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

# 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.

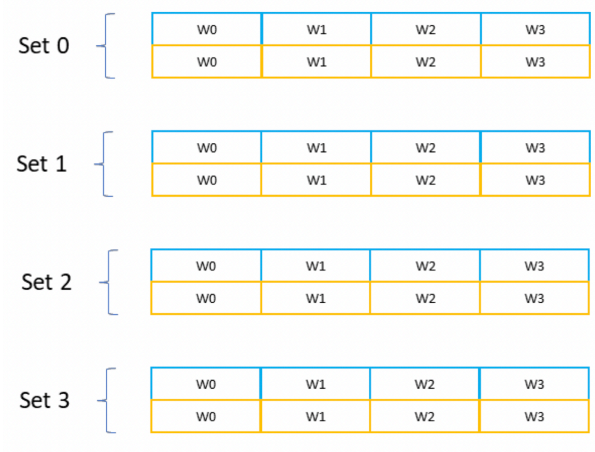
در نهایت نیز، خروجی این Mux و خروجی Mux ۱ * ۲ ای که پیش تر گفتیم، وارد یک Mux ۱*۲ دیگر میشوند که سلکتور آن، بیت ALUSrc است که در Control Unit ساخته بودیم. خروجی این Mux ، خروجی نهایی ۸ بیتی ALU خواهد بود.

# 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.

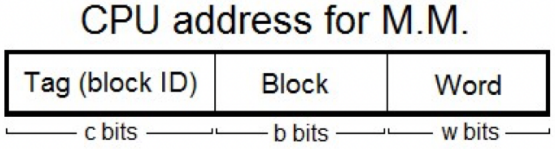
در نهایت در نظر داشته باشید که طراحی ماژول cache ، فقط برای حافظه ی داده یا همان data memory انجام می گیرد و نیازی به طراحی ماژول جداگانه برای instruction memory نخواهد بود و این حافظه مانند قبل، دستورات را در طی یک clock و بدون تاخیر در اختیارمان قرار خواهد داد.

## 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

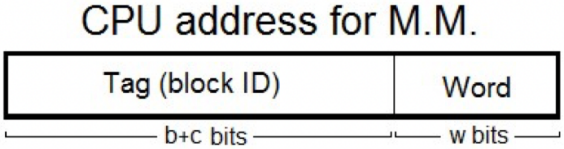
در روش اول، آدرسی که از CPU خوانده میشود، شامل ۳ قسمت Tag ، Block و Word است که هر یک به ترتیب شامل c ، b و w بیت هستند و روی هم، آدرس مورد نظر را در فرمت شکل ۳-۲ میسازند:




**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.

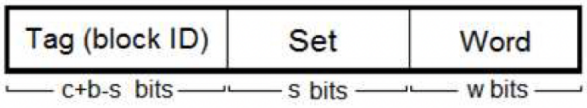
در روش دوم، بیت های block و tag با یکدیگر ادغام شده و به طور کلی، آدرس خوانده شده از CPU ، شامل دو بخش است! در ابتدا، c + b بیت برای آدرس بلاک و در نهایت w بیت برای مشخص کردن کلمه مورد نظر خواهیم داشت:




**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**

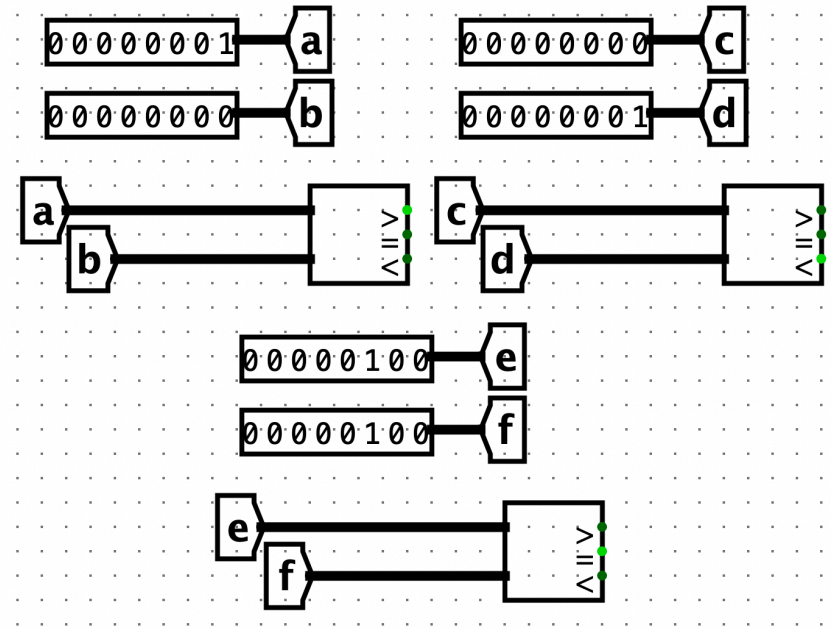
در این روش، s بیت ابتدا set مورد نظر را مشخص میکنند، سپس $b+c-s$ بیت، آدرس بلاک مورد نظر در این ست را مشخص میکنند و در نهایت w بیت نهایی، کلمه مورد نظر در بلاک مورد نظر را مشخص میکنند.

## 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:

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

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 $set.circ$

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:

- $memReadData$

$data - in$:

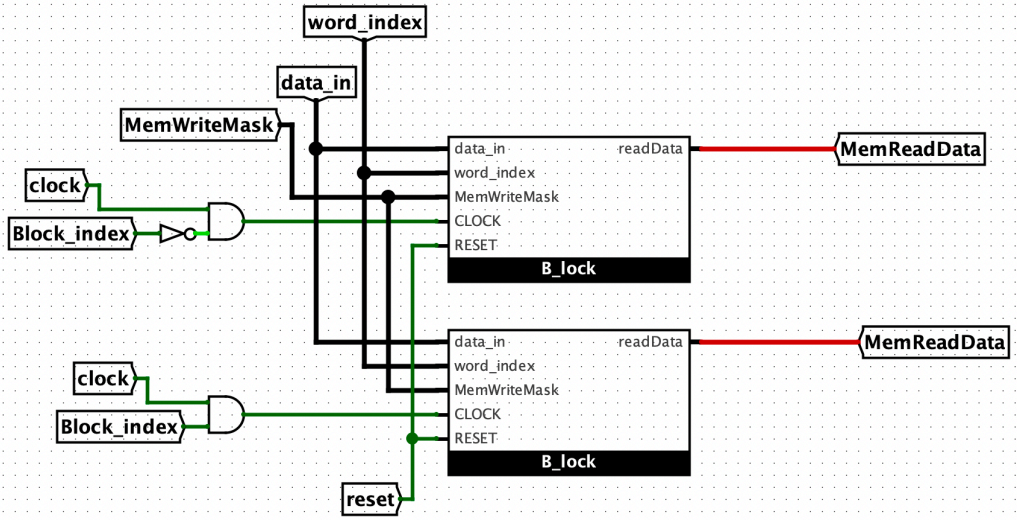
ورودی ۳۲ بیتی برای مشخص کردن داده ورودی به cache برای نوشتن(در صورت نیاز) است. البته داخل مدار، این ورودی به شکل هگز نمایش داده شده است.

$word - index$:

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.




**شکل ۳-۶: ماژول set**

$memWriteMask$:

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.

$CLOCk$ و $reset$:

پالس ساعت و ریست مدار هستند.

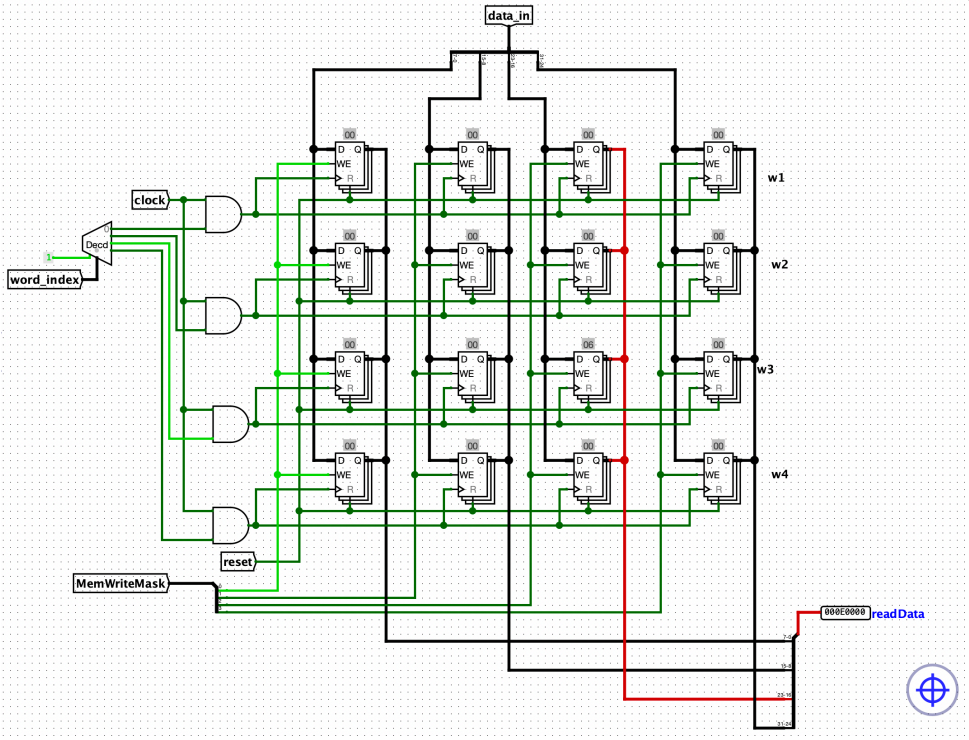
:$memReadData$

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 ۳-۷: نمونه برای نمایش خروجی**

### ۳-۳-۲ $block.circ$

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 ۳-۸: 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.

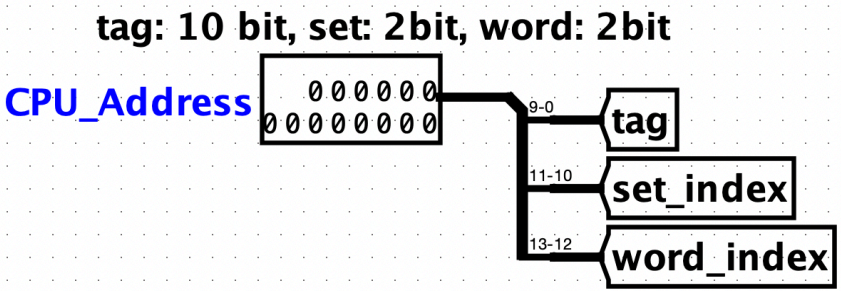
در واقع در بخش set ، ما ۲ عدد از این بلاک قرار داده ایم. در نتیجه ساختار درونی یک بلاک را نیز در حال حاضر داریم.

### ۳-۳-۳ $cache.circ$

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

- $CPU - address$
- $block - index$
- $data - in$
- $memWriteMask$
- $LRU$
- $CLOCK$
- $Reset$

حال به بررسی ورودی های مدار می پردازیم: ورودی اول، $CPU - Address$ است که همان آدرسی است که از پردازنده خوانده میشود و پیش تر به نحوه ساختاری آن اشاره کرده بودیم. در این آدرس، ۲ بیت برای مشخص کردن کلمه مورد نظر در بلاک مورد نظر قرار دارد(به علت آنکه در هر بلاک، ۴ کلمه داریم)، ۲ بیت برای مشخص کردن set مورد نظر(به علت آنکه در کل ۴ set داریم و برای مشخص کردن آن، ۲ بیت کافی است) و در نهایت میتوان نتیجه گرفت که ۱۰ بیت برای مشخص کردن $tag - ID$ نیاز است که در واقع همان مقدار $b + c - s$ را تشکیل میدهد. در نتیجه در آدرسی که از CPU خوانده میشود، بیت ۰ الی ۹ را برای بیان کردن $tag - ID$ در نظر میگیریم، بیت ۱۰ و ۱۱ را برای مشخص کردن set و در نهایت بیت ۱۲ و ۱۳ را برای مشخص کردن کلمه مورد نظر در بلاک در نظر میگیریم.




**شکل ۳-۹: فرم تجزیه آدرس پردازنده به بیت های مورد نظر**

ورودی بعدی، ورودی block-index است که پیش تر بیان کرده بودیم برای مشخص کردن بلاک مورد نظر در هر set استفاده خواهد شد(در صورت ۰ بودن، بیان گر بلاک اول و در غیر این صورت، بیان گر بلاک دوم خواهد بود)

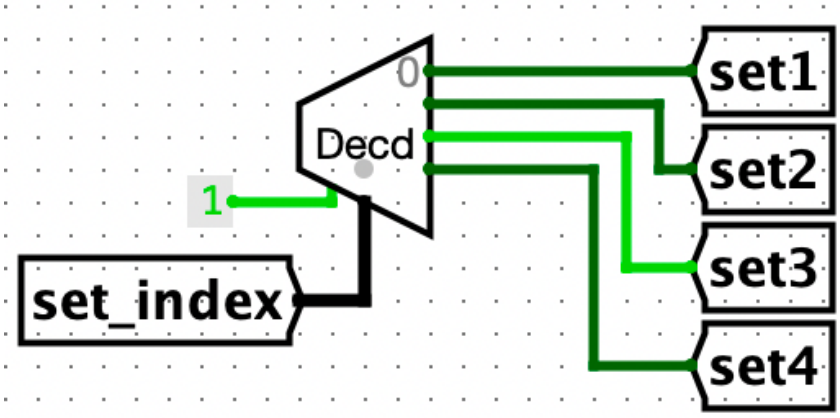
ورودی data-in همان ورودی ۳۲ بیتی یا ۸ بایتی ما خواهد بود.

ورودی memWriteMask برای مشخص کردن بیت های مورد نیاز به تغییر در هر کلمه است(در بخش set آن را کامل بررسی کردیم)

ورودی LRU یک ورودی تک بیتی است که در ادامه آنرا کامل بررسی میکنیم.

دو ورودی Clock و Reset نیز به ترتیب پالس ساعت و ریست مدار هستند.

ورودی set-index ، که پیش تر بیان کردیم بیت ۱۰ و ۱۱ آدرس خوانده شده از CPU است، بیانگر set مورد نظر برای دسترسی است. این ورودی را با استفاده از یک دیکودر ۲ به ۴، به خطوط خروجی set-۱ الی set-۴ تقسیم میکنیم که هر لاینی که فعال باشد، در واقع کلید دسترسی به set مورد نظر را مشخص میکند(در نتیجه نمیتوان دو set را به صورت همزمان در دسترس داشت، چرا که خروجی همواره یک لاین فعال و سه لاین غیر فعال خواهد بود)



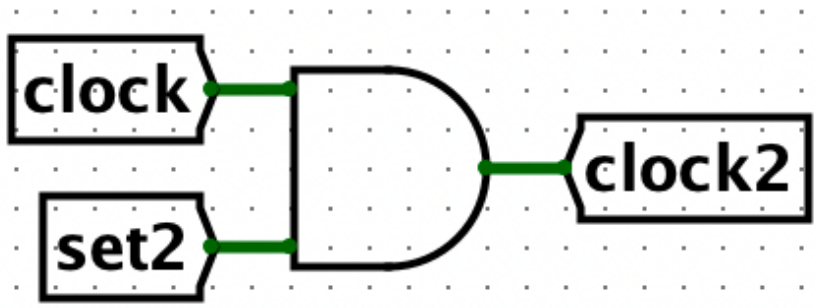

**شکل ۳-۱۰: تجزیه set-index توسط دیکودر**

در ادامه، چهار ورودی جدید را تعریف میکنیم:

ورودی های $clock-۱$ الی $clock-۴$ ، هر کدام بیانگر اجازه دسترسی به هر set در هر پالس بالارونده ساعت هستند، بدین صورت که تک بیت clock-i به شکل زیر تعریف میشود(برای i = ۱، ۲، ۳، ۴):

$$clock-i = \begin{cases} ۱ & \text{if i'th set has been considered and clock is on pos edge} \\ ۰ & \text{otherwise} \end{cases}$$

به عنوان مثال، شکل ۳-۱۳ بیان گر ۲ – $clock$ است که با استفاده از یک گیت and به دست آمده است:

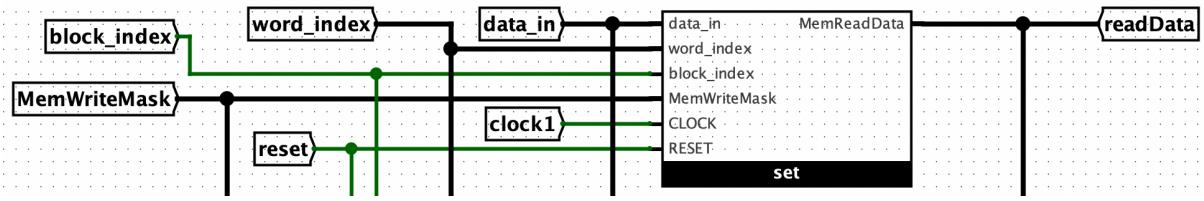



**شکل ۳-۱۱: ساخت بیت** ۲ – $clock$

حال مرحله به مرحله، حافظه نهان مان را میسازیم:

### مرحله اول: ساختار set ها در cache

در مرحله اولی، ساختار ۴ set را با استفاده از ورودی های تعریف شده، میسازیم: پیش تر بیان کرده بودیم که ساختار set به چه صورتی است، در این بخش، صرفا ۴ set مختلف را کنار هم قرار میدهیم و ورودی هارا مطابق با آن چیزی که تعریف کرده ایم، به ورودی های هر set وصل میکنیم. توجه کنید که خروجی این ۴ set ، یک readData مشترک است. شکل ۳-۱۴، یک نمونه از set هارا برای set اول نشان میدهد:




**شکل ۳-۱۲: نمونه ای از set ها**

توجه کنید که برای هر set ، ورودی $clock$ برابر با $clock - i$ است که i ، set نظیر است(به عنوان مثال، در شکل ۳-۱۴ ، $clock - ۱$ برای set اول در نظر گرفته شده است)

**مرحله دوم: تعاریف و ذخیره سازی بیت های کنترلی**

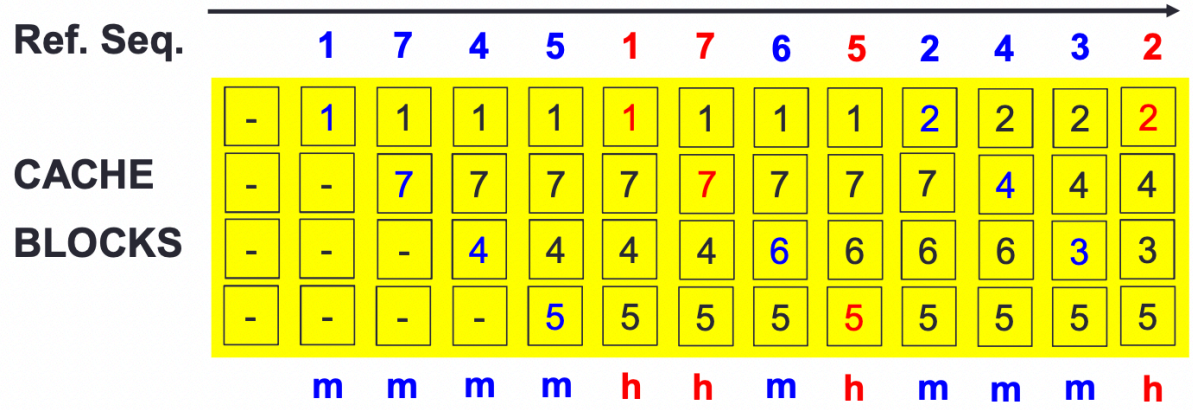
در این مرحله، ابتدا نیاز است که چند مفهموم را بیان کنیم و سپس مطابق با آنها، این مرحله از ساخت حافظه نهان مان را تکمیل کنیم:

- $dirty-bit$: هنگامی که یک بلاک از cache آپدیت میشود، نیاز است که این مقدار آپدیت شده، در $main-memory$ نیز آپدیت شود. این رخداد از لحاظ سرعت و کارایی خوب است، اما از لحاظ consistency یا همروندی میان اجزا ( cache ، مموری و $I/O-devices$) مطلوب نیست. برای درست کردن این مشکل، از یک تک بیت به نام $dirty-bit$ استفاده میکنیم. برای هر بلاک cache ، یک $dirty-bit$ در نظر میگیریم و زمانی که هر کلمه ای در بلاک، آپدیت میشود، این $dirty-bit$ را برابر با ۱ قرار میدهیم. در نتیجه بعد از آپدیت ها، بلاک های cache ای که دارای $dirty-bit$ برابر با ۱ هستند، در مموری مجددا بازنویسی میشوند.
- $replacement-policy$: هنگامی که در هر کدام از روش های mapping ، مقادیری در هر کلمه موجود باشند و دچار miss شویم(miss هنگامی رخ میدهد که مقداری که در cache میخواهیم، وجود نداشته باشد) ، نیاز است که مقدار جدید به گونه ای در cache با یکی از مقادیر قبلی، جایگزین شود. برای این کار، روش های متعددی وجود دار:
  - $random-policy$: به صورت random ، یکی از کلمه هارا انتخاب کرده و مقدار آن را جایگزین میکند.
  - $FIFO-policy$: به صورت $first-in-first-out$ عمل میکند.
  - $LRU-policy$: به صورت $least-recently-used$ عمل میکند و خود دارای دو فرم ساده تر است:
    - $NRU-policy$ : $not-recently-used$
    - $pseudo-LRU-policy$
  - $LFU-policy$: به صورت $least-frequently-used$ عمل میکند.
  - $opt-policy$: به صورت optimal عمل میکند. توجه کنید که این policy در واقعیت، امکان پذیر نیست، چراکه باید از آینده اطلاع داشته باشیم!
- $LRU$: در این policy ، ابتدا با اولویت میزان دسترسی داشتن به داده ها و سپس با اولویت زمان آپدیت شدن، به یک آدرسی میرسیم که هم زودتر وارد cache شده است و هم کمتر

دسترسی به آن داشته ایم. به عنوان مثال، در شکل ۳-۱۵، میتوان وارد شدن دیتا

۲ – ۳ – ۴ – ۲ – ۵ – ۶ – ۷ – ۱ – ۵ – ۴ – ۷ – ۱

به حافظه نهان را مشاهده کرد:




**شکل ۳-۱۳: نمونه ای از** $LRU - policy$

- $valid - bit$: بیتی که برای هر بلاک از cache وجود دارد و در صورتی که برابر با ۱ باشد، نشان میدهد که در بلاک مورد نظر، دیتای valid قرار دارد. به عبارتی:

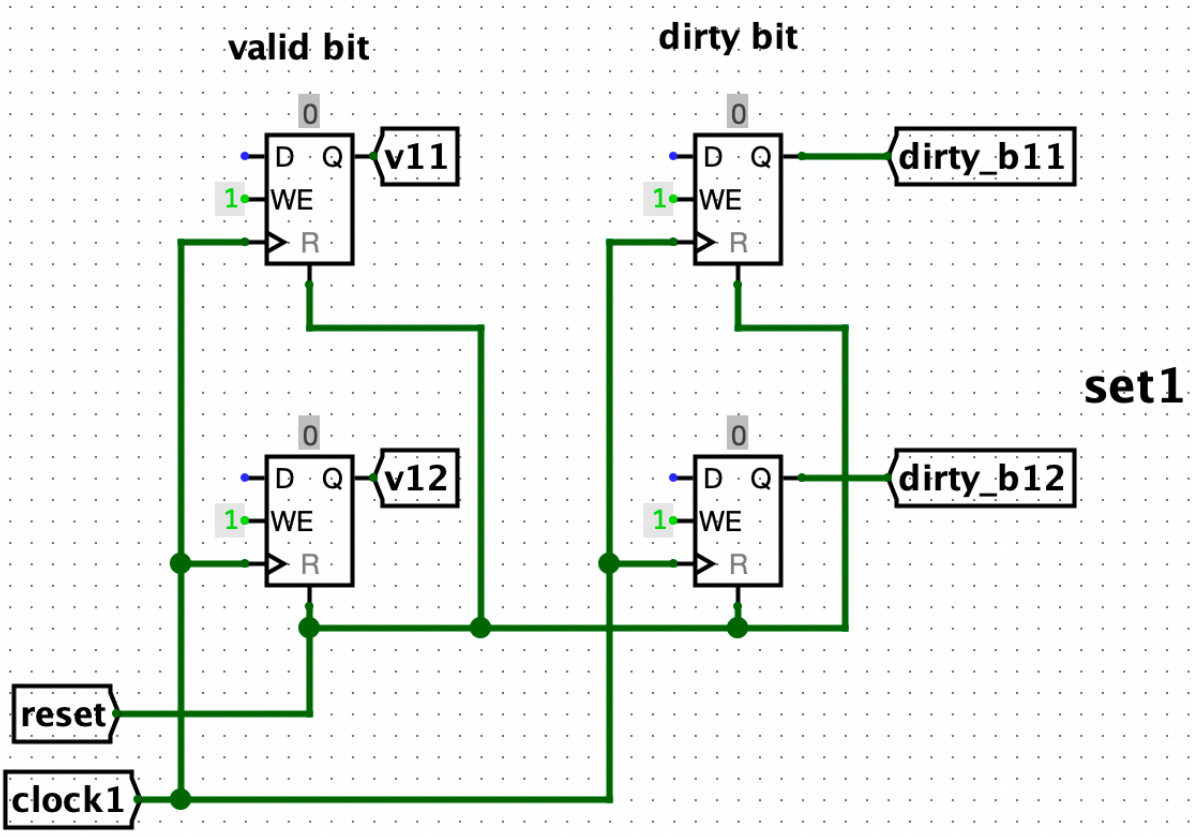
$$valid - bit = \begin{cases} ۱ & \text{block has valid data} \\ ۰ & \text{otherwise} \end{cases}$$

توجه کنید که:

- در ابتدا، چون cache خالی است، تمامی $valid - bit$ ها برابر ۰ هستند.
- با نوشته شدن مقداری در بلاک $B_i$ ، مقدار $valid - bit$ نظیر آن برابر با ۱ میشود.

حال به ادامه این مرحله میپردازیم. در این بخش، میخواهیم ابتدا $valid - bit$ ها و $dirty - bit$ هارا تعریف کنیم و در مرحله بعد، نحوه عوض شدن مقدار آنها را پیاده سازی کنیم. برای هر کدام از این بیت ها، به یک رجیستر جهت ذخیره سازی نیاز داریم. همچنین در کل، ۴ set داریم که هرکدام شامل ۲ بلاک هستند. در نتیجه در کل ۸ بلاک داریم و به موازات آن، ۸ $valid - bit$ و ۸ $dirty - bit$.

در نتیجه برای ساخت این بیت های کنترلی، میتوانیم مانند شکل ۳-۱۶ مدارمان را ببندیم:




**شکل ۳-۱۴: فرم ذخیره سازی** $valid-bit$ **و** $dirty-bit$

در این شکل، بیت های $v$۱۱ و $v$۱۲ بیانگر $valid-bit$ های set اول(در واقع هر یک برای یکی از بلاک های این set) و بیت های $dirty-b$۱۱ و $dirty-b$۱۲ بیانگر $dirty-bit$ های set اول(هر یک برای یک بلاک) هستند. همانطور که در شکل ۳-۱۶ میتوان مشاهده کرد، هنوز مقدار ورودی این ثبات هارا مشخص نکرده ایم و در این مرحله، صرفا نحوه ذخیره سازی آنرا پیاده کرده ایم. در نهایت نیز بیت های clock و reset نظیر به هر یک از این ثبات ها وارد می شوند.

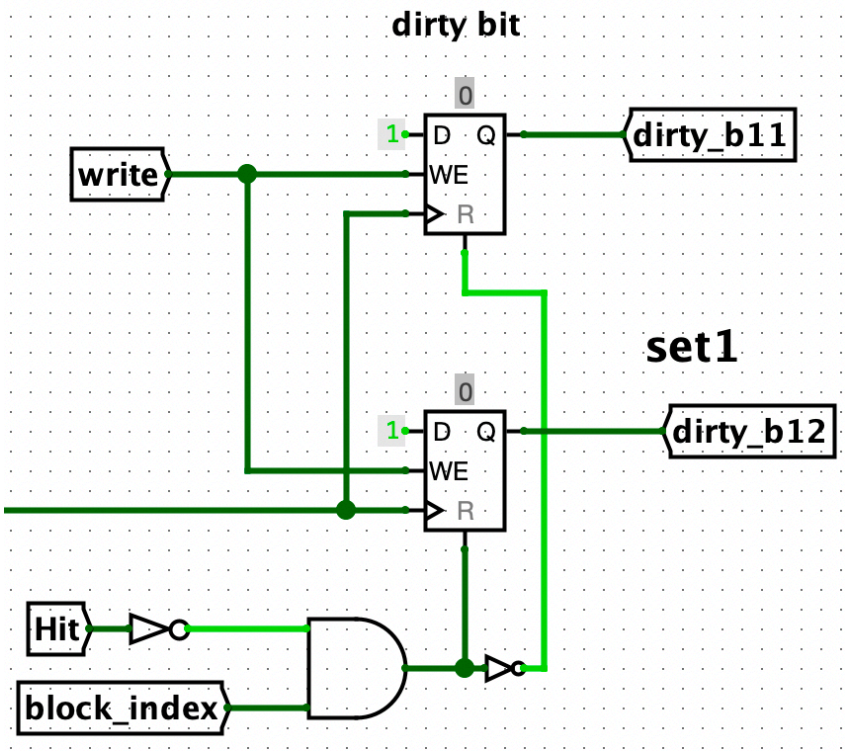
**مرحله سوم: ست کردن بیت های کنترلی**

در دو مرحله، این بخش را تکمیل میکنیم:

- $dirty - bit$:

در این بخش، $dirty - bit$ هارا بررسی میکنیم. همان طور که در مرحله قبل توضیح داده شد، $dirty - bit$ برای مشخص کردن وضعیت مورد نیاز برای بازنویسی داده در مموری است. در صورتی که داده ای در cahce آپدیت میشود، نیاز است که در مموری نیز بازنویسی شود، برای همین $dirty - bit$ را در این بخش برابر با ۱ قرار میدهیم. همچنین در صورتی که miss رخ دهد، باید مقدار $dirty - bit$ را برابر با ۰ قرار دهیم. در نتیجه، سیگنال های کنترلی ثبات های ذخیره کننده این دو بیت برای هر set را(که در مرحله قبل بیان کرده بودیم) به شکل زیر قرار میدهیم:

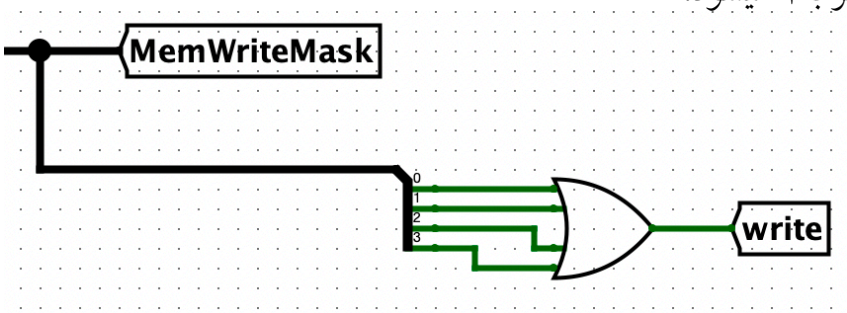
نیاز است در صورتی که miss رخ میدهد، $dirty - bit$ برابر با ۰ شود. در نتیجه، سیگنال miss را به ورودی reset این ثبات میدهیم تا در صورت ۱ شدن آن(معادل با miss )، ثبات ریست شده و مقدار $dirty - bit$ برابر با ۰ شود.




**شکل ۳-۱۵: ورودی های ثبات** $dirty - bit$

همچنین نیاز است تا در صورتی که مقداری در cache آپدیت شد، مقدار $dirty - bit$ برابر با ۱ شود. برای این کار، از ورودی $MemWriteMask$ کمک میگیریم. این ورودی مشخص

میکرد که کدام بایت های کلمه مورد نظر از بلاک و set مورد نظر نیاز به آپدیت دارند. در نتیجه، در صورتی که حداقل یکی از بیت های این ورودی برابر با ۱ باشد، یعنی قرار است که آپدیت رخ دهد. در نتیجه یک تک بیت write درست میکنیم که حاصل or همه بیت های ورودی $MemWriteMask$ باشد. در صورتی که حداقل یک بیت قرار بود آپدیت شود، این بیت برابر با ۱ میشود.




**شکل ۳-۱۶: ساخت بیت write**

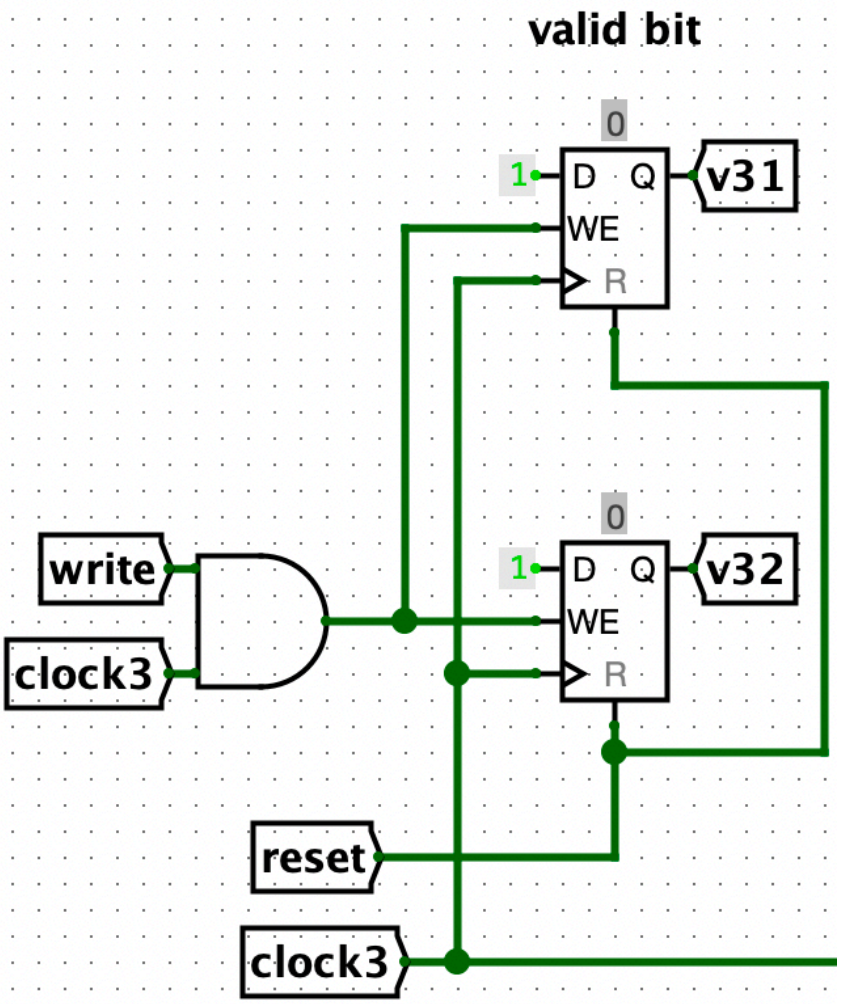
حال این بیت write را به ورودی enable در ثبات های $dirty-bit$ برای set نظیر میدهیم. همچنین ورودی دیتا را برابر با ۱ قرار میدهیم. با این کار، در صورتی که نیاز به آپدیت باشد، $dirty-bit$ برابر با ۱ شده و در صورت رخ دادن miss ، $dirty-bit$ برابر با ۰ میشود.

در انتها نیز، قرار است هنگامی که در یکی از set ها، miss رخ میدهد و همچنین $dirty-bit$ ای برابر با ۱ قرار دارد، در مموری، enable نظیر فعال شده و اجازه write داده شود. برای انجام این کار، یک تک بیت Hit داشتیم که حاصل or هر ۴ سیگنال hit برای set ها بود. در نتیجه مقدار not آن، نشان دهنده حالتی است که miss ای رخ داده است. در کنار این، همه ۸ بیتی که برای $dirty-bit$ ها ساخته شده اند را با یکدیگر or میگیریم. حاصل and این دو، بیانگر سیگنالی که میخواستیم، است. در صورت ۰ بودن این سیگنال، یعنی فقط میخواستیم که مقداری را آپدیت کنیم، اما در صورت یک بودن، باید مقداری را به مموری ارسال کنیم که نشان دهد نیاز به بازنویسی داریم.

- $valid-bit$:

بنابر تعریفی که برای $valid-bit$ ها ارائه کردیم، در صورتی که بلاکی خالی از دیتا باشد، $valid-bit$ باید برابر با ۰ باشد و در صورت پر شدن، برابر با ۱ شود. در نتیجه برای مشخص کردن سیگنال های ورودی این ثبات ها، به این شکل عمل میکنیم که با کلاک اول، $valid-bit$ هر دو ثبات برابر با ۱ می شود. همچنین در صورت فعال شدن reset ، این $valid-bit$ ها باید برابر با ۰ شوند. پس ورودی reset آن ها را به همان بیت reset میدهیم، ورودی کلاک را نیز به Clock میدهیم، ورودی دیتا هر دو ثبات $valid-bit$ هارا برابر با ۱ قرار میدهیم و

در نهایت، ورودی enable برای هر دو ثبات باید برابر با حاصل and با ورودی های clocki و write باشد. در نتیجه، فرم $valid - bit$ های یک set به شکل زیر خواهد بود:




**شکل ۳-۱۷: ساخت $valid - bit$ ها**

**مرحله چهارم: روش** $true - LRU$

ابتدا بیان میکنیم که روش $true - LRU$ چگونه است و بعد، آنرا پیاده سازی میکنیم.

روش $true - LRU$ مشابه تصویر ۳-۱۵ عمل میکند، بدین صورت که برای هر کدام از بلاک های یک set ، یک counter تعریف میکند و در هر مرحله، با توجه به آنها، بیان میکند که چه اتفاقی برای دیتا باید بیفتد. توجه کنید که سه اتفاق بیشتر ممکن نیست رخ دهد:

- رخ دادن hit
- miss شدن و وجود بلاک خالی
- miss شدن و عدم وجود بلاک خالی(در این صورت باید policy انجام دهیم)

حال در روش $true - LRU$ ، بدین صورت عمل میکنیم که counter همه بلاک هارا در ابتدا برابر با ۰ قرار میدهیم. سپس در صورت رخ دادن هر کدام از اتفاق های بالا، عملیات متناظر را انجام میدهیم:

- در صورت رخ دادن hit ، مقدار $C_i$ ای که برای این بلاک hit شده است را برابر ۰ قرار میدهیم و همچنین، تمامی $C_j$ هایی که از این $C_i$ کمتر اکید هستند را به اندازه یک واحد، زیاد میکنیم.
- در صورت رخ دادن miss و وجود بلاک خالی، دیتا مورد نظر را در این بلاک نوشته و همچنین مقدار $C_i$ ای که برای این بلاک miss شده است را برابر ۰ قرار میدهیم. در نهایت نیز، همه شمارنده های دیگر را یک واحد زیاد میکنیم.
- در صورت رخ دادن miss و عدم وجود سطر خالی، میگردیم و سطری که دارای عدد $۲^s - ۱$ است را پیدا میکنیم، دیتا جدید را در آن بازنویسی میکنیم، شمارنده نظیر آن را برابر با ۰ قرار میدهیم و بقیه شمارنده هارا به اندازه یک واحد زیاد میکنیم.

به عنوان مثال، در مثالی که در شکل ۳-۱۵ زدیم، مقدار counter ها مشابه تصویر زیر آپدیت خواهند شد:

مشاهده میشود که به جز مراحل ابتدایی که برای پر کردن سطر های خالی است، در باقی مراحل، یک اولویت بندی بین بلاک ها موجود است که بیان میکند در صورت رخ ندادن hit ، کدام سطر باید جایگزین شود.

| 0 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 |
| 0 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 |
| 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 3 |

**شکل ۳-۱۸: نمونه آپدیت شدن counter ها**

حال در این پردازنده، ما در هر ست، صرفا دو بلاک داریم! برای همین کارمان راحت تر است. به عبارتی، گویا در مثالی که بیان کردیم، به جای ۴ سطر، فقط ۲ سطر خواهیم داشت. برای همین به جز مرحله اول، در باقی مراحل، صرفا یک ۰ و یک ۱ در هر ستون خواهیم داشت. به عنوان مثال اگر میخواستیم همین توالی اعداد در مثال بیان شده را در ساختار پردازنده مورد نظرمان وارد کنیم، جدول نحوه ورودی دیتا ها به شکل زیر در خواهد آمد:

| - | 1 | 1 | 4 | 4 | 1 | 1 | 6 | 6 | 2 | 2 | 3 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | 7 | 7 | 5 | 5 | 7 | 7 | 5 | 5 | 4 | 4 | 2 |

**شکل ۳-۱۹: نحوه وارد شدن دیتا به cache**

همچنین جدول شمارنده ها به شکل زیر در می آید:

| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

**شکل ۳-۲۰: نحوه آپدیت شدن counter ها**

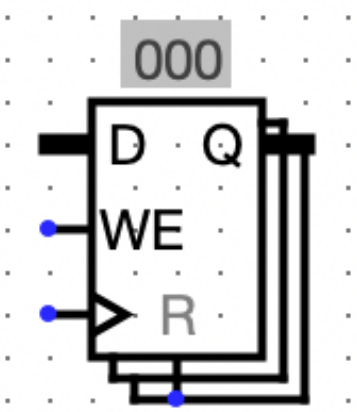
حال به سراغ پیاده سازی این بخش میرویم. همان طور که پیش تر گفتیم، در این روش و مطابق با پردازنده مورد نظر ما، ستون های جدول شمارنده ها، همیشه به دو فرم $\begin{pmatrix}۱\\۰\end{pmatrix}$ و یا $\begin{pmatrix}۰\\۱\end{pmatrix}$ خواهند بود(به جز ستون اول). علت نیز آن است که با بررسی هر حالت (در اینجا فرضا $\begin{pmatrix}۱\\۰\end{pmatrix}$ ) به ازای هر یک از ۳ اتفاق ممکن، حتما به حالتی میان هر دو این بردار ها خواهیم رسید. توجه کنید که اتفاق دوم، فقط در صورتی رخ خواهد داد که سطری خالی داشته باشیم، یعنی در مرحله ۱ الی ۳ که معلوم است که اتفاقی خواهد افتاد.

در نتیجه، برای پیاده سازی این بخش، یک تک بیت برای مشخص کردن بلاکی که مقدار counter آن برابر با ۱ است، در نظر میگیریم. این بیت، همان LRU ای است که در ابتدا تعریف کرده بودیم و به نوعی با مقدار خودش، مقدار counter برای هریک از counter هارا تعیین میکند:

$$LRU = \begin{cases} ۱ & \text{if counter of first block is } ۱ \\ ۰ & \text{otherwise} \end{cases}$$

حال به سراغ بررسی رخدادن miss و یا hit با توجه به مقداری که در حال حاضر، LRU مان دارد، میرویم:

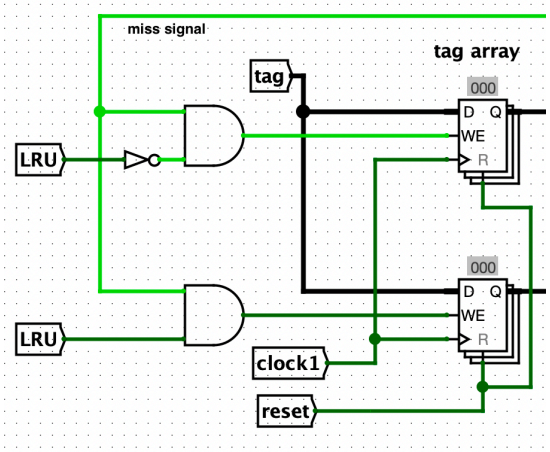
توجه کنید که هر set ، قرار است که یک LRU جداگانه داشته باشد که مقدار آن هارا با $LRU - ۱$ الی $LRU - ۴$ نمایش میدهیم. این بیت ها در مدار دیگرمان، که cacheController نام دارد و جلو تر آن را بررسی میکنیم، قرار دارند، ولی تضمین میشود که مقدار حال حاضر LRU در مدار cache ، مقدار LRU مورد نیاز برای set مورد نظر با توجه به آدرس CPU است. ابتدا ما به دو ثبات برای ذخیره سازی tag مورد نظر در هر set نیاز داریم. هر کدام از این ثبات ها، ساختاری مشابه شکل زیر دارند:




**شکل ۳-۲۱: یک ثبات ساده ۱۰ بیتی**

این ثبات دارای ورودی های متعارف است، یعنی ورودی R برای ریست کردن، ورودی CLK برای ورودی پالس ساعت، ورودی WE برای enable ، ورودی D برای دیتای ۱۰ بیتی ورودی و در نهایت خروجی Q ، برابر با خروجی است. با پالس ساعت، مقدار D با یک بودن enable ، در Q نمایش داده شده و در ثبات ذخیره میشود.

حال برای ورودی D در این ثبات، ما همان مقدار tag که در $CPU - Address$ به دست آورده بودیم را قرار میدهیم. برای ورودی R و CLK نیز معلوم است چه بیت هایی نیاز هستند. برای ورودی enable ، قرار است هنگامی که miss رخ داده است و همچنین ورودی LRU برای این ثبات فعال است، ورودی enable متناظر در ثبات فعال شود. پس مطابق آنچه در شکل زیر مشاهده میکنیم، ورودی enable ها را با استفاده از دو گیت and میسازیم:



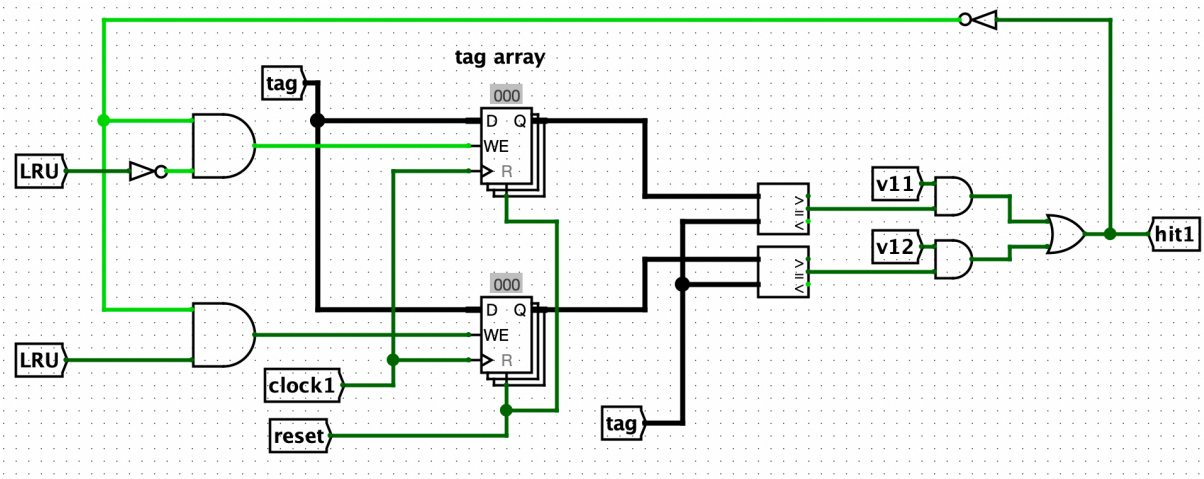

**شکل ۳-۲۲: ورودی های enable در miss**

پس تا به اینجا، بیان کردیم که در چه صورتی باید مقدار tag آپدیت شود و tag جدید در $CPU - Address$ ذخیره شود.

حال در ادامه، باید بررسی کنیم که در چه صورتی miss و یا hit رخ خواهد داد. پاسخ واضح است. هنگامی که tag حال حاضر در یکی از بلاک ها از قبل ذخیره شده باشد، یعنی مقدار آن از قبل در set مورد نظر از cache ذخیره شده بوده و hit رخ میدهد. بنابراین، کافیست مقدار tag های ذخیره شده را با tag جدید، به دو comparator مختلف بدهیم و خروجی مساوی آنها را در نظر بگیریم. در هر بلاک، در صورتی که ۱) مقدار خروجی تساوی برابر با ۱ و ۲) مقدار $valid - bit$ نظیر برابر با ۱ باشد، میتوانیم بگوییم که hit رخ داده است. پس خروجی تساوی این comparator هارا با $valid - bit$ نظیر، and میگیریم و رد نهایت، خروجی این دو را با یکدیگر or میگیریم. این خروجی، همان خروجی hit برای set با شماره i است و آنرا با hiti نشان میدهیم که i ، اندیس خروجی است. حال برای مشخص شدن رخ دادن miss ، صرفا کافیست که hit رخ نداده باشد. پس خروجی hiti را not میکنیم و آن را برابر با مقدار missi قرار میدهیم. پیش تر برای مشخص کردن enable ثبات ها، نیاز به miss داشتیم و آن را از این خروجی به دست می آوریم. در نتیجه مدار بررسی رخ دادن hit و یا miss ما کامل شده است. شکل زیر، کامل شده این مدار را نمایش میدهد:

این کار را برای هر چهار set مان انجام میدهیم و خروجی هارا مطابق مطالب گفته شده، به دست می آوریم. نحوه آپدیت شدن سیگنال LRU هر یک از set هارا در بخش بعدی بررسی میکنیم.

در انتها نیز میخواهیم مشخص کنیم که در مرحله بعدی، مقدار LRU باید چه وضعیتی داشته باشد. توجه کنید که میخواهیم بیان کنیم که این مقدار صرفا قرار است چگونه تغییر کند و بخش اصلی توضیح آن در بخش بعدی قرار دارد. به جدول زیر توجه کنید:

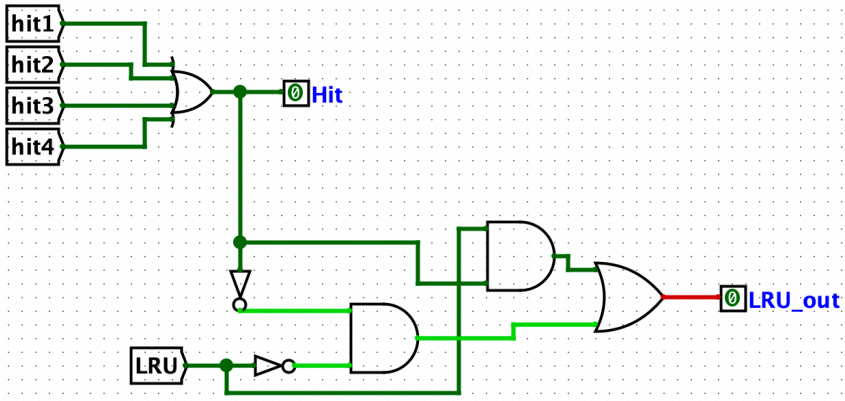



**شکل ۳-۲۳: مدار بررسی miss و hit**

| miss | Current LRU | Next LRU | توضیحات |
|---|---|---|---|
| 0 | 0 | 0 | هنگام رخ دادن hit، مقدار LRU دست نخورده باقی میماند. |
| 0 | 1 | 1 | |
| 1 | 0 | 1 | هنگام رخ دادن miss، مقدار LRU عوض میشود. |
| 1 | 1 | 0 | |

**شکل ۳-۲۴: جدول حالت LRU**

برای پیاده سازی این مدار نیز مشابه با تصویر پایین عمل میکنیم. یک سیگنال hit تعریف میکنیم که بیان میکند در وضعیت حال حاضر، فارغ از این که در کدام set بودیم، آیا hit رخداده است یا خیر. سپس با استفاده از دو گیت and و یک گیت or ، مشخص میکنیم که مقدار LRU در مرحله بعدی(که آنرا با $LRU - out$ نمایش میدهیم) برابر با چه جیزی خواهد بود:

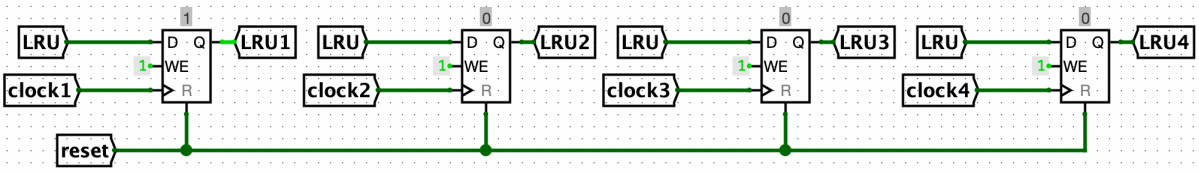



**شکل ۳-۲۵:** $next - LRU$

خروجی این مدار مطابق با جدول بیان شده خواهد بود. در بخش بعدی، بیان میکنیم که این خروجی به چه کاری خواهد آمد.

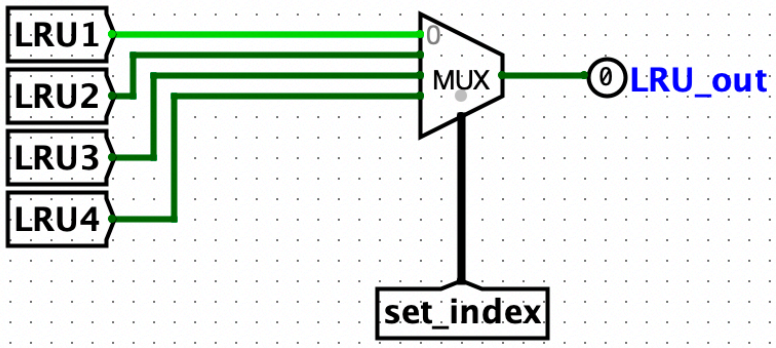
### ۴-۳-۳ *cacheController.circ*

در این بخش، میخواهیم نحوه آپدیت شدن LRU برای مدار cache را بررسی کنیم. توجه کنید که ورودی های این مدار، مشابه با مدار cache است و همچنین بیت های $set - i$ و $clock - i$ به شکل مشابه تعریف شده اند و آن هارا بررسی نمیکنیم. قسمتی که در این مدار جدید است، بخش آپدیت شدن LRU برای هر set است. چهار بیت برای هر یک از set ها تعریف میکنیم و آن هارا به ترتیب با $LRU - ۱$ الی $LRU - ۴$ نمایش میدهیم. مقدار هر کدام از این بیت ها در یک ثبات تک بیتی ذخیره شده اند. ورودی دیتا این ثبات ها، همان بیت LRU اصلی و ورودی Clock این ثبات ها، همان ورودی clock تعریف شده نظیر است:




**شکل ۳-۲۶: آپدیت کردن LRU نظیر set**

با ساخت این مدار، مقدار LRU های نظیر هر set ، آپدیت میشوند. حال در مرحله آخر، با استفاده از مدار زیر، مقدار LRU آپدیت شده را در خروجی ذخیره میکنیم:




**شکل ۳-۲۷: انتخاب LRU**

مراحل ساخت مدارمان به پایان رسید. فقط توجه کنید که در این مدار، مقدار خروجی $LRU - out$ در واقع به ورودی $LRU$ در مدار cache داده شده است و همچنین مقدار خروجی $LRU - out$ در مدار cache ، در واقع به ورودی $LRU$ در مدار cacheController داده شده است. با این کار، در مدار cacheController ، بیان میکنیم که کدام LRU نیاز است که آپدیت شود و سپس مقدار آپدیت شده را به مدار cache ، پاس میدهیم. همچنین در مدار cache نیز، مقدار بعدی LRU را به دست می آوریم و آنرا به کنترلر پاس میدهیم تا مقدار LRU را آپدیت کند. این چرخه دائم تکرار میشود.

## ۳-۴ جمع بندی

در این فاز، با ساخت یک حافظه نهان دارای چهار set و هر set شامل دو بلاک، پردازنده مان را کامل تر کردیم. فرآیند کلی در cache به این صورت است که ابتدا با s بیت، مشخص میکنیم که در کدام set قرار است با cache کار کنیم، سپس با استفاده از tag ، بلاک مورد نظر را در نظر میگیریم و رخ دادن miss و یا hit ، replacement ها، تغییرات $valid-bit$ ها و $dirty-bit$ ها و موارد دیگر را مورد بررسی قرار میدهیم و در نهایت، cache مان را آپدیت میکنیم.

# فصل ۴

# منابع