مقدمه

۱-۱- مقدمه

امروزه پیشرفت در تکنولوژی کامپیوترها، تقریباً همهٔ جنبه‌های جامعه را تحت تأثیر قرار داده است. پیشرفت‌های صورت گرفته در سخت‌افزار، به برنامه‌نویسان اجازه داده است که برنامه‌های واقعاً مفیدی ایجاد کنند. برنامه‌نویسان موفق همیشه به سرعت اجرای برنامه‌های خود توجه می‌کنند چون دادن نتیجه سریع به کاربر در موفقیت یک برنامه نقش حیاتی دارد. در دهه‌های ۱۹۶۰ و۱۹۷۰ اصلی‌ترین محدودیت در سرعت کامپیوترها اندازه حافظه کامپیوترها بود. بعدها پیشرفت‌های ایجاد شده در زمینهٔ طراحی کامپیوترها و تکنولوژی حافظه‌ها واقعاً اهمیت مسأله حافظه کم را کاهش داد. امروزه برنامه-نویسانی که به سرعت اجرای برنامه‌های خود علاقه دارند، باید به مسائلی توجه کنند که جایگزین مسأله کمبود حافظه شده‌اند. از جمله این مسائل می‌توان به ساختار سلسله مراتبی حافظه‌ها و موازات در پردازنده‌ها اشاره نمود.

برنامه‌نویسانی که در جستجوی راههایی هستند که کامپایلرها، سیستم عاملها، برنامه‌های پایگاه داده و حتی application هایی بسازند که قابل رقابت با محصول دیگران باشد، باید دانش خود را در ارتباط با ساختار وسازمان کامپیوتر افزایش دهند. ما در این کتاب توضیح خواهیم داد که داخل یک کامپیوتر چه چیزهایی وجود دارد و شما را با رموز برنامه‌نویسی آشنا خواهیم ساخت. همچنین توضیح داده خواهد شد که ساختار داخلی یک کامپیوتر چگونه بر کارآیی برنامه‌ها تأثیر می‌گذارد. مهمتر از همه اینکه شما یاد خواهید گرفت که چگونه یک کامپیوتر برای خود بسازید.

فصل اول زیر بنای این کتاب است. این بخش به مطالب اساسی و تعاریف می‌پردازد. همچنین بخش-های اصلی نرم‌افزار و سخت‌افزار را بیان می‌نماید. در این بخش مقدمه‌ای در مورد مدارهای فشرده (Integrated circuit) که تکنولوژی مؤثر در پیشرفت کامپیوترها بوده، خواهد آمد.

۱-۲- لایه‌های زیرین برنامه

برای اینکه واقعاً بتوانید با یک ماشین الکترونیکی مانند کامپیوتر ارتباط برقرار کنید باید از علامتهای الکترونیکی استفاده کنید. ساده‌ترین علائم قابل فهم برای این ماشین الکترونیکی، روشن(on) و خاموش (off) می‌باشد و بنابراین الفبای کامپیوتر دارای دو حرف می‌باشد. همان‌طور که ۲۶ حرف موجود در الفبای زبان انگلیسی محدودیتی بر مقدار مطالبی که نوشته می‌شود نمی‌گذارد، الفبای دو حرفی کامپیوتر نیز کارهایی که یک کامپیوتر انجام می‌دهد را محدود نمی‌کند. دو سمبلی که برای این دو حروف استفاده می‌شود ۰ و ۱ است و ما معمولاً زبان کامپیوتر را به صورت عددهایی در مبنای ۲ یا اعداد دودویی می‌شناسیم. به هر کدام از حروف (۰ و ۱) کامپیوتر یک رقم باینری یا بیت گفته می‌شود.

کامپیوترها فقط با فرمانهای ما کار می‌کنند، اسم هر فرمان تکی دستور (instruction) نامیده می‌شود. دستورات که مجموعه‌ای از بیت‌ها (۰ و ۱) می‌باشند که یک کامپیوتر می‌فهمد، می‌توانند به صورت اعداد در نظر گرفته شوند. به طور مثال مجموعهٔ بیت زیر را در نظر بگیرید:

۱۰۰۰۱۱۰۰۱۰۱۰۰۰۰۰۰

این مجموعه بیت یک دستور می‌باشد که می‌تواند به یک کامپیوتر بگوید که دو عدد را با هم جمع کند. در فصل ۳ توضیح داده خواهد شد که چرا ما برای دستورات و داده‌ها از اعداد استفاده می‌کنیم. استفاده از اعداد برای دستورات و داده‌ها پایه و اساس محاسبات است بنابراین ما فصل ۳ را از دست نخواهیم داد!

اولین برنامه‌نویس‌های کامپیوتر، از طریق اعداد باینری با آن ارتباط برقرار می‌کردند یعنی به زبان ماشین کدنویسی می‌کردند. برنامه نویسی به زبان ماشین کار سختی بود و به همین دلیل به سرعت یک مجموعه علامتها اختراع شدند که به روش فکر کردن انسانها نزدیک بودند و برنامه نویسان از طریق این مجموعه علامت‌ها برنامه نوشتند. اسمی که برای این زبان سمبلیک استفاده می‌شد، امروزه نیز کاربرد دارد و آن زبان اسمبلی است. در ابتدا برنامه نویسان کد نوشته شده با این علائم را به طور دستی به مجموعه ۰ و ۱ ها (زبان ماشین) ترجمه می‌کردند. ترجمه دستی نیز کار مشکلی بود. راهکار بعدی این بود که از خود کامپیوتر برای عمل ترجمه کمک بگیریم. برنامه‌نویسان اولیه برنامه‌هایی را جهت ترجمه از علامتهای نمادین به مجموعهٔ ۰ و ۱ ها نوشتند و این برنامه‌ها را اسمبلر (assembler) نامگذاری کردند. اسمبلرها برنامه‌هایی بودند که یک برنامه را که به زبان اسمبلی نوشته شده بود در ورودی خود دریافت می‌کردند و در خروجی خود مجموعه‌ای از ۰ و ۱ ها تولید می‌کردند.

به طور مثال برنامه‌نویس می‌تواند دستوری را به صورت زیر بنویسد:

Add A, B

و اسمبلر می‌تواند این دستور را به صورت زیر ترجمه نماید:

۱۰۰۰۱۱۰۰۱۰۱۰۰۰۰۰۰

این دستور به کامپیوتر می‌گوید که دو عدد A و B را با هم جمع کند.

با وجود ایجاد زبان اسمبلی، برنامه‌نویسی هنوز هم کار سختی بود. در زبان اسمبلی برنامه‌نویس باید برای هر دستور زبان ماشین که کامپیوتر می‌تواند آن را انجام دهد یک دستور اسمبلی بنویسد و این باعث می‌شود که برنامه‌نویس مانند یک ماشین فکر کند.

در اینجا یک سؤال ساده مطرح می‌شود:

اگر ما توانستیم یک برنامه‌ای بنویسیم که عمل ترجمه از زبان اسمبلی به زبان ماشین را انجام دهد، چه عاملی باعث می‌شود که نتوانیم برنامه‌ای بنویسیم که عمل ترجمه از نمادهای سطح بالاتر به زبان اسمبلی را انجام دهد.

جواب این است که هیچ چیز.

برنامه‌هایی که این نمادهای سطح بالاتر را قبول می‌کنند، کامپایلر نامیده می‌شوند و زبانهایی که آنها کامپایل می‌کنند، زبانهای برنامه‌نویسی سطح بالا نامیده می‌شوند. کامپایلرها یک برنامه‌نویس را قادر می‌سازند که عبارتی مانند عبارت زبان سطح بالای زیر را در برنامه‌های خود بنویسند.

A+B

یک کامپایلر ممکن است عبارت فوق را به عبارت زبان اسمبلی زیر کامپایل (ترجمه) نماید.

Add A, B

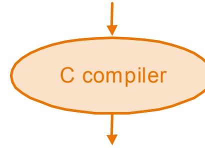
و یک اسمبلر ممکن است عبارت add A, B را به یک دستور باینری به صورت زیر ترجمه نماید که این دستور به کامپایلر بگوید دو عدد A و B را با هم جمع نماید.

۱۰۰۰۱۱۰۰۱۰۱۰۰۰۰۰

شکل ۱-۱ ارتباط بین این برنامه‌ها و زبانها را نشان می‌دهد.

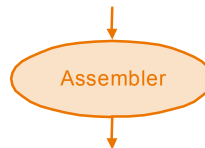
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
     muli $2, $5,4
     add  $2, $4,$2
     lw   $15, 0($2)
     lw   $16, 4($2)
     sw   $16, 0($2)
     sw   $15, 4($2)
     jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

شکل ۱–۱ ارتباط بین زبانهای برنامه‌نویسی و مبدل‌های آنها.

زبانهای برنامه‌نویسی سطح بالا چندین مزیت مهم دارند. اول اینکه آنها به برنامه‌نویس قابلیت این را می‌دهند که در یک زبان خیلی نزدیک به زبان طبیعی فکر کنند. در این زبانها معمولاً برنامه‌نویس از کلمات و علامت‌های ریاضی زبان انگلیسی استفاده می‌کند. بعلاوه آنها اجازه می‌دهند زبان‌هایی متناظر با کاربرد مورد نظر طراحی شوند. به طور مثال فرترن (Fortran) برای محاسبات ریاضی، کوبول (Cobol) برای پردازش داده‌ها تجاری، لیسپ (Lisp) برای دستکاری سمبل‌ها و ... طراحی شده‌اند.

دومین مزیت زبانهای برنامه‌نویسی سطح بالا افزایش تولیدات برنامه‌نویس است. در زبان‌های سطح بالا حجم کد کم است و تعداد زیادی کار را می‌توان در تعداد کمتری از خط‌های برنامه انجام داد این امر مزیت بزرگی نسبت به زبان اسمبلی که در آن حجم برنامه بسیار بزرگ می‌شود، می‌باشد.

آخرین مزیت این است که زبانهای برنامه‌سازی اجازه می‌دهند که برنامه‌ها از کامپیوتری که بر روی آن طراحی می‌شوند مستقل باشند. این به این دلیل است که کامپایلرها و اسمبلرها می‌توانند برنامه نوشته شده را به زبان ماشین کامپیوتری که می‌خواهد آن را اجرا کند تبدیل می‌کنند.

این سه مزیت آنقدر اهمیت دارند که امروزه تعداد برنامه‌های بسیار کمی با زبان اسمبلی نوشته می‌شود.

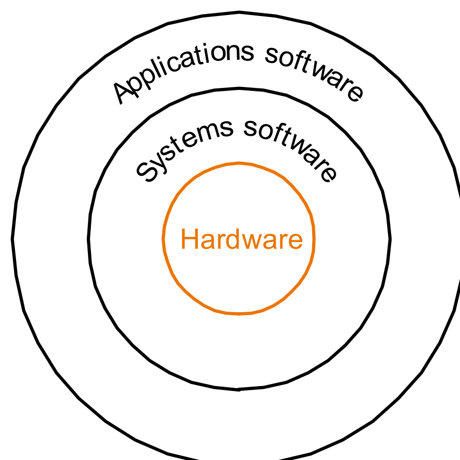
با پیشرفت برنامه‌نویسی، برنامه‌نویسان مشاهده کردند که استفاده مجدد از برنامه‌ها، از اینکه همه چیز را از صفر شروع کنیم خیلی بهتر است. بنابراین برنامه‌نویسان این کار را شروع کردند که روتین‌های پراستفاده را در داخل کتابخانه‌ها (Libraries) جمع کنند. یکی از اولین کتابخانه‌های روتین‌ها، برای خواندن و نوشتن داده بود که به طور مثال شامل روتین‌هایی نظیر روتین‌های کنترل پرینترها بودند که به طور مثال موجود بودن کاغذ داخل پرینتر را قبل از عملیات پرینت کردن چک می‌کنند.

به زودی معلوم گردید که اگر یک برنامه‌ای وجود داشته باشد که اجرا شدن بقیه برنامه‌ها را مدیریت کند یک مجموعه از برنامه‌ها می‌توانند به طور مؤثری اجرا شوند، به محض اینکه اجرای یک برنامه به پایان رسید، برنامهٔ ناظر، برنامهٔ دیگری را از صف برنامه‌های منتظر اجرا، برای اجرا انتخاب می‌کند و بنابراین مانع ایجاد تأخیر می‌شود.

این برنامه ناظر که به زودی کتابخانه روتین‌های ورودی/خروجی نیز به آن اضافه شد، پایه و اساس چیزی بود که امروزه به آن سیستم عامل می‌گوییم. سیستم‌های عامل برنامه‌هایی هستند که منابع یک کامپیوتر را مدیریت می‌کنند تا اینکه برنامه‌های دیگر به طور مؤثری بر روی آن کامپیوتر اجرا شوند.

بعدها نرم‌افزارها بر اساس استفاده آنها دسته‌بندی شدند. نرم‌افزارهایی که معمولاً سرویس‌های مفید در اختیار کاربران قرار می‌دهند، نرم‌افزارهای سیستم(System software) نامیده می‌شوند. سیستم‌عامل‌ها، کامپایلرها و اسمبلرها مثالهایی از نرم‌افزارهای سیستم می‌باشند. در مقابل نرم‌افزارهای سیستمی، نرم-افزارهای کاربردی (applications software) یا همان برنامه‌های کاربردی (application) قرار دارند که نامی است برای برنامه‌هایی که هدفشان کمک کردن به استفاده کننده‌های کامپیوتر است. به طور مثال از این دسته برنامه‌ها می‌توانیم به ادیتورهای متن اشاره کنیم.

شکل ۱–۲ لایه‌های سلسله مراتبی نرم‌افزار و جایگاه آنها نسبت به سخت‌افزار را نشان می‌دهد.



شکل ۱–۲: لایه‌های سلسله مراتبی نرم‌افزار و ارتباط آن با سخت‌افزار.

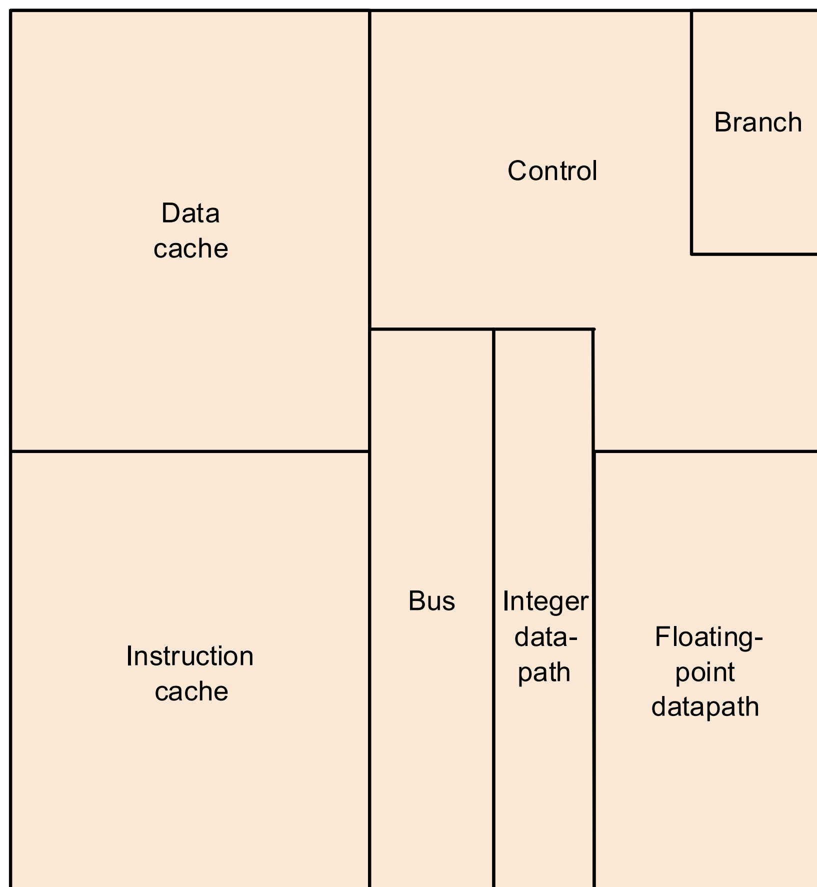
۱–۳– لایه‌های زیرین سخت‌افزار

در بخش قبل ما برای فهمیدن قسمتهای زیرین یک نرم‌افزار به آنچه که در زیر برنامه شما قرار دارد نگاه انداختیم، حال می‌خواهیم چنین کاری را در رابطه با سخت‌افزار انجام دهیم. اگر یک کامپیوتر رومیزی را در نظر بگیریم می‌بینیم که دارای کیبورد، موس، صفحه نمایش و یک کیس که سخت-افزارهای زیادی را در داخل خود نگه‌داری می‌کند، می‌باشد. این کامپیوتر معمولاً دارای یک اتصال شبکه نیز هست که ارتباط کامپیوتر را با کامپیوترهای دیگر، پرینتر و دیسکها برقرار می‌کند. دو ماجول اصلی کامپیوترها قطعات ورودی نظیر کیبورد و قطعات خروجی نظیر صفحه نمایش و پرینترها می-باشند. همان طور که از نامشان پیدا است، با استفاده از ورودی، کامپیوتر تغذیه می‌شود و داده‌ها در اختیار آن قرار می‌گیرد و با استفاده از خروجی‌ها نتایج حاصل از محاسبات کامپیوتر از آن خارج شده و در اختیار کاربر قرار می‌گیرد. در فصل ۸ وسایل ورودی و خروجی با جزئیات بیشتری شرح داده خواهند شد.

اجازه دهید تا یک مرور مقدماتی به سخت‌افزار کامپیوتر داشته باشیم. وقتی که ما کیس یک کامپیوتر را باز می‌کنیم، یک بوردی (board) را مشاهده می‌کنیم که با پلاستیک نازک سبز رنگی پوشیده شده است و بر روی آن تعداد زیادی مستطیل‌های کوچک سیاه و خاکستری قرار دارند. این بورد، مادربورد (mother board) نام دارد. مستطیل‌های کوچک روی مادربورد IC (Integrated circuit) یا چیپ نامیده می‌شوند. این بورد دارای سه قسمت مهم است، قسمتی که به قطعات ID (مثل درایو فلاپی و دیسک سخت) وصل می‌شود، حافظه و پردازنده. تعدادی بورد دیگر نیز به مادربورد وصل می‌شوند که از آن جمله می‌توان به کارت شبکه و کارت گرافیکی اشاره نمود.

حافظه محلی است که برنامه‌ها به هنگام اجرا بر روی آن قرار می‌گیرند. همچنین حافظه داده‌های مورد نیاز برنامه در حال اجرا را نیز نگهداری می‌کند. حافظه معمولاً بر روی دو بورد کوچک که تقریباً در وسط مادربورد نصب می‌شوند، قرار می‌گیرد.

پردازنده قسمت فعال بورد است که دستورات برنامه را اجرا می‌کند. پردازنده اعداد را جمع می‌کند، تست می‌کند، به قطعات IO علامت می‌دهد تا فعال شوند و ... . پردازنده به صورت یک مربع بزرگ بر روی مادربورد قرار دارد. معمولاً پردازنده را به اسم CPU می‌شناسیم که همان واحد پردازش مرکزی Central Processing Unit می‌باشد. شکل ۱–۳ محتویات داخلی یک CPU را نشان می‌دهد.



شکل ۱–۳: محتویات داخلی CPU

پردازنده از دو بخش اصلی تشکیل می‌شود:

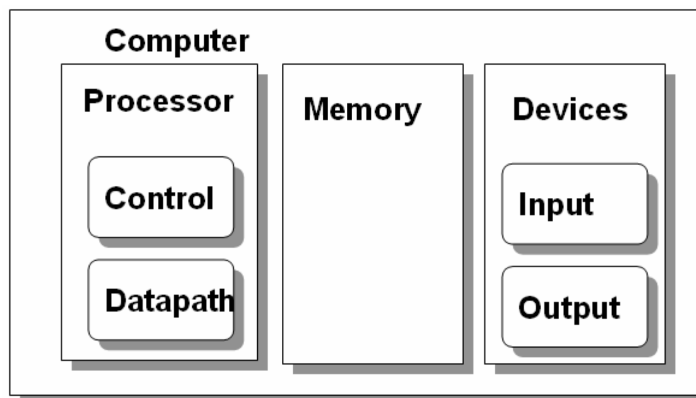
بخشdata path و بخش کنترل (Control) که این دو قدرت و تفکر کامپیوتر را می‌سازند.data path عملیات ریاضی را انجام می‌دهد و بخش کنترل به data path، حافظه و قطعات IO دستور می‌دهد که

چه کاری انجام دهند. بر اساس اینکه دستوری که می‌خواهد انجام شود، چه عملیاتی می‌خواهد انجام دهد، واحد کنترل وظیفهٔ سایر بخش‌ها را مشخص می‌کند.

فصل ۵ در مورد بخش‌های data path و کنترل صحبت خواهد کرد و فصل ۶ در مورد تغییراتی که باید در طراحی این دو بخش صورت بگیرد تا به سرعت بالاتری برسیم صحبت خواهد نمود.

شکل ۱-۳ مربوط به پردازنده پنتیوم اینتل می‌باشد. حجم سیلیکون استفاده شده برای این پردازنده ۹۱ میلی متر مربع بوده و در ساخت آن حدود ۳/۳ میلیون ترانزیستور استفاده شده است. حافظه کش cache این پردازنده تقریباً یک میلیون عدد از ۳ میلیون ترانزیستور را به خود اختصاص داده است. فصل ۷ توضیح خواهد داد که چرا کش منابع زیادی را مصرف می‌کند. قسمتهای دیگر چیپ در فصل-های بعدی توضیح داده خواهد شد. مدار پیش بینی دستور انشعاب (branch prediction) در فصل ۶ و گذرگاه سیستم Bus در فصل ۸ پوشش داده می‌شوند.

نکته مهم: ۵ بخش کلاسیک یک کامپیوتر عبارتند از ورودی، خروجی، حافظه‌data path و بخش کنترل که این دو بخش آخری معمولاً ترکیب شده و پردازنده (processor) نامیده می‌شود. شما می-توانید هر بخشی ازهر کامپیوتری را (چه کامپیوترهای قدیمی و چه کامپیوترهای فعلی) در یکی از این دسته‌ها قرار دهید. این پنج بخش در شکل ۱-۴ نشان داده شده‌اند.



شکل ۱-۴: پنج بخش کلاسیک یک کامپیوتر

پردازنده دستورات و داده‌ها را از حافظه دریافت می‌نماید؛ ورودی داده‌ها را در داخل حافظه می‌نویسد و خروجی داده‌ها را از حافظه می‌خواند. بخش کنترل، سیگنالهایی که عملیات data path، حافظه، ورودی و خروجی را مشخص می‌کنند، را تولید می‌نماید.

بر روی مادبورد معمولاً حافظه‌های نوع DRAM قرار دارد. DRAM علامت اختصاری عبارتDynamic Random Access Memory می‌باشد. برای نگهداری داده‌ها و دستورات یک برنامه

چندین DRAM در کنار یکدیگر استفاده می‌شوند. در مقایسه با حافظه‌های با قابلیت دسترسی ترتیبی مانند نوارهای مغناطیسی که زمان دستیابی به قسمت‌های مختلف با هم فرق دارد، قسمت RAM از عبارت DRAM بیانگر این مطلب است که دسترسی به قسمت‌های مختلف این حافظه به یک اندازه طول می‌کشد. حافظهٔ DRAM مانند یک بافر بزرگ یا یک آرایه بزرگ برای ذخیره داده‌ها مورد استفاده قرار می‌گیرد.

۱-۳-۱- معماری مجموعه دستورات

نکته‌ای که در اینجا باید به آن اشاره شود این است که ما هر چقدر وارد جزئیات سخت‌افزار و نرم‌افزار نشویم، به مدلهای سطح بالای ساده‌ای از سخت‌افزار و نرم‌افزار خواهیم رسید. استفاده از این روش (مدلهای لایه‌ای) یک تکنیک اساسی برای طراحی سیستم‌های کامپیوتری بسیار هوشمند می‌باشد. یکی از مهمترین سطوح تجرد[1]، واسط بین سخت‌افزار و نرم‌افزار سطح پایین[2] می‌باشد. به دلیل اهمیت این موضوع یک اسم ویژه به آن اختصاص داده شده است. معماری مجموعه دستورات (ISA[3]) ، یا به طور ساده معماری یک ماشین. ISA شامل همه چیزهایی که برنامه‌نویسان برای نوشتن برنامه‌های باینری زبان ماشین به آنها نیاز دارند می‌باشد. این نیازمندی‌ها شامل دستورات، قطعات I/O و غیره می‌باشد. اجزای مختلف یک معماری در فصل‌های ۳ ، ۴ ، ۷ و ۸ بحث خواهد شد. این واسط استاندارد به طراحان کامپیوتر اجازه می‌دهد تا مستقل از سخت‌افزار در مورد عملکردهای یک کامپیوتر صحبت کنند. به طور مثال ما می‌توانیم در مورد عملکرد یک ساعت دیجیتال (مانند نمایش زمان، زنگ اخطار و ....) مستقل از اینکه سخت‌افزار آن چیست (صفحه نمایش، دکمه‌های پلاستیکی، چرخ دنده‌ها و ...) صحبت کنیم. به همین دلیل طراحان کامپیوتر بین یک پیاده‌سازی[4] و یک معماری تفاوت قائل می‌شوند. یک پیاده‌سازی سخت‌افزاری است که سطوح تجرد معماری را فراهم می‌کند.

نکته مهم: نرم‌افزار و سخت‌افزار هر دو شامل لایه‌های سلسله مراتبی هستند، که لایه‌های زیرین جزئیات را از لایه‌های بالاتر مخفی نگه می‌دارند. پایه و اساس سطح تجرد این است که طراحان سخت‌افزار و نرم‌افزار پیچیدگی طراحی سیستم‌های کامپیوتری را کاهش دهند. یکی از واسط‌های کلیدی بین سطوح تجرد ISA می‌باشد: واسط بین سخت‌افزار و نرم‌افزار سطح پایین. این واسط امکان

1 - Abstract
2 - Low level software
Instruction Set Architecture–3
4 - Implementation

ایجاد پیاده‌سازی‌های مختلف با قیمت و سرعت متفاوت که نرم‌افزار یکسانی را اجرا می‌کنند در اختیار قرار می‌دهد.

استفاده‌کننده‌های macintash اثر تغییر ISA را درک می‌کنند. برنامه‌هایی که برای معماری Power PC طراحی شده‌اند بر روی ماشین‌هایی که بر اساس 6800 ساخته شده‌اند اجرا نمی‌شوند، و برنامه‌هایی مبتنی بر 6800 ، به درستی بر روی Power PC اجرا نمی‌شوند. در مقابل خانواده 86*80 شرکت اینتل پیاده‌سازی‌های مختلفی از یک معماری را در اختیار قرار می‌دهند. برنامه‌های نوشته شده برای 8086 اولیه در سال ۱۹۷۸ می‌توانند بر روی Pentium Pro اجرا شوند. Intel در طی سالیان قابلیت‌های زیادی را به سیستم‌های قبلی افزوده است. اما همهٔ پردازنده‌های نسل بعدی برنامه‌های نوشته شده بر روی پردازنده‌های نسل‌های قبلی را اجرا می‌کنند.

۱-۳-۲- محل مطمئن برای نگهداری داده‌ها

اگر تغذیه (برق) کامپیوتر قطع شود، همه چیز از بین می‌رود و این به دلیل این است که حافظه‌های داخل کامپیوتر فرار هستند و با قطع برق اطلاعات آنها از بین می‌رود. در مقام مقایسه یک نوار کاست با قطع برق اطلاعات خود زا از دست نمی‌دهد چون تکنولوژی آنها با حافظه کامپیوتر فرق می‌کند. در نوار کاست از قطعات مغناطیسی استفاده می‌شود.

کامپیوتر معمولاً دارای دو نوع حافظه است: حافظه اصلی و حافظه ثانوی

حافظه اصلی یا اولیه معمولاً از نوع DRAM بوده و اطلاعات آنها با قطع شدن برق از بین می‌رود ولی حافظه جانبی یا ثانویه از نوع مغناطیسی بوده و اطلاعات آنها با قطع شدن برق از بین نمی‌رود. بنابراین برای نگهداری داده‌ای که نباید از بین بروند از حافظه ثانویه استفاده می‌شود. به دلیل اینکه درحافظه-های ثانویه از قطعات مکانیکی استفاده می‌شود، سرعت آنها نسبت به DRAM ها خیلی پایین‌تر است. زمان دسترسی به داده درحافظه ثانویه (هارد دیسک) معمولا بین ۵ تا ۲۰ میلی ثانیه است در حالی که این زمان برای DRAM در حدود ۵۰ تا ۱۰۰ نانو ثانیه است. بنابراین DRAM حدود ۱۰۰۰۰۰ مرتبه سریعتر از هارد دیسک است.

هزینه تولید هارد دیسک‌ها نسبت به DRAM پایین‌تر است بنابراین قیمت آنها نسبت به DRAM کمتر است. در سال ۱۹۹۷ قیمت برای یک مگابایت از دیسک ۵۰ برابر کمتر از DRAM بود.

بنابراین دیسک‌های مغناطیسی و حافظه اصلی سه فرق عمده با هم دارند: دیسک‌ها غیر فرارند، به دلیل اینکه مغناطیسی هستند. سرعت دیسک‌ها کمتر است چون قطعات مکانیکی دارند و قیمت دیسک‌ها کمتر است چون هزینه تولید آنها در مقایسه با DRAM ها پایین‌تر است.

۱-۴- مدارهای مجتمع

تکنولوژی ساخت پردازنده‌ها و حافظه‌ها در طول سالیان همواره در حال پیشرفت بوده است و به همین دلیل پردازنده‌ها وحافظه‌ها همواره در حال بهبود بوده‌اند. جدول ۱-۱ تکنولوژی‌هایی را نشان می‌دهد که در سالهای مختلف مورد استفاده قرار گرفته‌اند. در این جدول یک تخمین از افزایش سرعت به ازای یک هزینه واحد آورده شده است.

جدول ۱-۱: افزایش سرعت در هزینه واحد برای تکنولوژیهای مختلف

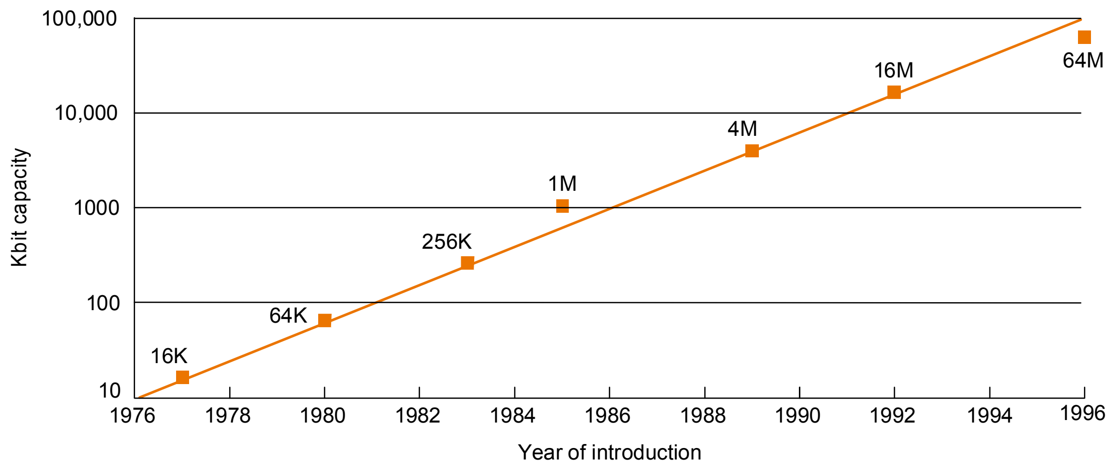
Year	Technology used in computers	Related Performance / unit cost
1951	Vacuum tub	1
1965	Transistor	35
1975	Integrated Circuit	900
1995	Very large-scale Integrated circuit	2,400,000

این بخش در مورد تکنولوژی که سرعت کامپیوتر را از سال ۱۹۷۵ تحت تأثیر قرار داده و در سالهای آتی نیز این اتفاق خواهد افتاد، صحبت خواهد نمود. چون تکنولوژی نشان می‌دهد که کامپیوترها چه کارهایی را می‌توانند انجام دهند و با چه سرعتی در حال پیشرفت می‌باشند، بنابراین ما اعتقاد داریم که همهٔ متخصصین کامپیوتر باید با اصول مدارهای مجتمع آشنا شوند.

یک ترانزیستور به زبان ساده یک سوئیچی است که دارای دو وضعیت روشن و خاموش (on و off) بوده و با جریان برق کنترل می‌شود. یک مدار مجتمع حدوداً ده‌ها تا صدها عدد از این ترانزیستورها را در داخل یک چیپ ترکیب می‌نماید برای نشان دادن نرخ افزایش سرسام‌آور تعداد ترانزیستورها از صدها عدد به میلیون‌ها عدد در داخل یک چیپ به آنها VLSI [1] گفته می‌شود.

شکل ۱-۵ میزان افزایش ظرفیت DRAM ها را از سال ۱۹۷۷ به بعد نشان می‌دهد. همان طور که مشاهده می‌شود ظرفیت DRAMها تقریبا هر سه سال، چهار برابر شده است که در نتیجه در طول بیست سال ظرفیت DRAM ها حدوداً ۱۶۰۰۰ برابر شده است! این میزان افزایش قابل توجه در سرعت و ظرفیت مدارهای مجتمع، طراحی سخت‌افزار و نرم‌افزار را تحت تأثیر قرار می‌دهد و همین عاملی است که فهمیدن مدارهای مجتمع را ضروری می‌سازد.

[1] - Very Large Scale Integrated



شکل ۱–۵: میزان افزایش ظرفیت DRAM ها در طول زمان. واحد محور y ها Kbits می‌باشد.

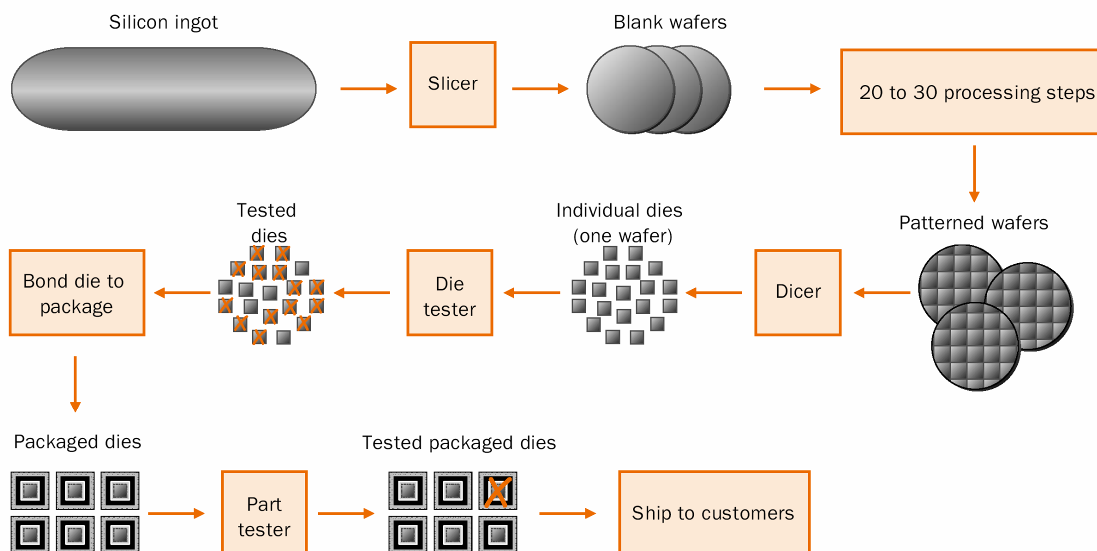
حال که ارتباط افزایش حجم حافظه‌ها را با تکنولوژی متوجه شدیم در این قسمت می‌خواهیم روند ساخته شدن یک چیپ را مورد بررسی قرار دهیم. ساختن یک چیپ از سیلیکون شروع می‌شود (سیلیکون ماده‌ای است که در سنگ یافت می‌شود و جزو عناصر گروه چهارم جدول تناوبی مندلیف می‌باشد). چون سیلیکون جریان الکتریسیته را به خوبی عبور نمی‌دهد به آن عنصر نیمه هادی گفته می‌شود. با استفاده از یک پروسهٔ شیمیایی مخصوص این امکان وجود دارد که موادی به سیلیکون اضافه نمائیم تا در سطح آن محل‌هایی ایجاد کنیم که در آن محل‌ها بتوانیم یکی از سه قطعه زیر را بسازیم:

- مواد هادی جریان الکتریسیته (شبیه به مس و آلومینیوم)
- مواد عایق(مانند پلاستیک و شیشه)
- فضایی (area) که تحت شرایط خاصی می‌تواند هادی و یا عایق (مانند یک کلید) باشد

ترانزیستورها در طبقه آخر قرار می‌گیرند. یک مدار VLSI شامل میلیونها هادی، عایق و سوئیچ می‌باشد که بر روی یک فضای کوچک سیلیکونی ساخته می‌شوند. پروسهٔ ساخت مدارهای مجتمع بر روی هزینه چیپ ها تأثیر زیادی دارد و بنابراین برای طراحان کامپیوتر اهمیت زیادی دارد. شکل ۱–۶ این پروسه ساخت را نشان می‌دهد. این پروسه از یک شمش سیلیکونی شروع می‌شود. سپس یک شمش برش خورده و به قطعات کوچکتر و نازکتری به نام Wafer تقسیم می‌شود. این ویفرها سپس چند مرحله پروسه را طی می‌کنند، که در آن پروسه‌ها یک سری موار شیمیایی بر روی هر ویفر قرار می‌گیرد که ترانزیستورها، هادیها و عایق‌ها را ایجاد می‌کنند، بعد از طی کردن این مراحل ویفرها برش داده می‌شوند و به قطعات کوچکتری به نام die تبدیل می‌شوند. بعد از این مرحله dieهای خراب کنار گذاشته می‌شوند. تعداد die های سالم به کل dieهای یک ویفر yield نامیده می‌شود. بعد از تست die ها، هر کدام از dieهای سالم در داخل یک package، به پین‌های ورودی و خروجی وصل می‌شود این

پروسه bonding نامیده می‌شود. Dieهای package شده دوباره تست می‌شوند چون ممکن است هنگام package کردن خرابی‌هایی اتفاق بیافتد. پس از تست شدن، pakage به مشتری فروخته می‌شود.

توجه: یکی از مسائل جدید در طراحی کامپیوتر توان تلفاتی می‌باشد. اهمیت توان اتلافی فقط برای کاربردهای قابل حمل (Protable) که طول عمر باتری در آنها مهم است (مانند کامپیوترهای کیفی) نیست بلکه برای کامپیوترهای رومیزی که نیز با افزایش کلاک اهمیت مسأله توان بیشتر می‌شود. پردازنده Alpha 21264 در فرکانس 600 مگاهرتز به طور شگفت‌انگیزی فقط 72 وات توان مصرف می‌کند. توان اتلافی یکی از مسائلی است که می‌تواند سرعت پردازنده‌ها را محدود کند.



شکل ۱–۶: پروسه ساخت چیپ

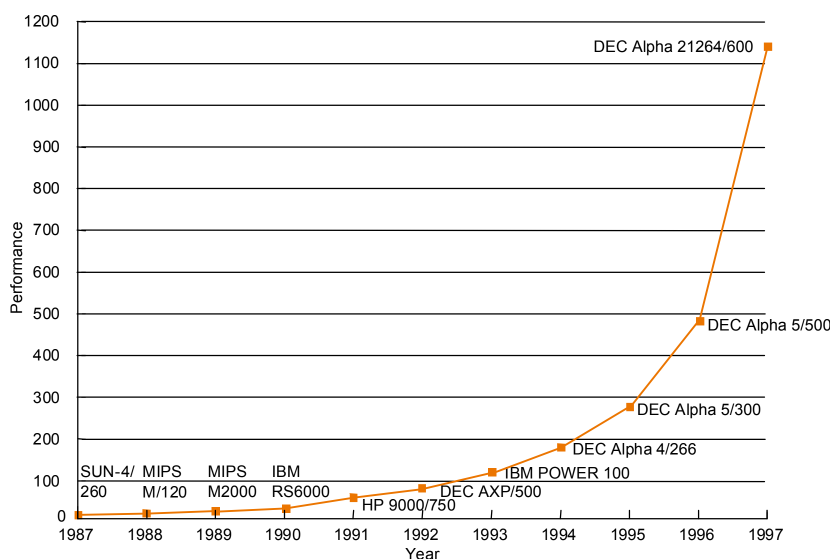
۱–۵– باورهای غلط (fallacies) و اشتباهات (Pitfalls)

هدف از داشتن یک بخش تحت این عنوان که در آخر هر فصل وجود دارد این است که مطالبی را توضیح دهد که ممکن است شما آنها را اشتباه متوجه شده باشید: ما این باور غلط را fallacy نامگذاری خواهیم کرد. وقتی که در مورد یک باور غلط صحبت می‌کنیم برای توضیح دادن آن یک مثال خواهیم آورد. همچنین ما در مورد اشتباهات (Pitfalls) صحبت خواهیم کرد. اغلب این اشتباهات

تعمیم دادن اصولی است که فقط در محدودهٔ کوچکتری درست می‌باشند. هدف از این بخش این است که به شما کمک کند تا جلوی اشتباهات خود را در ماشین‌هایی که استفاده و یا طراحی می‌کنید بگیرید.

اشتباه: چشم پوشی کردن از پیشرفت سخت‌افزار هنگامی که یک ماشین جدید را طراحی می‌کنیم اشتباه است.

فرض کنید که شما می‌خواهید یک ماشین را در مدت ۳ سال تولید کنید و ادعا دارید که این ماشین یک فروشنده فوق‌العاده خواهد بود چون سرعت آن ۳ برابر سرعت ماشین‌های دیگر که امروزه وجود دارند خواهد بود. متأسفانه ماشین شما احتمال دارد فروشنده ضعیفی باشد به دلیل اینکه افزایش سرعت برای صنعت (افزایش سرعت سخت‌افزار) ممکن است به ماشین‌هایی منجر شود که سرعتشان مثل سرعت ماشین شما باشد. به طور مثال فرض کنید ۵۰٪ رشد سالیانه سرعت داشته باشیم (به دلیل پیشرفت تکنولوژی ساخت سخت‌افزار)، در این صورت برای ماشینی که امروزه سرعت آن x می‌باشد می‌توان انتظار داشت که پس از ۳ سال سرعتی در حدود $1.5^3 x = 3.4x$ خواهد داشت. بنابراین ماشین شما مزیت سرعتی نخواهد داشت! بسیاری از پروژه‌ها در شرکتهای کامپیوتری به این دلیل تعطیل شدند که یا این قانون را در نظر نگرفتند (پیشرفت تکنولوژی ساخت سخت‌افزار) و یا اینکه دیرتر از موعد خودشان تمام شدند و سرعت ماشینی که با تأخیر به بازار آمد از میانگین سرعت صنعت پایین‌تر بود. این واقعیت ممکن است در همهٔ صنایع وجود داشته باشد ولی بهبود سریع هزینه بر سرعت (Cost / Performance) در صنعت کامپیوتر آن را به یک امر بسیار مهم تبدیل کرده است. شکل ۱–۷ میزان افزایش سرعت پردازنده‌ها در طول زمان را نشان می‌دهد.



شکل ۱–۷: افزایش سرعت کامپیوترها بین سالهای ۱۹۸۷ تا ۱۹۹۷

۱-۶- موضوع واقعی: ساختن چیپ‌های پنتیوم (Pentium)

در انتهای هر فصل این کتاب، بخشی تحت عنوان موضوع واقعی (Real stuff) وجود دارد که مطالب موجود در همان فصل را به کامپیوترهایی که در عمل از آنها استفاده می‌کنید، مرتبط می‌سازد. این بخش‌ها همیشه تکنولوژی‌هایی را که برای کامپیوترهای IBM PC مورد استفاده قرار می‌گیرد توضیح خواهند داد و اغلب تکنولوژی Apple Macintosh را نیز بحث خواهند کرد. برای این فصل ما مفاهیم مدارهای مجتمع را به چیپ‌هایی که در IBM PC ها استفاده می‌شوند، مرتبط خواهیم نمود.

۱-۷- صحبت پایانی

هر چند پیشگویی این امر مشکل است که واقعاً بهبود هزینه بر سرعت کامپیوترها در آینده چگونه خواهد بود، ولی آنها مطمئناً بهتر از کامپیوترهای فعلی خواهند بود. برای مشارکت در این پیشرفت، طراحان کامپیوتر و برنامه‌نویسان باید مسائل زیادی را بفهمند و یاد بگیرند.

هر جفت طراحان سخت‌افزار و نرم‌افزار سیستم‌های کامپیوتری را به صورت لایه‌ای طراحی می‌کنند که در آن لایه‌های پایین‌تر جزئیات را از دید لایه‌های بالاتر مخفی نگه می‌دارند. این اصول سطح تجرد برای فهمیدن سیستم‌های کامپیوتری، امروزه ضروری است. شاید مهمترین مثال از سطح تجرد، واسط بین سخت‌افزار و نرم‌افزار سطح پایین می‌باشد که ISA نامیده می‌شود. با فرض ثابت نگه داشتن ISA پیاده‌سازی‌های مختلفی از یک معماری با هزینه و سرعت متفاوت را ممکن می‌سازد. همه این پیاده-سازی‌ها نرم‌افزار یکسانی را اجرا می‌کنند. از طرف دیگر ممکن است این معماری مانع پیشرفت گردد و نیاز به این داشته باشد که واسط مورد نظر تغییر کند. تکنولوژی‌های کلیدی برای پردازنده‌های مدرن، سیلیکون و کامپایلر می‌باشند. به طور واضح شما باید ویژگیهای سیلیکون و کامپایلر را درک نمائید. چیزی که اهمیت آن کمتر از فهمیدن مفهوم مدارهای مجتمع نیست فهمیدن سرعت تغییرات مورد انتظار تکنولوژی‌های ساخت IC می‌باشد. یک مثال مرتبط با این مسأله این است که سرعت DRAM ها هر سه سال ۴ برابر می‌شود. در حالی که سیلیکون عامل مؤثری در پیشرفت سریع سخت‌افزار می-باشد، ایده‌های جدید در طراحی کامپیوترها، هزینه بر سرعت کامپیوترها را بهبود داده است. دو مورد از ایده‌های کلیدی، به کارگیری موازی سازی در کامپیوترها و سلسله مراتب حافظه می‌باشد. نوعاً موازی سازی از طریق پایپلاین و به کارگیری مفهوم دسترسی محلی به حافظه از طریق حافظه‌های کش می-باشد.

راهنمای فصول این کتاب

همان طور که توضیح داده شد پنج قسمت کلاسیکی برای کامپیوتر وجود دارد: datapath، بخش کنترل، حافظه، ورودی و خروجی این پنج قسمت به عنوان یک قالب کاری (frame work) برای قسمت‌های باقی مانده این کتاب مورد استفاده قرار می‌گیرد:

- مسیر داده (datapath): فصول ۴و ۵ و۶
- بخش کنترل: فصول ۵ و۶
- حافظه: فصل ۷
- ورودی: فصل ۸
- خروجی: فصل ۸

فصل ۶ توضیح خواهد داد که چگونه پایپلاین در پردازنده موازی‌سازی انجام می‌دهد و فصل ۷ در مورد اینکه چگونه سلسله مراتب حافظه خاصیت محلی بودن را بکار می‌گیرد، بحث خواهد نمود. بقیه فصل‌ها مقدمات و مباحث تکمیلی در مورد این موضوعات را مطرح می‌کنند. فصل ۲ در مورد سرعت پردازنده‌ها صحبت خواهد کرد و بنابراین توضیح می‌دهد که چگونه یک کامپیوتر را ارزیابی کنیم. فصل ۳ در مورد مجموعه دستورات که همان واسط بین کامپایلرها و ماشین است صحبت خواهد کرد و تأکید خواهد نمود که کامپایلرها و زبانهای برنامه‌نویسی چه نقشی در استفاده از خواص مجموعه دستورات خواهند داشت.

فصل ۹ این کتاب با بحثی در مورد سیستم های چند پردازنده‌ای این کتاب را به پایان خواهد برد.

۱-۸- چشم‌انداز تاریخی و مطالعه اضافی

در این کتاب در آخر هر فصل بخشی تحت این عنوان وجود دارد. در این بخش‌ها ما ممکن است توسعهٔ یک ایده را از طریق یک‌سری از ماشین‌ها بررسی کنیم یا بعضی از پروژه‌های مهم را توضیح دهیم. ما در هر مورد مراجع را معرفی خواهیم نمود و اگر علاقمند بودید می‌توانید به آنها مراجعه کنید و بیشتر آنها را ارزیابی کنید. در حالت کلی مطالب این بخش در مورد مطالب همان فصل مربوطه خواهد بود و مباحث تاریخی و تکمیلی را در مورد مطالب آن فصل بیان خواهد نمود.

۱-۹- کلمات و اصطلاحات کلیدی

لیستی از کلمات و اصطلاحات در آخر هر فصل و در پیوست آورده شده است. این کلمات ایده‌های کلیدی بحث شده در آن فصل و پیوست را منعکس می‌کند. اگر شما معنای کلمه و اصطلاح گفته شده

در این قسمت را هنوز نفهمیده‌اید، می‌توانید به واژگان کلمات در آخر کتاب مراجعه کنید. همهٔ کلمات و اصطلاحات کلیدی در این قسمت توضیح داده شده است.

فصل دوم

ارزیابی کارآیی پردازنده‌ها

۲-۱- مقدمه

این فصل توضیح خواهد داد که چگونه کارآیی[6] (سرعت) یک کامپیوتر را اندازه بگیریم و آن را گزارش کنیم. در این فصل سعی خواهیم نمود که به سؤالات زیر جواب دهیم:

- چرا کارآیی مهم است؟
- چگونه ما می‌توانیم کارآیی را به دقت تعریف کنیم؟
- چگونه طراحی سخت‌افزار کارآیی نرم‌افزار را تحت تأثیر قرار می‌دهد؟
- چگونه در دنیای واقعی کارآیی اندازه گرفته شود؟
- چرا بعضی از سخت‌افزارها بهتر از بقیه عمل می‌کنند؟
- کدام یک از فاکتورهای کارآیی به سخت‌افزار وابسته‌اند؟
- ما برای اجرا شدن سریع‌تر یک برنامه به یک ماشین جدید نیاز داریم یا نیازمند یک سیستم-عامل جدید هستیم؟
- مجموعه دستورات یک ماشین چگونه کارآیی را تحت تأثیر قرار می‌دهند؟

در این فصل عاملهای تعیین کننده در کارآیی کامپیوترها مورد بحث قرار خواهد گرفت. بررسی کردن کارآیی به این دلیل مهم است که کارآیی سخت‌افزار اغلب کلیدی‌ترین اثر را در کل سیستم متشکل از سخت‌افزار و نرم‌افزار برعهده دارد. ما معمولاً در این فصل کارآیی و سرعت را به جای هم استفاده می‌کنیم. ولی در بعضی از موارد این دو عبارت کاملاً معادل هم نیستند. به عنوان مثال اگر هدف ما استفاده از ماشینی باشد که تعداد کار بیشتری را در واحد زمان انجام دهد، ماشینی که سریعتر است بعضی مواقع نمی‌تواند هدف ما را برآورده کند و ما مجبور هستیم در این مواقع از ماشینی استفاده کنیم که قابلیت‌های بیشتری داشته باشد (کاراتر باشد). ما در این کتاب اغلب از اصطلاح کارآیی استفاده خواهیم کرد.

اظهار نظر کردن در مورد کارآیی یک سیستم واقعاً کار مشکلی است. سیستم‌های بزرگ نرم‌افزاری با جزئیات زیاد به همراه بازهٔ وسیعی از تکنیک‌های افزایش سرعت که توسط طراحان سخت‌افزار به کار گرفته می‌شوند، از جمله مواردی هستند که صحبت کردن دربارهٔ کارآیی را مشکل می‌نمایند.

تقریبا کار غیر ممکنی است که برگه راهنمایی از مجموعه دستورات یک کامپیوتر را بردارید و با یک برنامه مشخص، تعیین نمایید که آن کامپیوتر برنامه شما را چقدر سریعتر اجرا خواهد کرد. در واقع

[6] - Performance

برای برنامه‌های کاربردی مختلف، ممکن است مترهای مختلفی مناسب باشد و قسمتهای مختلف یک کامپیوتر ممکن است نقش متفاوتی در تعیین سرعت کل سیستم داشته باشند.

البته در انتخاب بین کامپیوترهای مختلف، کارآیی تقریباً همیشه یک مشخصه مهم به حساب می‌آید. اندازه‌گیری و مقایسه ماشین‌های مختلف برای خریداران و در نتیجه برای طراحان یک امر حیاتی محسوب می‌شود که این مطلب را فروشندگان کامپیوترها به خوبی درک می‌کنند. اغلب، فروشندگان دوست دارند که شما کامپیوترهای آنها را خیلی ساده نگاه کنید و این نگاه کردن ممکن است بدون توجه به نیاز خریدار، که برنامه‌های کاربردی خودش را بر روی آن اجرا خواهد کرد انجام بگیرد. در بعضی مواقع ادعاهایی در مورد کامپیوترها می‌شود که دید مفیدی برای بعضی از برنامه‌های کاربردی در اختیار قرار نمی‌دهند. بنابراین فهمیدن چگونگی اندازه گیری کارآیی یک کامپیوتر و محدودیتهای اندازه‌گیری در انتخاب یک ماشین، بسیار مهم می‌باشند.

علاقمندی ما نسبت به کارآیی کامپیوتر ماورای این است که فقط کارآیی کامپیوتر را از بیرون آن اندازه بگیریم. ما نیاز به این داریم که بفهمیم چه چیزهایی کارآیی یک کامپیوتر را تحت تأثیر قرار می‌دهند. فهمیدن اینکه چرا بعضی قسمتهای نرم‌افزار همان طور که ما انتظار داریم کار نمی‌کنند، چرا یک مجموعه از دستورات می‌تواند به گونه‌ای پیاده سازی شود که بهتر عمل کند، و اینکه چگونه بعضی از قابلیتهای سخت‌افزار کارآیی را تحت تأثیر قرار می‌دهند، همگی از مسائلی است که ما علاقمند به پیدا کردن جواب مناسب برای آنها هستیم.

به طور مثال برای بهبود سرعت یک سیستم نرم‌افزاری، ممکن است احتیاج داشته باشیم که فاکتورهای جدید سخت‌افزاری را که تأثیر بر کل سیستم دارند را بشناسیم و میزان اهمیت این فاکتورها را درک نمائیم. این فاکتورها ممکن است شامل موارد زیر باشد: برنامه‌های کاربران کدام دستورات ماشین را بیشتر استفاده می‌کنند، سخت‌افزار چگونه این مجموعه دستورات را پیاده‌سازی می‌کند و سیستم‌های IO و حافظه چقدر خوب کار می‌کنند؟ فهمیدن اینکه چگونه تأثیر این فاکتورها را بر کارآیی تعیین کنیم، برای درک انگیزه‌های پشت صحنه طراحی بعضی از قابلیت‌های ویژه ماشین که ما آنها را در فصل‌های آتی خواهیم دید، خیلی مهم می‌باشند. این فصل روشهای اندازه‌گیری کارآیی را توضیح خواهد داد. در بخش ۲–۲ ما معیارهای اندازه‌گیری کارآیی را از دو دیدگاه استفاده کننده کامپیوتر و طراح سخت افزار بیان خواهیم کرد. در بخش ۲–۳ مشاهده خواهیم کرد که چگونه این معیارها به هم وابستگی دارند و فرمولی جهت اندازه‌گیری کارآیی ارائه خواهد شد که در سراسر این کتاب از آن استفاده خواهد شد. بخش‌های ۲–۴ و ۲–۵ در مورد نحوه انتخاب benchmark هایی که برای ارزیابی

کارآیی کامپیوترها استفاده می‌شود صحبت خواهد کرد. همچنین یاد می‌گیریم که چگونه سرعت اجرای دسته‌ای از برنامه‌ها را به طور دقیق خلاصه و جمع‌بندی نمائیم و گزارش تهیه کنیم. بخش ۲–۶ یک دسته از benchmark های معمول استفاده شده برای CPUها را توضیح خواهد داد و با استفاده از این benchmark ها سرعت چند پردازنده اینتل را اندازه خواهد گرفت. سرانجام در فصل ۲–۷ ما به دسته‌ای از کج فهمی‌ها و دام‌هایی که معمولاً طراحان و تحلیل کننده‌های کارآیی را گرفتار می‌کنند، خواهیم پرداخت.

تعریف کارآیی

وقتی که گفته می‌شود کارآیی این کامپیوتر بالاتر از آن یکی است، منظور چیست؟ هر چند این سؤال ساده به نظرمی‌رسد ولی یک مقایسه با هواپیماهای مسافربری نشان می‌دهد که این سؤال چقدر قابل تأمل است.

شکل ۲–۱ تعدادی از هواپیماهای مسافربری را به همراه سرعت متوسط آنها، برد مسافتی و ظرفیتشان نشان می‌دهد. اگر بخواهیم ببینیم که کدام یک از هواپیماهای موجود در جدول کارآیی بالاتری دارد، اول باید کارآیی را تعریف کنیم. به طور مثال، با در نظر گرفتن اندازه‌گیری‌های مختلف، می‌بینیم که هواپیمایی که بالاترین سرعت متوسط را دارد concorde، و هواپیمایی که بیشترین ظرفیت را دارد 747 می‌باشد.

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

شکل ۲–۱: مقایسه کارآیی هواپیماهای مختلف

بنابراین اگر ما بخواهیم کارآیی این دو هواپیما را مقایسه کنیم. به دو صورت می‌توانیم این کار را انجام دهیم. می‌توانیم به این صورت تعریف کنیم که هواپیمای کاراتر هواپیمایی است که یک مسافر را از نقطه‌ای به نقطهٔ دیگر در کمترین زمان جابجا می‌کند. پس اگر شما علاقمند باشید که زودتر به مقصد برسید واضح است که concorde کاراتر می‌باشد. روش دوم مقایسه، برای این مثال تعداد مسافر جابجا شده در واحد زمان است. بر اساس این معیار همان طور که در ستون آخر جدول نیز نشان داده شده است 747 کاراتر می‌باشد.

به طور مشابه ما می‌توانیم کارآیی کامپیوترها را به طرق مختلف تعریف نمائیم. اگر شما یک برنامه را بر روی دو کامپیوتر مختلف اجرا کنید، خواهید گفت که کامپیوتری سریعتر است که آن برنامه را سریعتر اجرا می‌کند. ولی اگر برنامه خود را بر روی یک ترمینال در مرکز کامپیوتری اجرا کنید که دارای دو کامپیوتر است که به طور مشترک توسط چندین نفر استفاده می‌شوند در این صورت خواهید گفت که کامپیوتری سریعتر است که تعداد زیادتری برنامه را در روز اجرا کند. همان طور که یک کاربر دوست دارد که زمان پاسخ (response time) یا همان زمان اجرا (execution time) که مدت زمان بین شروع و خاتمه یک کار است، بر روی یک کامپیوتر برای برنامه او کمتر شود، در یک مرکز کامپیوتر، مدیر مرکز اغلب دوست دارد که تعداد کار انجام گرفته در واحد زمان (throughput) افزایش پیدا کند.

مثال

برای نشان دادن نحوۀ استفاده از ایده‌های جدید، مثالهای ویژه‌ای در سراسر این کتاب آورده شده است. ما ابتدا مثال را می‌آوریم و بعد به آن پاسخ می‌دهیم. سعی کنید که مثال را خودتان جواب دهید، اگر نتوانستید و یا به پاسخ خود اطمینان نداشتید جواب را مشاهده نمائید. مثالهایی که آورده می‌شود شبیه به مسائلی است که در آخر هر فصل آورده می‌شود. این اولین مثال از این کتاب است:

مثال: آیا تغییرات انجام شده زیرین بر روی یک سیستم کامپیوتری، **throughput** را زیاد می‌کند؟ زمان پاسخ دهی را کاهش می‌دهد؟ و یا هر دو؟

۱. جایگزین کردن پردازنده سیستم با یک نسخه سریعتر
۲. اضافه کردن یک پردازنده جدید به سیستمی که از چندین پردازنده برای کارهای مختلف استفاده می‌کند. به طور مثال سیستمی که عملیات رزواسیون یک خطوط هواپیمایی را انجام می‌دهد.

جواب: کم کردن زمان پاسخ دهی تقریباً همیشه **throughput** را بهبود می‌دهد. بنابراین در مورد اول، زمان پاسخ دهی و **throughput** هر دو بهتر می‌شوند. در مورد ۲، برنامۀ هیچ کسی سریعتر انجام نمی-گیرد، بنابراین فقط **throughput** بهتر می‌شود. اگر در مورد ۲، نیاز به زمان پاسخ دهی سریع همانند افزایش **throughput** اهمیت داشته باشد، سیستم ممکن است مجبور کند که درخواستها در داخل یک صف قرار بگیرند. در این حالت افزایش **throughput**، زمان پاسخ دهی را نیز کاهش خواهد داد چون زمان انتظار را در داخل صف کاهش خواهد داد. بنابراین در تعداد زیادی از سیستم‌های واقعی کامپیوتری، تغییر زمان پاسخ دهی یا **throughput**، اغلب دیگری را نیز تحت تأثیر قرار می‌دهد.

ما در مبحث کارآیی کامپیوترها، در ابتدا در طول چند فصل بر روی زمان پاسخ دهی تمرکز خواهیم کرد. در فصل ۸ که در ارتباط با سیستم‌های ورودی/ خروجی می‌باشد ما اندازه‌گیری‌های مربوط به throughput را نیز صحبت خواهیم کرد.

برای اضافه کردن سرعت (کارآیی)، ما می‌خواهیم که زمان پاسخ دهی یا زمان اجرا را کاهش دهیم، بنابراین می‌توان زمان اجرا و کارآیی را برای یک ماشین x به صورت زیر مرتبط کرد:

۱

کارآیی ماشین x = --------------------------

زمان اجرا بر روی ماشین x

فرمول فوق به صورت زیر نیز نوشته می‌شود:

$$performance_x = \frac{1}{(ExecutionTime)_x}$$

این به این معنی است که برای دو ماشین x و y ، اگر کارآیی x بیشتر از کارآیی y باشد داریم:

$$performance_x > performance_y$$
$$\Rightarrow \frac{1}{(ExecutionTime)_x} > \frac{1}{(ExecutionTime)_y}$$
$$\Rightarrow (ExecutionTime)_y > (ExecutionTime)_x$$

یعنی اینکه اگر x سریعتر از y باشد زمان اجرا بر روی y بیشتر از زمان اجرا بر روی x خواهد بود. در مبحث طراحی کامپیوتر، ما اغلب سرعت دو کامپیوتر مختلف را به صورت عددی مقایسه می‌کنیم. مثلاً می‌گوئیم ماشینx ، n بار سریعتر از ماشین y می‌باشد و می‌نویسیم.

$$\frac{performance_x}{performance_y} = n$$

اگرx، n بار سریعتر از y باشد در این صورت زمان اجرا بر رویy، n بار بیشتر از زمان اجرا بر روی x خواهد شد:

$$\frac{performance_x}{performance_y} = \frac{(ExecutionTime)_y}{(ExecutionTime)_x} = n$$

مثال:اگر ماشین A یک برنامه را در ۱۰ ثانیه و ماشین B آن را در ۱۵ ثانیه اجرا کند، ماشین A چند برابر از B سریعتر است؟

جواب: طبق فرمول بالایی می‌دانیم که اگر ماشین A ، n بار سریعتر از B باشد می‌توان نوشت:

$$\frac{performance_A}{performance_B} = \frac{(ExecutionTime)_B}{(ExecutionTime)_A} = n$$

بنابراین نسبت کارآیی به صورت زیر حساب می‌شود:

$$n = \frac{15}{10} = 1.5$$

بنابراین A ، 1.5 برابر سریعتر از B می‌باشد.

در مثال بالایی می‌توان گفت که ماشین B ، ۱/۵ برابر کندتر از ماشین A عمل می‌کند. چون داریم:

$$\frac{performance_A}{performance_B} = 1.5 \Rightarrow \frac{performance_A}{1.5} = performance_B$$

برای سادگی در مقایسه کارآیی کامپیوترها ما از عبارت سریعتر بودن استفاده خواهیم کرد. به دلیل اینکه کارآیی و زمان اجرا معکوس هم می‌باشند، افزایش کارآیی نیازمند این است که زمان اجرا را کاهش دهیم. برای جلوگیری از گیج شدن بین جمله‌های افزایش و کاهش، ما معمولاً وقتی که منظورمان "افزایش کارآیی" و "کاهش زمان اجراست"، می‌گوئیم "بهبود کارآیی"[۷] یا "بهبود زمان اجرا"[۸].

۲-۲- اندازه‌گیری کارآیی

اکنون زمان اندازه‌گیری کارآیی است، کامپیوتری که همان مقدار کار را در زمان کمتری انجام می‌دهد سریعتر است. زمان اجرای برنامه با ثانیه اندازه گرفته می‌شود. اما زمان می‌تواند بسته به اینکه ما چه چیزی را شمارش می‌کنیم، به گونه‌ای دیگر تعریف گردد. سرراست‌ترین تعریف زمان، زمان پاسخ‌دهی یا زمان انقضا می‌باشد[۹]. این اصطلاحات به معنی کل زمان لازم برای خاتمه یک کار است، که این زمان شامل دسترسی به حافظه دیسک، دسترسی به حافظه اصلی، دسترسی به IOها، سربار اجرای سیستم-عامل و هر زمان دیگری است.

[7] - Performance improvement
[8] - Execution time improvement
[9] - Elapsed time

کامپیوترها اغلب به صورت اشتراک زمانی استفاده می‌شوند و یک پردازنده ممکن است بر روی چندین برنامه به صورت همزمان کار کند. در این صورت سیستم ممکن است تلاشش در این راستا باشد که به جای کاهش زمان اجرای یک برنامه، throughput را کاهش دهد. بنابراین ما می‌خواهیم که بین زمان انتظار و زمانی که cpu فقط برای ما کار می‌کند تفاوت قائل شویم. زمان اجرای cpu (cpu execution time) یا به طور ساده cpu time مدت زمانی است که cpu صرف کار ما می‌کند و شامل زمان انتظار IO یا زمان اجرای برنامه‌های دیگر نمی‌شود. (به خاطر بیاورید که زمان پاسخ‌دهی به کاربر زمان انقضای برنامه است نه زمان cpu time).

cpu time می‌تواند به دو بخش تقسیم گردد: زمانی از cpu که صرف اجرای خود برنامه می‌شود (user cpu time) و زمانی از cpu که صرف اجرای سیستم‌عامل در ارتباط با این برنامه می‌شود. (معمولاً system cpu time).

سیستم عامل unix دستوری به نام time دارد که می‌تواند زمان انقضا را نشان دهد به طور مثال این دستور می‌تواند نتیجه‌ای مانند عبارت زیر را برگرداند:

90.7u 12.95 2:39 65%

در عبارت فوق زمان user cpu time 90.7 ثانیه، زمان system cpu time 12.9 ثانیه، زمان انقضا 2 دقیقه و 39 ثانیه (۱۵۹ ثانیه) و درصدی از زمان انقضا که مربوط به زمان cpu time می‌باشد به صورت زیر است:

$$\frac{90.7+12.9}{159}=0.65$$

یا 65 درصد. در این مثال بیش از $\frac{1}{3}$ از زمان انقضا صرف انتظار برای IO، اجرای برنامه‌های دیگر یا هر دو شده است.

بعضی مواقع وقتی صحبت از زمان اجرای cpu می‌شود ما معمولاً از زمان system cpu time چشم-پوشی می‌کنیم و این معمولاً به دلیل غیر دقیق بودن اندازه‌گیری سیستم‌عامل که خودش را اندازه می-گیرد و نیز به دلیل بی‌عدالتی در مواقعی است که می‌خواهیم system cpu time دو ماشین که سیستم-عاملهای مختلفی را اجرا می‌کنند مقایسه کنیم. از طرف دیگر، کدهای سیستمی در بعضی ماشینها، جزو کدهای کاربری ماشین دیگر می‌باشد. تقریباً هیچ برنامه‌ای نمی‌تواند بدون حضور سیستم‌عامل بر روی سخت‌افزار اجرا شود، بنابراین زمان اجرای برنامه را می‌توان مجموع user cpu time و system cpu timeدر نظر گرفت.

ما بین کارآیی محاسبه شده بر اساس زمان انقضا و براساس زمان cpu execution time فرق قائل خواهیم شد. ما عبارت کارآیی سیستم (system performance) را برای زمان انقضا برای یک سیستم خالی از برنامه به کار می‌بریم و عبارت cpu performanc را در ارتباط با user cpu time به کار خواهیم برد. ما در این فصل بر روی کارآیی cpu (cpu performance) تمرکز خواهیم کرد، هر چند بحث ما برای سرعت می‌تواند برای زمان انقضا و برای cpu time نیز مورد استفاده قرار گیرد.

هر چند که ما مانند کاربران کامپیوتر به زمان دقت می‌کنیم، وقتی ما جزئیات یک ماشین را بررسی می-کنیم راحت‌تر این است که معیارهای دیگری را برای سرعت در نظر بگیریم. علی الخصوص طراحان کامپیوتر ممکن است در مورد یک ماشین به این صورت فکر کنند که با استفاده از یک معیاری نشان دهند که سخت‌افزار آن کامپیوتر عملیات پایه‌ای را چقدر سریعتر انجام می‌دهد. تقریباً همهٔ کامپیوترها در ساختارشان از یک کلاک (clock) استفاده می‌کنند که این کلاک سیگنالی است که به صورت پریودیک تولید می‌شود و مشخص می‌کند اتفاقات در سخت‌افزار کی رخ دهند. به هر پریود کلاک یک سیکل کلاک (cycles , clocks , clock periods , clock ticks , ticks , clock cycle) گفته می‌شود. طراحان معمولاً طول پریود کلاک را به صورت زمان یک سیکل کامل کلاک (مانند ۲ نانو ثانیه) و یا به صورت نرخ کلاک (مانند ۵۰۰ مگاهرتز یا 500 mhz) که معکوس پریود کلاک است به کار می‌برند. در بخش بعدی ما ارتباط بین سیکل کلاک و زمان اجرای برنامه کاربر را توضیح خواهیم داد.

۲-۳- ارتباط بین معیارها (مترها)

کاربران و طراحان برای اندازه‌گیری سرعت از معیارهای مختلفی استفاده می‌کنند. اگر ما بتوانیم این معیارها را به هم مرتبط سازیم در این صورت می‌توانیم اثر یک تغییر در طراحی بر روی سرعت کامپیوتر را که توسط کاربر مشاهده می‌شود، مشخص نمائیم. فرمول زیر نحوه محاسبه زمان اجرای cpu را نشان می‌دهد. این فرمول ارتباط کلاک با زمان cpu را نشان می‌دهد.

پریود کلاک × تعداد کلاک‌های لازم برای اجرای برنامه = زمان اجرای cpu

چون فرکانس کلاک (clock rate) و پریود کلاک معکوس هم هستند فرمول بالایی را می‌توان به صورت زیر نیز نمایش داد:

$$\text{زمان اجرای cpu} = \frac{\text{تعداد کلاک‌های لازم برای اجرای برنامه}}{\text{فرکانس کلاک}}$$

این فرمول به وضوح نشان می‌دهد که طراح سخت‌افزار می‌تواند با کاهش طول پریود کلاک یا تعداد کلاک‌های لازم برای اجرای یک برنامه، سرعت اجرای برنامه را افزایش دهد (سرعت را بهبود دهد). ما در ادامه این فصل و در فصول ۵ و۶ و۷ خواهیم دید که طراح نمی‌تواند همزمان هم طول پریود کلاک را کاهش دهد و هم تعداد کلاکهای لازم برای برنامه را کاهش دهد و اغلب باید یک بالانس بین آنها برقرار کند (trade off). اکثر تکنیکهایی که تعداد کلاکها را کاهش می‌دهند معمولاً طول پریود کلاک را افزایش می‌دهند.

مثال: برنامه دلخواه ما بر روی کامپیوتر A که فرکانس کلاک آن ۴۰۰ مگاهرتز است در ۱۰ ثانیه اجرا می‌شود. ما می‌خواهیم به یک طراح کامپیوتر کمک کنیم که یک ماشین به نام B بسازد که برنامه ما را در ۶ ثانیه انجام دهد. طراح کامپیوتر به ما گفته است که افزایش فرکانس کلاک امکان‌پذیر است ولی این افزایش فرکانس طراحی بخش‌های دیگر cpu را تحت تأثیر قرار خواهد داد و باعث خواهد شد که ماشین B برای اجرای این برنامه تعداد کلاکهایی در حدود ۱/۲ برابر ماشین A نیاز داشته باشد. ما چه فرکانس کلاکی را برای رسیدن به این هدف از طراح بخواهیم؟

جواب: در ابتدا تعداد کلاکهای لازم برای اجرای برنامه بر روی ماشین A را حساب می‌کنیم:

$$\text{زمان اجرای برنامه بر روی A} = \frac{\text{تعداد کلاک‌های لازم برای اجرای برنامه بر روی A}}{\text{فرکانس کلاک ماشین A}}$$

$$10\ s = \frac{\text{تعداد کلاک‌های لازم برای اجرای برنامه بر روی A}}{400 \times 10^6}$$

$$\text{تعداد کلاک‌های لازم برای اجرای برنامه بر روی A} = 4000 \times 10^6$$

زمان اجرای برنامه توسط ماشین B به صورت زیر است:

$$\text{زمان اجرای برنامه بر روی B} = \frac{\text{تعداد کلاک‌های لازم برای اجرای برنامه بر روی B}}{\text{فرکانس کلاک ماشین B}}$$

$$6\ s = \frac{1.2 \times (\text{تعداد کلاک‌های لازم برای اجرای برنامه بر روی A})}{} = \frac{1.2 \times 4000 \times 10^6}{}$$

فرکانس کلاک ماشینB فرکانس کلاک ماشین B

فرکانس کلاک ماشینB = 800 MHz

بنابراین ماشین B برای اینکه برنامه ما را در ۶ ثانیه اجرا کند باید فرکانسش دو برابر فرکانس ماشین A باشد.

تلاقی سخت‌افزار و نرم‌افزار (Hardware Software Interface)

در سراسر این کتاب شما بخشهایی تحت این عنوان خواهید دید. این بخش‌ها تأثیر متقابل بعضی ویژگی‌های نرم‌افزار (نرم‌افزار می‌تواند یک برنامه، یک کامپایلر و یا یک سیستم‌عامل باشد) و بعضی ویژگی‌های سخت‌افزار بر همدیگر را بیان خواهد نمود و آن را به صورت کاملاً واضح و روشنی توضیح خواهد داد. این بخشها باعث خواهند شد که ما به خاطر بسپاریم که طراحی سخت‌افزار و نرم-افزار در بسیاری از موارد تأثیر متقابلی بر هم دارند.

در معادلات موجود در مثالهای قبلی جایی برای تعداد دستورات مورد نیاز یک برنامه در نظر گرفته نشده بود. اما به هر حال چون کامپایلر برای اجرا شدن برنامه دستورهایی به زبان ماشین تولید می‌کند و ماشین نیز برای اجرای برنامه باید آن دستورات را اجرا نماید، بنابراین زمان اجرای یک برنامه باید به تعداد دستورات تولید شده برای یک برنامه وابسته باشد. یکی از راههایی که می‌توان برای محاسبه زمان اجرا در نظر گرفت این است که بگوئیم زمان اجرا مساوی است با تعداد دستورات یک برنامه ضربدر زمان متوسط لازم برای اجرای یک دستور. بنابراین تعداد سیکلهای کلاک مورد نیاز برای یک برنامه می‌تواند به صورت زیر محاسبه شود:

(تعداد کلاک متوسط لازم برای اجرای هر دستور) × (تعداد دستورات برنامه) = تعداد کلاک‌های لازم برای اجرای یک برنامه

عبارت تعداد کلاکهای متوسط مورد نیاز برای هر دستور[10] اغلب CPI نامیده می‌شود. چون دستورات مختلف بسته به اینکه چه کاری انجام می‌دهند، ممکن است زمان اجرای متفاوتی داشته باشند، CPI متوسط زمان اجرای همهٔ دستوراتی است (زمان اجرا به کلاک) که در داخل یک برنامه قرار دارند. CPI روشی برای مقایسه دو پیاده‌سازی مختلف از یک مجموعه دستورات را نیز در اختیار قرار می‌دهد چون در این دو پیاده‌سازی تعداد دستورات مورد نیاز برنامه مطمئناً یکی خواهد بود.

[10] - clock cycles per instruction

مثال: فرض کنید ما دو پیاده‌سازی از یک مجموعه دستورات واحد در اختیار داشته باشیم. ماشین A برای یک برنامه دارای پریود کلاک ۱ نانوثانیه بوده و CPI آن ۲ می‌باشد و ماشین B دارای پریود کلاک ۲ نانوثانیه بوده و CPI آن ۱/۲ می‌باشد. برای این برنامه کدام ماشین سریعتر است و چقدر؟

جواب: می‌دانیم تعداد دستوراتی که هر کدام از ماشین‌ها برای این برنامه اجرا می‌کنند یکی است. ما این تعداد دستورات را I می‌نامیم. در ابتدا تعداد کلاکهای پردازنده را برای هر کدام از ماشین‌ها حساب می‌کنیم:

تعداد کلاک‌های لازم برای اجرای برنامه بر روی ماشین A = $I \times 2.0$

تعداد کلاک‌های لازم برای اجرای برنامه بر روی ماشین B = $I \times 1.2$

حال زمان اجرا برنامه را برای هر کدام از ماشین‌ها حساب می‌کنیم:

پریود کلاک × تعداد کلاک‌های لازم برای اجرای برنامه = زمان اجرای برنامه بر روی ماشین A

$= I \times 2.0 \times 1\,ns = 2 \times I\,ns$

همین طور برای ماشین B داریم:

زمان اجرای برنامه بر روی ماشین B $= I \times 1.2 \times 2\,ns = 2.4 \times I\,ns$

واضح است که ماشین A سریعتر است چون زمان اجرای پایین‌تری دارد.. برای اینکه بدانیم A چقدر سریعتر از B است به صورت زیر عمل می‌کنیم:

$$\frac{\text{کارآیی ماشین A}}{\text{کارآیی ماشین B}} = \frac{\text{زمان اجرای برنامه بر روی ماشین B}}{\text{زمان اجرای برنامه بر روی ماشین A}} = \frac{2.4 \times I\,ns}{2 \times I\,ns} = 1.2$$

بنابراین ماشین A برای این برنامه، 1.2 برابر سریعتر ار ماشین B می‌باشد.

مثال: آیا موارد زیر می‌توانند به تنهایی نشان دهندهٔ کارآیی باشند؟

۱. تعداد کلاکهای لازم برای یک برنامه
۲. تعداد دستورات یک برنامه
۳. پریود کلاک
۴. تعداد کلاک لازم برای اجرای یک دستور
۵. تعداد متوسط دستوراتی که در یک ثانیه اجرا می‌شوند.

جواب: هیچکدام از موارد نمی‌توانند به تنهایی نشان دهنده کارآیی باشند. کارآیی با زمان اجرا ارتباط معکوس دارد و باید زمان اجرا به نوعی در معیار مشخص شده باشد. در مورد اول تعداد کلاک معیار

مناسبی نیست چون مشخص نیست که پریود کلاک چقدر است و بنابراین نمی‌توانیم زمان اجرا را بدست آوریم. در مورد دوم تعداد دستورات نیز معیار مناسبی نیست چون نوع و زمان اجرای دستورات مشخص نیست و نمی‌توانیم زمان اجرای برنامه را بدست آوریم. دستورات مختلف زمان اجرای متفاوتی دارند به عنوان مثال یک دستور ضرب نسبت به یک دستور جمع زمان اجرای بالاتری دارد و دستورات ممیز شناور که با داده‌های ممیزدار کار می‌کنند نسبت به دستوراتی که از داده‌های نوع صحیح استفاده می‌کنند، زمان اجرای بالاتری دارند. در مورد سوم پریود کلاک نیز معیار مناسبی نیست چون مشخص نیست که برنامه به چند کلاک برای اجرا نیاز دارد و نمی‌توانیم زمان اجرای برنامه را مشخص کنیم. در مورد چهارم تعداد کلاک لازم برای اجرای هر دستور نیز معیار مناسبی نیست چون تعداد دستورات و پریود کلاک مشخص نشده و نمی‌توانیم زمان اجرا را حساب کنیم. در مورد پنجم تعداد متوسط دستوراتی که در یک ثانیه اجرا می‌شود نیز به تنهایی معیار مناسبی نیست چون نوع دستورات و تعداد کلاک لازم برای هر دستور و فرکانس کلاک مشخص نیست و نمی‌توانیم زمان اجرا را بدست آوریم.

مثال: اگر دو ماشین ISA یکسان داشته باشد کدام یک از موارد زیر در دو ماشین همیشه برابر می‌باشد؟

۱. فرکانس کلاک
۲. CPI
۳. زمان اجرا
۴. تعداد دستورات
۵. MIPS

جواب: اگر دو ماشین مختلف وجود داشته باشند که ISA یکسانی را پیاده سازی کرده باشند یعنی مجموعه دستوراتی که پشتیبانی می‌کنند مثل هم باشد، در این صورت حتماً تعداد دستورات یک برنامه که برای این دو ماشین کامپایل بشود در هر دو کامپایل یکی خواهد بود.

مثال: یک طراح کامپایلر می‌خواهد بین دو قطعه کد برای یک ماشین مشخص یکی را انتخاب نماید. بر اساس پیاده سازی سخت افزار، سه کلاس مختلف از دستورات وجود دارد: کلاس A، کلاس B و کلاس C که به ترتیب برای اجرا شدن نیاز به ۱، ۲ و ۳ کلاک دارند. قطعه کد اول دارای ۵ دستور است: ۲ مورد از کلاس A، ۱ مورد از کلاس B و ۲ مورد از کلاس C و قطعه کد دوم دارای ۶ دستور است: ۴ مورد از کلاس A، ۱ مورد از کلاس B و ۱ مورد از کلاس C.

الف) کدام قطعه کد سریعتر اجرا می‌شود و چقدر؟

ب) CPI را برای هر قطعه کد حساب کنید؟

جواب: الف) برای بدست آوردن تعداد کلاک از فرمول زیر استفاده می‌کنیم:

$$\text{تعداد کلاکهای لازم} = \sum_{i=1}^{n}\left(CPI_i \times C_i\right)$$

که در آن CPI_i تعداد کلاکهای لازم برای اجرای دستورات از نوع کلاس i می‌باشد و C_i تعداد دستورات کلاس i می‌باشد.

کلاک ۱۰ = 3 × 2 + 2 × 1 + 1 × 2 = تعداد کلاکهای لازم برای قطعه کد ۱

کلاک ۹ = 3 × 1 + 2 × 1 + 1 × 4 = تعداد کلاکهای لازم برای قطعه کد ۲

چون هر دو قطعه کد بر روی یک ماشین اجرا می‌شوند و در نتیجه پریود کلاک برای هر دو یکی است بنابراین قطعه کدی سریعتر است که تعداد کلاک کمتری لازم داشته باشد. بنابراین در این مثال با اینکه قطعه کد ۲ تعداد دستورات بیشتری دارد ولی سریعتر اجرا می‌شود.

ب) برای بدست آوردن CPI از فرمول زیر استفاده می‌کنیم:

$$CPI = \frac{\text{تعداد کلاکها}}{\text{تعداد دستورات}}$$

$$\text{قطعه کد اول}\quad CPI = \frac{۱۰}{۵} = ۲$$

$$\text{قطعه کد دوم}\quad CPI = \frac{۹}{۶} = ۱/۵$$

مثال: فرض کنید یک ماشین داشته باشیم که با سرعت MHZ ۵۰۰ کار کند و فرض کنید که این ماشین دارای سه کلاس دستور از نوع A و B و C باشد که به ترتیب برای اجرا شدن به یک، دو و سه کلاک نیاز داشته باشند. می‌خواهیم دو کامپایلر را بر روی این دو ماشین امتحان کنیم. این کامپایلرها برای تولید کد از یک برنامه بزرگ مورد استفاده قرار می‌گیرند.

کامپایلر اول به تعداد ۵ میلیارد دستور از نوع A، یک میلیارد دستور از نوع B و یک میلیارد دستور از نوع C استفاده می‌کند. و کامپایلر دوم به تعداد ۱۰ میلیارد دستور از نوع A، یک میلیارد دستور از نوع B و یک میلیارد دستور از نوع C استفاده می‌کند.

الف) کدام قطعه کد از لحاظ زمان اجرا سریعتر است؟

ب) کدام قطعه کد از لحاظ تعداد MIPS سریعتر است؟

جواب: MIPS به عنوان یک معیار برای مقایسه کارآیی پردازنده‌ها مورد استفاده قرار می‌گیرد. MIPS که مخفف عبارت Millions of Instruction Per second می‌باشد برای یک کامپیوتر به این صورت تعریف می‌شود: تعداد میلیون دستوری که یک کامپیوتر در یک ثانیه انجام می‌دهد. این مثال نشان می‌دهد که MIPS معیار مناسبی برای کارآیی نیست چون در بعضی مواقع نمی‌تواند زمان اجرا را به درستی نشان بدهد.

الف) برای زمان اجرا از فرمول زیر استفاده می‌کنیم:

زمان اجرا برای قطعه کد اول = $\left(\sum_{i=1}^{n} CPI_i \times C_i\right)$ × پریود کلاک

= (۵ × ۱ + ۱ × ۲ + ۱ × ۳) × ۱۰^{۹} × پریود کلاک

= ۱۰ × ۱۰ ۶ × پریود کلاک = (۱۰ × ۱۰^{۹}) / (۵۰۰ × ۱۰^{۶}) = ۲۰ s

زمان اجرای قطعه کد دوم = $\left(\sum_{i=1}^{n} CPI_i \times C_i\right)$ × پریود کلاک

= (۱۰×۱ + ۱×۲ + ۱×۳) × ۱۰^{۹} = ۱۵ × ۱۰^{۹} × پریود کلاک = (۱۵ × ۱۰^{۹}) / (۵۰۰ × ۱۰^{۶}) = ۳۰ s

پس از نظر زمان اجرا قطعه کد اول سریعتر است.

ب) برای MIPS از فرمول زیر استفاده می‌کنیم:

MIPS = تعداد دستورات / ...

10^6 × زمان اجرا

$$\text{MIPS قطعه کد اول} = \frac{(۵ + ۱ + ۱) \times ۱۰^۹}{۲۰ \times ۱۰^۶} = ۳۵۰$$

$$\text{MIPS قطعه کد دوم} = \frac{(۱۰ + ۱ + ۱) \times ۱۰^۹}{۳۰ \times ۱۰^۶} = ۴۰۰$$

بنابراین از نظر MIPS قطعه کد دوم سریعتر است.

مثال: تعداد دستورات برنامه زیر مشخص کنید.

```
      Li   $a0, 1000          //  a0 = 1000
Loop: sub  $a0, $a0,1         //  a0 = a0 - 1
      bne $a0, $0, Loop  //  if a0 != 0  go to loop
```

جواب: تعداد دستورات یک برنامه را می‌توان به دو صورت شمارش نمود: استاتیکی و دینامیکی. تعداد دستورات استاتیکی در حقیقت همان تعداد خطوط یا تعداد دستورات ظاهری برنامه است ولی تعداد دستورات دینامیکی تعداد دستوراتی است که در عمل اجرا می‌شوند. بنابراین برای این مثال داریم:

۳ = تعداد دستورات استاتیک

چون تعداد دستورات ظاهری برنامه ۳ می‌باشد.

۲۰۰۱ = تعداد دستورات دینامیک

توضیحات هر دستور در کنار همان دستور بعد از علامت // آورده شده است. اگر این برنامه را trace کنیم می‌بینیم که حلقه برنامه به تعداد ۱۰۰۰ مرتبه اجرا می‌شود. چون تعداد دستورات داخل حلقه ۲ است پس تعداد کل دسوراتی که در این حلقه اجرا خواهد شد ۲۰۰۰ مورد می‌شود. توجه داریم که یک دستور نیز در خارج حلقه قرار دارد. پس در کل تعداد دستورات دینامیکی برنامه که در عمل اجرا می‌شوند، ۲۰۰۱ مورد خواهد شد.

مثال: دو پروسسور زیر را که دارای ISA یکسان یا سازگار (ISA compatible) می‌باشند از نظر کارآیی باهم مقایسه کنید؟

۱. یک پردازنده ۸۰۰ مگاهرتز AMD Duron که CPI آن ۱/۲ است
۲. یک پردازنده ۱ گیگا هرتز پنتیوم III (P3) که CPI آن ۱/۵ است

جواب: چون دو پردازنده ISA یکسانی دارند پس برای یک برنامه فرضی تعداد دستورات برای هر جفت آنها یکی است و داریم:

پریود کلاک × CPI × تعداد دستورات = زمان اجرای برنامه روی AMD

= I × ۱/۲ × ۱/۲۵ ns = ۱/۵ ns × I

زمان اجرای برنامه روی P3 = I × ۱/۵ × ۱ ns = ۱/۵ ns × I

چون زمان اجرای برنامه فرضی بر روی دو پردازنده یکی است پس کارآیی آنها یکسان است!

مثال: دو کامپیوتر زیر را که کلاک مختلف ولی ISA یکسان دارند با هم مقایسه کنید.

۱. P4 2.5GHz
۲. P4 3GHz

جواب: چون دو کامپیوتر ISA یکسانی دارند پس تعداد دستورات یک برنامه فرضی بر روی آنها برابر است و همچنین چون دو کامپیوتر از یک نسل می‌باشند (هر دو از نسل P4 هستند) از نظر ساختاری مانند هم بوده و CPI آنها نیز یکی است و داریم:

پریود کلاک × CPI × تعداد دستورات = زمان اجرا

زمان اجرای ۱ = تعداد دستور × CPI × ۰/۴ ns

زمان اجرای ۲ = تعداد دستور × CPI × ۰/۳۳ ns

$$\frac{\text{کارآیی ۲}}{\text{کارآیی ۱}} = \frac{\text{زمان اجرای ۱}}{\text{زمان اجرای ۲}} = \frac{۰/۴}{۰/۳۳} = ۱/۲$$

پس پردازنده P4 3GHz به اندازه ۱/۲ برابر سریعتر از پردازنده P4 2.5GHz است.

مثال: مقایسه دو کامپایلر برای یک ماشین

فرض کنید دو کامپایلر برای یک ماشین داشته باشیم: کامپایلر معمولی و کامپایلر بهینه. کامپایلری که بهتر عمل می‌کند کدهایی تولید می‌کند که تعداد دستورات آن ۵ ٪ (پنج درصد) کمتر است و همچنین CPI آنها ۱۰٪ پایین‌تر است. این دو کامپایلر را مقایسه کنید.

جواب: فرض کنید کامپایلر معمولی را کامپایلر پایه در نظر گرفته و تعداد دستورات تولید شده توسط آن را برای یک برنامه فرضی با I ، CPI کد تولید شده توسط آن را با CPI_B و پریود کلاک ماشین را با T نشان دهیم در این صورت داریم:

پریود کلاک × CPI_B × تعداد دستورات پایه = زمان اجرای کد تولید شده توسط کامپایلر پایه

$= I \times CPI_B \times T$

$T \times CPI \times$ تعداد دستورات = زمان اجرای کد تولید شده با کامپایلر بهینه

$= (0.95 \times I) \times (0.9 \times CPI_B) \times T$

$= 0.86 \times (I \times CPI_B \times T)$

= ۰/۸۶ × زمان اجرای کامپایلر پایه

۱/۱۷ = ۱/۰۰ / ۰/۸۶ = زمان اجرای پایه / زمان اجرای بهینه = کارآیی کامپایلر بهینه / کارآیی کامپایلر پایه

توجه: با توجه به مثالهای متنوعی که حل کردیم می‌توان نتیجه گرفت که روشهای متعددی برای افزایش سرعت پردازنده‌ها وجود دارد. بعضی از این روشها مربوط به تکنیک‌های کامپایلری و نرم‌افزاری است که می‌توان توسط این تکنیکها کدهای با تعداد دستورات کم و با CPI پایین تولید نمود (CPI می‌تواند بر اساس نوع و تعداد دستورات از کدی به کد دیگر تغییر کند در واقع ترکیب دستورات[11] CPI را عوض می‌کند). بعضی دیگر از این روشها سخت افزاری هستند و می‌توان سخت‌افزارهایی طراحی نمود که پریود کلاک پایینی داشته باشند و تعداد کلاک لازم برای اجرای هر دستور نیز پایین باشد (در واقع با طراحی سخت افزارهای مناسب می‌توان CPI را کم کرد).

قانون امدال

قانون امدال در معماری کامپیوتر قانونی است که می‌توان به کمک آن ارزش کارهایی که قرار است انجام شوند را از قبل پیش بینی نمود. این کارها عمدتاً مربوط به بهبودهایی است که در سخت‌افزار پردازنده‌ها داده می‌شود. این قانون به شرح زیر است:

[11] - Instruction mix

قانون امدال: مقدار بهبودهایی که انجام می‌شود به میزان مؤثر بودن آنها محدود می‌شود.

برای توضیح این قانون سیستم واحدی دانشگاه را در نظر می‌گیریم. در این سیستم بعضی دروس یک واحدی، بعضی دو واحدی، بعضی سه واحدی و بعضی نیز چهار واحدی‌اند. همان‌طور که می‌دانیم در محاسبه معدل ترم، دروسی بیشتر تأثیر دارند که تعداد واحد آنها بالاتر است. بنابراین در این سیستم ارزش دروس چهار واحدی بیشتر از سه واحدی، ارزش دروس سه واحدی بیشتر از دو واحدی، ارزش دروس دو واحدی نیز بیشتر از یک واحدی است. بنابراین ما بهتر است بیشترین وقت و سرمایه‌گذاری را بر روی دروسی انجام دهیم که بیشترین تأثیر را در معدل دارند. در معماری کامپیوتر قانون امدال می‌گوید: قبل از اینکه بهبودی را انجام دهی یا کاری را انجام دهی باید به میزان مؤثر بودنش نیز توجه کنی و آن کار را وقتی انجام دهی که تأثیرش در مقابل کاری که انجام می‌دهی قابل توجه باشد. در قالب فرمول قانون امدال به صورت زیر بیان می‌شود:

$$\text{زمان اجرای برنامه بعد از بهبود} = \frac{\text{زمان اجرایی که تحت تأثیر قرار می‌گیرد}}{\text{میزان بهبود}} + \text{زمان اجرایی که تحت تأثیر قرار نمی‌گیرد}$$

مثال: فرض کنید که تصمیم گرفته باشیم سخت افزار پردازنده را تغییر دهیم تا سرعت اجرای دستورات floating point دو برابر شود. دو برابر کردن سرعت یک ایده خوب است ولی باید ببینیم در قبال کاری که انجام می‌دهیم چقدر بهبود در کل زمان اجرای یک برنامه به وجود می‌آید. به عبارتی باید ببینیم دستورات floating point چه میزان در برنامه ما نقش دارند. فرض کنید برای یک برنامه فرضی فقط ۱۰ درصد زمان اجرای برنامه (T) شامل دستورات floating point باشد. برای این برنامه تأثیر بهبود سخت‌افزار را بررسی کنید.

جواب: با استفاده از قانون امدال می‌توان نوشت:

$$\text{زمان اجرای برنامه بعد از بهبود} = \frac{۰/۱۰\ T}{۲} + ۰/۹۰\ T = ۰/۹۵\ T$$

یعنی زمان اجرای کل برنامه فقط ۵ درصد بهبود خواهد داشت. این میزان بهبود خیلی بالا نیست بنابراین بر طبق قانون امدال دو برابر کردن سرعت اجرای دستورات floating point به امید بهبود زمان اجرای برنامه در این مثال ارزش فنی ندارد.

نتیجه قانون امدال: قسمتهایی از برنامه را بهبود دهیم که بیشتر استفاده می‌شوند چون بیشترین تأثیر را در کل زمان اجرای برنامه دارند. قانون امدال بیان می‌کند که موارد پر استفاده را بهبود دهیم و یا به عبارتی می‌گوید: Make the common case fast.

۲-۴- انتخاب برنامه‌ها برای اندازه گیری کارآیی

کاربر کامپیوتری که هر روز مجموعه برنامه‌های ثابتی را اجرا می‌کند، یک کاندیدای خوب برای ارزیابی سرعت یک کامپیوتر جدید است. مجموعه برنامه‌هایی که توسط این کاربر اجرا می‌شوند، یک workload نامیده می‌شوند. برای مقایسه کارآیی دو سیستم کامپیوتری این کاربر به سادگی زمان اجرای یک workload را بر روی دو ماشین مقایسه می‌کند. اما مشکلی که در اینجا وجود دارد این است که اکثر کاربران در این شرایط قرار ندارند و باید روشهای دیگری را برای ارزیابی کارآیی استفاده کنند و امیدوار باشند که این روشها بتوانند نشان دهنده کارآیی workload های واقعی باشند. این روشها معمولاً مبتنی بر استفاده از مجموعه‌ای از benchmark ها می‌باشند. Benchmark ها مجموعه‌ای از برنامه‌ها هستند که که برای اندازه‌گیری کارآیی انتخاب می‌شوند. Benchmark ها یک workload را تشکیل می‌دهند که کاربر امیدوار است که نشان دهنده کارآیی یک workload واقعی باشد.

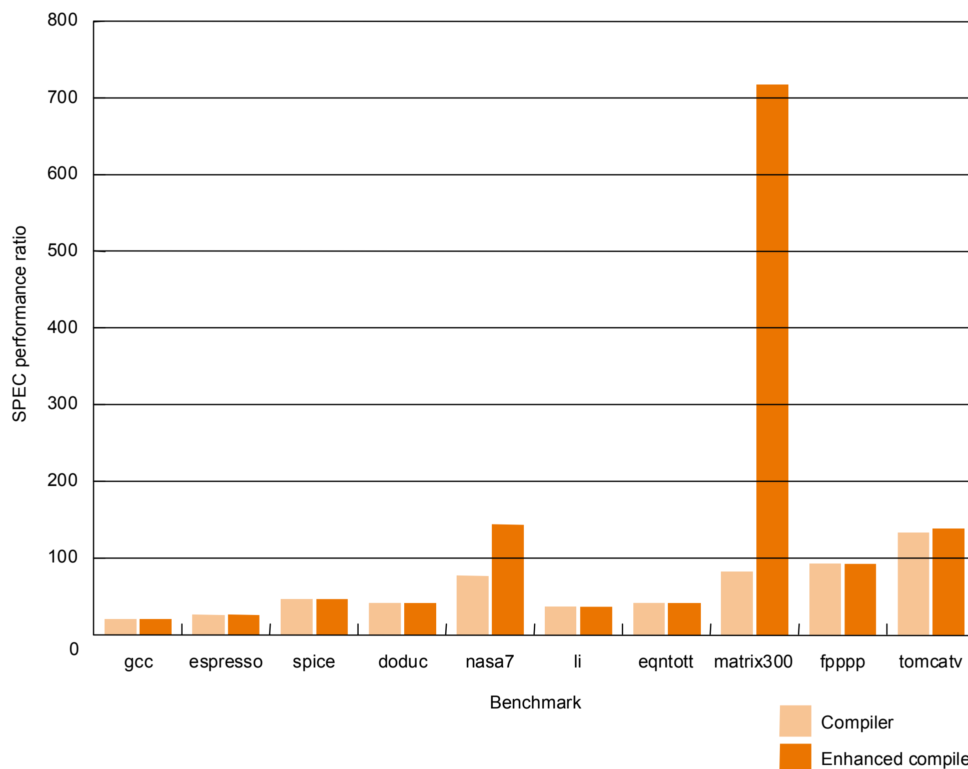
یک benchmark خوب باید بتواند کارآیی برنامه‌های واقعی را منعکس نماید و علی‌الخصوص benchmark باید دارای ترکیب واقعی از دستورات باشد (یعنی شامل ترکیبی از دستورات باشد که در برنامه‌های واقعی ظاهر می‌شوند). امروزه ثابت شده است که بهترین benchmark ها برنامه‌های واقعی هستند. تعدادی از benchmark های معمول عبارتند از:

- Adobe Photoshop for image processing
- BABCO SYSmark for office applications
- Unreal Tournament 2003 for 3D games
- SPEC (System Performance Evaluation Cooperative)

امروزه برخی از سازنده‌های کامپیوتر بر اساس اطلاعاتی که از کاربران و benchmark های آنان دارند، روشهای نادرستی را در پیش می‌گیرند و اطلاعات غلطی را به کابران می‌دهند. این سازندگان یا از تکنیکهای کامپایلری استفاده می‌کنند که برای یک benchmark مشخص کدهای بهینه‌ای تولید می‌کنند که بر روی سخت افزار سریعتر اجرا می‌شوند و یا اینکه از تکنیکهای سخت‌افزاری استفاده می‌کنند که یک ترکیب مشخص از دستورات را که مربوط به یک benchmark مشخص است را با سرعت بالاتری اجرا می‌کنند. استفاده از برنامه‌های واقعی به عنوان benchmark استفاده از روشهای فوق برای افزایش

سرعت اجرا شدن benchmark ها را تقریباً غیر ممکن می‌سازد حتی اگر بهبودی نیز حاصل شود این بهبود چون مربوط به برنامه‌های واقعی است، قابل استفاده خواهد بود.
استفاده کردن از benchmark هایی که اندازه آنها کوچک است و یا کارآیی آنها به قسمت کوچکی از کد برنامه بستگی دارد خطر بهینه سازی‌های کامپایلری یا سخت افزاری را به همراه خواهد داشت. به طور مثال مجموعه benchmark های SPEC در ابتدا برای این منظور انتخاب شدند که از برنامه‌های واقعی برای اندازه‌گیری کارآیی استفاده کنند. متأسفانه اولین نسخه از مجموعه SPEC که در سال ۱۹۸۹ وارد بازار شد، دارای یک benchmark به نام matrix300 بود که فقط دارای یک سری عملیات ضرب ماتریسی بود. در واقع ۹۹ درصد از زمان اجرا مربوط به یک خط از این برنامه بود. این واقعیت که زمان خیلی زیادی از اجرای برنامه مربوط به یک خط از برنامه بود که در آن یک خط، یک عملیات مشخص انجام می‌گرفت، باعث شد تا چند کمپانی کامپیوتری کامپایلرهایی را تولید و یا خریداری کنند تا زمان اجرای این benchmark را بهینه سازی کنند. شکل ۲-۳ نسبت کارآیی[12] (معکوس زمان اجرا) را برای یک ماشین با دو کامپایلر مختلف نشان می‌دهد. برنامه‌های این شکل مجموعه benchmark های SPEC89 می‌باشند. در این شکل برای هر برنامه دو میله وجود دارد که میله سمت راستی مربوط به کامپایلر بهینه‌سازی شده برای matrix300 می‌باشد. همان طور که مشاهده می‌شود کامپایلر بهینه-سازی شده برای matrix300 اثری بر روی ۸ مورد از ۱۰ مورد از برنامه‌های مجموعه benchmark های SPEC89 ندارد، اما کارآیی matrix300 را بیش از ۹ برابر بهبود داده است. کاربری که براساس matrix300 قضاوت می‌کند حسابی دچار اشتباه بزرگی خواهد شد به دلیل اینکه بهبود داده شده فقط مربوط به matrix300 بوده و برای برنامه‌های دیگر کاربردی نخواهد داشت.

[12] - Performance ratio



شکل ۲–۳: نسبت کارآیی مجموعه benchmark های SPEC89 برای یک ماشین با دو کامپایلر مختلف

ضعف مربوط به matrix300 باعث شد که این benchmark در نسخه بعدی مجموعه SPEC یعنی SPEC92 کنار گذاشته شود.

حال سؤال این است که چرا همه مردم از برنامه‌های واقعی برای اندازه‌گیری کارآیی استفاده نمی‌کنند؟ یکی از دلایل این است که benchmark های کوچک به علت کوچک بودن به راحتی کامپایل و شبیه-سازی می‌شوند (حتی بعضی مواقع دستی کامپایل می‌شوند) و به همین دلیل در شروع طراحی جذابیت‌های خاصی دارند. مخصوصاً چون در شروع طراحی یک ماشین هنوز کامپایلری برای آن نوشته نشده است، این benchmark ها جایگاه خاصی دارند. دلیل دیگری که برای استفاده از benchmark های کوچک وجود دارد این است که آنها راحت‌تر از برنامه‌های بزرگ استاندارد سازی می‌شوند و بنابراین نتایج منتشر شده زیادی برای benchmark های کوچک وجود.

اگرچه استفاده از benchmark های کوچک در مراحل اولیه طراحی قابل توجیه است، اما هیچ دلیل قانع کننده‌ای وجود ندارد که از آنها برای ارزیابی کارآیی سیستم‌های کامپیوتری که وارد بازار شده‌اند استفاده کنیم. در گذشته یکی از دلایلی که از benchmark های کوچک به جای برنامه‌های بزرگ استفاده می‌شد این بود که آنها بر روی ماشین‌های مختلف به راحتی پورت می‌شدند و از طریق آنها می‌شد سیستم‌های مختلف را مقایسه نمود. اما امروزه این دلیل خیلی درست نیست و استفاده از

benchmark های کوچک فقط در مراحل اولیه طراحی مجاز است و بعد از آن باید از برنامه‌های واقعی برای ارزیابی کارآیی ماشین‌ها استفاده شود.

پس از آنکه ما مجموعه مناسبی از benchmark ها را انتخاب نموده و توسط آنها کارآیی را اندازه‌گیری کردیم، می‌توانیم یک گزارش برای کارآیی تهیه کنیم. توضیحاتی که در گزارش آورده می‌شود باید طوری باشد که اندازه‌گیری ما قابل تکرار باشد. گزارش ما باید شامل یک لیست از همه چیزهایی باشد که ممکن است یک فرد دیگر برای تکرار آزمایشهای ما به آنها نیاز داشته باشد. این لیست باید شامل نسخه سیستم عامل، کامپایلرها، ورودی‌ها و همچنین نحوه پیکربندی ماشین باشد. به طور مثال توصیف سیستمی[13] ماشینی که برای بدست آوردن نتایج شکل ۲-۳ از آن استفاده شده است در شکل ۲-۴ نشان داده شده است

Hardware	
Model number	Powerstation 550
CPU	41.67 MHz POWER 4164
FPU	Integrated
Number of CPUs	1
Cache size per CPU	64K data / 8K instruction
Memory	64 MB
Disk subsystem	2 400-MB SCSI
Network interface	NA
Software	
OS type and rev	AIX v3.1.5
Compiler rev	AIX XL C/6000 Ver. 1.1.5 AIX XL Fortran Ver. 2.2
Other software	None
File system type	AIX
Firmware level	NA
System	
Tuning parameters	None
Background load	None
System state	Multiuser (single-user-login)

شکل ۲-۴: توصیف سیستمی ماشین استفاده شده برای اندازه‌گیری‌های شکل ۲-۳

[13] - System description

Benchmark های Synthetic

یک synthetic benchmark، یک برنامه است که فقط برای منظور اندازه‌گیری کارآیی نوشته شده است و معمولاً اندازه آن کوچک بوده و به راحتی بر روی CPU های مختلف پورت می‌شود. بنابراین برای مقایسه سیستم‌ها راحت‌تر می‌باشند. هدف از طراحی synthetic benchmark ها این بوده که یک برنامه واحد داشته باشیم که ویژگیهای تعداد زیادی از برنامه‌ها را داشته باشد و تعداد تکرار دستورات در داخل برنامه متناسب با تعداد تکرار دستورات در تعداد زیادی از benchmark ها باشد. معروفترین benchmark های synthetic عبارتند از Whetstone و Dhrystone .

چون این نوع benchmark ها واقعی نیستند نشان دهنده هیچ چیز خاصی نیستند. به طور مثال هیچ خروجی در اختیار کاربر قرار نمی‌دهند که کاربر بداند برنامه به درستی اجرا شده است و نیز اگر بر روی یک ماشین کارآیی آنها بهبود پیدا کرد هیچ دلیلی وجود ندارد که برای برنامه‌های واقعی نیز چنین باشد و کارآیی آن ماشین برای سایر برنامه‌ها نیز بهبود یابد. اندازه این benchmark ها کوچکتر است و بنابراین به راحتی تکنیکهای بهینه سازی کامپایلر و سخت‌افزار برای آنها قابل اعمال است.

نکته: بنا به دلایل فوق synthetic benchmark ها برای اندازه گیری کارآیی انتخاب مناسبی نیستند.

۲-۵- مقایسه کردن و خلاصه سازی کارآیی

حال که یاد گرفتیم برنامه‌هایی را به عنوان benchmark انتخاب نمائیم و برای مقایسه از معیارهای زمان اجرا و throughput استفاده کنیم، ممکن است تصور کنید که مقایسه کردن کارآیی کار سر راستی باشد. اما معمولاً به جای یک benchmark از چند benchmark استفاده می‌شود و ما باید یاد بگیریم که چگونه کارآیی یک گروه از benchmark ها را اندازه‌گیری کنیم. اگر چه خلاصه کردن یک مجموعه از اندازه‌گیری‌ها اطلاعات کمی در اختیار قرار می‌دهد، ولی بازار و حتی کاربران اغلب ترجیح می‌دهند که یک عدد ساده از کارآیی داشته باشند و با استفاده از آن مقایسه‌های خود را انجام دهند. حال سؤال کلیدی این است که این عدد چگونه محاسبه می‌شود؟ شکل ۲-۵ زمان اجرای دو برنامه را بر روی دو ماشین مختلف نشان می‌دهد.

	Computer A	Computer B
Program 1 (seconds)	1	10
Program 2 (seconds)	1000	100
Total time (seconds)	1001	110

شکل ۲-۵: زمان اجرای دو برنامه بر روی دو ماشین مختلف

طبق تعریفی که ما از سرعت داشتیم، عبارتهای زیر در مورد شکل ۲-۵ صادق هستند:

- ماشین A برای برنامه ۱، ۱۰ مرتبه سریعتر از ماشین B عمل می‌کند.
- ماشین B برای برنامه ۲، ۱۰ مرتبه سریعتر از ماشین A عمل می‌کند.

این دو جمله خیلی گیج کننده هستند. در نهایت کدام ماشین سریعتر است؟

ساده‌ترین راه برای خلاصه‌سازی کارآیی در این موارد که بیش از چند برنامه داریم این است که مجموع زمانهای اجرا را نسبت به هم مقایسه کنیم. بنابراین:

$$\frac{\text{کارآیی ماشین B}}{\text{کارآیی ماشین A}} = \frac{\text{مجموع زمان اجرای برنامه‌ها بر روی ماشین A}}{\text{مجموع زمان اجرای برنامه‌ها بر روی ماشین B}} = \frac{\text{۱۰۰۱}}{\text{۱۱۰}} = \text{۹/۱}$$

یعنی اینکه ماشین B برای مجموعه برنامه‌های ۱ و۲ ، ۹/۱ مرتبه سریعتر از ماشین A است.

این نوع خلاصه سازی مستقیماً متناسب با زمان اجرا می‌باشد. اگر workload ما به صورتی باشد که برنامه‌های ۱ و ۲ را به یک اندازه اجرا کنیم، رابطه فوق می‌تواند برای مقایسه ماشین‌های مختلف استفاده شود.

بعضی مواقع به جای رابطه فوق میانگین زمان اجرا که آن هم متناسب با زمان اجراست را به کار می‌برند. این میانگین، میانگین حسابی[14] یا AM نامیده می‌شود و از فرمول زیر بدست می‌آید:

$$AM = \frac{1}{n}\sum time_i$$

که در آن $time_i$ زمان اجرای برنامه i ام موجود در workload متشکل از n برنامه است. هر چقدر AM کوچکتر باشد، کارآیی ماشین بالاتر خواهد بود.

AM زمانی قابل استفاده است که برنامه‌های موجود در workload به یک اندازه اجرا شوند، اگر غیر از این باشد می‌توانیم برای هر برنامه یک وزنی اختصاص دهیم که نشان دهنده تعداد تکرار برنامه در آن workload باشد. به طور مثال اگر ۲۰ درصد کارها در workload برنامه ۱ و ۸۰ درصد کارها در workload برنامه ۲ باشد در این صورت وزنها به صورت ۰/۲ و ۰/۸ خواهند بود. با استفاده از جمع کردن حاصل‌ضرب فاکتورهای وزن در زمان اجرا فرمول میانگین حسابی وزن‌دار[15] یا WAM بدست می‌آید:

$$WAM = \sum_{i=1}^{n} weight_i \times time_i$$

---

[14] - Arithmetic Mean
[15] - Weighted Arithmetic Mean

فرمول میانگین حسابی حالت خاصی از فرمول میانگین حسابی وزن‌دار می‌باشد که در آن وزن همه برنامه‌ها مساوی است.

۲-۶- موضوع واقعی: benchmark های SPEC95 و کارآیی پردازنده‌ها

SPEC معروفترین مجموعه benchmark های CPU می‌باشد. SPEC که مخفف عبارت System Performance Evaluation Cooperative (مجموعه ارزیابی کننده کارآیی سیستم) می‌باشد، در سال ۱۹۸۹ توسط تعدادی از کمپانی‌های کامپیوتر برای این منظور ایجاد شد که اندازه‌گیری و خلاصه سازی کارآیی از طریق یک پروسه اندازه‌گیری کنترل شده، بهبود داده شود و همچنین benchmark های واقعی‌تری برای اندازه‌گیری کارآیی مورد استفاده قرار گیرند. تاکنون چهار نسخه از SPEC وارد بازار شده است: SPEC89، SPEC92، SPEC95 و SPEC2000. مجموعه SPEC95 نسخه سال ۹۵ SPEC می‌باشد که شامل ۸ برنامه اعداد صحیح و ۱۰ برنامه ممیز شناور می‌باشد. مجموعه برنامه‌های SPEC95 در شکل ۲-۶ نشان داده شده است.

Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
ijpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Naiver Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

شکل ۲-۶: مجموعه benchmark های SPEC95

معمولاً دو گزارش جداگانه برای مجموعه برنامه‌های صحیح و برنامه‌های ممیز شناور داده می‌شود. روند اندازه‌گیری با استفاده از برنامه‌های مجموعه SPEC به این صورت است که در ابتدا عددهای اندازه‌گیری شده برای زمان اجرا با تقسیم کردن زمان اجرای برنامه بر روی ماشین SUN sparc 10/40 به زمان اجرای بدست آمده بر روی ماشینی که می‌خواهیم کارآیی آن را اندازه بگیریم، به صورت نرمال

شده درآورده می‌شود. سپس این عدد نرمال شده به یک معیار برای اندازه‌گیری کارآیی منجر می‌شود که نسبت SPEC ratio نامیده می‌شود. هر چقدر عدد بدست آمده برای SPEC ratio بزرگتر باشد نشان دهنده کارآیی بالاتر می‌باشد (در واقع SPEC ratio با زمان اجرا نسبت معکوس دارد). خلاصه-سازی‌های SPECint95 و SPECfp95 با استفاده از میانگین هندسی SPEC ratio ها بدست می‌آید که اولی مربوط به برنامه‌های صحیح و دومی مربوط به برنامه‌های ممیز شناور مجموعه SPEC می‌باشد. فرمول میانگین هندسی[16] به صورت زیر است:

$$\sqrt[n]{\prod_{i=1}^{n}(execution\ \ time\ \ ratio)_i}$$

که در آن $(execution\ \ time\ \ ratio)_i$ زمان اجرای نرمال شده بر اساس یک ماشین مرجع برای برنامه i– ام مجموعه benchmark ها می‌باشد.

همان طور که قبلاً نیز توضیح داده شده است برای یک ISA مفروض، افزایش کارآیی CPU می‌تواند به یکی از سه روش زیر صورت بگیرد:

۱– افزایش فرکانس کلاک

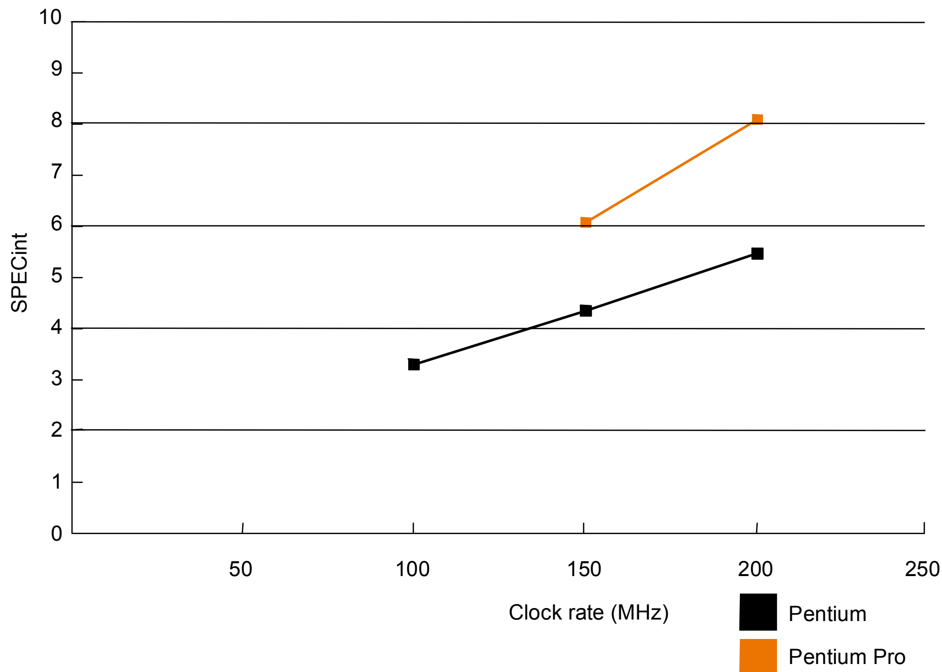
۲– بهبود ساختار پردازنده که باعث کم شدن CPI شود

۳– بهبودهای کامپایلری که باعث کم شدن تعداد دستورات یا تولید دستوراتی با میانگین CPI پایین‌تر می‌شود

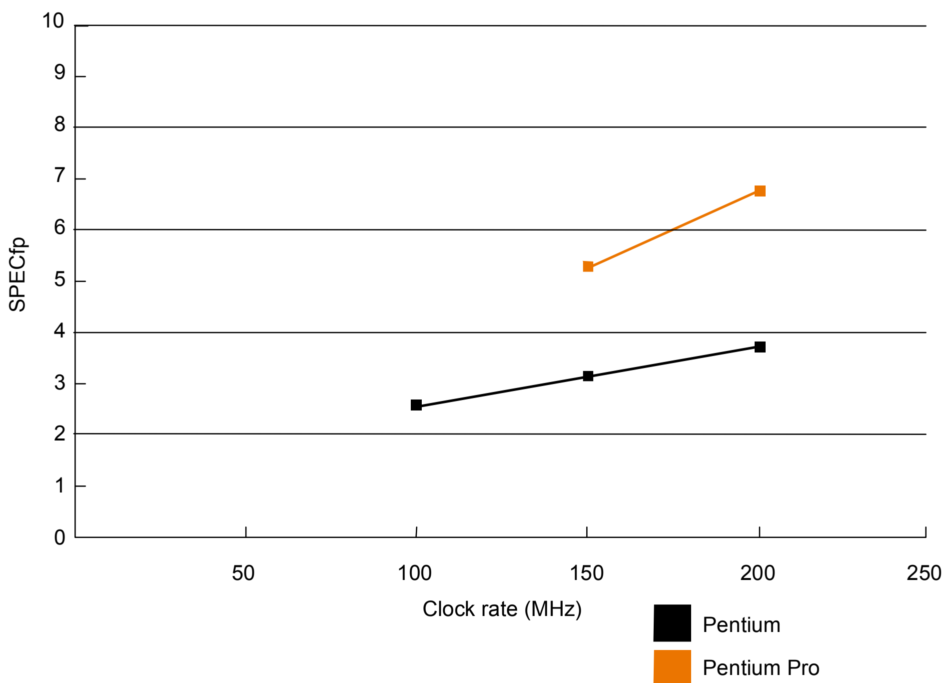
برای نشان دادن بهبودهای فوق بر روی کارآیی، شکل‌های ۲–۷ و ۲–۸ به ترتیب اندازه‌گیری‌های SPECint95 و SPECfp95 را برای پردازنده‌های Pentium و Pentium pro نشان می‌دهند. چون SPEC نیاز به این دارد که حتماً روی سخت‌افزار واقعی اجرا شود و سیستم حافظه تأثیر زیادی بر روی کارآیی دارد، سیستم‌های دیگری که بر اساس این پردازنده‌ها ساخته می‌شوند ممکن است مقداری با این شکلها متفاوت باشند چون سیستم حافظه آنها ممکن است قدری متفاوت باشد. ماشین‌های اینتل که در اینجا اندازه‌گیری شده‌اند سیستم‌های حافظه و کامپایلر قوی‌تری استفاده کرده‌اند و این بدان معناست که اکثر ماشین‌های دیگری که با استفاده از این پردازنده ساخته می-شوند کارآیی پایین‌تری برای benchmarks های SPEC خواهند داشت.

---

[16] - Geometric Mean



شکل۲–۷: SPECint محاسبه شده برای پردازنده‌های Pentium و Pentium pro در فرکانس‌های مختلف



شکل۲–۸: SPECfp محاسبه شده برای پردازنده‌های Pentium و Pentium pro در فرکانس‌های مختلف

چند برداشت مهم از شکلهای ۲–۷ و ۲–۸ قابل مشاهده است. مهمترین آنها بهبود کارآیی پردازنده Pentium pro نسبت به Pentium می‌باشد. در یک فرکانس فرضی SPECint نشان می‌دهد که

Pentium pro، ۱/۴ تا ۱/۵ برابر سریعتر از Pentium می‌باشد و SPECfp95 نشان می‌دهد که Pentium pro ، ۱/۷ تا ۱/۸ برابر سریعتر می‌باشد. اگرچه بهبودهای کامپایلری مشخص برای هر پردازنده‌ای وجود دارد، اکثر بهبودهای کارآیی برای Pentium pro از طریق بهبودهای ساختاری (سخت‌افزاری) ایجاد شده است.

برداشت مهم دیگری که از شکلها به دست می‌آید این است که وقتی که فرکانس کلاک با استفاده از ضریب مشخص اضافه می‌شود، کارآیی پردازنده با ضریب کمتری اضافه می‌شود. به طور مثال وقتی که فرکانس کلاک Pentium از ۱۰۰ مگاهرتز به ۲۰۰ مگاهرتز افزایش پیدا می‌کند (دو برابر می‌شود)، کارآیی SPECint 95 فقط ۱/۷ و کارآیی SPECfp95 فقط ۱/۴ برابر بهبود می‌یابد. دلیل این امر به سیستم حافظه پردازنده بر می‌گردد. در کل، چون سرعت حافظهٔ اصلی تغییر چندانی نمی‌کند، اضافه شدن سرعت پردازنده در گلوگاه سیستم حافظه گرفتار می‌شود. این تأثیر در benchmark های ممیز شناور بیشتر مشهود است چون آنها نسبت به برنامه‌های integer بزرگتر هستند. این رفتار مثال خوبی برای قانون امدال می‌باشد که بیانگر میزان بهبود بر اساس میزان مؤثر بودن آن می‌باشد.

در مقام مقایسه، میزان افزایش کارآیی پردازنده Pentium pro، با اضافه شدن فرکانس کلاک بهتر ازPentium می‌باشد. هر چند که این میزان افزایش برابر ضریب افزایش کلاک نیست. به طور مثال، بهبود کارآییSPECfp95 از فرکانس ۱۵۰ به ۲۰۰ مگاهرتز در Pentium pro، ۱/۲۴ می‌باشد در حالی که پردازنده Pentium بهبودی در حدود ۱/۱۸ را برای این بازه نشان می‌دهد. در فصل ۷ در مورد تأثیر سیستم حافظه در کارآیی پردازنده صحبت خواهد شد و اینکه چرا پردازنده Pentium pro سیستم حافظه بهتری دارد.

۲–۸ نکات پایانی

ما در این فصل بر روی performance یا همان کارآیی سیستم‌ها و نحوه اندازه‌گیری آن تمرکز کردیم. تمرکز کردن تنها بر روی کارآیی بدون در نظر گرفتن هزینه[17] خیلی منطقی نیست و ما باید بین این دو پارامتر تعادل برقرار کنیم.

[17] - Cost

ما در این فصل دیدیم که با استفاده از زمان اجرای برنامه‌های واقعی بر روی پردازنده‌ها، یک روش قابل اطمینان برای اندازه‌گیری و گزارش کردن کارآیی وجود دارد. این زمان اجرا وابسته به تعدادی فاکتور مهم می‌باشد که این وابستگی در فرمول زیر مشاهده می‌شود:

پریود کلاک × CPI × تعداد دستورات = زمان اجرا

ما به وفور از این فرمول و فاکتورهای آن بهره خواهیم گرفت. همان طور که توضیح داده شد هیچ کدام از این فاکتورها به تنهایی نمی‌توانند نشان دهنده کارآیی باشند و فقط حاصل‌ضرب آنها که مساوی زمان اجراست قابل استفاده است. مطمئناً دانستن این فرمول به تنهایی برای طراحی کردن یا ارزیابی کردن یک کامپیوتر نمی‌تواند کافی باشد. ما باید بفهمیم که چگونه جنبه‌های مختلف طراحی، این فاکتورهای مهم را تحت تأثیر قرار می‌دهند. این نوع دانش شامل موارد متعددی می-شود از جمله: تأثیر معماری مجموعه دستورات بر تعداد دستورات دینامیکی، تأثیر pipelining و سیستم حافظه بر روی CPI، و تأثیر تکنولوژی و سازمان یک کامپیوتر بر روی فرکانس کلاک. هنر طراحی کامپیوتر فقط اعمال کردن عدد در فرمول کارآیی نیست بلکه مشخص کردن این موضوع هست که طراحی‌های مختلف چگونه کارآیی و هزینه را تحت تأثیر قرار می‌دهند.

اکثر کاربران کامپیوتر به هر دو مورد هزینه و کارآیی دقت می‌کنند. فهمیدن ارتباط بین جنبه‌های مختلف طراحی و کارآیی آن کار مشکلی است و مشخص کردن هزینه قابلیتهای مختلف یک طرح نیز کار به مراتب مشکل‌تری است. هزینه یک ماشین فقط به هزینه قسمتهای تشکیل دهنده آن محدود نیست، بلکه هزینه‌هایی همچون هزینه‌های نیروی انسانی برای جمع‌آوری سیستم[۱۸] ، هزینه تحقیقات، هزینه توسعه سیستم، تبلیغات بازار و ... را نیز باید به آن اضافه نمود. امروزه به دلیل پیشرفت سریعی که در تکنولوژی‌های پیاده سازی ایجاد می‌شود، اهمیت دادن بیش از حد به مورد هزینه در طول مدت‌هایی حدود ۶ ماه یا یک سال خیلی منطقی به نظر نمی‌رسد.

و نکته آخر اینکه معماری‌های مختلف کامپیوتر توسط کارآیی و هزینه سنجیده می‌شوند و پیدا کردن تعادل بین این دو همیشه به عنوان یک هنر در طراحی کامپیوتر مطرح بوده است.

مطالب مهم این فصل:

- Performance یا همان کارآیی یک معیار مهم برای مقایسه سیستم‌هاست.
- دو روش معتبر برای اندازه‌گیری کارآیی وجود دارد: محاسبه زمان اجرا و محاسبه throughput
- فرمول اصلی کارآیی: پریود کلاک × CPI × تعداد دستورات = زمان اجرا

---

[18] - Assemble

- بهترین راه برای اندازه‌گیری کارآیی این است که از برنامه‌های واقعی استفاده کنیم.
- Benchmark ها برنامه‌هایی هستند که برای اندازه‌گیری کارآیی انتخاب می‌شوند و باید به اندازه کافی بزرگ باشند که اجرای آنها توسط کامپایلر و یا سخت افزار بهینه سازی نگردد و تا حد ممکن از برنامه‌های واقعی انتخاب شوند. مهمترین مجموعه benchmark های شناخته شده مجموعه SPEC می‌باشد.
- قانون امدال بیان می‌کند که ما از یک بهبود داده شده در طرح چقدر می‌توانیم انتظار بهبود کارآیی را داشته باشیم. میزان بهبود کارآیی به میزان مؤثر بودن تغییر داده شده بستگی دارد.
- اگر به جای یک برنامه چند برنامه برای ارزیابی کارآیی استفاده شود و بخواهیم کارآیی را خلاصه سازی کنیم از فرمول‌های میانگین حسابی و یا میانگین حسابی وزن‌دار استفاده می-کنیم. میانگین هندسی نیز قابل استفاده است ولی میانگین هندسی دارای ضعف‌هایی است که نمی‌توان آن را به عنوان یک معیار خوب قبول کرد.
- گزارش تهیه شده برای کارآیی باید طوری باشد که توسط دیگران قابل تکرار باشد. بنابراین باید همه چیزهایی که برای تکرار یک آزمایش لازم است در گزارش آورده شود.

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

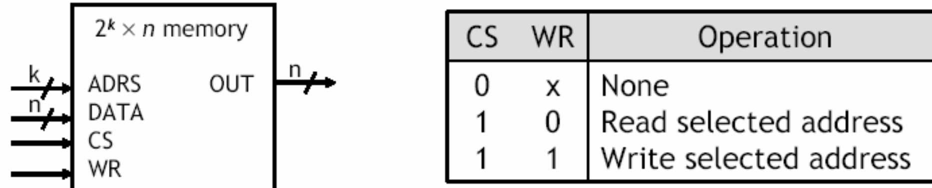
رجیسترها دارای سرعت زیادی هستند و کارکردن با آنها راحت‌تر است ولی ما فقط ۳۲ عدد رجیستر در اختیار داریم و هر کدام از رجیسترها فقط می‌توانند ۳۲ بیت داده در داخل خود نگهداری کنند. اگر ما ساختمان داده‌هایی نظیر آرایه‌ها و ساختمان‌ها[1] داشته باشیم نمی‌توانیم آنها را داخل رجیسترها ذخیره‌سازی کنیم چون تعداد 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 [۱] : 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[۲] 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[۳]

[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

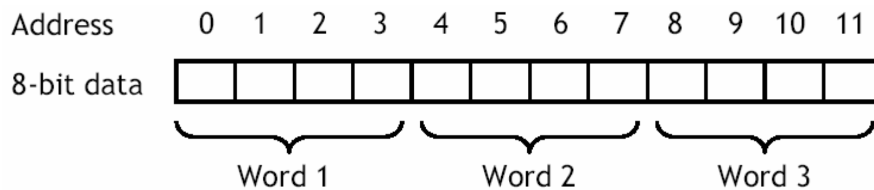
دسترسی به داده‌های ۳۲ بیتی حافظه در پردازنده MIPS

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 می-شوند، 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 ۲^5 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 بازهٔ $+2^{15}$ 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. اگر $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 می-شود 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: $9\times16=144$.

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 می-شوند، 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*[۱] 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*[۲] 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;
}
```

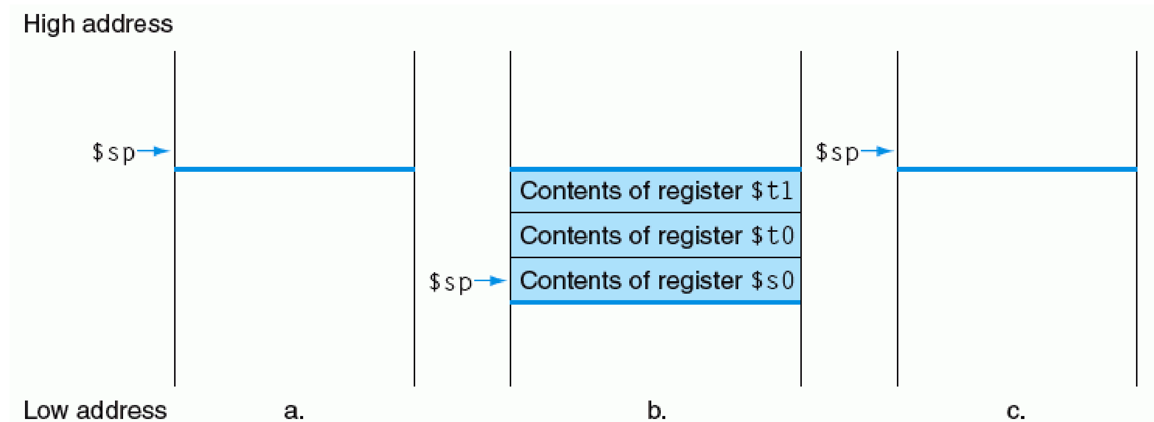
پاسخ: کد اسمبلی به صورت زیر است:

---

[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 قرارداد 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	

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
	پرش رجیستری	jr $ra	goto L	برای بازگشت از تابع
	پرش و پیوند	jal L	$ra = PC + 4; goto L	برای فراخوانی تابع

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 [۱] ، 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[۲] 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 می-روند 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، می-توانیم 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 $1 \neq 0$. و در صورتی $at=1 می‌شود که $a0 < $a1 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 می-کند 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

فصل ۴

طراحی واحد محاسبه و منطق (ALU)

ALU یکی از مهمترین قسمتهای یک کامپیوتر و واحدی است که همهٔ عملیات محاسباتی مانند جمع، تفریق و... را انجام می‌دهد. در این بخش به کمک گیت‌های پایه AND، OR، NOT و مالتی پلکسر یک واحد ALU را طراحی خواهیم نمود. از آنجا که پردازنده ما ۳۲ بیتی است، ما به یک ALU ۳۲ بیتی نیاز خواهیم داشت. در این قسمت ابتدا یک ALU یک بیتی را طراحی نموده و سپس ۳۲ عدد از این ALU های یک بیتی را به هم وصل نموده و یک ALU ۳۲ بیتی طراحی خواهیم نمود. بنابراین ابتدا طراحی یک ALU یک بیتی را توضیح می‌دهیم.

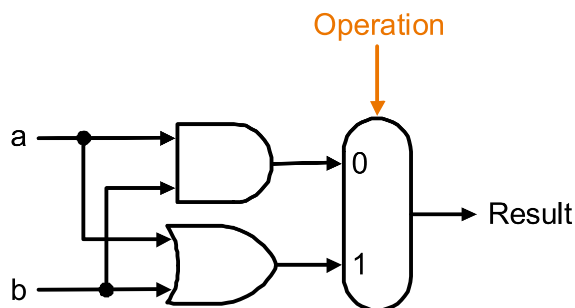
لازم به ذکر است که سیستم‌های مختلفی برای نمایش اعداد به صورت باینری وجود دارد که از جمله معروفترین آنها می‌توان به سیستم اندازه علامت[1]، سیستم متمم ۱[2] و سیستم متمم ۲[3] اشاره نمود. محاسبات در سیستم متمم ۲ ساده‌تر از دو سیستم دیگر بوده و طراحی سخت افزار آن نیز ساده‌تر است به همین دلیل در این فصل به صورت پیش‌فرض، توضیحاتی که ارائه می‌شود برای سیستم متمم ۲ است مگر اینکه صریحاً سیستم دیگری را ذکر کنیم.

۴-۱- ALU یک بیتی

شکل ۱ یک واحد منطقی که دو عملیات AND و OR را پیاده‌سازی می‌کند را نشان می‌دهد. مالتی پلکسر استفاده شده، بسته به مقدار ورودی Operation یکی از عملیات a AND b یا a OR b را انتخاب می‌کند. در واقع خط ورودی Operation، مالتی‌پلکسر را کنترل نموده و یکی از ورودی‌های خط داده[4] مالتی‌پلکسر را انتخاب می‌نماید. توجه کنید که ما از قصد اسامی Operation و result را به ترتیب برای خط انتخاب و خروجی مالتی‌پلکسر به کار بردیم چون operation نوع عملیات ALU را تعیین می‌کند و نتیجه عملیات ALU نیز بر روی خروجی result ظاهر می‌شود.

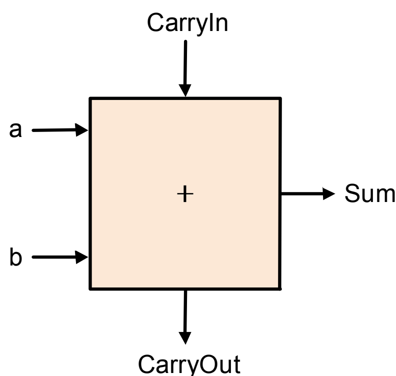
---

1 - Sign magnitude
2 - On's complement
3 - Two's complement
4 - Data



شکل ۱: یک ALU ساده با دو عملیات AND و OR

عملیات بعدی که می‌خواهیم به ALU خود اضافه کنیم عملیات جمع کردن[1] است. همان‌طور که می‌دانیم یک جمع‌کننده کامل[2] دارای ۳ ورودی و ۲ خروجی می‌باشد. سه ورودی آن a، b و CarryIn و دو خروجی آن Sum، CarryOut می‌باشد. بلوک دیاگرام یک جمع کننده یک بیتی در شکل ۲ نشان داده شده است.



شکل ۲: بلاک دیاگرام یک جمع کننده یک بیتی

اگر جدول درستی یک جمع‌کننده کامل را رسم کنیم و مقدار خروجی‌ها را بر اساس ورودی‌ها مشخص کنیم و سپس با استفاده از جدول کارنو معادله خروجی‌ها را به دست آوریم، چنین بدست می‌آید:

$$CarryOut = a.b + a.CarryIn + b.CarryIn$$

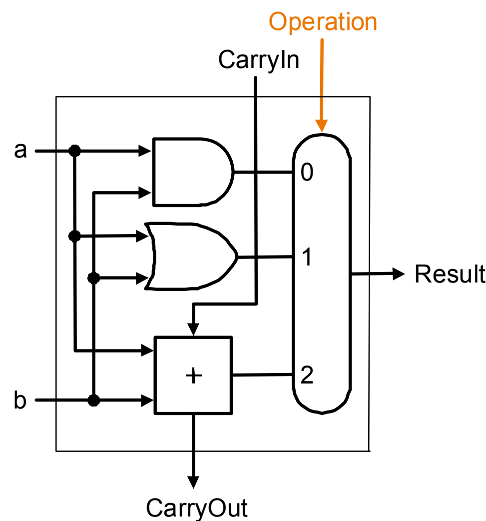
$$sum = a.\bar{b}.\overline{CarryIn} + \bar{a}.b.\overline{CarryIn} + \bar{a}.\bar{b}.CarryIn + a.b.CarryIn$$

---

[1] - Add
[2] - Full Adder

حال مدار جمع کننده را به ALU یک بیتی خود اضافه می‌کنیم، شکل ۳ چگونگی انجام این کار را نشان داده است. در این شکل ALU قادر است که سه عملیات مختلف را بر طبق اینکه ورودی Operation چه مقداری داشته باشد انجام دهد:

- اگر Operation=0 باشد عملیات AND انجام می‌شود.
- اگر Operation=1 باشد عملیات OR انجام می‌شود.
- اگر Operation=2 باشد عملیات Add انجام می‌شود.



شکل ۳: یک ALU ساده با ۳ عملیات AND، OR، و جمع کردن

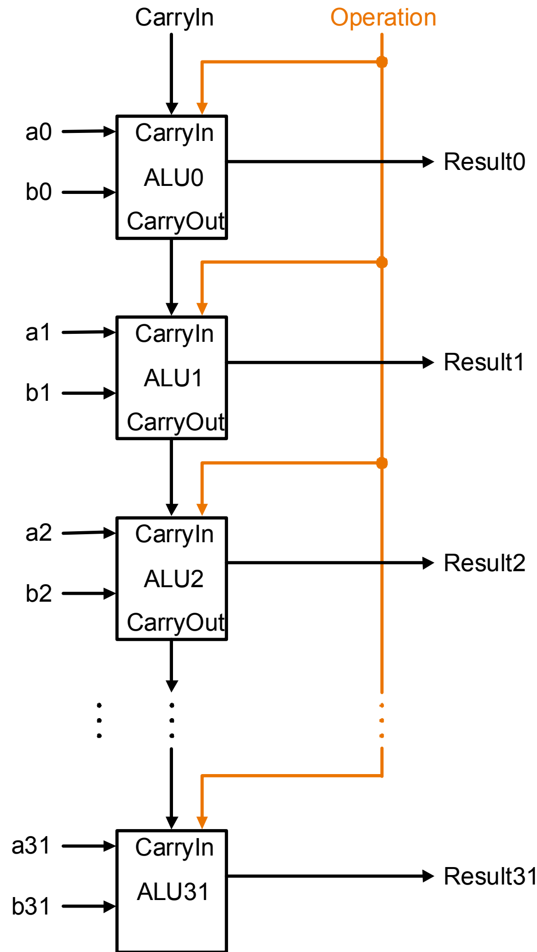
بعضی مواقع طراحان می‌خواهند که ALU عملیات بیشتری را انجام دهد مثلاً خروجی 0 نیز تولید کند. ساده‌ترین راه برای اضافه‌کردن یک عملیات به ALU این است که تعداد ورودی‌های مالتی‌پلکسر را زیاد کنیم و عملیات جدید را به خطوط ورودی جدید وصل کنیم مثلاً برای تولید صفر توسط ALU کافی است که به یکی از ورودی‌های ALU مقدار 0 را وصل کنیم.

۴-۲- طراحی یک ALU ۳۲ بیتی

حال که ما ALU یک بیتی را طراحی کردیم، می‌توانیم آن را به صورت یک جعبه سیاه[1] در طراحی یک ALU ۳۲ بیتی استفاده کنیم. ALU ۳۲ بیتی از اتصال ۳۲ عدد ALU یک بیتی به صورت

[1] - Black Box

شکل ۴ ایجاد می‌شود در این شکل منظور از ai یعنی بیت i ام عدد ۳۲ بیتی a مثلاً $a2$ یعنی بیت دوم عدد ۳۲ بیتی a.



شکل ۴: طرح یک ALU ۳۲ بیتی ساده که از اتصال ۳۲ ALU یک بیتی ساده تشکیل شده است

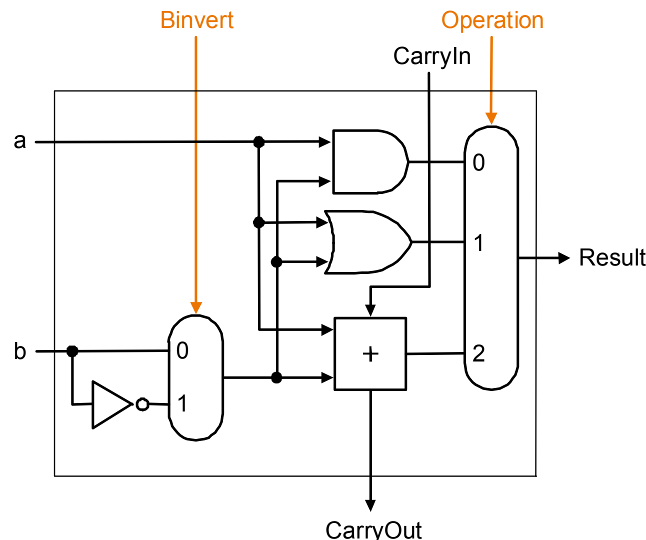
همان طور که در این شکل دیده می‌شود خروجی نقلی حاصل از طبقهٔ اول (CarryOut) به ورودی نقلی طبقهٔ دوم (CarryIn) و خروجی نقلی طبقهٔ دوم به ورودی نقلی طبقهٔ سوم وصل شده است و به همین ترتیب این کار تا طبقهٔ آخر یعنی طبقهٔ ۳۱ انجام می‌شود و سرانجام از خروجی نقلی آن خارج شود. این کار همانند ایجاد یک موج توسط سنگی است که در آب یک برکه ساکت و آرام انداخته می‌شود چون موج ایجاد شده توسط این سنگ از محل سنگ منتشر می‌شود تا اینکه به کنار برکه برسد. در مدار ALU ۳۲ بیتی نیز رقم نقلی از طبقه اول تا طبقه آخر به صورت موج منتشر می‌شود

و در نهایت از آن خارج می‌شود. به همین دلیل به جمع‌کننده‌ای که از اتصال مستقیم رقم‌های نقلی جمع کننده‌های یک بیتی ایجاد می‌شود جمع کننده موج گونه[1] گفته می‌شود.

تفریق کردن همانند جمع کردن یک عدد با مقدار منفی یک عدد دیگر در سیستم مکمل۲ [2] می‌باشد. همان‌طور که می‌دانیم برای بدست آوردن مقدار منفی یک عدد در این سیستم کافی است که همهٔ بیتها را معکوس نموده (این عملیات مکمل ۱ کردن نام دارد) و سپس مقدار یک را به آن اضافه کنیم. بنابراین تفریق دو عدد به صورت زیر بدست می‌آید:

$$a - b = a + (-b) = a + (\overline{b} + 1)$$

بنابراین برای انجام دادن عملیات تفریق لازم است که بتوانیم تک تک بیت‌های عدد b را معکوس کنیم. در نتیجه هر کدام از ALU های یک بیتی استفاده شده باید غیر از عملیات AND، OR، و جمع کردن، باید بتوانند بیت b را نیز معکوس کنند. ALU یک بیتی جدید که عملیات معکوس کردن را نیز انجام می‌دهد در شکل ۵ نشان داده شده است. همان طور که در این شکل دیده می‌شود برای معکوس کردن هر بیت ما نیاز به یک مالتی پلکسر ۲ به ۱ داریم که بین b و $\overline{b}$ یکی را انتخاب کند (عمل انتخاب توسط بیت Binvert انجام می‌شود).



شکل ۵: یک ALU یک بیتی ساده که عملیات معکوس کردن را نیز پشتیبانی می‌کند

[1] - Ripple carry adder
[2] - Two's complement system

حال بیائید ۳۲ عدد از این ALU های جدید یک بیتی را به هم متصل کرده و یک ALU ۳۲ بیتی بسازیم. مالتی‌پلکسر ۲ به ۱ اضافه شده بسته به مقدار ورودی binvert مقدار b و یا معکوس شده آن را در اختیار قرار می‌دهد. حال که ما در این ۳۲ طبقه معکوس بیت‌های b را در اختیار داریم برای انجام عملیات تفریق a-b کافی است که a را با معکوس عدد b جمع کرده و یک ۱ به حاصل جمع اضافه کنیم. توجه کنید که ورودی نقلی اولین طبق برای عملیات جمع صفر قرار داده می‌شود. اگر ما این ورودی نقلی را ۱ قرار دهیم چه اتفاقی می‌افتد؟ در این صورت اگر مالتی‌پلکسرهای ۲ به ۱، b را انتخاب کنند عملیات a+b+1 و اگر $\bar{b}$ را انتخاب کنند عملیات $a+\bar{b}+1$ انجام خواهد گرفت که همان عملیات تفریق است:

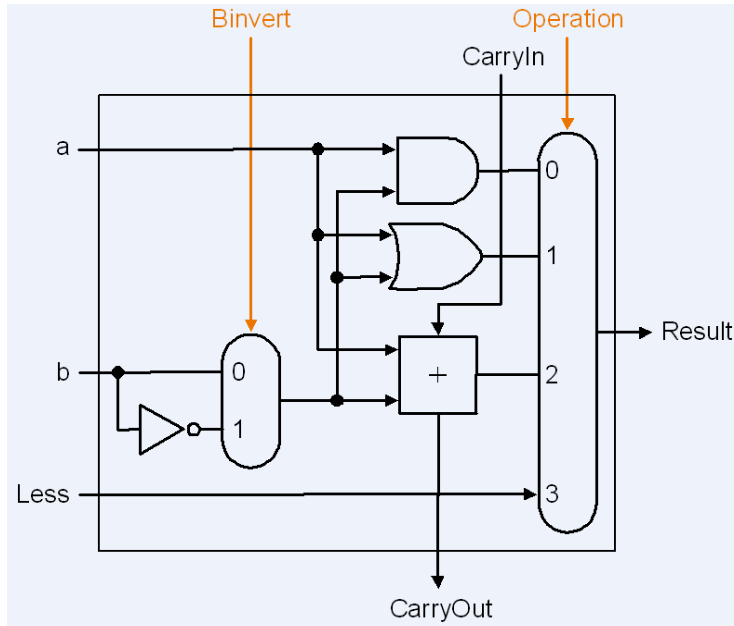
$$a+\bar{b}+1 = a+(\bar{b}+1) = a+(-b) = a-b$$

نکته: به دلیل ساده‌تر بودن طراحی سخت‌افزار برای مدار جمع‌کننده در سیستم متمم ۲، در محاسبات اعداد صحیح کامپیوترها به جای سیستم‌های دیگر از این سیستم استفاده می‌شود.

۴-۳- طراحی ALU ۳۲ بیتی برای پردازنده MIPS

مجموعه عملیات add، Subtract، And و OR تقریباً در هر کامپیوتری وجود دارند. اکثر دستورات پردازنده MIPS توسط ALU طراحی شده در بخش قبلی قابل انجام هستند ولی باید توجه نمود که طراحی ALU برای MIPS هنوز کامل نشده است. یکی از دستورات MIPS که لازم است پشتیبانی شود، دستور slt[1] می‌باشد. به خاطر بیاورید که این دستور اگر $rs < rt$ بود، مقدار یک و در غیر این صورت مقدار صفر را تولید می‌کرد. بنابراین دستور slt همهٔ بیتهای خروجی ALU را غیر از بیت پایین آن صفر خواهد کرد و بیت پایین آن را بر اساس نتیجهٔ مقایسه یک یا صفر خواهد نمود. برای اینکه ALU دستور slt را انجام دهد نیاز به این داریم که در همه ۳۲ طبقه ALU، مالتی‌پلکسرهای ۳ ورودی را گسترش دهیم و یک ورودی برای دستور slt در نظر بگیریم. ما این ورودی جدید را Less نام‌گذاری کرده و آن را فقط برای slt استفاده می‌کنیم. شکل ۶ یک ALU یک بیتی را که مالتی‌پلکسر آن به ۴ ورودی گسترش یافته تا عملیات slt را پشتیبانی کند، نشان می‌دهد.

[1] - Set on less then



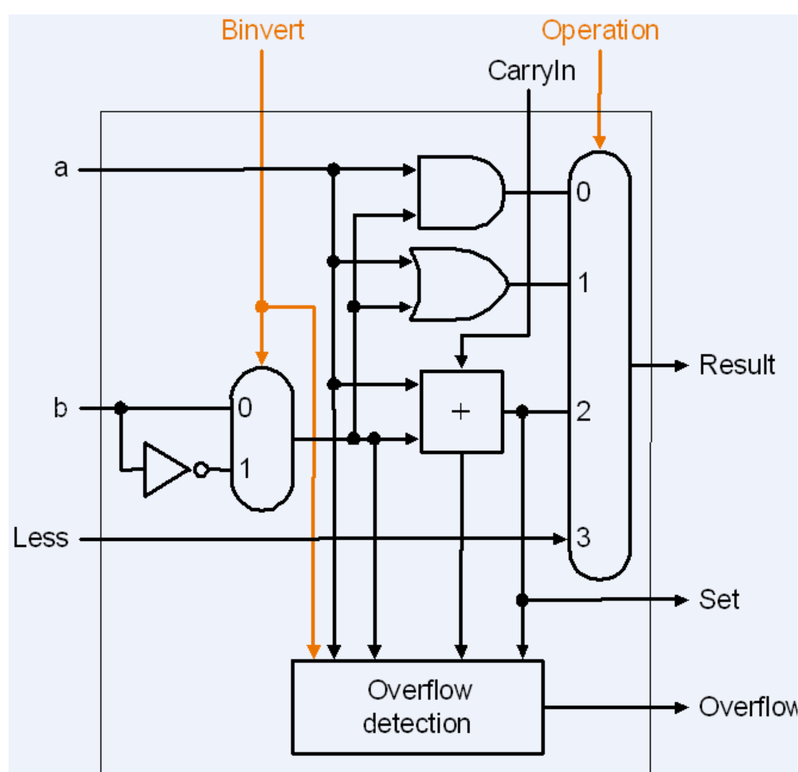
شکل ۶: **ALU** یک **بیتی** که یک عملیات دیگر (بر روی خط Less) به آن اضافه شده است

طبق توضیح بالا ما باید ورودی Less را برای ۳۱ طبقهٔ بالای ALU (۳۱ بیت با ارزش) مساوی صفر قرار دهیم چون برای دستور slt، ۳۱ بیت بالای نتیجه ALU همیشه صفرند. چیزی که باقی می‌ماند این است که ما چگونه عمل مقایسه را انجام دهیم و بیت پایین ALU (بیت شماره صفر) را برای دستور slt به درستی مقداردهی کنیم. بیائید ببینیم که وقتی b را از a کم می‌کنیم چه اتفاقی می‌افتد؟ اگر حاصل تفریق منفی باشد در این صورت $a < b$ می‌باشد چون داریم:

$$a - b < 0 \Rightarrow a < b$$

ما می‌خواهیم که بیت پایین نتیجه ALU (بیت شماره صفر خروجی ALU) در صورتی که a<b باشد به ۱ و در غیر این صورت به صفر مقداردهی شود. یعنی اگر a-b منفی باشد این بیت ۱ و در غیر این صورت صفر شود. این نتیجه دقیقاً مطابق با مقدار بیت علامت نتیجه عملیات تفریق می‌باشد چون همان طور که می‌دانیم بیت علامت یک عدد اگر منفی باشد ۱ و در غیر این صورت صفر است. بنابراین با این توضیح ما فقط کافی است که بیت علامت خروجی جمع‌کننده (که برای slt عملیات تفریق انجام می‌دهد) را به ورودی Less مالتی‌پلکسر طبقهٔ اول وصل کنیم. همان طور که می‌دانیم بیت علامت یک عدد، بیت با ارزشتر یا همان بیت آخر (بیت شماره ۳۱ اگر شماره بیت‌ها را از صفر شروع

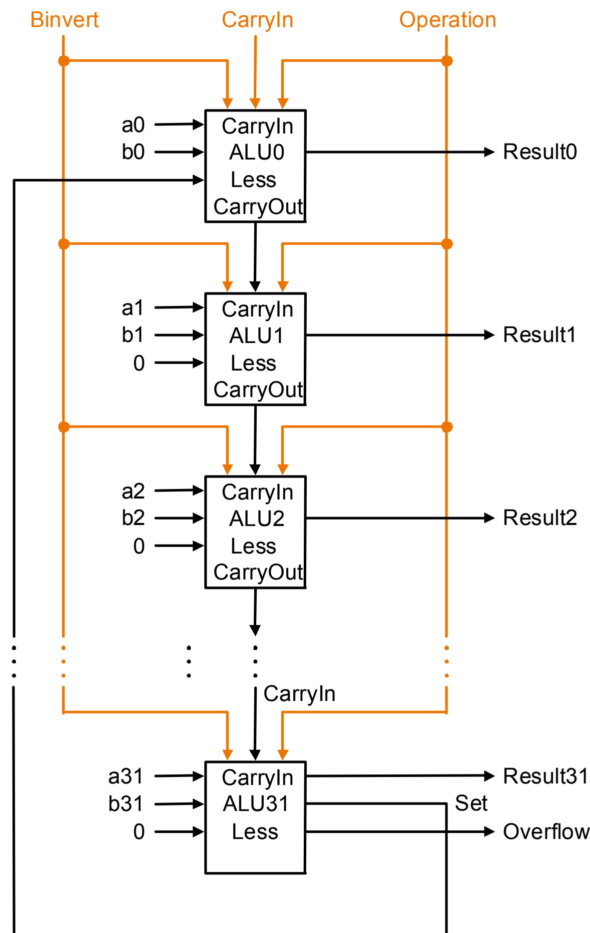
کنیم) آن عدد خواهد بود. بنابراین برای عملیات تفریق، بیت علامت نتیجه عملیات، بیت خروجی حاصل شده از جمع کننده طبقه آخر ALU (خروجی sum جمع کننده طبقه ۳۱) خواهد بود. اما متأسفانه بیت خروجی جمع کننده طبقه آخر به عنوان یک خروجی از ALU ما خارج نشده است. بنابراین ما برای طبقهٔ آخر ALU به یک ALU یک بیتی جدید نیاز خواهیم داشت که یک خروجی اضافه‌تر دارد که مستقیماً به خروجی جمع‌کننده وصل شده است. در شکل ۷ این ALU یک بیتی نشان داده شده است و خروجی اضافه شده در آن Set نام گرفته است که فقط برای دستور slt استفاده می-شود.



شکل ۷: ALU یک بیتی مورد نیاز برای طبقه آخر ALU ۳۲ بیتی که عملیات slt را پشتیبانی می‌کند

همان طور که ما برای طبقهٔ آخر به یک ALU یک بیتی مخصوص نیاز داریم، مدار تشخیص دهنده سرریز نیز باید در این ALU یک بیتی مخصوص قرار داده شود چون امکان بروز سرریز فقط در طبقهٔ آخر چک می‌شود. سرریز برای یک ALU ۳۲ بیتی وقتی اتفاق می‌افتد که نتیجه حاصل از محاسبات، داخل ۳۲ بیت جا نشوند و بیشتر از ۳۲ بیت نیاز داشته باشند. می‌توان نشان داد که برای سیستم متمم۲ سرریز وقتی اتفاق می‌افتد که علامت عدد عوض می‌شود یعنی انتظار داریم که نتیجه

ALU یک عدد مثبت باشد ولی منفی می‌شود و یا اینکه انتظار داریم نتیجه ALU یک عدد منفی باشد ولی مثبت می‌شود. و باز می‌توان نشان داد که در سیستم متمم ۲ سرریز وقتی اتفاق می‌افتد که نقلی‌های خروجی از دو طبقهٔ آخر مقدارهای متفاوتی داشته باشند یعنی $Carryout_{30} \neq Carryout_{31}$ و این وضعیت را می‌توان با یک گیت xor تشخیص داد $V = Carryout_{30} \oplus carryout_{31}$. در این عبارت V وقتی یک می‌شود که $Carryout_{30} \neq Carryout_{31}$ باشد. همان طور که می‌دانیم $Carryout_{30}$ همان ورودی نقلی برای طبقهٔ ۳۱ یا همان طبقهٔ آخر می‌باشد. بنابراین اگر ما بخواهیم یک ALU ۳۲ بیتی بسازیم، در طبقهٔ آخر آن باید از این ALU یک بیتی مخصوص که یک خروجی set و یک خروجی Overflow دارد استفاده کنیم و در بقیه طبقات از ALU های یک بیتی ساده استفاده کنیم. شکل ۸ ALU ۳۲ بیتی طراحی شده برای پردازنده MIPS با امکان Overflow و پشتیبانی از دستور slt را نشان می‌دهد.



شکل ۸:ALU ۳۲ بیتی برای پردازنده MIPS

توجه کنید که ما هر وقت که بخواهیم ALU عملیات تفریق را انجام دهد، ورودی CarryIn طبقهٔ اول و همچنین ورودی Binvert را یک قرار می‌دهیم. برای عملیات جمع و عملیات منطقی ما می‌خواهیم که این دو ورودی مقدار صفر داشته باشند. بنابراین ما می‌توانیم برای کنترل ساده ALU این دو ورودی را در قالب یک ورودی ترکیب نموده و اسم آن را Bnegate قرار دهیم. در شکل ۸، این کار نیز انجام شده است.

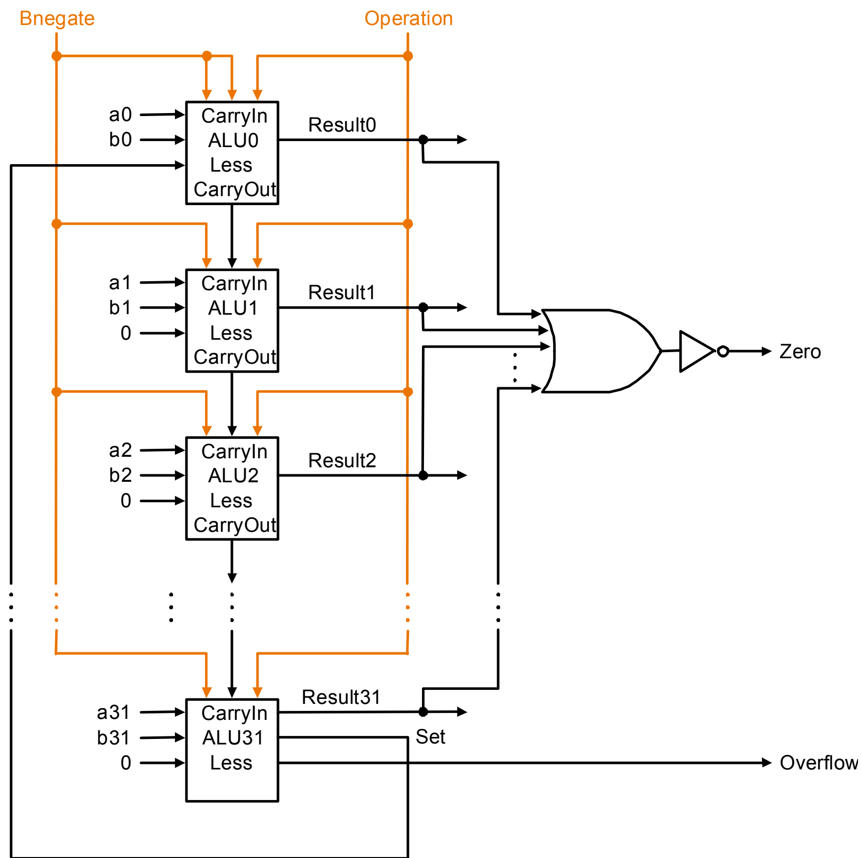
برای اینکه بتوانیم دستورات بیشتری از پردازنده MIPS را با این ALU انجام دهیم ما باید دستورات پرش شرطی را نیز به عملیات ALU اضافه کنیم. همان طور که می‌دانیم دستورات پرش شرطی به شرطی پرش را انجام می‌دهند که محتوای دو رجیستر مساوی باشند یا نباشند (در beq شرط مساوی بودن و در bne شرط نامساوی بودن چک می‌شود). ساده‌ترین روش برای تست مساوی بودن با ALU طراحی شده این است که b را از a کم کنیم و بعد بررسی کنیم که آیا نتیجه عملیات صفر می-شود یا نه، به دلیل اینکه داریم:

$$a = b \Rightarrow a - b = 0$$

بنابراین اگر ما به ALU طراحی شده بخشی را اضافه کنیم که صفر شدن نتیجه را بررسی کند، می‌توانیم دستورهای beq و bne را پشتیبانی نمائیم. ساده‌ترین راه برای بررسی صفر شدن نتیجه این است که همهٔ بیت‌های خروجی را با هم OR نموده و بعد سیگنال بدست آمده را از یک گیت معکوس کننده عبور دهیم:

$$\text{Zero} = (\overline{\text{Result31} + \text{Result30} + \ldots + \text{Result2} + \text{Result1} + \text{Result0}})$$

شکل ۹ ALU ۳۲ بیتی پردازنده MIPS را که قابلیت انجام دستورات beq و bne نیز به آن اضافه شده است را نشان می‌دهد.



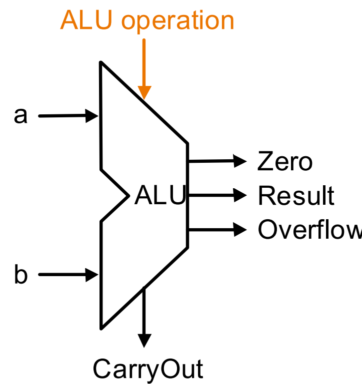
شکل ۹: ALU ۳۲ بیتی پردازنده MIPS که دستورات slt، beq، bne را پیاده سازی می‌کند و امکان تشخیص overflow را نیز دارد

ما می‌توانیم ۲ بیت کنترلی Operation و یک بیت کنترلی Bnegate را با هم ترکیب نموده و فرض کنیم که ALU ما در کل ۳ بیت کنترلی برای عملیات خود داشته باشد. ما با استفاده از این ۳ بیت کنترلی به ALU می‌گوئیم که عملیات add، Subtract، AND، OR و غیره را انجام دهد. جدول ۱ خطوط کنترلی ALU و عملیات انجام گرفته از طریق آنها را نشان می‌دهد.

جدول ۱: عملیاتی که توسط ALU طراحی شده با ۳ بیت کنترلی قابل انجام است

Operation	عملیات
000	And
001	Or
010	Add
110	Sub
111	Slt

در نهایت حال که ما فهمیدیم چه چیزی در داخل ALU ۳۲ بیتی قرار گرفته است، ما می‌توانیم این ALU را به صورت یک سمبل (Symbol) یا Box در آورده و در مراحل بعدی از آن استفاده نمائیم. شکل ۱۰ بلاک دیاگرام یا جعبه سیاه ALU ۳۲ بیتی نهایی طراحی شده برای پردازنده MIPS را نشان می‌دهد. طبق توضیحات قبلی، در این شکل ورودی‌های a و b ۳۲ بیتی، ورودی operation ۳ بیتی، خروجی Result ۳۲ بیتی و بقیه خروجی‌ها یک بیتی خواهند بود. ما در طراحی پردازنده از این بلاک دیاگرام استفاده خواهیم کرد.



شکل ۱۰: بلاک دیاگرام ALU طراحی شده برای پردازنده MIPS

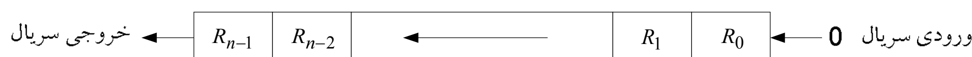
۴-۴- عملیات شیفت

عملیات شیفت[1] یا انتقال برای انتقال یا جابجایی سری داده‌ها به کار گرفته می‌شوند. آنها همچنین به همراه عملیات حسابی و منطقی دیگر مورد استفاده قرار می‌گیرند. محتوای یک ثبات می‌تواند به چپ یا به راست شیفت پیدا کند. در همان زمانی که بیت‌ها شیفت داده می‌شوند، اولین فلیپ فلاپ اطلاعات دودویی خود را از ورودی سری دریافت می‌کند. در حین شیفت به چپ، بیت ورودی سریال به سمت راست‌ترین مکان منتقل می‌شود و در حین عمل شیفت به راست، بیت ورودی سریال به سمت چپ‌ترین مکان منتقل می‌شود. اطلاعاتی که از طریق ورودی سری منتقل می‌گردد تعیین کننده نوع شیفت است. سه نوع شیفت وجود دارد: منطقی[2]، چرخشی[3] و حسابی[4].

[1] - Shift
[2] - Logical shift
[3] - Rotate shift
[4] - Arithmetic shift

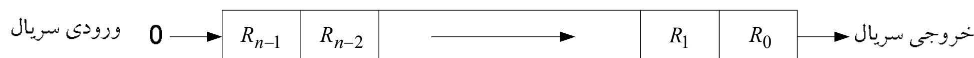
۴-۴-۱- شیفت منطقی

شیفت منطقی مقدار 0 را از طریق ورودی سری انتقال می‌دهد و به دو صورت شیفت منطقی به چپ و شیفت منطقی به راست صورت می‌گیرد. ما سمبل shl و shr را به ترتیب برای عملیات شیفت به چپ و شیفت به راست به کار خواهیم برد. عملیات شیفت منطقی به چپ به صورت شکل ۱۱ انجام می‌گیرد. به عبارتی در این عملیات، بیت R_0 به بیت R_1، بیت R_1 به بیت R_2 و به همین ترتیب هرکدام از بیت‌ها، به مکان بالاتر از خود (سمت چپ) منتقل می‌شوند. در حین این عملیات مقدار 0 وارد R_0 شده و مقدار قبلی R_{n-1} (سمت چپ‌ترین بیت) به بیرون منتقل شده و از بین می‌رود. به طور مثال اگر یک رجیستر ۸ بیتی را که دارای محتوای 00101101 می‌باشد یکبار به سمت چپ شیفت دهیم محتوای آن به صورت 01011010 خواهد شد.



شکل ۱۱: شیفت منطقی به چپ

شیفت منطقی به راست نیز به صورت شکل ۱۲ انجام می‌شود. همان طور که مشاهده می‌شود این عملیات دقیقاً همانند شیفت منطقی به چپ است با این تفاوت که جهت انتقال از چپ به راست بوده و مقدار 0 به سمت راست‌ترین بیت منتقل شده و مقدار قبلی بیت R_0 از بین خواهد رفت. به طور مثال اگر یک رجیستر ۸ بیتی را که دارای محتوای 00101101 می‌باشد یکبار به سمت راست شیفت دهیم محتوای آن به صورت 00010110 خواهد شد.

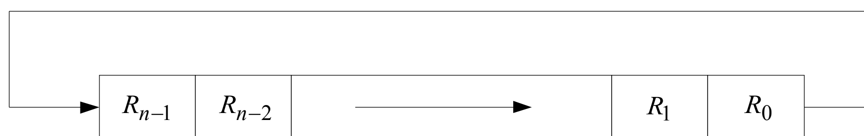


شکل ۱۲: شیفت منطقی به راست

۴-۴-۲- شیفت چرخشی

شیفت چرخشی (که عمل چرخش نیز خوانده می‌شود) بیت‌های ثبات را از طریق دو انتها بدون از دست دادن هر گونه اطلاعات می‌چرخاند. این عمل با اتصال خروجی سری به ورودی سری ثبات تحقق می‌یابد. عملیات شیفت چرخشی نیز به دو صورت انجام می‌شود: شیفت چرخشی به راست و شیفت چرخشی به چپ. عملیات شیفت چرخشی به راست در شکل ۱۳ نشان داده شده است. به طور

مثال اگر یک رجیستر ۸ بیتی را که دارای محتوای 00101101 می‌باشد یکبار به سمت راست شیفت چرخشی دهیم محتوای آن به صورت 10010110 خواهد شد.



شکل ۱۳: شیفت چرخشی به راست

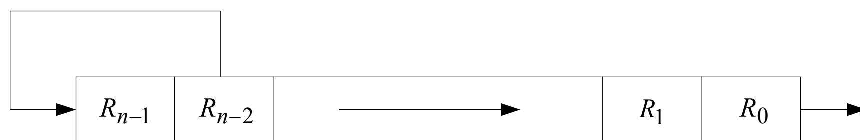
عملیات شیفت چرخشی به چپ نیز مشابه با عملیات شیفت چرخشی به راست است ولی جهت انتقال داده‌ها به سمت چپ می‌باشد.

۴-۴-۳- شیفت حسابی

شیفت حسابی عملی است که یک عدد دودویی علامت‌دار را به چپ یا به راست شیفت می‌دهد. شیفت حسابی نیز به دو صورت می‌تواند انجام شود: شیفت حسابی به راست و شیفت حسابی به چپ. شیفت حسابی به چپ دقیقاً همانند شیفت منطقی به چپ است. به طور مثال اگر یک رجیستر ۸ بیتی را که دارای محتوای 00000010 می‌باشد یکبار به سمت چپ شیفت حسابی دهیم محتوای آن به صورت 00000100 خواهد شد. اگر خوب دقت کنید متوجه می‌شوید که شیفت حسابی به چپ، یک عدد دودویی علامت‌دار را در 2 ضرب می‌کند به دلیل اینکه محتوای رجیستر قبل از شیفت ۲ بوده و بعد از شیفت ۴ شده است. یک شیفت حسابی به راست نیز عدد را بر 2 تقسیم می‌کند. شیفت‌های حسابی نباید بیت علامت را تغییر دهند زیرا وقتی عدد را در 2 ضرب یا تقسیم می‌کنیم علامت همچنان باقی می‌ماند.

لازم به ذکر است که سمت چپ‌ترین بیت در ثبات، بیت علامت را نگه می‌دارد، و بقیه بیت‌ها عدد را حفظ می‌کنند. بیت علامت برای اعداد مثبت، 0 و برای اعداد منفی،1 است. شکل ۱۴ نمونه‌ای از یک ثبات n بیت را نشان می‌دهد. بیت R_{n-1} در سمت چپ‌ترین مکان بیت علامت را نگه می‌دارد. شیفت به راست حسابی بیت علامت را عوض نمی‌کند و همه بیت‌ها (از جمله علامت) را به راست شیفت می‌دهد. بنابراین R_{n-1} تغییر نمی‌کند، R_{n-2} بیت R_{n-1} را دریافت می‌نماید و برای سایر بیت‌های

ثبات نیز به همین ترتیب. بیت واقع در R_0 از دست می‌رود. عملیات شیفت حسابی به چپ در شکل ۱۴ نشان داده شده است.



شکل ۱۴: شیفت حسابی به راست

شیفت حسابی به چپ یک 0 وارد R_0 می‌نماید، و کلیه بیت‌های دیگر را به چپ شیفت می‌دهد. در این عملیات مقدار اولیه R_{n-1} از دست رفته و با بیت R_{n-2} جایگزین می‌شود. اگر بیت واقع در R_{n-1} پس از شیفت عوض شود، در این صورت علامت معکوس شده است. این هنگامی رخ می‌دهد که ضرب در 2 سبب سرریز گردد. سرریز در شیفت به چپ هنگامی رخ می‌دهدکه قبل از شیفت R_{n-1} و R_{n-2} مساوی نباشد. یک فلیپ فلاپ به نام V برای کشف یک سرریز حاصل از شیفت حسابی به چپ می‌تواند مورد استفاده قرار گیرد.

$$V = R_{n-1} \oplus R_{n-2}$$

اگر V= 0 باشد، سرریز وجود ندارد، ولی اگر V=1 گردد، سرریز وجود داشته و پس از شیفت علامت عوض خواهد شد.

۴-۴-۴- دستورات شیفت پردازنده MIPS

دستور شیفت منطقی به چپ:

```
sll  rd, rt, shamt  //  rd = rt << shamt
```

مثال:

```
sll  $v0, $t0, 4    //  $v0 = $t0 << 4
```

دستور شیفت منطقی به راست:

```
srl  rd, rt, shamt    //  rd = rt >> shamt
```

دستور شیفت حسابی به راست:

```
sra  rd, rt, shamt   //  rd = rt >> shamt
```

۴-۵- عملیات ضرب

ضرب دو عدد ممیز ثابت دودوئی با نمایش مقدار علامت[1] با قلم و کاغذ، توسط فرآیند شیفت-های متوالی و جمع انجام می‌شود. این روش را با مثالی می‌توان بهتر تشریح نمود.

```
 23        10111  مضروب Multiplicand
 19      * 10011  مضروب فیه Multiplier
 ----    -------
           10111
          10111
         00000
        00000     +
       10111
      ---------
437   110110101  حاصل ضرب Product
```

فرآیند به این صورت است که بیت‌های مضروب‌فیه متوالیاً، با شروع از کم ارزشترین بیت بررسی می‌شوند. اگر بیت مضروب‌فیه ۱ باشد، مضروب در پائین کپی می‌شود، در غیر این صورت صفرها در پائین کپی می‌گردند. اعدادی که در سطرهای متوالی کپی می‌شوند نسبت به سطر قبل یک بیت به چپ شیفت داده می‌شوند. نهایتاً اعداد با هم جمع شده و حاصل‌جمع همه سطرها نتیجه ضرب خواهد بود. علامت حاصل‌ضرب با توجه به علامت‌های مضروب و مضروب‌فیه معین می‌شود. اگر علامتها یکی باشند علامت حاصل‌ضرب مثبت و در غیر این صورت منفی خواهد بود.

۴-۵-۱- پیاده سازی سخت‌افزاری برای داده‌های علامت‌دار

هنگام پیاده‌سازی ضرب در یک کامپیوتر دیجیتال، بهتر است فرآیند کمی تغییر یابد. اولاً به جای تهیه ثباتهائی به تعداد بیت‌های مضروب‌فیه برای ذخیره و جمع همزمان چند عدد دودوئی، بهتر است جمع کننده‌ای برای جمع فقط دو عدد دودوئی در نظر گرفته شود و مرتباً حاصل‌ضربهای جزئی[2] را در یک ثبات نگهداری نمائیم. ثانیاً به جای شیفت مضروب به چپ، حاصل‌ضرب جزئی به راست شیفت داده شود که در نتیجه موقعیت نسبی حاصل‌ضرب جزئی و مضروب همان موقعیت مطلوب خواهد بود. ثالثاً وقتی‌که بیتی از مضروب‌فیه صفر باشد، لزومی ندارد که صفرها را با حاصل‌ضرب جزئی جمع کنیم زیرا مقدار آن را عوض نمی‌کند.

---

[1] - Sign magnitude
[2] - Partial product

سخت‌افزار ضرب در شکل ۱۵ و فلوچارت آن در شکل ۱۶ نشان داده شده است. همان طور که مشاهده می‌شود در سخت افزار ضرب چند رجیستر یا ثبات به همراه چند فلیپ فلاپ مشاهده می-شود. فلیپ فلاپ‌ها برای نگهداری علامت عددها به کار می‌روند. همچنین در این سخت افزار یک جمع کننده نیز تعبیه شده است که حاصل جمع‌های میانی را انجام می‌دهد.

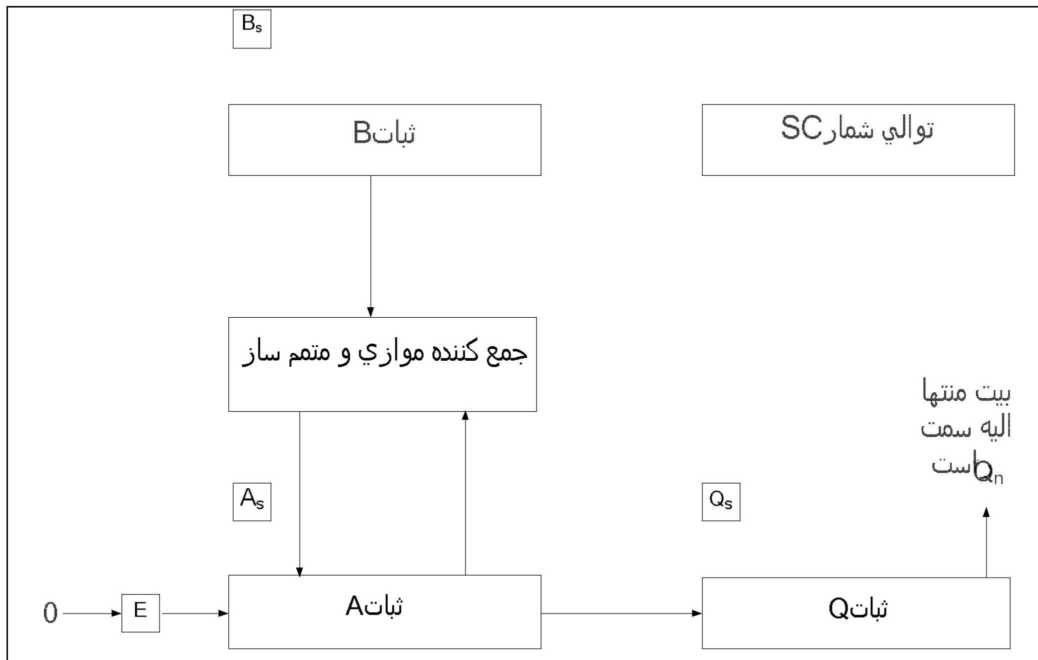
چون تعداد بیت‌های حاصل ضرب دو عدد، از تعداد بیت‌های هر دو عدد بیشتر است، بنابراین در این الگوریتم حاصل‌ضرب نهائی در ثباتهای A و Q قرار خواهد گرفت.

در ابتدا مضروب‌فیه در ثبات Q و علامتش در Q_s ذخیره می‌شود. همچنین مضروب در داخل ثبات B و علامتش در B_s قرار داده می‌شود و ثباتA نیز با صفر پر می‌شود. توالی شمار[1] SC نقش شمارنده را دارد و در ابتدا با عددی برابر با تعداد بیتهای مضروب‌فیه مقداردهی می‌شود. شمارنده پس از هر بار تشکیل حاصل‌ضرب جزئی یک واحد کم می‌شود. وقتی که محتوای شمارنده به صفر برسد، حاصل‌ضرب تکمیل و فرآیند متوقف می‌گردد.

در هر مرحله، حاصل‌جمع A وB تشکیل حاصل‌ضرب جزئی را می‌دهند که به ثبات EA منتقل می‌گردد. حاصل‌ضرب جزئی و مضروب‌فیه توأماً به راست شیفت داده می‌شوند. این شیفت توسط عبارت shr EAQ نشان داده می‌شود. بیت E به با ارزشترین بیت A نقل مکان می‌یابد، و یک 0 وارد E می‌شود، بیت کم ارزشتر A به با ارزشترین بیت Q منتقل می‌شود، به عبارتی یک بیت از حاصل‌ضرب جزئی به داخل Q شیفت پیدا کرده و بیت‌های مضروب‌فیه را یک واحد به سمت راست می‌راند. به این ترتیب، سمت راست‌ترین فلیپ فلاپ در ثبات Q، که با Q_n مشخص شده است حاوی بیتی از مضروب‌فیه خواهد بود که در مرحله بعدی باید بررسی شود.

---

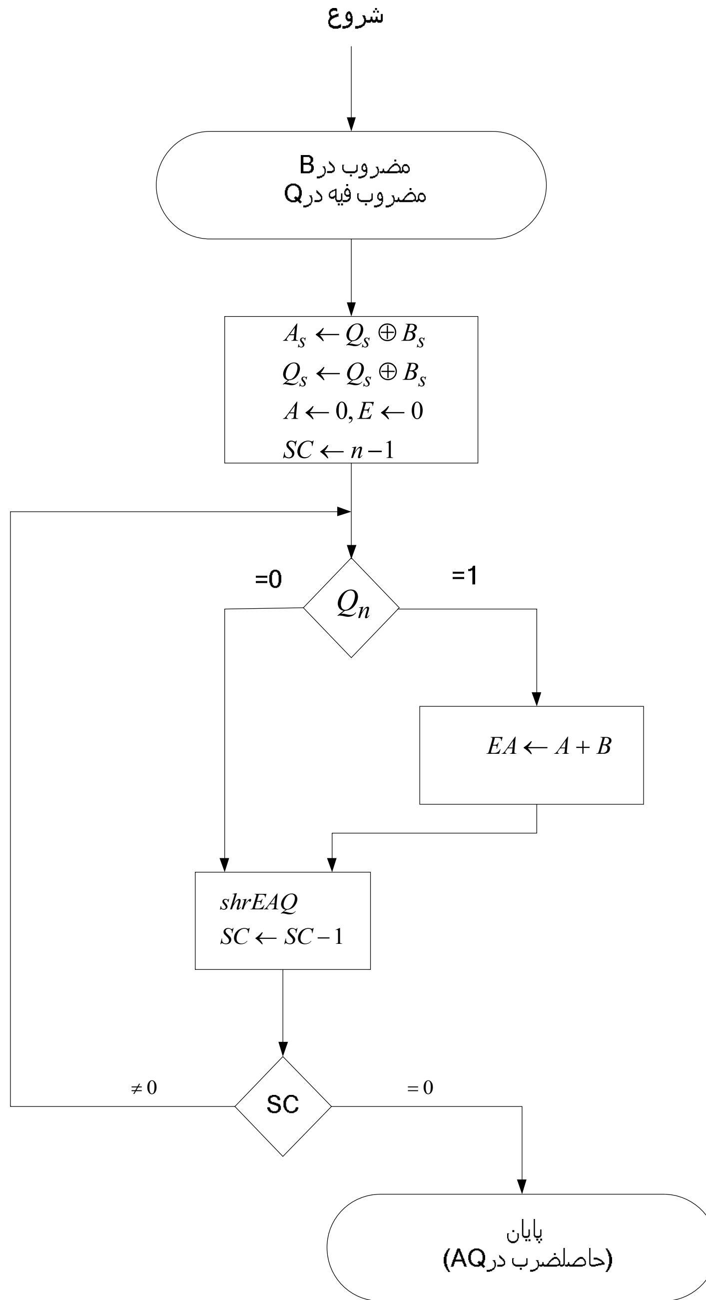
[1] - Sequence Counter



شکل ۱۵: سخت افزار عمل ضرب

۴-۵-۲- الگوریتم سخت‌افزاری

فلوچارت الگوریتم سخت‌افزاری ضرب در شکل ۱۶ نشان داده شده است. در شروع کار مضروب در B و مضروب‌فیه در Q و علامت‌های مربوطه به ترتیب در B_s و Q_s قرار دارند. در ابتدا علامت‌ها با هم مقایسه شده و نتیجه این مقایسه علامت حاصل ضرب دو عدد را تعیین می‌کند که در A_s و Q_s قرار داده می‌شود.



شکل ۱۶: فلوچارت عمل ضرب

ثباتهای A و E پاک شده و توالی شمار SC برابر با تعداد بیتهای مضروب فیه می‌شود. در اینجا فرض می‌کنیم که عملوندها از حافظه‌های n بیتی به ثباتها منتقل شده‌اند. چون هر عملوند باید همراه با علامتش ذخیره شود لذا یک بیت از کلمه توسط علامت اشغال شده و مقدار عملوند n-1 بیتی خواهد بود.

پس از دادن مقادیر اولیه، بیت کم‌ارزشتر مضروب فیه که در Q_n قرار دارد تست می‌شود. اگر این بیت برابر با 1 باشد، مضروب موجود در B با حاصل‌ضرب جزئی موجود در A جمع می‌شود. و اگر نتیجه تست 0 باشد، کاری انجام نمی‌شود. سپس مجموعه ثبات EAQ یک بار به سمت راست شیفت داده می‌شود تا حاصل‌ضرب جزئی را تشکیل دهد. سپس توالی شمار 1 واحد کم شده و مقدار جدید آن چک می‌شود. اگر برابر صفر نباشد، فرآیند تکرار شده و حاصل‌ضرب جزئی جدید تشکیل می‌گردد. وقتی که SC=0 شود فرآیند متوقف می‌گردد. توجه کنید که حاصل‌ضرب جزئی حاصل در A هر بار یک بیت به Q منتقل می‌گردد تا نهایتاً جای مضروب‌فیه را می‌گیرد. حاصل‌ضرب نهائی در هر دو ثبات A و Q واقع است به این ترتیب که A بیت‌های با ارزشتر و Q بیت‌های کم ارزشتر را نگه می‌دارند. مثال عددی قبلی، برای روشن شدن فرآیند ضرب سخت‌افزاری در جدول ۲ تکرار شده است. روند، مراحل مشخص شده در فلوچارت را دنبال می‌کند.

جدول ۲: مثال عددی برای ضرب کننده دودویی

مضروب B = 10111	E	A	Q	SC
مضروب فیه در Q	0	00000	10011	101
Q_n=1 ؛ جمع B		10111		
اولین حاصلضرب جزیی	0	10111		
شیفت EAQ به راست و کم کردن شمارنده	0	01011	11001	100
Q_n=1 ؛ جمع B		10111		
دومین حاصلضرب جزیی	1	00010		
شیفت EAQ به راست و کم کردن شمارنده	0	10001	01100	011
Q_n=0 ؛ شیفت EAQ به راست و کم کردن شمارنده	0	01000	10110	010
Q_n=0 ؛ شیفت EAQ به راست و کم کردن شمارنده	0	00100	01011	001
Q_n=1 ؛ جمع B		10111		
پنجمین حاصلضرب جزیی	0	11011		
شیفت EAQ به راست و کم کردن شمارنده	0	01101	10101	000
حاصلضرب نهایی در AQ برابر 0110110101 است				

اگر فرض کنیم که در الگوریتم فوق هر کدام از مراحل در یک کلاک قابل انجام باشند در این صورت یک ضرب ۳۲ بیتی برای اجرا شدن نیاز به حداقل ۳۲ کلاک خواهد داشت (برای مقداردهی-های اولیه و همچنین مراحل پایانی کار که نتیجه قرار است در رجیستر و یا حافظه نوشته شود نیز کلاک‌هایی لازم است). به عبارتی می‌توان گفت که با استفاده از پیاده سازی فوق CPI دستور ضرب

حداقل برابر ۳۲ خواهد بود. به همین دلیل دستور ضرب در پردازنده‌ها معمولاً از منظر پیاده سازی و هزینه، دستور سنگینی به حساب می‌آید.

۴-۵-۳- الگوریتم ضرب بوث

الگوریتم بوث رویه‌ای را برای ضرب اعداد دودوئی در نمایش متمم ۲ علامت‌دار ارائه می-نماید. مبنای کار الگوریتم بر این اساس استوار است که رشته‌های 0 در مضروب‌فیه نیازی به جمع ندارند بلکه فقط جابجائی (شیفت) لازم دارند و رشته‌های 1 در مضروب‌فیه از بیت مرتبه 2^k تا بیت 2^m را می‌توان معادل $2^{k+1}-2^m$ تلقی کرد. به طور مثال عدد دودوئی 001110 (+14) دارای رشته‌های 1 از 2^1 تا 2^3 است(m=1,k=3). این عدد را می‌توان به صورت $2^{k+1}-2^m = 2^4 - 2^1 = 16 - 2 = 14$ نوشت. بنابراین ضرب M*14 را، که درآن M مضروب و 14 مضروب‌فیه است را می‌توان به صورت $M \times 2^4 - M \times 2^1$ انجام داد. لذا حاصل‌ضرب با چهار بار شیفت به چپ M و تفریق یک‌بار شیفت به چپ داده شده M از آن به دست می‌آید.

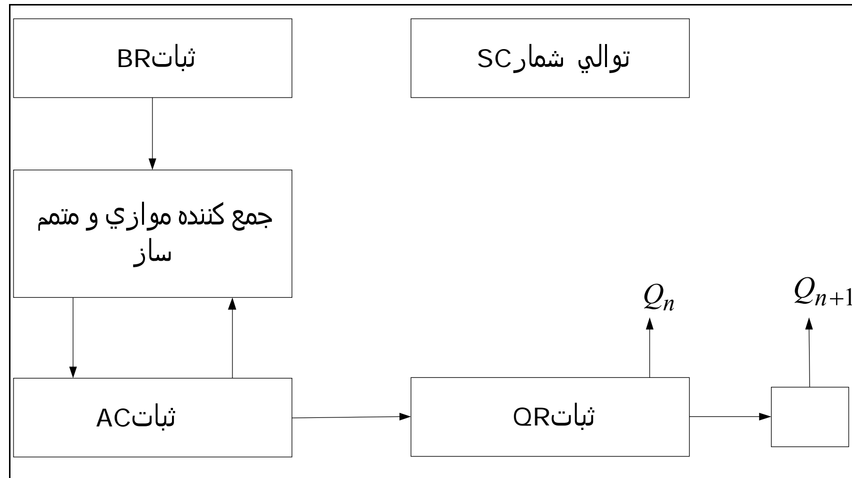
همانند همهٔ روشهای ضرب، الگوریتم بوث نیز به بررسی بیت‌های مضروب فیه و شیفت حاصل-ضرب جزئی نیاز دارد. قبل از شیفت، ممکن است مضروب طبق قواعد زیر با حاصل‌ضرب جزئی جمع شود، از آن تفریق شود و یا حاصل‌ضرب جزئی بلا تغییر باقی بماند.

۱. به محض برخورد با اولین 1 کم ارزش در رشته 1 ها در مضروب‌فیه، مضروب از حاصل‌ضرب جزئی کم می‌شود.

۲. به محض برخورد با اولین 0 (به شرطی که قبل از آن ۱ باشد) در رشته‌ای از 0 ها در مضروب-فیه، مضروب با حاصل‌ضرب جزئی جمع می‌شود.

۳. وقتی که بیت جاری مضروب‌فیه همانند بیت قبلی باشد، حاصل‌ضرب جزئی تغییر نمی‌کند.

الگوریتم فوق برای مضروب‌فیه‌های مثبت و یا منفی به فرم متمم 2 قابل استفاده است. این بدان علت است که مضروب فیه منفی با رشته‌ای از 1 ها خاتمه می‌یابد و آخرین عمل تفریق با وزن مناسب خواهد بود. مثلاً مضروب‌فیه 14- در نمایش متمم 2 عبارت است از 110010 و به صورت $-2^4 + 2^2 - 2^1 = -14$ با آن رفتار می‌شود.

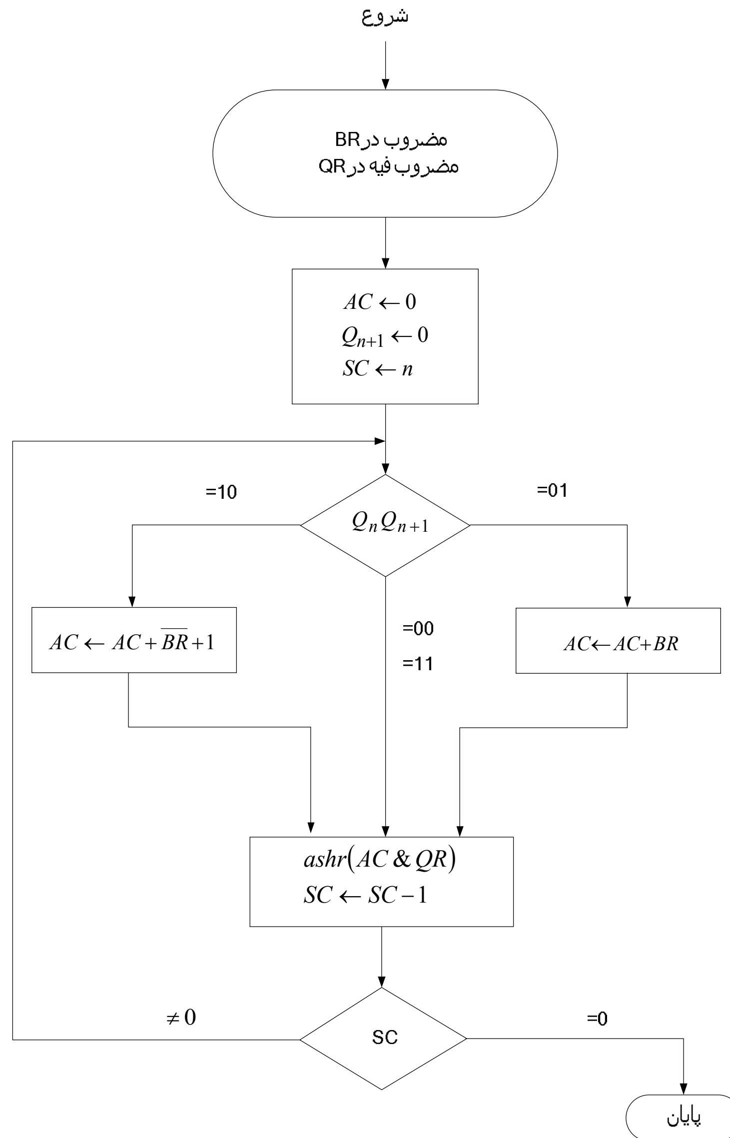


شکل ۱۷: سخت افزار الگوریتم بوث

پیاده‌سازی سخت‌افزار الگوریتم بوث آرایش ثبات شکل ۱۷ را نیاز دارد. این شکل مشابه شکل ۱۵ است جز اینکه بیت‌های علامت از بقیه ثباتها تفکیک نشده‌اند. برای ملاحظه این اختلاف، ما ثبات-های A و B و Q را به ترتیب AC و BR و QR نام‌گذاری کرده‌ایم. Q_n کم‌ارزشترین بیت در ثبات QR است. یک فلیپ فلاپ اضافی Q_{n+1} برای بررسی همزمان دو بیت از مضروب‌فیه به QR ملحق شده است. فلوچارت الگوریتم بوث در شکل ۱۸ نشان داده شده است. در ابتدا AC و بیت الحاقی پاک می-شوند و شمارنده SC برابر تعداد بیت‌های مضروب‌فیه یعنی n قرار داده می‌شود. در هر مرحله دو بیت از مضروب‌فیه که در Q_n و Q_{n+1} قرار دارند مورد بررسی قرار می‌گیرند. اگر دو بیت برابر 10 باشند بدان معنی است که اولین 1 در رشته 1 ها فرا رسیده است. این حالت، تفریق مضروب از حاصل‌ضرب جزئی موجود در AC را لازم می‌دارد. اگر دو بیت 01 باشد، مفهوم این است که با اولین 0 در رشته 0 ها برخورد شده است. این حالت جمع مضروب با حاصل‌ضرب جزئی موجود در AC را لازم می‌دارد. هرگاه دو بیت برابر باشند، حاصل‌ضرب جزئی تغییر نمی‌کند.

قدم بعدی شیفت حاصل‌ضرب جزئی و مضروب‌فیه (شامل بیت Q_{n+1}) به راست است. این یک عمل شیفت حسابی به راست است که AC و QR را به راست جابه‌جا کرده و بیت علامت در AC بلا-تغییر می‌ماند. در آخر هر مرحله شمارنده SC کاهش یافته و با صفر مقایسه می‌شود. حلقه محاسبه الگوریتم بوث n بار تکرار می‌گردد.

در الگوریتم بوث سرریز نمی‌تواند رخ دهد زیرا جمع و تفریق مضروب یک در میان انجام می-شود. در نتیجه دو عددی که جمع می‌شوند، همواره علامت‌های مخالف دارند، و به این ترتیب امکان وقوع سرریز وجود نخواهد داشت.



شکل ۱۸: الگوریتم بوث برای ضرب اعداد متمم ۲ علامت‌دار

مثال عددی الگوریتم بوث در جدول ۳ برای n=5 نشان داده شده است. این مثال، ضرب $(-9)\times(-13)=+117$ را نشان می‌دهد. توجه کنید که مضروب فیه در QR منفی و مضروب نیز در BR منفی است. نتیجه نهایی عملیات ضرب ۱۰ بیتی بوده و در جفت ثبات AC و QR قرار می‌گیرد.

برای این مثال، حاصل‌ضرب یک عدد مثبت است. توجه کنید که مقدار نهائی Q_{n+1} علامت اولیه مضروب‌فیه است و نباید بخشی از حاصل‌ضرب تلقی شود.

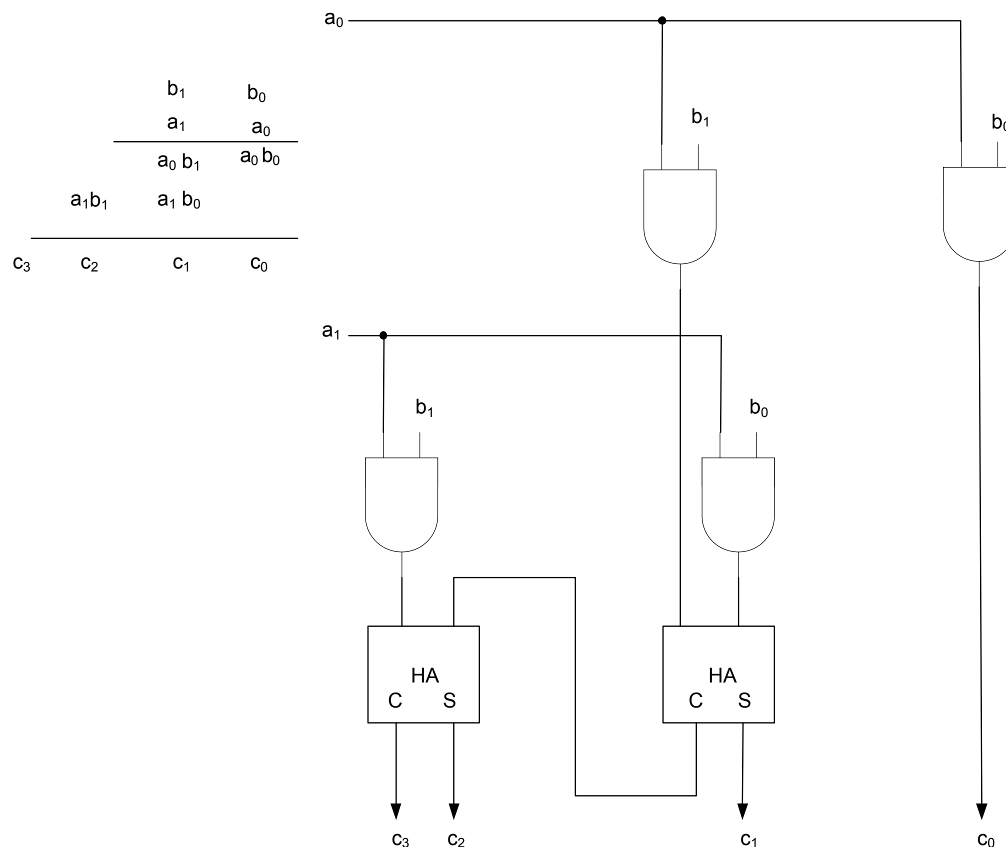
جدول ۳: مثال ضرب با الگوریتم بوث

Q_nQ_{n+1}	BR = 10111 $\overline{BR}+1=01001$	AC	QR	Q_{n+1}	SC
	مقدار اولیه	00000	10011	0	101
10	تفریق BR	01001			
		01001			
	ashr	00100	11001	1	100
11	ashr	00010	01100	1	011
01	جمع BR	10111			
		11001			
	ashr	11100	10110	0	010
00	ashr	11110	01011	0	001
10	تفریق BR	01001			
		00111			
	ashr	00011	10101	1	000

۴-۵-۴- ضرب کننده آرایه‌ای

تست یک به یک بیت‌های مضروب‌فیه و تشکیل حاصل‌ضرب جزئی یک عمل ترتیبی است که دنباله‌ای از اعمال جمع و شیفت را نیاز دارد. بنابراین مدارهای فوق مدارهای ترتیبی هستند. مدار ضرب کننده را می‌توان با یک مدار کاملاً ترکیبی نیز پیاده سازی نمود. در این قسمت قصد داریم که یک ضرب کننده ترکیبی طراحی کنیم. به دلیل ساختار منظم سخت افزار در این ضرب کننده به آن ضرب کننده آرایه‌ای گفته می‌شود. برای اینکه ببینیم یک ضرب کننده ترکیبی چگونه پیاده‌سازی می‌شود، ضرب دو عدد ۲ بیت را مطابق شکل ۱۹ در نظر بگیرید. بیت‌های مضروب b_1 و b_0 و بیت‌های مضروب‌فیه a_1 و a_0 و حاصل‌ضرب $c_3c_2c_1c_0$ می‌باشند. اولین حاصل‌ضرب جزئی با ضرب a_0 در b_1b_0 بدست می‌آید. ضرب دو بیت مانند a_0 و b_0 برابر 1 خواهد بود به شرطی که هر دو 1 باشند، در غیر این صورت 0 است. این همان عمل AND است و بنابراین ضرب دو بیت را می‌توان با یک گیت AND پیاده‌سازی نمود. همان طور که در شکل دیده می‌شود، اولین حاصل‌ضرب جزئی با دو گیت AND

حاصل می‌شود. دومین حاصل‌ضرب جزئی از ضرب a_1 در b_1b_0 و شیفت آن به چپ بدست می‌آید. دو حاصل‌ضرب جزئی به وسیله دو مدار نیم جمع‌کننده (HA) با یکدیگر جمع شده و نتیجه نهایی بدست می‌آید. معمولاً تعداد بیت‌های حاصل‌ضرب جزئی بیشتر است و لازم خواهد بود تا از مدار تمام جمع کننده استفاده شود. دقت داشته باشید که کم ارزشترین بیت حاصل‌ضرب لازم نیست وارد جمع-کننده شود زیرا توسط خروجی گیت AND تشکیل می‌گردد.



شکل ۱۹: ضرب کننده آرایه‌ای ۲ بیت در ۲ بیت

یک مدار ضرب‌کننده دودوئی با بیت‌های بیشتر را می‌توان به روشی مشابه ساخت. هر بیت از مضروب‌فیه با هر بیت از مضروب، به تعداد بیت‌های مضروب فیه، AND می‌شود. خروجی دودوئی در هر یک از سطوح گیت‌های AND به طور موازی با حاصل‌ضرب جزئی قبلی جمع می‌شود و حاصل-ضرب جزئی جدیدی را تشکیل می‌دهد. آخرین سطح، حاصل‌ضرب را ایجاد می‌نماید.

توجه: برای مضروب‌فیه j بیتی و مضروب k بیتی به تعداد $j \times k$ گیت AND و $(j-1)k$ جمع کننده k بیتی برای تولید حاصل‌ضرب j+k بیتی مورد نیاز است.

به عنوان مثال دوم، یک مدار ضرب‌کننده را در نظر بگیرید که یک عدد دودوئی چهار بیتی را در یک عدد سه بیتی ضرب می‌کند. فرض کنید مضروب را با $b_3b_2b_1b_0$ و مضروب‌فیه را با $a_2a_1a_0$ نشان دهیم. چون k=4 و j=3 است، 12 گیت AND و 2 جمع کننده 4 بیتی مورد نیاز است تا حاصل-ضرب هفت بیتی را ایجاد کند. نمودار منطقی این ضرب‌کننده در شکل ۲۰ نشان داده شده است.

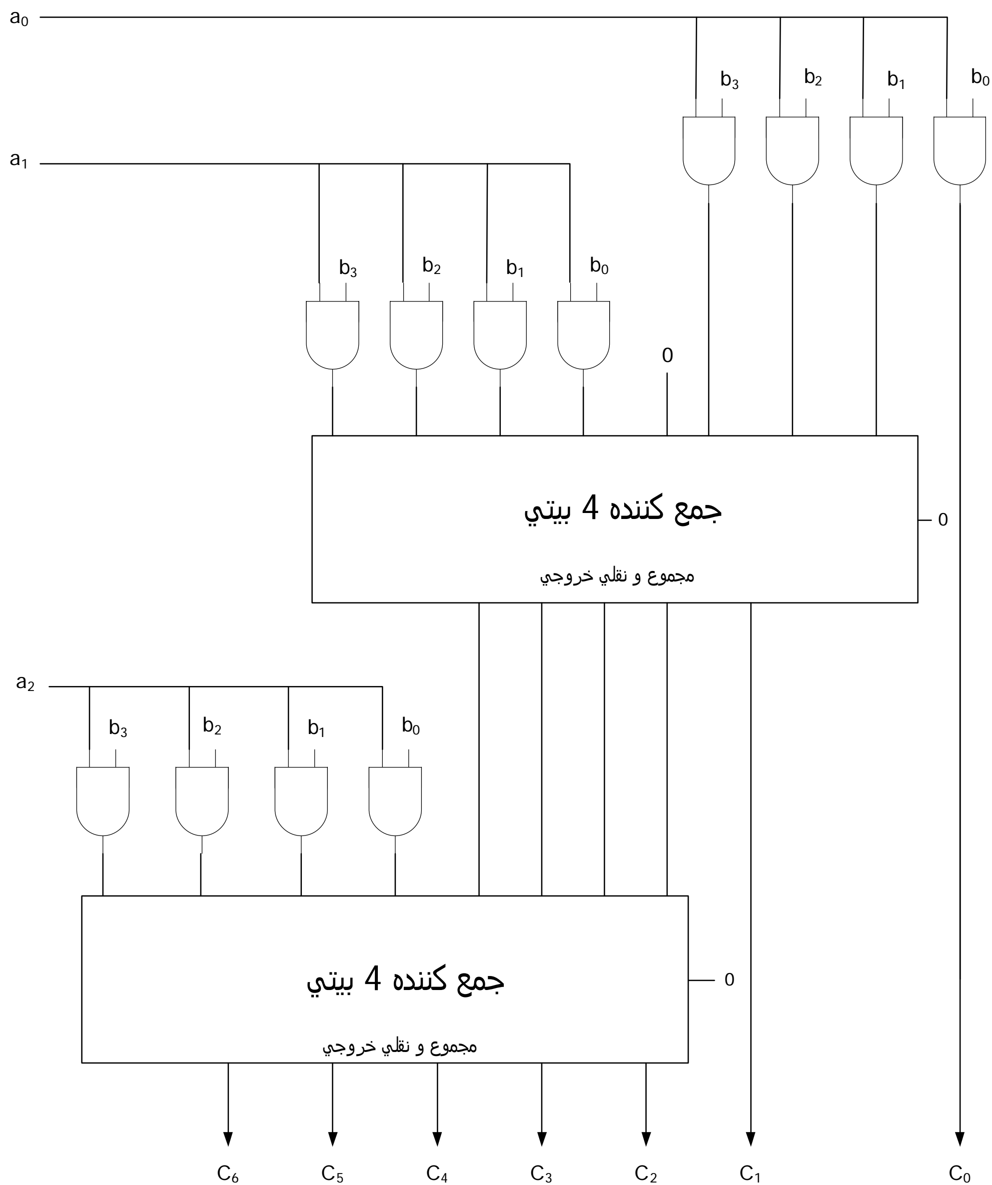
اگر سخت افزار عملیات ضرب را به صورت آرایه‌ای پیاده سازی کنیم، در این صورت عملیات ضرب در یک کلاک قابل انجام است. در این مدار تأخیر مسیر از ورودی‌ها به خروجی‌ها به دلیل وجود تعداد زیادی گیت بسیار بالا است به طور مثال حالتی را در نظر بگیرید که می‌خواهیم دو عدد ۳۲ بیتی را در هم ضرب کنیم در این حالت به تعداد ۳۱ عدد جمع کننده علاوه بر گیتهای AND در مسیر وجود خواهد داشت. بنابراین اگر بخواهیم این عملیات را در یک کلاک انجام دهیم باید پریود کلاک حداقل به اندازه ماکزیمم تأخیر مدار باشد که در نتیجه آن فرکانس کاری پردازنده به شدت پائین خواهد آمد.

۴-۵-۵- عملیات و دستورات ضرب پردازنده MIPS

پردازنده MIPS دو عدد رجیستر ۳۲ بیتی مخصوص به نام‌های Hi و Lo برای عملیات ضرب در نظرگرفته است. این دو رجیستر حاصل‌ضرب دو عدد ۳۲ بیتی را که ۶۴ بیت است نگهداری می‌کنند. ۳۲ بیت پر ارزش حاصل‌ضرب در Hi و ۳۲ بیت کم ارزش آن در Lo ریخته می‌شود. لازم به ذکر است که این دو رجیستر جزو ۳۲ رجیستر پردازنده MIPS که قبلاً توضیح داده شده است (رجیسترهای عمومی)، نیستند بلکه رجیسترهای مخصوصی هستند که فقط برای عملیات ضرب و تقسیم استفاده می-شوند. پردازنده MIPS، دارای دو نوع دستور ضرب است: ضرب اعداد علامت دار و ضرب اعداد بدون علامت. دستور mult[1] برای ضرب دو عدد علامت‌دار و دستور multu[2] برای ضرب دو عدد

[1] - Multiply
[2] - Multiply unsigned

بدون علامت به کار می‌رود. می‌توانیم هر موقع که لازم شد محتوای رجیسترهای Hi و Lo را به داخل رجیسترهای عمومی پردازنده منتقل کنیم. این کار به کمک دو دستور mfhi[1] و mflo[2] انجام می‌شود.



شکل ۲۰: ضرب کننده آرایه‌ای ۴ بیت در ۳ بیت

[1] - Move from high
[2] - Move from low

۴-۶- عملیات تقسیم

در تقسیم دو عدد ممیز ثابت در نمایش مقدار–علامت با قلم و کاغذ، از اعمال مقایسه، شیفت و تفریق استفاده می‌شود. تقسیم دودوئی از تقسیم دهدهی ساده‌تر است زیرا ارقام خارج قسمت 0 یا 1 هستند و نیازی نیست تا بدانیم مقسوم علیه چند بار در مقسوم جای می‌گیرد. فرآیند تقسیم با مثال عددی شکل ۲۱ توضیح داده شده است. مقسوم‌علیه B از پنج بیت و مقسوم A از ده بیت تسکیل شده است. در ابتدا پنج بیت با ارزش‌تر مقسوم با مقسوم‌علیه مقایسه می‌شوند. چون این پنج بیت از B کوچکتر است، مقایسه B را با شش بیت با ارزشتر A آزمایش می‌کنیم. این عدد شش بیتی بزرگتر از B است، لذا رقم 1 را برای خارج قسمت می‌نویسیم. سپس مقسوم‌علیه را یک بار به راست شیفت می‌دهیم و آن را از مقسوم کم می‌کنیم. تفاضل بدست آمده باقیمانده جزئی[1] خوانده می‌شود زیرا تقسیم ممکن است در همین مرحله پایان یابد و خارج‌قسمتی برابر با 1 و باقیمانده‌ای برابر با این باقیمانده جزئی داشته باشد.

توضیح	A	B	
(448) Dividend یا مقسوم	A= 0111000000	B=10001	Divisor یا مقسوم علیه (17)
خارج قسمت 5 بیت دارد، 5 بیت از A ، A<B	01110	11010	Quotient خارج قسمت یا (26)
6 بیت از A ، A>=B	011100		
جابجایی B به راست و تفریق ؛ قرار دادن 1 در Q	-10001		
باقیمانده بزرگتر یا مساوی با B	-010110		
جابجایی B به راست و تفریق ؛ قرار دادن 1 در Q	--10001		
باقیمانده کوچکتر از B ؛ قرار دادن 0 در Q ؛ شیفت B به راست	--001010		
باقیمانده بزرگتر یا مساوی با B	---010100		
جابجایی B به راست و تفریق ؛ قرار دادن 1 در Q	----10001		
باقیمانده کوچکتر از B ؛ قرار دادن 0 در Q	----000110		
باقیمانده نهایی (6)	-----00110		

شکل ۲۱: مثال تقسیم دودویی

روند تقسیم با مقایسه باقیمانده جزئی با مقسوم‌علیه ادامه می‌یابد. اگر باقیمانده جزئی بزرگتر یا مساوی مقسوم‌علیه باشد، بیت خارج قسمت برابر 1 شده و سپس مقسوم‌علیه یک واحد به راست شیفت داده شده و از باقیمانده جزئی کم می‌شود. اگر باقیمانده جزئی کوچکتر از مقسوم علیه باشد، بیت

[1] - Partial reminder

خارج قسمت 0 می‌شود و تفریق لازم نیست. در هر صورت مقسوم‌علیه یک مرتبه به راست شیفت داده می‌شود. توجه کنید که در نتیجه این روند هم خارج‌قسمت و هم باقیمانده بدست خواهد آمد.

۴-۶-۱- پیاده‌سازی سخت‌افزاری تقسیم برای داده‌های با نمایش مقدار-علامت

هنگام پیاده‌سازی تقسیم در یک کامپیوتر دیجیتال، بهتر است روند کمی تغییر یابد. به جای شیفت مقسوم‌علیه به سمت راست، مقسوم یا باقیمانده جزئی را به چپ شیفت می‌دهیم و به این ترتیب موقعیت نسبی دو عدد همان موقعیت مورد نظر خواهد بود. تفریق را می‌توان با جمع A و متمم 2 عدد B انجام داد. نسبت اندازه‌ها از رقم نقلی انتهائی مشخص می گردد.

سخت‌افزار مربوط به پیاده‌سازی تقسیم همان سخت‌افزار ضرب است و از قطعات شکل ۱۵ تشکیل شده است. در اینجا ثبات EAQ به چپ شیفت داده شده و 0 در Q_n وارد شده و مقدار قبلی E هم از دست می‌رود. مثال عددی شکل ۲۱ برای روشنتر کردن روند تقسیم مجدداً در شکل ۲۲ نشان داده شده است. مقسوم‌علیه در ثبات B و مقسوم که طولی دو برابر دارد در ثباتهای A و Q ذخیره می-شوند. مقسوم به سمت چپ شیفت داده می‌شود و متمم 2 مقسوم‌علیه با آن جمع شده و لذا عمل تفریق تحقق می‌یابد. اطلاعات مربوط به نسبت اندازه‌ها در E موجود است. اگر E=1 باشد به معنی $A \geq B$ است که در این صورت باقیمانده جزئی به سمت چپ شیفت پیدا کرده و بیت 1 وارد Q_n می-گردد. اگر E=0 باشد، یعنی A<B است و لازم نبوده که B از A کم شود، به همین دلیل B مجدداً با A جمع می‌شود تا باقیمانده جزئی در A به مقدار قبلی‌اش باز گردانده شود. در این حالت باقیمانده جزئی به چپ شیفت داده شده و بیت 0 وارد Q_n می‌گردد. روند مجدداً تکرار می‌گردد تا اینکه هر پنج بیت خارج‌قسمت ایجاد شود.

توجه کنید که ضمن شیفت باقیمانده جزئی به چپ، بیت‌های خارج‌قسمت نیز شیفت داده می-شود و پس از پنج بار شیفت، خارج‌قسمت در Q و باقیمانده در A خواهد بود. قبل از نمایش الگوریتم به صورت فلوچارت، باید علامت نتیجه و حالت سرریز احتمالی را در نظر داشت. علامت خارج‌قسمت از علامت‌های مقسوم و مقسوم‌علیه تعیین می‌شود. اگر علامت‌های این دو یکسان باشند، علامت خارج قسمت مثبت خواهد بود و اگر مخالف باشند، علامت منفی است. علامت باقیمانده همانند علامت مقسوم است.

مقسوم علیه $B = 10001$ $\qquad \overline{B}+1 = 01111$

	E	A	Q	SC
مقسوم		01110	00000	5
shl EAQ	0	11100	00000	
جمع $\overline{B}+1$		01111		
E=1	1	01011		
قرار دهید $Q_n=1$	1	01011	00001	4
shl EAQ	0	10110	00010	
جمع $\overline{B}+1$		01111		
E=1	1	00101		
قرار دهید $Q_n=1$	1	00101	00011	3
shl EAQ	0	01010	00110	
جمع $\overline{B}+1$		01111		
E=0 ، بگذارید $Q_n=0$	0	11001	00110	
جمع B		10001		
بازيابي باقيمانده	1	01010		2
shl EAQ	0	10100	01100	
جمع $\overline{B}+1$		01111		
E=1	1	00011		
قرار دهید $Q_n=1$	1	00011	01101	1
shl EAQ	0	00110	11010	
جمع $\overline{B}+1$		01111		
E=0 ، بگذارید $Q_n=0$	0	10101	11010	
جمع B		10001		
بازيابي باقيمانده	1	00110	11010	0
باقیمانده در A		00110		
خارج قسمت در Q			11010	

شکل ۲۲: مثال تقسیم دودویی با سخت افزار دیجیتال

۴-۶-۲- سرریز در تقسیم

عمل تقسیم ممکن است منجر به سرریز در خارج قسمت شود. هنگام استفاده از کاغذ و قلم، این امر مشکلی ایجاد نمی‌کند ولی هنگام پیاده‌سازی عمل با سخت‌افزار، مسئله ساز خواهد بود. دلیل این است که طول ثباتها محدود است و نمی‌توانند عددی را که اندازه آن از طول استاندارد تجاوز می‌کند، نگهداری کنند. برای درک موضوع، سیستمی را که دارای ثبات‌های پنج بیتی است در نظر بگیرید. ما از یک ثبات برای نگهداری مقسوم علیه و از دو ثبات برای نگهداری مقسوم استفاده می‌کنیم. با توجه به مثال شکل ۲۱ می‌بینیم که اگر پنج بیت با ارزشتر مقسوم، عددی بزرگتر از مقسوم علیه

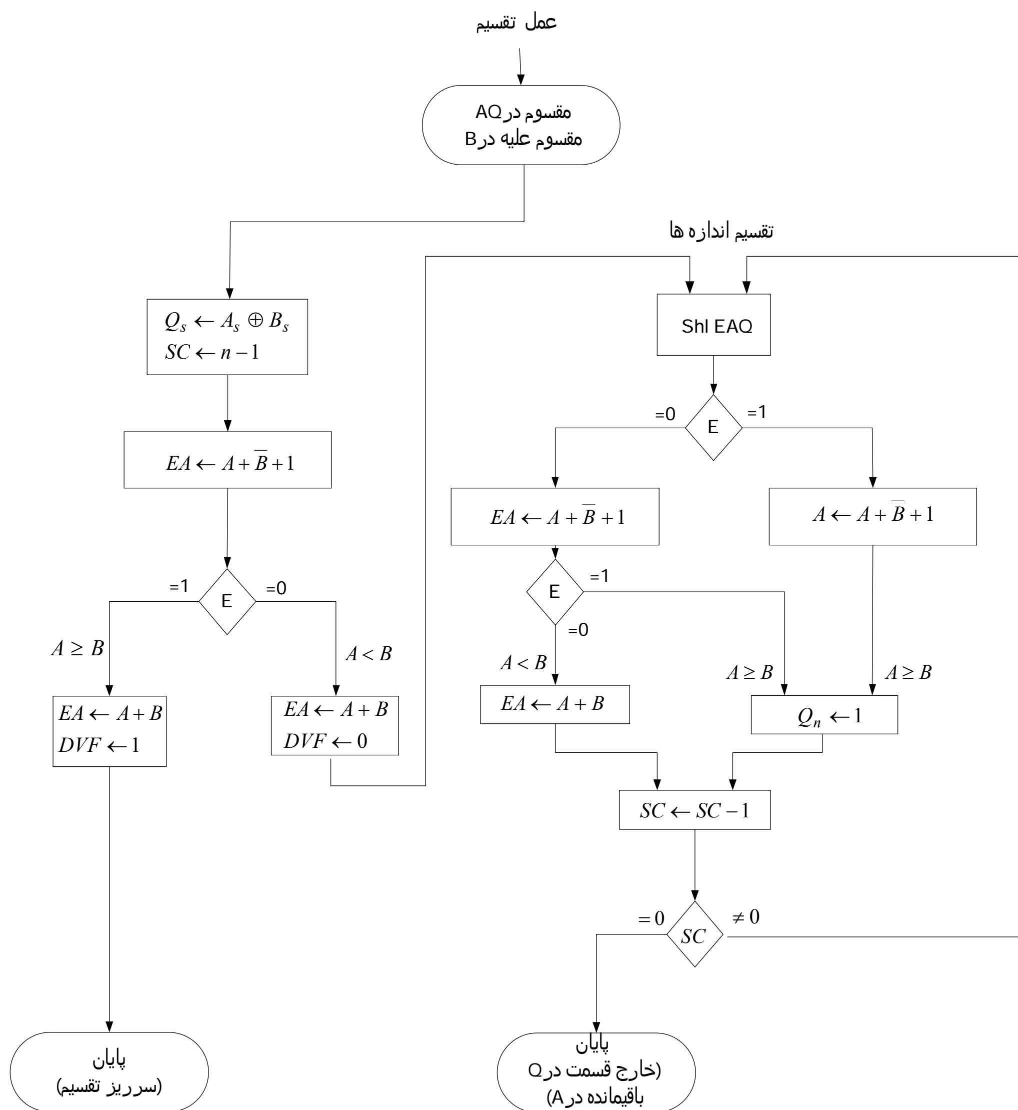
باشد، خارج قسمت شش بیتی خواهد بود. خارج قسمت می‌باید در یک ثبات پنج بیتی استاندارد ذخیره شود، بنابراین بیت سرریز مستلزم یک فلیپ فلاپ اضافی برای ذخیره بیت ششم خواهد بود. در عملیات عادی کامپیوتر باید از به کارگیری این بیت سرریز خودداری شود زیرا در این صورت طول کل خارج قسمت بیش از مقداری خواهد بود که بتوان آن را به واحدهای حافظه‌ای که طول استاندارد دارند، یعنی ثباتها، منتقل کرد. در سخت‌افزار یا نرم‌افزار کامپیوتر و یا ترکیبی از هر دو باید امکاناتی را جهت آشکارسازی این وضعیت فراهم کرد.

اگر تعداد بیت‌های مقسوم دو برابر مقسوم علیه باشد، حالت سرریز را می‌توان به طریق زیر بیان کرد: حالت سرریز در تقسیم هنگامی اتفاق می‌افتد که بیت‌های نیمه پر ارزشتر مقسوم، عددی بزرگتر یا مساوی مقسوم علیه را تشکیل دهند. مسأله دیگری که باید در تقسیم در نظر داشت این است که از تقسیم بر صفر باید جلوگیری شود. مدار چک کننده وضعیت سرریز در تقسیم، مراقبت از این وضعیت را نیز می‌تواند بر عهده گیرد. این بدان علت است که اگر مقسوم علیه صفر باشد مقسوم قطعاً از مقسوم علیه بزرگتر و یا مساوی آن است. حالت سرریز معمولاً با 1 شدن فلیپ فلاپ خاصی مشخص می‌گردد و آن را فلیپ فلاپ سرریز خوانده و با DVF نشان می‌دهیم. در قبال وقوع سرریز در عملیات تقسیم، به طرق مختلف می‌توان عکس العمل نشان داد. در بعضی کامپیوترها مسؤولیت با برنامه‌نویس است تا پس از هر دستورالعمل تقسیم، DVF را چک کند. در این صورت می‌توان به زیر روالی انشعاب کرده و اقدامی اصلاحی همچون تغییر مقیاس داده‌ها برای جلوگیری از سرریز انجام داد. در برخی کامپیوترهای قدیمی‌تر، وقوع سرریز در تقسیم سبب توقف کامپیوتر می‌شد و این حالت، توقف تقسیم نامیده می‌شد. امروزه متوقف کردن کار کامپیوتر توصیه نمی‌شود زیرا اتلاف کننده وقت است. روشی که در اکثر کامپیوترها انجام می‌شود تولید درخواست وقفه به هنگام 1 شدن DVF است. این وقفه موجب می‌شود تا کامپیوتر اجرای برنامه جاری را به تعویق انداخته و به روال سرویس‌دهی برای اقدام اصلاحی مورد تقاضا انشعاب نماید. معمول‌ترین اقدام اصلاحی خارج شدن از برنامه و چاپ پیغام خطائی است که دلیل عدم امکان اتمام برنامه را شرح می‌دهد. از این پس، وظیفه کاربری که برنامه را نوشته این است که مقیاس داده‌ها را تغییر دهد و یا هر اقدام اصلاحی دیگری را به عمل آورد. بهترین راه جلوگیری از سرریز تقسیم استفاده از داده‌های ممیز شناور است. در بخش مربوط به محاسبات ممیز

شناور خواهیم دید که سرریز تقسیم به سادگی به هنگام استفاده از اعداد ممیز شناور قابل پیش‌گیری است.

۴-۶-۳- **الگوریتم** سخت‌افزاری عملیات تقسیم

الگوریتم سخت‌افزاری عملیات تقسیم در فلوچارت شکل ۲۳ نشان داده شده است. مقسوم در A و Q، و مقسوم‌علیه درB قرار دارد. در ابتدا علامت نتیجه به عنوان قسمتی از خارج قسمت به Qs منتقل می‌شود. برای مشخص کردن تعداد بیت‌های خارج قسمت، عدد ثابتی (برای مثال بالایی عدد ۵) در توالی شمارSC قرار داده می‌شود.



شکل ۲۳: فلوچارت عمل تقسیم

فرض می‌کنیم که عملوندها از واحد حافظه n بیتی به ثبات‌ها منتقل شوند. چون عملوند همراه با علامتش ذخیره می‌شود، یک بیت از کلمه حافظه بوسیله علامت و n-1 بیت به وسیله اندازه آن اشغال می‌گردد. حالت سرریز تقسیم با تفریق مقسوم‌علیه درB از نیمی از بیت‌های ذخیره شده در مقسوم یعنی A تست می‌شود. اگر $A \geq B$ باشد فلیپ فلاپ سرریز تقسیم (DVF) برابر 1 شده و عمل به طور ناقص متوقف می‌شود. تقسیم اندازه‌های دو عدد با جابه‌جا کردن مقسوم موجود در AQ به چپ شروع می-شود که در این جابجایی بیت پر ارزشتر به E منتقل می‌شود اگر بیت انتقال یافته به E برابر 1 باشد، درمی‌یابیم که EA>B است زیرا EA از رقم 1 و به دنبال آن n-1 بیت تشکیل شده است، در حالیکه B فقط n-1 بیت است.

اگر عمل شیفت به چپ مقدار 0 را وارد E کند، قسمت بالایی مقسوم با متمم 2 مقسوم‌علیه جمع می‌شود (عمل تفریق A-B) و مقدار رقم نقلی به E انتقال می‌یابد. اگر E=1 شود، $A \geq B$ است، بنابراین بیت 1 به Q_n منتقل خواهد شد. اگر E=0 باشد، A<B است و عدد اولیه مقسوم، با جمع B با A، دوباره بازیابی می‌شود. در این حالت مقدار 0 را در Q_n قرار می‌دهیم.

این روند مجدداً تکرار می‌گردد. پس از n-1 تکرار، اندازه خارج‌قسمت در ثبات Q و باقیمانده در ثبات A خواهد بود. علامت خارج‌قسمت در Q_s و علامت باقیمانده نیز که همان علامت مقسوم است در A_s قرار دارد.

۴-۶-۴- عملیات تقسیم در پردازنده MIPS

پردازنده MIPS برای عملیات تقسیم نیز از رجیسترهای Hi و Lo استفاده می‌کند. بعد از انجام عملیات تقسیم، مقدار باقیمانده تقسیم داخل رجیستر Hi و مقدار خارج قسمت تقسیم داخل رجیستر Lo ریخته می‌شود. برای خواندن محتوای این رجیسترها و انتقال آنها به داخل رجیسترهای عمومی پردازنده از دستورهای mfhi و mflo استفاده می‌شود. MIPS برای تقسیم اعداد صحیح علامت‌دار و بدون علامت، به ترتیب از دستورهای div[1] و divu[2] استفاده می‌کند.

---

[1] - Divide
[2] Divide unsigned

۴-۷- انجام عملیات ضرب و تقسیم به کمک دستورهای شیفت

با توجه به مطالبی که در ارتباط با الگوریتم‌های سخت‌افزاری و پیاده سازی عملیات ضرب و تقسیم بیان شد، می‌توان این‌طور نتیجه گرفت که دستورات ضرب و تقسیم دستوراتی هستند که دارای CPI بالاتری نسبت به دستوراتی مثل جمع و تفریق می‌باشند. بنابراین در یک برنامه هر چقدر تعداد دستورات ضرب و تقسیم بیشتر باشد به احتمال زیاد زمان اجرای برنامه بیشتر خواهد بود و برنامه دیرتر جواب خواهد داد. در مقام مقایسه زمان اجرای دستورات شیفت نیز از زمان اجرای دستورات ضرب و تقسیم کمتر است.

با توجه به مطالب فوق کامپایلرها تا حد امکان سعی می‌کنند دستورات ضرب و تقسیم کمتری تولید نمایند. در برخی موارد یکی از عملوندهای ضرب یا تقسیم، یک عدد ثابت است. در چنین مواردی کامپایلر سعی می‌کند که به جای دستور ضرب یا تقسیم از ترکیبی از عملیات شیفت، جمع و تفریق استفاده نماید. به مثال زیر توجه کنید.

مثال: دستورات ماشین را برای عملیات زیر طوری بنویسید که زمان اجرای کد کمترین باشد.

$$v0 = t0 * 5$$

جواب: می‌توانیم ۵ را به صورت 5=4+1 بنویسیم و در نتیجه داریم: v0 = t0*(4+1)=t0*4+t0

از آنجا که هر شیفت به چپ معادل یک ضرب در ۲ است می‌توان برای عملیات t0*4 ، t0 را به اندازه دو واحد به چپ شیفت داد. پس می‌توان کد اسمبلی را به صورت زیر نوشت:

```
sll $v0, $t0, 2  // v0 = t0 * 4
add $v0, $v0, $t0  // v0 = t0 *4 + t0
```

اگر فرض کنیم که دستورات add و sll در یک کلاک و دستور ضرب در ۳۲ کلاک انجام بگیرد، در این صورت کدی که نوشته شده، در ۲ کلاک اجرا خواهد شد. اما اگر از دستور ضرب در نوشتن این کد استفاده کنیم، زمان اجرا مساوی ۳۲ کلاک خواهد شد.

پس در نتیجه، می‌توانیم عددهای ثابت را به جمع و تفریقی از توانهای ۲ تبدیل کنیم و هر کدام از عملیاتی را که یکی از عملوندهای آن توانی از ۲ است را با عملیات شیفت پیاده سازی کنیم. با انجام

این کار می‌توان عملیات ضرب و تقسیم مورد نظر را با ترکیبی از دستورات add، sub و شیفت پیاده سازی نمود و بدین طریق بهبود قابل ملاحظه‌ای در زمان اجرا به وجود آورد.

۴-۸- عملیات حسابی ممیز شناور

بسیاری از زبانهای برنامه‌نویسی امکاناتی برای مشخص کردن عددهای ممیز شناور دارند. متداول‌ترین روش، مشخص کردن آنها با عبارت real (حقیقی) در برابر اعداد ممیز ثابت است که با عبارت Integer (صحیح) مشخص می گردند. هر کامپیوتری که دارای کامپایلر برای اینگونه زبانهای سطح بالا باشد باید امکاناتی برای انجام عمل‌های حسابی ممیز شناور داشته باشد. این عملیات غالباً در سخت‌افزار کامپیوتر پیاده‌سازی می‌شوند. اگر سخت‌افزاری برای این اعمال وجود نداشته باشد، کامپایلر باید طوری طراحی شود که دارای مجموعه‌ای از زیر روال‌های نرم‌افزاری ممیز شناور باشد یعنی این که دستورات ممیز شناور در این زیر روال‌ها با دستورات دیگر پیاده سازی گردد. هر چند روش سخت افزاری گرانتر است، ولی کارآیی آن به قدری از روش نرم افزاری بیشتر است که در اکثر کامپیوترها، سخت‌افزار ممیز شناور کار گذاشته می‌شود و فقط در کامپیوترهای خیلی کوچک از آن صرف‌نظر می‌شود.

۴-۸-۱- مفاهیم اساسی

یک عدد ممیز شناور در ثباتهای کامپیوتر از دو بخش تشکیل می‌شود: مانتیس m و نمای e. این دو بخش نماینده عددی است که از ضرب m در r به نمای e بدست می‌آید. یعنی:

$$m \times r^e$$

مانتیس ممکن است که عددی کسری یا صحیح باشد. اطلاعات محل ممیز و مقدار پایه r در ثباتها وارد نمی‌شوند و در محاسبات، این اطلاعات مقدار پیش فرضی دارند. به طور مثال نمایش کسری و پایه 10 را در نظر بگیرید. عدد دهدهی 537.25 در یک ثبات با m=53725 و e=3 نمایش داده می‌شود و چنین تفسیر می گردد که نماینده عدد ممیز شناور زیر است:

$$0.53725 \times 10^3$$

اگر با ارزشترین رقم مانتیس یک عدد ممیز شناور، غیر صفر باشد آن را نرمالیزه می‌کنیم. در نتیجه مانتیس دارای بیشترین تعداد ممکن رقم‌های معنی‌دار خواهد بود. صفر را نمی‌توان نرمالیزه کرد زیرا رقم غیر صفر ندارد. در نمایش ممیز شناور، صفر را با مقدار تمام 0 در مانتیس و نما نمایش می-دهیم.

نمایش ممیز شناور محدوده اعدادی را که می‌توان در یک ثبات جای داد مشخص می‌کند. کامپیوتری با کلمه‌های 48 بیتی را در نظر بگیرید. چون یک بیت برای علامت رزرو شده‌است محدوده اعداد صحیح ممیز ثابت $\pm\left(2^{47}-1\right)$ خواهد بود که تقریباً $\pm 10^{14}$ می‌باشد. از این48 بیت می‌توان برای نمایش یک عدد میز شناور با 36 بیت برای مانتیس و 12 بیت برای نما استفاده کرد. با فرض نمایش کسری برای مانتیس و با انتساب دو بیت برای علامت‌ها، محدوده اعدادی که می‌توان جای داد برابر است با:

$$\pm\left(1-2^{-35}\right)\times 2^{2047}$$

این عدد از کسری که حاوی 35 رقم 1، نمای 11 بیتی (منهای علامت آن) و اینکه $2047 = 2^{11} - 1$ می‌باشد، بدست می‌آید. بزرگترین عددی که در ۴۸ بیت می‌توان جای داد 10^{615} است، که عدد بسیار بزرگی است. مانتیس می‌تواند 35 بیتی باشد (جدا از علامت) و اگر عدد صحیح تلقی شود می‌تواند تا $(2^{35} - 1)$ را ذخیره نماید. این تقریباً برابر با 10^{10} می‌باشد، که معادل یک عدد دهدهی10 رقمی است. کامپیوترهایی که دارای طول کلمه‌های کوتاهتری هستند برای نمایش اعداد ممیز شناور از دو یا چند کلمه استفاده می‌نمایند. یک میکروکامپیوتر 8 بیتی ممکن است از چهار کلمه برای نمایش یک عدد ممیز شناور استفاده کند که یکی از کلمات 8 بیتی برای نما و سه کلمه (24 بیت) دیگر برای مانتیس در نظر گرفته شوند.

اعمال حسابی با اعداد ممیز شناور بسیار پیچیده‌تر از اعداد ممیز ثابت است واجرای آنها زمان بیشتری می‌برد و سخت‌افزار پیچیده‌تری نیاز دارد. جمع یا تفریق دو عدد مستلزم معین کردن محل ممیز است زیرا قبل از جمع یا تفریق مانتیس‌ها، باید بخش نمای دو عدد با هم مساوی شود. هم ردیف کردن با جابه‌جایی یکی از مانتیس‌ها و تنظیم نمای آن تا جایی که با دیگری برابر شود انجام می‌شود. به طور مثال جمع اعداد ممیز شناور زیر را در نظر بگیرید.

$$0.5372400 \times 10^{2}$$

$$+\ 0.1580000 \times 10^{-1}$$

لازم است که دو نما قبل از جمع مانتیس‌ها با هم مساوی شوند. می‌توانیم عدد اول را سه مکان به چپ، یا عدد دوم را سه مکان به راست شیفت دهیم. وقتی که مانتیس‌ها در ثبات‌ها ذخیره شوند، شیفت به چپ سبب از دست دادن ارقام با ارزشتر می‌شود. روش دوم ترجیح دارد چون فقط میزان دقت را کاهش می‌دهد در حالی که روش اول ممکن است موجب بروز خطا شود. شیوه هم ردیف کردن معمولاً به این ترتیب است که مانتیسی را که نمای کوچکتر دارد به اندازه اختلاف بین نماها به راست شیفت می‌دهیم. پس از این عمل، مانتیس‌ها می‌توانند با هم جمع شوند.

$$0.5372400 \times 10^{2}$$

$$+\ 0.0001580 \times 10^{2}$$

$$0.5373980 \times 10^{2}$$

هنگامی که دو مانتیس نرمالیزه شده با هم جمع شوند حاصل‌جمع ممکن است رقم سرریز داشته باشد. سرریز را به سادگی می‌توان با شیفت حاصل‌جمع به راست و افزایش نما تصحیح کرد. در تفریق مانند مثال زیر نتیجه ممکن است دارای صفرهایی در مکان‌های بالا رتبه باشد.

$$0.56780 \times 10^{5}$$

$$-\ 0.56430 \times 10^{5}$$

$$0.00350 \times 10^{5}$$

وقتی که عدد ممیز شناوری دارای 0 در مکان‌های با ارزشتر باشد گوئیم فروریز[1] دارد. برای نرمالیزه کردن اعداد فروریز دار، لازم است تا مانتیس به چپ شیفت داده شود و از نما یک واحد کسر گردد تا رقم غیر صفر در اولین مکان ظاهر شود. در مثال فوق، لازم است تا مانتیس را دوبار به چپ شیفت دهیم تا 0.35000×10^3 حاصل گردد. در بسیاری از کامپیوترها، نرمالیزه کردن پس از هر عمل صورت می‌گیرد تا از نرمالیزه شدن نتایج اطمینان حاصل شود.

ضرب و تقسیم ممیز شناور، نیازی به هم ردیف کردن مانتیس‌ها ندارند. حاصلضرب با ضرب دو مانتیس و جمع نماها شکل می‌گیرد. عمل تقسیم از تقسیم دو مانتیس و تفریق نماها بدست می‌آید. نکته‌ای که در این قسمت باید به آن تأکید کنیم این است که در محاسبات ممیز شناور اعمال اجرا شده با مانتیس‌ها مشابه اعداد ممیز ثابت است، بنابراین هر دو نوع می‌توانند از ثباتها و مدارهای مشترکی استفاده کنند. این در حالی است که عملیات مربوط به نما در محاسبات ممیز شناور عبارتند از: مقایسه و افزایش (برای هم ردیفی مانتیس‌ها)، جمع و تفریق (برای ضرب و تقسیم)، و کاهش (برای نرمالیزه کردن نتایج).

نما را می‌توان به یکی از سه روش ممکن نمایش داد: مقدار علامت‌دار، متمم 2 علامت‌دار و متمم 1 علامت‌دار.

چهارمین روشی که در بسیاری از کامپیوترها بکار گرفته می‌شود نمای خورانده شده یا بایاس شده است. در این نمایش، بیت علامت به عنوان یک عضو جدا در نظر گرفته نمی‌شود. بایاس عدد مثبتی است که به هنگام تشکیل عدد ممیز شناور به هر نما اضافه می‌شود. در نتیجه تمام نماها در درون مثبت خواهند بود. مثال زیر می‌تواند این نوع نمایش را روشن سازد. فرض کنید نماها از 50- تا 49+ متغیر باشند. در این صورت ثبات نما حاوی عدد e+50 خواهد بود که e نمای واقعی است و 50 مقدار بایاس می‌باشد. لذا نماها در ثبات به صورت اعداد مثبتی از 00 تا 99 نمایش داده خواهند شد. نماهای مثبت در ثبات‌ها محدوده اعداد 50 تا 99 و نماهای منفی محدوده 00 تا 49 خواهند داشت.

[1] - Underflow

مزیت استفاده از نماهای بایاس شده این است که فقط اعداد مثبت را شامل می‌شوند. در نتیجه مقایسه نسبت اندازه‌های آنها ساده‌تر بوده و لازم نیست به علامت آنها توجه کنیم. لذا هنگام هم ردیف کردن مانتیس‌ها می‌توان از مقایسه‌گر مقدار (اندازه) برای مقایسه اندازه‌های آنها استفاده کرد. مزیت دیگر آنها این است که کوچکترین نمای ممکن، تمام 0 است.

در مثال‌های بالا، از عددهای دهدهی برای نمایش دادن برخی از مفاهیمی که هنگام کار با اعداد ممیز شناور باید آنها را دانست استفاده کردیم. واضح است که مفاهیم مشابهی در مورد اعداد دودویی نیز صادق است.

۴-۸-۲- جمع و تفریق

الگوریتم جمع و تفریق ممیز شناور می‌تواند به چهار بخش متوالی تقسیم گردد:

۱. تست صفر بودن عملوندها
۲. هم ردیف کردن مانتیس‌ها
۳. جمع یا تفریق مانتیس‌ها
۴. نرمالیزه کردن نتیجه

عدد ممیز شناوری که صفر باشد قابل نرمالیزه شدن نیست. اگر این عدد طی محاسبات مورد استفاده قرار گیرد، نتیجه نیز ممکن است صفر شود. به جای تست صفر حین نرمالیزه کردن، در ابتدای کار صفر را چك می‌کنیم و در صورت لزوم پردازش را خاتمه می‌دهیم. هم ردیف کردن مانتیس‌ها باید قبل از اجرای اعمال جمع و تفریق باشد. پس از جمع یا تفریق مانتیس‌ها، نتیجه ممکن است غیر نرمالیزه باشد. با به کارگیری روش نرمالیزه کردن، نتیجه قبل از انتقال به حافظه نرمالیزه می‌شود.

۴-۸-۳- ضرب

ضرب دو عدد ممیز شناور با ضرب مانتیس‌ها و جمع نماها انجام می‌شود. در این عمل مقایسه نماها و هم ردیف کردن مانتیس‌ها قبل از عملیات لزومی ندارد. ضرب مانتیس‌ها مشابه ضرب اعداد

ممیز ثابت است و حاصلضرب با دقتی دو برابر حاصل می‌شود. پاسخ حاصل با دقتی مضاعف[1] در ممیز ثابت برای افزایش دقت حاصلضرب به کار می‌رود. در نمایش ممیز شناور، محدوده دقت معمولی[2] همراه با نما به اندازه کافی دقت دارد و بنابراین اعداد به صورت غیر مضاعف یا معمولی نگهداری می-شوند. لذا نیمه با ارزشتر حاصلضرب مانتیس‌ها برداشته شده و در کنار نمای حاصل شده حاصلضربی با دقت معمولی برای ممیز شناور به وجود می‌آید.

الگوریتم ضرب به چهار بخش زیر تقسیم می‌شود:

۱. چک کردن صفر

۲. جمع کردن نماها

۳. ضرب مانتیس‌ها

۴. نرمالیزه کردن حاصلضرب

مراحل ۲ و ۳ می‌توانند به طور همزمان انجام شوند مشروط بر اینکه برای مانتیس‌ها و نماها، جمع‌کننده‌های جداگانه‌ای وجود داشته باشند.

در عملیات ضرب ممیز شناور سرریز نمی‌تواند رخ دهد، لذا نیازی به چک کردن آن نیست. فقط باید بعد از عملیات، نرمال سازی صورت گیرد.

۴-۸-۴- تقسیم

در تقسیم ممیز شناور نماها تفریق و مانتیس‌ها تقسیم می‌شوند. تقسیم مانتیس مشابه ممیز ثابت است جز اینکه مقسوم مانتیسی با دقت معمولی دارد. به خاطر بسپارید که مانتیس مقسوم کسری است و نه عدد صحیح.

تست سرریز تقسیم همانند نمایش ممیز ثابت است. با این وجود سرریز تقسیم در ممیز شناور مشکلی ایجاد نمی‌کند. اگر مقسوم بزرگتر یا مساوی مقسوم‌علیه باشد، کسر مقسوم به راست شیفت

---

[1] - Double precision

[2] - Single precision

داده می‌شود و نمای آن یک واحد افزایش می‌یابد. برای عملوندهای نرمالیزه شده این عمل برای تضمین عدم وقوع سر ریز تقسیم کافی است. عمل فوق هم ردیف کردن مقسوم خوانده می‌شود.

تقسیم دو عدد ممیز شناور نرمالیزه شده همواره خارج‌قسمت نرمالیزه شده‌ای را به دست می‌دهد به شرطی که هم ردیف کردن مقسوم قبل از تقسیم انجام شده باشد. بنابراین بر خلاف سایر اعمال، خارج‌قسمت حاصل از تقسیم نیازی به نرمالیزه شدن ندارد.

الگوریتم تقسیم می‌تواند به پنج بخش زیر تقسیم شود:

۱. چک کردن برای وجود صفر

۲. مقدار دهی اولیه برای ثباتها و تعیین علامت

۳. هم ردیف کردن مقسوم

۴. تفریق نمادها

۵. تقسیم مانتیس‌ها

توجه: در عمل تقسیم ممیز شناور به هنگام تفریق نمای مقسوم‌علیه از نمای مقسوم، چون هر دو نما در ابتدا بایاس شده هستند، با عمل تفریق تفاضل بایاس نشده آنها بدست می‌آید. پس باید بعد از این تفریق دوباره عدد بایاس به نتیجه اضافه شود.

فصل ۵

طراحی یک پردازنده ساده

۵-۱- مقدمه

در فصل دوم دیدم که کارایی یک پردازنده به سه عامل کلیدی وابسته بود: تعداد دستورات، پریود کلاک و تعداد کلاک متوسط لازم برای اجرای هر دستور یا همان CPI. در فصل سوم و چهارم همان طور که دیدیم کامپایلر و معماری مجموعه دستورات (ISA)، تعداد دستورات یک برنامه را تعیین می‌کنند. اما پریود کلاک و تعداد کلاک لازم برای اجرای هر دستور توسط سخت‌افزار تعیین می‌شود. در این فصل یک پردازنده ساده برای مجموعه دستورات پردازنده MIPS طراحی خواهیم نمود. پردازنده‌ای که ما طراحی می‌کنیم، تعداد محدودی از دستورات را پیاده سازی خواهد نمود و شامل همهٔ دستورات پردازنده MIPS نخواهد بود. به عبارتی پردازنده طراحی شده یک زیر مجموعه[1] از دستورات پردازنده MIPS را پشتیبانی خواهد کرد. دستوراتی که پشتیبانی خواهد شد به شرح زیر است:

- دستورات مراجعه به حافظه مشتمل بر lw و sw
- دستورات محاسباتی و منطقی مشتمل بر add، sub، and، or و slt
- دستورات پرش شرطی و غیر شرطی مشستمل بر beq و j یا همان jump

در زیر مجموعه انتخاب شده سایر دستورات اعداد صحیح مانند دستورات ضرب و تقسیم و یا دستورات ممیز شناور وجود ندارند و پردازنده‌ای که در این فصل طراحی می‌کنیم آنها را پشتیبانی نخواهد نمود. با وجود این، اصول اولیه و کلیدی طراحی یک پردازنده در این فصل توضیح داده خواهد شد و می‌توان دستورات دیگر را به روش مشابه، به این مجموعه اضافه و پیاده سازی نمود. به هنگام پیاده سازی ما این شانس را خواهیم داشت که ببینیم چگونه معماری مجموعه دستورات جنبه-های مختلف طراحی سخت‌افزار را تحت تأثیر قرار می‌دهد و همچنین انتخاب استراتژی‌های مختلف پیاده‌سازی چگونه پریود کلاک و CPI یک ماشین را تعیین می‌کند. بسیاری از اصول کلیدی طراحی که در فصل ۳ مطرح شدند همانند اصول «موارد پر استفاده را سریعتر کن» و «منظم‌تر بودن ساده‌تر می-کند» را در این فصل به هنگام طراحی سخت‌افزار لمس خواهیم نمود. علاوه بر این مباحثی که در این فصل مطرح خواهد شد همانند ایده‌هایی است که امروزه در طراحی انواع مختلف پردازنده استفاده می‌شود.

[1] - Sub set

۵-۱-۱- مروری بر پیاده‌سازی

در فصل‌های ۳ و۴ ما دستورات مختلف پردازنده MIPS، مشتمل بر دستورات محاسباتی و منطقی، دستورات مراجعه به حافظه و دستورات پرش را بررسی کردیم. برای پیاده‌سازی سخت‌افزاری این دستورات، اکثر کارهایی که لازم است انجام گیرد مشابه هم بوده و مستقل ازکلاس واقعی دستورات می‌باشد. برای همهٔ دستورات دو مرحله اول یکسان است:

۱. فرستادن شمارنده برنامه (PC) به حافظه دستورات که برنامه را نگهداری می‌کند و واکشی[1] (خواندن) دستور از حافظه.

۲. خواندن محتوای یک یا دو رجیستر با استفاده از فیلدهای موجود در دستور. به کمک این فیلدها می‌توانیم عمل انتخاب بین رجیسترها را انجام دهیم و تصمیم بگیریم که کدام یک از آنها باید خوانده شوند. برای دستور lw ما نیاز داریم که فقط یک رجیستر را بخوانیم ولی برای اکثر دستورات دیگر دو رجیستر لازم است.

بعد از این دو مرحله، کاری که لازم است انجام شود تا اینکه دستور اجرا شود، به نوع دستور و به کلاس آن بستگی دارد. خوشبختانه برای هر کدام از سه کلاس دستورات (مراجعه به حافظه، دستورات محاسباتی و منطقی و پرش‌ها)، کاری که لازم است انجام شود یکی است و بستگی به دستورات داخل آن کلاس ندارد.

حتی با وجود تفاوتهایی که بین کلاس‌های مختلف دستورات وجود دارد، برخی تشابهات هم بین آنها وجود دارد. به طور مثال همهٔ کلاس‌های دستورات بعد از خواندن محتوای رجیسترها، از واحد ALU استفاده می‌کنند. دستورات مراجعه به حافظه از ALU برای محاسبه آدرس حافظه، دستورات محاسباتی و منطقی برای انجام عملیات و دستورات پرش برای انجام عمل مقایسه استفاده می‌کنند. همان طور که می‌بینیم، سادگی و منظم بودن معماری مجموعه دستورات، باعث می‌شود تعداد زیادی دستور همانند هم اجرا شوند و این باعث می‌شود طراحی سخت‌افزار نیز ساده‌تر شود.

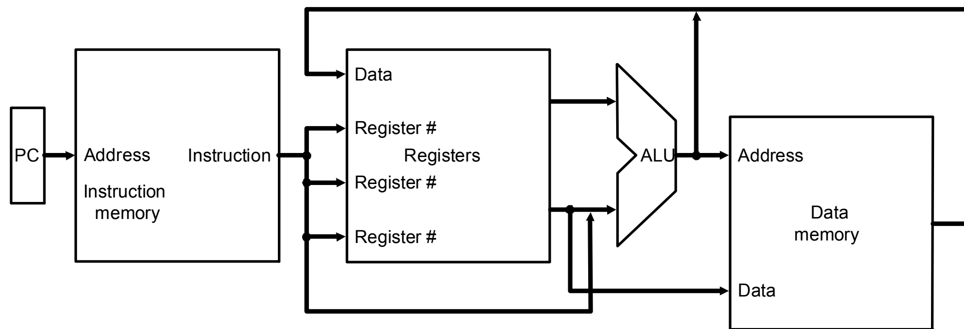
بعد از اینکه از ALU استفاده کردیم، کارهایی که لازم است انجام گیرد تا دستور خاتمه پیدا کند، بین کلاس‌های مختلف متفاوت است. یک دستور مراجعه به حافظه نیاز به دسترسی به حافظه جهت انجام عمل ذخیره سازی برای دستورsw و یا خواندن برای دستورlw دارد. یک دستور محاسباتی و منطقی باید نتیجه حاصل شده از ALU را داخل یکی از رجیسترها بنویسد. در نهایت برای یک دستور پرش، ممکن است لازم شود که آدرس دستور بعدی را بر اساس نتیجه مقایسه عوض کنیم.

---

[1] - Fetch

در شکل ۱ یک مرور سطح بالا از پیاده‌سازی سخت‌افزار پردازنده MIPS نشان داده شده است. در ادامه این فصل ما این شکل را با جزئیات بیشتری توضیح خواهیم داد. ما به این شکل واحدهای عملیاتی بیشتری اضافه خواهیم نمود و تعداد خطوط ارتباطی بین واحدهای مختلف را نیز افزایش خواهیم داد.

همچنین یک واحد کنترل[1] که کار انجام شده برای کلاسهای مختلف دستورات را کنترل می‌کند، نیز به این شکل اضافه خواهد شد. قبل از اینکه ما شروع کنیم و یک سخت‌افزار کامل طراحی کنیم، نیار به یک سری مفاهیم مطرح شده در مدار منطقی خواهیم داشت به همین دلیل در ابتدا مروری خواهیم داشت بر این مفاهیم.



شکل ۱: یک مرور کلی بر پیاده سازی سخت افزاری زیر مجموعه‌ای از دستورات پردازنده MIPS که در آن فقط ماجولهای عملیاتی اصلی و ارتباطات اصلی نشان داده شده است

۵-۱-۲- مروری بر مدارهای منطقی و مفهوم کلاک

مدارهای منطقی به دو دسته کلی مدارهای ترکیبی[2] و ترتیبی[3] تقسیم‌بندی می‌شوند. مدارهای ترکیبی مدارهایی هستند که در آنها عناصر حافظه وجود ندارد و خروجی‌های مدار در هر لحظه فقط به ورودیهای مدار وابسته‌اند. به طور مثال ALU یک مدار ترکیبی است. بر عکس مدارهای ترتیبی مدارهایی هستند که دارای عناصر حافظه‌اند و در آنها خروجی مدار در هر لحظه علاوه بر ورودیهای مدار به حالتهای داخلی یا همان مقادیر ذخیره شده در داخل حافظه‌ها نیز بستگی دارد. از جمله مدارهای ترتیبی می‌توانیم به رجیسترها و حافظه‌ها اشاره کنیم.

هر عنصری از مدار ترتیبی که دارای حافظه باشد، یک عنصر حالت[4] نامیده می‌شود. هر عنصر حالت حداقل دارای دو ورودی و یک خروجی می‌باشد. یکی از ورودیها داده‌ای است که داخل عنصر حالت ذخیره خواهد شد و ورودی دیگر کلاک می‌باشد که مشخص می‌کند عمل ذخیره‌سازی چه

[1] - Control unit
[2] - Combinational
[3] - Sequential
[4] - State element

موقع انجام خواهد شد. خروجی عنصر حالت هم داده‌ای است که در کلاک قبلی در داخل آن ذخیره شده است. به طور مثال فلیپ فلاپ نوع D یکی از ساده‌ترین عناصر حالت می‌باشد که دارای دو ورودیD وکلاک و یک خروجیQ می‌باشد. علاوه بر فلیپ فلاپ نوعD، سخت‌افزار پردازنده MIPS که در این فصل پیاده‌سازی خواهیم کرد، دارای دو عنصر حالت دیگر نیز می‌باشد: حافظه‌ها و رجیسترها. ماژول مربوط به حافظه‌ها و رجیسترها به همراه چند ماژول دیگر در شکل ۱ نشان داده شده است.

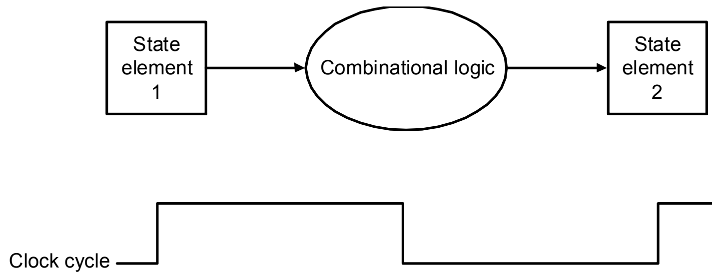
عملیات نوشتن در داخل عنصر حالت با توجه به کلاک انجام خواهد شد اما عملیات خواندن از عنصر حالت در هر زمانی امکان‌پذیر است.

۵-۱-۳- متودولوژی کلاک

متودولوژی کلاک[1] تعیین می‌کند که سیگنالها چه موقع می‌توانند خوانده شوند و چه موقع می‌توانند نوشته شوند. مشخص کردن زمانبندی خواندن‌ها و نوشتن‌ها امر مهمی است، به دلیل اینکه وقتی یک سیگنال در یک زمان هم نوشته شود و هم خوانده شود در این صورت مقداری که خوانده می‌شود، می‌تواند مقدار قبلی نوشته شده در زمانهای قبل، مقدار فعلی که نوشته می‌شود و یا ترکیبی از این دو باشد. بدیهی است که قابل پیش بینی نبودن در طراحی کامپیوتر پذیرفتنی نیست. یک متودولوژی کلاک برای از بین بردن این عدم قطعیت طراحی می‌شود. برای سادگی، ما یک متودولوژی کلاک حساس به لبه[2] را به کار خواهیم کرد. یک متودولوژی کلاک حساس به لبه به این معنی است که مقادیر ذخیره شده در داخل ماشین فقط در لبه‌های کلاک می‌توانند تغییرکنند. بنابراین عناصر حالت مقدارشان فقط در لبه‌های کلاک می‌توانند تغییرکند و بروز شود[3]. به دلیل اینکه فقط عناصر حالت می‌توانند مقدار داده[4] را داخل خود نگهداری کنند، بنابراین مدارهای ترکیبی باید ورودی‌های خود را از عناصر حالت دریافت کنند و خروجی را به عناصر حالت تحویل دهند. مقادیر ورودی مدارهای ترکیبی همان مقادیری هستند که در کلاک قبلی داخل عناصر حالت ذخیره شده‌اند و خروجی آنها مقادیری است که باید در کلاک بعدی داخل عناصر حالت ذخیره شوند. شکل ۲ دو عنصر حالت را نشان می‌دهد که یک مدار ترکیبی را در میان گرفته‌اند و با یک کلاک واحد کار می‌کنند. همهٔ سیگنالها باید از عنصر حالت شماره ۱ شروع شده، از مدار ترکیبی رد شده و به طرف عنصر حالت شماره ۲ منتشر شوند. مدت زمان لازم برای انتشار نباید از یک پریود کلاک بیشتر شود. بنابراین تأخیر انتشار از عنصر حالت ۱ تا عنصر حالت ۲ پریود کلاک را تعیین می‌کند.

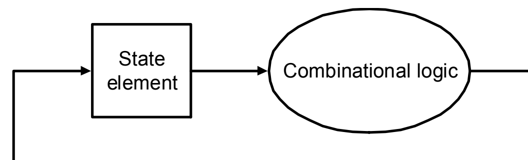
---

[1] - Clocking methodology
[2] - Edge triggered
[3] - Update
[4] - Data value



شکل ۲: مدارهای ترکیبی، عناصر حالت و کلاک، ارتباط نزدیکی به هم دارند

برای سادگی، ما سیگنال کنترل عملیات نوشتن (write) را به هنگامی که عنصر حالت در هر لبهٔ کلاک نوشته می‌شود را نشان نمی‌دهیم. اما اگر یک عنصر حالت در هر لبهٔ کلاک مقدارش بروز نمی‌شود، در این صورت یک خط کنترلی نوشتن مورد نیاز خواهد بود. سیگنالهای کلاک و write ورودی هستند و عنصر حالت فقط زمانی تغییر پیدا می‌کند که به هنگام اتفاق افتادن لبهٔ کلاک، ورودی write فعال باشد. متودولوژی حساس به لبه بودن این امکان را در اختیار ما قرار می‌دهد که محتوای یک رجیستر را بخوانیم و مقدار خوانده شده را از یک مدار ترکیبی رد کنیم و نتیجه بدست آمده را در لبهٔ کلاک بعدی در همان رجیستر بنویسیم. این امر در شکل ۳ نشان داده شده است. فرض اینکه عملیات نوشتن در لبهٔ مثبت کلاک اتفاق بیافتد یا در لبهٔ منفی آن، مهم نیست، به دلیل اینکه ورودی‌های مدار ترکیبی که خروجی عناصر حالت هستند تا لبه انتخاب شده کلاک نمی‌توانند تغییر کنند. با استفاده از متودولوژی زمان‌بندی حساس به لبه، هیچ نوع مسیر feedback و یا Loop ای در داخل یک سیکل کلاک وجود نخواهد داشت و مدار به درستی کار خواهد کرد.



شکل ۳: یک متودولوژی حساس به لبه این امکان را در اختیار قرار می‌دهد که بتوانیم در یک سیکل کلاک عملیات خواندن و نوشتن در داخل یک عنصر حالت را به درستی انجام دهیم

تقریباً همهٔ عناصر حالت و مدارهای ترکیبی در پردازنده MIPS دارای ورودی‌ها و خروجی‌های ۳۲ بیتی خواهند بود، به دلیل اینکه اکثر داده‌هایی که توسط پردازنده MIPS دستکاری می‌شوند دارای طول ۳۲ بیت هستند. در جاهایی که هرکدام از این ورودی‌ها و یا خروجی‌ها دارای طولی غیر از ۳۲ بیت باشند، مشخصاً اعلام خواهیم کرد. در شکل‌هایی که رسم خواهد شد اگر طول سیگنال بیشتر از ۱ بیت باشد در این صورت خط مربوط به آن سیگنال ضخیم‌تر رسم خواهد شد که نشان دهندهٔ طول بیش از یک است. سیگنالی که طول آن بیشتر از ۱ بیت است اصطلاحاً باس[1] یا گذرگاه نامیده می‌شود.

[1] - Bus

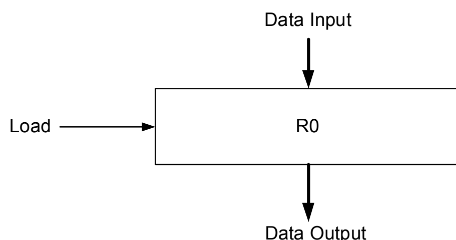
بعضی مواقع ما چند تا باس را با هم ترکیب نموده و یک باس ضخیم‌تر ایجاد خواهیم کرد. به طور مثال یک باس ۳۲ بیتی را از ترکیب دو باس ۱۶ بیتی ایجاد خواهیم کرد. در این صورت برچسب‌هایی که روی خطوط قرار می‌دهیم به وضوح نشان خواهد داد که دو تا باس به هم متصل شده و یک باس بزرگتر را ایجاد کرده‌اند پیکانهایی نیز بر روی خطوط شکل‌ها رسم می‌شود که نشان دهنده جهت جریان داده‌ها بین عناصر حالت خواهد بود.

۵-۱-۴- پیاده‌سازی زیر مجموعه‌ای از دستورات MIPS

ما چند پیاده‌سازی مختلف از سخت‌افزار ارائه خواهیم کرد. در این فصل یک پردازنده ساده طراحی خواهد شد که در آن هر دستوری در یک کلاک انجام می‌گیرد ولی پریود این کلاک بزرگ می‌باشد. این پیاده‌سازی ساده همانند شکل ۱ خواهد بود. در این پردازنده ساده، هر دستوری در یک لبهٔ کلاک شروع به اجرا می‌کند و در لبهٔ‌کلاک بعدی اجرای آن خاتمه پیدا می‌کند. به دلیل اینکه پردازنده ساده طراحی شده در این فصل سرعت پایین‌تری دارد، ما یک پردازنده سریعتر را در فصل ۶ ارائه خواهیم کرد که از تکنیک پایپلاین استفاده می‌کند.

۵-۲- رجیسترها

به دلیل استفاده زیاد و کاربردهای بسیار رجیسترها در طراحی پردازنده، این بخش از فصل به مجموعه رجیسترهای پردازنده، اختصاص پیدا کرده است. رجیسترها در داخل یک پردازنده معمولاً به عنوان فضای ذخیره سازی موقت مورد استفاده قرار می‌گیرند. رجیسترها از حافظه‌های اصلی سریعترند و استفاده از آنها راحت‌تر است. هر رجیستری به تعداد بیت‌هایی که می‌تواند ذخیره کند در داخل خود فلیپ فلاپ دارد. به طور مثال یک رجیستر ۴ بیتی دارای ۴ عدد فلیپ‌فلاپ می‌باشد. هر رجیستری دارای ۲ ورودی به نام‌های clock و reset می‌باشد که بین همه فلیپ فلاپها مشترک است. ورودی کلاک، پالس ساعت را تأمین می‌کند و ورودی reset، اگر فعال شود همه فلیپ فلاپ‌ها و در نتیجه رجیستر را پاک می‌کند. بلاک دیاگرام یک رجیستر در شکل ۴ نشان داده شده است. ورودی‌های کلاک و reset به دلیل سادگی در این شکل نشان داده نشده‌اند، ولی به صورت ضمنی می‌دانیم که این ورودی‌ها وجود دارند.



شکل ۴: بلاک دیاگرام رجیستر

با توجه به بلاک دیاگرام، هر رجیستر دارای دو ورودی به نام‌های Load و Data Input (ورودی داده) می‌باشد. هر موقع که Load=1 باشد، داده ورودی در داخل رجیستر ذخیره خواهد شد. و هر موقع که Load=0 باشد، رجیستر محتوای فعلی خود را حفظ خواهد نمود. محتوای داخلی رجیستر همیشه از طریق خروجی Data Output (خروجی داده)، بدون توجه به ورودی Load در دسترس خواهد بود.

همان‌طور که گفته شد، یک رجیستر n بیتی دارای n عدد فلیپ فلاپ است. بنابراین تعداد بیت-هایی که می‌توان داخل رجیستر ذخیره نمود، n بیت است. پس ورودی Data Input شامل n خط است. همچنین طول داده‌ای که می‌توان از داخل رجیستر خواند، n بیت است. بنابراین خروجی Data Output نیز باید دارای n خط باشد. به دلیل اینکه تعداد خطوط Data Input و Data Output بیشتر از یک خط است، این خطوط در دیاگرام ضخیم‌تر رسم شده‌اند.

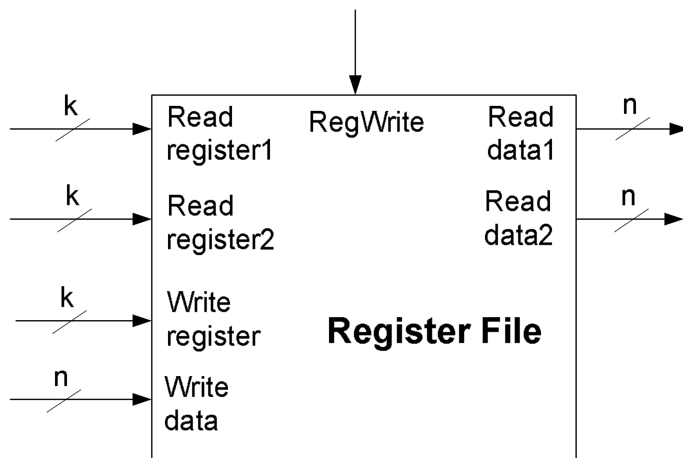
۵-۲-۱- رجیسترهای عمومی پردازنده

هر پردازنده‌ای در داخل خود تعدادی رجیستر برای کاربردهای عمومی دارد که معمولاً در دستورات محاسباتی و منطقی از آنها استفاده می‌کند. در پردازنده‌های مدرن، برای افزایش کارآیی و منظم‌تر شدن طراحی، رجیسترهای عمومی به صورت خاصی گروه‌بندی شده و در داخل یک ماژول به نام بانک رجیستر[1] قرار داده می‌شوند. در این بخش بانک رجیستر را مورد بررسی دقیق قرار می‌دهیم.

۵-۲-۱-۱- بانک رجیستر

بانک رجیستر بسیار شبیه به حافظه RAM می‌باشد که کلمات ذخیره شده در آن همان داده‌های داخل رجیسترهای بانک رجیستر می‌باشد. برای دسترسی به محتویات هر کدام از رجیسترهای داخل بانک رجیستر، باید آدرس آن رجیستر را مشخص کنیم. بلاک دیاگرام یک نمونه بانک رجیستر در شکل ۵ نشان داده شده است

[1] - Register File



شکل ۵: بانک رجیستر

این بانک رجیستر شامل مشخصات زیر است:

- این بانک رجیستر دارای دو پورت خواندن و یک پورت نوشتن است. یعنی در هر زمان می‌توان محتوای دو رجیستر مختلف از مجموعه رجیسترها را خواند و استفاده نمود و در همان حین یک عملیات نوشتن به داخل یکی از رجیسترها انجام داد.
- پورتهای Read register1، Read register2 و Write register برای آدرس‌دهی مورد استفاده قرار می‌گیرند. یعنی با هر کدام از این خطوط می‌توان یک رجیستر را آدرس‌دهی نمود. تعداد رجیسترها مساوی 2^k است در نتیجه تعداد خطوط آدرس می‌شود k.
- داده‌های خوانده شده از دو رجیستر بر روی پورتهای Read register1 و Read register2 خارج می‌شود. داده‌ای هم که قرار است داخل یکی از رجیسترهای بانک رجیستر نوشته شود، بر روی پورت Write data قرار داده می‌شود. هر رجیستر قادر است n بیت اطلاعات را ذخیره نماید. بنابراین طول هر کدام از پورتهای مربوط به داده مساوی n می‌باشد.
- به دلیل اینکه تعداد خطوط آدرس k و تعداد خطوط داده n می‌باشد، در نتیجه اندازه این بانک رجیستر مساوی $2^k \times n$ می‌باشد.

۵-۲-۱-۲- عملیات خواندن از بانک رجیستر

همان طور که ذکر شد، می‌توان در یک حین از دو رجیستر متفاوت عملیات خواندن را انجام داد. برای انجام این کار کافی است که آدرس دو رجیستر را به ترتیب بر روی پورتهای Read

Read register2 و register1 قرار داد. در این صورت داده مربوط به این رجیسترها به ترتیب بر روی پورتهای Read data1 و Read data2 ظاهر خواهد شد. به طور مثال فرض کنید برای دستوری مانند add $3,$2,$5 را که عملیات $3=$2+$5 را انجام می‌دهد، تصمیم گرفته باشیم محتوای دو رجیستر با شماره های 2 و 5 را از بانک رجیستر بخوانیم، در این صورت باید بر روی پورت Read register1 عدد 2 و بر روی پورت Read register2 عدد 5 قرار داده شود. با انجام این کار محتوای رجیستر 2 بر روی پورت Read data1 و محتوای رجیستر 5 بر روی پورت Read data2 به بیرون فرستاده خواهد شد.

۵-۲-۱-۳- عملیات نوشتن در داخل بانک رجیستر

برای انجام عملیات نوشتن، آدرس رجیستری که قرار است نوشته شود بر روی پورت Write register و داده‌ای که قرار است نوشته شود بر روی پورت Write data قرار داده می‌شود و همچنین پورت RegWrite نیز 1 می‌شود. با انجام این کار در لبه مثبت کلاک داده مورد نظر در آن رجیستر نوشته خواهد شد. به طور مثال فرض کنید بخواهیم عدد 398 را در داخل رجیستر شماره 8 بنویسیم. برای این کار باید مقدار 398 را بر روی پورت Write data و مقدار 8 را بر روی پورت Write register قرار دهیم و پورت ورودی RegWrite را نیز 1 نمائیم.

توجه: همان طور که گفته شد بانک رجیستر شامل رجیسترهای عمومی پردازنده است و چون هر رجیستری به کلاک نیاز دارد، باید بانک رجیستر ورودی کلاک هم داشته باشد. در بلاک دیاگرام بانک رجیستر ورودی کلاک نشان داده نشده است ولی به طور ضمنی این ورودی وجود دارد. ما هر موقعی که لازم باشد می‌توانیم از بانک رجیستر بخوانیم ولی عملیات نوشتن فقط در لبه مثبت کلاک انجام خواهد شد.

۵-۲-۱-۴- طراحی سخت افزار بانک رجیستر

پرازنده MIPS دارای ۳۲ رجیستر ۳۲ بیتی است. بنابراین برای بانک رجیستر آن تعداد خطوط آدرس و خطوط داده باید به ترتیب مساوی ۵ و ۳۲ باشد. وقتی که می‌خواهیم از بین ۳۲ رجیستر، دو مورد را انتخاب کنیم و محتویات آنها را بخوانیم، حتماً باید برای هر کدام از انتخابها از یک مالتی-پلکسر استفاده کنیم چون فقط با مالتی‌پلکسر است که می‌توانیم عمل انتخاب را انجام دهیم. از طرفی برای عملیات نوشتن باید فقط خط Load یکی از رجیسترها را فعال کنیم چون در عملیات نوشتن فقط یکی از رجیسترها نوشته خواهد شد. همان طور که می‌دانیم دیکدر مداری است که در هر زمان فقط یکی از خروجی‌های آن فعال می‌شود و بقیه خروجی‌ها غیر فعال‌اند. بنابراین به هنگام نوشتن داخل

بانک رجیستر می‌توان از یک دیکدر برای فعال کردن خط Load یکی و فقط یکی از رجیسترها استفاده نمود.

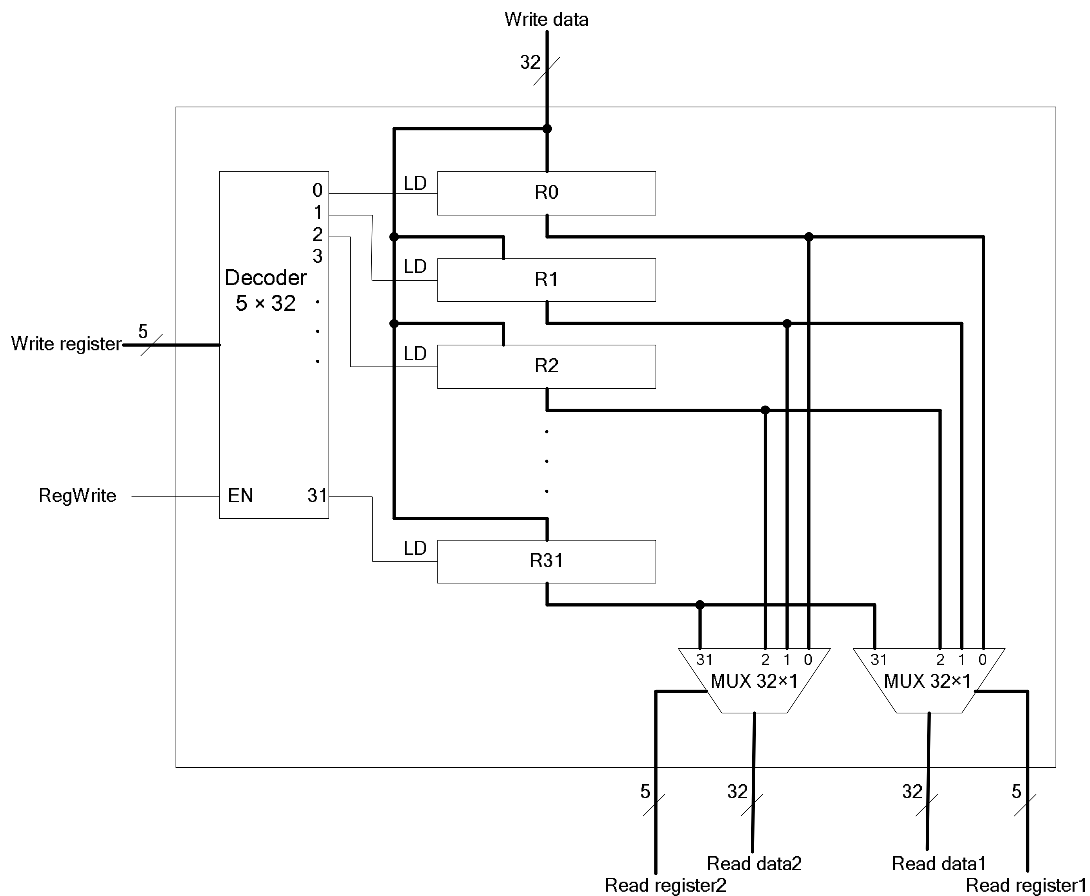
شکل ۶ مدار داخلی بانک رجیستر پردازنده MIPS را نشان می‌دهد. خطوط پر رنگ در این شکل نشان دهنده این است که این خط شامل چند خط (چند بیت) است به طور مثال خط write data که پر رنگ است شامل ۳۲ خط است. همان طور که در این شکل دیده می‌شود، برای انتخاب یک رجیستر از بین ۳۲ رجیستر از یک مالتی پلکسر ۳۲ به ۱ استفاده می‌شود و چون مالتی پلکسر دارای ۳۲ خط ورودی است، پس تعداد خطوط انتخاب مالتی‌پلکسر باید ۵ باشد. از آنجایی که باید این امکان وجود داشته باشد که ما بتوانیم محتوای دو رجیستر را در یک زمان بخوانیم، به همین دلیل در این شکل دو مالتی پلکسر تعبیه شده است. ورودی‌های Read register1 و Read register2 برای انتخاب دو رجیستر از بین ۳۲ رجیستر در عملیات خواندن استفاده می‌شوند، به همین دلیل باید این دو ورودی را به خطوط انتخاب مالتی پلکسرها وصل کنیم (هر کدام به یک مالتی پلکسر)، و از آنجا که هر کدام از مالتی پلکسرها از بین ۳۲ رجیستر یکی را انتخاب می‌کنند بنابراین هر کدام از این ورودی‌ها باید شامل ۵ بیت باشند. خروجی هر کدام از مالتی پلکسرها محتوای یکی از ۳۲ رجیستر موجود است و چون هر رجیستری ۳۲ بیتی است بنابراین خروجی مالتی پلکسرها نیز ۳۲ بیتی است. خروجی دو مالتی پلکسر به ترتیب به خروجی‌های Read data1 و Read data2 وصل شده است، بنابراین هر کدام از این خروجی‌ها باید ۳۲ بیتی باشد.

همان طور که ذکر شد از دیکدر می‌توان برای عملیات نوشتن داخل بانک رجیستر و فعال کردن خط Load رجیسترها استفاده نمود. در مدار بانک رجیستر پردازنده MIPS به دلیل وجود ۳۲ رجیستر، از یک دیکدر ۵ به ۳۲ استفاده شده است. ورودی Write register برای انتخاب رجیستری که نوشته خواهد شد به کار می‌رود، به همین دلیل این ورودی را باید به خطوط ورودی دیکدر وصل کنیم و تعداد بیت‌های آن باید مساوی ۵ باشد. همچنین دیکدر استفاده شده دارای یک خط فعال ساز[1] به نام EN است که هر موقع فعال شود (۱ شود)، یکی از خروجی‌های دیکدر فعال خواهد شد و بنابراین عملیات نوشتن داخل یکی از ۳۲ رجیستر انجام خواهد گرفت. همان طور که می‌دانیم در پردازنده MIPS همه دستورات نتیجه خود را داخل رجیستر نمی‌نویسند. به طور مثال دستوراتی مانند add و sub نتیجه خود را داخل یک رجیستر می‌نویسند ولی دستوراتی مانند sw و beq داخل رجیسترها چیزی نمی‌نویسند. بنابراین باید بتوانیم عملیات نوشتن داخل بانک رجیستر را به صورت

---

[1] - Enable

کنترل شده انجام دهیم. ورودی WriteReg که به خط فعال ساز دیکدر وصل می‌شود برای این منظور استفاده می‌شود.

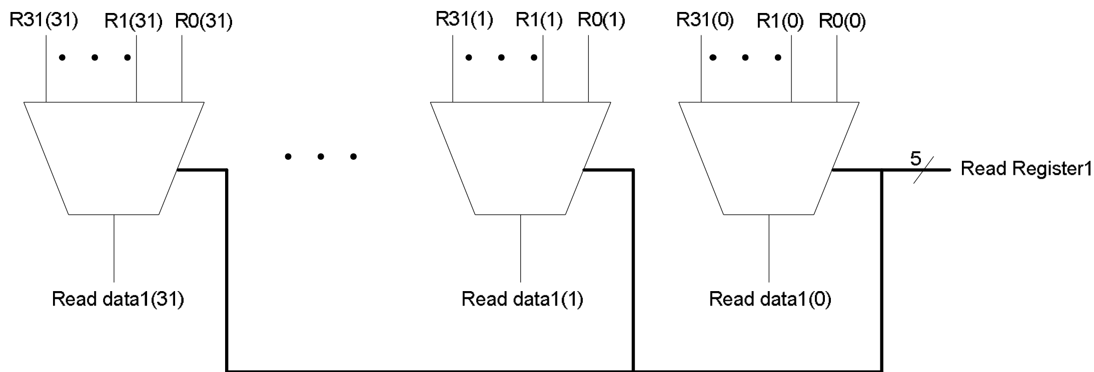


شکل ۶: مدار داخلی بانک رجیستر پردازنده MIPS

خط Write data به ورودی داده همه رجیسترهای بانک رجیستر وصل شده است. وقتی عملیات نوشتن انجام می‌شود، این داده ورودی داخل یکی از رجیسترها نوشته خواهد شد. به دلیل اینکه هر رجیستر شامل ۳۲ بیت است، این خط ورودی نیز باید شامل ۳۲ بیت باشد.

نکته‌ای که در اینجا باید ذکر شود این است که هر کدام از مالتی پلکسرهای بانک رجیستر نشان داده شده از چند مالتی پلکسر تشکیل شده‌اند: یک مالتی پلکسر برای انتخاب از بین بیت‌های 0 ، یک مالتی پلکسر برای انتخاب از بین بیت‌های ۱ و به همین ترتیب. در کل چون ۳۲ بیت داریم، هر کدام از مالتی پلکسرها شامل ۳۲ مالتی پلکسر هستند که خط انتخاب همه این مالتی پلکسرها مشترک است. برای روشنتر شدن مطلب، شکل ۷ را مشاهده کنید. در این شکل مالتی پلکسر مربوط به Read data1 نشان داده شده که شامل ۳۲ مالتی پلکسر می‌باشد. مالتی پلکسر سمت راست برای انتخاب از بین بیت-

های 0 ، مالتی پلکسر بعدی برای انتخاب از بین بیت‌های ۱ و به همین ترتیب تا اینکه مالتی پلکسر آخری هم برای انتخاب از بین بیت‌های ۳۱ رجیسترهای بانک رجیستر به کار می‌رود. همان طور که مشاهده می‌شود خط انتخاب همه این مالتی پلکسرها مشترک است بنابراین در همه مالتی پلکسرها، بیت‌های مربوط به یک رجیستر انتخاب خواهد شد. به طور مثال اگر مالتی پلکسر اولی بیت 0 مربوط به رجیستر R1 را انتخاب کند (Read Register = 00001)، حتماً رجیستر بعدی بیت ۱ رجیستر R1 را انتخاب خواهد کرد، و به همین ترتیب. بنابراین در نهایت همه بیت‌های مربوط به R1 انتخاب خواهند شد.



شکل ۷: مدار هر کدام از مالتی پلکسرهای بانک رجیستر

۵-۲-۲- رجیسترهای مخصوص

در داخل هر پردازنده تعدادی رجیستر خارج از مجموعه رجیسترهای عمومی یا عام منظوره وجود دارند. این رجیسترها استفاده خاصی دارند و نمی‌توان در دستورات پردازنده از آنها استفاده نمود. به عبارتی توسط کاربر قابل دسترسی نیستند. در پردازنده MIPS، تعدادی رجیستر وجود دارد که استفاده خاص داشته و خارج از مجموعه رجیسترهای عمومی یا بانک رجیستر قرار گرفته‌اند. یکی از این رجیسترها، رجیستر شمارنده برنامه یا PC[1] می‌باشد. رجیستر PC برای دسترسی به حافظه دستورات مورد استفاه قرار می‌گیرد و آن را آدرس دهی می‌کند. به عبارتی دستور بعدی که در داخل پردازنده اجرا خواهد شد از آدرس PC خوانده خواهد شد. در پردازنده MIPS دو رجیستر مخصوص دیگر به نام‌های Hi و Lo وجود دارد که فقط برای عملیات ضرب و تقسیم از آنها استفاده می‌شود.

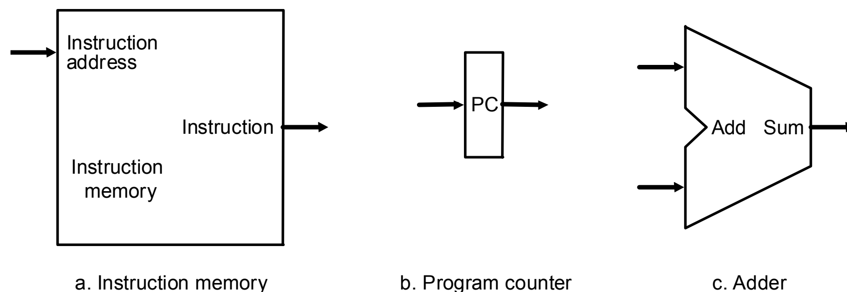
تعریف: به رجیسترهایی که استفاده مخصوص دارند رجیسترهای خاص منظوره یا رجیسترهای با کاربرد خاص نیز گفته می‌شود. و به رجیسترهای عمومی رجیسترهای با عام منظوره یا رجیسترهای با کاربرد عام نیز گفته می‌شود.

[1] - Program Counter

۵-۳- ساختن یک مسیر داده

مسیر داده یا data path واحدی از پردازنده است که همهٔ عملیات محاسباتی و منطقی مورد نیاز همهٔ دستورات را انجام می‌دهد. یک روش ساده برای طراحی مسیر داده، پیدا کردن ماژولهای اصلی مورد نیاز برای اجرای هر کدام از کلاس‌های دستورات می‌باشد. بنابراین ما برای هر کدام از کلاس-های دستورات، ماژولهای مورد نیاز را پیدا کرده و بخش‌هایی از مسیر داده که برای اجرای این کلاس دستور مورد نیاز ایت را طراحی خواهیم کرد. در کنار عناصر مسیر داده که برای هر بخش نشان خواهیم داد، سیگنالهای کنترلی مربوطه را نیز در برخی موارد توضیح خواهیم داد.

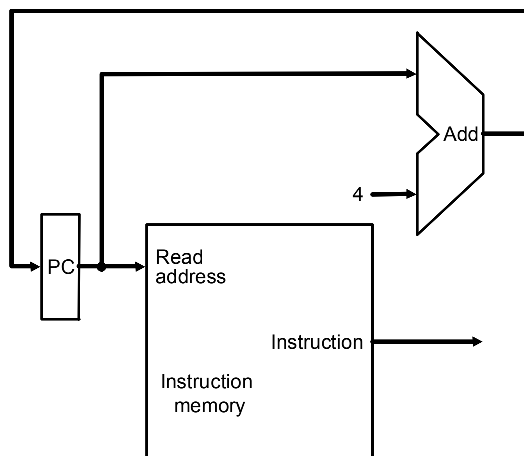
اولین ماژولی که مورد نیاز است یک محل ذخیره‌سازی برای نگهداری دستورات یک برنامه است. یک واحد حافظه که یک عنصر حالت می‌باشد برای نگهداری دستورات به کار می‌رود و به کمک یک خط آدرس می‌توانیم این حافظه را آدرس دهی کرده و دستورات را از داخل آن بخوانیم. عنصر حافظه مورد نیاز در شکل ۸ نشان داده شده است. آدرس دستور نیز باید در داخل یک عنصر حالت نگهداری شود که به آن شمارنده برنامه[1] یا PC گفته می‌شود. شمارنده برنامه نیز در شکل ۸ نشان داده شده است. و در نهایت ما نیاز به این داریم که PC را اضافه کنیم تا اینکه آدرس دستور بعدی بدست آید بنابراین نیاز به یک جمع کننده نیز داریم. این جمع کننده یک مدار ترکیبی است که می‌تواند از همان ALU که در فصل ۴ طراحی شد بدست آید، برای این منظور کافی است که خط کنترلی ALU را طوری مقداردهی کنیم که همیشه عملیات جمع را انجام دهد، به عبارتی باید به این صورت مقداردهی کنیم: $\text{ALU Operation} = 010$. ما این ALU خاص را که فقط عملیات جمع انجام می‌دهد، با برچسب Add نشان خواهیم داد. جمع کننده آدرس نیز در شکل ۸ نشان داده شده است.



شکل ۸: برای ذخیره سازی و دسترسی به دستورات، دو عنصر حالت مورد نیاز است. همچنین یک جمع کننده نیز برای محاسبه آدرس دستور بعدی مورد نیاز است.

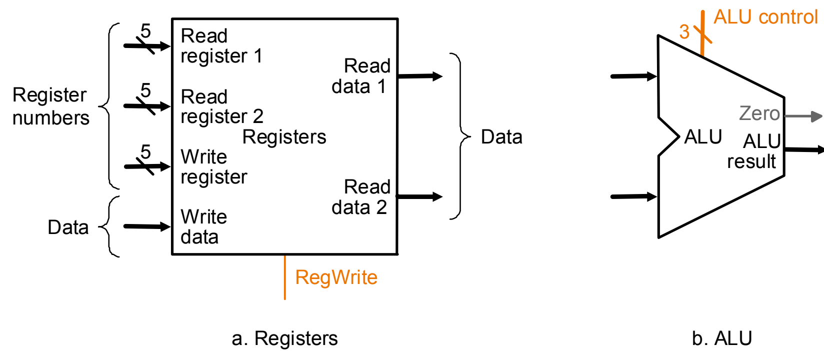
[1] - Program counter

برای اجرای هر دستور، در ابتدا باید دستور از حافظه خوانده شود. به این عمل، واکشی دستور یا Fetch گفته می‌شود. برای اینکه آمادگی این را داشته باشیم که پس از اجرای این دستور، دستور بعدی را نیز اجرا کنیم، ما باید شمارنده برنامه را طوری اضافه کنیم تا به ۴ بایت بعدتر اشاره کند چون هر دستوری در پردازنده MIPS دارای طول ۴ بایت است. بنابراین برای اینکه PC به دستور بعدی اشاره کند باید به آن ۴ واحد اضافه نمود. مسیر داده مورد نیاز برای این کار در شکل ۹ نشان داده شده است. همان طور که مشاهده می‌شود، این شکل از ۳ عنصر نشان داده شده در شکل ۸ استفاده می‌نماید.



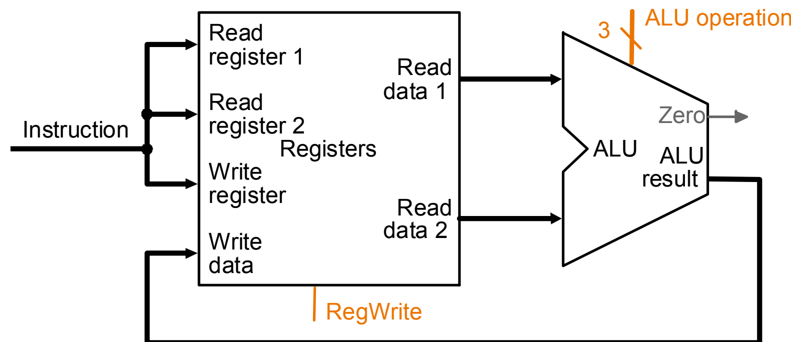
شکل ۹: قسمتی از مسیر داده که برای واکشی دستور از حافظه و همچنین افزایش PC به کار می‌رود

حال بیائید دستورات نوع R را در نظر بگیریم. همهٔ این دستورات محتوای دو رجیستر را می-خوانند، یک عملیات ALU بر روی محتوای رجیسترهای خوانده شده انجام می‌دهند و نتیجه حاصل شده را در داخل یک رجیستر می‌نویسد. دستورات این کلاس شامل add، sub، slt، and و or می-باشند. به طور مثال دستور add $t1, $t2, $t3 را در نظر بگیرید. این دستور محتوای دو رجیستر$t2 و $t3 را خوانده و با هم جمع کرده و نتیجه را در $t1 می‌نویسد. برای اجرای دستورات نوع R ما نیاز به یک بانک رجیستر داریم تا از طریق آن محتوای دو رجیستر را بخوانیم. همچنین برای انجام عملیات، نیاز به یک ALU داریم. در نهایت که عملیات انجام گرفت نتیجه باید در داخل بانک رجیستر ذخیره گردد. ماژولهای لازم برای دستورات نوع R در شکل ۱۰ نشان داده شده‌اند. ALU نشان داده شده در این شکل همان ALU طراحی شده در فصل ۴ می‌باشد. این ALU دو ورودی ۳۲ بیتی دریافت نموده و یک خروجی ۳۲ بیتی را به عنوان نتیجه تولید می‌کند. بانک رجیستر نشان داده شده در این شکل همان بانک رجیستر طراحی شده در این فصل است.



شکل ۱۰: دو عنصر مورد نیاز برای پیاده سازی دستورات نوع R عبارتند از: بانک رجیستر و ALU

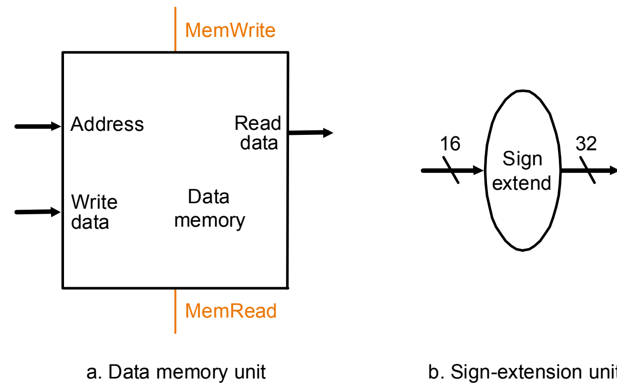
مسیر داده لازم برای اجرای دستورات نوع R که از ماژولهای شکل ۱۰ استفاده می‌کند در شکل ۱۱ نشان داده شده است. به دلیل اینکه شماره رجیسترهایی که باید خوانده شوند و یا نوشته شوند از فیلدهای مربوطه در دستوری که اجرا می‌شود، استخراج می‌شوند، ما برای این شکل یک ورودی به نام Instruction در نظر گرفته‌ایم. این ورودی باید به خروجی حافظه در شکل ۹ وصل گردد چون این حافظه، دستورات برنامه را نگهداری می‌کند و خروجی آن همان دستوری است که اجرا خواهد شد.



شکل ۱۱: مسیر داده دستورات نوع R

حال بیائید دستورات مراجعه به حافظه lw و sw را که دارای شکل کلی lw \$t1, offset_value(\$t2) و sw \$t1, offset_value(\$t2) می‌باشند را بررسی نمائیم. این دستورات با جمع کردن رجیستر پایه که در اینجا t2 بوده با یک مقدار ثابت ۱۶ بیتی علامت دار که همان Offset می‌باشد، یک آدرس برای حافظه محاسبه می‌کنند. اگر دستور sw باشد، مقداری که در داخل حافظه ذخیره خواهد شد باید از بانک رجیستر خوانده شود. به دلیل اینکه محتوای یک رجیستر در داخل حافظه ذخیره خواهد شد که باید آن را از داخل بانک رجیستر خواند (در اینجا رجیستر t1). اگر دستور از نوع lw باشد، مقداری که از حافظه خوانده می‌شود باید در داخل بانک رجیستر ذخیره شود (در داخل رجیستر t1 بانک رجیستر). بنابراین برای دستورات مراجعه به حافظه، ما به ALU و بانک رجیستر نشان داده شده در شکل ۱۰ نیاز خواهیم داشت.

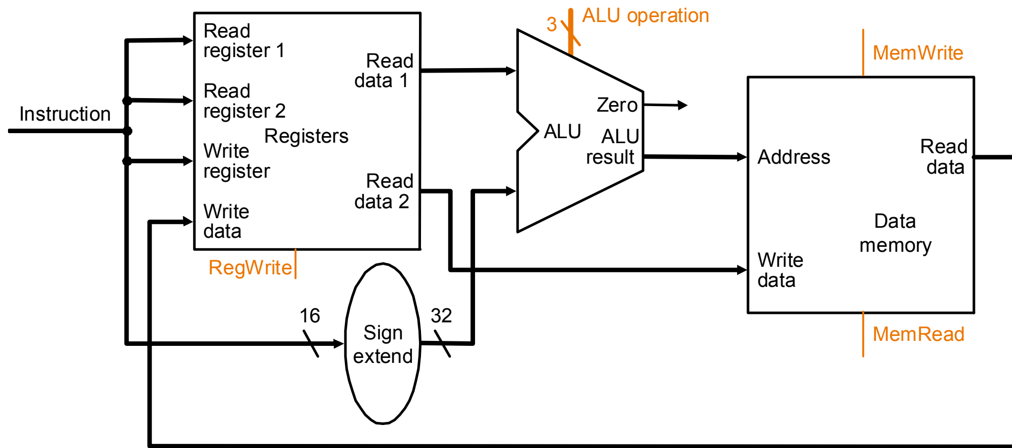
علاوه بر این، ما به یک واحد به نام واحد گسترش بیت علامت که مقدار ثابت ۱۶ بیتی فیلد Offset را به ۳۲ بیت تبدیل می‌کند نیاز داریم. این واحد ۱۶ بیت بالا را با علامت عدد پر می‌کند. همچنین ما به یک ماژول حافظه برای خواندن یا نوشتن داده‌ها نیاز داریم. این ماژول حافظه چون برای داده‌ها استفاده می‌شود، حافظه داده[1] نامیده می‌شود. چون حافظه داده، هم می‌تواند خوانده شود و هم می‌تواند نوشته شود، بنابراین باید دارای خطوط کنترلی برای عملیات نوشتن و خواندن باشد. همچنین این حافظه باید یک ورودی برای آدرس و یک ورودی برای داده‌ای که نوشته خواهد شد داشته باشد. شکل ۱۲ ماژولهای حافظه داده و گسترش بیت علامت را نشان می‌دهد.



شکل ۱۲: ماژول‌های حافظه داده و گسترش بیت علامت که در دستورهای مراجعه به حافظه از آنها استفاده می‌شود

شکل ۱۳ مسیر داده مربوط به دستورات lw و sw را نشان می‌دهد. همان طور که دیده می‌شود در این شکل ماژولهای ALU، بانک رجیستر، گسترش بیت علامت و حافظه داده‌ها وجود دارند. در این شکل ALU یک عدد ثابت ۱۶ بیت که تبدیل به ۳۲ بیت شده را با محتوای یک رجیستر که از بانک رجیستر خوانده می‌شود با هم جمع کرده و آدرس حافظه را نتیجه می‌دهد. شماره رجیستری که محتوای آن از بانک رجیستر خوانده می‌شود و همچنین عدد ثابت ۱۶ بیتی در فیلدهای مربوطه در دستورهای lw و sw قرار دارند. بنابراین یکی از ورودیهای این مسیر داده دستور خوانده شده از حافظه است.

[1] - Data memory



شکل ۱۳: مسیر داده دستورات مراجعه به حافظه

دستور beq دارای سه عملوند[1] می‌باشد: دو رجیستر که با هم مقایسه می‌شوند و یک مقدار ثابت ۱۶ بیتی که برای بدست آوردن آدرس پرش استفاده می‌شود. به آدرس پرش، آدرس مقصد پرش یا branch target address نیز گفته می‌شود. شکل دستور beq به صورت beq $t1,$t2,offset. برای پیاده‌سازی این دستور ما باید مقدار ثابت ۱۶ بیتی را با محتوای شمارنده برنامه جمع کنیم.

در مورد دستورات پرش باید به دو نکته مهم توجه کنیم:

- معماری مجموعه دستورات بیان می کند که آدرس base برای دستورات پرش، آدرس دستور بعد از دستور پرش می‌باشد. به دلیل اینکه ما محاسبه PC+4 (آدرس دستور بعدی) را در مسیر داده مربوط به واکشی دستور انجام داده‌ایم، راحت‌تر این است که ما این مقدار را برای محاسبه آدرس دستور پرش نیز استفاده کنیم.
- معماری مجموعه دستورات همچنین بیان می کند که فیلد offset باید دو مرتبه به سمت چپ شیفت پیدا کند. به دلیل این که این offset، آفست یک کلمه ۳۲ بیتی است. همان طور که در فصل ۳ نیز توضیح داده شد، این میزان شیفت باعث می‌شود که محدودهٔ مؤثر آفست با ضریب ۴ افزایش پیدا کند.

برای اینکه مورد دوم را در سخت‌افزار به درستی انجام دهیم باید فیلد آفست را دو مرتبه به سمت چپ شیفت دهیم.

علاوه بر محاسبهٔ آدرس پرش، ما باید این مورد را نیز مشخص کنیم که آیا دستور beq بعد از اجرا واقعاً به آدرس مقصد پرش، پرش را انجام خواهد داد و یا اینکه پرش انجام نخواهد شد و

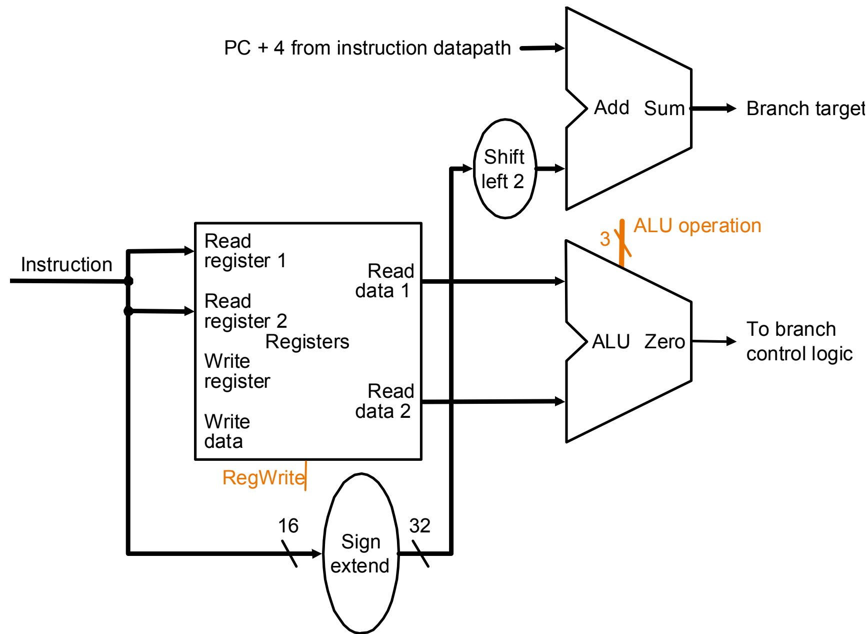
[1] - Operand

بعد از دستور beq دستور پشت سر آن اجرا خواهد شد. موقعی که شرط مقایسه برقرار می‌شود (یعنی دو عملوند با هم مساوی می‌شوند)، مقدار جدید pc، آدرس محاسبه شده برای مقصد پرش خواهد بود و در این حالت می‌گوییم که پرش انجام می‌شود و یا اصطلاحاً گفته می‌شود Branch is taken.

اگر دو عملوند با هم مساوی نباشند، مقدار جدید PC مساوی PC+4 خواهد بود (آدرس دستور پشت سر دستور پرش). در این حالت گفته می‌شود که پرش انجام نمی‌شود و یا اصطلاحاً Branch is not taken..

بنابراین مسیر داده دستور پرش باید دو عملیات را انجام دهد: محاسبه آدرس مقصد پرش و مقایسه محتوای رجیسترها. (دستورهای پرش برای انجام شدن درست عملیات نیاز دارند که مسیر داده مربوط به واکشی دستور را مقداری تغییر دهند که این مورد را بعداً توضیح خواهیم داد.)

شکل ۱۴ مسیر داده دستور beq را نشان می‌دهد. برای محاسبهٔ آدرس مقصد پرش، مسیر داده دستورbeq باید شامل یک واحد گسترش بیت علامت و یک واحد ALU باشد. برای انجام مقایسه ما باید از بانک رجیستر استفاده کنیم که محتوای دو رجیستر را در اختیار ما قرار دهد و همچنین از یک ALU نظیر ALU فصل ۴ که عملیات مقایسه این دو رجیستر را انجام دهد. به دلیل اینکه این ALU یک خروجی به نام Zero دارد که نشان می‌دهد نتیجه عملیات ALU صفر شده است یا خیر، ما می-توانیم به کمک ALU عملیات تفریق را بر روی محتوای دو رجیستر انجام دهیم (خط کنترل ALU را طوری مقداردهی کنیم که عملیات Sub را انجام دهد) و بعد خروجی Zero را بررسی کنیم اگر این خروجی یک شود نشان دهنده این است که دو رجیستر با هم مساوی هستند و در غیر این صورت مساوی نیستند. در ادامه ما نحوه اتصال خطوط کنترلی را توضیح خواهیم داد.



شکل ۱۴: مسیر داده دستور beq

دستور jump به این صورت انجام می‌شودکه ۲۸ بیت پایین PC را با ۲۶ بیت از بیت‌های دستورالعمل j که ۲ بیت به سمت چپ شیفت داده می‌شود، پر می‌کنیم. این شیفت به سادگی با قرار دادن دو بیت (00) در سمت راست ۲۶ بیت انجام می‌شود. نتیجه شیفت ۲۸ بیت می‌شود که این ۲۸ بیت به جای ۲۸ بیت پایین PC قرار می‌گیرد.

حال که ما مسیر داده مورد نیاز برای اجرای کلاس‌های مختلف دستورات را توضیح دادیم، می-توانیم این مسیر داده‌ها را با هم ترکیب نموده و یک مسیر داده کلی ایجاد کنیم و خطوط کنترلی را نیز به آن اضافه کنیم تا اینکه طراحی ما کامل شود. در بخش‌های بعدی ما یک پردازنده ساده به کمک مسیر داده‌هایی که توضیح داده شد، طراحی خواهیم نمود. این پردازنده هر دستوری را در یک کلاک انجام خواهد داد. چون هر دستور در این پردازنده در یک کلاک انجام می‌شود، نامی که به آن اختصاص خواهیم داد، پردازنده تک سیکلی یا Single cycle processor خواهد بود.

۵-۴- طراحی یک پردازنده ساده

در این بخش ما ساده‌ترین پیاده‌سازی ممکن از زیر مجموعه انتخاب شده پردازنده MIPS را ارائه خواهیم کرد. این پیاده‌سازی شامل یک مسیر داده ساده و یک بخش کنترل ساده است. مسیر داده پردازنده ساده از ترکیب مسیر داده‌های معرفی شده در بخش قبلی ایجاد خواهد شد. پردازنده‌ای که ما طراحی خواهیم کرد دستورات lw، sw، beq، add، sub، and، or و slt را پشتیبانی خواهد کرد.

۵-۴-۱- طراحی یک مسیر داده ساده

فرض کنید که ما بخواهیم یک مسیر داده را از قطعه‌هایی که در شکل ۹، شکل ۱۱، شکل ۱۳ و شکل ۱۴ نشان داده شده‌اند بسازیم. این مسیر داده ساده هر دستوری را در یک کلاک اجرا خواهد کرد. اگر هر دستوری در یک کلاک اجرا شود، در این صورت هیچ کدام از مسیرهای داده مرجع نمی‌توانند بیش از یک بار در یک دستور استفاده شوند، بنابراین اگر ماژولی بیش از یک بار استفاده شود باید به تعداد مورد نیاز از آن ماژول قرار دهیم.

در مسیر داده ساده‌ای که طراحی می‌شود ممکن است یک ماژول در بیش از چند دستور استفاده شود. برای اینکه از یک ماژول در بیش از چند دستور استفاده شود، ما باید امکان اتصال چند ورودی را به ورودی آن ماژول فراهم کنیم و یک سیگنال کنترلی داشته باشیم که از بین آنها یکی را انتخاب کند. عمل انتخاب معمولاً به کمک مالتی پلکسر صورت می‌گیرد.

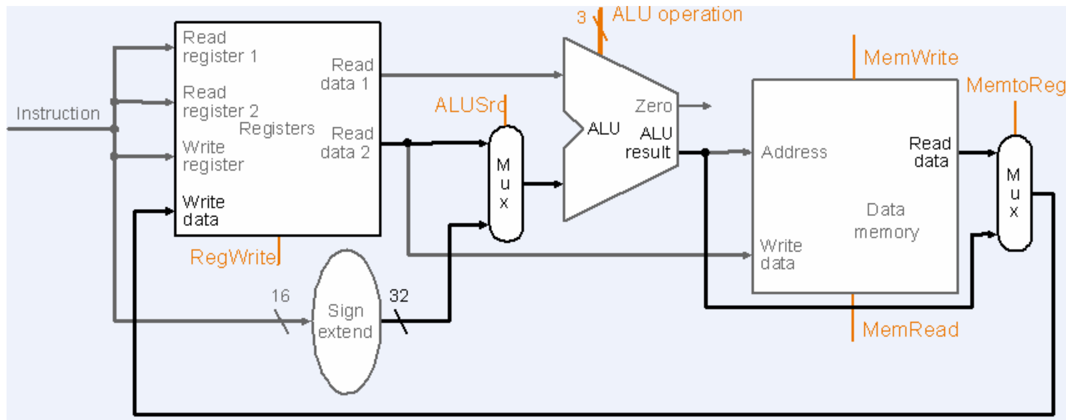
مثال: مسیر داده دستورات نوع R نشان داده شده در شکل ۱۱ و مسیر داده دستورات مراجعه به حافظه نشان داده شده در شکل ۱۳، شباهت زیادی به هم دارند.

تفاوتهای مهمی که بین آنها وجود دارد به شرح زیر است:

- ورودی دومی که وارد ALU می‌شود یا یک رجیستر است (اگر دستور از نوع R باشد) و یا داده‌ای است که گسترش بیت علامت داده شده است (در دستورات مراجعه به حافظه)
- مقداری که در داخل یک رجیستر ذخیره خواهد شد یا از ALU می‌آید (دستورات نوع R) و یا از حافظه(دستور load)

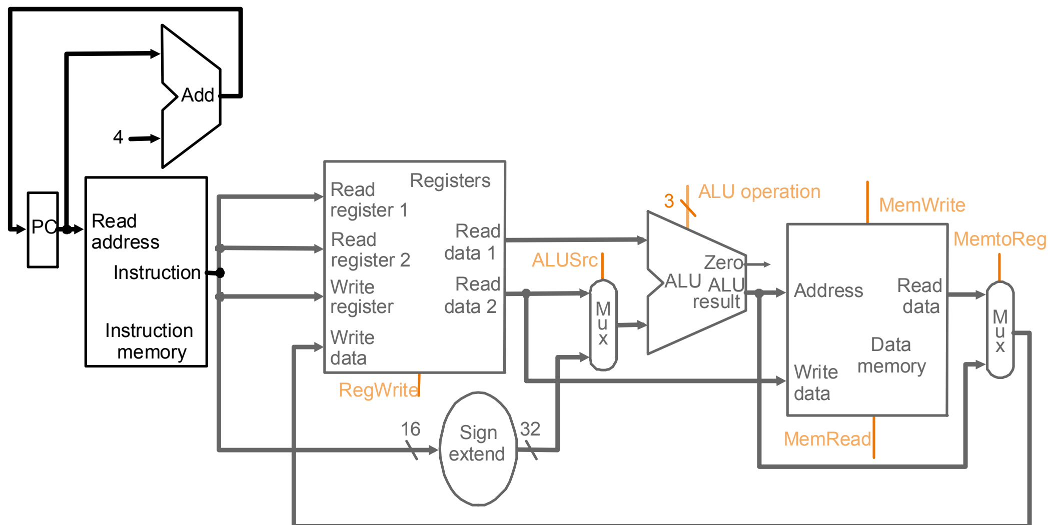
توضیح دهید که چگونه می‌توانیم این دو مسیر داده را با هم ترکیب کنیم به شرطی که از ماژولهای مشترکی که در این دو شکل وجود دارند فقط یکبار استفاده کنیم؟ (به طور مثال ماژول بانک رجیستر در هر دو شکل وجود دارد و در ترکیب دو مسیر داده فقط باید از یک بانک رجیستر استفاده کنیم).

جواب: برای ترکیب این دو مسیر داده و استفاده از فقط یک ALU و یک بانک رجیستر، ما باید برای ورودی دوم ALU این امکان را فراهم کنیم که از دو منبع مختلف داده دریافت کند، و همین طور برای داده‌ای که داخل بانک رجیستر ذخیره خواهد شد باید از دو منبع مختلف داده استفاده کنیم. بنابراین ما به یک مالتی‌پلکسر در ورودی دوم ALU و یک مالتی‌پلکسر دیگر در ورودی دوم بانک رجیستر نیاز خواهیم داشت. شکل ۱۵ مسیر داده ترکیب شده را نشان می‌دهد.



شکل ۱۵: ترکیب مسیر داده دستورات مراجعه به حافظه و دستورات نوع R

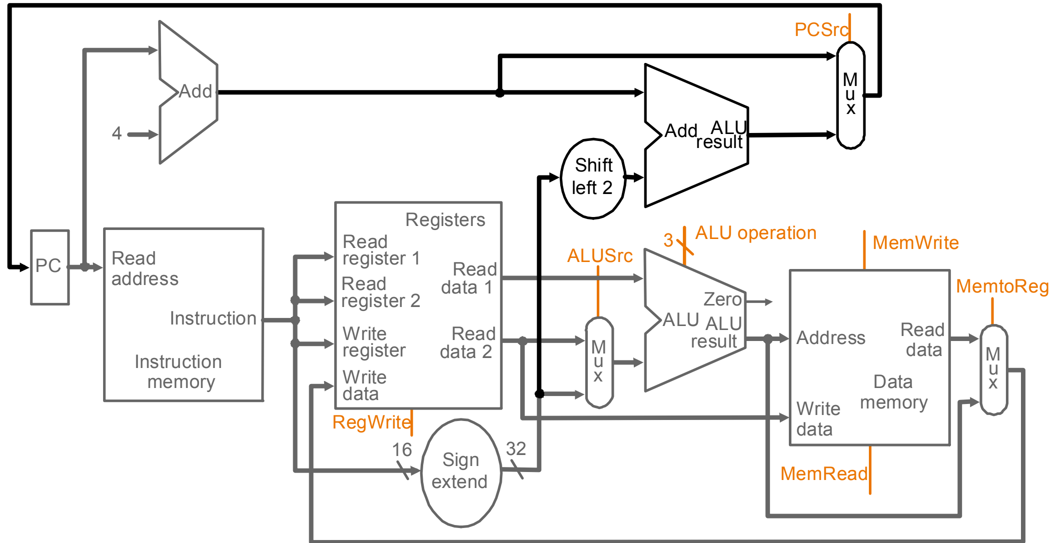
مسیر داده مربوط به واکشی دستور که در شکل ۹ نشان داده شده است به راحتی می‌تواند به مسیر داده شکل ۱۵ اضافه گردد. شکل ۱۶ نتیجه را نشان می‌دهد. این مسیر داده دارای دو حافظه جدا یکی برای دستورات و دیگری برای داده‌ها است. همچنین این مسیر داده به دو ALU احتیاج دارد به دلیل اینکه یکی از ALU ها برای اضافه کردن PC و دیگری در همان زمان برای اجرای دستور به کار می‌رود.



شکل ۱۶: ترکیب مسیرهای داده دستورات مراجعه به حافظه، دستورات نوع R و واکشی دستور

حال ما می‌توانیم با اضافه کردن مسیر داده دستورات پرش نشان داده شده در شکل ۱۴ به شکل ۱۶ یک مسیر داده ساده برای معماری MIPS ایجاد کنیم. شکل ۵-۱۳ مسیر داده‌ای را که ما با ترکیب همهٔ تکه مسیرهای داده جداگانه بدست آورده‌ایم نشان می‌دهد. دستور پرش از ALU اصلی برای مقایسه دو عملوند رجیستری استفاده می‌کنید، بنابراین ما باید جمع کننده‌ای که در شکل ۵-۱۰ برای بدست آوردن آدرس پرش استفاده می‌شود را در مسیر داده نهایی حفظ کنیم.

همان طور که در شکل ۵-۱۳ نشان داده شده است یک مالتی‌پلکسر دیگر نیز مورد نیاز خواهد بود تا از بین آدرس بعدی (PC+4) و آدرس مقصد پرش یکی را برای نوشته شدن در داخل PC یکی را انتخاب کنیم.



شکل ۱۷: مسیر داده ساده برای معماری MIPS که از ترکیب همه قطعه مسیرهای قبلی بدست آمده است

حال که ما طراحی مسیر داده ساده را کامل کردیم، می‌توانیم واحد کنترل را به آن اضافه کنیم و بدین ترتیب، کار طراحی پردازنده ساده را کامل کنیم. کار واحد کنترل همان طور که از نام آن پیداست کنترل عملکرد بخش‌های مختلف پردازنده است. واحد کنترل است که بسته به دستوری که اجرا می‌شود، به مسیر داده فرمان می‌دهد که عملیات مشخصی را انجام دهد مثلاً اگر دستور add اجرا می‌شود، عملیات جمع را انجام دهد. واحد کنترل باید توانایی دریافت ورودیها را داشته باشد و سیگنالهای write مورد نیاز برای همهٔ عناصر حالت، خطوط انتخاب همهٔ مالتی‌پلکسرها و خطوط کنترلی ALU را تولید نماید. از آنجایی که کنترل ALU تا اندازه‌ای با کنترل عناصر دیگر متفاوت است، بنابراین بهتر است قبل از طراحی بقیه قسمتهای واحد کنترل، در ابتدا طراحی واحد کنترل ALU را انجام دهیم.

۵-۴-۲- طراحی واحد کنترل ALU

اگر به یاد داشته باشید ALU طراحی شده در فصل ۴ دارای سه ورودی کنترلی بود. فقط ۵ ترکیب از ۸ ترکیب ممکن ورودیها در این ALU استفاده می‌شد و عملکرد ALU مطابق با جدول زیر بود:

ورودی‌های کنترل ALU	عملی که ALU انجام می‌دهد
000	AND
001	OR
010	add
110	subtract
111	Set on less than (slt)

بسته به کلاس دستور، ALU باید یکی از ۵ عملیات این جدول را انجام دهد. برای دستورات lw و sw ما از ALU برای محاسبه آدرس حافظه استفاده می‌کنیم که در این حالت ALU عملیات Add را انجام می‌دهد. برای دستورات نوع R، ALU باید یکی از ۵ عملیات add، sub، and، or یا slt را انجام دهد (بسته به اینکه فیلد ۶ بیتی Funct در فرمت دستور چه مقداری داشته باشد.) برای دستور beq نیز باید عملیات تفریق توسط ALU انجام شود.

ما می‌توانیم خطوط کنترلی ALU را توسط یک واحد کنترل کوچک تولید نمائیم. این واحد کنترلی کوچک، در ورودی خود فیلد ۶ بیتی Funct و دو بیت را که ما آن را ALUOp نام‌گذاری می‌کنیم دریافت می‌کند. ALUOp مشخص می‌کند که عملیاتی که باید انجام گیرد، add (00) برای دستورهای lw و sw، یا sub (10) برای beq و یا توسط فیلد Funct دستور مشخص شود (10). خروجی این واحد کنترل کوچک سه بیت کنترلی مربوط به خطوط کنترل ALU می‌باشد.

شکل ۱۸ یک جدول صحت را نشان می‌دهد که در آن خروجی‌های واحد کنترل کوچک که همان خطوط کنترل ALU می‌باشند بر اساس ورودیها که همان فیلد Funct و دو بیت ALUOp می‌باشند، مقداردهی شده‌اند. برای کامل‌تر شدن شکل، ارتباط بین بیت‌های ALUOp و opcode دستور نیز نشان داده شده است. در ادامه این فصل خواهیم دید که بیت‌های ALUOp توسط واحد کنترل اصلی تولید خواهند شد.

opcode دستور	ALUOp	عملیاتی که دستور انجام می‌دهد	فیلد Funct	عملیاتی که ALU انجام می‌دهد	خطوط کنترلی ALU
LW	00	Load word	xxxxxx	add	010
SW	00	Store word	xxxxxx	add	010
branch equal	01	Branch equal	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110

R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	Set on less than	101010	Set on less than	111

شکل ۱۸: نحوه مقدارگیری خطوط کنترل ALU بر اساس بیت‌ها ورودی ALUOp و Funct (فیلد Funct)

استفاده از چند سطح برای عملیات دیکدینگ (به این معنی که واحد کنترل اصلی بیت‌های ALUOp را تولید نماید و پس از آن ALUOp به عنوان ورودی به واحد کنترل ALU که سیگنالهای کنترلی ALU را تولید می‌کند فرستاده شود)، یک تکنیک معمول، برای پیاده‌سازی است. استفاده از چند سطح کنترل می‌تواند موجب کوچکتر شدن اندازه واحد کنترل اصلی شود. استفاده از چند واحد کنترل کوچک به جای یک واحد کنترل بزرگ ممکن است باعث بهبود سرعت واحد کنترل نیز بشود. بهینه‌سازی‌هایی از این دست، امر مهمی است به دلیل اینکه واحد کنترل معمولاً در مسیر بحرانی پردازنده قرار داشته و باعث کاهش فرکانس می‌شود.

روشهای زیادی برای طراحی این مدار که دارای دو بیت ورودی ALUOp و ۶ بیت ورودی Funct بوده و سه بیت خروجی برای کنترل ALU را تولید می‌کند، وجود دارد. به دلیل اینکه فقط تعداد کمی از ۶۴ حالت ممکن فیلد Funct در اینجا استفاده می‌شود و فیلد Funct فقط زمانی استفاده می‌شود که دو بیت ALUOp مساوی 10 باشند، بنابراین ما می‌توانیم یک مدار کوچک طراحی کنیم که این حالتهای ممکن را تشخیص داده و بیت‌های کنترلی ALU را تولید نماید.

به عنوان یک گام از طراحی این مدار، بهتر است که یک جدول صحت برای ترکیب‌های ممکن از فیلد Funct و بیت‌های ALUOp ایجاد کنیم. شکل ۱۹ این جدول را نشان می‌دهد و در آن مقادیر خطوط کنترلی ALU (operation) بر اساس ورودیها مشخص شده‌اند. به دلیل اینکه جدول صحت کامل برای این مثال خیلی بزرگ است ($۲^۸$=۲۵۶ سطر جدول) و همچنین برای ما مهم نیست که به ازای بعضی از این ترکیبات ورودی، بیت‌های کنترلی ALU چه مقداری داشته باشند، بنابراین ما در این جدول فقط سطرهایی را نشان داده‌ایم که به ازای آن خطوط کنترل ALU باید مقدار مشخص داشته باشند. در تمامی این فصل ما به این صورت عمل خواهیم کرد و فقط سطرهایی را نشان خواهیم داد که مورد نیاز می‌شوند و سطرهایی را که مقدار آنها برای ما مهم نیست را نشان نخواهیم داد. در جدول نشان داده شده در شکل ۱۹، در برخی از این سطرها مقدار بعضی از ورودیها X است. این X به معنی این است که خروجی موجود در این سطرها به مقدار این ورودی بستگی ندارد و ورودی می‌تواند هر مقداری داشته باشد. به طور مثال وقتی که بیتهای ALUOp مساوی 00 هستند (سطر اول جدول)، ما همیشه خروجی‌ها (خطوط کنترل ALU) را بدون توجه به فیلد Funct مساوی 010 قرار می‌دهیم. در این حالت بیت‌های ورودی Funct در این سطر جدول اهمیتی نخواهند داشت به همین دلیل مقدار آنها

را X قرار داده‌ایم. به محض اینکه جدول صحت آماده شد، ما می‌توانیم این جدول را ساده‌سازی نموده (به طور مثال ممکن است از جدول کارنو استفاده کنیم) و آن را تبدیل به مدار کنیم.

ALUOp		فیلد Funct						Operation (خطوط کنترل ALU)
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

شکل ۱۹: جدول صحت برای سه بیت کنترلی ALU

ما قصد داریم که این مدار (مدار واحد کنترل ALU) را طراحی کنیم. فرض کنید این مدار دارای سه خروجی جداگانه به نام‌های Operation0، Operation1 و Operation2 باشد که هر کدام از آنها متناظر با یکی از بیت‌های خروجی ستون آخر جدول شکل ۱۹ باشد. مدار لازم برای هر کدام از این خروجی‌ها با ترکیب سطرهایی از جدول که در آنها این خروجی خاص ۱ شده است، بدست می-آید. به طور مثال بیت کم ارزش خطوط کنترلی ALU (Operation0) فقط در دو سطر آخر جدول شکل ۱۹ دارای مقدار ۱ می‌باشد بنابراین جدول صحت برای Operation0 فقط این دو سطر را خواهد داشت. شکل ۲۰ جدول صحت را برای هر سه بیت کنترلی ALU نشان می‌دهد. ما از ساختار مشترک در هر جدول صحت استفاده کردیم تا اینکه حالتهای بی اهمیت بیشتری داشته باشیم. به طور مثال، ۵ سطر از جدول شکل ۱۹ که خط خروجی Operation1 را ۱ می‌کنند، فقط به دو سطر در شکل ۲۰ کاهش پیدا کرده است. مطلبی که در اینجا لازم است ذکر شود این است که حالت‌های بی‌اهمیت در مدار به این منظور استفاده می‌شوند که تعداد گیت‌های مدار و تعداد ورودی‌های لازم برای هر گیت را کاهش دهند و باعث بهبود مدار شوند. با استفاده از جدول ساده شده شکل ۲۰، ما می‌توانیم مدار شکل ۲۱ را بدست بیاوریم. ما این مدار را واحد کنترل ALU نام‌گذاری می‌کنیم. همان طور که مشاهده شد، مدار کنترل ALU، به دلیل اینکه فقط سه خروجی تولید می‌کند و برای طراحی آن فقط تعداد کمی از سطرهای جدول مورد استفاده قرار می‌گیرد، مدار ساده‌ای است. اگر تعداد ورودی‌ها و خروجی‌ها بیشتر شود طراحی واحد کنترل نیز سخت‌تر می‌شود.

ALUOp		فیلد Funct					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

(الف) جدول درستی برای Operation2=1

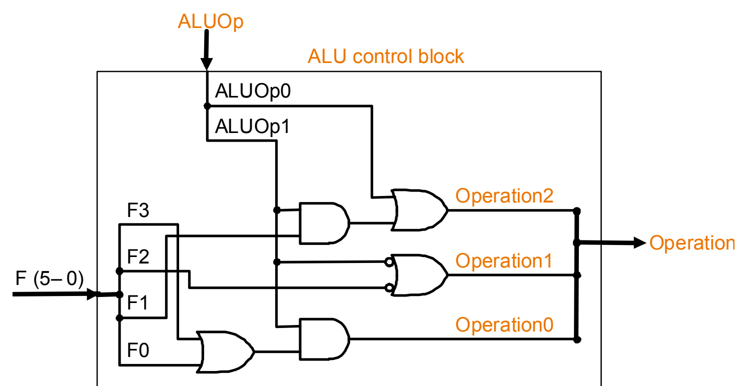
ALUOp		فیلد Funct					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

(ب) جدول درستی برای Operation1=1

ALUOp		فیلد Funct					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

(ج) جدول درستی برای Operation0=1

شکل ۲۰: جدول درستی برای بیت‌های کنترلی ALU



شکل ۲۱: مدار واحد کنترل ALU

۵-۴-۳- طراحی واحد کنترل اصلی

حال که طراحی واحد کنترل ALU را انجام دادیم، در این بخش قصد داریم که طراحی واحد کنترل اصلی که بقیه قسمتهای پردازنده را کنترل می‌کند را طراحی کنیم. برای انجام این کار اجازه دهید که فیلدهای دستور و خطوط کنترلی مسیر داده نهایی را که طراحی کردیم، دوباره بررسی کنیم. برای درک این مطلب که چگونه فیلدهای دستور را به خطوط کنترل مسیر داده وصل کنیم، بهتر است که سه فرمت دستور گفته شده برای سه نوع دستور را دوباره مرور کنیم. این سه فرمت در شکل ۲۲ نشان داده شده‌اند.

0	rs	rt	rd	shamt	funct
31-26	25-21	20-16	15-11	10-6	5-0

(الف) فرمت دستورات نوع R

35 or 43	rs	rt	address
31-26	25-21	20-16	15-0

(ب) فرمت دستورات Load و store

4	rs	rt	address
31-26	25-21	20-16	15-0

(ج) فرمت دستور beq

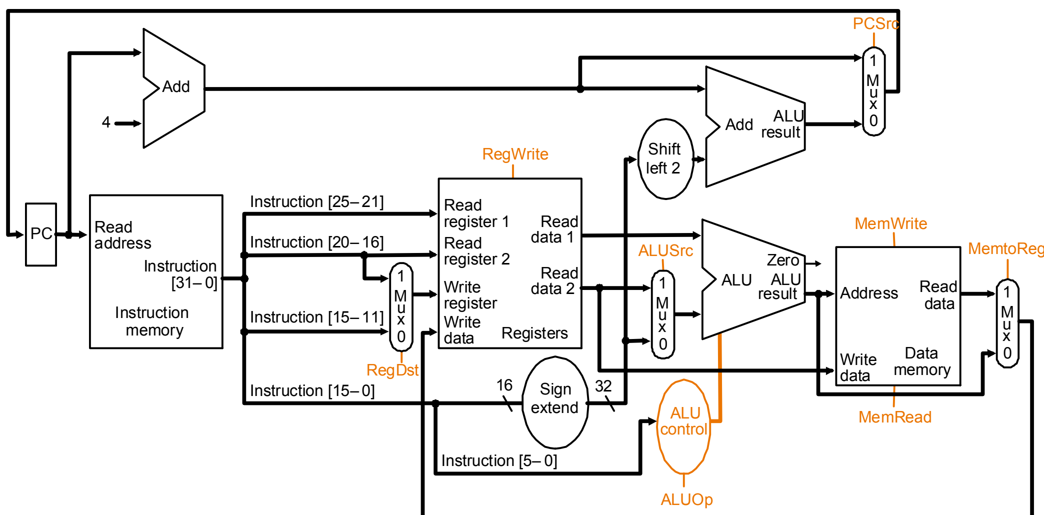
شکل ۲۲: فرمت دستورات نوع R (الف)، مراجعه به حافظه (ب) و beq (ج)

از این شکل می‌توان نکات زیر را استخراج نمود:

- فیلد OP که opcode نیز نامیده می‌شود همیشه در بیت‌های ۳۱ تا ۲۶ قرار دارد. ما به این فیلد تحت عنوان OP[5-0] مراجعه خواهیم کرد.
- دو رجیستری که باید خوانده شوند همیشه با فیلدهای rs و rt که در مکانهای ۲۵ تا ۲۱ و ۲۰ تا ۱۶ قرار دارند، مشخص می‌شود. این مطلب برای دستورهای نوع R، beq و sw درست است.
- رجیستر base برای دستورات lw و sw همیشه در مکانهای ۲۵ تا ۲۱ (rs) قرار می‌گیرد.
- عدد ثابت ۱۶ بیتی (offset) برای دستورات beq، lw و sw همیشه در مکانهای ۱۵ تا ۰ قرار داده می‌شود.

- رجیستر مقصد در یکی از دو محل قرار دارد: برای دستور lw رجیستر مقصد در مکانهای ۲۰ تا ۱۶ (rt) قرار دارد، در حالی که برای دستورهای نوع R در مکانهای ۱۵ تا ۱۱ (rd) قرار دارد. بنابراین ما نیاز داریم که از یک مالتی‌پلکسر استفاده کنیم تا مشخص کنیم که کدام فیلد دستور، شماره رجیستری را که نوشته خواهد شد، مشخص کند.

با استفاده از این اطلاعات ما می‌توانیم به مسیر داده کلی طراحی شده در شکل ۱۷، برچسب‌های دستور و یک مالتی‌پلکسر (به ورودی Write register از بانک رجیستر) اضافه کنیم. شکل ۲۳ این تغییرات را نشان می‌دهد. بعلاوه در این شکل واحد کنترل ALU، سیگنال write برای عناصر حالت، سیگنال read حافظه، و سیگنالهای کنترلی مالتی‌پلکسرها نیز نشان داده شده‌اند. به دلیل اینکه همهٔ مالتی‌پلکسرها دارای دو ورودی هستند، همهٔ آنها دارای یک خط کنترل (خط انتخاب) می‌باشند.



شکل ۲۳: مسیر داده‌ای که به آن واحد کنترل ALU، برچسب‌های دستور و یک مالتی پلکسر اضافه شده است

شکل ۲۳ دارای ۷ سیگنال کنترلی یک بیتی و یک سیگنال کنترلی دو بیتی به نام ALUOp می-باشد ما قبلاً توضیح دادیم که سیگنال کنترلی ALUOp چگونه کار می‌کند. قبل از هر کاری بهتر است کار ۷ سیگنال کنترلی دیگر را توضیح بدهیم و پس از آن به نحوه مقدار گرفتن این سیگنالها به هنگام اجرای دستور بپردازیم. شکل ۲۴ کار این ۷ سیگنال کنترلی را توضیح داده است.

اسم سیگنال کنترلی	تأثیر آن وقتی که غیرفعال می‌شود	تأثیر آن وقتی که فعال می-شود
RegDst	شماره رجیستر مقصد که به write register می‌رسد از فیلد	شماره رجیستر مقصد که به write register می‌رسد از فیلد

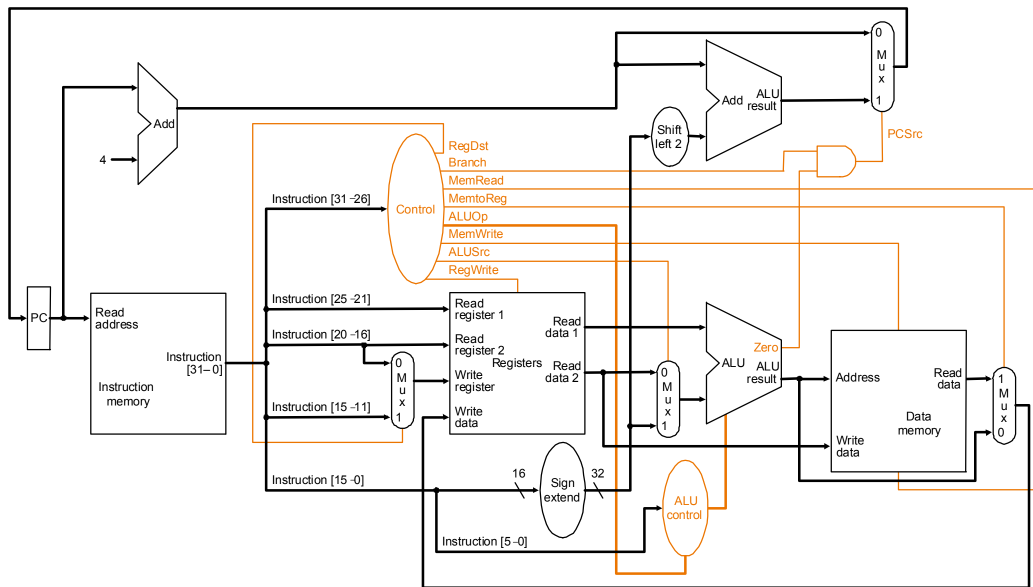
	rt (بیت‌های ۲۰ تا ۱۶) می‌آید	rd (بیت‌های ۱۵ تا ۱۱) می‌آید
RegWrite	هیچ کار (None)	در رجیستر مشخص شده با آدرس write register مقدار داده قرار گرفته بر روی خط write data نوشته می‌شود
ALUSrc	ورودی دوم ALU از خروجی دوم بانک رجیستر (Read data2) می‌آید	ورودی دوم ALU ، مقدار گسترش بیت علامت داده شده عدد ثابت ۱۶ بیتی است
PCSrc	خروجی جمع کننده‌ای که جمع PC+4 را انجام می‌دهد به PC منتقل می‌شود	خروجی جمع کننده‌ای که آدرس پرش را حساب می‌کند به PC منتقل می‌شود
MemRead	هیچ کار (None)	محتوای خانه‌ای از حافظه که آدرس آن هم‌اکنون بر روی خطوط آدرس قرار دارد، خوانده شده و بر روی خط خروجی حافظه قرار می‌گیرد
MemWrite	هیچ کار (None)	داده موجود بر روی خط write data حافظه در آدرس مشخص شده نوشته می‌شود
MemtoReg	داده‌ای که بر روی خط write data بانک رجیستر قرار می‌گیرد از ALU می‌آید	داده‌ای که بر روی خط write data بانک رجیستر قرار می‌گیرد از حافظه داده می‌آید

شکل ۲۴: تأثیر فعال شدن ۷ سیگنال کنترلی بر روی مسیر داده

حال که ما عملکرد همهٔ سیگنالهای کنترلی را مشاهده کردیم، می‌توانیم به نحوه مقداردهی آنها بپردازیم. واحد کنترل می‌تواند مقدار همهٔ این سیگنالهای کنترلی را بر اساس فیلد opcode دستور مشخص نماید. تنها استثنائی که در این زمینه وجود دارد سیگنال کنترلی PCSrc است. این سیگنال کنترلی باید زمانی یک شود که دستور beq اجرا شده و خروجی Zero از ALU که برای مقایسه

تساوی به کار می‌رود یک گردد. برای تولید سیگنال PCSrc ما نیاز داریم سیگنال Zero را با یک سیگنال از خروجی‌های واحد کنترل به نام Branch ، AND کنیم.

این ۹ سیگنال کنترلی (۷ سیگنال شکل ۲۴ و دو سیگنال ALUOp) می‌توانند بر اساس ۶ بیت ورودی واحد کنترل که همان ۶ بیت opcode است، مقداردهی شوند. شکل ۲۵ مسیر داده طراحی شده را به همراه واحد کنترل و سیگنالهای کنترلی نشان می‌دهد. قبل از اینکه ما تلاش کنیم که برای این کنترل معادله‌ای بنویسیم یا جدول صحتی تشکیل دهیم، بهتر است که عملیات کنترلی را توضیح دهیم. به دلیل اینکه مقداردهی خطوط کنترلی فقط به opcode وابسته است، ما بر اساس مقدار opcode تعیین می کنیم که مقدار سیگنالهای کنترلی باید ۰ ، ۱ یا X باشد. شکل ۲۶ مشخص می کند که چگونه سیگنالهای کنترلی بر اساس فیلد opcode مقدار می گیرند.



شکل ۲۵: مسیر داده ساده همرا با واحد کنترل

دستور	RegDst	ALUSrc	MemtoReg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
نوع R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

شکل ۲۶: مقدار سیگنالهای کنترلی توسط فیلد opcode دستور مشخص می‌شود

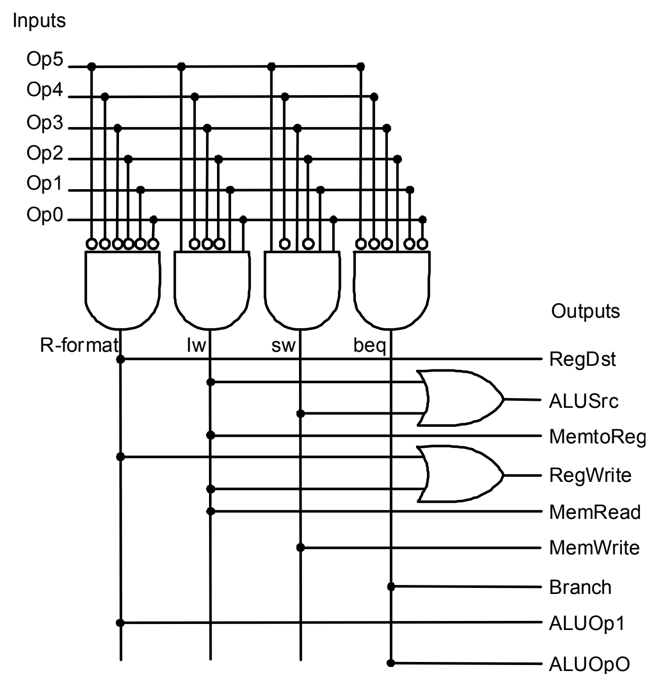
واحد کنترل می‌تواند با استفاده از محتویات شکل ۲۶ به طور دقیق طراحی شود. واحد کنترلی که ما طراحی می‌کنیم، مداری است که دارای ۶ بیت ورودی (فیلد opcode) بوده و در خروجی خود سیگنالهای کنترلی مورد نیاز را تولید می‌نماید. برای طراحی این مدار باید جدول صحت را برای هر کدام از خروجی‌ها تشکیل دهیم. قبل از رسم جدول صحت بهتر است که opcode دستورهای پیاده-سازی شده را شرح دهیم. جدول زیر مقدار فیلد opcode را برای هر کدام از دستورات هم به صورت دهدهی و هم به صورت باینری نشان می‌دهد:

اسم دستور	opcode دستور به صورت دهدهی	opcode دستور به صورت باینری					
		Op5	Op4	Op3	Op2	Op1	Op0
نوع R	0	0	0	0	0	0	0
lw	35	1	0	0	0	1	1
sw	43	1	0	1	0	1	1
beq	4	0	0	0	1	0	0

با استفاده از اطلاعات این جدول ما می‌توانیم عملکرد واحد کنترل را با یک جدول صحت بزرگ که در شکل ۲۷ نشان داده شده است، توضیح دهیم. این شکل عملکرد واحد کنترل را به طور کامل شرح می‌دهد و ما می‌توانیم با استفاده از آن مدار واحد کنترل را با گیت‌های منطقی طراحی کنیم. اگر از همان روشی که برای طراحی واحد کنترل ALU استفاده کردیم، در اینجا نیز استفاده کنیم، مدار شکل ۲۸ برای واحد کنترل اصلی بدست می‌آید. از آنجایی که هر کدام از خروجی‌ها فقط با استفاده از گیت‌های AND ، OR و به صورت دو طبقه طراحی شده‌اند، می‌توانیم در طراحی واحد کنترل اصلی از آرایه‌های منطقی قابل برنامه‌ریزی یا اصطلاحاً PLA استفاده کنیم. مدار شکل ۲۸ با استفاده از PLA طراحی شده است.

ورودی و خروجی	اسم سیگنال	نوع R	lw	sw	beq
ورودی‌ها	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
خروجی‌ها	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

شکل ۲۷: عملکرد واحد کنترل پردازنده single cycle که به طور کامل با جدول صحت نشان داده شده است



شکل ۲۸: مدار واحد کنترل پردازنده single cycle