارد شده بود.

قسمت SRAM :

خواسته ۱ : در ابتدای گزارش کار درباره معماری پردازنده با تغیرات اعمال شده در سطح RTL و قسمت های اضافه شده توضیحات کاملی نوشته شود.

ما برای پیاده سازی این قسمت ، تغییراتی در استیج mem داده ایم، شماتیک تغییرات این استیج در شکل زیر آورده شده است:



مطابق شکل ما دو ماژول کنترلر و خود sram را تعریف کردیم، توضیحات در ادامه می آید

```
2   module SRAM(CLK, RST, SRAM_WE_N, SRAM_ADDR, SRAM_DQ);
3           input CLK, RST, SRAM_WE_N;
4           input [16:0]SRAM_ADDR;
5           inout [31:0]SRAM_DQ;
6
7           reg [31:0]memory[0:511];
8
9           assign #30 SRAM_DQ = SRAM_WE_N ? memory[SRAM_ADDR] : 32'bz;
10
11          always@(posedge CLK) begin
12                  if(~SRAM_WE_N) begin
13                          memory[SRAM_ADDR] = SRAM_DQ;
14                  end
15          end
16  endmodule
```

حافظه sram ، یک حافظه ۵۱۲ خانه ای هست که هرخانه ۳۲ بیت گنجایش داده دارد(معادل با ۲ کیلوبایت) . برای نوشتن و خواندن داده ، یک پورت دوطرفه ۳۲ بیت SRAM_DQ درنظرگرفته شده است. این پورت به کنترلر متصل می شود، پس خواندن و نوشتن به صورت همزمان غیر ممکن هست.

سیگنال ورودی SRAM_WE_N وضعیت خواندن/نوشتن را مشخص می کند، در صورت یک بودن ، به معنای خواندن و در صورت صفر بودن به معنای نوشتن در حافظه هست.

عملیات خواندن به صورت ترکیبی و با تاخیر ۳۰ نانوثانیه انجام می شود و عملیات نوشتن ، با لبه کلاک مخصوص SRAM انجام خواهدشد. توجه کنید که فرکانس sram نصف فرکانس پردازنده هست.

```
module SRAM_Controller(clk, rst, write_en, read_en, address, writeData, readData, ready, SRAM_DQ, SRAM_ADDR, SRAM_WE_N);
        input clk, rst, write_en, read_en;
        input [31:0]address, writeData;
        output [31:0]readData;
        output ready;
        inout [31:0]SRAM_DQ;
        output [16:0]SRAM_ADDR;
        output SRAM_WE_N;

        wire [31:0]shifted_addr;
        reg [2:0]count;

        assign shifted_addr = (address - 1024) >> 2;
        assign SRAM_ADDR = shifted_addr[16:0];
        assign SRAM_WE_N = ~write_en;
        assign SRAM_DQ = write_en ? writeData : 32'bz;
        assign readData = read_en ? SRAM_DQ : 32'bz;
        assign ready = ~((read_en | write_en) & (count < 5));

        always@(posedge clk) begin
                if(read_en | write_en) begin
                        if(count == 5)
                                count <= 0;
                        else
                                count <= count + 3'b1;
                end
                else
                        count <= 3'b0;
        end

endmodule
```

ماژول کنترلر وظیفه زمان بندی درست ارسال و دریافت داده و آماده کردن داده و آدرس جهت ارسال به حافظه را دارد، هم چنین پردازنده را از آماده بودن یا نبودن داده در جریان می گذارد.

آدرسی که وارد کنترلر می شود ، یک آدرس خام است، در خطوط ۱۳ و ۱۴ ، با کم کردن آفست ۱۰۲۴ و شیفت دو واحد و برداشتن ۱۷ بیت با ارزش، آن را به سیم SRAM_ADDR می دهیم تا به sram برود.

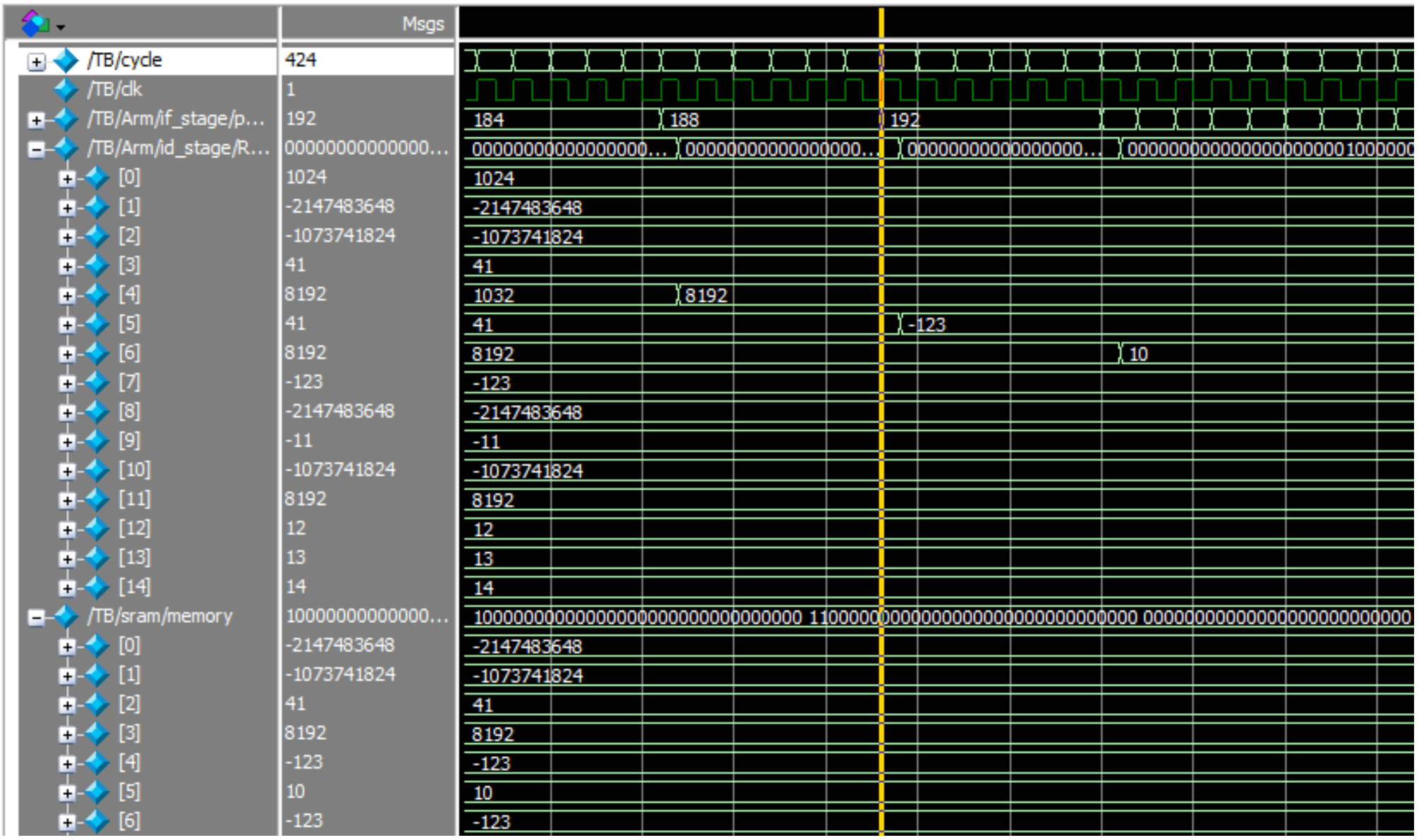
sram همواره در حال خواندن هست مگر اینکه سیگنال نوشتن بیاید ، توجه کنید که سیگنال write که از پردازنده می آید به صورت نقیض شده وارد sram شده است.

فعالیت واحد حافظه باتوجه به کندبودن آن ، باید ۶ کلاک طول بکشد ، به همین ما یک کانتر تعبیه کرده ایم که شش واحد میشمارد ، کانتر درصورت بودن درخواست نوشتن یا خواندن شروع به شمارش خواهدکرد. در حین این شمارش پردازنده باید متوقف شود ، به همین علت سیگنال ready در حین شمارش صفر هست و زمانی که شمارش به پایان برسد مجدد یک خواهدشد.

این سیگنال ready برای توقف پردازنده به رجیستر PC و رجیسترهای واسط IF ، ID و EX می رود و اگر صفر بود، تمام رجیسترهای اشاره شده را فریز خواهدکرد.

باتوجه به این که یک bus دوطرفه بین کنترلر و حافظه اش هست ، جهت انتقال داده توسط سیم کنترلی نوشتن یا write تعیین خواهدشد ، پس در زمان نوشتن داده ، کنترلر این bus را درایو می کند و در غیر این صورت حافظه درایو می کند.

اکنون شبیه سازی مدار را در مودلسیم انجام می دهیم:



مقادیر رجیسترها و حافظه صحیح می باشند، همان طورکه میبینید ، اجرای برنامه حدود 425 سیکل طول کشیده است.

خواسته ۲ و ۳ : نتایج برنامه ریزی روی برد را توضیح دهید و میزان کارایی پردازنده را با حالتی که از حافظه داخلی استفاده میشد، مقایسه کنید. در قسمت بعد نتایج سنتز آورده شود، و هزینه سخت افزار نسبت به حالتی که از حافظه داخلی استفاده میشد، مقایسه شود.

حالت پردازنده	معمولی	forwarding	SRAM
تعداد سیکل اجرا برنامه	۲۷۹	۱۹۴	۴۲۵

تعداد سیکل های پردازنده نسبت به حالت farwarding 119% و نسبت به حالت معمولی ، 52% افزایش داشته است.

نرخ CPI در ادامه خواهدآمد.

حالت	CPI	Execution Time	Dedicated logic registers	Total combinational functions	Total logic elements
معمولی	279/170 = 1.64	279	1525/6272 (24%)	2380/6272 (38%)	2856/6272 (46%)
forwarding	194/170= 1.14	194	750/6272 (12%)	2042/6272 (32%)	2509/6272 (40%)
SRAM	425/170= 2.5	425	817/6272 (13%)	2191/6272 (34%)	2690/6272 (42%)

	Resource	Usage
1	Estimated Total logic elements	2,690
2		
3	Total combinational functions	2191
4	Logic element usag...mber of LUT inputs	
1	-- 4 input functions	1540
2	-- 3 input functions	491
3	-- <=2 input functions	160
5		
6	Logic elements by mode	
1	-- normal mode	2076
2	-- arithmetic mode	115
7		
8	Total registers	817
1	-- Dedicated logic registers	817
2	-- I/O registers	0

	Name	Value
1	Number of entity instances	1
2	Entity Instance	SRAM:sram|altsyncram:memory_rtl_0
1	-- OPERATION_MODE	DUAL_PORT
2	-- WIDTH_A	32
3	-- NUMWORDS_A	256
4	-- OUTDATA_REG_A	UNREGISTERED
5	-- WIDTH_B	32
6	-- NUMWORDS_B	256
7	-- ADDRESS_REG_B	CLOCK1
8	-- OUTDATA_REG_B	UNREGISTERED
9	-- RAM_BLOCK_TYPE	AUTO
10	-- READ_DURING_WRITE_MODE_MIXED_PORTS	DONT_CARE

در شکل بالا می بینید که حافظه sram را به یکی از lpm های کوارتوس ، معادل کرده است، علت ۲۶۵ کلمه ای شدن آن در قسمت خطاها توضیح داده شده است.

با وجود sram ، هزینه المان های منطقی در حدود ۲ درصد افزایش یافته که قابل چشم پپوشی هست ، اما به ازای آن از حافظه های fpga استفاده بیشتری شده است به طوری که sram به memory block های برد نگاشت شده است.

از سوی دیگر متاسفانه نرخ cpi بیش از دوبرابر نسبت به حالت قبل شده است.

	Fmax	Restricted Fmax	Clock Name
1	32.8 MHz	32.8 MHz	clk
2	576.04 MHz	250.0 MHz	SRAM_CLK

باتوجه به ماکسیمم فرکانس مدار یعنی ۳۲ مگاهرتز ، نسبت کارایی به هزبنه برابر است با 32000000/2690=11890

خطاها و مشکلات:

در هنگام سنتز ، نرم افزار این ارور را داد که تعداد خانه های حافظه sram در کد بیش از ظرفیت دستگاه هست و امکان پیاده سازی نیست، ما نیز به همین جهت حافظه sram را که ۵۱۲ خانه ۳۲ بیتی را داشت را به ۲۵۶ خانه تقلیل دادیم و خطا برطرف شد!

قسمت CACHE :

ماژول حافظه کش ، از بلاک متعددی ساخته شده است ، هم چنین یک کنترلر نیز جریان داده و ارتباط کش با sram و پردازنده را کنترل می کند، در این پیاده سازی پردازنده تنها با کش ارتباط می گیرد و ارتباط مستقیم با sram ندارد.

```
 9        reg valid_left[0:63];
10        reg valid_right[0:63];
11
12        reg [9:0]tag_left[0:63];
13        reg [9:0]tag_right[0:63];
14
15        reg[31:0]word_high_right[0:63];
16        reg[31:0]word_low_right[0:63];
17        reg[31:0]word_high_left[0:63];
18        reg[31:0]word_low_left[0:63];
19
20        reg lru[0:63];
21
22        reg[63:0]data_out;
23        reg[31:0]word_out;
24        reg hit, miss, s_left, s_right;
```

در شکل بالا ، بلوک های داده آمده اند، tag_left , tag_right دو ساختمان ۱۰ بیتی ۶۴ خانه ای هستند که تگ آدرس را نگه می دارند، یکی برای way راست و دیگری برای way چپ.

ما دو ساختان کلمه برای way راست و way چپ داریم، هر ساختمان خود شامل دو کلمه high و low می باشد.

هم چنین یک ساختمان تک بیتی برای نگه داری وضعیت سیاست جایگزینی هر ردیف cache به نام lru تعریف شده است.

در ابتدای شکل نیز دو ساختمان تک بیت ۶۴ خانه ای یکی برای way چپ و دیگری برای way راست ، اماده که معتبر بودن یا نبودن داده موجود در آن سمت ردیف متناظر را نشان می دهد.

کش یک خروجی ۶۴ بیتی به نام data_out می دهد و در ادامه باتوجه به وضعیت word offset ، ۳۲ بیت بارزش یا کم ارزش آن انتخاب شده و در word_out ریخته خواهدشد.

```
34  ////////////////////////////////////
35          always @(*) begin
36                  if(  ( tag_left[addr[8:3]]==addr[18:9] ) & valid_left[addr[8:3]] & mem_r_en  ) s_left=1'b1;
37                          else s_left=1'b0;
38
39                  if(  ( tag_right[addr[8:3]]==addr[18:9] ) & valid_right[addr[8:3]] & mem_r_en  ) s_right=1'b1;
40                          else s_right=1'b0;
41
42                  if( {s_left,s_right}==2'b10 ) data_out={ word_high_left[addr[8:3]], word_low_left[addr[8:3]] };
43                          else if( {s_left,s_right}==2'b01 ) data_out={ word_high_right[addr[8:3]], word_low_right[addr[8:3]] };
44                                  else data_out=64'bz;
45
46                  if(addr[2]) word_out=data_out[63:32];
47                          else word_out=data_out[31:0];
48
49
50                  hit=s_left | s_right;
51                  miss=~hit;
52          end
```

شکل بالا رویه خواندن از کش را نشان می دهد، ابتدا باید گفت در یک آدرس ۳۲ بیتی ، ۲ بیت اول برای byte addressability پردازنده و مضرب ۴ بودن دستورات هست. بیت سوم برای word offset هست. چون کش ما ۶۴

ردیف دارد ، بنابراین ۶ بیت بعدی آدرس برای مشخص کردن ردیف کش است. در انتها ۱۰ بیت باقی مانده به عنوان تگ آدرس به کار می رود.

در کد بالا ابتدا به سراغ ردیف متناظر آدرس در کش می رویم، چک می کنیم که کدام یک از تگ های راست و چپ این ردیف با تگ آدرس برابرند؟ و همچنین آیا این داده معتبر هست یا خیر؟ در صورت پاسخ مثبت هر دوسال ، سیگنال select آن way یک خواهدشد. توجه داشته باشید که باتوجه به سیاست جایگزینی ، یک شدن همزمان select هردو way راست و چپ غیرممکن می باشد.

سپس درصورتی که سلکت راه راست فعال بود ، ۶۴ بیت داده متناظرش را خروجی می دهیم و اگر راه چپ نیز هم چنین، اگر آدرس ورودی با هیچ تگی مطابقت نداشت ، خروجی کش z خواهدبود.

در انتها با توجه بخ بیت wordoffset ، ۳۲ بیت با ارزش یا کم ارزش داده خروجی مشخص می شود.

اگر یکی از سلکت ها یک شده باشد به معنی وجود داده مورد انتظار در کش هست ، پس به همین علت سیگنال hit یک خواهدشد، در غیراین صورت صفر هست.

```
54          integer i;
55          initial begin
56                  for( i=0; i<64; i=i+1 ) begin
57                          valid_left[i]=1'b0;
58                          valid_right[i]=1'b0;
59                          lru[i]=1'b1;     end
60          end
```

در ابتدای اجرای برنامه باتوجه به کد بالا ، مقادیر پیشفرض valid صفر هستند و مقادیر پیشفرض سیاست جایگزینی یک هست(به این معنی که ابتدا برای نوشتن در کش به سراغ way راست خواهیم رفت)

```
62  always @(posedge clk) begin
63          if( ~hit & mem_r_en & ready_sram ) begin
64                  if( ~lru[addr[8:3]] ) begin
65                          { word_high_left[addr[8:3]], word_low_left[addr[8:3]] }=readData;
66                          tag_left[addr[8:3]]=addr[18:9];
67                          valid_left[addr[8:3]]=1'b1;
68                          lru[addr[8:3]]=1'b1;
69                          end
70                  else if ( lru[addr[8:3]] ) begin
71                          { word_high_right[addr[8:3]], word_low_right[addr[8:3]] }=readData;
72                          tag_right[addr[8:3]]=addr[18:9];
73                          valid_right[addr[8:3]]=1'b1;
74                          lru[addr[8:3]]=1'b0;
75                          end
76          end
77          else if(mem_w_en) begin
78                                  if(tag_right[addr[8:3]]==addr[18:9]) begin valid_right[addr[8:3]]=1'b0; tag_right[addr[8:3]]=10'bz; end
79                                  else if(tag_left[addr[8:3]]==addr[18:9]) begin valid_left[addr[8:3]]=1'b0; tag_left[addr[8:3]]=10'bz; end
80          end
81  end
```

شکل بالا رویه نوشتن در کش را نشان می دهد. در این برنامه نوشتن فقط در یک حالت رخ می دهد: حالتی که پردازنده دستور load فرستاده باشد و داده آدرس مورد نظر در کش پیدا نشود، در این حالت به سراغ sram رفته و داده او را بر می داریم و به پردازنده می دهیم و نهایتا یک کپی از آن در کش می نویسیم.

پس به طور خلاصه وقتی وارد فاز نوشتن می شویم که سه شرط باشد: دستور mem_r فعال باشد(خواندن) و کش miss شده باشد و کار sram تمام شده باشد یعنی ready_sram یک باشد. سپس باتوجه به وضعیت بیت سیاست جایگزینی ، یکی از way های راست یا چپ را انتخاب می کنیم و ۶۴ بیت داده آمده از sram و تگ آدرسش را قسمت

داده و تگ کش می نویسیم. هم چنین بیت valid متناظر را نیز یک می کنیم . در انتها سیاست جایگزینی آن ردیف کش آپدیت کرده و نقیض می کنیم.

حال اگر از طرف پردازنده یک دستور store آمده باشد ، کش با آن کاری ندارد و به sram خواهدرفت. باتوجه به این موضوع پس اگر داده ای در آدرس متناظر کش باشد ، دیگر معتبر نخواهد بود. در این جا نیز ما هر وقت سیگنال write دیدیم، می رویم چک می کنیم که آیا تگ آدرس با تگ کش برابر هست؟ اگر بود بلافاصله valid آن از بین خواهدرفت.

```
89      assign ready= (hit | ready_sram) | ~(mem_r_en | mem_w_en) ;
90      assign rdata=hit ? word_out : addr[2]? readData[63:32] : readData[31:0];
```

در این قسمت نیز پردازنده در مواقعی باید متوقف شود، هنگامی که یک miss رخ می دهد ما ۷ کلاک توقف داریم : یک کلاک برای کش + ۶ کلاک برای sram. هم چنین هنگام نوشتن داده نیز ۷ کلاک توقف داریم : ۱ کلاک برای کش + ۶ کلاک برای sram . در انتها در هنگام رخ دادن hit ، توقفی صورت نمی گیرد و همان یک کلاک عادی هست.

باتوجه به این نکته ، پس یک سیگنال ready از کش داریم ؛ این سیگنال در زمان های hit یک هست ، اگر hit نداشتیم هنگامی که کار sram (ready_sram) تمام شد نیز یک می شود. در حالات غیر خواندن و نوشتن همیشه یک می ماند.

خروجی داده ۳۲ بیتی که از cache به پردازنده می رود ، دو حالت دارد: یا از خود کش مستقیم آمده است و یا از sram آورده شده است. این موضوع با سیگنال hit مشخص خواهدشد.

```
83      always@(posedge clk) begin
84              sram_addr <= addr;
85              sram_wdata <= wdata;
86      end
```

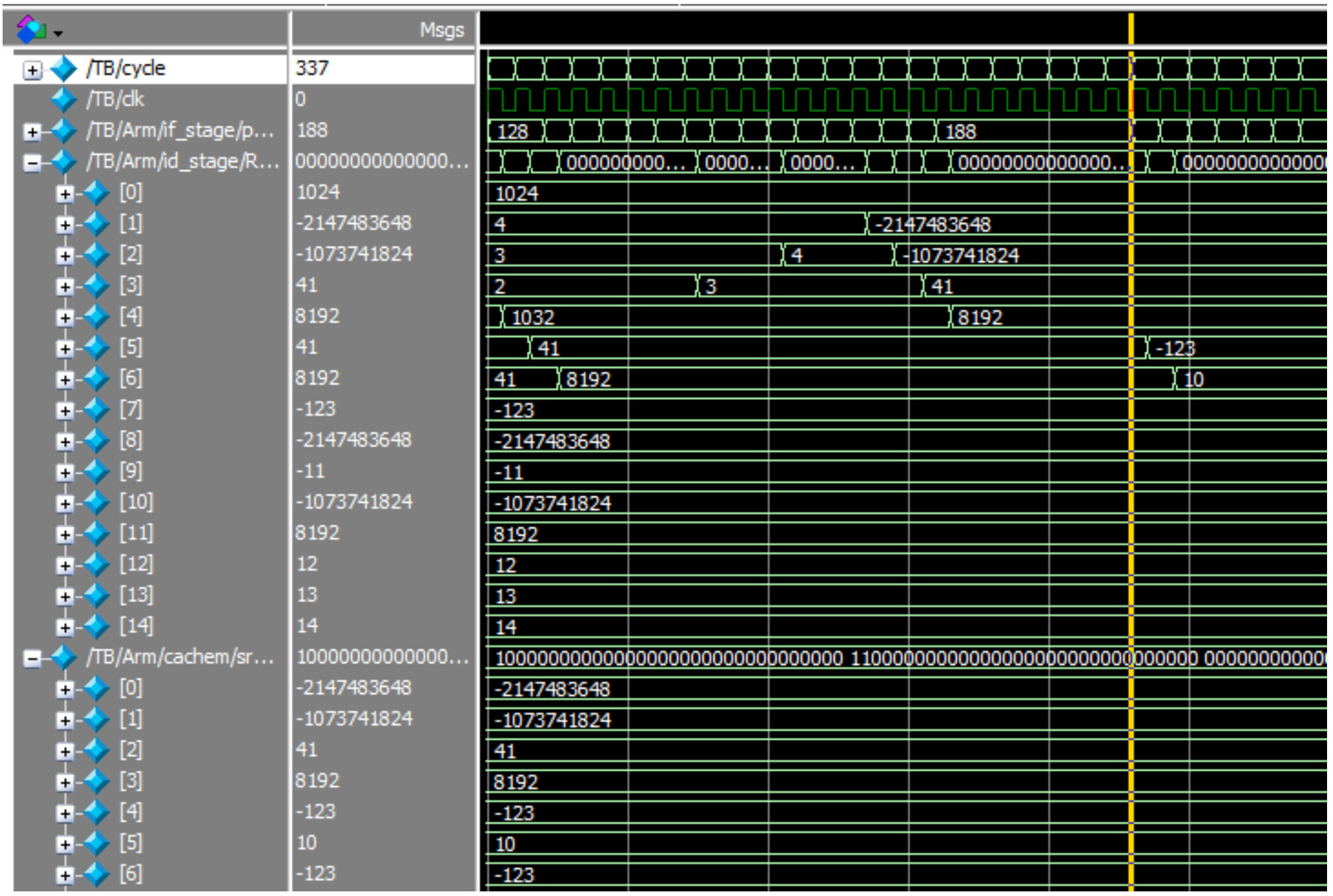
```
22      always@(posedge clk) begin
23              sram_write_en <= write_en;
24              sram_read_en <= read_en;
25      end
```

SRAM ، یک کلاک پس از کش کارش را شروع می کند(اگر نیاز به sram باشد). برای اینکه این اتفاق بیفتد، لازم است سیگنال های کنترلی و داده ای حیاتی که به ماژول sram controller می روند ، یک واحد رجیستر شوند. به همین علت در شکل بالا می بینید که سیگنال های کنترلی نوشتن و خواندن و هم چنین آدرس و داده به اندازه یک کلاک رجیستر شده و سپس وارد sram خواهندشد.

```
9       assign #30 SRAM_DQ0 = SRAM_WE_N ? memory[{SRAM_ADDR[16:1],1'b0}] : 32'bz;
10      assign #60 SRAM_DQ1 = SRAM_WE_N ? memory[{SRAM_ADDR[16:1],1'b1}] : 32'bz;
```

ما در ماژول sram نیز تغییراتی دادیم ، در هربار دسترسی به sram برای خواندن ، ۶۴ بیت داده نیاز هست، چرا؟ باتوجه به اصل دسترسی محلی ، ما دوخانه از حافظه را برمی گردانیم، همان طور که در شکل می بینید ، حافظه باتوجه به آدرس فرستاده شده ، wordoffset را کنار می گذارد و دوخانه مجاور هم را خروجی خواهد داد. البته زمان دسترسی این دوخانه مجموعا برابر دودسترسی متوالی یعنی ۶۰ نانوثانیه نیز لحاظ شده است که جای نگرانی ندارد زیرا کمتر از ۶ کلاک طول خواهدکشید.

در ادامه ما مدار را در مودلسیم سیمولیت می کنیم:



مقادیر رجیسترها و حافظه صحیح می باشند، همان طورکه میبینید ، اجرای برنامه حدود 340 سیکل طول کشیده است.

حالت پردازنده	معمولی	forwarding	SRAM	CACHE
سیکل اجرا برنامه	۲۷۹	۱۹۴	۴۲۵	۳۴۰

در حالت کش نسبت به حالت قبل ، حدود 20% تعداد سیکل های برنامه کمتر شده است.

در حالت کش نسبت به حالت فورواردینگ ، حدود 75% تعداد سیکل های برنامه بیشتر شده است.

در حالت کش نسبت به حالت معمولی ، حدود 22% تعداد سیکل های برنامه بیشتر شده است.

حالت	CPI	Execution Time	Dedicated logic registers	Total combinational functions	Total logic elements
معمولی	279/170 = 1.64	279	1525/6272 (24%)	2380/6272 (38%)	2856/6272 (46%)
forwarding	194/170= 1.14	194	750/6272 (12%)	2042/6272 (32%)	2509/6272 (40%)
SRAM	425/170= 2.5	425	817/6272 (13%)	2191/6272 (34%)	2690/6272 (42%)
CACHE	340/170= 2	340	2941/6272 (46%)	3743/6272 (59%)	6263/6272 (99%)

	Resource	Usage
1	Estimated Total logic elements	6,263
2		
3	Total combinational functions	3743
4	Logic element usag...mber of LUT inputs	
1	-- 4 input functions	3025
2	-- 3 input functions	548
3	-- <=2 input functions	170
5		
6	Logic elements by mode	
1	-- normal mode	3628
2	-- arithmetic mode	115
7		
8	Total registers	2941
1	-- Dedicated logic registers	2941
2	-- I/O registers	0

	Fmax	Restricted Fmax	Clock Name
1	34.33 MHz	34.33 MHz	clk

نرخ cpi در کش نسبت به قبلی بهبود یافته است، اما هزینه گزاف و زیادی روی دوش ما گذاشته است که به علت رجیسترهای حافظه نهان می باشند.

حداکثر فرکانس مدار نیز ۳۴ مگاهرتز هست بنابراین نسبت کارایی به هزینه برابر با 34000000/6263=5400 می باشد که نسبت به حالت های قبل بسیار کاهش یافته است!

خطاها و مشکلات :

- در کد اولیه ای که برای کش زده بودیم، هنگام نوشتن داده در کش ابدا منتظر نمی ماندیم تا ready_sram کارش تمام شود و از آنجا که مقدار دیلی خواندن از sram را کم زده بودیم ، اصلا متوجه این موضوع نشدیم. هنگامی دیلی واقعی خواندن از sram را گذاشتیم ، خروجی مدار اشتباه شد و از آنجا پی به اهمیت وجود ready_sram در شروط اغاز نوشتن در کش بردیم.
- هم چنین مدار برای miss و نوشتن در حافظه ۶ کلاک توقف داشت و این اشتباه بود. دلیلش این بود که سیگنال های ورودی واحد sram به اندازه یک کلاک رجیستر نشده بودند و در واقع sram همزمان با cache راه می افتاد که اشتباه بود. پس از رجیستر کردن سیگنال های sram ، مشکل برطرف شد.
- در هنگام سنتز خطای زیر ظاهر شد:

170012 Fitter requires 478 LABs to implement the design, but the device contains only 392 LABs

به این معنی که تعداد بلوک های منطقی مدار بیشتر از ظرفیت برد می باشد، ما برای اینکه مشکل را برطرف کنیم، تعداد خانه های sram که ۵۱۲ تا بود را کم کردیم.