



دانشگاه صنعتی شریف  
دانشکده مهندسی کامپیوتر

مهندسی کامپیوتر

# گزارش پروژه معماری کامپیوتر

نگارش

آرش ضیایی، فرید 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:

امیر محمد درخشنده - شماره دانشجویی: ۴۰۰۱۰۱۱۵۳  
فربد حاجی محمد علی - شماره دانشجویی: ۴۰۱۰۱۱۰۳۸  
آرش ضیایی - شماره دانشجویی: ۴۰۰۱۰۵۱۰۹

# فهرست مطالب

۱	موضوع پروژه	۱
۲	فاز اول	۲
۲	هدف فاز	۱-۲
۴	Datapath و طراحی CPU	۲-۲
۴	instruction spliter	۱-۲-۲
۵	Control Unit	۳-۲
۷	ALU	۴-۲
۱۲	منابع	۳

# 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

#### ۱-۲ هدف فاز

در فاز اول پروژه، می‌خواهیم یک Datapath جامعی از پردازنده مان طراحی کنیم. برای این منظور، ماژول‌های 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 سه فرمت متفاوت هستند که در ادامه بیان‌شان می‌کنیم:

• فرمت R

• فرمت I

• فرمت J

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

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

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 می‌شوند. 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
6 bits	5 bits	5 bits	16 bts

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

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 (`instSplitter`) که نحوه طراحی آنرا در ادامه بیان میکنیم، میکنیم.

### ۱-۲-۲ instruction splitter

این ماژول، در واقع مدار درونی پیچیده ای ندارد. صرفاً 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 `instSplitter` 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.

## ۳-۲ Control Unit

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 مموری نیاز دارند، آیا به رجیستر نیاز دارند، آیا به ALU (Arithmetic Logic Unit) 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 Opcode 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 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 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 می باشد، 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.

## ۴-۲ ALU

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 ماژول ALU مان، باید بخش های مختلف دستور ورودی را، که پیش تر به وسیله splitter، instruction، اجزای دستور اصلی را از هم جدا کرده بودیم، به ورودی ماژول بدهیم. همچنین، چند ورودی به جز اعداد instruction splitter 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 immediate ورودی که در دستور اولیه، آن را استخراج کرده ایم.
- Shift: 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  $B \text{ OR } A$  را خروجی میدهد و در دیگری، حاصل  $\text{Imm OR } A$  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  $\text{AND } A$  را خروجی میدهد و در دیگری، حاصل  $\text{Imm AND } A$  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  $\text{NOR } A$  را خروجی میدهد و در دیگری، حاصل  $\text{Imm NOR } A$  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  $\text{XOR } A$  را خروجی میدهد و در دیگری، حاصل  $\text{Imm XOR } A$  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 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  $5 \times 32$  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  $1 \times 2$  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  $5 * 32$  Mux 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  $1 * 2$  Mux ای که پیش تر گفتیم، وارد یک  $1 * 2$  Mux 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

### منابع