خدا هست

گزارش پروژه (تمرین سوم) درس طراحی سیستم های دیجیتال (DSD) – دکتر فصحتی

طراحی پردازنده 16 بیتی (ALU, Controller, Memory, Register File)

امیرمحمد شربتی 402106112

1404/5/5

طراحی بنده در این تمرین شامل ۷ فایل اصلی وریلاگ و در نتیجه 7 فایل برای تست آنها است. به تبع این‌ها در پوشه src فایل‌های دیگری از جمله فایل های vcd و vvp هم وجود دارد (چون بنده برای simulationکدها از Icarus استفاده می‌کنم) (هنگام آپلود فایل ها در کوئرا به دلیل زیاد بودن حجم فایل، اخطار گرفتم. بنابراین فایل های vcd که مربوط به waveform هستند و حجم خیلی بیشتری نسبت به سایر فایل ها دارند را حذف کردم).

- فایل های اصلی شامل ماژول های اصلی برنامه:

Csa_16 ، mul_16 ، div_16 ، alu_16 ، memory ، register file و controller

برخی از این فایل ها شامل چند ماژول هستند. مثلا چون پیاده سازی جمع کننده در سطح gate level است، بنابراین فایل csa_16 شامل ماژول های مختلف از جمله mux و FA و csa است. همچنین برای هر فایل، بنده یک ماژول تست هم نوشتم که با همین نام ها به اضافه tb_ هستند. مثلا memory_tb برای تست حافظه. حال به توضیحی در مورد هر فایل می‌پردازم:

- ***csa_16***: این ماژول پیاده سازی سطح گیت جمع کننده این پردازنده با الگوریتم خواسته شده در داک است. همین ماژول برای تفریق هم استفاده می‌شود. پایین ترین لایه پیاده سازی یک Full Adder با گیت های پایه است. سپس به کمک همین ماژول، یک ماژول برای جمع دو عدد 4 بیتی پیاده سازی کردم. سپس طراحی مالتی پلکسر 2 به 1 و سپس 8 به 4. یعنی در واقع همان 2 به 1ای که هر ورودی 4 بیتی است. پس از این یک جمع کننده 4 carry select بیتی طراحی کردم. در واقع این ماژول به تنهایی کار خاصی انجام نمی‌دهد و بخشی از ماژول بزرگتر csa 16 بیتی است. در این نوع جمع کننده بلوک اول یک جمع کننده 4 ripple carry بیتی ست. سه بلوک دیگر csa 4 بیتی هستند. مشابه چیزی که در داک آمده این جمع کننده نوعی carry select adder است که بلوک های آن تعداد بیت برابر دارند. پیاده سازی واضحا یک مدار ترکیبی ساده بدون کلاک است. انجام این عملیات تنها یک چرخه طول می‌کشد.

در تست این مدار هم بنده سعی کردم حالات بسیاری را پوشش دهم و این کار را با for های تو در تو انجام دادم. یک حلقه که 2^7 تا تغییر میکند که صفر را هم پوشش میدهد. و دیگری که شامل اعداد کمی غیر رند تر است. اجرای این تست حدود 5 ثانیه در لپ تاپ بنده زمان برد.

❖ *mul_16*: این فایل شامل دو ماژول است. برای پیاده سازی ضرب کننده باید دو الگوریتم را پیاده سازی کنیم. برای ضرب دو عدد 8 بیتی از الگوریتم ساده shift and add استفاده شده است. این ماژول پیاده سازی یک مدار ترتیبی است. در داخل بلوک always سه بلوک if else وجود دارد. یکی اینکه برای reset مدار است. دومی برای شروع ضرب و مقدار دهی های اولیه است. بلوک سوم بخش اصلی این الگوریتم است. در صورت یک بودن بیت سمت راست B، مقدار A به حاصل اضافه می‌شود. سپس B به راست شیفت می‌خورد و A به چپ. این کار 8 بار تکرار میشود (در واقع به تعداد بیت های B) تا نتیجه نهایی حاصل شود.

در ماژول karatsuba_mul_16 این الگوریتم برای ضرب 16 بیتی پیاده سازی شده است. در واقع الگوریتم اصلی Karatsuba کاملا به صورت بازگشتی است ولی اینجا فقط یک مرحله را بدین نحو انجام می‌دهیم. در پیاده سازی بنده از سیگنال active هم استفاده کرده‌ام. توجه شود ممکن است این سه بخش z0, z2 و z3 ((z2 = (z3 - (z0 + z2)) به صورت همزمان آماده نشوند، پس بنده در ماژول قبلی، done را صفر نکردم. دلیل استفاده از active هم همین است. اگر این سیگنال نباشد، هیچ وقت این محاسبه به پایان نمی‌رسد. در نهایت در این ماژول پس از یک کلاک done صفر میشود (چون active در بدست آوردن حاصل نهایی صفر می‌شود.)

توجه شود که این یک مدار multi cycle است. پس از سیگنال های clk ، start، reset و done استفاده شده است.

در ماژول تست هم بنده مشابه تست جمع کننده، از حلقه استفاده کردم و حالات زیادی را تست کردم. این تست هم حدود 4 ثانیه طول می‌کشد تا نتیجه اش حاصل شود.

❖ *div_16:* این ماژول هم کار تقسیم را انجام میدهد. ابتدا به کمک علامت مقسوم و مقسوم علیه، علامت خارج قسمت و باقی مانده مشخص میشود. سپس قدر مطلق دو عدد بر هم تقسیم می‌شود. توجه شود که در الگوریتم restoring division چاره ای جز این نداریم. علی رغم ضرب که با هر علامتی درست کار میکند، در این الگوریتم تقسیم، ابتدا باید علامات مشخص و سپس تقسیم دو عدد مثبت صورت گیرد. حلقه اصلی 16 بار اجرا میشود و مطابق الگوریتم صورت سوال، باقی مانده و خارج قسمت بدست می‌آیند. توجه شود اگر مقسوم علیه ورودی صفر باشد، بنده صفر را به عنوان خارج قسمت و باقی مانده خروجی می‌دهم.

ماژول تست هم مشابه ماژول های تست قبلی سعی کردم به کمک حلقه حالات بسیاری را تست کنم. اگر مقسوم علیه هم صفر شود، همانطور که پیشتر اشاره شد، فرض کردم خارج قسمت و باقی مانده صفر است.

❖ ***alu_16 :*** این ماژول ALU است که کارهای controller در دستورات R_Type را راحت می‌کند. در ابتدا از سه ماژول قبلی instance گرفته شده است. FSMای که بنده متصور شدم شامل 4 حالت است. حالت اولیه (یا بیکار IDLE) و سه حالت دیگر برای تکمیل محاسبه سه ماژول اولیه (یعنی ضرب، تقسیم و جمع‌وتفریق). هر سه منطق مشابهی دارند. حاصل ماژول مربوطه به result ریخته می‌شود، done یک می‌شود و state به حالت اولیه برمی‌گردد. تفاوت در این است که ماژول مربوط به جمع و تفریق، یک مدار ترکیبی است که در یک کلاک نتیجه اش حاصل می‌شود، اما ضرب و تقسیم multi cycle هستند و باید تا اتمام محاسبات (mul_done == 1 یا div_done == 1) منتظر باشیم.

بنده این ماژول را هم تست کردم. در ابتدا پاسخ مورد انتظار برای 4 عمل محاسبه می‌شود. سپس در حلقه هر 4 عملیات تست می‌شوند. ابتدا قرار دادن صحیح opcode و فعال کردن سیگنال start برای یک چرخه، سپس انتظار برای پایان عملیات و در نهایت بررسی پاسخ بدست آمده. (الان که فکر می‌کنم می‌بینم که شاید بهتر بود یک task برای این تست می‌نوشتم تا تست 4 عملیات تکرار کد نباشه... :)) در نهایت طی 4 ثانیه خدا رو شکر تایید درستی ماژول ALU چاپ می‌شود.

❖ ***register_file :*** این ماژول شامل پورت های read/write _enable برای تعیین نوع عملیات است. در ورودی شماره ثباتی که در ان نوشته می‌شود یا دو ثباتی که از آن ها خوانده می‌شود و همچنین دیتایی که قرار است نوشته شود مشخص می‌شود. در خروجی هم دو دیتایی که از ثبات های مشخص شده خوانده شده، حاضر می‌شود. این register file مشابه مطلوبات داک شامل 4 ثبات 16 بیتی است. نوشتن در کلاک بالارونده و خواندن در کلاک پایین رونده انجام میشود. همیچنین بنده یک سیگنال reset هم قرار دادم که ممکن است در پردازنده های واقعی چندان کاربردی نباشد. در تست این ماژول هم بنده چند عدد در ثبات ها نوشته و سپس میخوانم.

❖ *memory*: ساده ترین ماژول این برنامه، همین ماژول حافظه است. در ورودی این ماژول مشابه register file سیگنال های read/write _enable قرار داردو همچنین آدرسی که میخواهیم به آن دسترسی داشته باشیم و دیتایی که میخواهیم بنویسیم هم در ورودی این ماژول هستند. تنها پورت خروجی هم دیتای خوانده شده از حافظه است. بنده اولویت را به نوشتن داده‌ام. بدین معنی که اگر هر دو enable فعال شوند، نوشتن صورت گیرد. توجه شود مطابق داک، این حافظه word addressable است. آدرس دهی به خانه های دو بایتی است و تعداد خانه های حافظه هم 2^{16} است که با 16 بیت مشخص می‌شود.

برای تست این برنامه، بنده دو task برای خواندن و نوشتن در حافظه نوشته‌ام. ابتدا در چند خانه مقادیری را می‌نویسم و سپس محتویات چند خانه از جمله آدرسی که در آن چیزی نوشته نشده است را هم می‌خوانم (که مطابق انتظار خروجی این آدرس x است.)

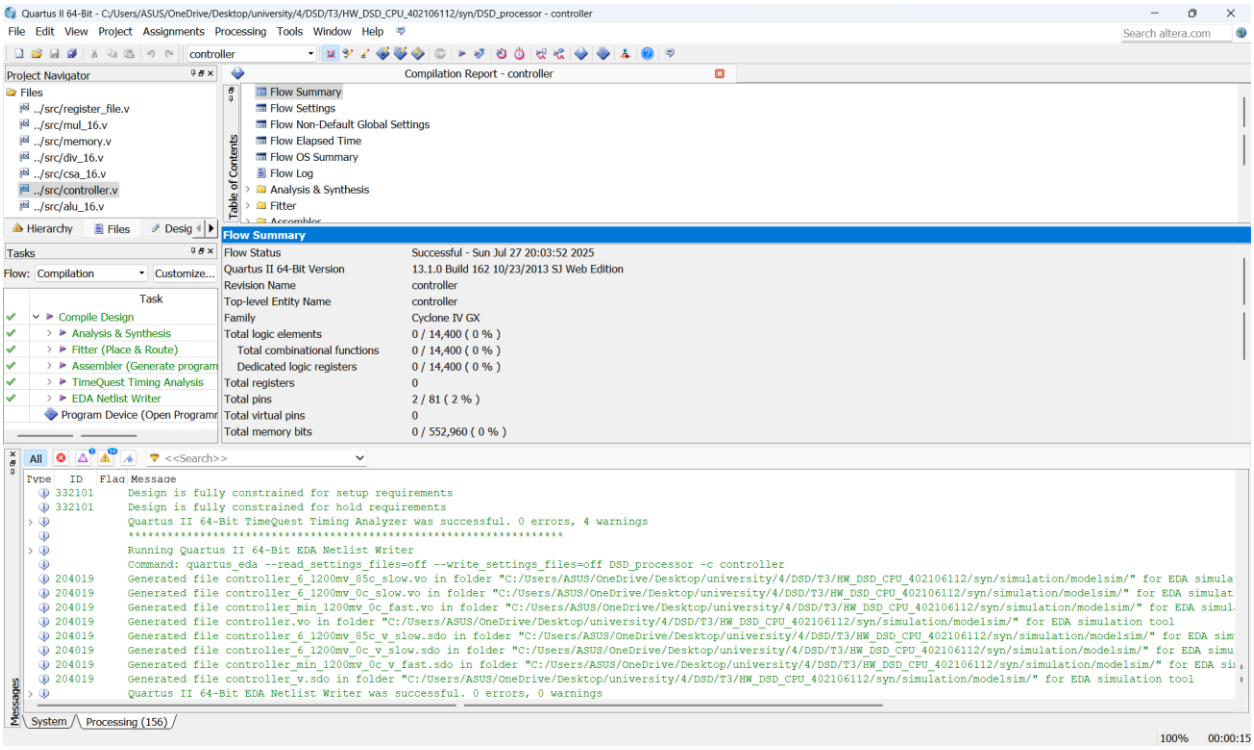
❖ *controller :* سخت ترین و مهم‌ترین ماژول این پروژه controller است. در واقع این ماژول top level entity این برنامه است که از تمام ماژول های دیگر در این ماژول استفاده شده است. در ابتدا PC، instruction و سایر متغیرهای لازم را تعریف کردم. سپس از ماژول های memory، alu و register_file ، instance گرفتم. برای نمونه گیری از هر یک متغیر های لازم را تعریف کردم. پس از آن حالات مختلف FSM این ماژول را تعریف کردم. 11 حالت شد. توجه شود که تعداد حالات را می‌شد کمتر کرد، ولی به خاطر خوانایی بالای کد بنده از حالات بیشتری استفاده کردم. اگر هدف کارایی بالاتر پردازنده باشد، به طوری که تعداد چرخه میانگین برای هر دستور کمتر شود، می‌توان برخی از این state ها را با تغییرات کمی در کد حذف کرد. مثلا چون خروجی memory در دو بخش دستور و دیتا کاربرد دارد و در واقع یک حافظه یکپارچه با دو قابلیت داریم، پس تعریف instruction صرفا خوانایی کد را بالاتر می‌برد. در صورت عدم استفاده از متغیر instruction می‌توان به جای آن از mem_read_data استفاده کرد و مشکلی هم در برنامه پیش نمی‌آید. در این صورت میتوان به راحتی FETCH_WAIT_2 را حذف کرد و در ماشین حالت مستقیما از FETCH_WAIT_1 به حالت DECODE رفت.

ابتدا دستور را از درون حافظه fetch میکنیم. سپس decode دستور را داریم که بخش های مختلف این رشته باینری 16 بیتی مشخص شود. بخش های opcode , rd , rs1 در هر دو نوع دستور یکسان هستند. در دستورات M_Type بقیه بیت های برای آدرس هستند ولی در دستور نوع R، دو بیت دیگر برای ثبات دیگر است. در state دیگر دو ثباتی که لازم است را می‌خوانیم که این هم در دو نوع دستور متفاوت است. سپس وارد EXECUTION می‌شویم. می شد به راحتی برای هر نوع دستور یک حالت جدا گانه داشت ولی ترجیح بنده بر همین شکل بود. در این state، برای دستورات نوع R و load , store سیگنال های لازم مقداردهی می‌شوند. کار دستور Store در اینجا تمام می‌شود. دستورات محاسباتی باید تا اتمام محاسبه و یک شدن alu_done صبر کنیم پس به حالت ALU_WAIT نیاز داریم. برای Load هم مشابه fetch به دو حالت نیاز داریم. اما اگر می‌خواستیم که کارایی را بهتر کنیم و از سیگنال میانی result (مشابه instruction) استفاده نکنیم، میشد LOAD_WAIT_2 را حذف کرد. در این صورت باید از حالت LOAD_WAIT_1 به state جدیدی می‌رفتیم که در آنجا نه محتویات result بلکه محتویات mem_read_data مشابه حالت WRITE_BACK در ثبات مقصد نوشته شود. ولی خب بنده سعی کردم کارهای مشابه (یعنی مثلا نوشتن در ثبات مقصد) را در یک state انجام دهم. به نظرم اینطور بهتر است. ولی خب یک چرخه به اجرای load در این بخش اضافه می‌شود. (مشابه fetch دستورات). در نهایت هم حالت نوشتن در ثبات مقصد و DONE را داریم که به معنای پایان محاسبه است. سپس PC یکی زیاد میشود و دستور بعد این چرخه را تکرار می‌کند.

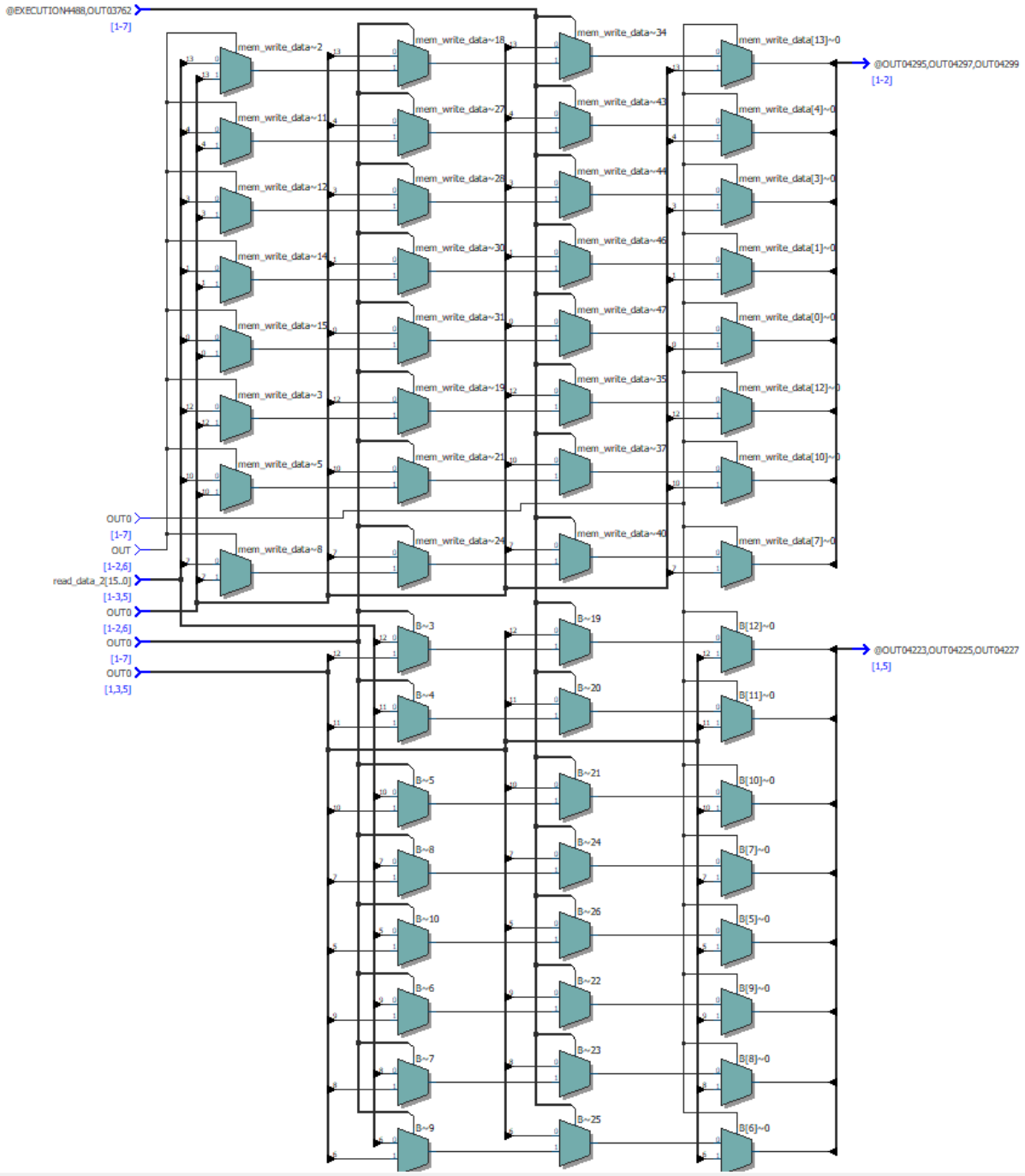
برای تست این برنامه که در واقع تست کل پردازنده است، بنده پس از تعریف متغیرهای لازم و instance گیری از controller ، به صورت مستقیم 4 عدد در 4 ثبات پردازنده قرار دادم. سپس رشته باینری 11 دستور را در آدرس های 0 تا 10 حافظه قرار دادم و pc را هم صفر کردم. در هر خط از این تست که بنده دستوری را در حافظه قرار می‌دهم، معادل شبه اسمبلی آن را هم در کنار آن کامنت کردم. همچنین حاصل مورد انتظار را هم نوشتم. در نهایت چون 11 دستور داریم، به تعداد 11 بار منتظر یک شدن ready می‌مانیم. (بنده در سایر ماژول ها از done استفاده کردم ولی چون در صورت سوال از ready استفاده شده، در نهایت برای ماژول اصلی بنده هم از ready استفاده کردم.) در نهایت مقادیر ثبات ها و حافظه در کنار مقدار مورد انتظار چاپ می‌شود.

خدا رو شکر همه چیز طبق انتظار است. در دستورات بنده از هر نوع دستور محاسباتی حداقل دو بار استفاده کرده ام. همچنین load , store را هم یک بار در دستورات قرار دادم.

❖ خدا رو شکر تمام ماژول ها را تست کردم و تست نهایی هم مطابق انتظار بود. آخرین بخش این برنامه هم مربوط به سنتزپذیری آن می‌باشد. در تمام مراحل سعی کردم اصول سنتز پذیری کد وریلاگ را رعایت کنم. در نهایت به کمک Quartus این برنامه را سنتز کردم و مشکلی در سنتز وجود نداشت. این نتیجه سنتز این برنامه ها در کوارتوس است:



به عنوان حسن ختام این پروژه این هم عکس مداری که کوارتوس کشیده است را قرار می‌دهم:


@EXECUTION4488,OUT03762
[1-7]
mem_write_data~2
mem_write_data~11
mem_write_data~12
mem_write_data~14
mem_write_data~15
mem_write_data~3
mem_write_data~5
mem_write_data~8
mem_write_data~18
mem_write_data~27
mem_write_data~28
mem_write_data~30
mem_write_data~31
mem_write_data~19
mem_write_data~21
mem_write_data~24
mem_write_data~34
mem_write_data~43
mem_write_data~44
mem_write_data~46
mem_write_data~47
mem_write_data~35
mem_write_data~37
mem_write_data~40
mem_write_data[13]~0
mem_write_data[4]~0
mem_write_data[3]~0
mem_write_data[1]~0
mem_write_data[0]~0
mem_write_data[12]~0
mem_write_data[10]~0
mem_write_data[7]~0
@OUT04295,OUT04297,OUT04299
[1-2]
OUT0
[1-7]
OUT
[1-2,6]
read_data_2[15..0]
[1-3,5]
OUT0
[1-2,6]
OUT0
[1-7]
OUT0
[1,3,5]
B~3
B~4
B~5
B~8
B~10
B~6
B~7
B~9
B~19
B~20
B~21
B~24
B~26
B~22
B~23
B~25
B[12]~0
B[11]~0
B[10]~0
B[7]~0
B[5]~0
B[9]~0
B[8]~0
B[6]~0
@OUT04223,OUT04225,OUT04227
[1,5]