دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

مهندسی کامپیوتر

گزارش پروژه معماری کامپیوتر

نگارش

آرش ضیایی، فربد حاجی محمدعلی، امیرمحمد درخشنده

استاد درس

استاد سربازی آزاد

خرداد ۱۴۰۲

به نام خدا

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

عنوان:

گزارش پروژه معماری کامپیوتر

نگارش:

امیرمحمد درخشنده - شماره دانشجویی: ۴۰۰۱۰۱۱۵۳

فربد حاجی محمدعلی - شماره دانشجویی: ۴۰۱۰۱۱۰۳۸

آرش ضیایی - شماره دانشجویی: ۴۰۰۱۰۵۱۰۹

فهرست مطالب

۱ موضوع پروژه ۱

۲ فاز اول ۲

۲-۱ هدف فاز ۲

۲-۲ Datapath و طراحی CPU ۴

۲-۲-۱ instruction spliter ۴

۲-۳ Control Unit ۵

۲-۴ ALU ۷

۳ فاز دوم ۱۲

۳-۱ هدف فاز ۱۲

۳-۲ $2-way-set-associative$ چیست؟ ۱۴

۳-۳ فایل های اضافه شده ۱۶

۳-۳-۱ $get-biggest.circ$ ۱۷

۳-۳-۲ $set.circ$ ۱۸

۳-۳-۳ $block.circ$ ۲۰

۳-۳-۴ $cache.circ$ ۲۱

۴ منابع ۲۲

فصل ۱

موضوع پروژه

در این پروژه، قرار است به پیاده سازی یک پردازنده ی MIPS بپردازیم. پیاده سازی پروژه به صورت شماتیک انجام می شود و در طول آن، برای طراحی و پیاده سازی پردازنده، از نرم افزار logisim-evolution استفاده خواهیم کرد.

این پروژه به صورت کلی شامل ۴ فاز خواهد بود. در فاز اول میخواهیم Datapath کلی پروژه را بسازیم. در فاز دوم، ماژول Cache را وارد پردازنده مان میکنیم. در فاز سوم، کاری خواهیم کرد که پردازنده ما، دستورات را به شکل pipeline و در ۵ مرحله انجام دهد. در فاز نهایی نیز، یک dynamic branch predictor به صورت one-level به مدارمان اضافه خواهیم کرد.

فصل ۲

فاز اول

۲-۱ هدف فاز

در فاز اول پروژه، میخواهیم یک Datapath جامعی از پردازنده مان طراحی کنیم. برای این منظور، ماژول های RegisterFile و Memory را از قبل در اختیار داریم. ورودی پردازنده نهایی مان قرار است یک دستور ۳۲ بیتی را بگیرد و مطابق با توضیحات ارائه شده، خروجی مدنظر را بسازد. دستورات به طور کلی دارای سه فرمت متفارت هستند که در ادامه بیان شان میکنیم:

- فرمت R
- فرمت I
- فرمت J

در دستورات R ، فرمت دستورات به شکل زیر خواهند بود:

Opcode	rs[3]	rt[2]	rd[1]	Sh.Amount	Func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

شکل ۲-۱: فرمت دستورات R

در این فرمت، دستورات شامل سه رجیستر است که دو رجیستر rs و rt مبدأ هستند و رجیستر rd مقصد است. در این فرمت، opcode تمام دستورات برابر با صفر است و تفاوت بین دستورات با توجه به فیلد func مشخص می شوند. همچنین فیلد Sh.amount برای دستورات شیفت به

راست یا چپ استفاده میشوند.

در دستورات I ، فرمت دستورات به شکل زیر خواهند بود:

Opcode	rs	rt	Imm[۴]
6 bits	5 bits	5 bits	16 bts

شکل ۲-۲: فرمت دستورات I

در این فرمت تمایز بین دستورات با فیلد opcode مشخص میشود که این مقدار مخالف صفر است.

در دستورات J نیز، فرمت دستورات به شکل زیر خواهند بود:

Opcode	address
6 bits	26 bits

شکل ۲-۳: فرمت دستورات J

همچنین توجه کنید که تعداد ثبات های عمومی و طول کلمه در این معماری، ۳۲ بیت است.
حال به سراغ طراحی Datapath اصلی پردازنده مان میرویم.

۲-۲ Datapath و طراحی CPU

برای مشاهده مدار این بخش، به فایل cpu.circ مراجعه کنید. ابتدا، دستور ورودی ما به وسیله PC یا همان Program counter ، از داخل ماژول ،Memory که از ابتدا آنرا در اختیار داشتیم، گذر کرده و دیتا خوانده شده در این ماژول - که آنرا MemReadData مینامیم - را وارد ماژول جدیدی (instSpliter) که نحوه طراحی آنرا در ادامه بیان میکنیم، میکنیم.

۲-۲-۱ instruction spliter

این ماژول، در واقع مدار درونی پیچیده ای ندارد. صرفا با توجه به فرمت ورودی دستور، بخش های مختلف آنرا از دستور اصلی، جدا میکنیم و در خروجی های آن نمایش میدهیم. خروجی های آن نیز دارای ،target OP دارای ۲۶ بیت، imm دارای ۱۶ بیت، ۳ رجیستر، Sh.amount و مقدار func است.

حال که بخش های مختلف دستور ورودی مان را به دست آورده ایم، ابتدا نیاز داریم که فرمت دستورمان را برای شروع کار، شناسایی کنیم. (توجه کنید که تا الان، در ،instSpliter ما فقط بلوک های ۵ یا ۶ و ... بیتی مختلف را - طبق ۳ حالت کلی ای که داریم - جدا کرده ایم و در حال حاضر، نمیدانیم که کدام بلوک ها به کار ما خواهند آمد!) برای شناسایی فرمت دستور، یک ماژول CU یا Unit Control طراحی میکنیم که با ورودی گرفتن OP (که در هر ۳ حالت ممکن، ۶ بیت ابتدایی دستور است) و ۶ بیت آخر دستور (که در دستورات R-format ،بتوان از آن استفاده کرد)، فرمت دستور و تعدادی سیگنال خروجی که در آینده نیاز خواهیم داشت را طراحی کند.

Control Unit ۳-۲

در control unit مان، در ورودی، op یا opcode و همچنین func را داریم. ابتدا ورودی op که شامل ۶ بیت است را به ۱۲ بیت متمایز تقسیم میکنیم(هر کدام از بیت ها و not اش)، و سپس طبق جدولی که از پیش داشتیم، و استفاده از گیت های ،and مشخص میکنیم که هر op برای چه دستوری خواهد بود. در نهایت نیز طبق اینکه هر کدام از دستور ها برای کدام فرمت هستند، آیا به موری نیاز دارند، آیا به رجیستر نیاز دارند، آیا به $ALU(ArithmeticLogicUnit)$ نیاز دارند و ...، سیگنال های خروجی مورد نیازمان را درست میکنیم. به عنوان مثال، سیگنال Branch در خروجی، هر وقت ۱ شود، به معنای آن است که دستور وارد شده، دستوری شرطی خواهد بود. اگر سیگنال ALU-V برابر ۱ شود، به معنای آن است که دستور وارد شده، به ماژول ALU برای حساب کردن حاصل، نیاز خواهد داشت و الی آخر.

در ادامه، بیان میکنیم که هر یک از سیگنال های خروجی، بیانگر چه چیزی در Control unit هستند:

- ALUSrc: در صورتی که حداقل یکی از بیت های Opcode ناصفر باشد، برابر با ۱ میشود. علت نیز شناسایی آن است که در فرمت R هستیم یا خیر. داشتیم که در فرمت ،R Opcode برابر با صفر است و در سایر فرمت ها، Opcode ناصفر است.
- ALUop: همان Opcde را در خروجی نمایان میکند(هدف از تولید مجدد همین بلوک، برای یکپارچگی ورودی های ALU است)
- ALU-V: برای آن است که مشخص کنیم دستور ورودی نیازی به ALU دارد یا خیر. در دستورات Syscall و ،Jal نیازی به ALU نخواهیم داشت.
- RegWrite: اگر دستورمان Jr و تعداد زیادی از دستورات در فرمت R نباشد، این سیگنال برابر با ۱ میشود. از این سیگنال برای شناسایی دستوراتی استفاده میشود که نیاز به آپدیت کردن یک رجیستر داریم.
- RegDst: مشخص میکند که آیا در دستور R هستیم یا خیر. از این سیگنال برای مشخص کردن نیازمان به رجیستر مقصد استفاده میکنیم.
- MemToReg: در صورت ۱ بودن این سیگنال، یعنی نیاز به خواندن چیزی از موری و اضافه کردن آن به رجیستری که در دستور قرار دارد، داریم. مانند lw و lb که باید چیزی را از موری خوانده و آنرا در رجیستری ذخیره کنیم.

- MemWriteMask: بنابر جدول داخل داک، در دستورات lb و sb صرفا نیاز به ۸ بیت اول (یا در واقع بایت اول عدد) داریم. در صورتی که دستورمان یکی از این دو دستور باشد، خروجی این سیگنال برابر با ۰۰۰۱ خواهد بود. این خروجی بدان معنا است که در مموری، برای مقدار en ورودی ۴ رم موجودمان، فقط رمی که بایت اول را دارا میباشد، فعال میشود و مقدارش دست خوش تغییراتی میشود و بقیه بایت ها دست نخورده باقی میمانند و نیازی به آنها نداریم. همچنین در دستورات lw و ،sw نیاز به هر ۴ بیت داریم. پس خروجی را به شکل ۱۱۱۱ تنظیم میکنیم که در مموری، ورودی en هر ۴ رم فعال باشد که بتوانیم از آنها استفاده کنیم. در دستورات دیگر نیز نیازی به مموری نداریم و خروجی این سیگنال برابر با ۰۰۰۰ خواهد بود.

- Branch: این سیگنال مشخص میکند که دستورمان شرطی است یا خیر. مقدار آن نیز برابر با or سیگنال های Branch ها است. در صورتی که حداقل یکی از آنها برابر با ۱ باشند، خواسته ما برآورده میشود.

- Jump: مشابه سیگنال Branch است و بیان میکند که در دستور Jump قرار داریم یا خیر.

- syscall: نشان میدهد که آیا دستورمان syscall است یا خیر. این بیت در PC ، مقدار key enable را تعیین میکند و در صورتی که ۰ باشد، PC غیر فعال میشود.

- JAL: نشان میدهد که آیا دستورمان JAL است یا خیر. این بیت در PC ، آدرس دستور را در رجیستر شماره ۳۱ ذخیره میکند.

ALU ۴-۲

در ماژول ،ALU همانطور که در داک توضیحات داشتیم، تعدادی عملیات باید بنا بر دستور ورودی روی داده های داخل رجیستر های مشخص شده انجام شود. به عنوان مثال عملیات جمع، تفریق، Xor و .... حال ماژرول ALU مان، باید بخش های مختلف دستور ورودی را، که پیش تر به وسیله spliter ،instruction اجزای دستور اصلی را از هم جدا کرده بودیم، به ورودی ماژول بدهیم. همچنین، چند ورودی به جز اعداد instruction spliter نیز نیاز خواهیم داشت، مانند PC.

حال این سوال مطرح میشود که هدف کلی این ماژول چیست؟

در این ماژول، قرار است که عملیات های مختلف روی اعداد ورودی مان انجام شود، و در همین ماژول ذخیره شوند، سپس با توجه به ورودی op و func ، و همچنین با استفاده از Mux با اندازه های متفاوت، مشخص کنیم که کدام یکی از پاسخ هایی که ساخته ایم باید در خروجی نمایش داده شود. مسئولیت تعیین این که کدام عملیات را میخواهیم انجام بدهیم، با ماژول دیگری است که آنرا ALU Controller مینامیم و در ادامه معرفی اش میکنیم. اما پیش از آن، سیگنال های ورودی و خروجی ماژول ALU مان را معرفی میکنیم و عملیات های آنها را بررسی میکنیم:

- **ورودی های اصلی**
  - A: عدد اول ورودی که از رجیستر ورودی در دستور اولیه استخراج کرده ایم.
  - B: عدد دوم ورودی که از رجیستر ورودی در دستور اولیه استخراج کرده ایم.
  - ALUOp: پیش تر در Control Unit آنرا ساخته و بررسی کرده بودیم.
  - Func: پیش تر در Control Unit آنرا ساخته و بررسی کرده بودیم.
  - Imm: عدد immadiate ورودی که در دستور اولیه، آن را استخراج کرده ایم.
  - Shamt: مقدار Shift رو به راست یا چپ را نشان میدهد.
  - ALUSrc: پیش تر در Control Unit آنرا ساخته و بررسی کرده بودیم.

- **عملیات های منطقی**

این دسته، شامل ۴ حالت اصلی هستند که هریک را بررسی میکنیم:

- $OrGate$: این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل B OR A را خروجی میدهد و در دیگری، حاصل Imm OR A را خروجی میدهد (در صورتی که دستورمان I-Format باشد، محتویات رجیستر A با مقدار Imm باید Or شود).
- $AndGate$: این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل AND A B را خروجی میدهد و در دیگری، حاصل Imm AND A را خروجی میدهد (در صورتی که دستورمان I-Format باشد، محتویات رجیستر A با مقدار Imm باید AND شود).
- $NorGate$: این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل NOR A B را خروجی میدهد و در دیگری، حاصل Imm NOR A را خروجی میدهد (در صورتی که دستورمان I-Format باشد، محتویات رجیستر A با مقدار Imm باید NOR شود).
- $XORGate$: این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل XOR A B را خروجی میدهد و در دیگری، حاصل Imm XOR A را خروجی میدهد (در صورتی که دستورمان I-Format باشد، محتویات رجیستر A با مقدار Imm باید XOR شود).

توجه کنید که ۸ خروجی تولید شده را به فرمت $AfunctionB$ ذخیره میکنیم. به عنوان مثال، مقدار AND شده A و B را به شکل A-and-B ذخیره میکنیم.

- **عملیات های ریاضی**

این دسته شامل ۹ حالت است که هر یک را بررسی میکنیم:

- add: جمع ساده دو عدد A و B
- sub: تفاضل ساده دو عدد A و B
- addi: جمع ساده دو عدد A و Imm
- subi: تفاضل ساده دو عدد A و Imm
- addu: جمع unsigned دو عدد A و B
- subu: تفاضل unsigned دو عدد A و B
- addiu: جمع unsigned دو عدد A و Imm
- Mult: ضرب ساده دو عدد A و B
- Div: تقسیم ساده دو عدد A و B

توجه کنید که مقادیر خروجی تولید شده در این عملیات ها، هر کدام متناظر با متغیر تعریف شده در بالا هستند، به عنوان مثال، حاصل تفاضل دو عدد A و B به شکل sub وجود دارد.

- **عملیات های شیفت**

این دسته شامل ۵ حالت است که هر یک را بررسی میکنیم:

- S-LL: مقدار شیفت رو به چپ عدد A را به اندازه Shamt در خروجی می سازد.
- S-LLV: مقدار شیفت رو به چپ عدد A را به اندازه B در خروجی می سازد.
- S-RL: مقدار شیفت رو به راست عدد A را به اندازه Shamt در خروجی می سازد.
- S-RLV: مقدار شیفت رو به راست عدد A را به اندازه B در خروجی می سازد.
- S-RA: مقدار شیفت sign رو به راست عدد A را به اندازه Shamt در خروجی می سازد.

توجه کنید که مقادیر خروجی تولید شده در این عملیات ها، هر کدام متناظر با متغیر تعریف شده در بالا هستند.

- **گیت های شرطی**

در این بخش، ۳ عملیات داریم که علامت یکی از اعداد را مشخص خواهند کرد (در واقع باید مقاسیه کنیم که عدد ورودی از صفر بیشتر، کمتر و یا مساوی با آن است). ۳ عملیات هم داریم که مقایسه دو عدد ورودی خواهد بود. حال هریک را در ادامه بررسی میکنیم:

- مقایسه عدد A با صفر
  - BGTZ: مشخص میکند که آیا عدد A از ۰ بیشتر است یا خیر.
  - BGEZ: مشخص میکند که آیا عدد A بیشتر یا مساوی ۰ است یا خیر.
  - BLEZ: مشخص میکند که آیا عدد A کمتر یا مساوی ۰ است یا خیر.
- مقایسه عدد A با B
  - BNE: مشخص میکند که آیا عدد A و B نابرابر هستند یا خیر.
  - BEQ: مشخص میکند که آیا عدد A و B برابر هستند یا خیر.
  - SLT: اگر عدد A از B کوچکتر باشد، بیت خروجی را برابر با ۱ قرار میدهد.

توجه کنید که در هر دو بخش، با استفاده از یک comparator ، مقادیر ورودی را با یکدیگر مقایسه کردیم. ساختار comparator نیز بدین شکل است که دو عدد در ورودی گرفته و ۳ بیت خروجی دارد که در هر لحظه، دقیقا یکی از آنها برابر با یک خواهد بود. یکی برای حالتی که عدد اول از عدد دوم بزرگتر باشد، یکی برای حالت تساوی و در نهایت بیت سوم برای حالتی که عدد اول از عدد دوم کوچکتر باشد.

- **Mux ها:**

تعدادی Mux ۵ * ۳۲ داریم که ورودی های هر یک را، متناظر با function مورد نظر و مقدار خروجی ای که درست کردیم، قرار میدهیم. توجه کنید که علت استفاده از این ۲ Mux ، آن است که ورودی های selector آنها با یکدیگر فرق دارند. همچنین در ورودی های دیتا این ماژول ها، متناظر با func مورد نظر، دیتا ساخته شده را قرار میدهیم. برای این کار نیز از یک Splitter استفاده کرده ایم که ۵ بیت اول و بیت ششم مقدار func را از یکدیگر جدا میکند و مقدار ۵ بیتی را در سلکتور Mux ها قرار میدهد. در نهایت، خروجی در Mux را وارد یک Mux ۱ * ۲ میکند که سلکتور آن، همان بیت ششم خواهد بود.

علت: دستورات ریاضی و منطقی دارای بیت ششم ۰ در func هستند، در حالی که دستورات شیفت و ضرب و تقسیم، دارای بیت ششم یک هستند.

همچنین موازی با این عملیات ها، یک Mux ۵ * ۳۲ دیگر نیز داریم که دستورات شرطی و تعدادی از دستورات که در فرمت R نیستند را در بر میگیرد. صرفا ورودی سلکتور آن، برابر با ۵ بیت اول ALUOp یا همان opcode خواهد بود.

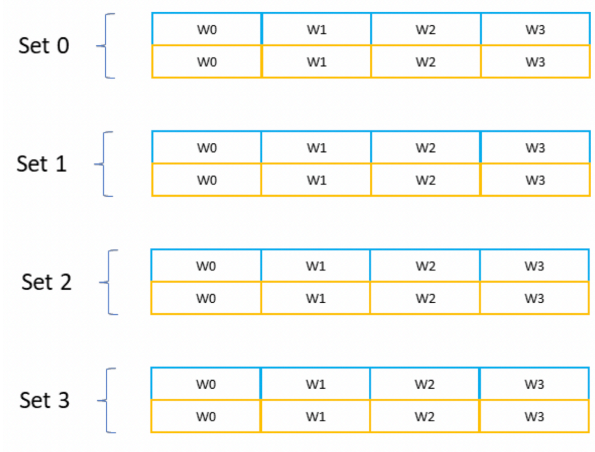
در نهایت نیز، خروجی این Mux و خروجی Mux ۱ * ۲ ای که پیش تر گفتیم، وارد یک Mux ۱*۲ دیگر میشوند که سلکتور آن، بیت ALUSrc است که در Control Unit ساخته بودیم. خروجی این Mux ، خروجی نهایی ۸ بیتی ALU خواهد بود.

فصل ۳

فاز دوم

۳-۱ هدف فاز

در فاز دوم پروژه، قصد داریم یک cache برای پردازنده مان طراحی کنیم. هدف این فاز، پیاده سازی یک ماژول حافظه ی نهان یا همان cache و افزودن آن به پردازنده ی فاز قبل می باشد. ماژول حافظه ای که در این فاز در اختیار داریم، مشابه ماژول حافظه ای است که در فاز قبل استفاده کرده بودیم، با این تفاوت که چند کلاک زمان می برد تا خروجی را آماده کند. برای ساده شدن طراحی، تعداد کلاک های مورد نیاز برای این حالات، که زمان میگیرد تا خروجی را آماده کند، برابر با مقدار ثابتی خواهد بود و نیازی به استفاده از سیگنالی مانند ready و یا در خروجی memory نخواهد بود.



شکل ۳-۱: فرمت بلاک های cache

توجه کنید که در نظر داریم که طراحی خود را به گونه ای تغییر دهیم که در صورت رخ دادن

miss هنگام دسترسی به cache ، پردازنده، به اندازه ی تعداد کلاک مشخصی منتظر بماند تا ماژول cache، داده را از حافظه بخواند و در اختیار پردازنده قرار بدهد. در واقع cache به صورت یک میانجی بین حافظه و پردازنده قرار است عمل کند و دیگر دسترسی مستقیم به حافظه نخواهیم داشت و همه ی دسترسی ها از مسیر cache عبور میکند.

طراحی ماژول cache را میخواهیم به صورت $2 - way - set - associative$ انجام دهیم و در کل نیز ۴ ست خواهیم داشت. در نهایت نیز اندازه هر بلاک برابر با ۴ کلمه خواهد بود. برای درک بهتر، به تصویر شماره ۳-۱ توجه کنید.

از تصویر فوق برای طراحی ماژول cache استفاده خواهیم کرد. همچنین برای سیاست جایگزینی، یعنی انتخاب way موردنظر، میخواهیم رروش LRU را پیاده سازی کنیم. با توجه به $2 - way$ بودن هر ست، پیاده سازی True LRU گزینه ای منطقی و ساده به حساب می آید.

لازم به ذکر است که طراحی این ماژول قرار است به صورت write back انجام گیرد. یعنی داده مورد نظر فقط در cache نوشته می شود و هنگام جایگزین شدن، در حافظه نوشته می شود.

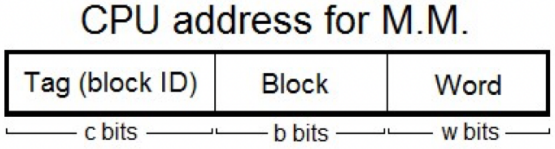
در نهایت در نظر داشته باشید که طراحی ماژول cache ، فقط برای حافظه ی داده یا همان data memory انجام می گیرد و نیازی به طراحی ماژول جداگانه برای instruction memory نخواهد بود و این حافظه مانند قبل، دستورات را در طی یک clock و بدون تاخیر در اختیارمان قرار خواهد داد.

۲-۳ $2-way-set-associative$ ۲ چیست؟

به طور کلی برای mapping ، ۳ روش جامع داشتیم که در واقع، دو تا از آنها، زیر مجموعه سومی هستند:

- direct mapping
- fully-associative mapping
- set-associative mapping

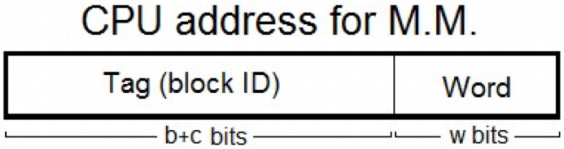
در روش اول، آدرسی که از CPU خوانده میشود، شامل ۳ قسمت Tag ، Block و Word است که هر یک به ترتیب شامل c ، b و w بیت هستند و روی هم، آدرس مورد نظر را در فرمت شکل ۳-۲ میسازند:



شکل ۳-۲: direct mapping cpu address format

به عبارتی، c بیت اول، آدرس کش مموری را مشخص میکنند، b بیت بعدی، آدرس بلاک را مشخص میکنند و در نهایت، w بیت آخر، کلمه مورد نظر در بلاک را بیان میکنند.

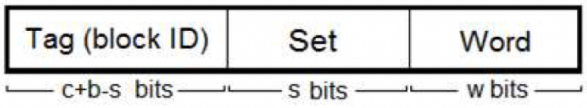
در روش دوم، بیت های block و tag با یکدیگر ادغام شده و به طور کلی، آدرس خوانده شده از CPU ، شامل دو بخش است! در ابتدا، c + b بیت برای آدرس بلاک و در نهایت w بیت برای مشخص کردن کلمه مورد نظر خواهیم داشت:



شکل ۳-۳: fully associative mapping cpu address format

در این روش، c + b بیت برای مشخص کردن بلاک استفاده میشود که به شکل مستقیم، لاین ای که ۲ به توان w کلمه در آن قرار دارد را مشخص میکند و بعد کلمه مورد نظر یافت میشود.

در روش آخر، که دو روش قبلی زیر مجموعه ای از آن خواهند بود، بلاک ها در قالب تعدادی مجموعه به نام set دسته بندی میشوند:



شکل ۳-۴: set-associative mapping cpu address format

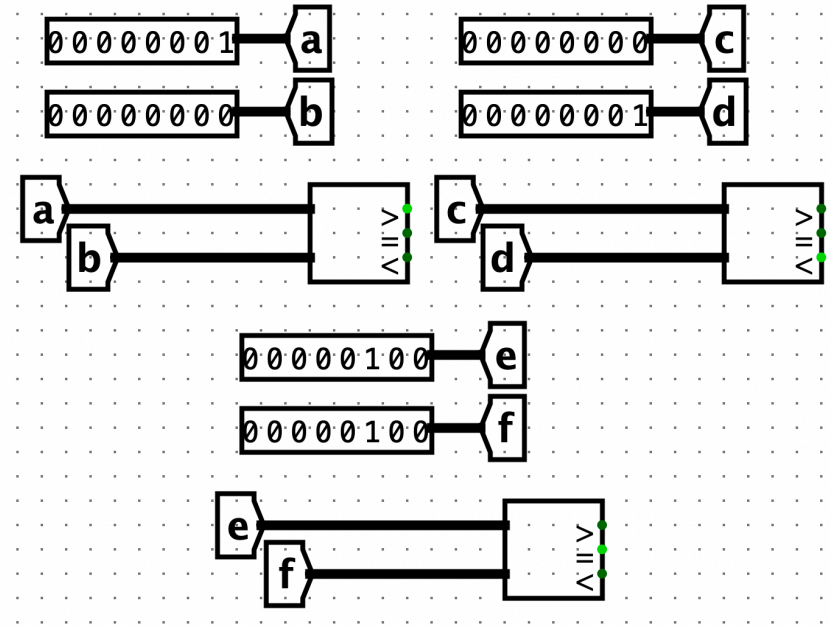
در این روش، s بیت ابتدا set مورد نظر را مشخص میکنند، سپس $b+c-s$ بیت، آدرس بلاک مورد نظر در این ست را مشخص میکنند و در نهایت w بیت نهایی، کلمه مورد نظر در بلاک مورد نظر را مشخص میکنند.

۳-۳ فایل های اضافه شده

در این فاز، به محتویات فایل circuits ، تعدادی فایل جدید اضافه شده اند که در ادامه، هر یک را جداگانه بررسی میکنیم:

- $get - biggest.circ$
- $cache.circ$
- $cacheController.circ$
- $block.circ$
- $set.circ$

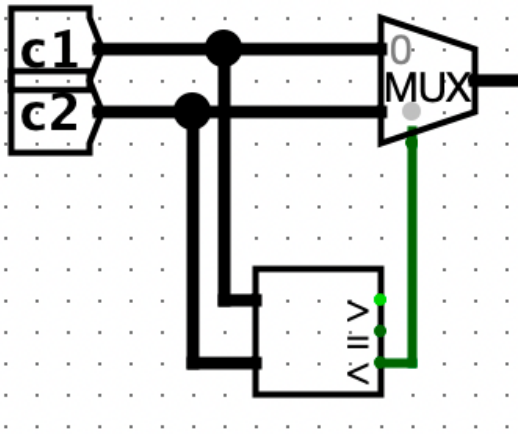
همچنین در این فایل ها، از ماژولی به نام comparator استفاده شده است که نحوه کار آن، به این صورت است که دو ورودی با نام های a و b را اگر بگیرد، در صورتی که $a > b$ باشد، خروجی اول فعال شده و بقیه غیر فعال هستند. در صورت برابری این دو ورودی، خروجی دوم و در نهایت در صورتی که $b > a$ باشد، خروجی سوم فعال خواهد بود. شکل ۳-۲، این ماژول را نشان میدهد که دو ورودی را گرفته و برای وضعیت های مختلف، خروجی مورد نظر فعال میشود:



شکل ۳-۵: ماژول comparator

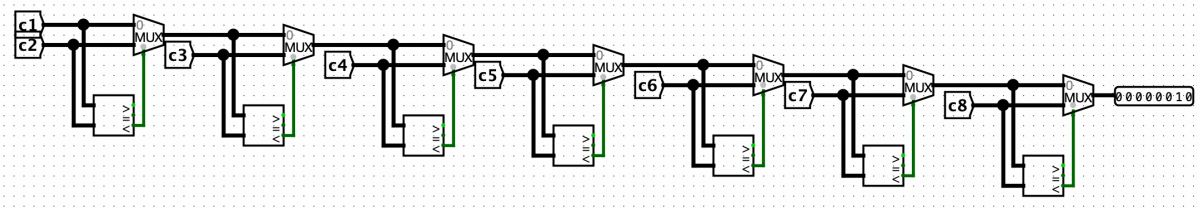
۱-۳-۳ $get - biggest.circ$

حال در فایل اول، یعنی فایل $get - biggest.circ$، هدفمان آن است که بزرگترین عدد میان چند ورودی را پیدا کنیم. نحوه کار به این صورت است که ۸ ورودی ۸ بیتی را در اختیار داریم و آنها را با C_1، C_2 و ... C_8 نامگذاری کرده ایم. طی ۷ مرحله، عملیات انجام شده در شکل ۳-۳ را انجام میدهیم.



شکل ۳-۶: بلاک مشخص کننده عدد بزرگتر

نحوه کار بدین صورت است که دو ورودی را به یک comparator میدهیم و خروجی سوم آن را وارد یک $Mux۲*۱$ میکنیم، یعنی در صورتی که ورودی دوم از ورودی اول، بیشتر باشد، خروجی Mux برابر با ورودی دومش است و در غیر این صورت، ورودی اولش در خروجی ظاهر میشود. دو ورودی این Mux نیز، همان دو ورودی comparator به همان ترتیب هستند. هدف از ساختن این بلاک، مشخص کردن عدد بزرگتر بین دو ورودی است، چرا که در صورتی که ورودی دوم بزرگتر باشد، خود در خروجی ظاهر میشود، در صورتی که ورودی اول بزرگتر باشد، خودش در خروجی ظاهر میشود و در صورت برابری نیز، فرقی ندارد که کدام در خروجی ظاهر میشوند(در این بلاک، همان ورودی اول را قرار داده ایم)



شکل ۳-۷: یافتن بزرگترین عدد

در نتیجه، بعد از ۷ مرحله انجام این عملیات، و ورودی دادن C_1 و C_2 در ابتدا، ورودی دادن خروجی عملیات این دو و C_3 در بلاک بعدی و الی آخر، خروجی کلی مدار شکل ۳-۴ برابر با بزرگترین عدد بین این ۸ عدد خواهد بود.

۲-۳-۳ $set.circ$

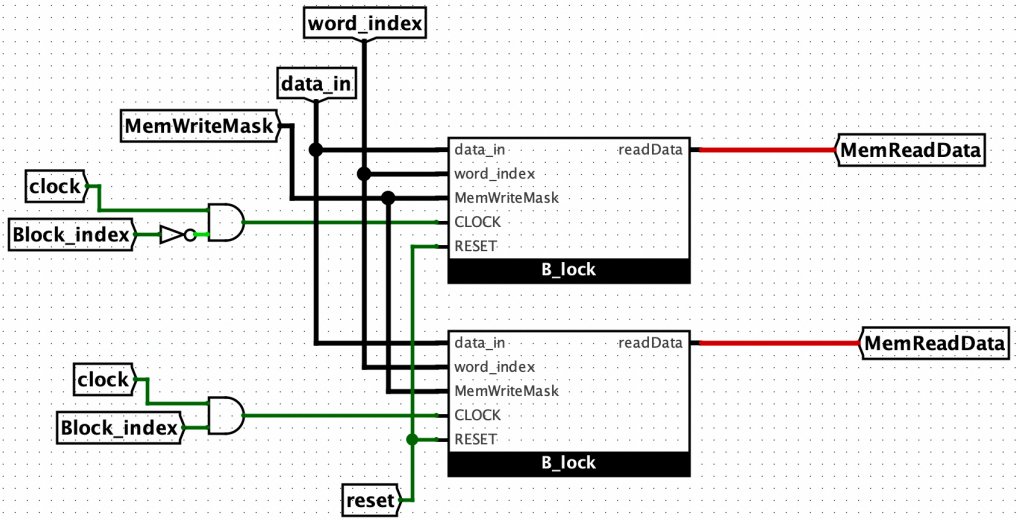
در این فایل، میخواهیم بخش set را پیاده سازی کنیم. همان طور که در تعریف فاز دو مشاهده کردیم، قرار است که حافظه نهان ما، دارای ۴ ست و هر ست دارای دو بلاک باشد. حال ما در این فایل، صرفا میخواهیم یک ست دارای دو بلاک را پیاده سازی کنیم.

ورودی های مدار:

- $data - in$
- $word - index$
- $block - index$
- $memWriteMask$
- $CLOCK$
- $Reset$

خروجی های مدار:

- $memReadData$



شکل ۳-۸: ماژول set

:$data - in$

ورودی ۳۲ بیتی برای مشخص کردن داده ورودی به cache برای نوشتن(در صورت نیاز) است. البته داخل مدار، این ورودی به شکل هگز نمایش داده شده است.

:$word - index$

این ورودی در واقع همان w بیت برای مشخص کردن کلمه در لاین مورد نظر است. از انجایی که بیان کردیم حافظه نهان ما در کل دارای ۴ ست است، و هر ست دارای ۴ بلاک و همچنین هر بلاک دارای ۴ کلمه است، در نتیجه صرفا ۲ بیت برای مشخص کردن ۴ حالت word داریم. در نتیجه این ورودی دارای ۲ بیت به شکل باینری خواهد بود.

:$block - index$

این ورودی شامی یک تک بیت خواهد بود که بیان میکند در این ست، داخل کدام بلاک باید قرار داشته باشیم. اگر ۰ باشد، در بلاک اول و در غیر این صورت، در بلاک دوم هستیم.

:$memWriteMask$

ورودی شامل ۴ بیت برای مشخص کردن بایت متناظر در کلمه مودر نظر یک بلاک است. به عبارتی، مقدار این ورودی در سیستم باینری، مشخص میکند که میخواهیم به کدام بایت های کلمه مورد نظر در بلاک مورد نظر دسترسی داشته باشیم. به عنوان مثال، اگر این ورودی برابر ۱۰۱۱ باشد، بدین معنی است که میخواهیم به بایت اول، دوم و چهارم کلمه مورد نظر، دسترسی داشته باشیم. این ورودی را در بخش block دقیق تر توضیح میدهیم.

به طور کلی، این ورودی برای بیان کردن read/write است، منتهی با بیان اینکه کدام بایت ها قرار است عوض شوند.

:$CLOCk$

پالس ساعت است.

:*Reset*

ورودی برای ریست کردن کل مدار است.

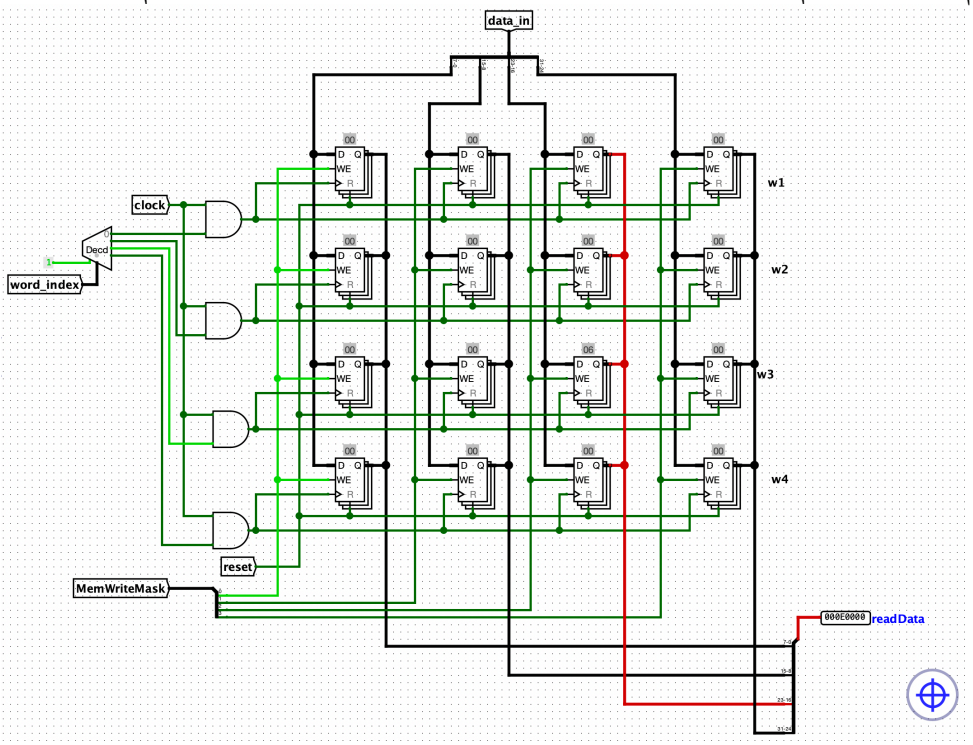
:*memReadData*

به شکل یک خروجی ۳۲ بیتی در هگز است که در هر پالس ساعت، بسته به این که ورودی کلمه مان چه چیزی است، بایت های ۰ را به صورت نظیر برابر با ۰ و بایت های غیر ۰ را برابر E قرار میدهد. به عنوان مثال، اگر برای کلمه سوم، ۰af۲d۱۰۵ را ذخیره کرده باشیم، در خروجی مقدار ۰EEEEE۰E را مشاهده خواهیم کرد.

شکل ۳-۹: نمونه برای نمایش خروجی

۳-۳-۳ *block.circ*

در بخش قبل، مدار set را تعریف کردیم و همچنین بیان کردیم که در این مدار، از دو مدار block میخواهیم استفاده کنیم. در این بخش، مدار داخلی block را بررسی خواهیم کرد:



شکل ۳-۱۰: ماژول block

همان طور که در شکل ۳-۱۰ میتوان مشاهده کرد، ورودی ها و خروجی مطابق با بخش set تعریف شده است.

در ابتدا، توسط یک $mux۲*۴$ ، ورودی word-index را چک میکنیم که بیان میکند میخواهیم به کدام کلمه در بلاک دسترسی داشته باشیم.
یادآوری: هر بلاک دارای ۴ کلمه ۴ بایتی بود. در نتیجه، در کل، ۱۶ رجیستر برای ذخیره سازی بایت های آنها نیاز داریم که رجیستر های هر لاین مربوط به یک کلمه و از راست به چپ هستند.

خروجی mux را با استفاده از ۴ گیت and ، با ورودی CLOCK ترکیب میکنیم و خروجی این ۴ لاین، مشخص میکند که در هر پالس ساعت، کدام کلمه قرار است بررسی شود.

ورودی ۳۲ بیتی(یا ۴ بایتی) data-in نیز در بالای تصویر قرار دارد که هر بایت آن، وارد بایت نظیر در هر کلمه میشود.

همچنین ورودی memWriteMask نیز دارای ۴ بیت است که مشخص میکند کدام بایت ها از کلمه مشخص شده توسط word-index قرار است write شوند.(هر بیت آن به ترما بایت های هر شماره با خودش در بخش WE وارد میشود.

یک ورودی Reset نیز برای ریست شدن مدار به تمامی رجیستر ها میدهیم.

نحوه کار این مدار نیز مشابه همان چیزی است که در بخش set بررسی کردیم، مثال شکل ۹-۳، در اینجا نیز برقرار خواهد بود.

در واقع در بخش set ، ما ۲ عدد از این بلاک قرار داده ایم. در نتیجه ساختار درونی یک بلاک را نیز در حال حاضر داریم.

۴-۳-۳ *cache.circ*

در این بخش، که مهم ترین قسمت این فاز است، ساختار درونی حافظه نهان را بررسی میکنیم:
ورودی های این مدار:

فصل ۴

منابع