

به نام خدا



معماری کامپیوتر

پروژه پایانی - بخش اول - طراحی پردازنده

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

استاد:

جناب آقای دکتر اسدی

نام، نام خانوادگی و شماره دانشجویی اعضای گروه:

سپهر پورقناد - ۹۷۱۰۱۳۵۹

سید محمدصادق کشاورزی - ۹۷۱۰۶۲۴۹

امیرمهدی نامجو - ۹۷۱۰۷۲۱۲

۱ مقدمه

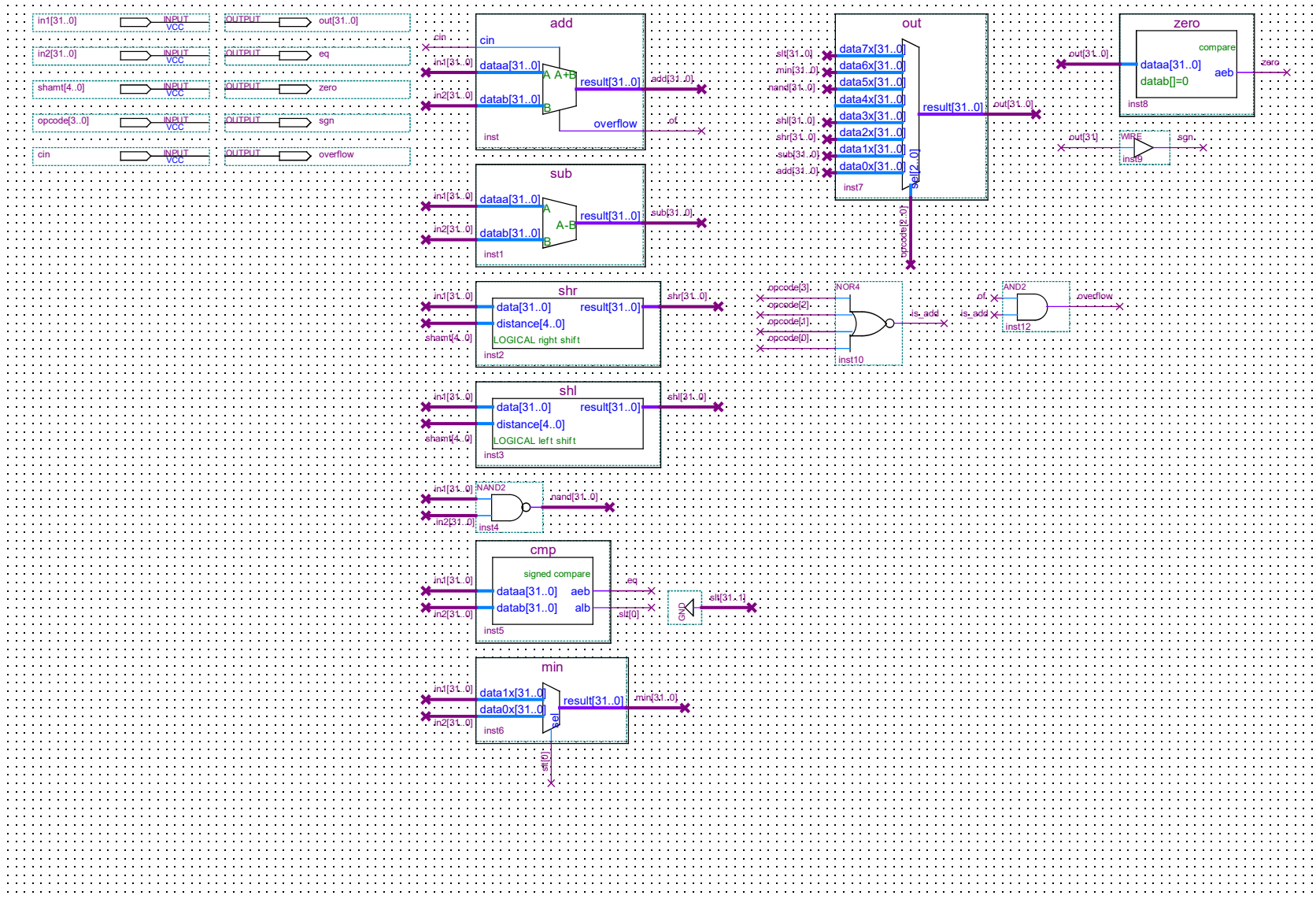
هدف اصلی ما در این پروژه، طراحی یک پردازنده ساده بر اساس دستورات داده شده در مستندات آن و تست کردن آن به شکلی که حالات مختلف آن پوشش داده شود بوده است. در طراحی پردازنده داده شده، ما از اصول کلی که در درس آموخته ایم و عموماً بر اساس پردازنده مبتنی بر MIPS بوده اند، استفاده کرده ایم. البته در این جا ساختار کلی ما از لحاظ کلی ساده تر از MIPS است. در ادامه گزارش، به توضیح نحوه طراحی بخش های مختلف این پردازنده می پردازیم.

۲ طراحی ALU

بخش ALU در اصل سیستم محاسبه‌گر اصلی ما است و از آنجایی که بیش‌تر دستورات داده شده در مستندات پروژه برای اجرا به آن وابستگی دارند، از اهمیت بالایی برخوردار است. برای طراحی ALU به این شکل عمل کرده‌ایم: پنج ورودی برای آن در نظر گرفته‌ایم که شامل داده ورودی ۱ و داده ورودی ۲ هر دو به صورت ۳۲-بیتی، مقدار شیفت به صورت ۵-بیتی، مقدار cin یعنی carry ورودی به صورت ۱-بیتی و همچنین Opcode به صورت ۴-بیتی بوده است.

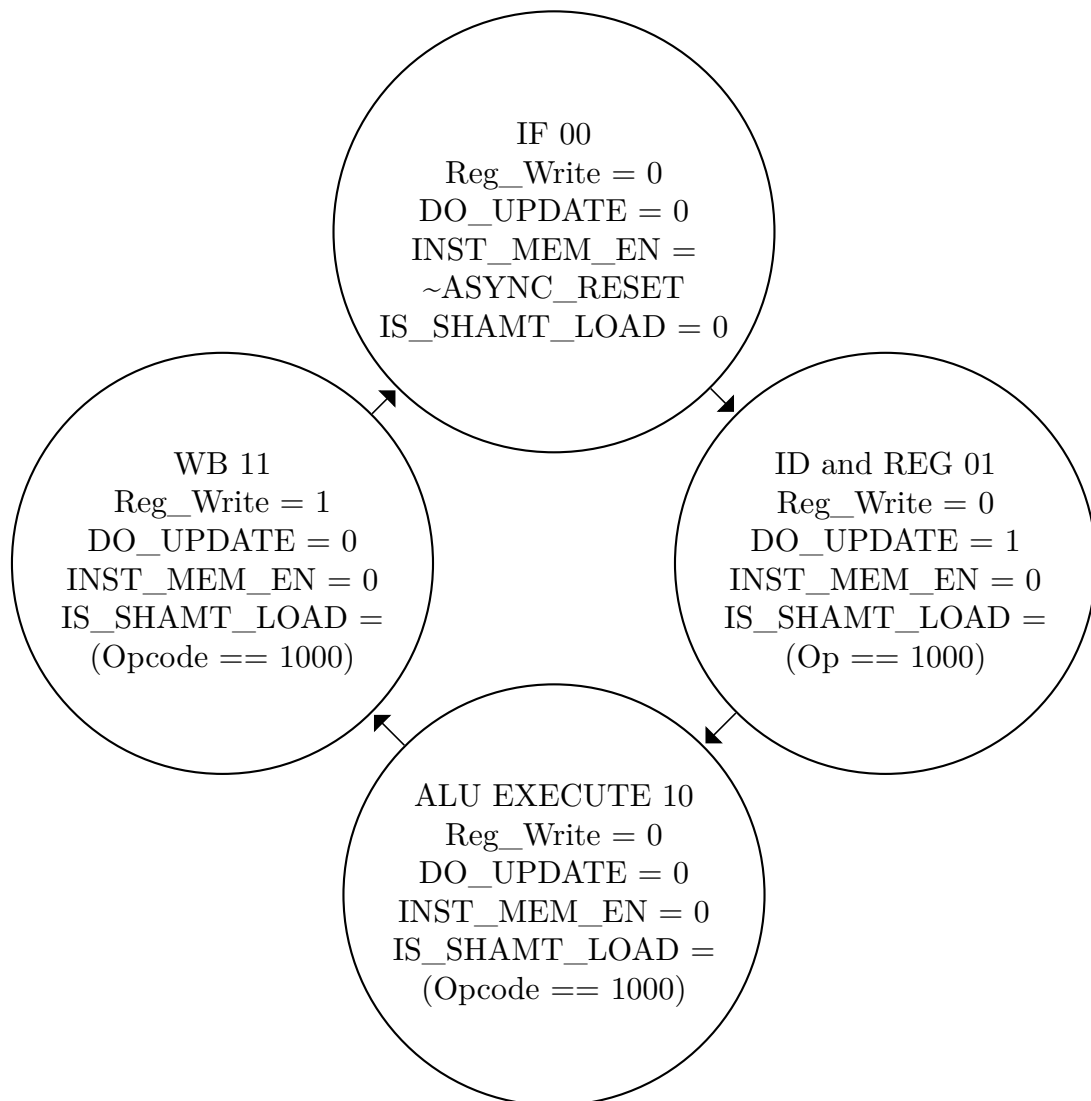
برای عملیات‌های مختلف آن، از ماژول‌هایی که در Quartus در اختیار ما قرار گرفته است، استفاده کرده‌ایم. برای جمع از ماژول جمع‌کننده استفاده کرده و همچنین از آنجایی که کوارتوس برای آن از سیستم محاسبه Overflow هم پشتیبانی می‌کند، آن را هم قرار داده‌ایم. برای تفریق از ماژول تفریق‌گر استفاده کرده‌ایم، البته روش‌های دیگری نظیر نقیض کردن ورودی دوم و استفاده از $cin=1$ به عنوان ورودی جمع‌کننده هم برای این کار وجود داشت ولی برای طراحی ساده‌تر و امکان دیباگ راحت‌تر، ترجیح دادیم که از ماژول جداگانه استفاده کنیم. برای شیفت به چپ و راست، ماژول‌های مربوط به آن‌ها را قرار داده‌ایم. این ماژول‌ها مقدار ShiftAmount را با نام distance و خود داده را هم با عنوان data به عنوان ورودی می‌گیرند. برای NAND یک گیت NAND با ۲ ورودی قرار داده‌ایم که به ورودی‌های آن، با توجه به این که کوارتوس توانایی تشخیص این که از مقادیر چندبیتی استفاده کرده‌ایم را هم دارد، مقادیر ۳۲-بیتی مربوطه را داده و مقدار ۳۲-بیتی خروجی گرفته‌ایم. برای عملیات set on less than از یک ماژول Comparator استفاده کرده‌ایم. این ماژول به عنوان ورودی دو داده ۳۲-بیتی را گرفته و در خروجی دو داده به ما می‌دهد؛ داده aeb در صورتی که 1 باشد، بیانگر برابر بودن دو مقدار و در غیر این صورت، بیانگر نامساوی بودن آنان است، داده alb هم در صورت یک بودن، نشانگر کوچک‌تر بودن a از b و در غیر این صورت نشان‌دهنده بزرگ‌تر مساوی بودن b از a است. از سیگنال alb برای حالت slt استفاده کرده‌ایم و از آنجایی که این خروجی ۱-بیتی است ولی به عنوان خروجی ALU ما ۳۲ بیت می‌خواهیم، سایر بیت‌ها را از طریق اتصال به GND صفر کردیم. در نهایت هم از یک تسهیم‌کننده به همراه سیگنال alb برای انتخاب ورودی کوچک‌تر از بین دو ورودی استفاده کرده‌ایم.

برای انتخاب این که خروجی کدام ماژول باید به عنوان خروجی نهایی داده‌شود، از یک مالتی پلکسر ۳۲-بیتی ۸ به ۱ استفاده کرده‌ایم که البته حالت 100 آن استفاده خاصی ندارد. برای خروجی sgn که نشانگر علامت خروجی است، از بیت 31 مقدار خروجی این مالتی پلکسر استفاده کرده‌ایم. همچنین برای خروجی eq هم از خروجی aeb ماژول Comparator استفاده کرده‌ایم. برای Overflow هم از آنجایی که سؤال گفته فقط برای جمع باید در نظر گرفته شود، گیت‌هایی قرار داده‌ایم که بر اساس OpCode تنها در صورتی که در حالت جمع باشیم، مقدار Overflow ماژول جمع‌کننده به خروجی برود و در غیر این صورت، 0 به خروجی برود. یک ماژول به اسم zero هم قرار داده‌ایم که در اصل در پیاده‌سازی داخلی همان comparator است با این تفاوت که یکی از داده‌های آن به شکل درونی صفر شده است و مشخص می‌کند که آیا خروجی نهایی برابر صفر است یا نه. شکل کلی ماژول ALU در صفحه بعد قرار دارد.



۳ طراحی واحد کنترل

برای طراحی این پردازنده، ما از ۴ کلاک استفاده کرده‌ایم که در شکل زیر State Machine آن قرار گرفته‌است. ضمناً این ماشین حالت را به صورت میلی (Mealy) طراحی کرده‌ایم. یعنی ورودی‌های هر حالت، روی نتیجه آن تأثیر دارند. دلیل این موضوع، متفاوت بودن اجرای دستور لود Shamt با سایر دستورات است. در اصل می‌شد برای این دستور چند استیت متفاوت در نظر گرفت ولی برای ساده‌تر شدن طراحی و کارکرد پردازنده بدون نیاز به پیچیدگی‌های مختلف، چهار استیت هر دو را یکسان طراحی کردیم و در عوض سیستم به صورت میلی است که ورودی OpCode روی خروجی‌های هر استیت تأثیر دارد.

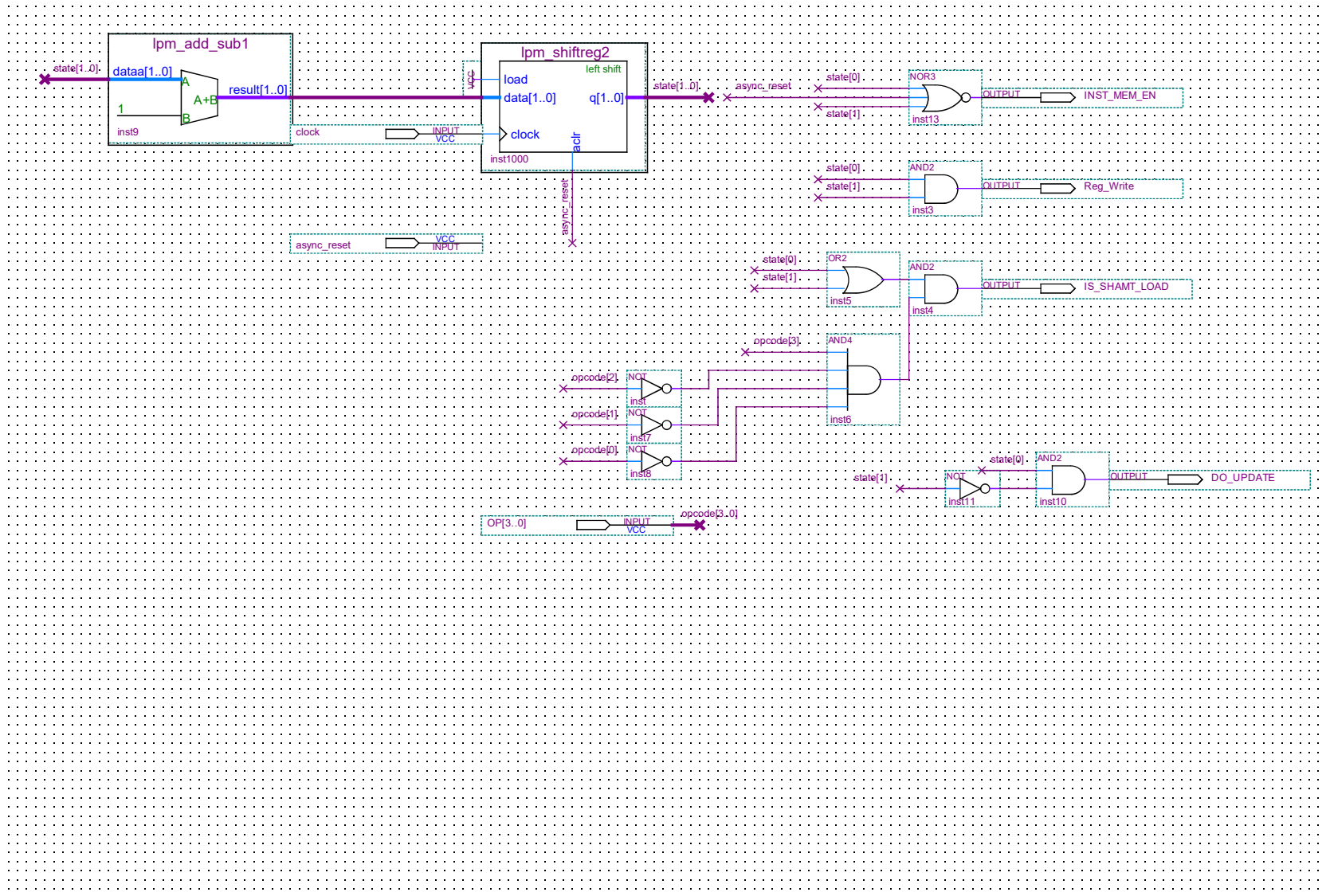


سیگنال Reg_Write همان طور که از نام آن مشخص است، مشخص کننده این است که نوشتن روی رجیستر فایل در آن کلاک باید انجام بشود یا خیر. این سیگنال تنها در حالت WriteBack انجام می شود.

سیگنال DO_UPDATE مشخص کننده این است که PC در آن کلاک باید آپدیت بشود یا نه. ما برای اجرای درست دستوراتمان، تنها در حالت 01 این سیگنال را 1 می کنیم. چگونگی تأثیر این سیگنال را در ادامه و بخش مربوطه، توضیح خواهیم داد. دقت کنید که با توجه به وجود یک فلیپ فلاپ در ورودی ROM برای نگه داری آدرس و یک فلیپ فلاپ در خروجی ROM برای نگه داری داده خروجی، پس از تغییر مقدار PC، در لبه بالارونده اول مقدار جدید PC توسط ROM به عنوان آدرس خوانده می شود، و سپس در لبه بالارونده دوم داده مربوط به آدرس جدید در خروجی قرار می گیرد و به این ترتیب، دستور بعدی، در شروع حالت 00 اجرای دستور بعدی آماده واکشی است.

سیگنال دیگر INST_MEM_EN است که در اصل این مشخص می کند که Instruction Register مقدار جدید را از روی مموری دریافت بکند یا نه. در هر سه استیت 01, 10, 11 مقدار آن صفر است و در حالت IF هم اگر سیستم در وضعیت ریست شدن نباشد، مقدار آن 1 می شود و اگر کل سیستم در وضعیت ریست شدن باشد، مقدار آن صفر است که کلاً Instruction Register تغییر نکند.

در نهایت سیگنال IS_SHAMT_LOAD مشخص کننده این است که Write_Data رجیستر فایل باید از خروجی ALU دریافت شود یا بیت 5 تا 14 دستور. در اصل این سیگنال تنها در استیت WB اثرگذار است اما در سه استیت دیگر هم از آن جایی که می توان مقدار آن را مشخص کرد، آن را مقداردهی کرده ایم. این سیگنال تنها زمانی 1 می شود که Opcode برابر 1000 یعنی مربوط به دستور Shamt Load باشد. شکل واحد کنترل در صفحه بعد قرار دارد.



۴ مقداردهی حافظه دستورات

برای حافظه دستورات ما از بخش Mega-Plugin Wizard ارائه شده توسط Quartus و مقداردهی ROM بر اساس فایل mif مشابه راهنمای قرار گرفته در مستند پروژه استفاده کرده‌ایم. یکی از نکات قابل توجه این جاست که در CycloneII که تابه‌حال طراحی‌های کوارتوس را با آن انجام دادیم، نمی‌توان رام چندان بزرگی استفاده کرد و مجبور شدیم از رمی با اندازه ۳۲۷۶۸ کلمه ۲۰ بیتی استفاده کنیم و برای تعداد کلمه بزرگ‌تر، در هنگام کامپایل به خطای در اختیار نبودن منبع برای سنتز چنین چیزی روی CycloneII بر می‌خوردیم.

محتویات فایل mif قرار گرفته بدین صورت است. در جلوی دستورات به صورت کامنت، کاری که دستور انجام می‌دهد هم مشخص شده است:

```
WIDTH=20;
DEPTH=32768;
ADDRESS_RADIX = UNS;          -- The radix for address values is unsigned
DATA_RADIX = BIN;             -- The radix for data values is Binary
CONTENT                        -- start of (address : data pairs)
BEGIN
  0 : 100000000000000000001; -- load shamt r1 <-1
  1 : 100000000000000000010; -- load shamt r2 <-1
  2 : 100000000000000000011; -- load shamt r3 <-10
  3 : 1000000000000000000100; -- load shamt r4 <-100
  4 : 001100000111110000001; -- r1 = r1 << 30
  5 : 000000000000000000001; -- r1 = r0 + r1 (0)
  6 : 000000000000000000010; -- r2 = r0 + r2 (0)
  7 : 000000000000000000011; -- r3 = r0 + r3 (0)
  8 : 0000000000000000000100; -- r4 = r0 + r4 (0)
  9 : 00000000100001000101; -- r5 = r2 + r2
 10 : 00000000010000100110; -- r6 = r1 + r1 -> must overflow
 11 : 00100001000000100100; -- r4 = r4 >> 1
 12 : 01010001000001100111; -- r7 = r4 ~& r3
 13 : 01110000000010001000; -- r8 = set on less than (r0 , r4)
 14 : 01100000100010001001; -- r9 = min(r2 , r4)
 15 : 1000000000000000101010; -- load shamt r10 <-1
 16 : 00010000000101001011; -- r11 = r0 - r10 (answer must be -1)
 17 : 01110010110001101100; -- r12 = set on less than (r11 , r3) -> answer = 1
 18 : 01100010110000001101; -- r13 = min(r11 , r0) -> answer r11 that is -1
 19 : 000000000000000000001; -- r1 = r0 + r1 (0)
 20 : 000000000000000000010; -- r2 = r0 + r2 (0)
 21 : 000000000000000000011; -- r3 = r0 + r3 (0)
 22 : 000000000000000000010; -- r4 = r0 + r4 (0)
```



```

23 : 00000000000010100101; -- r5 = r0 + r5 (0)
24 : 00000000000011000110; -- r6 = r0 + r6 (0)
25 : 00000000000011100111; -- r7 = r0 + r7 (0)
26 : 00000000000010000100; -- r8 = r0 + r8 (0)
27 : 00000000000010010101; -- r9 = r0 + r9 (0)
28 : 00001000000010100101; -- r10 = r0 + r10 + cin(1)
29 : 00000000000010110101; -- r11 = r0 + r11
30 : 00000000000010100101; -- r10 = r0 + r10
31 : 00000000000011000110; -- r12 = r0 + r12
32 : 00000000000011010101; -- r13 = r0 + r13
[33..32767] : 00000000000000000000;
END;

```

نکته مهم در مورد دستورات آخر که اکثراً جمع با صفر هستند، این است که به کمک این دستورات و Output های قرار داده شده، مقادیر هر کدام از رجیسترها را مشاهده کنیم که ببینیم آیا در نهایت مقادیر قرار گرفته در آنان درست است یا نه. خروجی های نهایی رجیسترها، باید به شکل زیر باشد (یعنی در خط های آخر، باید این اعداد را مشاهده کنیم):

```

R1 = 1073741824
R2 = 1
R3 = 2
R4 = 1
R5 = 2
R6 = -2147483648 -- this is caused by overflow on address 10
R7 = -3
R8 = 1
R9 = 1
R10 = on line 26 it is 1 and then on address 28 it is 2
R11 = -1
R12 = 1 (is -1 less than 2)
R13 = -1 (minimum of -1 and 0)

```

ضمناً باید توجه کنیم که این ROM ها به صورت Word-Addressable هستند و از طرفی PC ما چهار واحد چهار واحد جلو می رود و همچنین با توجه به فضای داده شده، برای یافتن آدرس درست تنها به ۱۵

بیت نیاز دارند، ما از بیت 2 تا PC 16 را به آن به عنوان ورودی آدرس داده‌ایم. دو بیت اول در اصل همیشه 0 هستند و با توجه به Word-Addressable بودن رام نیازی به آن نیست.

۵ PC و RF

برای این دو بخش، همان طور که در صورت پروژه خواسته شده بود، از ماژول‌های طراحی شده تمرین ۶ و تمرین ۷ استفاده شده‌است. ماژول تمرین ۷ عیناً قرار گرفته و ماژول تمرین ۶ هم با تغییر کوچکی در مورد Reset مورد استفاده قرار گرفته است. از آن جایی که ای Reset که ما در این جا قرار دادیم، در همه بخش‌ها به صورت Active High بود، ریست تمرین ۶ که مربوط به PC بود را هم به همین صورت تغییر داده‌ایم. دو نکته مهم وجود دارد. برای این که در این جا، از مواردی نظیر Jump در PC استفاده نمی‌کنیم و همواره یا PC باید ثابت باشد، یا ۴ واحد به جلو برود، از این روش استفاده کردیم که PC را همواره در حالت Branch قرار داده‌ایم. در حالت‌هایی که می‌خواهیم PC ثابت بماند، به عنوان Offset به آن مقدار 1- را می‌دهیم که با دو واحد شیفت به چپ درون واحد محاسبه‌گر PC تبدیل به 4- می‌شود از آن جایی که در حالت Branch مقدار آن به صورت $PC + 4 + Branch\ Offset$ محاسبه می‌شود، در این حالت، PC ثابت می‌ماند. در زمان‌هایی هم که می‌خواهیم PC به اندازه ۴ واحد جلو برود، مقدار 0 را به عنوان Offset می‌دهیم. این که چه مقداری به PC داده شود، با سیگنال DO_UPDATE کنترل می‌شود. نکته مهم دیگر هم این است که مقدار ورودی Write Data رجیستر فایل از طریق یک مالتی‌پلکسر با سیگنال کنترلی IS SHAMT LOAD کنترل می‌شود که در صورتی که 0 باشد، خروجی ALU مد نظر قرار گرفته و در صورتی که 1 باشد، مقدار Extend Zero شده بیت‌های ۵ تا ۱۵ دستور معیار قرار می‌گیرد.

۶ کنار هم قرار دادن قطعات

برای کنار هم قرار دادن قطعات استفاده شده، تقریباً از روشی مشابه آن چه که در اسلایدهای درس مشاهده کرده‌ایم استفاده شده‌است. به این نکته هم توجه شده که در خروجی RF دو عدد فلیپ‌فلاپ برای ذخیره خروجی آن‌ها و همچنین در خروجی ALU هم یک فلیپ‌فلاپ دیگر قرار داده‌ایم. در اصل این‌ها معادل همان رجیسترهای اسلایدها هستند که عملاً چون رجیسترهای کوارتوس در این مورد خاص، به جز استفاده از منابع بیش‌تر به خاطر قابلیت شیفت دادن، هیچ مزیتی به فلیپ‌فلاپ ندارند، مستقیماً از فلیپ‌فلاپ استفاده کرده‌ایم.

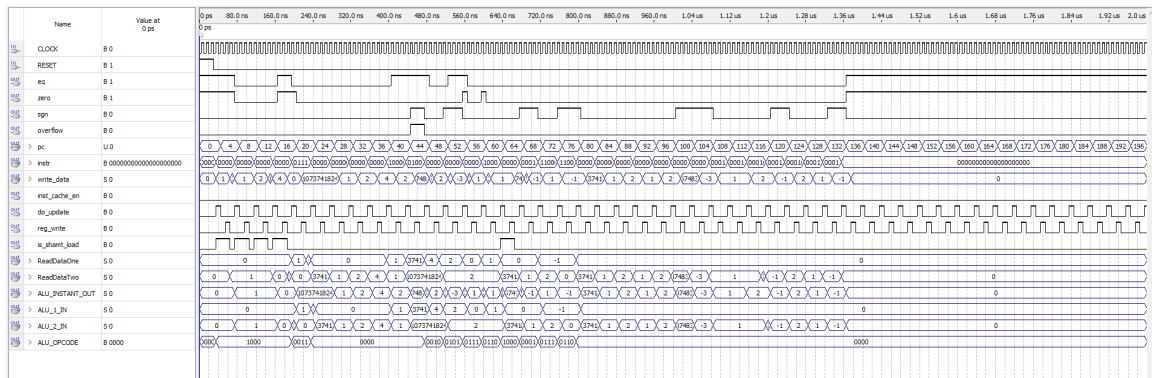
نکته قابل توجه دیگر هم این است که برای شلوغ نشدن یکسری از اتصالات، به جای وصل کردن سیم‌ها به صورت خط روی شکل، از این قابلیت کوارتوس استفاده کرده‌ایم که اگر دو سیم نام مشابهی داشته باشند، اتصال بین آن‌ها را خود کوارتوس تشخیص می‌دهد. بدین ترتیب، به جز برای بخش‌هایی که برای وضوح بیش‌تر شکل لازم بوده، در سایر بخش‌ها از کشیدن سیم‌ها به صورت خط اجتناب کرده‌ایم و از نام‌گذاری‌ها استفاده کرده‌ایم که جلوی شلوغ شدن شکل را می‌گیرد. شکل آن در صفحه بعد قرار دارد.



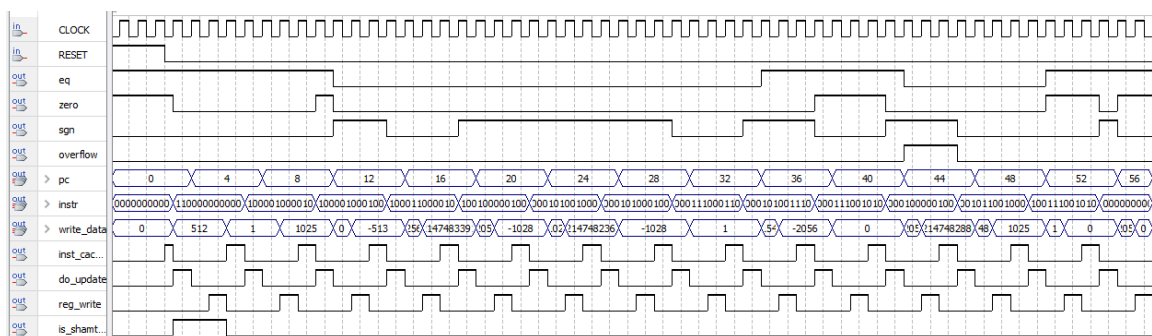
۷ تست پردازنده

برای تست پردازنده هم، همان دستوراتی که در بخش مقداردهی حافظه دستورات ذکر کردیم را استفاده کردیم. خروجی تمامی سیگنال‌های موردنظر با توجه به موارد گفته‌شده، درست بوده‌است. توجه کنید که از بخشی از ویو فرم به بعد، دیگر دستور قابل توجهی نداشتیم و صرفاً جمع کردن 0 با 0 بوده است که برای این که حافظه خالی نباشد، قرار داده‌ایم. نکته مهمی که وجود دارد، این است که حتماً باید سه کلاک اول RESET برابر 1 باشد. دلیل این موضوع این است که حافظه دستورات ساخته شده توسط کوارتوس، دو فلیپ‌فلاپ در ورودی و خروجی خود دارد و سه کلاک طول می‌کشد تا عملکرد آن‌ها به حالت باثبات و پایدار برسد (در کلاک اول PC صفر می‌شود، در کلاک دوم حافظه آدرس 0 را به عنوان ورودی می‌خواند، و در کلاک سوم محتوای خانه 0 م حافظه را در خروجی قرار می‌دهد). پس از آن، عملکرد سیستم درست است و برای همین باید سه کلاک اول سیستم در وضعیت RESET باشد. در زیر شکل آن را قرار داده‌ایم. توجه کنید که شکل قابل زوم کردن است. ضمن این که یک فایل ویدیویی هم در کنار گزارش کار قرار دارد که روی تک‌تک کلاک‌ها حرکت می‌کنیم و امکان مشاهده تک‌تک سیگنال‌ها میسر است.

ضمناً لازم به ذکر است که فایل اصلی پروژه CPU.bdf و فایل اصلی تست و ویو فرم CPU2.vwf نام دارد و بقیه فایل‌ها مربوط به سایر قسمت‌ها و یا ویو فرم‌های تست مازول‌های جداگانه هستند.



برای بررسی بیشتر، Waveform زیر نیز موجود است که محتوای حافظه آن در پایین آمده‌است:



```
WIDTH=20;  
DEPTH=32768;
```

```
ADDRESS_RADIX=UNS;  
DATA_RADIX=BIN;
```

```
CONTENT BEGIN
```

```
0      : 1000110000000000000001; -- R[1] <- 512  
1      : 00001000010000100010; -- R[2] <- 1025  
2      : 00011000010001000011; -- R[3] <- -513  
3      : 00101000110000100100; -- R[4] <- 2147483391  
4      : 00111001000001000101; -- R[5] <- -1028  
5      : 01010001010010000110; -- R[6] <- -2147482365  
6      : 01100001010001000111; -- R[7] <- -1028  
7      : 01110001110001101000; -- R[8] <- 1  
8      : 00000001010011101001; -- R[9] <- -2058 {eq = 1}  
9      : 00010001110010101010; -- R[10] <- 0 {eq = 1, zero = 1}  
10     : 00000001000001001011; -- R[11] <- -2147482880 {overflow = 1}  
11     : 00010010110010001100; -- R[12] <- 1025 {overflow = 0}  
12     : 01111001110010101101; -- R[13] <- 0  
[13..32767] : 00000000000000000000;  
END;
```