فصل سوم

دستورات زبان ماشین

۳-۱- مقدمه

برای اینکه به یک کامپیوتر دستور دهیم که کاری انجام دهد، باید با زبان او صحبت کنیم. کلمات یک زبان کامپیوتری همان دستورات هستند و مجموعه کلمات یا لغتنامه یک کامپیوتر، مجموعه دستورات[1] نامیده می‌شوند. در این فصل مجموعه دستورات یک کامپیوتر واقعی را هم به فرمی که توسط کاربر نوشته می‌شود و هم به فرمی که توسط ماشین خوانده می‌شود، مورد مطالعه قرار خواهیم داد. زبان ماشین این کامپیوتر واقعی به صورت مرحله به مرحله و به روش بالا به پایین توضیح داده خواهد شد. از نمادی شروع می‌کنیم که ممکن است شبیه یک زبان برنامه نویسی محدود شده باشد، و آن را مرحله به مرحله پالایش می‌کنیم تا زبان واقعی یک کامپیوتر را ببینید.

در این فصل مجموعه دستورات MIPS معرفی خواهد شد. این مجموعه دستورات، نمونه‌ای از چند مجموعه دستوراتی است که از دهه‌ی ۱۹۸۰ به بعد طراحی شده است. این پردازنده هم اکنون جزء پردازنده‌های معروف و پرفروش دنیای کامپیوتر به حساب می‌آید. فقط حدود ۱۰۰ میلیون از این ریزپردازنده معروف در سال ۲۰۰۲ ساخته شده است. این پردازنده را می‌توان در محصولات شرکت-های AII Technology، Broadcom، Cisco، NEC، Nintendo، Silicon Graphics، Sony، Texas Instrument، Toshiba و دیگر تولید کنندگان پیدا کرد.

همان طور که می‌دانیم برنامه‌ای که با زبانهای سطح بالایی همچون C نوشته می‌شود، برای اینکه بر روی ماشین اجرا شود باید توسط کامپایلرهایی همچون gcc کامپایل گردد. نتیجه عملیات کامپایل یک فایل قابل اجرا می‌باشد که شامل دستورات زبان ماشین پردازنده‌ای است که عملیات کامپایل برای آن انجام گرفته است. این دستورات نشان دهندهٔ عملیاتی هستند که توسط آن پردازنده قابل انجام هستند. موقعی که شما برنامه را اجرا می‌کنید این دستورات داخل حافظه بار[2] شده و توسط پردازنده اجرا می-شوند. بنابراین مجموعه دستورات به عنوان واسطی بین سخت‌افزار و نرم‌افزار عمل می‌کند. در حقیقت کامپایلر باید دستوراتی را تولید کند که سخت افزار برای آنها طراحی شده است. مجموعه دستورات نقطه‌ی مشترک بین طراح سخت افزار و طراح کامپایلر است چون کامپایلر دستوراتی که تولید می‌کند باید از یک مجموعه دستورات مشخص باشد و سخت افزار هم باید برای یک مجموعه دستورات مشخص طراحی شده باشد.

---

[1] - Instruction Set
[2] - Load

انتخاب یک مجموعه‌ی دستورات برای اینکه طراحی را به کمک آن انجام دهیم و اینکه شکل و قالب هر دستور به چه صورتی باشد آنقدر اهمیت دارد که یک اصطلاح خیلی مهم به نام ISA[1] یا "معماری مجموعه‌ی دستورات" برای آن در نظر گرفته شده است. ISA آنقدر اهمیت دارد که طراحی سخت افزار و کامپایلر بر اساس آن صورت می‌گیرد. اگر دو پردازنده از دو شرکت مختلف دارای ISA یکسان باشند، کامپایلرشان نیز یکی خواهد بود و هر برنامه‌ای که بر روی یکی از آنها اجرا شود بر روی دیگری نیز حتماً اجرا خواهد شد. ولی اگر ISA یکسان نداشته باشند هم کامپایلرشان متفاوت خواهد بود و هم اینکه برنامه‌های یکی روی دیگری اجرا نخواهد شد. در مورد سخت افزار پردازنده هم باید بگوییم که هر پردازنده‌ای از ابتدا برای یک ISA مشخص ساخته می‌شود. در مورد ISA در ادامه این فصل توضیحات مبسوطی ارائه خواهد شد. در این فصل ما معماری مجموعه دستورات پردازنده MIPS را توضیح خواهیم داد و همه‌ی مباحث نرم افزاری و سخت افزاری که از این به بعد مطرح خواهد شد بر اساس این ISA خواهد بود.

پردازنده‌های قدیمی مجموعه دستورات پیچیده‌ای را استفاده می‌کردند که به آنها CISC [2] می‌گفتند. در این پردازنده‌ها تعداد زیادی دستور قوی وجود داشت که نوشتن برنامه اسمبلی را برای برنامه‌نویس راحت‌تر می‌کردند ولی همین دستورت قدرتمند باعث می‌شدند که طراحی خود پردازنده پیچیده شود و کار طراحان سخت‌افزار خیلی سخت شود. اما اکثر پردازنده‌های جدید از مجموعه دستورات کاهش یافته و ساده‌ای استفاده می‌کنند که به آنها پردازنده‌هایی با معماری RISC [3] گفته می‌شود. در این پردازنده‌ها دستورات ساده و نسبتاً کمی وجود دارد و برای برنامه‌نویسی آنها از زبانهای برنامه‌نویسی سطح بالا و کامپایلرهای قدرتمندی استفاده می‌شود که کار برنامه‌نویس را راحت‌تر می‌کند. در این پردازنده‌ها کاربر کمتر به زبان اسمبلی برنامه می‌نویسد و بیشتر با زبانهای سطح بالا کدنویسی می‌کند. سادگی دستورات و کم بودن تعداد آنها در پردازنده‌های RISC باعث شده است که طراحی سخت-افزار توسط طراح راحت‌تر شده و بعدها به راحتی قابل بهینه‌سازی باشد. قابل ذکر است که پردازنده‌ی MIPS از معماری‌های پیشرفته RISC استفاده می‌کند.

توجه: امروزه اکثریت قریب به یقین پردازنده‌ها از معماری‌های RISC استفاده می‌کنند و حتی اکثر پردازنده‌های CISC همانند پردازنده‌های اینتل با استفاده از تکنیکهای RISC پیاده سازی می‌شوند.

---

[1] - Instruction Set Architecture
[2] - Complex Instruction Set Computer
[3] - Reduced Instruction Set Computer

۳-۲- عملیات سخت افزار کامپیوتر

هر کامپیوتری باید توان انجام محاسبات را داشته باشد. نماد زبان اسمبلی زیر را در نظر بگیرید:

```
add  a, b, c
```

در این نماد به کامپیوتر دستور داده می‌شود که دو متغیر b و c را جمع کرده و حاصل را در a قرار دهد. این نمادگذاری به دلیل اینکه در آن هر دستورالعمل حسابی فقط یک عملیات را انجام می‌دهد و باید سه متغیر داشته باشد، انعطاف پذیر نیست. به طور مثال با نماد فوق نمی‌توانیم حاصل جمع چهار متغیر b، c، d، و e را در a قرار دهیم و باید از رشته دستورالعمل‌های زیر استفاده کنیم:

```
add  a, b, c    # a = b + c
add  a, a, d    # a = (b + c) + d
add  a, a, e    # a = ((b + c) + d) + e = b + c + d + e
```

در خطوط فوق کلمات نوشته شده در سمت راست علامت # ، توضیحات برای خواننده هستند که کامپیوتر آنها را در نظر نمی‌گیرد و هر خط این زبان حداکثر دارای یک دستورالعمل می‌باشد. اختلاف دیگر آن با زبان برنامه نویسی C در این است که در آن برخلاف زبان C توضیحات همیشه در انتهای یک خط پایان می‌یابند. تعداد طبیعی عملوندها برای عملیاتی نظیر جمع، سه عملوند است: دو عددی که باید جمع شوند و مکانی که حاصل جمع باید در آن قرار گیرد. این نیازمندی که هر دستورالعمل دقیقاً سه عملوند داشته باشد، نه بیشتر و نه کمتر، برای هماهنگی با این فلسفه که سخت افزار را ساده-تر کنید، تعیین شده است. سخت افزار برای تعداد عملوندها متغیر، پیچیده‌تر از تعداد عملوندهای ثابت است. این وضعیت اولین اصل از چهار اصل پایه‌ای طراحی سخت افزار را بیان می‌کند.

اصل شماره ۱ طراحی: سادگی به نظم کمک می‌کند.

در دو مثال زیر رابطه‌ی بین برنامه‌های نوشته شده به زبان C و این نمادگذاری ابتدایی نشان داده می-شود. کامپایلر عمل تبدیل را از یک برنامه سطح بالا به زبان اسمبلی و زبان ماشین انجام می‌دهد. در مثالهای زیر در واقع شما عمل کامپایلر را خودتان به صورت دستی انجام می‌دهید.

مثال: کد اسمبلی معادل با قطعه برنامه‌ی زیر را نشان دهید.

```
a = b + c;
d = a – e;
```

پاسخ: چون یک دستورالعمل در نمادگذاری نشان داده شده بر روی دو متغیر مبدأ عمل کرده و حاصل را در یک متغیر مقصد قرار می‌دهد، بنابراین قطعه کد بالایی مستقیماً به دو دستورالعمل زبان اسمبلی زیر تبدیل می‌شود:

```
add  a, b, c   #  a = b + c
sub  d, a, e   #  d = a – e
```

مثال: یک عبارت با پیچیدگی بیشتر را به صورت زیر در نظر بگیرید:

$$f = (g + h) - (i + j);$$

کامپایلر C چه کدی را برای این عبارت تولید خواهد کرد؟

پاسخ: این عبارت پیچیده با چند دستور ساده‌ی اسمبلی پیاده سازی می‌شود. ابتدا جمع (g + h)، سپس جمع (i + j)، و در نهایت عملیات تفریق انجام می‌گیرد.

```
add  t0, g, h
add  t1, i, j
sub  f, t0, t1
```

در این کد اسمبلی، t0 و t1 به عنوان متغیرهای موقتی برای ذخیره کردن نتایج برای استفاده در آینده مورد استفاده قرار گرفته‌اند.

توجه: نمایش‌های نمادینی (زبان اسمبلی) که در این بخش معرفی شدند، در واقع همان چیزی است که پردازنده واقعاً می‌فهمد. در بخش‌های آتی نمایش نمادین زبان ماشین MIPS توضیح داده خواهد شد.

۳-۳- عملوندهای سخت افزار کامپیوتر

برای هر عملیاتی تعدادی ورودی وجود دارد که عملیات بر روی آنها انجام می‌شود. در معماری کامپیوتر به هر کدام از ورودی‌های یک عملیات[1] ، عملوند[2] گفته می‌شود. هر کدام از رجیسترها، خانه-های حافظه و ثابت‌های عددی می‌توانند عملوند یک دستور باشند. در این بخش عملوندهای مختلف موجود در معماری MIPS مورد بررسی قرار می‌گیرند.

۳-۳-۱- عملوندهای رجیستر

تعداد عملوندهای دستورالعمل‌های حسابی در زبان اسمبلی برخلاف برنامه‌های سطح بالا محدود می-باشند و باید از تعداد محدودی مکان‌های خاص که مستقیماً در سخت افزار ساخته شده و رجیستر نامیده می‌شوند، انتخاب شوند. رجیسترها همانند آجرهای ساختمان یک کامپیوتر می‌باشند: رجیسترها عناصر اولیه‌ای هستند که در طراحی سخت افزار به کار گرفته می‌شوند که پس از تکمیل کامپیوتر برای برنامه نویس قابل مشاهده می‌باشند. در فصل‌های ۵ و ۶ نقش کلیدی رجیسترها در طراحی و ساخت

[1] - Operator
[2] - Operand

سخت افزار نشان داده خواهد شد. در این فصل نیز مشاهده خواهید کرد که استفاده مؤثر از رجیسترها چگونه در کارآیی برنامه نقش کلیدی بر عهده دارد.

اندازه رجیستر در معماری MIPS، ۳۲ بیت است. معمولاً یک داده‌ی ۳۲ بیتی را در معماری MIPS، کلمه[1] می‌نامند. لازم به یادآوری است که به ۱۶ بیت یا دو بایت، یک نیم کلمه[2] گفته می‌شود.

یکی از تفاوتهایی که بین متغیرهای یک زبان برنامه نویسی مانند C و رجیسترها این است که تعداد رجیسترها محدودند. معمولاً کامپیوترهای امروزی ۳۲ رجیستر دارند. معماری MIPS نیز دارای ۳۲ رجیستر می‌باشد. بنابراین در ادامه مسیر گام به گام و از بالا به پایین نمایش نمادین زبان MIPS، این محدودیت را نیز اضافه می‌کنیم که هر یک از سه عملوند دستورالعمل‌های حسابی MIPS باید از یکی از ۳۲ رجیستر ۳۲ بیتی انتخاب شوند.

دلیل محدود بودن تعداد رجیسترها (برای MIPS ۳۲ است) را می‌توان در دومین اصل از اصول طراحی سخت افزار یافت:

اصل شماره ۲ طراحی: کوچک‌تر سریعتر است.

هر چقدر تعداد رجیسترها بیشتر شود تأخیر مجموعه رجیسترها هم به دلیل بزرگ شدن حجم سخت-افزار بیشتر خواهد شد. این امر باعث می‌شود که پریود کلاک افزایش پیدا کرده و زمان زیادی صرف اجرای دستورات شود. دلیل دیگر عدم استفاده‌ی بیشتر از ۳۲ رجیستر، تعداد بیت‌هایی است که در قالب دستورالعمل قرار می‌گیرد. این مورد در ادامه این فصل توضیح داده خواهد شد.

اگر چه می‌توانیم دستورالعمل‌ها را به سادگی با استفاده از عدد رجیسترها از صفر تا ۳۱ بنویسیم، اما قرارداد MIPS استفاده از نام دو کاراکتری است که قبل از آن علامت $ (دلار) قرار می‌گیرد. علت این نامگذاری در ادامه‌ی این فصل توضیح داده خواهد شد. فعلاً از $s0 و $s1 و ... برای رجیسترهای متناظر با متغیرهای زبان C و از $t0 و $t1 و ... برای رجیسترهای موقت که هنگام کامپایل برنامه به دستورالعمل‌های MIPS مورد نیاز هستند، استفاده خواهیم کرد.

مثال: یکی از وظایف کامپایلر، اختصاص دادن رجیسترها به متغیرهای برنامه است. برای نمونه، دستور انتساب مثال قبل را در نظر بگیرید:

$$F = (g + h) - (i + j)$$

---

[1] - Word
[2] - Half word

با فرض اینکه کامپایلر متغیرهای f ، g ، h ، i و j را به ترتیب به رجیسترهای $s0، $s1، $s2، $s3 و $s4 منتسب کرده باشد، کد MIPS حاصل از کامپایل را بدست آورید.

پاسخ: برنامه کامپایل شده، شبیه مثال قبل است با این تفاوت که در آن متغیرها با نام رجیسترها جایگزین شده‌اند و دو رجیستر موقت $t0 و $t1 را به جای متغیرهای موقت t0 و t1 استفاده کرده‌ایم:

```
add   $t0, $s1, $s2
add   $t1, $s3, $s4
sub   $s0, $t0, $t1
```

۳-۳-۲- عملوندهای حافظه

رجیسترها دارای سرعت زیادی هستند و کارکردن با آنها راحت‌تر است ولی ما فقط ۳۲ عدد رجیستر در اختیار داریم و هر کدام از رجیسترها فقط می‌توانند ۳۲ بیت داده در داخل خود نگهداری کنند. اگر ما ساختمان داده‌هایی نظیر آرایه‌ها و ساختمان‌ها[1] داشته باشیم نمی‌توانیم آنها را داخل رجیسترها ذخیره‌سازی کنیم چون تعداد عناصر آنها ممکن است از تعداد رجیسترها بیشتر باشد. همچنین اگر فقط از رجیسترها استفاده کنیم نمی‌توانیم با داده‌هایی که طول آنها از ۳۲ بیت بزرگتر است کار بکنیم. بنابراین ما برای ذخیره‌سازی داده‌های خود غیر از رجیسترها به حافظه هم نیاز پیدا می‌کنیم. حافظه RAM در مقایسه با رجیسترها داده‌های بیشتری را ذخیره می‌نماید ولی از آنجا که حافظه‌ها سرعت پایین‌تری دارند تا جایی که ممکن است بهتر است از رجیسترها استفاده کنیم. در زمانهای گذشته استفاده از رجیسترها کار برنامه‌نویس‌ها بود. به طور مثال زبان برنامه‌نویسی C که دارای کلمه کلیدی register می‌باشد، این امکان را در اختیار قرار می‌دهد که برنامه‌نویس متغیرهای پر استفاده را که بهتر است در داخل رجیسترها قرار داده شوند، با این کلمه کلیدی تعریف نماید. کامپایلر هم تا حد امکان سعی می‌کند این نوع متغیرها را در داخل رجیسترها قرار دهد. اما کامپایلرهای مدرن یک کار جالب انجام می‌دهند و آن اینکه رجیسترها را بدون دخالت برنامه‌نویس به صورت هوشمندانه‌ای برای متغیرها استفاده می‌کنند و تعداد دسترسی‌ها به حافظه RAM را کاهش می‌دهند.

مرور حافظه

حافظه همانند یک آرایه بزرگ می‌باشد و همان طور که یک آرایه برای دسترسی به عناصر خود یک اندیس دارد که آن عناصر را آدرس‌دهی می‌کند حافظه نیز برای دسترسی به خانه‌های خود یک آدرس دارد که آن خانه‌ها را آدرس‌دهی می‌کند. اگر تعداد خطوط آدرس یک حافظه، k باشد در این صورت

[1] - Structure

این حافظه دارای 2^k خانه خواهد بود و اگر هر خانه از حافظه نیز دارای n بیت باشد در این صورت اندازه حافظه به صورت $2^k \times n$ نشان داده خواهد شد. بلوک دیاگرام یک حافظه RAM به همراه جدول درستی آن در شکل ۱ داده شده است.

CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

شکل ۱: بلاک دیاگرام یک حافظه به همراه جدول صحت آن

عملکرد حافظه RAM به صورت زیر است:

ورودی CS [1] : برای فعال کردن یا غیر فعال کردن حافظه به کار می‌رود.

ورودی ADRS : آدرس خانه‌ای از حافظه را مشخص می‌کند که به آن دسترسی خواهد شد.

خط WR برای انتخاب عملیات خواندن یا نوشتن حافظه به کار می‌رود.

برای خواندن از حافظه خط WR باید مساوی صفر شود در این صورت محتوای خانه‌ای که با آدرس ADRS مشخص شده است بر روی خروجی OUT قرار خواهد گرفت. اگر اصطلاح Memory را به عنوان نام آرایه حافظه و ADRS را به عنوان اندیس آن در نظر بگیریم در واقع به هنگام خواندن حافظه عملیات OUT=Memory[ADRS] انجام می‌شود. برای نوشتن به حافظه، خط WR باید مساوی یک شود در این حالت داده‌ای که روی خط DATA قرار دارد در آدرس ADRS نوشته خواهد شد یعنی عملیاتی نظیر Memory[ADRS]=DATA انجام می‌گیرد.

اگر اندازه هر خانه از حافظه یک بایت باشد، در این صورت در هر آدرس یا خانه حافظه می‌توان یک بایت را نوشت یا خواند. به اصطلاح در این صورت حافظه قابلیت دسترسی بایتی [2] را داراست. حافظه پردازنده MIPS نیز قابلیت دسترسی به صورت بایتی را دارد. پردازنده MIPS می‌تواند تا ۳۲ بیت خط آدرس را پشتیبانی کند یعنی اینکه در این پردازنده می‌توانیم حافظه‌ای با اندازهٔ $2^{32} \times 8$ یا 4GB (۴ گیگا بایت) داشته باشیم. البته این اندازه حافظه خیلی زیاد است و در عمل کمتر ماشین MIPS این اندازه حافظه را در اختیار دارد!

ذخیره و بازیابی بایتهای حافظه [3]

[1] - Chip Select
[2] - Byte addressable
[3] - Loading and Storing bytes

مجموعه دستورات پردازنده MIPS دارای دستوراتی برای دسترسی به حافظه می‌باشد (دستورهای Load وStore). MIPS در دسترسی به حافظه از روش آدرس‌دهی شاخص‌دار[1] استفاده می‌کند یعنی در عملوند دستورات مراجعه به حافظه، یک عدد ثابت علامت‌دار و یک رجیستر وجود دارند که این دو مقدار با هم جمع شده و یک آدرس مؤثر برای حافظه تولید می‌کنند. دستور بار کردن (load byte) یا lb پردازنده MIPS یک بایت داده را از حافظه خوانده و به داخل یک رجیستر منتقل می‌کند. مثالی از دستور lb به صورت زیر است:

lb $t0, 20($a0) # $t0 =Memory[$a0 + 20]

شکل دستور ذخیره کردن بایت (store byte) یا sb همانند lb است با این تفاوت که sb یک بایت داده را از یک رجیستر به داخل حافظه منتقل می‌کند. مثالی از sb به صورت زیر است:

sb $t0, 20($a0) # Memory[$a0 + 20] =$t0

آدرس دهی شاخص‌دار برای دسترسی به خانه‌های متوالی حافظه همانند عناصر آرایه‌ها بسیار مفید است. در این صورت عدد ثابت مشخص کننده آدرس پایه یا همان شروع آرایه و رجیستر نشان دهندهٔ عنصری از آرایه است که مورد دسترسی قرار خواهد گرفت. به طور مثال اگر $a0=0 باشد در این صورت دستور lb $t0, 2000($a0) اولین خانه از یک آرایه را که از آدرس 2000 شروع می‌شود مشخص خواهد نمود. اگر $a0=8 باشد در این صورت دستور lb $t0,2000($a0) به بایت نهم آرایه که در آدرس 2008 است اشاره خواهد نمود.

توجه: مثال فوق دلیل اینکه اندیس آرایه‌ها در زبانهای برنامه‌نویسی C و جاوا به جای ۱ از ۰ شروع می‌شوند را نشان می‌دهد.

آدرس دهی شاخص‌دار را می‌توان به گونه دیگری هم در نظر گرفت. در این حالت نقش عدد ثابت و رجیستر عوض می‌شود. یعنی اینکه رجیستر آدرس پایه یا شروع آرایه و عدد ثابت اندیس را مشخص می‌نمایند. این حالت برای مواقعی مفید خواهد بود که دقیقاً بدانیم که به کدام عنصر آرایه یا ساختمان دسترسی خواهیم داشت. به طور مثال اگر $a0=2000 باشد، در این صورت دستور lb $t0,0($a0) بایت اول آرایه‌ای که از آدرس ۲۰۰۰ در حافظه قرار گرفته است را مورد دسترسی قرار خواهد داد و دستور lb $t0, 8($a0) برای دسترسی به عنصر نهم به کار خواهد رفت.

توجه: مقدار ثابت در دستورالعمل‌های انتقال داده (load و store)، *آفست*[2] ، و رجیستر افزوده شده برای تشکیل آدرس، رجیستر *پایه*[3] نامیده می‌شود.

---

[1] - Index addressing
[2] - offset
[3] - Base register

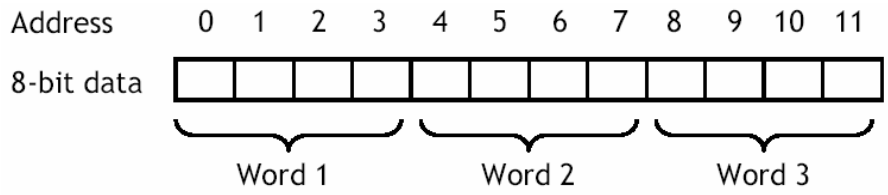
دسترسی به داده‌های ۳۲ بیتی حافظه در پردازنده MIPS

در پردازنده MIPS می‌توان با استفاده از دستورهای lw و sw به داده‌های ۳۲ بیتی حافظه دسترسی پیدا کرد. مثالی از این دستورات به صورت زیر است:

```
lw $t0, 20($a0)      # $t0 =Memory[$a0 + 20]
sw $t0, 20($a0)      # Memory[$a0 + 20] =$t0
```

اکثر زبانهای برنامه‌نویسی از نوع داده ۳۲ بیتی پشتیبانی می‌کنند به طور مثال انواع داده‌های صحیح، داده‌های ممیز شناور و آدرس‌های حافظه (اشاره‌گرها)، ۳۲ بیتی هستند. در این کتاب فرض ما این است که داده‌های ما ۳۲ بیتی هستند مگر در مواقعی که طول داده را مشخص نماییم. منظور از کلمه نیز در این کتاب همان کلمه ۳۲ بیتی است مگر در مواردی که طول کلمه را صریحاً گفته باشیم.

همان طور که گفته شد، در پردازنده MIPS حافظه قابلیت دسترسی بایتی دارد. بنابراین یک کلمه ۳۲ بیتی چهارخانه متوالی از حافظه اصلی را اشغال می‌کند. چگونگی قرار گرفتن کلمه‌های ۳۲ بیتی در حافظه، در شکل زیر نشان داده شده است.



شکل ۲: طرز قرار گرفتن کلمه‌های ۳۲ بیتی در حافظه پردازنده MIPS

در معماری MIPS، کلمه‌های ۳۲ بیتی باید به صورت تراز شده[1] در حافظه قرار بگیرند و تراز بودن به این معنا است که کلمه‌های ۳۲ بیتی باید از آدرس‌هایی شروع شوند که آن آدرس‌ها بر ۴ بخش‌پذیر باشند. بنابراین آدرس‌های ۰ ، ۴ ، ۸ و ۱۲ آدرس‌های معتبری برای کلمه‌های ۳۲ بیتی هستند. ولی آدرس‌های ۱ ، ۲ ، ۳ ، ۵ ، ۶ ، ۷ ، ۹ ، ۱۰ و ۱۱ آدرس‌های معتبری برای کلمه‌های ۳۲ بیتی نیستند. آدرس‌های تراز نشده برای حافظه برای کلمه‌های ۳۲ بیتی باعث ایجاد خطای bus error می‌شود که احتمالاً به این خطا در اجرای برنامه‌های خود برخورد کرده باشید.

توجه: تراز بودن کلمه‌های ۳۲ بیتی حافظه مطمئناً محدودیتهایی را بر زبانهای برنامه‌نویسی سطح بالا و همچنین کامپایلرها تحمیل می‌کند ولی همین امر باعث می‌شود که طراحی پردازنده راحت‌تر شود و همچنین سرعت آن بالاتر رود.

[1] - Aligned

ما روش آدرس‌دهی حافظه را برای دسترسی‌های بایتی توضیح دادیم حال می‌خواهیم همان مسأله را برای دسترسی‌های ۳۲ بیتی توضیح دهیم. به طور مثال فرض کنید که آرایه‌ای از کلمه‌های ۳۲ بیتی از آدرس ۲۰۰۰ حافظه شروع شده باشد در این صورت اولین عنصر آرایه در آدرس ۲۰۰۰ خواهد بود و دومین عنصر حافظه در آدرس ۲۰۰۴ خواهد بود نه در آدرس ۲۰۰۱ و این به این دلیل است که هر عنصر آرایه دارای ۴ بایت (۳۲ بیت) می‌باشد. اگر رجیستر $a0=2000 باشد در این صورت دستور lw $t0 , 0($a0) ، به عنصر اول آرایه دسترسی خواهد داشت. اما دستور lw $t0 , 8($a0) به عنصر سوم آرایه که در آدرس ۲۰۰۸ است دسترسی خواهد داشت.

محاسبات با استفاده از حافظه

در پردازنده MIPS به طور مستقیم نمی‌توان بر روی کلمه‌های حافظه عملیات محاسباتی انجام داد و فقط برای دستورهای load و store به حافظه مراجعه می‌شود. برای انجام محاسبه با استفاده از داده‌های ذخیره شده در حافظه باید موارد زیر را انجام داد:

۱. داده‌های مورد نظر را با استفاده از دستورهای load به داخل رجیسترها منتقل نماییم.
۲. عملیات را با استفاده از رجیسترها انجام داده و نتایج را در داخل رجیسترها ذخیره کنیم.
۳. نتایج را که هم اکنون در داخل رجیسترها قرار دارند با استفاده از دستورهای store به داخل حافظه منتقل نماییم.

نکته: در پردازنده MIPS همه عملیات محاسباتی بر روی رجیسترها انجام می‌گیرد و نتیجه داخل یک رجیستر ذخیره می‌شود.

مثال: فرض کنید که A یک آرایه‌ی ۱۰۰ عنصری از بایتها باشد و کامپایلر متغیرهای g و h را به رجیسترهای $s1 و $s2 منتسب کرده باشد. آدرس شروع آرایه که آدرس پایه نامیده می‌شود، در $s3 قرار دارد. عبارت انتساب زیر را کامپایل کنید:

g = h + A[8]

پاسخ: هر چند در این عبارت، یک عملیات وجود دارد، اما به دلیل اینکه یکی از عملوندها در حافظه قرار دارد، بنابراین ابتدا باید A[8] را به داخل یک رجیستر منتقل کرده و سپس آن را با h جمع کنیم. آدرس این عنصر آرایه، برابر است با حاصل جمع آدرس پایه‌ی آرایه A که در رجیستر $s3 وجود دارد، و یک عدد که عنصر موجود در اندیس 8 را انتخاب می‌کند. از آنجا که اندیس آرایه در زبان C از

صفر شروع می‌شود، بنابراین آدرس پایه نشان دهنده اولین عنصر آرایه خواهد بود و چون هر عنصر آرایه، یک بایت است، بنابراین برای رسیدن به اندیس 8 باید به آدرس پایه عدد ۸ را اضافه کنیم.
بنابراین با استفاده از دو دستور زیر می‌توان این مثال را انجام داد که در آن دستور اول عنصر آرایه را به داخل یک رجیستر موقت منتقل می‌کند و دستور دوم آن رجیستر موقت را به h اضافه کرده و نتیجه را در g ذخیره می‌کند.

```
lw   $t0, 8($s3)
add  $s1, $s2, $t0
```

مثال: فرض کنید رجیستر $s2 به متغیر h اختصاص پیدا کرده باشد و آدرس پایه‌ی آرایه A در $s3 قرار گرفته باشد. با فرض اینکه عناصر آرایه‌ی A ، ۳۲ بیتی باشند، کد اسمبلی ماشین MIPS را برای عبارت زیر بنویسید.

A[12] = h + A[8]

پاسخ: برای این مثال ابتدا باید A[8] را به داخل یک رجیستر منتقل نمود، بعد عملیات جمع را انجام داد و بعد نتیجه بدست آمده را که داخل یک رجیستر قرار گرفته به A[12] منتقل نمود. در اینجا به دلیل اینکه کلمات ۳۲ بیتی هستند، کلمه اول (A[0]) در آدرس $s3 (آدرس پایه)، کلمه دوم (A[1]) چهار بایت با کلمه اول فاصله داشته و در آدرس $s3+4، و به همین ترتیب تا اینکه کلمه نهم (A[8]) در آدرس $s3+32 (8×4=32) و کلمه سیزدهم (A[12]) در آدرس $s3+48 (12×4=48) قرار دارد.
بنابراین کد اسمبلی MIPS به صورت زیر نوشته می‌شود:

```
lw   $t0, 32($s3)   # t0 = A[8]
add  $t0, $s2, $t0  # t0 = h + A[8]
sw   $t0, 48($s3)   # A[12] = h + A[8]
```

big endian و little endian

به دلیل اینکه هر خانه حافظه دارای یک بایت می‌باشد، ذخیره کردن یک کلمه نیاز به چهار بایت دارد. برای ذخیره کردن یک کلمه در حافظه دو روش بسیار معروف وجود دارد: big endian و little endian. در روش big endian، بایت با ارزش کلمه در آدرس پایین‌تر ذخیره می‌شود و به دنبال آن بایت‌های دیگر قرار می‌گیرند تا اینکه بایت کم ارزش کلمه نیز در آدرس بزرگتر ذخیره می‌شود. به طور مثال فرض کنید بخواهیم یک کلمه را در آدرس ۴ حافظه ذخیره کنیم. در این صورت می‌دانیم که آدرسهای از ۴ تا ۷ به این کلمه اختصاص پیدا می‌کند. در روش big endian ، بایت ۳ (byte3) که با

ارزش‌ترین بایت است در آدرس ۴ ، byte2 که بایت با ارزش بعدی است در آدرس ۵ ، byte1 در آدرس ۶ و در نهایت byte0 که کم ارزش‌ترین بایت کلمه است در آدرس ۷ قرار می‌گیرد.

روش little endian درست بر عکس روش big endian است، یعنی در ذخیره یک کلمه، بایت کم ارزش در آدرس پایین و بایت با ارزش در آدرس بالا قرار می‌گیرد.

تلاقی سخت افزار و نرم افزار: کامپایلر، علاوه بر متناظر کردن متغیرها با رجیسترها، ساختمان داده-هایی نظیر آرایه‌ها و ساختارها را به مکان‌های حافظه تخصیص می‌دهد. سپس کامپایلر می‌تواند آدرس شروع صحیح برای آرایه را در داخل یک رجیستر قرار داده و در دستورالعمل انتقال داده، آن را در کنار یک عدد ثابت (آفست) برای بدست آوردن آدرس عناصرش استفاده کند.

تلاقی سخت افزار و نرم افزار: تعداد متغیرها در بسیاری از برنامه‌ها، بیشتر از تعداد رجیسترهای کامپیوترهاست. در نتیجه کامپایلر تلاش می‌کند تا متغیرهایی که بیشتر مورد استفاده قرار می‌گیرند را در رجیسترها و بقیه را در حافظه نگهدارد. فرآیند در حافظه قرار دادن متغیرهایی که از آنها کمتر استفاده می‌شود (یا متغیرهایی که بعداً مورد نیاز هستند)، ریختن رجیسترها [1] نامیده می‌شود.

تفصیل بیشتر: اگرچه رجیسترهای MIPS در این کتاب ۳۲ بیتی هستند، اما نوع مجموعه دستورالعمل ۶۴ بیتی با ۳۲ رجیستر ۶۴ بیتی نیز وجود دارد. به معماری مجموعه دستوراتی که دارای ۳۲ رجیستر ۳۲ بیتی است، MIPS32 و به معماری مجموعه دستوراتی که دارای ۳۲ رجیستر ۶۴ بیتی است، MIPS64 گفته می‌شود. در این فصل، زیر مجموعه‌ای از MIPS32 مورد بررسی قرار می‌گیرد.

۳-۳-۳- عملوندهای بلافصل [2] (فوری) یا ثابت

در بسیاری از برنامه‌ها از یک مقدار ثابت به عنوان عملوند استفاده می‌شود. مثالی از این نوع عملوندها، افزایش شاخص یک آرایه به اندازه یک عدد ثابت برای اشاره به عنصر بعدی است. عملوندهای بلافصل فراوانی بالایی در برنامه‌ها دارند، به طور مثال حدود نیمی از دستورالعمل‌های حسابی MIPS برای آزمون کارآیی SPEC2000 ، دارای مقدار ثابتی به عنوان عملوند می‌باشند.

می‌خواهیم با استفاده از دستوراتی که تاکنون مطالعه کرده‌ایم، یک مقدار ثابت را از حافظه به داخل یک رجیستر منتقل کنیم (ثابت‌ها به هنگام بار شدن برنامه در حافظه قرار می‌گیرند). به طور مثال، برای اضافه کردن ثابت ۴ به رجیستر $s3 می‌توانیم از کد زیر استفاده کنیم:

```
lw   $t0, AddrConstant4($s1)  #  $t0 = constant  4
add  $s3, $s3, $t0            #  $s3 = $s3 +$t0 ; ($t0 = 4)
```

[1] - Register Spilling
[2] - Immediate

در کد فوق فرض بر این است که AddrConstant4، نشان دهنده آدرسی از حافظه است که در آن آدرس عدد ۴ ذخیره شده است.

روش دیگر به جای استفاده از دستور بار کردن، استفاده از یک دستورالعمل جمع است که در آن یکی از عملوندها، مقدار ثابت است. این دستورالعمل جمع با یک عملوند ثابت، جمع فوری یا addi نامیده و به صورت زیر استفاده می‌شود:

```
addi   $s3, $s3, 4    # $s3 = $s3 + 4
```

دستورالعمل‌های فوری باعث کاهش تعداد مراجعات به حافظه و همچنین باعث کاهش تعداد دستورات می‌شوند و به همین دلایل باعث می‌شوند که سرعت اجرای برنامه‌ها افزایش پیدا کند. وجود دستورالعمل‌های فوری به دلیل اینکه ثابت‌ها در برنامه‌ها زیاد استفاده می‌شوند، می‌باشد. در واقع با این کار طراحان MIPS طبق قانون امدال، عمل کرده و دستوری طراحی کرده‌اند که سرعت موارد پر استفاده در برنامه را افزایش دهند. بنابراین اصل سوم طراحی سخت افزار به این صورت تشریح می-شود:

اصل شماره ۳ طراحی: موارد پر استفاده را سریع‌تر کنید.

رجیستر صفر($0) یا $zero در پردازنده MIPS همیشه مقدار صفر را نگهداری می‌کند و نمی‌توان محتوای آن را تغییر داد. با توجه به این مطلب می‌توان با استفاده از رجیستر $0 و عملوندهای ثابت، رجیسترهای MIPS را مقداردهی اولیه[1] کرد یا اینکه محتوای یک رجیستر را به داخل رجیستر دیگر کپی نمود:

```
addi $a0, $0, 2000   # $a0 = 0 + 2000 = 2000
add  $a1, $t0, $0    # $a1 = $t0 +0 = $t0
```

مثال: با استفاده از دستورات اسمبلی پردازنده MIPS محتوای دو کلمه اول از آرایه‌ای که از آدرس ۲۰۰۰ شروع می‌شوند را به ترتیب ۰ و ۲۳ قرار دهید.

جواب:

```
addi $a0, $0, 2000  # $a0 = 2000
sw   $0, 0($a0)     # M[2000] = A[0] = 0
addi $t0, $0, 23    # $t0 = 23
sw   $t0, 4($a0)    # M[2004] = A[1] = 23
```

[1] - Initialize

۲-۴- نمایش دستورالعمل‌ها در کامپیوتر

همه‌ی ما انسانها اجسامی که در اطراف‌مان قرار دارند را می‌شناسیم. به طور مثال اگر کسی یک خودکار، قلم، و یا کتاب را به ما نشان دهد و بپرسد: این چیست؟ بلافاصله اسم آن شیء را به او می-گوئیم. ما به این دلیل می‌توانیم این کار را انجام دهیم که در ذهن ما یک سری الگوهایی از اجسام شکل گرفته‌اند که این الگوها را در طول زمان به ما آموزش داده‌اند و ما با استفاده از آن الگوها می-توانیم اجسام را تشخیص دهیم و محیط اطراف خود را بشناسیم. یک کامپیوتر هم دقیقاً مانند انسان رفتار می‌کند و بر اساس یک سری الگوهایی که برایش تعریف شده است رفتار می‌کند. کاری که کامپیوتر انجام می‌دهد اجرای دستورات مختلف است و باید قبل از اجرای هر دستور تشخیص دهد که این، چه دستوری است. پس از اینکه بر اساس یک سری الگو نوع دستور را تشخیص داد، باید شیوه انجام دستور را نیز بشناسد. اگر نقشه ایران در اختیار شما قرار بگیرد و به شما گفته شود که از تهران به یک شهر مشخصی از ایران بروید، بسته به اینکه به کدام شهر بخواهید مسافرت کنید، مسیر حرکت شما نیز متغیر خواهد بود. هر دستور کامپیوتر هم به مانند یک نقشه، از قسمت‌های مختلفی تشکیل شده است و این قسمت‌های مختلف، روش اجرای دستور را تعیین خواهند کرد. به طور مثال اگر بر اساس الگوهای داخلی، نوع دستور جمع (add) تشخیص داده شود، گام بعدی این است که بر اساس قسمت‌های مختلف دستور، تشخیص داده شود که این دستور جمع، بر روی چه ورودی‌هایی انجام می‌شود (حافظه، رجیستر و یا عدد ثابت) و پس از انجام عملیات، نتیجه در کجا ذخیره می‌شود. اگر کامپیوتر، همه‌ی این تشخیص‌ها را انجام دهد، با استفاده از مداراتی که در داخل آن تعبیه شده است، می‌تواند آن دستور را به طور کامل اجرا کند.

در این بخش از کتاب و بخش‌های بعدی، بر روی الگوی دستورات MIPS تمرکز خواهیم نمود. در واقع اصلی‌ترین هدف این فصل از کتاب، معرفی همین الگوهاست. هر کامپیوتری الگوهای تعریف شده و مختص خودش را دارد و بر اساس همین الگوهاست که طراحی پردازنده‌ها انجام می‌شود. در فصل‌های بعدی که طراحی پردازنده MIPS انجام خواهد شد، استفاده این الگوها را خواهید دید.

در این بخش می‌خواهیم تفاوت بین دستوراتی که انسان به کامپیوتر می‌دهد را با دستوراتی که کامپیوتر می‌بیند توضیح دهیم. ابتدا مروری سریع بر چگونگی نمایش اعداد در کامپیوتر خواهیم داشت.

انسانها در امور روزمره مبنای ده را برای اعداد به کار می‌برند. اما اعداد را در هر مبنایی می‌توان نشان داد. به طور مثال عدد ۱۲۳ در مبنای ۱۰ برابر با 1111011 در مبنای ۲ است.

اعداد در سخت افزار کامپیوتر به صورت یک سری سیگنال با حالت بالا[1] و پایین[2] نمایش داده می‌شوند، بنابراین کامپیوتر اعداد را در مبنای ۲ در نظر می‌گیرد. می‌توانیم حالت بالا یا پایین را به صورت روشن یا خاموش، درست یا نادرست، و 1 یا 0 نیز در نظر بگیریم. دستورالعمل‌ها نیز در کامپیوتر به صورت یک سری سیگنالهای بالا و پایین ذخیره می‌شوند و می‌توانند به صورت عدد به نمایش درآیند. در حقیقت هر قسمت از یک دستورالعمل به صورت یک عدد مجزا در نظر گرفته می‌شود و از کنار هم قرار گرفتن این اعداد، دستورالعمل شکل می‌گیرد.

از آنجا که رجیسترها تقریبا بخشی از همهٔ دستورالعمل‌ها هستند، می‌توان به صورت قراردادی، نام‌های رجیستر را به اعداد نگاشت کرد. در زبان اسمبلی MIPS ، رجیسترهای $s0 تا $s7 به رجیسترهای ۱۶ تا ۲۳ و رجیسترهای $t0 تا $t7 به رجیسترهای ۸ تا ۱۵ نگاشت می‌شوند. بنابراین، $s0 به معنای رجیستر ۱۶، $s1 به معنای رجیستر ۱۷، $s2 به معنای رجیستر ۱۸ و ... ، $t0 به معنای رجیستر ۸، $t1 به معنای رجیستر ۹ و نظیر آن است. در بخش بعدی قرارداد مربوط به بقیه‌ی ۳۲ رجیستر را بیان خواهیم کرد.

مثال: می‌خواهیم دستور زبان اسمبلی MIPS زیر را به صورتی که کامپیوتر به آن نگاه می‌کند، نشان دهیم:

```
add  $t0, $s1, $s2
```

پاسخ: همان طور که در بالا اشاره شد، یک دستور از چند قسمت تشکیل می‌شود و هر قسمت از دستور و همچنین کل دستور به صورت عدد در حافظه کامپیوتر ذخیره می‌شوند. بر اساس همین اعداد هست که کامپیوتر راجع به کارهایی که می‌خواهد انجام دهد، تصمیم‌گیری می‌کند. برای بدست آوردن نمایش عددی این دستور، ابتدا دستور را به صورت ترکیبی از اعداد دهدهی و سپس اعداد دودویی نمایش می‌دهیم. نمایش دهدهی به صورت زیر است:

0	17	18	8	0	32

هر کدام از بخش‌های دستورالعمل یک *میدان*[3] نامیده می‌شود، اولین و آخرین میدان (که در این مثال شامل 0 و 32 هستند) به کامپیوتر MIPS می‌گویند که این دستورالعمل، عمل جمع را انجام می‌دهد. میدان دوم شمارهٔ رجیستری را که اولین عملوند مبدأ عملیات جمع است ($s1 = 17) به دست می‌دهد و سومین میدان، عملوند دیگر مبدأ را برای جمع نشان می‌دهد ($s2 = 18). میدان چهارم در بردارندهٔ شمارهٔ رجیستری است که حاصل جمع در آن قرار می‌گیرد ($t0 = 8)، میدان پنجم در این دستورالعمل

[1] - High
[2] - Low
[3] - Field

بدون استفاده است، بنابراین برابر با صفر قرار داده شده است. بدین ترتیب این دستورالعمل رجیستر $s1 را به رجیستر $s2 می‌افزاید و حاصل جمع را در رجیستر $t0 قرار می‌دهد.

این دستورالعمل را می‌توان به صورت میدان‌هایی با اعداد دودویی به جای اعداد دهدهی، به صورت زیر نمایش داد:

000000	10001	10010	01000	00000	100000
۶ بیت	۵ بیت	۵ بیت	۵ بیت	۵ بیت	۶ بیت

برای تمایز با زبان اسمبلی، نوع عددی دستورالعمل را زبان ماشین، و ترتیبی از چنین دستورالعمل‌هایی را کد ماشین می‌نامیم. ترکیب عددی فوق برای دستورالعمل، قالب[1] دستورالعمل نامیده می‌شود.

تعداد بیت‌های یک دستورالعمل در MIPS دقیقاً ۳۲ بیت است که با اندازهٔ یک کلمهٔ داده مساوی است. همهٔ دستورالعمل‌های MIPS برای رعایت اصل طراحی شماره ۱ (سادگی به قاعده مندی کمک می‌کند)، ۳۲ بیتی هستند.

به نظر می‌آید که باید آمادهٔ نوشتن و خواندن رشته‌های بلند و خسته کننده اعداد دودویی شوید، اما با استفاده از مبنایی بالاتر از دودویی که به راحتی به دودویی تبدیل می‌شود، می‌توان از این کار خسته کننده، دوری ورزید. از آنجا که تقریباً اندازهٔ داده در همهٔ کامپیوترها مضربی از ۴ است، اعداد شانزده شانزدهی[2] (مبنای ۱۶) متداول هستند. چون مبنای ۱۶ توانی از ۲ است، با تعویض هر گروه چهارتایی ارقام دودویی با یک رقم شانزده شانزدهی و برعکس، تبدیل‌های این دو مبنا را به یکدیگر، انجام می‌دهیم. به طور مثال $(63)_{16} = (01100011)_2$ و $(17)_{16} = (010111)_2$.

از آنجا که از مبناهای مختلف اعداد بطور مرتب استفاده می‌کنیم، برای جلوگیری از اشتباه، اعداد دودویی را با زیرنویس 2 یا two و شانزده شانزدهی را با زیرنویس 16 یا hex مشخص می‌کنیم. اگر زیرنویس وجود نداشته باشد، منظور مبنای ۱۰ است. لازم به ذکر است که زبانهای برنامه سازی C و جاوا از نماد 0xnnnn برای اعداد شانزده شانزدهی استفاده می‌کنند. به طور مثال عدد 0x12 نشان دهنده عدد ۱۲ در مبنای ۱۶ است که مساوی ۱۸ در مبنای ۱۰ است: 0x12 = (12)16 = 18

میدان‌های MIPS

برای ساده‌تر شدن بحث روی میدان‌های MIPS ، به آن‌ها نامهایی به صورت زیر داده می‌شود:

op	rs	rt	rd	shamt	funct
۶ بیت	۵ بیت	۵ بیت	۵ بیت	۵ بیت	۶ بیت

[1] - Format
[2] - Hexadecimal

معنای هر کدام از میدان‌های دستورالعمل‌های MIPS به صورت زیر است:

- op: از کلمه‌ی operation گرفته شده است. این میدان، عملیات اصلی دستورالعمل، که معمولاً *کد عمل*[1] نامیده می‌شود را مشخص می‌کند.
- rs : اولین عملوند مبدأ است که یک رجیستر است.
- rt: دومین عملوند مبدأ است که یک رجیستر است.
- Rd: عملوند مقصد است که نتیجهٔ عملیات را نگه می‌دارد. این عملوند نیز یک رجیستر است.
- Shant: این میدان، مقدار شیفت را در دستورات نوع شیفت مشخص می‌کند. (دستورالعمل‌های شیفت در بخش ۲-۵ بررسی می‌شوند) فعلاً از این میدان استفاده نمی‌شود، بنابراین حاوی صفر است).
- Funct: از کلمه‌ی function یا عملکرد گرفته شده است. این میدان در کنار میدان op یک عملیات را مشخص می‌کند. در واقع این میدان، نوع خاصی از عملیات را در میدان op انتخاب می‌کند.

هنگامی که یک دستورالعمل به میدان‌های بیشتری از میدان‌های نشان داده شده، نیاز داشته باشد، مشکل پیش می‌آید. برای مثال یک دستورالعمل بار کردن کلمه مانند lw $t0, 247($s2) ، باید دو رجیستر و یک ثابت را مشخص کند. اگر آدرس از یکی از میدان‌های ۵ بیتی قالب بالا استفاده کند، مقدار ثابت موجود در دستورالعمل بار کردن کلمه، نمی‌تواند مقداری بیشتر از 2^5 یا ۳۲ داشته باشد. از آنجایی که این ثابت برای انتخاب عناصر آرایه‌ها به کار می‌رود و بعضاً تعداد عناصر آرایه بیشتر از ۳۲ است، بنابراین به نظر می‌رسد که تعداد ۵ بیت برای مشخص کردن آدرس کافی نباشد. اگر ما بخواهیم برای دستورات حافظه‌ای load و store از همان قالب قبلی با همان تعداد میدان استفاده کنیم، در این صورت مجبور خواهیم بود که یکی از میدان‌ها را به دلیل عدد ثابت در دستورهای انتقال حافظه بزرگتر از ۵ بیت در نظر بگیریم که این باعث خواهد شد تا طول دستورالعمل از ۳۲ بیت بیشتر شود. چون یکی از اهداف MIPS ثابت نگه داشتن طول دستورالعمل است، به همین دلیل برای ثابت نگه داشتن طول دستور مجبور شد قالب دیگری را که متفاوت از قالب قبلی است برای دستورات انتقال حافظه‌ای به کار برد. داشتن تعداد قالب‌های اضافه برای یک پردازنده ویژگی خوبی نیست ولی بعضی مواقع نظیر حالتی که توضیح داده شد، مجبور هستیم قالب اضافه کنیم. یک پردازنده برای اینکه بتواند دستورهای

[1] - Opcode

مختلف با ویژگیهای متفاوت را اجرا کند مجبور به تن دادن به قالب‌های اضافه در برابر از دست دادن بعضی از ویژگیهای خوب است. البته این یکی از اصول طراحی سخت افزار است:

اصل شماره ۴ طراحی: طراحی خوب به مصالحهٔ خوب نیاز دارد.

مصالحهٔ انتخابی طراحان MIPS هم طول نگهداشتن همهٔ دستورالعملهاست، بنابراین برای هر نوع دستورالعمل باید قالب متفاوتی وجود داشته باشد. برای مثال، قالب نشان داده شده در بالا، نوع R (برای رجیستر) یا قالب R نامیده می‌شود. نوع دوم قالب دستورالعمل، نوع I (برای فوری) یا قالب I نامیده می‌شود و برای دستورالعمل‌های انتقال داده و فوری به کار می‌رود. میدان‌های قالب I به صورت زیر است:

op	rs	rt	ثابت یا آدرس
۶ بیت	۵ بیت	۵ بیت	۱۶ بیت

آدرس ۱۶ بیتی بدان معناست که دستورالعمل انتقال حافظه‌ای می‌تواند به هر بایتی که در ناحیه $(rs-2^{15}, rs+2^{15})$ و یا به هر کلمه‌ای که در ناحیه $(rs-2^{13}, rs+2^{13})$ قرار دارد، دسترسی داشته باشد. بطور مشابه، در عملیات جمع فوری (addi) ، یک رجیستر با یک ثابت جمع می‌شود که اگر از قالب I استفاده کنیم، این ثابت‌ها در بازه‌ی $2^{15}+$ و $2^{15}-$ قرار دارند. همان طور که دیده می‌شود، در این قالب، داشتن بیشتر از ۳۲ رجیستر، دشوار است، چون میدان‌های rs و rt هر کدام به یک بیت دیگر نیاز دارند که جای دادن آن را در یک کلمه مشکل‌تر می‌کند. بیایید یک بار دیگر به دستورالعمل بار کردن کلمه نگاه کنیم:

```
lw  $t0, 32($s3)  #  $t0 = A[8]
```

در اینجا ۱۹ (برای $s3) در میدان rs ، ۸ (برای $t0) در میدان rt و ۳۲ در میدان آدرس قرار دارد.

35	19	8	32
۶ بیت	۵ بیت	۵ بیت	۱۶ بیت

توجه کنید که معنای میدان rt در این دستوالعمل تغییر کرده است: در یک دستورالعمل بار کردن کلمه، میدان rt ، رجیستر مقصد را مشخص می‌کند که نتیجه بار کردن را در خود نگه می‌دارد.

اگرچه قالب‌های چندگانه، سخت‌افزار را پیچیده می‌کنند، اما می‌توانیم با شبیه کردن قالب‌ها از پیچیدگی سخت‌افزار بکاهیم. بطور مثال سه میدان اول قالب‌های نوع R و نوع I اندازه و نام یکسان دارند و چهارمین میدان در نوع I با طول سه میدان آخر نوع R برابر است. اگر نگران تمایز این قالب‌ها هستید باید بگوییم که، قالب‌ها با مقادیر اولین میدان متمایز می‌شوند: به هر قالب، مجموعه‌ای متمایز از

مقادیر در میدان اول (op) منتسب می‌شود بطوری که سخت‌افزار بداند که چگونه با نیمهٔ دوم دستورالعمل به عنوان سه میدان (نوع R) یا یک میدان (نوع I) رفتار کند. شکل ۳ اعداد به کار رفته در هر میدان را برای دستورالعمل‌های MIPS که در بخش ۲-۳ بیان شده‌اند، نشان می‌دهد.

دستورالعمل	قالب	op	rs	rt	rd	shamt	funct	آدرس
add (جمع)	R	0	reg	reg	reg	0	32	n.a.
sub(تفریق)	R	0	reg	reg	reg	0	34	n.a.
addi (جمع فوری)	I	8	reg	reg	n.a.	n.a.	n.a.	ثابت
lw (بار کردن کلمه)	I	35	reg	reg	n.a.	n.a.	n.a.	آدرس
sw (ذخیره کردن کلمه)	I	43	reg	reg	n.a.	n.a.	n.a.	آدرس

شکل ۳: کدگذاری دستورات MIPS : در جدول بالا، reg به معنای شماره رجیستر بین ۰ تا ۳۱، آدرس به معنای آدرس ۱۶ بیتی و n.a. به معنای ظاهر نشدن این میدان در این قالب است. توجه کنید که دستورالعمل‌های add و sub دارای مقدار یکسان در میدان op هستند. سخت‌افزار برای تصمیم‌گیری در مورد نوع دستورالعمل از funct استفاده می‌کند، برای جمع از (32) و برای تفریق از (34).

توجه: اگر در قالب‌های نوع R و I دقت کنید، متوجه می‌شوید که برای هر میدان رجیستر، ۵ بیت در نظر گرفته شده است. دلیل این امر این است که پردازنده MIPS دارای ۳۲ رجیستر می‌باشد و برای مشخص کردن ۳۲ عدد، ۵ بیت مورد نیاز است. اما اگر تعداد رجیسترها بیشتر از ۳۲ باشد، برای میدان رجیستر باید بیشتر از ۵ بیت در نظر گرفت که این کار باعث می‌شود که طول دستور بزرگتر شود و بیشتر از ۳۲ بیت شود. بنابراین همان طور که قبلاً هم اشاره شده بود، تعداد رجیسترها، طول دستور را تحت تأثیر قرار می‌دهند.

مثال: در این مثال می‌خواهیم رابطه بین آنچه که برنامه نویس می‌نویسد و آنچه که کامپیوتر اجرا می‌کند را ارائه کنیم. اکر $t1 آدرس پایه‌ی آرایه A را داشته باشد و $s2 متناظر با h باشد، عبارت انتساب زیر:

A[300]= h + A[300]

به صورت زیر کامپایل می‌شود:

```
lw   $t0, 1200($t1)  #  $t0 = A[300]
add $t0, $s2, $t0    #  $to =  h + A[300]
sw  $t0, 1200($t1)  #   A[300] = h + A[300]
```

کد زبان ماشین MIPS برای این سه دستوالعمل را بدست آورید؟

پاسخ: برای راحتی ابتدا دستورالعمل‌های زبان ماشین را با اعداد دهدهی نمایش می‌دهیم. از شکل ۳ می‌توانیم سه دستورالعمل زبان ماشین را به صورت زیر تعیین کنیم:

دستورالعمل lw با 35 در اولین میدان (op) مشخص می‌شود. بنابراین برای دستوالعمل lw ، عدد ۳۵ در اولین میدان (op) ، رجیستر پایه ۹ (st1) در دومین میدان (rs)، و رجیستر مقصد ۸ ($t0) در میدان سوم (rt) قرار می‌گیرد. آفست ۱۲۰۰ نیز در آخرین میدان (آدرس) قرار می‌گیرد.

دستورالعمل جمع، با 0 در اولین میدان (op) و ۳۲ در آخرین میدان (funct) مشخص می‌شود. سه عدد 18، 8 و 8 به ترتیب در میدان‌های دوم، سوم و چهارم قرار می‌گیرند و به ترتیب متناظر با عملوندهای نوع رجیستر $s2، $t0 و $t0 هستند.

دستورالعمل sw با 43 در اولین میدان مشخص می‌شود. بقیهٔ میدان‌های این دستورالعمل، معادل دستوالعمل lw می‌باشد.

پس شکل دهدهی دستورات به صورت زیر است:

op	rs	rt	address		
			rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

معادل دودویی این شکل دهدهی به صورت زیر است (۱۲۰۰ در مبنای ده برابر با 0000 1011 0100 0000 در مبنای دو می‌باشد).

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

به شباهت نمایش دودویی اولین و آخرین دستورالعمل توجه کنید. تنها اختلاف در بیت سوم از سمت چپ می‌باشد. چنانچه در فصل ۵ و ۶ خواهیم دید، شباهت نمایش دستوالعمل‌های مرتبط، طراحی سخت‌افزار را آسان می‌کند. این دستوالعمل‌ها مثالی دیگر از قاعده‌مندی معماری MIPS هستند.

قسمتهایی از زبان اسمبلی MIPS که تا این قسمت بررسی شده است، در شکل ۴ خلاصه شده است.

زبان اسمبلی MIPS

دسته	دستورالعمل	مثال	معنی	توضیحات
حسابی	جمع	add $s1, $s2, $s3	$s1 = $s2 + $s3	دارای سه عملوند رجیستری
	تفریق	sub $s1, $s2, $s3	$s1 = $s2 - $s3	دارای سه عملوند رجیستری
انتقال داده	بارکردن کلمه	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	انتقال داده از حافظه به رجیستر
	ذخیره کردن کلمه	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	انتقال داده از رجیستر به حافظه

زبان ماشین MIPS

نام	قالب	مثال						توضیحات
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
addi	I	8	18	17	100			addi $s1, $s2, 100
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
اندازه‌ی میدان		۶ بیت	۵ بیت	۵ بیت	۵ بیت	۵ بیت	۶ بیت	همه دستورات MIPS ، ۳۲ بیتی هستند
قالب R	R	op	rs	rt	rd	shamt	funct	قالب دستورالعمل حسابی
قالب I	I	op	rs	rt	address (constant)			قالب دستورالعمل انتقال داده

شکل ۴: جدول پایین ساختارهای زبان ماشین MIPS معرفی شده در بخش ۲-۴ را نشان می‌دهد. قالب‌های دستوالعمل MIPS بررسی شده تا کنون، R و I هستند. ۱۶ بیت اول یکسان می‌باشند: هر دو شامل میدان op هستند که عملیات پایه را مشخص می‌کند، یک میدان rs که یکی از عملوندهای مبدأ و میدان rt که عملوند دیگر مبدأ را نشان می‌دهد مگر در مورد بارکردن کلمه که رجیستر مقصد را مشخص می‌کند. قالب R ، ۱۶ بیت آخر را به میدان rd که مشخص کننده رجیستر مقصد، میدان shamt که در بخش ۲-۵ بررسی می‌شود و میدان funct که عملیات خاص دستوالعمل قالب R را مشخص می‌کند، تقسیم می‌نماید. قالب I ، ۱۶ بیت آخر را به عنوان یک میدان آدرس و یا به عنوان یک عدد ثابت استفاده می‌کند.

خودآزمایی: چرا MIPS دستورالعمل تفریق فوری ندارد؟

پاسخ: دو دلیل وجود دارد:

۱. ثابت‌های منفی خیلی کم در C و جاوا ظاهر می‌شوند، بنابراین عمومی نبوده و پشتیبانی خاصی از آنها نمی‌شود.

۲. از آنجا که میدان فوری ثابت‌های منفی و مثبت را نگه می‌دارد، جمع فوری با عدد منفی معادل تفریق فوری با عدد مثبت است، بنابراین به تفریق فوری نیازی نیست.

۲-۵- عملیات منطقی

هرکامپیوتری علاوه بر عملیات ریاضی، قادر است عملیات منطقی را نیز انجام دهد. عملیات منطقی بر روی تک تک بیت‌های یک کلمه انجام می‌شود. به طور مثال وقتی می‌خواهیم عملیات NOT منطقی را انجام دهیم، این عملیات هر کدام از بیت‌های کلمه را معکوس می‌کند. همین طور وقتی که می‌خواهیم عملیات AND را بر روی دو کلمه انجام دهیم، بیت‌های هم رتبه این دو کلمه نظیر به نظیر با هم AND می‌شوند. عملیات منطقی در زبان برنامه سازی C و دستورات معادل آنها در ماشین MIPS ، در شکل ۵ نشان داده شده است.

عملیات منطقی	عملگرهای C	دستورالعمل‌های MIPS
شیفت به چپ	<<	sll
شیفت به راست	>>	srl
AND بیت به بیت	&	and, andi
OR بیت به بیت	|	or, ori
NOT بیت به بیت	~	nor

شکل ۵: عملگرهای منطقی C و دستورات MIPS معادل آنها

اولین دسته از عملیات منطقی نشان داده شده، عملیات شیفت می‌باشد. این عملیات، تمامی بیت‌های یک کلمه را به سمت چپ یا راست جابجا کرده و جاهای خالی را با 0 پر می‌کند. به طور مثال اگر رجیستر $s0 دارای مقدار زیر باشد:

$$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001)_2 = 9$$

و آن را ۴ بار شیفت به چپ دهیم، مقدار زیر بدست می‌آید:

$$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000)_2 = 144$$

در پردازنده MIPS نام دستورالعمل شیفت به چپ منطقی sll[1] و نام دستورالعمل شیفت به راست منطقی srl[2] می‌باشد. دستورالعمل زیر عملیات بالا را با فرض اینکه نتیجه در رجیستر $t2 قرار می‌گیرد، انجام می‌دهد:

```
sll    $t2, $s0, 4      #  $t2 = $s0 << 4
```

[1] - Shift Left Logical
[2] - Shift Right Logical

در فرمت نوع R ، میدان shamt مقدار شیفت را نشان می‌داد و گفتیم که کاربرد آن در دستورات شیفت است. دستورالعمل‌های sll و srl از فرمت نوع R می‌باشند. نسخه زبان ماشین دستور sll که در بالا توضیح داده شد به صورت زیر است:

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

همان طور که دیده می‌شود در میدان‌های op و funct ، 0 ، در میدان rd ، $t2 ، در میدان rt ، $s0 و در میدان شیفت مقدار 4 قرار می‌گیرد. در این دستور از میدان rs استفاده نمی‌شود و به همین دلیل مقدار 0 در آن قرار گرفته است.

شیفت به چپ منطقی در محاسبات، اهمیت زیادی دارد، به دلیل اینکه شیفت به چپ به اندازه i بیت باعث ضرب عدد در 2^i می‌شود (علت این امر در فصل ۳ توضیح داده شده است). برای مثال بالا، عدد ۴ بار به سمت چپ شیفت می‌یابد که باعث می‌شود در 2^4 یا 16 ضرب شود. بنابراین نتیجه عملیات می‌شود: $9 \times 16 = 144$.

شیفت به راست منطقی نیز اهمیت زیادی دارد. در این عملیات شیفت به اندازه i بیت باعث تقسیم عدد در 2^i می‌شود.

از عملیات منطقی دیگر که در شکل ۵ نشان داده شده است، عملیات AND می‌باشد. AND عملیات بیت به بیت را انجام داده و نتیجه آن برای هر بیت خروجی زمانی 1 است که هر دو بیت متناظر ورودی 1 باشد. به طور مثال اگر رجیستر $t2 دارای مقدار زیر باشد:

$$(0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000)_2$$

و رجیستر $t1 دارای مقدار زیر باشد:

$$(0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000)_2$$

در این صورت، پس از اجرای دستورالعمل AND زیر:

```
and   $t0, $t1, $t2   #   $t0 = $t1 & $t2
```

مقدار رجیستر $t0 به صورت زیر خواهد بود:

$$(0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000)_2$$

همان طور که دیده می‌شود، می‌توان با استفاده از عملیات AND ، یک الگوی بیتی را به گروهی از بیت‌ها اعمال نمود. در محل بیت‌هایی که می‌خواهیم 0 اعمال کنیم، بیت‌ها را با بیت‌های 0 ، AND

می‌کنیم. در این صورت گفته می‌شود که آن بیت‌ها پوشش[1] یافته‌اند. چون پوشش برخی از بیت‌ها را مخفی می‌کند.

عملیات منطقی دیگر که دوگان عملیات AND است، OR می‌باشد. OR نیز به صورت بیت به بیت انجام می‌گیرد و نتیجه‌ی آن برای هر بیت، زمانی 1 است که هردو و یا یکی از بیت‌های متناظر 1 باشد. با فرض اینکه در مثال قبلی، رجیسترهای $t1 و $t2 تغییر نکنند، دستورالعمل OR زیر:

```
or   $t0, $t1, $t2   #  $t0 = $t1 | $t2
```

باعث می‌شود مقدار زیر در رجیستر $t0 قرار گیرد:

$$(0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 0000\ 0000)_2$$

آخرین عملیات شکل ۵ ، عملیات NOT است. در عملیات NOT ، هر کدام از بیت‌ها معکوس می‌شوند، یعنی اگر 0 باشد، تبدیل به 1 و اگر 1 باشد، تبدیل به 0 می‌شود. طراحان MIPS ، برای اینکه از قالب دستورات موجود استفاده کنند، تصمیم گرفتند از دستورالعمل NOR (NOT OR) به جای NOT استفاده کنند. ایده این بود که اگر در عملیات NOR ، یکی از عملوندها صفر باشد، عملیات NOR ، معادل NOT خواهد شد:

$$A\ NOR\ 0 = NOT\ (A\ OR\ 0) = NOT\ (A)$$

با فرض اینکه رجیستر $t1 مثال قبل تغییر نکرده باشد و رجیستر $t3 دارای مقدار 0 باشد، در این صورت، نتیجه دستورالعمل زیر:

```
nor   $t0, $t1, $t3   #  $t0 = ~ ($t1 | $t3)
```

به صورت زیر خواهد بود که در رجیستر $t0 قرار می‌گیرد:

$$(1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111)_2$$

مقادیر ثابت، علاوه بر عملیات حسابی، در عملیات منطقی AND و OR نیز مفیدند، به همین دلیل، MIPS دارای دستورالعمل‌های and فوری[2] (andi) و or فوری[3] (ori) نیز می‌باشد. مقادیر ثابت، در عملیات NOR کاربردی ندارد، به دلیل اینکه NOR برای معکوس کردن بیت‌های یک عملوند به کار می‌رود، بنابراین MIPS ، دستور NOT فوری ندارد. در شکل ۶ مجموعه دستورات MIPS ، که تاکنون بررسی شده‌اند، لیست شده است. همان طور که دیده می‌شود، ما این دستورات را در سه دسته اصلی حسابی، منطقی و انتقال داده، دسته بندی کرده‌ایم.

---

[1] - Mask
[2] - AND immediate
[3] - OR immediate

دسته	دستورالعمل	مثال	معنی	توضیحات
حسابی	جمع	add $s1, $s2, $s3	$s1 = $s2 + $s3	دارای سه عملوند رجیستری
	تفریق	sub $s1, $s2, $s3	$s1 = $s2 - $s3	دارای سه عملوند رجیستری
	جمع فوری	add $s1, $s2, 100	$s1 = $s2 + 100	جمع رجیستر با ثابت
منطقی	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	دارای سه عملوند رجیستری
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	دارای سه عملوند رجیستری
	nor	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)	دارای سه عملوند رجیستری
	andi	andi $s1, $s2, 100	$s1 = $s2 & 100	AND رجیستر با ثابت
	ori	ori $s1, $s2, 100	$s1 = $s2 | 100	OR رجیستر با ثابت
	شیفت به چپ منطقی	sll $s1, $s2, 10	$s1 = $s2 << 10	شیفت به چپ به اندازه ثابت
	شیفت به راست منطقی	srl $s1, $s2, 10	$s1 = $s2 >> 10	شیفت به راست به اندازه ثابت
انتقال داده	بارکردن کلمه	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	انتقال از حافظه به رجیستر
	ذخیره کردن کلمه	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	انتقال از رجیستر به حافظه

شکل ۶: مجموعه دستورات MIPS که تاکنون بررسی شده‌اند.

۳-۶- دستورالعمل‌های تصمیم‌گیری

یکی از تفاوتهایی که کامپیوتر با وسایل محاسباتی معمولی دارد، توانایی آن در تصمیم گیری است. با دستورات تصمیم گیری، کامپیوتر می‌تواند انتخاب کند که کاری را انجام دهد و یا انجام ندهد. همچنین می‌توان به وسیله این دستورات، حلقه ایجاد کرد و دسته‌ای از دستورات را چند بار تکرار نمود. تصمیم‌گیری در زبانهای برنامه سازی سطح بالا به کمک if و دستورهای حلقه انجام می‌شود. زبان اسمبلی MIPS ، دو دستور تصمیم گیری معروف به نام‌های beq و bne دارد. البته در MIPS، دستورات دیگری نیز برای تصمیم گیری وجود دارد، که ما در این مبحث آنها را بررسی نمی‌کنیم و فقط به beq و bne بسنده می‌کنیم. دستور beq به صورت زیر است:

beq register1, register2, Label

beq[1] یعنی "پرش در صورت مساوی بودن". این دستورالعمل محتوای دو رجیستر (در اینجا register1 و register2) را با هم مقایسه می‌کند و در صورتی که مساوی باشند به آدرس برچسب[2] مشخص شده در دستور، پرش می‌کند.

دستور bne به صورت زیر نوشته می‌شود:

```
bne  register1, register2, Label
```

bne[3] یعنی "پرش در صورت نامساوی بودن". این دستورالعمل محتوای دو رجیستر (در اینجا register1 و register2) را با هم مقایسه می‌کند و در صورتی که مساوی نباشند به آدرس برچسب مشخص شده در دستور، پرش می‌کند.

دستورهای beq و bne را معمولاً دستورهای پرش یا انشعاب شرطی می‌نامند، چون پرش آنها مبتنی بر یک عبارت شرطی است که ممکن است نتیجه آن درست باشد و یا غلط. در صورت برقراری شرط، پرش انجام می‌شود. در مقابل دستورات پرش شرطی، دستورات پرش غیر شرطی قرار دارند که بدون در نظر گرفتن شرطی، همیشه پرش را انجام می‌دهند.

مثال: در قطعه کد زیر، اگر پنج متغیر f ، g ، h ، i و j ، به ترتیب متناظر با پنج رجیستر $s0 تا $s4 باشند، در این صورت کد کامپایل شده عبارت if زیر را بدست آورید:

```
if (i == j)  f = g + h;
   else  f = g – h;
```

پاسخ: کد کامپایل شده به صورت زیر است:

```
      bne  $s3, $s4, Else   # goto Else if i != j
      add  $s0, $s1, $s2    # f = g + h
      j      Exit           # goto Exit
Else: sub   $s0, $s1, $s2   # f = g - h
Exit:
```

در این عبارت ما به جای شرط مساوی، نامساوی بودن را بررسی کرده‌ایم. عموماً اگر شرط مخالف را برای عملیاتی که در بخش then مربوط به if قرار دارد، چک کنیم، کد مؤثرتری بدست می‌آید. در این کد، عملیات قسمت then (f = g + h) را بعد از دستور bne نوشته‌ایم و اگر شرط bne برقرار نباشد (دو رجیستر $s3 و $s4 نامساوی نباشند و یا به عبارتی مساوی باشند)، اجرا خواهد شد (دستور add $s0, $s1, $s2). پس از این دستور، دستور j Exit قرار دارد. دستور j ، دستور پرش غیر شرطی است که باعث می‌شود بدون هیچ شرطی، پرش انجام شود. دلیل استفاده از j در این قسمت این است که

---

[1] - Branch if equal
[2] - Label
[3] - Branch if not equal

اگر عملیات قسمت then انجام گرفت، عملیات قسمت else نباید انجام گیرد. دستور j باعث می‌شود که به قسمت Exit این کد منتقل شویم و عملیات قسمت else اجرا نشوند. برچسب Exit نشان دهنده پایان این قطعه کد است. اما اگر شرط bne برقرار باشد، به برچسب Else پرش انجام می‌شود که در آن عملیات قسمت else نوشته شده است (sub $s0, $s1, $s2). پس از اجرای این دستور، قسمت Exit اجرا خواهد شد و مشخص است که دیگر قسمت مربوط به then انجام نخواهد شد.

تلاقی سخت افزار و نرم افزار: کامپایلرها معمولاً دستورهای پرش و برچسب‌هایی تولید می‌کنند که در زبان‌های برنامه نویسی ظاهر نمی‌شوند. عدم نوشتن آشکار دستورهای پرش و برچسب‌ها، یکی از مزایای زبانهای برنامه‌نویسی سطح بالا و از دلایل ساده‌تر بودن آنهاست.

مثال: یک حلقه while در زبان برنامه نویسی C را به صورت زیر در نظر بگیرید:

```
while (A[i] == k)
    i += 1;
```

با فرض اینکه i و k به رجیسترهای $s3 و $s5 منتسب شده باشند و آدرس شروع آرایه A در $s6 قرار گرفته باشد، کد اسمبلی MIPS مربوط به این قطعه کد را بدست آورید. فرض کنید عناصر آرایه ۳۲ بیتی هستند.

پاسخ: کد اسمبلی به صورت زیر است:

```
Loop: sll   $t1, $s3, 2    #  $t1 = 4 * i
      add   $t1, $t1, $s6  #  $t1 = 4 * i  + $s6
      lw    $t0, 0($t1)    #  $t0 = A[i]
      bne   $t0, $s5, Exit #  if A[i] != k then goto Exit
      add   $s3, $s3, 1    #  i = i + 1
      j     Loop           #  goto Loop
Exit:
```

در این کد نیز به جای شرط مساوی از شرط نامساوی استفاده کرده‌ایم. ابتدا باید A[i] را از حافظه بخوانیم و بعد مقایسه را انجام دهیم. قبل از خواندن از حافظه نیز باید آدرس A[i] را بدست آورد. از بخش ۳-۳ می‌دانیم که برای بدست آوردن آدرس A[i] باید 4 * i را با آدرس پایه (آدرس شروع) آرایه جمع کنیم. با دو دستور اول آدرس را محاسبه کرده‌ایم. برای ضرب کردن i در ۴ از شیفت منطقی به چپ به اندازه ۲ استفاده شده است چون هر شیفت منطقی به چپ عدد را در ۲ ضرب می‌کند، پس با دوبار شیفت، عدد در ۴ ضرب می‌شود. دستور سوم کد اسمبلی A[i] را به داخل $t0 منتقل کرده است. دستور چهارم bne می‌باشد که شرط حلقه را چک می‌کند. اگر شرط نامساوی درست باشد، می‌دانیم که

باید از حلقه خارج شویم و در این کد نیز این کار انجام شده است و bne در صورت برقراری شرط به برچسب Exit پرش را انجام می‌دهد. اگر پرش صورت نگیرد، دستور بعد از bne که در واقع عملیات بدنه حلقه while است، اجرا خواهد شد (add $s3, $s3, 1 # i = i + 1). پس از انجام این عملیات، دوباره باید به اول حلقه برگردیم و شرط حلقه را دوباره چک کنیم.

تعریف: به هر کدام از قطعه کدهای اسمبلی مثال‌های if و حلقه while ، یک بلوک اصلی دستورالعمل گفته می‌شود. در واقع بلوک‌های اصلی ترتیبی از دستورالعمل‌ها هستند که داخل آنها دستور پرش وجود ندارد، مگر اینکه دستور پرش آخرین دستور بلوک باشد. همچنین در این ترتیب دستورالعمل، مقصد پرش یا همان برچسب‌های پرش وجود ندارد، مگر اینکه برچسب در اولین دستور بلوک باشد.
یکی از اولین مراحل کامپایل، شکستن برنامه به این بلوک‌های اصلی است. احتمالاً یکی از معمولترین مقایسه‌ها، مقایسه تساوی یا عدم تساوی است. اما بعضی از مواقع لازم است که مقایسه بزرگتر یا کوچکتر بودن را نیز انجام دهیم. به طور مثال در یک حلقه ممکن است شرط اینکه یک متغیر فرضاً از 0 کوچکتر است یا نه، را برای ورود به حلقه چک کنیم. این مقایسه‌ها در زبان اسمبلی MIPS به کمک دستورالعملی به نام slt[1] انجام می‌شود. این دستورالعمل، دو رجیستر را مقایسه کرده و در صورت کوچکتر بودن اولی از دومی، رجیستر سوم را 1 و در غیر این صورت، 0 می‌کند. به طور مثال، دستورالعمل زیر را در نظر بگیرید:

```
slt   $t0, $s3, $s4
```

در این دستور، رجیسترهای $s3 و $s4 مقایسه می‌شوند. اگر $s3 کوچکتر از $s4 باشد، مقدار 1 ، و در غیر این صورت مقدار 0 داخل رجسیتر $t0 قرار می‌گیرد.
استفاده از عملوندهای ثابت در مقایسه‌ها، رایج است. در پردازنده MIPS چون محتوای رجیستر شماره صفر ($0 یا $zero)، همیشه 0 است، بنابراین می‌توانیم مقایسه با مقدار ثابت صفر را با استفاده از دستورهای beq و bne ، به راحتی انجام دهیم (یکی از رجیسترها را رجیستر $0 در نظر می‌گیریم).
برای مقایسه با مقادیر ثابت دیگر، می‌توان از یک دستورالعمل دیگر به نام slti که شبیه slt است، در کنار beq و bne استفاده نمود. در دستور slti ، یک رجیستر با یک مقدار ثابت مقایسه می‌شوند و بر اساس نتیجه این مقایسه، یک رجیستر دیگر 0 یا 1 می‌شود. به طور مثال در دستور زیر رجیستر $s2 با مقدار ثابت 10 مقایسه می‌شود:

```
slti  $t0, $s2, 10     #  if ($s2 < 10) then $t0 = 1 else $t0 = 0
```

[1] - Set on less than

نکته: با توجه به اصل طراحی شماره ۱ که می‌گفت «سادگی به نظم کمک می‌کند»، معماری MIPS دستور «پرش در صورت کوچکتر بودن» ندارد. چون این دستور در صورت استفاده، یا CPI بالاتری خواهد داشت و یا اینکه پریود کلاک را افزایش خواهد داد. بنابراین به جای استفاده از این دستور نسبتاً پیچیده، بهتر است که از دو دستور ساده‌تر استفاده کنیم. برای پشتیبانی از دستوری نظیر «پرش در صورت کوچکتر بودن»، ممکن است تغییراتی در سخت افزار ایجاد کنیم که زمان اجرای بقیه دستورات و در نتیجه زمان اجرای کل برنامه را نیز تحت تأثیر خود قرار دهد. این گونه تغییرات به هیچ عنوان قابل قبول نیستند.

مثال: پس از اجرای قطعه کد زیر، محتوای $v0 را پیدا کنید:

```
addi  $t0, $0, 20
addi  $t1, $0, 50
slt  $v0, $t0, $t1
```

پاسخ: در این قطعه کد، دستور اول مقدار 20 را داخل رجیستر $t0 قرار می‌دهد. دستور دوم مقدار 50 را داخل رجیستر $t1 قرار می‌دهد. دستور سوم بررسی می‌کند که آیا $t0 از $t1 کوچکتر است یا نه. در صورت برقراری شرط و کوچکتر بودن، مقدار 1 و در غیر این صورت، مقدار 0 را داخل $v0 قرار می‌دهد. در این مثال چون $t0<$t1 (20<50) است، بنابراین مقدار 1 داخل $v0 قرار خواهد گرفت.

مثال: پس از اجرای قطعه کد زیر محتوای $v0 را پیدا کنید:

```
addi  $t0, $0, 20
slti  $v0, $t0, 50
```

پاسخ: این مثال، شبیه مثال قبلی است، با این تفاوت که دستور دوم از آن حذف شده و مقایسه رجیستر $t0 با عدد 50 با استفاده از دستور slti به جای slt انجام شده است. در دستور slti، می‌توان یک رجیستر را با یک مقدار ثابت، مقایسه کرد. در این مثال، $t0 با عدد ثابت 50 مقایسه شده و چون $t0<50 (20<50) است، بنابراین مقدار 1 داخل $v0 قرار خواهد گرفت.

توجه: دستورهای beq و bne ، به دلیل اینکه دارای دو رجیستر و یک عدد ثابت یا آدرس، در شکل دستور می‌باشند، از قالب نوع I می‌باشند. دستور slti نیز به همین دلیل از قالب نوع I می‌باشد. اما دستور slt به دلیل اینکه دارای سه رجیستر در شکل دستور است، از قالب نوع R محسوب می‌شود.

در شکل ۷ دستورهایی از معماری MIPS ، که تاکنون بررسی شده‌اند، نشان داده شده است. همچنین معادل زبان ماشین این دستورات، در شکل ۸ نشان داده شده است.

دسته	دستورالعمل	مثال	معنی	توضیحات
حسابی	جمع	add $s1, $s2, $s3	$s1 = $s2 + $s3	دارای سه عملوند رجیستری
	تفریق	sub $s1, $s2, $s3	$s1 = $s2 - $s3	دارای سه عملوند رجیستری
	جمع فوری	add $s1, $s2, 100	$s1 = $s2 + 100	جمع رجیستر با ثابت
منطقی	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	دارای سه عملوند رجیستری
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	دارای سه عملوند رجیستری
	nor	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)	دارای سه عملوند رجیستری
	andi	andi $s1, $s2, 100	$s1 = $s2 & 100	AND رجیستر با ثابت
	ori	ori $s1, $s2, 100	$s1 = $s2 | 100	OR رجیستر با ثابت
	شیفت به چپ منطقی	sll $s1, $s2, 10	$s1 = $s2 << 10	شیفت به چپ به اندازه ثابت
	شیفت به راست منطقی	srl $s1, $s2, 10	$s1 = $s2 >> 10	شیفت به راست به اندازه ثابت
انتقال داده	بارکردن کلمه	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	انتقال از حافظه به رجیستر
	ذخیره کردن کلمه	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	انتقال از رجیستر به حافظه
پرش شرطی	پرش در صورت تساوی	beq $s1, $s2, L	If ($s1 == $s2) goto L	مقایسه مساوی بودن
	پرش در صورت عدم تساوی	bne $s1, $s2, L	If ($s1 != $s2) goto L	مقایسه نامساوی بودن
	slt	slt $s1, $s2, $s3	If ($s2 < $s3) then $s1=1 else $s1=0	مقایسه کوچکتر بودن
	slti	slti $s1, $s2, 100	If ($s2 < 100) then $s1=1 else $s1=0	مقایسه کوچکتر بودن از ثابت
پرش غیر شرطی	پرش	j L	goto L	پرش بلاشرط به آدرس مقصد

شکل ۷: معماری MIPS بررسی شده تا این قسمت

تلاقی سخت افزار و نرم افزار: کامپایلرهای معماری MIPS، با استفاده از دستورات slt ، slti ، beq ، bne و مقدار ثابت 0 (که همیشه با خواندن رجیستر $zero در دسترس است)، همه‌ی شرط‌های نسبی نظیر مساوی، نامساوی، کوچکتر، کوچکتر یا مساوی، بزرگتر و بزرگتر یا مساوی را تولید می‌کنند.

تلاقی سخت افزار و نرم افزار: اگرچه در زبانهای برنامه نویسی سطح بالا مانند C و جاوا، دستورهای زیادی برای تصمیم گیری و حلقه‌ها وجود دارد، اما در سطح پایین‌تر که زبان اسمبلی است، تعداد محدودی دستور پرش شرطی وجود دارد که عبارتهای تصمیم گیری زبانهای سطح بالا را پیاده سازی می‌کنند.

نام	قالب	مثال						توضیحات
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
addi	I	8	18	17	100			addi $s1, $s2, 100
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
and	R	0	18	19	17	0	36	and $s1, $s2, $s3
or	R	0	18	19	17	0	37	or $s1, $s2, $s3
nor	R	0	18	19	17	0	39	nor $s1, $s2, $s3
andi	I	12	18	17	100			andi $s1, $s2, 100
ori	I	13	18	17	100			ori $s1, $s2, 100
sll	R	0	0	18	17	10	0	sll $s1, $s2, 10
srl	R	0	0	18	17	10	2	srl $s1, $s2, 10
beq	I	4	17	18	25			beq $s1, $s2, 100
bne	I	5	17	18	25			bne $s1, $s2, 100
slt	R	0	18	19	17	0	42	slt $s1, $s2, $s3
j	J	2	2500					j 10000
اندازه‌ی میدان		۶ بیت	۵ بیت	۵ بیت	۵ بیت	۵ بیت	۶ بیت	همه دستورات MIPS ، ۳۲ بیتی هستند
قالب R	R	op	rs	rt	rd	shamt	funct	قالب دستورالعمل حسابی
قالب I	I	op	rs	rt	address (constant)			قالب دستورالعمل انتقال داده

شکل ۸: زبان ماشین دستورات MIPS بررسی شده تا این قسمت

۲-۷- دستورهای پشتیبانی رویه‌ها در MIPS

استفاده از توابع و رویه‌ها[1] (زیر روال‌ها) باعث آسانی فهم و بهره‌گیری مجدد از کد می‌شود. برنامه-نویسان، در صورت استفاده از توابع و رویه‌ها، می‌توانند فقط روی یک بخش تمرکز نموده و کد نویسی کنند. روش استفاده از توابع و رویه‌ها به این صورت است که برنامه اصلی، یک تابع را فراخوانی نموده و ورودی‌های (پارامترهای) لازم را در اختیار او قرار می‌دهد. سپس کنترل به داخل

[1] - Procedures

تابع منتقل شده و اجرای تابع شروع می‌شود. تابع عملیات مشخصی را انجام داده و نتایج حاصل را جهت استفاده در برنامه اصلی در جایی ذخیره می‌نماید. سپس کنترل به برنامه اصلی منتقل شده و برنامه اصلی، درست از دستور بعد از فراخوانی تابع، به اجرای خود ادامه می‌دهد.

تعریف: به برنامه‌ای که فراخوانی می‌کند، فراخواننده[1] و به برنامه‌ای که فراخوانی می‌شود، فراخوانده[2] می‌گویند.

بنابراین هر فراخوانی شامل موارد زیر است:

۱. قرار دادن ورودی‌ها در مکانی که تابع به آن دسترسی داشته باشد.
۲. انتقال جریان اجرا به داخل تابع.
۳. بدست آوردن منابع ذخیره سازی برای متغیرهای محلی در داخل تابع.
۴. انجام کار مشخص شده توسط تابع.
۵. قرار دادن نتایج در مکانی که برنامه فراخواننده به آن دسترسی دارد.
۶. بازگرداندن کنترل اجرا به برنامه اصلی.

همان گونه که قبلاً نیز توضیح داده شده است، رجیسترها سریع‌ترین مکان برای نگهداری داده‌ها در کامپیوتر هستند. بنابراین باید تا حد امکان از آنها استفاده کنیم. معماری MIPS از قواعد زیر جهت فراخوانی توابع استفاده می‌کند:

- $a0 - $a3 : چهار رجیستر آرگومان برای ارسال پارامترها
- $v0 -$v1 : دو رجیستر برای برگشت دادن نتایج
- $ra : یک رجیستر برای نگهداری آدرس بازگشت

معماری MIPS برای فراخوانی توابع، دستورالعملی به نام jal[3] دارد. این دستور دو کار مهم انجام می‌دهد. ابتدا آدرس دستورالعمل بعدی (دستور بعد از jal) را در رجیستر $ra قرار می‌دهد و سپس به آدرس مشخص شده در دستور، پرش می‌نماید. علت نامگذاری jal این است که این دستور پرش می‌کند ولی همچنان پیوند خود را با برنامه اصلی حفظ می‌کند. کنترل اجرا، حتماً باید بعد از اجرای تابع به برنامه اصلی منتقل شود. این پیوند، در رجیستر $ra ذخیره شده و *آدرس بازگشت* نامیده می‌شود. برای اینکه از نقاط مختلف برنامه بتوانیم فراخوانی را انجام دهیم، به این آدرس بازگشت نیاز داریم. شکل دستور jal به صورت زیر است:

---

[1] - Caller
[2] - Callee
[3] - jump and link

```
jal ProcedureAddress
```

که در آن ProcedureAddress ، آدرس تابعی است که فراخوانده می‌شود.

در داخل هر پردازنده‌ای، یک رجیستر ویژه به نام *شمارنده برنامه*[1] یا PC وجود دارد که به دستوری که در حال اجراست اشاره می‌کند. در پردازنده MIPS به دلیل اینکه هر دستورالعمل چهار بایت از حافظه را اشغال می‌کند و درست به همین دلیل، آدرس دو دستور متوالی چهار واحد با هم اختلاف دارند، بنابراین دستور بعد از دستور جاری دارای آدرسی برابر با PC+4 خواهد بود (چون آدرس دستور جاری که در حال اجراست مساوی PC است). اگر فرض کنیم دستور فعلی که در حال اجراست، دستور jal باشد، این دستور باعث انتقال اجرا به داخل تابع می‌شود و پس از اینکه اجرای تابع تمام شد، کنترل باید به برنامه اصلی بازگردانده شود. وقتی که کنترل به برنامه اصلی بازگشت، باید دستور بعد از jal که آدرس آن چهار واحد بیشتر از آدرس jal است اجرا شود. بنابراین وقتی که دستور jal اجرا می‌شود، باید آدرس دستور بعد (PC+4) را به عنوان آدرس بازگشت در داخل رجیستر $ra ذخیره کند.

برای اینکه کنترل اجرا از داخل تابع به برنامه اصلی بازگردد، در پردازنده‌های مختلف دستوراتی از قبیل return وجود دارد. در معماری MIPS ، دستور معادل این دستور، دستور jr یا *پرش رجیستری*[2] نام دارد. شکل این دستور به صورت زیر است:

```
jr $ra
```

این دستور، یک پرش بدون شرط به آدرس مشخص شده در رجیستر $ra انجام می‌دهد. از آنجا که از قبل، آدرس بازگشت را داخل رجیستر $ra ذخیره کرده بودیم (با دستور jal)، به همین دلیل بازگشت به دستور بعد از دستور jal در برنامه اصلی، با موفقیت انجام می‌شود.

خلاصه مبحث فراخوانی: برنامه فراخواننده یا فراخوان، مقادیر پارامترها را در $a0 - $a3 قرار داده و از jal X برای پرش به رویه X ، استفاده می‌کند. سپس، رویه محاسبات را انجام داده و نتایج را در $v0- $v1 قرار می‌دهد و کنترل را با استفاده از jr $ra به فراخواننده باز می‌گرداند.

استفاده از رجیسترهای بیشتر

اگر در فراخوانی یک تابع، علاوه بر چهار رجیستر آرگومان و دو رجیستر مقادیر بازگشت، به رجیسترهای بیشتری نیاز داشته باشیم، مجبور هستیم از حافظه برای تبادل داده بین فراخواننده و

---

[1] - Program Counter
[2] - Jump register

فراخوانده استفاده کنیم زیرا نمی‌دانیم که تعداد آرگومانها و مقادیر بازگشتی چند تاست. همچنین هر رجیستری که فرخواننده از آن استفاده می‌کند باید پس از استفاده، همان مقادیر قبل از فراخوانی را داشته باشد. برای نگهداری مقادیر رجیسترها نیز مجبور هستیم از حافظه استفاده کنیم، به دلیل اینکه ممکن است تعداد زیادی فراخوانی تودرتو[1] داشته باشیم و مجبور باشیم تعداد زیادی مقدار را نگهداری کنیم.

ساختمان دادهٔ ایده آل برای ذخیره کردن مقدار رجیسترها، پشته[2] است که در آن، آخرین مقدار وارد شده، اولین مقدار خارج شده می‌باشد. یک پشته نیاز به اشاره‌گری به آخرین مکان پشته دارد تا نشان دهد که در فراخوانی تابع بعدی، مقدار رجیسترها در کجا ذخیره شود و یا بازگشت از تابع، مقادیر قبلی رجیسترها از کجا بازیافت شود. در معماری MIPS ، به اشاره‌گر پشته، sp[3] گفته می‌شود که یکی از ۳۲ رجیستر داخلی می‌باشد.

برای پشته دو عملیات مهم به نام push و pop وجود دارد که پس از هر کدام از این عملیات باید اشاره‌گر پشته تنظیم شود که به بالاترین مکان پشته اشاره کند. در معماری MIPS ، دستور push و pop وجود ندارند و به جای آنها از دستورات sw و lw استفاده می‌شود. همچنین در معماری MIPS ، تنظیم اشاره‌گر پشته به صورت اتوماتیک انجام نمی‌گیرد و باید حتماً داخل برنامه تنظیم شود. در معماری MIPS ، پشته در جهت کاهش آدرسها رشد می‌کند و چون دستورات sw و lw ، انتقال ۳۲ بیتی یا ۴ بایتی انجام می‌دهند، بنابراین باید با هر push ، ۴ واحد از sp پشته کم شده و با pop ، ۴ واحد به sp اضافه شود.

مثال: تابع زیر را در نظر گرفته و آن را به کد اسمبلی MIPS تبدیل کنید. فرض کنید که متغیرهای g ، h ، i و j متناظر با رجیسترهای آرگومان $a0 ، $a1 ، $a2 و $a3 ، و f متناظر با $s0 باشد.

```
int   leaf_example (int g , int h , int i , int j)
{
    int f;

    f =  (g + h) – (i + j);
    return f;
}
```

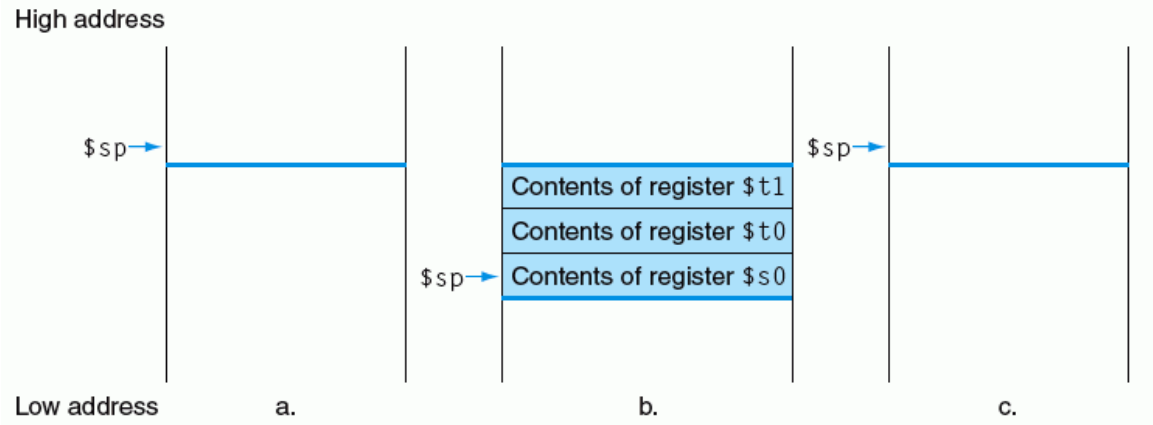
پاسخ: کد اسمبلی به صورت زیر است:

---

[1] - Nested
[2] - Stack
[3] - Stack Pointer

```
Leaf_example:

addi  $sp, $sp, -12
sw     $t1, 8($sp)
sw    $t0, 4($sp)
sw    $s0, 0($sp)

add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub   $s0, $t0, $t1

add   $v0, $s0, $zero

lw     $s0, 0($sp)
lw     $t0, 4($sp)
lw     $t1, 8($sp)
addi   $sp, $sp, 12

jr     $ra
```

توضیح: برنامه کامپایل شده با برچسب تابع آغاز می‌شود و چون در داخل تابع محتوای رجیسترهای $t0 ، $t1 و $s0 ، تغییر می‌کند و ممکن است فراخواننده به مقدار قبلی این رجیسترها (قبل از فراخوانی) نیاز داشته باشد، بنابراین قبل از هر کاری، مقدار این رجیسترها داخل پشته ذخیره شده‌اند. قسمت بعدی کد عملیات داخل تابع را پیاده سازی کرده و نتیجه را داخل رجیستر $v0 ذخیره می‌کند. در خاتمه مقادیر ذخیره شده در داخل پشته، بازیابی شده و با دستور jr کنترل اجرا به برنامه اصلی باز می‌گردد. محتوای پشته و اشاره‌گر پشته، برای این مثال، قبل از فراخوانی، حین اجرای تابع و بعد از بازگشت از تابع، در شکل ۹ نشان داده شده است.



شکل ۹: محتوای پشته و اشاره‌گر پشته (الف) قبل از فراخوانی، (ب) در حین اجرای تابع و (پ) پس از بازگشت از تابع

در مثال بالا از رجیسترهای موقت استفاده کردیم و فرض کردیم که مقادیر قدیمی آنها باید ذخیره و بازیابی شوند. برای اجتناب از ذخیره و بازیابی یک رجیستر که مقدار آن هرگز مورد استفاده قرار نخواهد گرفت، معماری MIPS، ۱۸ رجیستر را به دو گروه تقسیم می‌کند:

- $t0 - $t9 : ده رجیستر موقت که فراخوانده (تابع یا رویه)، محتوای آنها را در فراخوانی محفوظ نگه نمی‌دارد.
- $s0 - $s7 : هشت رجیستر ذخیره شده که باید هنگام فراخوانی تابع یا رویه محفوظ نگه داشته شوند (اگر از آنها در داخل تابع استفاده شود، باید در داخل تابع آنها را ذخیره و بازیابی نمود).

این قراداد ساده، میزان انتقال رجیسترها به حافظه و بر عکس را کاهش می‌دهد. در مثال بالا، از آنجا که فراخواننده، طبق قرارداد، انتظار ندارد که محتوای رجیسترهای $t0 و$t1 در فراخوانی تابع محفوظ نگه داشته شوند، می‌توانیم دو دستور بازیابی و دو دستور بار کردن را از برنامه حذف کنیم. هنوز باید $s0 را ذخیره و بازیابی کنیم، چون فراخوانده، طبق قرارداد، باید فرض کند که فراخواننده به مقدار آن نیاز دارد.

در شکل ۱۰ همه‌ی مواردی که باید داخل تابع ذخیره و بازیابی شوند، نشان داده شده است (محتوای این رجیسترها محفوظ می‌ماند). همچنین مواردی که لازم نیست ذخیره شوند نیز آورده شده است (محتوای این رجیسترها محفوظ نمی‌ماند). در شکل ۱۱ نیز شماره عددی مربوط به هر رجیستر و مورد استفاده و اینکه آیا محفوظ می‌ماند یا نه، نشان داده شده است.

در و و نیز به ترتیب دستورهای زبان اسمبلی و زبان ماشین بررسی شده تا این قسمت ارائه شده است.

رجیسترهایی که محفوظ می‌مانند	رجیسترهایی که محفوظ نمی‌مانند
رجیسترهای ذخیره شده: $s0-$s7	رجیسترهای موقت: $t0-$t9
رجیستر اشاره‌گر پشته: $sp	رجیسترهای آرگومان: $a0-$a3
رجیستر آدرس بازگشت: $ra	رجیسترهای مقدار بازگشت: $v0-$v1

شکل ۱۰: مواردی که باید و نباید داخل تابع ذخیره و بازیابی شوند

نام	شماره رجیستر	مورد استفاده	محفوظ می‌ماند یا نه
$zero	0	همیشه صفر است	بدون تغییر
$v0-$v1	2-3	مقادیر نتایج و ارزیابی عبارت	خیر
$a0-$a3	4-7	آرگومان‌ها	خیر
$t0-$t7	8-15	موقت	خیر
$s0-$s7	16-23	ذخیره شده	بله
$t8-$t9	24-25	موقت‌های بیشتر	خیر
$gp	28	اشاره‌گر سراسری	بله
$sp	29	اشاره‌گر پشته	بله
$fp	30	اشاره‌گر فریم	بله
$ra	31	آدرس بازگشت	بله

شکل ۱۱: شماره و قرارداد رجیسترهای MIPS

توضیحات	معنی	مثال	دستورالعمل	دسته
دارای سه عملوند رجیستری	$s1 = $s2 + $s3	add $s1, $s2, $s3	جمع	حسابی
دارای سه عملوند رجیستری	$s1 = $s2 - $s3	sub $s1, $s2, $s3	تفریق	
جمع رجیستر با ثابت	$s1 = $s2 + 100	add $s1, $s2, 100	جمع فوری	
دارای سه عملوند رجیستری	$s1 = $s2 & $s3	and $s1, $s2, $s3	and	منطقی
دارای سه عملوند رجیستری	$s1 = $s2 | $s3	or $s1, $s2, $s3	or	
دارای سه عملوند رجیستری	$s1 = ~($s2 | $s3)	nor $s1, $s2, $s3	nor	
AND رجیستر با ثابت	$s1 = $s2 & 100	andi $s1, $s2, 100	andi	
OR رجیستر با ثابت	$s1 = $s2 | 100	ori $s1, $s2, 100	ori	
شیفت به چپ به اندازه ثابت	$s1 = $s2 << 10	sll $s1, $s2, 10	شیفت به چپ منطقی	
شیفت به راست به اندازه ثابت	$s1 = $s2 >> 10	srl $s1, $s2, 10	شیفت به راست منطقی	
انتقال از حافظه به رجیستر	$s1 = Memory[$s2 + 100]	lw $s1, 100($s2)	بارکردن کلمه	انتقال داده
انتقال از رجیستر به حافظه	Memory[$s2 + 100] = $s1	sw $s1, 100($s2)	ذخیره کردن کلمه	

مقایسه مساوی بودن	If ($s1 == $s2) goto L	beq $s1, $s2, L	پرش در صورت تساوی	پرش شرطی
مقایسه نامساوی بودن	If ($s1 != $s2) goto L	bne $s1, $s2, L	پرش در صورت عدم تساوی	
مقایسه کوچکتر بودن	If ($s2 < $s3) then $s1=1 else $s1=0	slt $s1, $s2, $s3	slt	
مقایسه کوچکتر بودن از ثابت	If ($s2 < 100) then $s1=1 else $s1=0	slti $s1, $s2, 100	slti	
پرش بلاشرط به آدرس مقصد	goto L	j L	پرش	پرش غیر شرطی
برای بازگشت از تابع	goto L	jr $ra	پرش رجیستری	
برای فراخوانی تابع	$ra = PC + 4; goto L	jal L	پرش و پیوند	

شکل ۱۲: معماری MIPS بررسی شده تا این بخش

نام	قالب	مثال						توضیحات
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
addi	I	8	18	17	100			addi $s1, $s2, 100
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
and	R	0	18	19	17	0	36	and $s1, $s2, $s3
or	R	0	18	19	17	0	37	or $s1, $s2, $s3
nor	R	0	18	19	17	0	39	nor $s1, $s2, $s3
andi	I	12	18	17	100			andi $s1, $s2, 100
ori	I	13	18	17	100			ori $s1, $s2, 100
sll	R	0	0	18	17	10	0	sll $s1, $s2, 10
srl	R	0	0	18	17	10	2	srl $s1, $s2, 10
beq	I	4	17	18	25			beq $s1, $s2, 100
bne	I	5	17	18	25			bne $s1, $s2, 100
slt	R	0	18	19	17	0	42	slt $s1, $s2, $s3
j	J	2	2500					j 10000
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 1000
اندازه‌ی میدان		۶ بیت	۵ بیت	۵ بیت	۵ بیت	۵ بیت	۶ بیت	همه دستورات MIPS ، ۳۲ بیتی هستند
قالب R	R	op	rs	rt	rd	shamt	funct	قالب دستورالعمل حسابی
قالب I	I	op	rs	rt	address (constant)			قالب دستورالعمل انتقال داده

شکل ۱۳: زبان ماشین MIPS بررسی شده تا این بخش

۲-۸- کار با کاراکترها و رشته‌ها

در زبانهای برنامه نویسی، علاوه بر نوع داده‌های ۱۶ بیتی و ۳۲ بیتی، نوع داده ۸ بیتی نیز زیاد مورد استفاده قرار می‌گیرد. به طور مثال کد اسکی[1]، که به عنوان یک استاندارد برای تبادل اطلاعات مورد استفاده قرار می‌گیرد، شامل یک بایت برای هر کدام از حروف و ارقام می‌باشد. همچنین رشته‌ها که امروز در زبانهای برنامه نویسی استفاده می‌شوند، از کاراکترها تشکیل شده‌اند که هر کاراکتر در حافظه کامپیوتر، یک بایت را اشغال می‌کند و نوع داده آن ۸ بیت است. نمایش کد اسکی برای کاراکترها در شکل ۱۴ نشان داده شده است.

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

شکل ۱۴: نمایش اسکی کاراکترها

می‌توان با استفاده از دستورات lw و sw و یک سری دستورات دیگر برای انتقال بایت بین حافظه و رجیسترها استفاده کرد (چون می‌توان با یک سری از دستورات، یک بایت را از یک کلمه بیرون کشید). اما به هر حال، به علت رایج بودن حضور متن‌ها در برخی برنامه‌ها، معماری MIPS ، دستورالعمل‌هایی برای انتقال بایت دارد. این دستورات، که قبلاً نیز توضیح داده‌ایم، lb و sb می‌باشند. دستورالعمل بارکردن بایت (lb) ، یک بایت را از حافظه خوانده و آن را در ۸ بیت سمت راست یک رجیستر قرار می‌دهد. دستورالعمل ذخیره کردن بایت (sb) ، یک بایت را از ۸ بیت سمت راست رجیستر برداشته و آن را در حافظه می‌نویسد. در زبان برنامه سازی C ، یک رشته از تعدادی کاراکتر تشکیل می‌شود و انتهای رشته به کاراکتر تهی (NULL) که معادل اسکی آن 0 است ختم می‌شود. بنابراین رشته "Cal" در C با ۴ بایت به صورت اعداد دهدهی 0 ، 108 ، 97 و 67 نمایش داده می‌شود.

[1] - ASCII

مثال: تابع strcpy در زبان C ، با استفاده از یک بایت تهی که قرارداد خاتمه رشته در C می‌باشد، در رشته X کپی می‌کند:

```
void  strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')     /* copy & test byte */
    i += 1;
}
```

با فرض اینکه آدرس‌های پایه‌ی x و y به ترتیب در $a0 و $a1 و i در $s0 ، کد اسمبلی MIPS مربوط به آن را بنویسید.

پاسخ: کد اسمبلی MIPS به صورت زیر است:

```
strcpy:
        addi   $sp, $sp, -4
        sw     $s0, 0($sp)

        add    $s0, $zero, $zero   #    i = 0

L1:     add    $t1, $s0, $a1    #  $t1 = i + $a1   (address of y[i])
        lb     $t2, 0($t1)      #  $t2 = y[i]

        add    $t3, $s0, $a0    #  $t3 = i + $a0   (address of x[i])
        sb     $t2, 0($t3)      #  x[i] = y[i]

        beq    $t2, $zero, L2   #  if  (y[i] == 0) goto L2

        addi   $s0, $s0, 1      #  i = i + 1
        j      L1

L2:     lw     $s0, 0($sp)
        addi   $sp, $sp, 4
        jr     $ra
```

در کد اسمبلی فوق به دلیل اینکه محتوای رجیستر $so (i) از بین می‌رود، بنابراین بر طبق قرارداد، در ابتدای تابع، مقدار آن را در پشته ذخیره و در انتهای تابع، مقدار آن را از پشته بازیابی کرده‌ایم. بعد از ذخیره کردن در پشته، مقدار i ($s0) را صفر کرده‌ایم. سپس در یک حلقه تا رسیدن به کاراکتر تهی، رشته y را داخل x کپی کرده‌ایم (L1). برای کپی کردن، ابتدا آدرس y[i] را حساب کرده و آن را از حافظه خوانده داخل رجیستر $t2 قرار می‌دهیم، سپس آدرس x[i] را حساب کرده و مقدار $t2 را در

آن آدرس ذخیره می‌کنیم. سپس با دستور beq چک می‌کنیم که آیا به انتهای رشته رسیده‌ایم یا نه (اگر به انتهای رشته رسیده باشیم مقدار y[i] که داخل $t2 قرار دارد مساوی صفر می‌شود). اگر به انتهای رشته نرسیده بودیم، دستور بعدی اضافه کردن i و برگشتن به اول حلقه است.
آخرین دستور این تابع نیز به مانند هر تابع، دستور jr است که به برنامه‌ی فراخواننده بازگشت می‌کند.

۲-۹- طرز استفاده از ثابت‌های ۳۲ بیتی در معمار MIPS

اگرچه معمولاً ثوابت کوتاه بوده و در یک میدان ۱۶ بیتی جای می‌گیرند، اما گاهی اوقات به ثوابت بزرگتری نیاز داریم. در MIPS می‌توان ثابت‌های ۳۲ بیتی را نیز داخل یک رجیستر قرار داد برای این کار باید از دو دستور lui و ori به صورت زیر استفاده نمود:

- دستور lui [1]، ۱۶ بیت بالایی یک رجیستر را با عدد ثابت پر می‌کند و ۱۶ بیت پایین آن را پاک می‌کند (با مقدار 0 پر می‌کند).
- دستور or منطقی بلافصل یا ori[2] که به دنبال دستور lui استفاده می‌شود، ۱۶ بیت پایینی یک رجیستر را با یک عدد ثابت ۱۶ بیتی و ۱۶ بیت بالایی آن را با ۱۶ بیت 0 ، or منطقی می‌کند (عملیات منطقی بیت به بیت انجام می‌شوند).

به طور مثال برای قرار دادن عدد 0000 0000 1001 0000 1101 0011 0000 0000 در داخل یک رجیستر به صورت زیر عمل می‌شود. این عدد در سیستم دهدهی معادل با عدد 4000000 و در سیستم هگزادسیمال معادل عدد 003D0900 است. ۱۶ بیت بالای این عدد، مساوی 1101 0011 0000 0000 بوده و معادل با 61 دهدهی می‌باشد. همچنین ۱۶ بیت پایین این عدد، مساوی 0000 0000 1001 0000 بوده و معادل با 2304 دهدهی می‌باشد. اگر بخواهیم عدد 4000000 را داخل رجیستر $s0 قراردهیم، از دستورات lui و ori به صورت زیر استفاده می‌کنیم.

```
lui  $s0, 61          #  $s0 = (003D 0000)16
ori  $s0, $s0, 2304   #  $s0 = (003D 0900)16
```

توجه: چون اکثر مواقع ثابت‌های ۱۶ بیتی برای منظورهای ما کافی است دستورات پردازنده MIPS نیز از ثابت‌های ۱۶ بیتی استفاده می‌کنند. استفاده از ثابت‌های ۳۲ بیتی نیز در MIPS امکان‌پذیر است تنها هزینهٔ اضافی همان طور که در مثال قبل دیدیم یک دستور اضافه و یک رجیستر موقت اضافه است. این مطلب نشان دهنده رعایت قانون امدال از جانب طراحان پردازنده MIPS می‌باشد. چون ثابت‌های

[1] - Load upper immediate
[2] - OR immediate

۱۶ بیتی در برنامه‌ها بیشتر از ثابت‌های ۳۲ بیتی استفاده می‌شوند، طراحان MIPS نیز دستوراتی طراحی کردند که از ثابت‌های ۱۶ بیتی استفاده می‌کنند نه از ثابت‌های ۳۲ بیتی. با این حال در موارد نادری که لازم است از ثابت‌های ۳۲ بیتی استفاده این کار را می‌توان با چند دستور انجام داد. در واقع طراحان MIPS بر روی موارد پر استفاده سرمایه گذاری کرده‌اند نه بر روی موارد کم استفاده و این همان ایده قانون امدال است.

توجه: بعضی از اسمبلرها مانند SPIM ممکن است شبه دستورهایی داشته باشند که با ثابت‌های بزرگتر کار کنند.

توجه: SPIM شبیه ساز[1] پردازنده MIPS است.

ثابت‌های ۳۲ بیتی در دستورهای بارکردن و ذخیره کردن

همان طور که قبلاً مشاهده کردیم، شکل دستور بارکردن به صورت lw $t0, Constant ($a0) می‌باشد. محدودیت داشتن ثابت‌های ۱۶ بیتی ممکن است در دسترسی به آرایه‌های بزرگ مشکل ایجاد کند. به طور مثال اگر آدرس شروع یک آرایه بزرگتر از ۳۲۷۶۷ ($2^{16} \div 2$) باشد، این ثابت ۱۶ بیتی نمی‌تواند آن را نمایش دهد و همچنین اگر تعداد عناصر آرایه بیشتر از ۳۲۷۶۷ باشد باز هم این ثابت ۱۶ بیتی نمی‌تواند اندیس مناسب را نمایش دهد. در چنین شرایطی راه حل MIPS این است که از چند دستور برای پیاده سازی چنین عملی استفاده کنیم. ما باید آدرس واقعی عنصر مورد نظر آرایه را محاسبه نموده و به طور دستی آن را در داخل یک رجیستر قرار دهیم و سپس از این رجیستر برای آدرس‌دهی استفاده کنیم. به طور مثال، قطعه کد زیر می‌تواند عنصر یک میلیونم (بایت یک میلیونیم) یک آرایه را که از آدرس دهدهی 3000000 شروع می‌شود در داخل یک رجیستر بار نماید. (در واقع آدرس این عنصر 4000000 خواهد بود و ما نحوه قرار دادن عدد 4000000 در داخل یک رجیستر را در بالا توضیح داده‌ایم).

```
Lui   $s0, 61
Ori   $s0, $s0, 2304  #   $s0 = 4000000 (decimal)
Lb    $t1,  0($s0)    #   $t1 = Mem[4000000]
```

در این مثال ما ابتدا آدرس عنصر را که 4000000 بوده و یک عدد ثابت ۳۲ بیتی است (این عدد به بیشتر از ۱۶ بیت نیاز دارد)، در داخل رجیستر $s0 قرار داده‌ایم و سپس با استفاده از دستور

[1] - Simulator

lb $t1,0($s0) ، عنصر مورد نظر را از حافظه خوانده‌ایم. همان طور که می‌دانیم در این دستور، آدرسی که مورد دسترسی قرار خواهد گرفت به صورت 4000000 = $s0 = $s0 + 0 خواهد بود، یعنی همان آدرس عنصر مورد نظر.

۲-۱۰- آدرس دهی دقیق در دستورات انشعاب و پرش

دستورالعمل‌های پرش در معماری MIPS ، از قالبی به نام نوع J ، استفاده می‌کنند. در این قالب که سومین و آخرین قالب در معماری MIPS است، از ۶ بیت برای میدان عملوند و از بقیه‌ی بیت‌ها برای میدان آدرس استفاده می‌شود. بنابراین دستورالعمل زیر:

```
j    10000    # goto location 10000
```

به صورت زیر می‌تواند در کد ماشین نوشته شود (بعداً خواهیم دید که آدرس دهی در دستور پرش اندکی متفاوت است):

2	10000
۶ بیت	۲۶ بیت

توجه: کد عمل برای دستور پرش 2 می‌باشد.

دستورالعمل انشعاب شرطی، بر خلاف دستورالعمل پرش باید دو عملوند را علاوه بر آدرس انشعاب مشخص نماید. بنابراین دستور:

```
bne    $s0, $s1, Exit      #   if ($s0 != $s1) goto Exit
```

به کد ماشین زیر تبدیل می‌شود که فقط ۱۶ بیت برای آدرس در نظر می‌گیرد:

5	16	17	Exit
۶ بیت	۵ بیت	۵ بیت	۱۶ بیت

اگر مجبور بودیم که آدرس‌های برنامه را در میدان ۱۶ بیتی جای دهیم، نمی‌توانستیم برنامه‌ای بزرگتر از 2^{16} بایت داشته باشیم، که برای برنامه‌های امروزی عدد کوچکی است.یک راه دیگر، مشخص کردن یک رجیستر است که همیشه به آدرس انشعاب اضافه شود، طوری که دستورالعمل انشعاب برای آدرسی که می‌خواهیم به آن پرش کنیم (آدرس مقصد) محاسبه‌ی زیر را انجام دهد:

آدرس انشعاب + رجیستر = شمارنده‌ی برنامه

این مجموع به برنامه اجازه می‌دهد تا به اندازه‌ی 2^{32} بایت باشد. در این راه حل، هنوز می‌توان از انشعاب‌های شرطی با میدان آدرس ۱۶ بیتی استفاده نمود و مشکل اندازه‌ی آدرس را حل کرد. در اینجا می‌توان این سؤال را مطرح نمود که کدام رجیستر؟

پاسخ پرسش بالا، به نحوه‌ی استفاده‌ی انشعاب شرطی باز می‌گردد. انشعاب‌های شرطی، معمولاً در حلقه‌ها و عبارت‌های if دیده می‌شوند، بنابراین گرایش به انشعاب به یک دستورالعمل نزدیک دارند. به طور مثال، تقریباً نیمی از همه‌ی انشعاب‌های شرطی در آزمون کارآیی SPEC2000 به مکان‌هایی می‌روند که کمتر از ۱۶ دستورالعمل با آن فاصله دارند. از آنجا که شمارنده‌ی برنامه (PC)، حاوی آدرس دستورالعمل جاری است، اگر از PC به عنوان رجیستری که به آدرس اضافه می‌شود، استفاده کنیم، می‌توانیم در محدوده $\pm 2^{15}$ کلمه از دستورالعمل جاری انشعاب انجام دهیم. تقریباً همه‌ی حلقه‌ها و عبارت‌های if کوچکتر از 2^{16} کلمه هستند، بنابراین PC یک گزینه‌ی ایده‌آل است.

این نوع آدرس دهی انشعاب، *آدرس دهی نسبی PC*[1] نامیده می‌شود. همان طور که در فصل ۵ خواهیم دید، افزایش یک واحدی PC برای اشاره به دستورالعمل بعدی، برای سخت افزار راحت‌تر است. بنابراین آدرس MIPS نسبت به آدرس دستورالعمل بعدی (PC + 4)، برخلاف دستورالعمل جاری (PC)، نسبی می‌باشد. MIPS همانند بسیاری از کامپیوترهای اخیر، از آدرس‌دهی نسبی برای همه‌ی انشعاب‌های شرطی استفاده می‌کند.

از طرف دیگر، دستورالعمل‌های پرش و پیوند، توابعی را فراخوانی می‌کنند که لزوماً ممکن است خود تابع فراخواننده نباشد، و بنابراین ممکن است آدرسی که می‌خواهیم به آن پرش کنیم، در فاصله‌ی دوری قرار گرفته باشد، پس برای این حالت باید از روش‌های دیگر آدرس‌دهی استفاده کنیم. معماری MIPS با بهره‌گیری از قالب J برای دستورالعمل‌های پرش و پرش و پیوند، از آدرس‌دهی طولانی برای فراخوانی توابع دور استفاده می‌کند.

از آنجا که طول هر دستورالعمل MIPS ، ۴ بایت است، MIPS به جای اینکه از تعداد بایتها استفاده کند، از تعداد کلمه‌ها تا دستورالعمل بعدی برای آدرس دهی نسبی PC استفاده می‌کند. بنابراین میدان ۱۶ بیتی می‌تواند با تفسیر میدان به صورت آدرس کلمه‌ی نسبی به جای بایت نسبی، ۴ برابر فاصله بیشتر را آدرس دهی نماید. به طور مشابه، میدان ۲۶ بیتی در دستورالعمل‌های پرش نیز یک آدرس کلمه است، یعنی می‌تواند آدرس بایت ۲۸ بیتی را ارائه کند.

[1] - PC relative addressing

مثال: با فرض اینکه یک حلقه‌ی while به کد اسمبلی زیر تبدیل شده باشد:

```
Loop: sll   $t1, $s3, 2    #  $t1 = 4 * i
      add  $t1, $t1, $s6  #  $t1 = 4 * i  + $s6
      lw    $t0, 0($t1)     #  $t0 = A[i]
      bne  $t0, $s5, Exit #  if A[i] != k then goto Exit
      add  $s3, $s3, 1     #  i = i + 1
      j       Loop          #  goto Loop
Exit
```

و با فرض اینکه آغاز حلقه را در آدرس 80000 در حافظه قرار دهیم، کد ماشین MIPS برای این حلقه را بدست آورید.

پاسخ: کد ماشین و آدرس‌ها به صورت زیر است:

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

می‌دانیم که در پردازنده MIPS ، هر آدرس حافظه شامل یک بایت است و همچنین طول دستورالعمل ۳۲ بیت و یا ۴ بایت است. بنابراین آدرس دو دستور پشت سرهم چهار واحد اختلاف دارند. دستورالعمل bne در سطر چهارم، ۲ کلمه یا ۸ بایت را به آدرس دستورالعمل بعدی (80016) می‌افزاید و مقصد انشعاب را به جای اینکه نسبت به خود دستورالعمل انشعاب (12 + 80012) یا با استفاده از آدرس کامل مقصد (80024) مشخص کند، نسبت به دستورالعمل بعدی (8 + 80016) مشخص می‌کند. دستورالعمل پرش در آخرین سطر، از آدرس کامل (80000 = 4 × 20000) که متناظر با برچسب LOOP است، استفاده می‌کند.

ثابت‌های بزرگ در دستورهای پرش

مطالعات تجربی بر روی برنامه‌های واقعی نشان می‌دهد که دستورهای پرش شرطی در اکثر مواقع به مقصدهایی پرش می‌کنند که فاصلهٔ آنها با دستور پرش کمتر از ۳۲۷۶۷ دستور می‌باشد. دلیل این امر این است که دستورهای پرش اکثراً در داخل حلقه‌ها و یا در مواردی مورد استفاده قرار می‌گیرند که برنامه‌نویس می‌خواهد حجم کد را کاهش دهد. بنابراین اکثر مواقع پرش به فاصله‌های نزدیک انجام می‌شود. اگر شما احتیاج به این داشته باشید که به فاصله‌های بزرگتری پرش انجام دهید، می‌توانید از

دستور branch در کنار دستور jump استفاده کنید. به طور مثال اگر بر چسب Far نشان دهنده فاصله دورتر باشد، عبارتی همانند دستور beq $a0, $s1,Far (که می‌دانیم با ثابتهای ۱۶ بیتی امکان‌پذیر نیست) می‌تواند با استفاده از دو دستور زیر پیاده‌سازی شود.

```
      bne   $s0, $s1, Next
      j     Far
Next: ….
```

در این مثال Next برچسبی است که فاصله نزدیک و Far برچسبی است که فاصله دور را نشان می-دهد. دستور j مورد استفاده در این مثال می‌تواند به فاصله دورتر پرش بدون شرط انجام دهد.

توجه: باز هم در این مثال می‌بینیم که طراحان MIPS روی این نکته دقت داشته‌اند که موارد معمول‌تر و پرکاربردتر را سریعتر کنند (قانون امدال). به دلیل اینکه معمولاً در برنامه‌ها به فواصل دور پرش صورت نمی‌گیرد.

۳-۱۱- شبه دستورات

اسمبلرهای پردازنده MIPS برای اینکه کار برنامه‌نویسی به زبان اسمبلی راحت‌تر شود، تعدای دستور اضافه بر مجموعه دستورات در اختیار قرار می‌دهند که این دستورات جزو دستورات واقعی که بر روی سخت افزار پردازنده اجرا می‌شوند نیستند بلکه هر کدام از آنها خود نماینده یک یا چند دستور واقعی هستند که برای راحت‌تر شدن برنامه‌نویسی و کاهش حجم برنامه مورد استفاده قرار می‌گیرند. به این دستورات اضافه، شبه دستور[1] گفته می‌شود.

به طور مثال شما می‌توانید شبه دستورات li و move را به صورت زیر استفاده کنید:

```
li $a0, 2000     #  $a0 = 2000
move $a1, $t0  #  $a1 = $t0
```

دستورات فوق احتمالاً واضح‌تر از دستورات واقعی زیر باشند که قبلاً توضیح داده شدند.

```
addi $a0, $0, 2000 # $a0 = 2000
add $a1, $t0, $0     # $a1 = $t0
```

توجه: نرم افزار اسمبلر که برنامه اسمبلی را به کد ماشین تبدیل می‌کند در هر جای برنامه اگر شبه دستوری ببیند دستورات معادل آن را از مجموعه دستورات واقعی جایگزین خواهد کرد.

نکته: در هر محاسبه‌ای، قبل از انجام محاسبه، حتماً باید شبه دستورات را با معادل آنها از دستورات واقعی جایگزین کرد.

[1] - Pseudo instruction

دستورهای شبه پرش[1]

پردازنده MIPS فقط دو دستور برای پرش شرطی دارد که عبارتند از beq و bne. بقیه دستورات پرش شرطی را که تا به حال دیده‌اید همگی شبه دستور بودند! اسمبلر MIPS ، شبه دستورهای پرش را با استفاده از دستور slt پیاده‌سازی می‌کند. به طور مثال شبه دستور blt[2] به صورت blt $a0, $a1, Label در واقع از دو دستور واقعی به صورت زیر تشکیل شده است.

```
slt   $at, $a0, $a1   #  if ($a0 < $a1) $at = 1; else $at = 0;
bne  $at, $0, Label #  if ($at=0)  goto Label;
```

در کد فوق اگر $at = 1 شود، در این صورت پرش صورت می‌گیرد چون $at=1 . $1 \neq 0$ و در صورتی می‌شود که $a1 > $a0 باشد. بنابراین پرش وقتی انجام می‌شود که $a1 > $a0 باشد.

دستورهای شبه پرش بلافصل

برای دستور slt نسخهٔ بلافصلی به نام slti وجود دارد که یک رجیستر را با یک عدد ثابت مقایسه می‌کند با استفاده از دستور slti می‌توان شبه دستورهای پرش شرطی بلافصل را ایجاد کرد. به طور مثال شبه دستور blti $a0,5,label در واقع از دو دستور واقعی زیر تشکیل شده است.

```
slt   $at, $a0, 5
bne  $at, $0, Label    #  Branch if $a0 < 5
```

[1] - Pseudo branches
[2] - branch-if-less-than