



دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

مهندسی کامپیوتر

گزارش پروژه معماری کامپیوتر

نگارش

امیر محمد درخشنده، فربد حاجی محمدعلی، آرش ضیایی

استاد درس

استاد سربانی 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 - طراحی <i>ALU</i>	۲
۲ هدف فاز	۱-۲
۴ CPU و طراحی Datapath	۲-۲
۴ instruction spliter	۱-۲-۲
۵ Control Unit	۳-۲
۷ ALU	۴-۲
۱۲	فاز دوم - طراحی <i>cache</i>	۳
۱۲ هدف فاز	۱-۳
۱۴ چیست؟ <i>2-way-set-associative</i>	۲-۳
۱۶ فایل های اضافه شده	۳-۳
۱۷ <i>set.circ</i>	۱-۳-۳
۱۹ <i>block.circ</i>	۲-۳-۳
۲۱ <i>cache.circ</i>	۳-۳-۳
۳۷ <i>cacheController.circ</i>	۴-۳-۳
۳۸ جمع بندی	۴-۳

۴ فاز سوم - طراحی *pipeLine*

۳۹	۱-۴ هدف فاز
۴۰	۲-۴ <i>pipeline</i> چیست؟
۴۱	۳-۴ طراحی <i>pipeline</i>
۴۱	۱-۳-۴ تقسیم بندی منطقی
۴۲	۲-۳-۴ پیاده سازی
۴۵	۴-۴ بررسی مخاطرات ممکن
۴۵	۵-۴ مخاطره چیست؟
۴۵	۶-۴ مخاطرات ممکن
۴۶	۷-۴ Hazard
۴۶	۱-۷-۴ Structural Hazard
۴۸	۲-۷-۴ Data Hazard
۵۱	۳-۷-۴ Control Hazard
۵۵	۸-۴ رفع مخاطرات

فصل ۱

موضوع پروژه

در این پروژه، قرار است به پیاده سازی یک پردازنده میانی 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، ماثول 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 مدارمان اضافه خواهیم کرد.

فصل ۲

فاز اول - طراحی ALU

۱-۲ هدف فاز

در فاز اول پروژه، میخواهیم یک جامعی از پردازنده مان طراحی کنیم. برای این منظور، مارژول 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 متفاوت هستند که در ادامه بیان شان میکنیم:

- فرمت R

- فرمت I

- فرمت J

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

Opcode	'rs	'rt	'rd	Sh.Amount	Func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

شکل ۱-۲: 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 خواهد بود:

Opcode	rs	rt	Imm
6 bits	5 bits	5 bits	16 bits

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 خواهد بود:

Opcode	address
6 bits	26 bits

شکل ۳-۲: 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 CPU و طراحی Datapath

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

۱-۲-۲ instruction spliter

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

حال که بخش‌های مختلف دستور ورودی مان را به دست آورده‌ایم، ابتداء نیاز داریم که فرمت دستورمان را برای شروع کار، شناسایی کنیم. (توجه کنید که تا الان، در `instSpliter` ما فقط بلوک‌های ۵ یا ۶ و ... بیتی مختلف را - طبق ۳ حالت کلی ای که داریم - جدا کرده‌ایم و در حال حاضر، نمیدانیم که کدام بلوک‌ها به کار مانند خواهند آمد!) برای شناسایی فرمت دستور، یک مازول Unit Control (UC) یا `CU` 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 متمازیز 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 مموری 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 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 J_r 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 مموری و اضافه کردن آن به رجیستری که در دستور قرار دارد، داریم. مانند lw و lb که باید چیزی را از مموری خوانده و آنرا در رجیستری ذخیره کنیم.

• MemWriteMask: ینابر 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 `1000` خواهد بود. این خروجی بدان معنا است که در مموری، برای مقدار `en` ورودی `4` 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 `4` بیت داریم. پس خروجی را به شکل `1111` تنظیم میکنیم که در مموری، ورودی `en` هر `4` رم فعال باشد که بتوانیم از آنها استفاده کنیم. در دستورات دیگر نیز نیازی به مموری نداریم و خروجی این سیگنال برابر با `0000` 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 `1` 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` را تعیین میکند و در صورتی که `0` 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 مژول 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 مژول ALU مان، باید بخش های مختلف دستور ورودی را، که پیشتر به وسیله instruction، spliter وجز اعداد instruction spliter نیز نیاز خواهیم داشت، مانند PC.

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

در این مژول، قرار است که عملیات های مختلف روی اعداد ورودی مان انجام شود، و در همین مژول 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 مژول دیگری است که آنرا ALU Controller مینامیم و در ادامه معرفی اش میکنیم. اما پیش از آن، سیگنال های ورودی و خروجی مژول 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: پیشتر در Control Unit آنرا ساخته و بررسی کرده بودیم.
- Func: پیشتر در Control Unit آنرا ساخته و بررسی کرده بودیم.
- Imm: عدد immediate ورودی که در دستور اولیه، آن را استخراج کرده ایم.
- Shamt: مقدار Shift را به راست یا چپ را نشان میدهد.
- ALUSrc: پیشتر در Control Unit آنرا ساخته و بررسی کرده بودیم.

• عملیات های منطقی

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

B OR A : این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل $A \text{ OR } B$ را خروجی میدهد و در دیگری، حاصل $B \text{ OR } A$ را خروجی میدهد (در صورتی که دستورمان $I\text{-Format}$ باشد، محتویات رجیستر A با مقدار B Imm باید Or شود).

AND Gate : این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل $A \text{ AND } B$ را خروجی میدهد و در دیگری، حاصل $B \text{ AND } A$ را خروجی میدهد (در صورتی که دستورمان $I\text{-Format}$ باشد، محتویات رجیستر A با مقدار B Imm باید AND شود).

NOR Gate : این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل $A \text{ NOR } B$ را خروجی میدهد و در دیگری، حاصل $B \text{ NOR } A$ را خروجی میدهد (در صورتی که دستورمان $I\text{-Format}$ باشد، محتویات رجیستر A با مقدار B Imm باید NOR شود).

XOR Gate : این گیت دو مرحله در مدارمان ظاهر میشود که در یکی، حاصل $A \text{ XOR } B$ را خروجی میدهد و در دیگری، حاصل $B \text{ XOR } A$ را خروجی میدهد (در صورتی که دستورمان $I\text{-Format}$ باشد، محتویات رجیستر A با مقدار B Imm باید XOR شود).

توجه کنید که \wedge خروجی تولید شده را به فرمت $AfunctionB$ ذخیره میکنیم. به عنوان مثال، مقدار AND شده A و B را به شکل $A\text{-and-}B$ 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 Imm و A –

subi: تفاضل ساده دو عدد Imm و A –

addu: جمع دو عدد unsigned A و B –

subu: تفاضل دو عدد unsigned A و B –

addiu: جمع دو عدد Imm و unsigned A –

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 مقایسه 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، دقیقاً 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 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 استفاده کرده ایم که ۵ بیت اول و بیت ششم مقدار func را از یکدیگر جدا میکند Splitter و مقدار ۵ بیتی را در سلکتور Mux ها قرار میدهد. در نهایت، خروجی در Mux را وارد یک Mux ۱ میکند که سلکتور آن، همان بیت ششم خواهد بود.

علت: دستورات ریاضی و منطقی دارای بیت ششم 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 ۳۲ * ۵ 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.

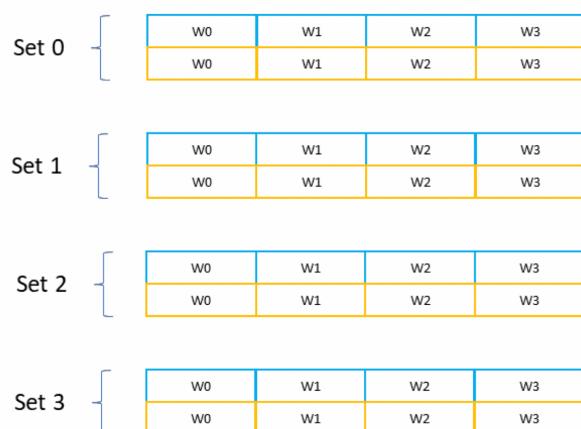
در نهایت نیز، خروجی این Mux و خروجی Mux ۱ * ۲ ای که پیشتر گفتیم، وارد یک Mux ۱ * ۲ دیگر میشوند که سلکتور آن، بیت ALUSrc است که در Control Unit ساخته شده است. خروجی این Mux، خروجی نهایی ۸ بیتی ALU خواهد بود.

فصل ۳

فاز دوم - طراحی cache

۱-۳ هدف فاز

در فاز دوم پروژه، قصد داریم یک cache برای پردازنده مان طراحی کنیم. هدف این فاز، پیاده سازی یک ماژول حافظه‌ی نهان یا همان cache و افزودن آن به پردازنده‌ی فاز قبل می‌باشد. ماژول حافظه‌ای که در این فاز در اختیار داریم، مشابه ماژول حافظه‌ای است که در فاز قبل استفاده کرده بودیم، با این تفاوت که چند کلاک زمان می‌برد تا خروجی را آماده کند. برای ساده شدن طراحی، تعداد کلاک‌های 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 ۱-۳: فرمت بلاک‌های cache

توجه کنید که در نظر داریم که طراحی خود را به گونه‌ای تغییر دهیم که در صورت رخدادن

miss هنگام دسترسی به cache ، پردازنده، به اندازه‌ی تعداد کلاک مشخصی منتظر بماند تا مازول 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 همه‌ی دسترسی‌ها از مسیر cache عبور می‌کند.

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

از تصویر فوق برای طراحی مازول cache استفاده خواهیم کرد. همچنین برای سیاست جایگزینی، یعنی انتخاب way موردنظر، می‌خواهیم روش LRU را پیاده سازی کنیم. با توجه به *way – ۲* بودن هر سط، پیاده سازی True LRU گزینه‌ای منطقی و ساده به حساب می‌آید.

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

در نهایت در نظر داشته باشید که طراحی مازول cache ، فقط برای حافظه‌ی داده یا همان instruction memory انجام می‌گیرد و نیازی به طراحی مازول جداگانه برای خواهد بود و این حافظه مانند قبل، دستورات را در طی یک clock و بدون تأخیر در اختیارمان قرار خواهد داد.

۲-۳ چیست؟ – way – set – associative

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 Word و Block و Tag است که هر یک به ترتیب شامل c ، b و w بیت هستند و روی هم، آدرس مورد نظر را در فرمت شکل ۲-۳ میسازند:

CPU address for M.M.

Tag (block ID)	Block	Word
c bits	b bits	w bits

شکل ۲-۳: 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، $b + c$ بیت برای آدرس بلاک و در نهایت w بیت برای مشخص کردن کلمه مورد نظر خواهیم داشت:

CPU address for M.M.

Tag (block ID)	Word
$b+c$ bits	w bits

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:



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

در این روش، s بیت ابتدای set مورد نظر را مشخص میکنند، سپس $b + c - s$ بیت، آدرس بلاک مورد نظر در این سرترا 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 محتويات فایل circuits ، تعدادی فایل جدید اضافه شده اند که در ادامه، هر یک را جداگانه بررسی ميکنیم:

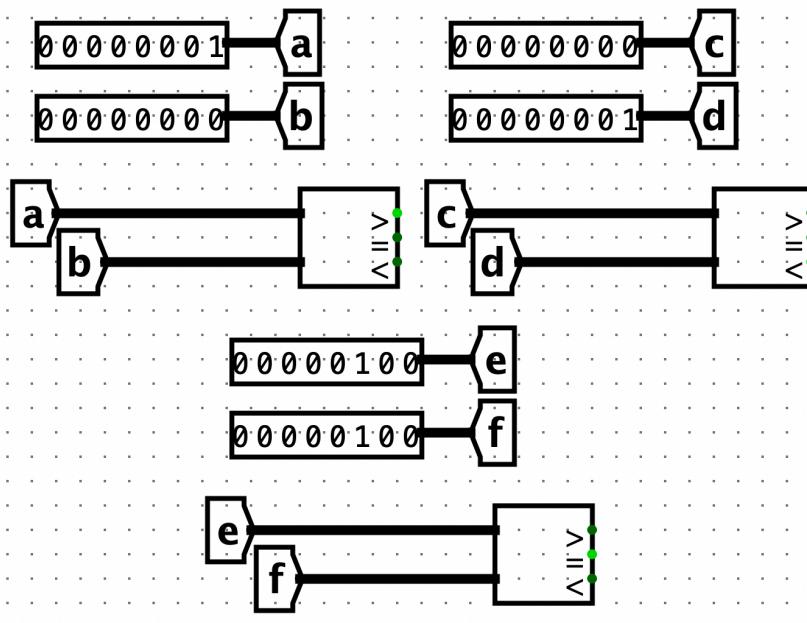
cache.circ •

cacheController.circ •

block.circ •

set.circ •

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



شکل ۳-۵: ماژول comparator

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.

وروودی های مدار:

data-in •

word-index •

block-index •

memWriteMask •

CLOCK •

Reset •

خروجی های مدار:

memReadData •

:*data-in*

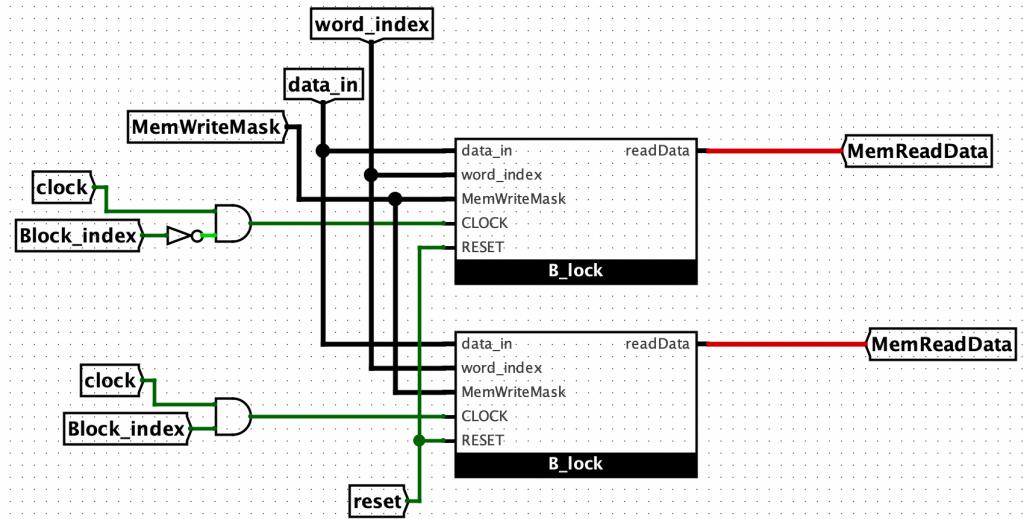
وروودی ۳۲ بیتی برای مشخص کردن داده ورودی به cache برای نوشتن(در صورت نیاز) است. البته داخل مدار، این ورودی به شکل هگز نمایش داده شده است.

:*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 ه باشد، در بلاک اول و در غیر این صورت، در بلاک دوم هستیم.



شکل ۳-۶: مازول set

:memWriteMask

ورودی شامل ۴ بیت برای مشخص کردن بایت متناظر در کلمه مودرنظر 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 ۱۱۰۱ 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.

:reset و CLOCk

پالس ساعت و ریست مدار هستند.

:memReadData

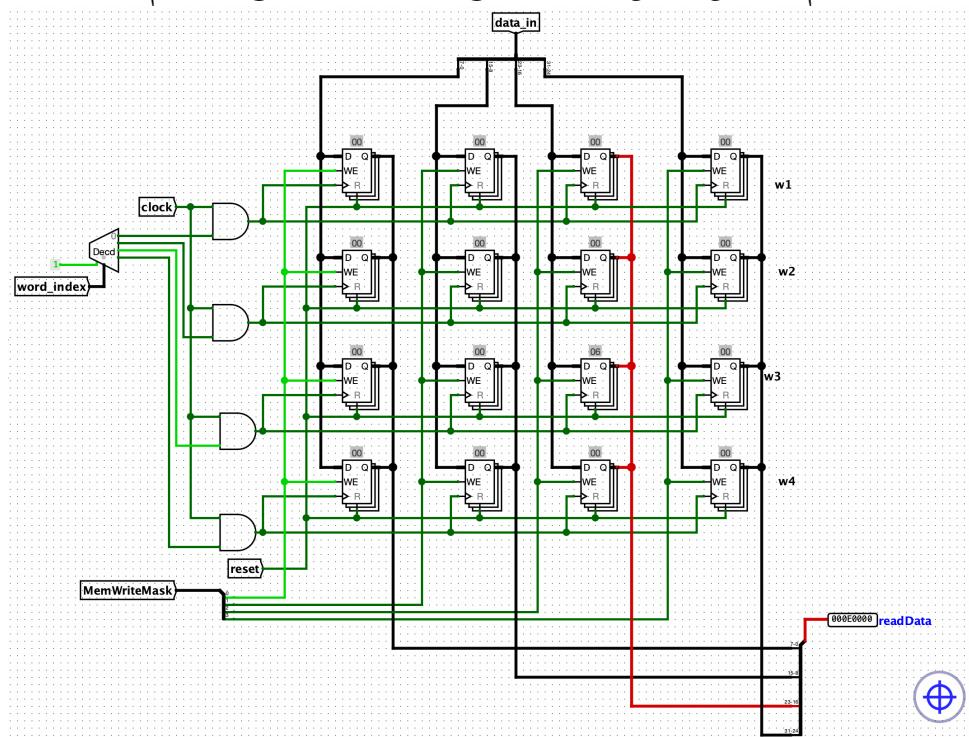
به شکل یک خروجی ۳۲ بیتی در هگز است که در هر پالس ساعت، بسته به این که ورودی کلمه مان چه چیزی است، بایت های ° را به صورت نظیر برابر با ° و بایت های غیر ° را برابر E قرار میدهد. به عنوان مثال، اگر برای کلمه سوم، af2d105° را ذخیره کرده باشیم، در خروجی مقدار EEEEE° E مشاهده خواهیم کرد.



شکل ۷-۳: نمونه برای نمایش خروجی

block.circ ۲-۳-۳

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 ۸-۳: مازول block

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

در ابتدا، توسط یک 4×2 mux، 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.

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 •

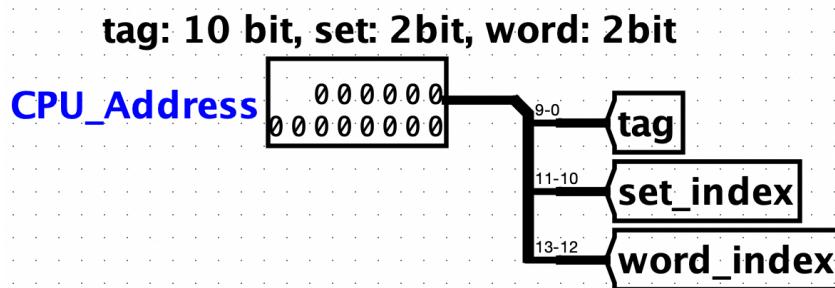
memWriteMask •

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 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 $s + c - b$ را تشکیل میدهد. در نتیجه در آدرسی که از CPU خوانده میشود، بیت 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)

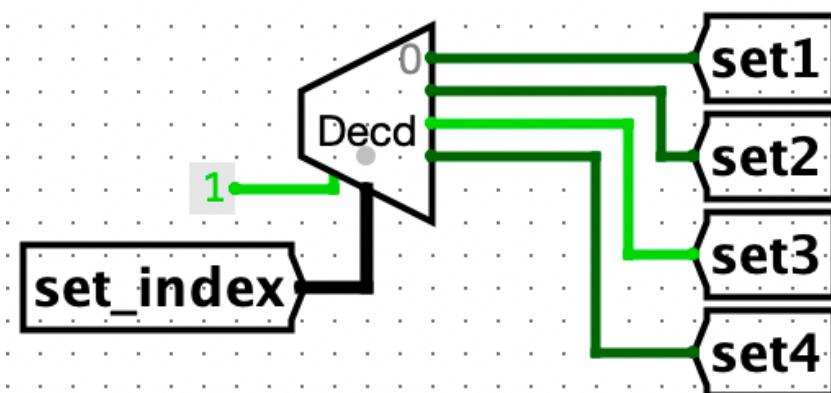
ورودی data-in همان ورودی ۳۲ بیتی یا ۸ بایتی ما خواهد بود.

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 ۴ به ۶، به خطوط خروجی- i set- i تقسیم میکنیم که هر لاینی که فعال باشد، در واقع کلید دسترسی به 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)



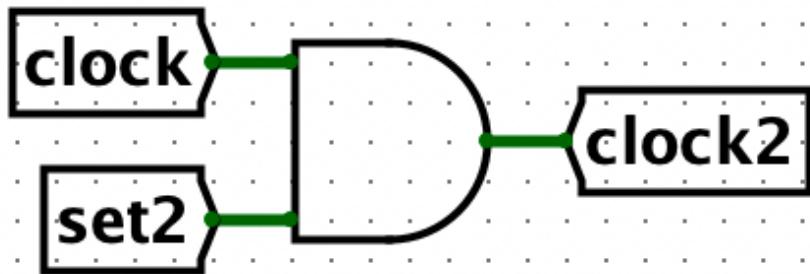
شکل ۱۰-۳: تجزیه set-index توسط دیکودر

در ادامه، چهار ورودی جدید را تعریف میکنیم:

ورودی های $clock - i$ (۰ الی ۴)، هر کدام بیانگر اجازه دسترسی به هر set در هر پالس بالارونده ساعت هستند، بدین صورت که تک بیت i -clock معرفی میشود(برای $i = ۰, ۱, ۲, ۳, ۴$):

$$clock - i = \begin{cases} 1 & \text{if } i\text{'th set has been considered and clock is on pos edge} \\ 0 & \text{otherwise} \end{cases}$$

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

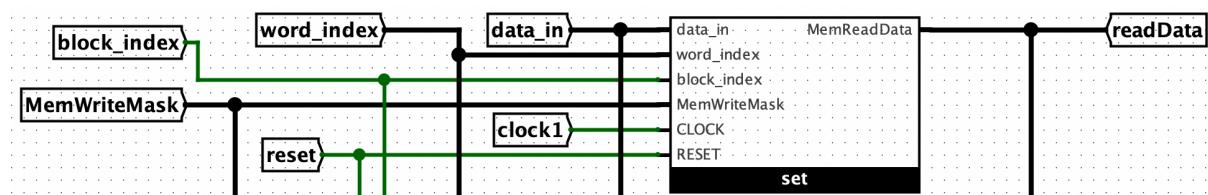


شکل ۱۱-۳: ساخت بیت ۲ - clock

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 کردیم 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 ۱۴-۳، یک نمونه از set هارا برای اول نشان میدهد:



شکل ۱۲-۳: نمونه ای از set ها

توجه کنید که برای هر set ، ورودی clock - i است که set i نظیر است (به عنوان مثال، در شکل ۱۴-۳ clock - ۱، ۱۴-۳ 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 مفهوم 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 cache 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، چایگزین 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:

*not – recently – used :NRU – policy **

*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

۱ - ۷ - ۴ - ۵ - ۱ - ۷ - ۶ - ۵ - ۲ - ۴ - ۳ - ۲

به حافظه نهان را مشاهده کرد:

Ref. Seq.	1	7	4	5	1	7	6	5	2	4	3	2
CACHE BLOCKS	-	1	1	1	1	1	1	1	2	2	2	2
	m	m	m	m	h	h	m	h	m	m	m	h
	-	-	7	7	7	7	7	7	7	4	4	4
	-	-	-	4	4	4	4	6	6	6	3	3
	-	-	-	-	5	5	5	5	5	5	5	5

شکل ۳-۱۳: نمونه ای از *LRU - policy*

- 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:

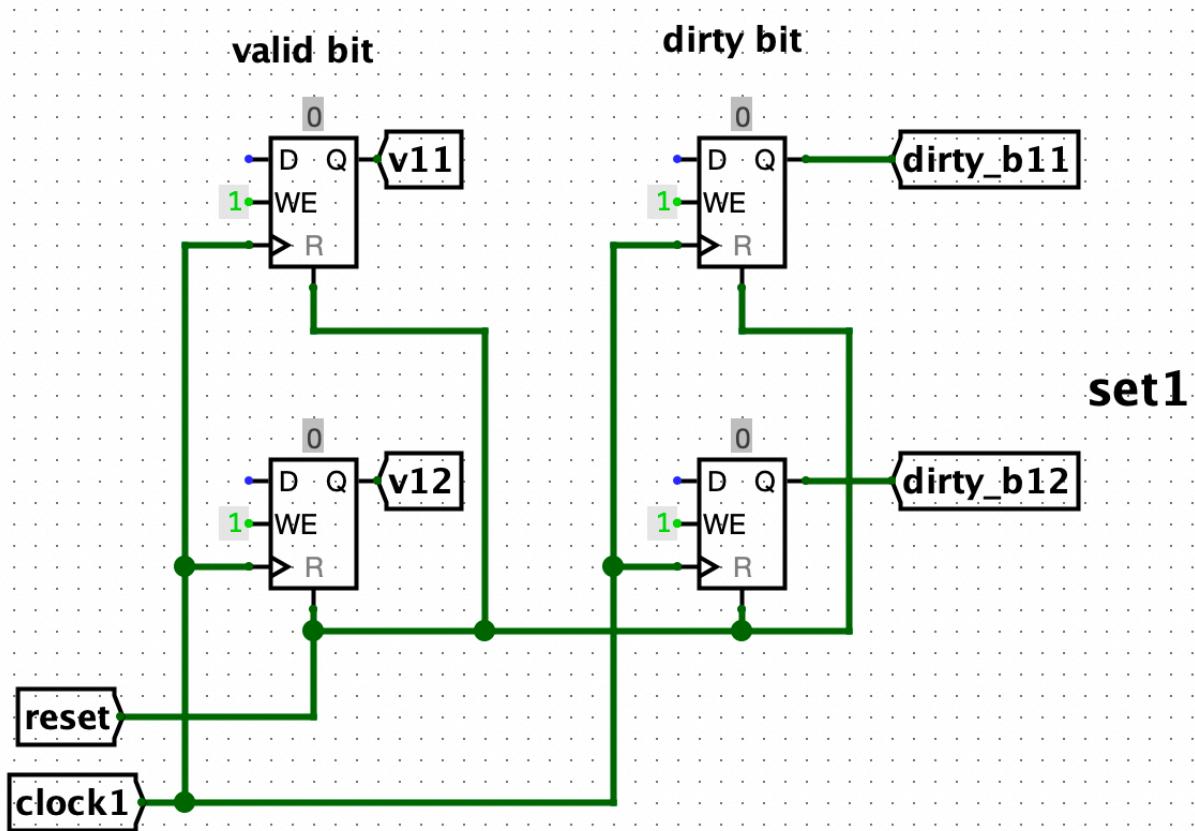
$$valid-bit = \begin{cases} 1 & \text{block has valid data} \\ 0 & \text{otherwise} \end{cases}$$

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 هر کدام شامل ۲ بلاک هستند. در نتیجه در کل ۸ بلاک داریم و به موازات آن، ۸ *valid-bit* و ۸ *dirty-bit* هستند.

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



شکل ۱۶-۳: فرم ذخیره سازی *dirty-bit* و *valid-bit*

در این شکل، بیت های v_{11} و v_{12} بیانگر *valid-bit* های set اول (در واقع هر یک برای یکی از بلاک های این set) و بیت های b_{11} و b_{12} بیانگر *dirty-bit* های set اول (هر یک برای یک بلاک) هستند. همانطور که در شکل ۱۶-۳ 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

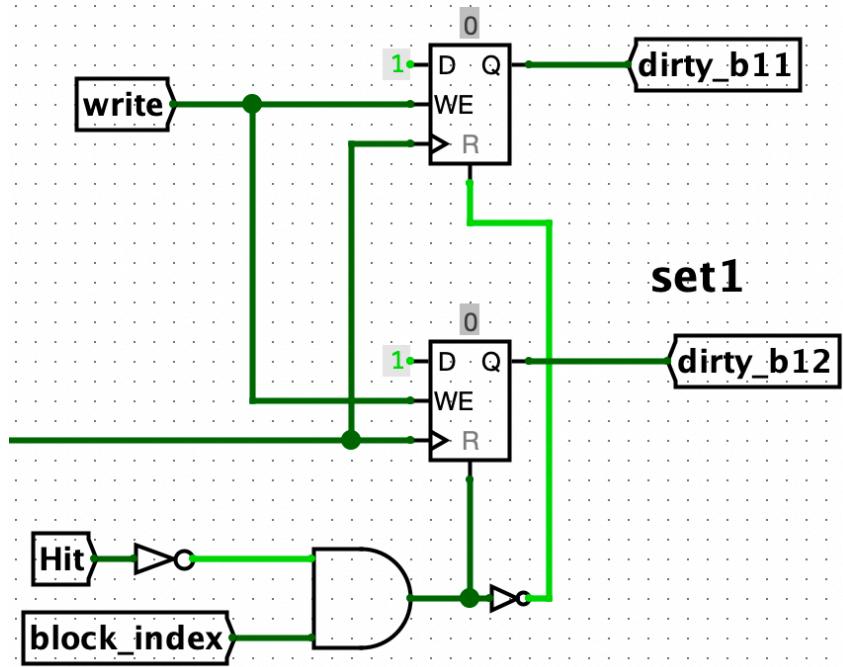
در دو مرحله، این بخش را تکمیل میکنیم:

: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 cache 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.

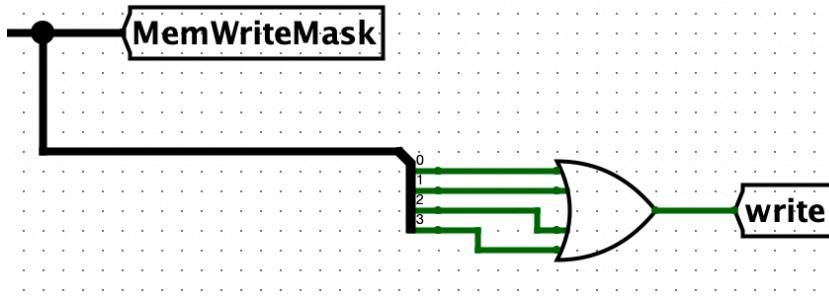
dirty bit



شکل ۱۵-۳: ورودی های ثبات dirty – bit

همچنین نیاز است تا در صورتی که مقداری در cache آپدیت شد، مقدار dirty – bit 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 ۱۶-۳: ساخت بیت write

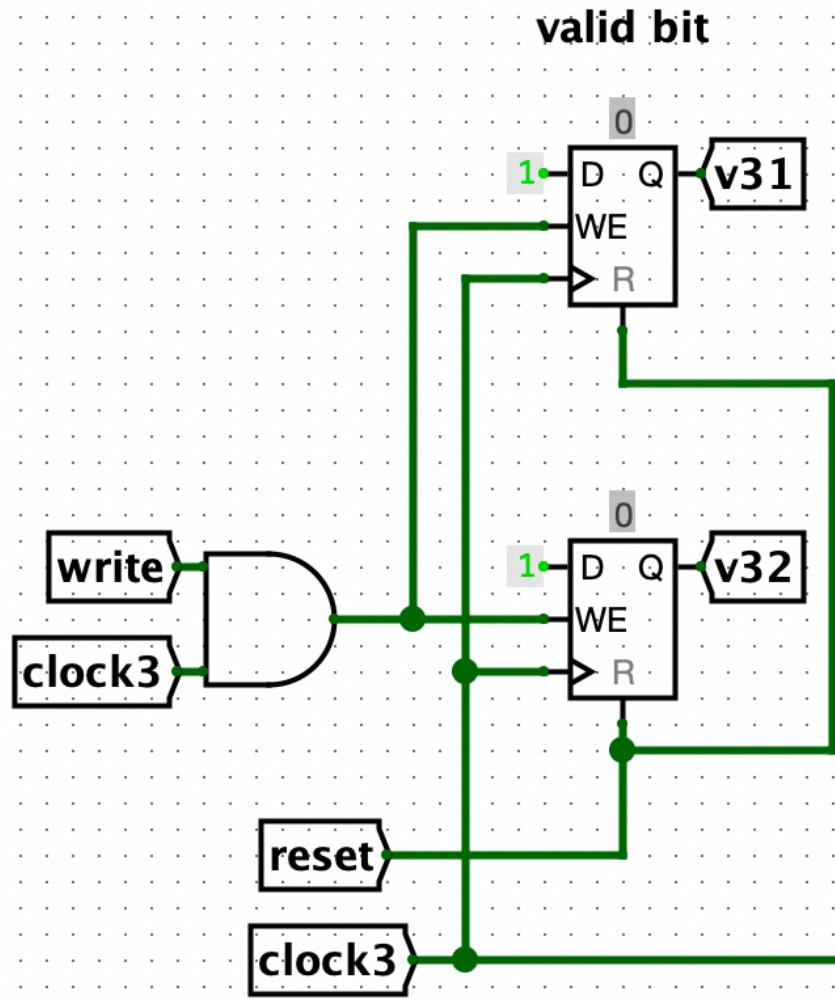
حال این بیت write را به ورودی enable در ثبات های dirty-bit برای set نظیر میدهیم. همچنین ورودی دیتا را برابر با ۱ قرار میدهیم. با این کار، در صورتی که نیاز به آپدیت باشد، dirty-bit برابر با ۱ شده و در صورت رخ دادن miss-bit برابر با ۰ میشود.

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 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 ۱ 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 پرشدن، 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 های *valid-bit* به شکل زیر خواهد بود:



شکل ۱۷-۳: 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 ۱۵-۳ 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 بیفت. 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 سطربی که دارای عدد ۱ – 2^s 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 ۱۵-۳ 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 ۱۸-۳: نمونه آپدیت شدن counter ها

حال در این پردازنده، ما در هر سرت، 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 ۱۹-۳: 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 ۲۰-۳: نحوه آپدیت شدن counter ها

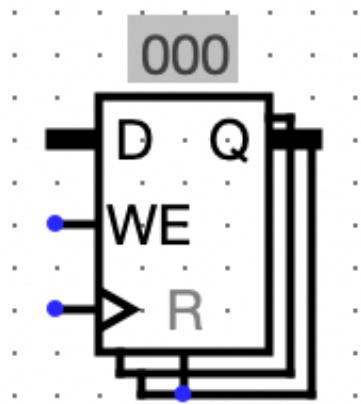
حال به سراغ پیاده سازی این بخش میرویم. همان طور که پیش تر گفتیم، در این روش و مطابق با پردازنده مورد نظر ما، ستون های جدول شمارنده ها، همیشه به دو فرم $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ و یا $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ خواهند بود(به جز ستون اول). علت نیز آن است که با بررسی هر حالت (در اینجا فرضا $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$) 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:

$$LRU = \begin{cases} 1 & \text{if counter of first block is } 1 \\ 0 & \text{otherwise} \end{cases}$$

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

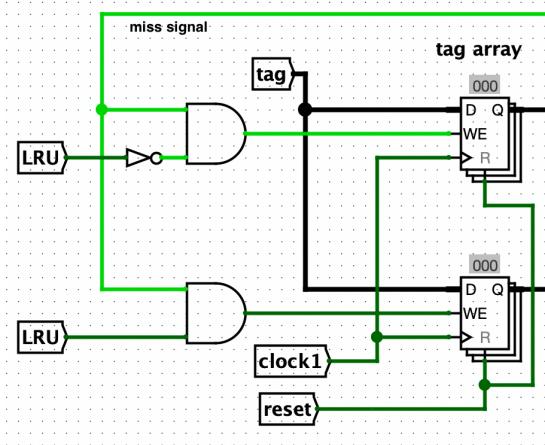
توجه کنید که هر set ، قرار است که یک LRU جداگانه داشته باشد که مقدار آن هارا با $LRU - 1$ نمایش میدهیم. این بیت ها در مدار دیگرمان، که cacheController نام دارد و جلو تر آن را بررسی میکنیم، قرار دارند، ولی تضمین میشود که مقدار حال حاضر LRU در مدار cache ، مقدار LRU مورد نیاز برای set مورد نظر با توجه 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 ۲۱-۳: 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:

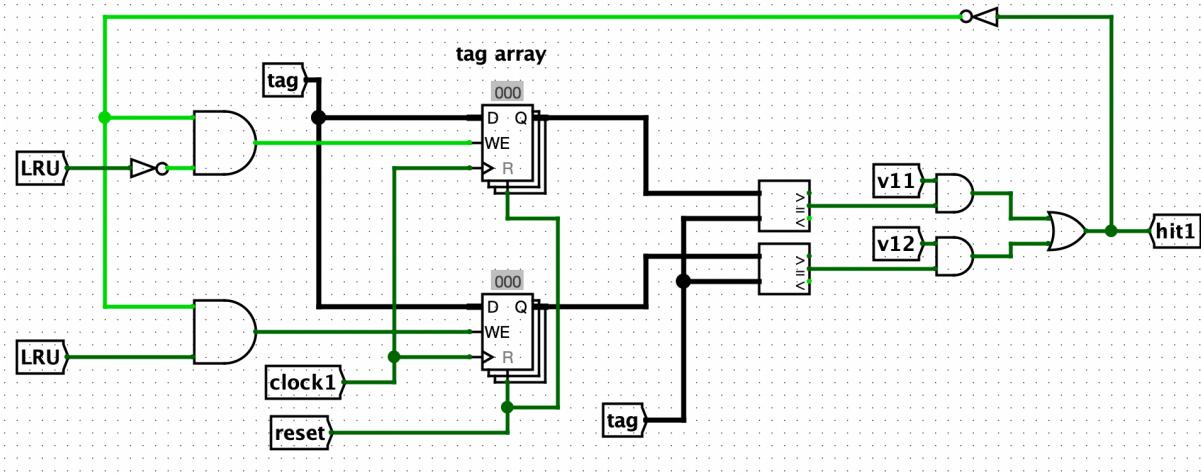


شکل ۲۲-۳: ورودی های miss enable در

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 بلاک‌ها 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 tag‌های ذخیره شده را با tag جدید، به دو comparator مختلف بدھیم 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 hit_i نشان میدهیم که i ، اندیس خروجی است. حال برای مشخص شدن رخ دادن miss، صرفاً کافیست که hit رخ نداده باشد. پس خروجی $not hit_i$ را میکنیم و آن را برابر با مقدار miss _{i} قرار میدهیم. پیش‌تر برای مشخص کردن enable ثبات‌ها، نیاز به miss داشتیم و آن را از این خروجی به دست می‌آوریم. 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 می‌آوریم. 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 صرفاً قرار است چگونه تغییر کند و بخش اصلی توضیح آن در بخش بعدی قرار دارد. به جدول زیر توجه کنید:

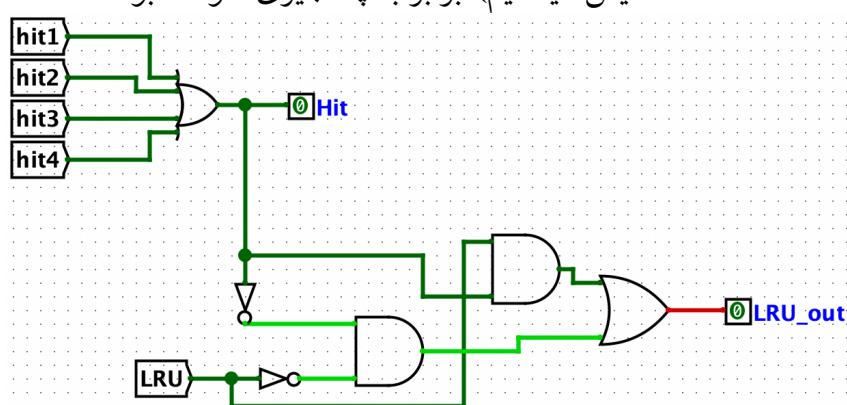


شکل ۲۳-۳: مدار بررسی hit و miss

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 ۲۴-۳: 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:

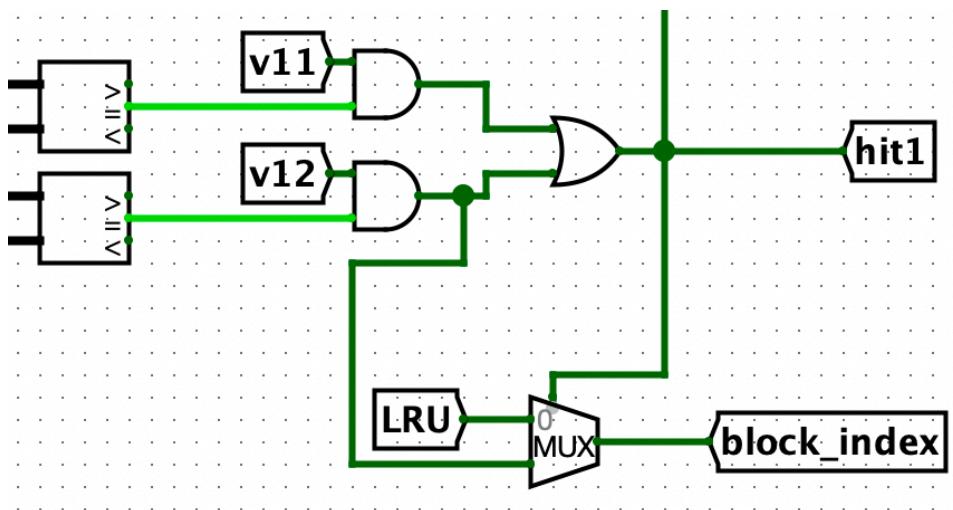


شکل ۲۵-۳: next - LRU

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

در انتهای نیز نیاز است تا سیگنال های کنترلی *dirty-bit* و *valid-bit* ای که در اختیار داشتیم، و با استفاده از سیگنال های دیگری که تا به اینجه تعریف کرده ایم، برای مموری مشخص کنیم که در چه صورتی به *read* و در چه صورتی نیاز به *write* است. برای این کار، در شکل ۲۳-۳، میدانیک که در صورت رخ دادن *miss*، باید روی بلاکی که شمارنده آن در جدول شمارنده ها (تصویر ۲۰-۳) برابر با ۱ است، سیگنال *write* فعال شده و داخل مموری، بیان کنیم که کدام بلاک از *set* قرار است که مورد بازنویسی قرار گیرد. برای این کار، از همان بیت *LRU* میتوانیم استفاده کنیم که در صورت ۰ بودن، بیانگر بلاک اول و در غیر این صورت بیانگر بلاک دوم خواهد بود.

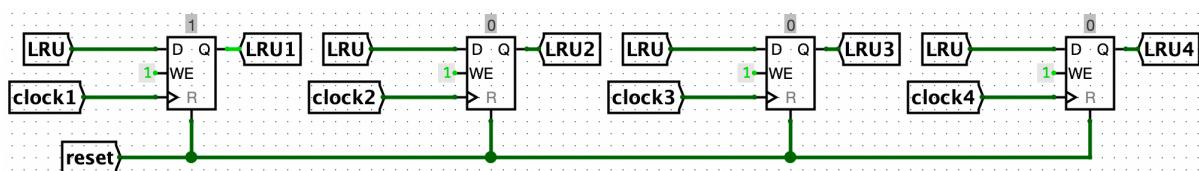
همچنین در صورتی که *miss* رخ نداده و داخل وضعیت *hit* قرار داشته باشیم، در صورتی که *valid-bit* بلاک دوم برابر با ۰ باشد، یعنی این بلاک خالی از دیتا بوده و در بلاک اول، *hit* رخ داده است. در غیر این صورت نیز مهم نیست که مقدار خروجی برابر با چه مقداری است، چرا که در *hit*، با مموری کاری نداریم و دیتا در *cahce* پیدا شده است. بنابر این، این دو خروجی را وارد یک *mux* با ۲ ورودی دیتا و یک ورودی *select* میکنیم که ورودی *select* همان سیگنال *hit* است (در صورت ۰ بودن، در وضعیت *miss* هستیم، در نتیجه ورودی لاین اول دیتا، همان *LRU* و دیگری برابر ورودی دوم برای مشخص سازی *hit* است). در انتهای نیز، خروجی این *Mux*، اندیس بلاکی که قرار است مورد بررسی قرار بگیرد را در اختیار ما می گذارد، یعنی همان تک بیت *block-index*. بنابر این مقدار این بیت را در این مرحله آپدیت میکنیم. در شکل زیر، نحوه ساخت مدار این قسمت را در تکمیل تصویر ۲۵ میتوانید مشاهده کنید:



شکل ۲۶-۳: نحوه آپدیت شدن *block-index*

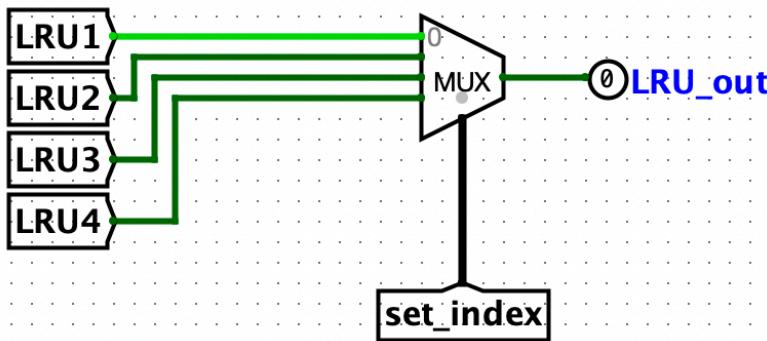
همچنین با به دست آمدن $block - index$ ، در ثبات هایی که وظیفه ذخیره سازی $dirty-bit$ را داشتند، برای ورودی ریست، یک تغییری ایجاد میکنیم که یا در صورت فعال سازی سیگنال ریست اصلی فعال شود، یا در صورتی که وضعیت miss رخ داده و همچنین $block - index$ به کدام بلاک اشاره دارد. حاصل and این دو را با بیت ریست اصلی، or میگیریم و هر یک را به بلاک نظریه میدهیم.

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 i و $i - set$ 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 $1 - LRU$ الی $4 - LRU$ 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 ۲۷-۳: 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 ۲۸-۳: انتخاب LRU

مراحل ساخت مدارمان به پایان رسید. فقط توجه کنید که در این مدار، مقدار خروجی – LRU – out در واقع به ورودی cache در مدار LRU داده شده است و همچنین مقدار خروجی $LRU - out$ در مدار cache، در واقع به ورودی LRU در مدار cacheController 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.

۴-۳ 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 d 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 رخدادن miss و یا hit ، تعییرات replacement ها، $valid-bit$ ها و $dirty-bit$ ها و موارد دیگر را مورد بررسی قرار میدهیم و در نهایت، cache مان را آپدیت میکنیم.

فصل ۴

فاز سوم - طراحی *pipeLine*

۱-۴ هدف فاز

در این فاز، میخواهیم به پردازنده‌ای که تا به الان ساخته ایم، قابلیت اجرای ۵ مرحله‌ای دستورات در فرم *pipe-line* بدهیم. یعنی دستورات باید در قالب ۵ مرحله زیر اجرا شوند:

• *PC*: دریافت دستورات از *instruction – Memory* و جلو بردن *Instruction – fetch*

• ترجمه *opcode* به سیگنال‌های کنترلی و خواندن رجیسترها

• اجرای عملیات‌های *ALU* و یا محاسبه مقصد *jump* و *branch*

• دسترسی به حافظه (در صورت نیاز) *Memory*

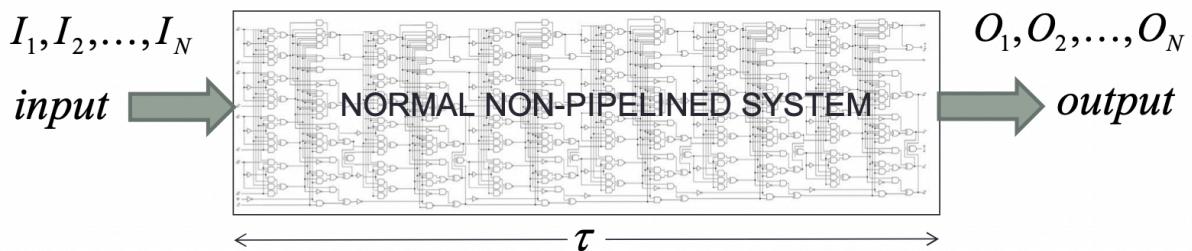
• *RegisterFile* کردن *Update : Write – back*

برای سادگی، مازول *RegisterFile* از قبل در اختیارمان قرار داده شده است و همچنین به صورت *negative – edge – triggered* طراحی شده است. بنابراین با ساختار درونی این مازول، کاری نخواهیم داشت و رجیسترهای هر *state* از *pipeline* میتوانند به صورت *positive-edge-triggered* استفاده شوند.

همچنین هدف دوم از این بخش، بررسی و تحقیق درباره انواع مخاطراتی که ممکن است در طراحی ما، بعد از پیاده‌سازی *pipeline* رخ بدهند، است که به بررسی آنها در انتهای آن، خواهیم پرداخت.

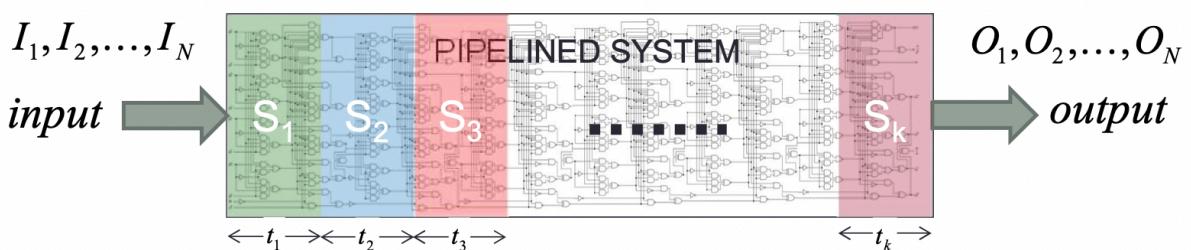
۲-۴ pipeline چیست؟

در بحث فرآیند و اجرای عملیات های پردازنده های گوناگون، همواره زمان اجرای یک دستور و دریافت و اماده سازی خروجی مورد نظر طبق آن، از اهمیت زیادی برخوردار است. به عنوان مثال، در صورتی که پردازنده ای بسازیم که در یک فرآیند مشخص، با دریافت ورودی، خروجی مورد نظر را در یک سری عملیات بسازد، زمان مورد نیاز اجرا برای N دستور ورودی، برابر با $\tau \times N$ خواهد بود که τ بیانگر زمان مورد نیاز برای اجرای یک تک دستور در طول پردازنده است.



شکل ۱-۴: پردازنده ساده بدون pipeline

حال فرض کنید که پردازنده ای داشته باشیم که بخش های مختلف آن، قسمت بندی شده باشند و در هر مرحله، بعد از اجرای کار بخش اول، تا انتهای اجرای عملیات برای دستور اول، این بخش بیکار نماند و کار دستور بعدی را شروع کند. در این صورت، با فرض آنکه k دسته برای پردازنده در نظر گرفته باشیم و همچنین در صورتی که زمان مورد نیاز برای اجرای کار هر بخش برابر با $\frac{\tau}{k}$ باشد (به عبارتی $t_1 = t_2 = \dots = t_k = \frac{\tau}{k}$)، آنگاه زمان مورد نیاز برای اجرای N دستور در این پردازنده برابر با $(N - 1) \times \frac{\tau}{k} + \tau$ خواهد بود که مقداری کوچتر از $\tau \times N$ به ازای k های بیشتر از ۱ است. به این روش تقسیم بندی کردن پردازنده، *pipe-line* گفته میشود و در این بخش تصمیم داریم تا آنرا روی پردازنده مان پیاده سازی کنیم.



شکل ۲-۴: پردازنده با قابلیت pipeline

در پیاده سازی مورد نظر ما، قرار است که k برابر با ۵ باشد و بخش های مختلف آن نیز در بخش ۱-۴ بیان شدند.

۳-۴ طراحی pipeline

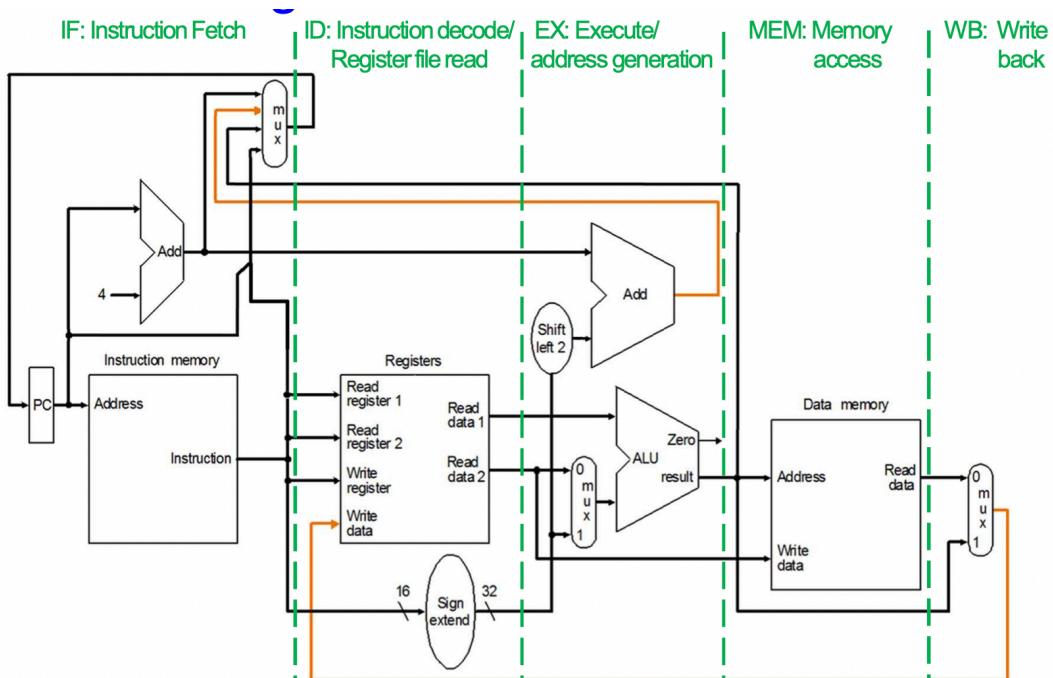
۱-۳-۴ تقسیم بندی منطقی

بخش های مختلف

در این بخش، ساختار کلی یک پردازنده را بررسی خواهیم کرد و در ادامه، سعی میکنیم هر یک از ۵ بخش مورد نظر برای pipeline را پیاده سازی کنیم.

در ابتدا، دستور ورودی از مموری واکشی میشود و داخل instruction – register (IR) ذخیره میشود.

سپس در بخش دوم، دستور مورد نظر شکسته شده و یا به عبارتی، decode میشود و بخش های مختلف آن مانند مقادیر رجیسترها، opcode، سیگنال های کنترلی و ... بدست می آید که همگی داخل ماژولی به نام register – file قرار میگیرند تا به مرحله بعدی بروند. در انتهای این مرحله، بعد از اجرای کامل ماژول این بخش، به مقادیر immediate خوانده شده از رجیستر فایل دسترسی خواهیم داشت و همچنین مقدار PC زیاد میشود.



شکل ۳-۴: تقسیم بندی برای ساختار pipeline

در بخش سوم، یعنی Execute، دستور اجرا میشود. در صورتی که عملیات از نوع عملیات

و یا *arithmetic* و *logical* *load – store* باشد، مقدار خروجی و در صورتی که عملیات از نوع *load – store* باشد، آدرس مورد نظر در خروجی نمایش داده می‌شود. همه این عملیات هارا مازول *ALU* انجام خواهد داد که آنرا در فاز اول، پیاده سازی کرده بودیم.

در مرحله بعدی، در صورتی که عملیات از نوع *load – store* باشد، مقدار مورد نظر از مموری خوانده و یا در آن نوشته می‌شود.

در مرحله اخر نیز، نیاز است تا نتیجه مورد نظر دستور در رجیستری که در آدرس به عنوان رجیستر مقصود (*rd*) بیان شده است، ذخیره شود.

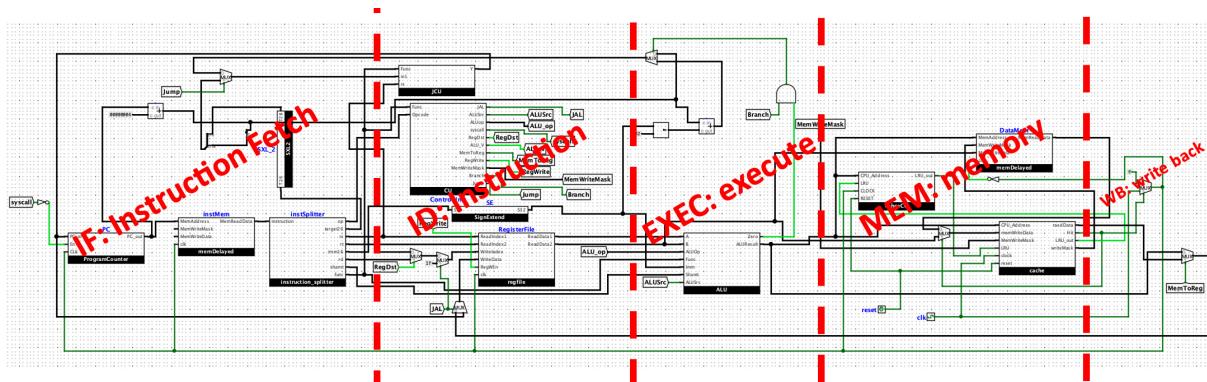
ارتباط بخش‌های مختلف

توجه داشته باشید که بخش‌های مختلفی را که بیان کردیم، یا از قبل در اختیار داشته ایم و یا در فاز‌های گذشته، آنها را ساخته ایم. در این فاز برای *pipeline*، صرفا نیاز است که سیگنال‌های کنترلی میان بخش‌های هارا بررسی کنیم و بیان کنیم که در هر مرحله، کدام بخش یا بخش‌ها از ۵ بخش اصلی، اجازه فعالیت را برای دستورات ورودی دارند.

۲-۳-۴ پیاده سازی

برای پیاده سازی هدفی که بیان کردیم، نیاز داریم که ابتدا، مدار *CPU* حال حاضرمان را مطابق با شکل ۳-۴ تقسیم بندی کنیم و در نهایت، سیگنال‌های بین آنها را با استفاده از تعدادی رجیستر و مهم‌تر از همه، بررسی ورودی‌های *WE* شان، بررسی کنیم.

شکل زیر، تقسیم بندی منطقی *CPU* مشابه با شکل ۳-۴ را نشان میدهد:



شکل ۴-۴: تقسیم بندی برای ساختار *pipeline*

در ادامه، با توجه به جداسازی ای که ارائه دادیم، میخواهیم مرز های بین بخش هارا با استفاده از رجیستر ها، به وسیله ورودی های *data* شان، کنترل کنیم. برای اینکار، با مرز بین بخش اول و دوم، یعنی *IF* و *ID* شروع میکنیم. توجه کنید که سیگنال هایی که از بخش اول بدست آمده و در بخش دوم استفاده میشوند، به شکل زیر هستند(جداسازی قسمت های مختلف دستور ورودی در :

:(splitter

Opcode •

target ۲۶ •

rs •

rt •

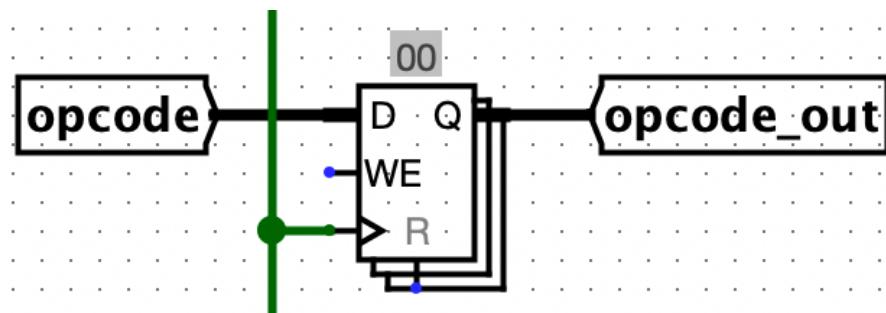
imm ۱۶ •

rd •

shamt •

func •

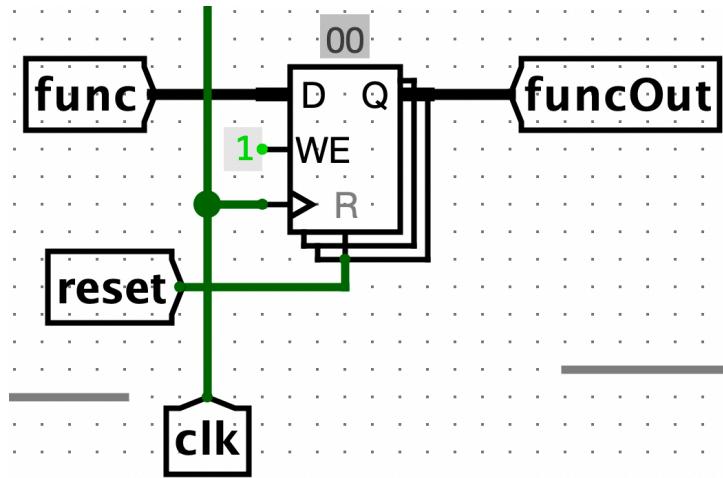
برای هر کدام از این بخش ها، یک رجیستر(با توجه به تعداد بیت های هر یک) در نظر میگیریم و ورودی هارا با همین نام ها(*<name>-out*) و خروجی هارا با اندیس(*<name>*) نمایش میدهیم، به عنوان مثال، در شکل زیر، رجیستر مورد نظر برای *opcode* را میتوان مشاهده کرد:



شکل ۵-۴: رجیستر *opcode*

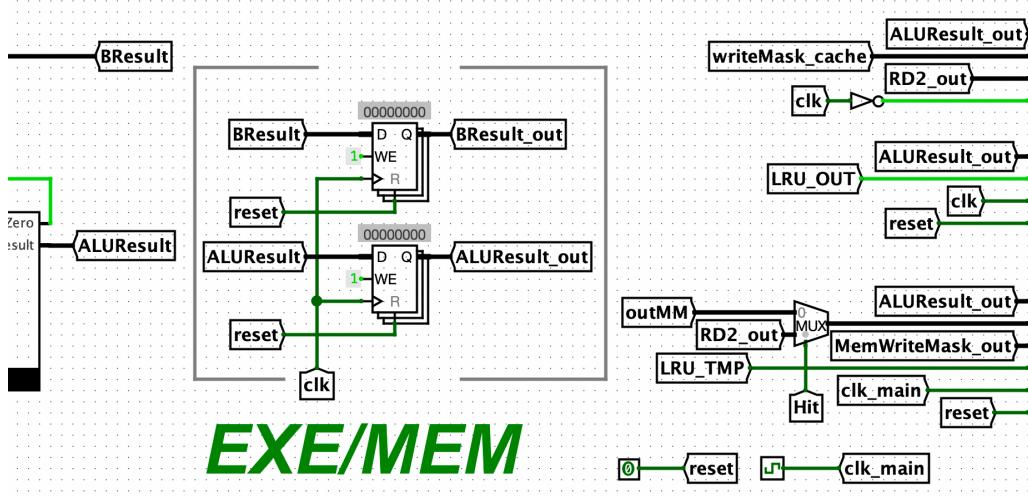
همچنین برای ورودی *WE*، *reset* و *clock* نیز همان ورودی های اصلی متناظر و ورودی ۱ برای *WE* را قرار میدهیم. با این کار، صرفا داریم بیان میکنیم که سیگنال های میانی باید در هر کلاک

آپدیت شوند و مرحله به مرحله جلو بروند. شکل ۴-۶، نمایشی از وضعیت نهایی رجیستر های بیان شده است:



شکل ۴-۶: رجیستر کامل شده

در نتیجه این بخش نیز اینگونه کامل خواهد شد. در ادامه، تصاویر کامل شده پردازنده دارای قابلیت *pipeline* را میتوان مشاهده کرد، بدین صورت که صرفاً قابلیت این را دارد که دستورات ورودی را در هر کلاک، وارد پردازنده کند و مرحله به مرحله آنها را جداگانه و به صورت همرونده، جلو ببرد.



شکل ۷-۴ : EXE - MEM

تصویر ۷-۴، نمایشی از رجیستر های بین بخش ۳ و ۴، یعنی *EXE* و *MEM* را میتوان مشاهده کرد. همانطور که معلوم است، خروجی های سمت چپ رجیستر ها، هریک وارد یک رجیستر شده و در قسمت *MEM*، مورد استفاده قرار گرفته اند.

۴-۴ بررسی مخاطرات ممکن

در این بخش، به بررسی انواع مخاطراتی که ممکن است برای مدارمان رخ دهد و همچنین بررسی راه حل های آنها خواهیم پرداخت:

۵-۴ مخاطره چیست؟

۶-۴ مخاطرات ممکن

Hazard ۷-۴

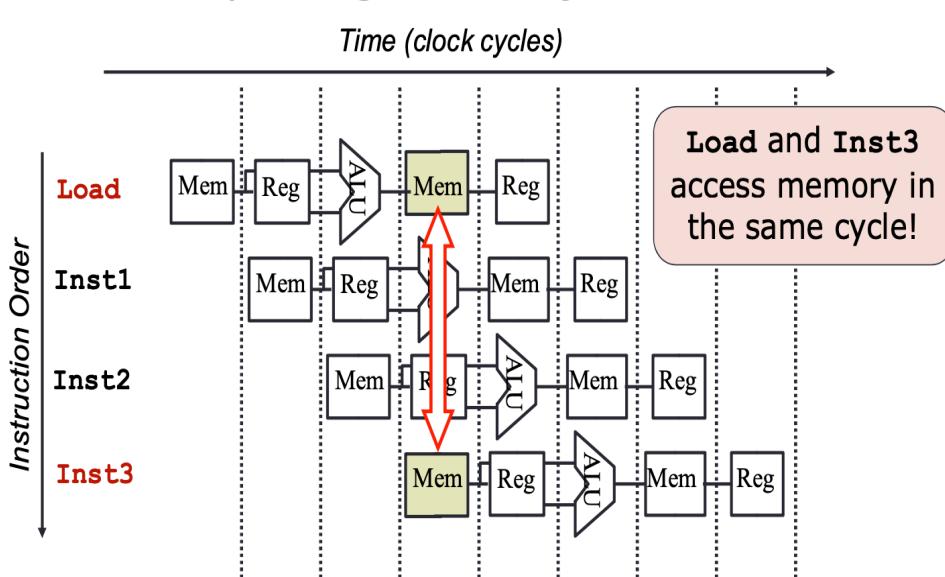
Structural Hazard ۱-۷-۴

مخاطره‌ی structural هنگامی رخ می‌دهد که دو دستور در یک استیج از pipeline بخواهند به یک منبع یکسان از حافظه دسترسی داشته باشند. اما از آنجایی که در ساختار پردازنده‌ی ما حافظه‌ی مخصوص به دستورات و حافظه‌ی اصلی جدا هستند، در نتیجه چنین مخاطره‌ای رخ نمی‌دهد. در صورتی که منبع یکسان بود و مخاطره‌ی structural رخ داد، دو راه حل برای رفع این مخاطره وجود دارد:

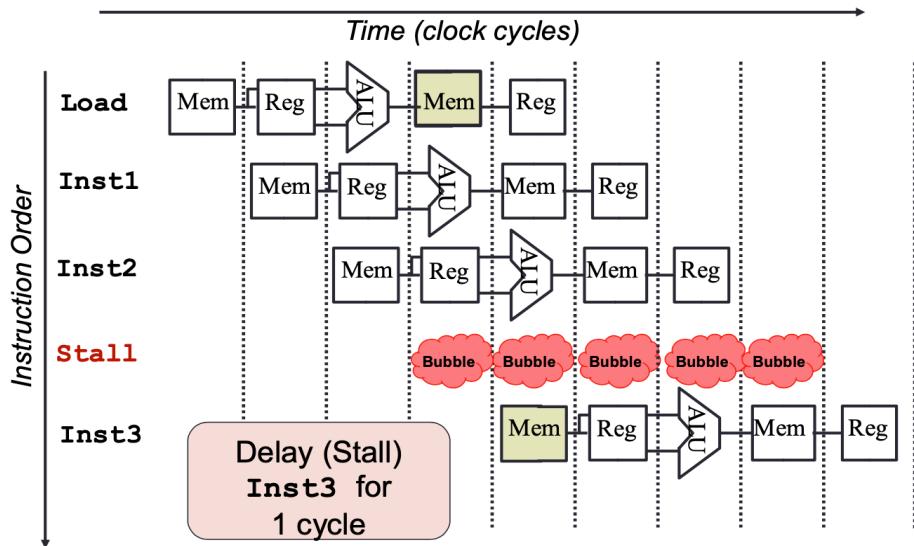
۱. بین دستورات stall بیندازیم تا دستور ابتدایی یک کلاک به جلو برود و مخاطره رخ ندهد.
۲. حافظه‌ی داده‌ها و حافظه‌ی دستورات را جدا کنیم (همانگونه که در پروژه درس وجود دارد)

به مثال زیر توجه کنید:

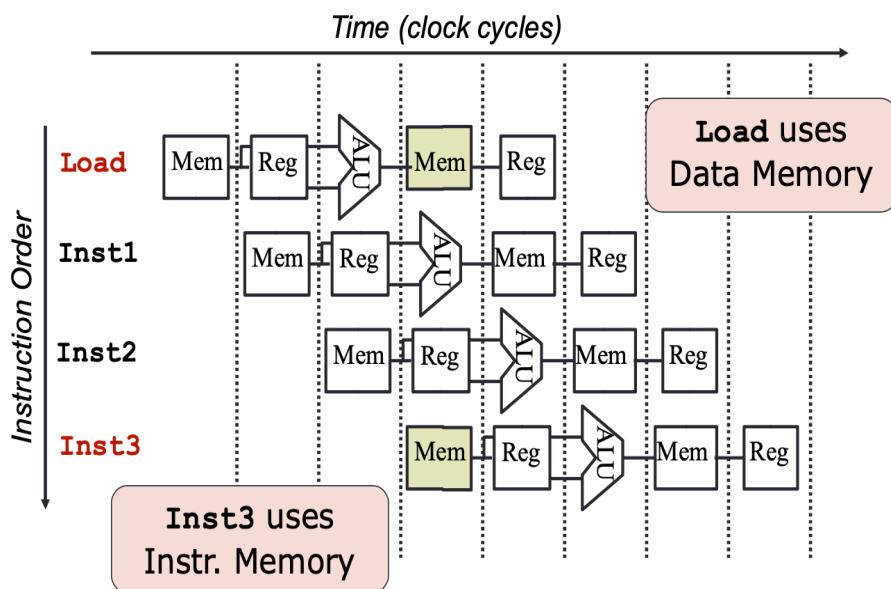
۱. هنگامی که مخاطره‌ی structural رخ می‌دهد:



۲. طریقه‌ی رفع کردن مخاطره – روش اول:



۳. طریقه‌ی رفع کردن مخاطره – روش دوم:



هنگامی که چند دستور با یک رجیستر مشترک یکسان سروکار دارند. این حالت خود، به چند دسته تقسیم می شود: (Data Dependencies)

:RAW(Read After Write) . ۱

این حالت زمانی به وجود می آید که دستور بعدی از رجیستر نوشته شده توسط یک دستور قبلی خوانده شود. از آنجایی که ممکن است در خط لوله هنوز داده‌ی داخل رجیستر بروزرسانی نشده باشد، پس ممکن از داده‌ی بروزرسانی نشده در دستور بعدی خوانده شود و در اینجاست که مخاطره رخ می دهد. به مثال زیر توجه کنید:

```
i1: add $1, $2, $3 #writes to $1
i2: sub $4, $1, $5 #reads from $1
```

در این مثال، اگر i2 از رجیستر \$2 بخواند قبل از اینکه در دستور i1 عمل WriteBack انجام شود، پس نتیجه‌ی i2 نادرست خواهد بود.

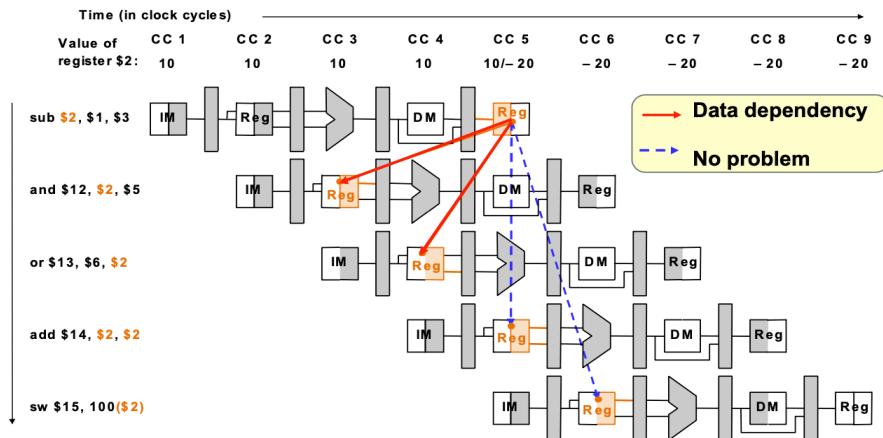
طریقه رفع کردن مخاطره‌ی RAW:

به مثال زیر توجه کنید:

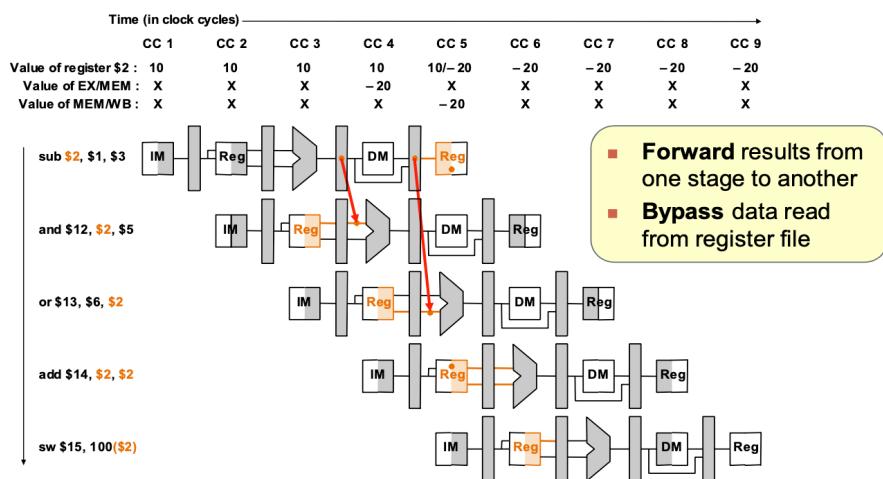
```
sub $2, $1, $3      #i1
and $12, $2, $5    #i2
or  $13, $6, $2    #i3
add $14, $2, $2    #i4
sw   $15, 100($2)  #i5
```

همانطور که مشاهده می شود، از رجیستر \$2 در این چند دستور به صورت مشترک استفاده شده است.

- Value from prior instruction is needed before write back



برای رفع مخاطرات، از روش forwarding استفاده می کنیم. به این صورت که نتیجه را قبل از اینکه در رجیستر اصلی ریخته شود در اختیار هر دستور بعدی که نیاز دارد، قرار می دهیم.



دسته های دوم و سوم:

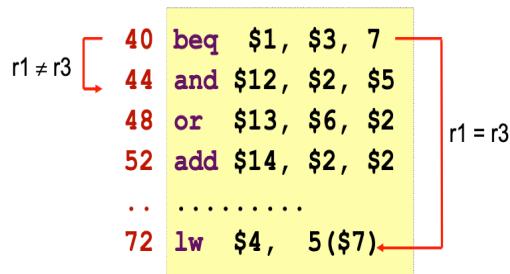
دسته های دوم و سوم عبارت اند از:

1. WAW(Write After Write)
2. WAR(Write After Read)

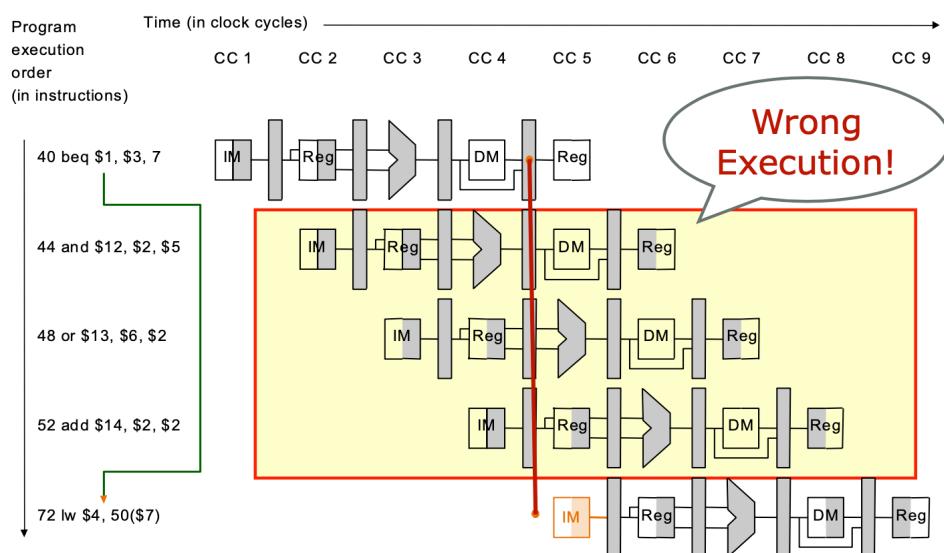
خوшибختانه در پروژه درس(بطور کلی در پردازنده‌ی mips) این دو دسته از مخاطره‌های داده، رخ نمی‌دهند.

Control Hazard ۳-۷-۴

مخاطره‌ی Control زمانی رخ می‌دهد که، که pipeline تصمیمات درستی در رابطه با دستورات شرطی نمی‌گیرد و دستوراتی وارد خط لوله می‌شوند که باید دور ریخته شوند و نیازی به آنها نیست. به مثال زیر توجه کنید:

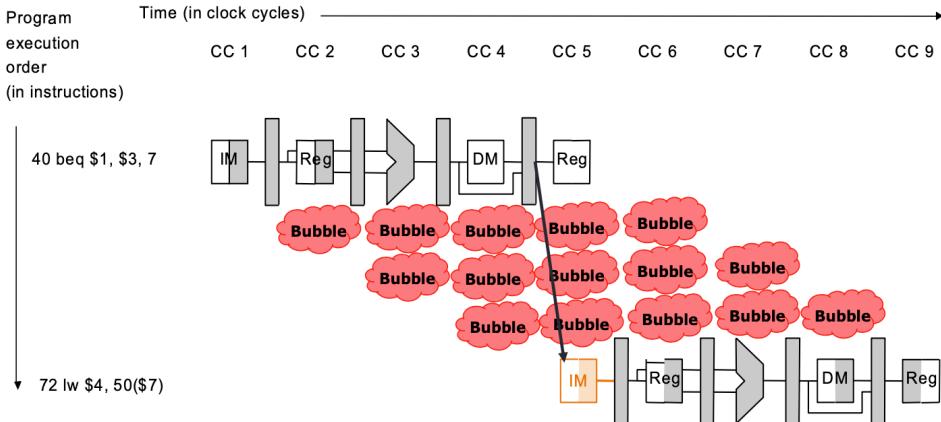


در اینجا، ابتدا دستور اول وارد خط لوله می‌شود و بعد از اینکه مرحله IF روی آن انجام شد، دستور بعدی نیز وارد خط لوله می‌شود. نتیجه‌ی دستور شرطی در مرحله MEM مشخص می‌شود اما تا آن لحظه دستور بعدی وارد شده و اجرا شده است، چه بسا که نتیجه‌ی شرط درست بوده و دستور بعدی نباید اجرا می‌شد. در اینجاست که مخاطره‌ی کنترلی رخ می‌دهد.



طرق رفع این مخاطره:

1. ساده ترین راه حل برای رفع این مخاطره، صبر کردن تا زمانی است که نتیجهٔ دستور شرطی آماده شود. که باید چندین بار stall در خط لوله به وجود بیاوریم.

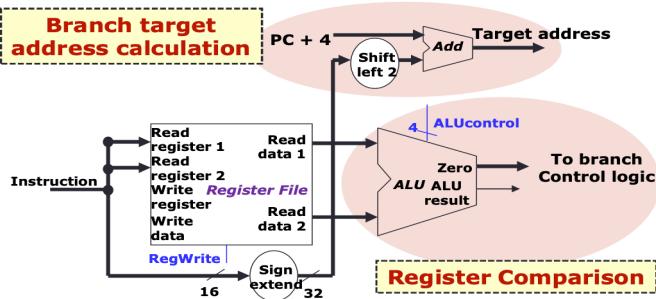


- Wait until the branch outcome is known and then fetch the correct instructions
→ Introduces 3 clock stall cycles

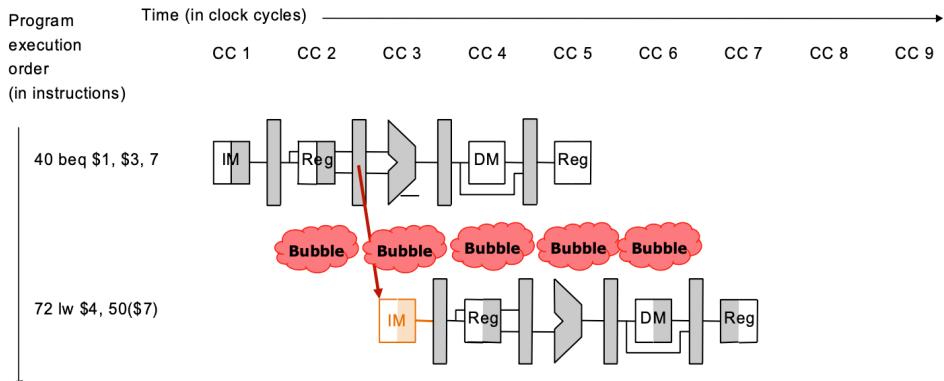
:Early Branch Resolution . ۲

در این راه، می‌خواهیم نتیجهٔ دستور شرطی را سریع‌تر مشخص کنیم. بنابراین، به جای اینکه نتیجهٔ شرط را در مرحله MEM نتیجهٔ شرط را مشخص کنیم، می‌توانیم در مرحله IF این کار را انجام دهیم.

- Move branch target address calculation
- Move register comparison → cannot use ALU for register comparison any more



در مثال ذکر شده داریم (با توجه به اینکه در اینجا در دستور شرطی از رجیستری استفاده شده است که در دستورات قبلی در آن نوشته شده است، پس باید یک stall در اینجا داشته باشیم):



- Wait until the branch decision is known:
 - Then fetch the correct instruction
 - Reduced from 3 to 1 clock cycle delay

در این روش، از Branch Prediction استفاده می کنیم، که خود به دو

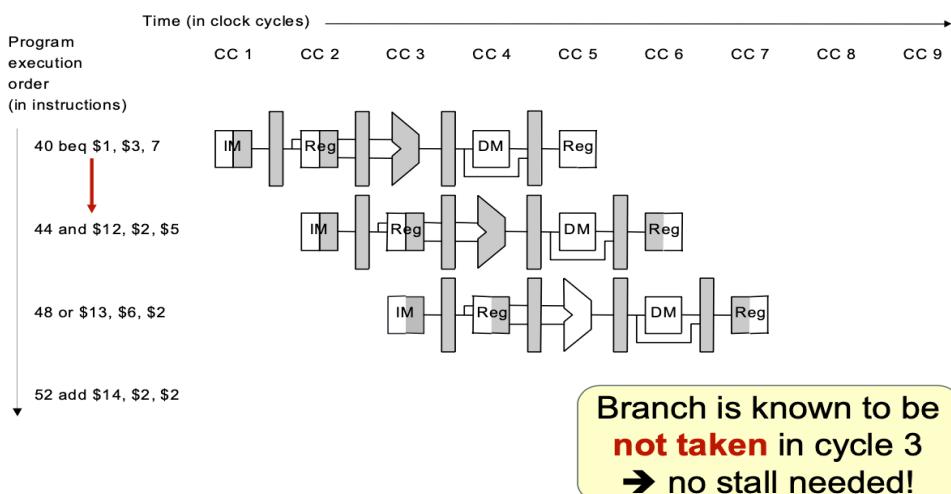
دسته تقسیم می شود:

:Static Prediction (۱)

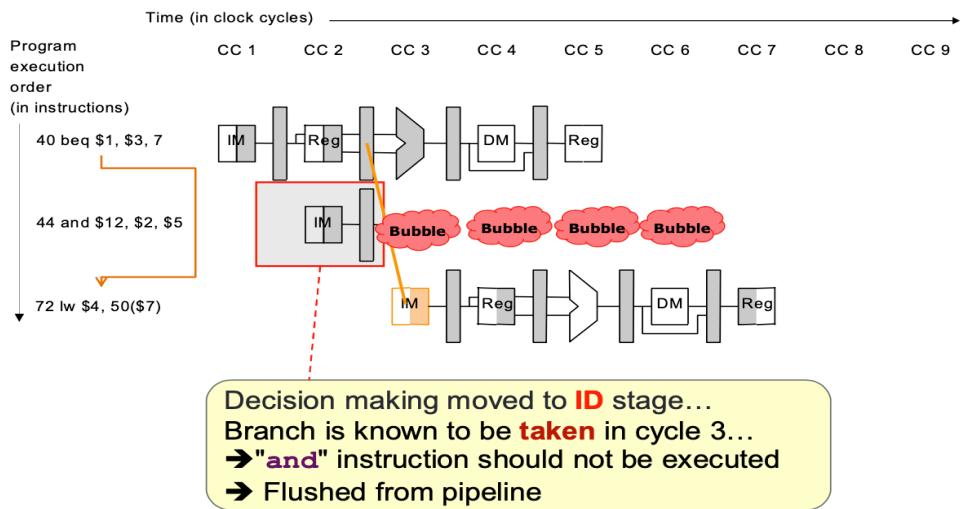
در این روش به عنوان مثال همه دستورات شرطی را taken یا not taken فرض کنیم و یا دستورات شرطی با اپکد خاطری را taken و بقیه را not taken در نظر می گیریم. در

مثال فوق داریم:

Branch Prediction: Correct Prediction



Branch Prediction: Wrong Prediction

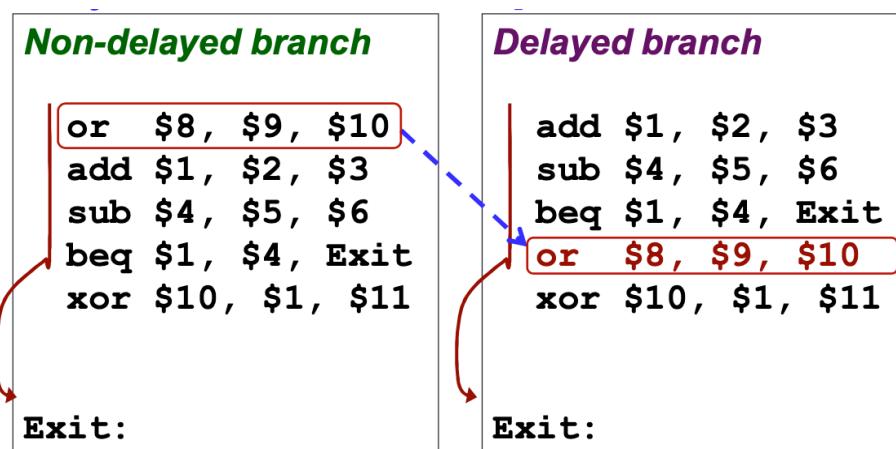


:Dynamic Prediction (ب)

به عنوان مثال در این روش فرض می کنیم که در داخل حلقه، در بعضی از مواقع دستور شرطی مورد نظر taken و در مراحل خاطی از حلقه not taken در نظر گرفته شود.

:Delayed Branching .۴

با توجه به اینکه نتیجه ی دستور شرطی به طور معمول در مرحله MEM مشخص می شود، پس باید تعدادی stall باید در خط لوله گذاشته شود. حال می توانیم تا آماده شدن نتیجه ی دستور شرطی، تعدادی از دستوراتی که به دستور شرطی وابسته نیستند را، در این بازه انجام دهیم که این روش از به کار بردن stall های زیاد جلوگیری می کند.



- The "or" instruction is moved into the delayed slot:
 - Get executed regardless of the branch outcome
 - Same behavior as the original code!