تمرین ۴

سید علیرضا غضنفری

00-01 FALL

Contents

سوال ۱

شیوه آدرس دهی شبه مستقیم یک نوع دستور ۱ آدرس میباشد که علاوه بر opcode تنها شامل یک آدرس است. این تک آدرس تقریبا بخش نسبتا بزرگی از آدرس موثر عملوند را به ما نشان میدهد ولی بدلیل محدودیت های بیتی که وجود دارد نمیتواند ادرس موثر کامل را در خود جای دهد پس از روش هایی طراح ها استفاده میکنند تا این بخش بزرگ ادرس را کامل کنند. به این نوع ادرس دهی شبه مستقیم میگوییم.

در MIPS طول دستورات ۳۲ بیت میباشد و از انجایی که ۶ بیت از آن ۳۲ بیت برای opcode است پس تنها ۲۶ بیت برای بخش ادرس میماند. برای داشتن یک ادرس موثر کامل به منظور پیدا کردن operand نیاز داریم تا یک ادرس ۳۲ بیتی داشته باشیم. از انجایی که کلمات در MIPS ۳۲ بیتی هستند و در واقع ۴ بایت هستند پس هر ۴ بایت یک ادرس از حافظه را دارند. پس این ادرس موثر کاملی که باید تشکیل دهیم ۴ ادرس به ۴ ادرس باید مقدارش زیاد شود. نتیجتا میتوان دو بیت ۰ به منظور همین افزایش ۴ واحدی به سمت راست عدد ۲۶ بیتی اضافه کرد و حاصل را یک ادرس ۲۸ بیتی کرد. حال ۴ بیت مابقی را بدین شکل انتخاب میکنیم که ۴ بیت پرارزش تر pc را برداشته و به عنوان ۴ بیت پرارزش تر این ادرس ۳۲ بیتی قرار میدهیم و ۲۸ بیت بقیه را هم از همان محاسبات قبلی جایگزین میکنیم.

با توجه به توضحیات بالا باید این را در نظر داشت که به دلیل اینکه در این پردازنده بنا بر طراحی که شکل گرفته لازم بوده است که ۶ بیت از هر دستور به opcode اختصاص داده شود پس نمیتوان یک عدد ۳۲ بیت مستقیم را در دستور قرار داد و برای همین در این پردازنده از سیستم ادرس دهی شبه مستقیم استفاده میشود.

سوال ۲

در ابتدا باید به انواع دستوراتی که در این پردازنده ها وجود دارد مراجعه کنیم. وقتی دستورات را مشاهده میکنیم که دستورات R-type که در این پردازنده ها وجود دارد در واقع ۵ بیت اضافه تحت عنوان function دارد. این ۵ بیت علاوه بر آن ۶ بیتی است که همه ی انواع دستورات این معماری برای opcode دارند هست که به دو دلیل تعبیه شده است. دلیل اول میتواند در آینده برای توسعه دستورات پردازنده مورد استفاده قرار گیرد و دلیل دوم نیز تعداد بالا دستوارت R-type است که از حدی که بیت های opcode میتوانند ایجاد کنند فراتر رفته است پس نیاز به چند بیت اضافه تر وجود دارد تا بتوان همه دستورات مورد نیاز را طراحی کرد.

از طرفی در طراحی پردازنده این را هم باید در نظر داشته باشیم که واحد کنترل به خودی خود و برای تولید همان سیگنال های کنترلی مربوط به opcode بسیار شلوغ است پس تا میشود علاقه داریم تا واحد کنترل را نیز ساده سازی کنیم تا فهم کار ها و طراحی واحد های پردازنده ساده تر باشد. بدین منظور برای ایجاد سیگنال هایی که مستقیما به ALU خواهند رفت و فقط روی محاسبات ما تعیین کننده هستند میتوان از یک واحد کنترلی جدا استفاده کرد که یک سری سیگنال ورودی دارد که از واحد کنترل اصلی میایند و یک سری سیگنال نیز دارد که مستقیما از دستور خوانده میشوند (function) و در نهایت به یک دسته سیگنال کنترلی خروجی میرسد که انها را به ALU میدهد تا مشخص کند که در این زمان ALU باید چه عملیاتی را انجام دهد.

فرایندی که این بخش انجام میدهد نیز بدین صورت است که ۶ بیت ورودی function دارد که مستقیم از دستورات میگیرد و دو بیت نیز (برای مثال است این تعداد بیت بستگی به این دارد که در طراحی پردازنده ای که مورد بررسی است چند حالت از حالات opcode را به دستورات محاسباتی منطقی اختصاص دهند) به عنوان سیگنال های کنترلی با نام ورودی میگیرد به صورتی که این سیگنال ها در واقع خروجی های واحد کنترل اصلی پردازنده هستند. از طرفی هم ۴ سیگنال (التبه این تعداد هم بستگی به تعداد عملیات هایی داد که ALU مورد نظر میتواند اجرا کند) خروجی کنترلی برای ALU دارد که این ۴ سیگنال مستقیما وارد ALU میشوند به عنوان ورودی هایی که operation را تعیین میکنند.

یکی از مزیت هایی که این روش دارد این است که خوب واحد کنترل ساده تر شده است و فهم کار و نوع طراحی آن ساده تر میباشد و ساده تر میتوان آن را پیاده کرد. این امر سبب این میشود که توسعه این پردازنده و دستوراتش نیز ساده تر باشد و برای توسعه آن مجبور نشویم که طراحی کل واحد کنترل را تغییر دهیم.

از کاستی هایی که اضافه کردن این بخش به پردازنده دارد نیز خوب این است که باید تعدادی از بیت های دستورهایمان را اختصاص دهیم به این بخش و طول دستور ها کوتاه تر میشود در نتیجه تعداد ثبات های کمتری را در اختیار داریم تا با انها پردازنده طراحی کنیم.

سوال ۳

دستور ۳ آدرسه

تعداد = ۱۵

opcode	Address1	Address2	Address3
4	4	4	4

در این دستور ها تعداد ۱۵ تا است ولی با ۴ بیت میتوان ۱۶ حالت بدست اورد پس یک حالت اضافی داریم که آن را ۰۰۰۰ فرض کنید.

دستور دو آدرسه

تعداد = ۱۴

0000	Opcode-part2	Address1	Address2
4	4	4	4

در این دستورها تعداد ۱۴ تا است ولی با ۴ بیتی که در بخش دوم opcode داریم ۱۶ حالت را دارد پس دو حالت اضافه داریم. پس برای بخش دوم opcode نیز دو حالت اضافی را فرض میکنیم و جدا میکنیم برای دستورات بعدی از جمله ۰۰۰۰ و ۰۰۰۱.

دستور تک آدرسه

تعداد = ۳۱

0000	(0000 or 0001)	Opcode-part3	Address1
4	4	4	4

در این دستور ها تعداد ۳۱ تا است ولی با توجه به جدول بالا برای بخش دوم opcode دو حالت داریم و برای بخش سوم نیز ۴ بیت داریم که در نتیجه ۳۲ حالت ایجاد میکند پس در اینجا نیز یک حالت اضافه است که آن را 0000-0001-0000 در نظر میگیریم.

دستور بدون آدرس

تعداد = ۱۶

0000	0001	0000	Opcode-part4
4	4	4	4

در این دستور ها نیز تعداد باید ۱۶ تا باشد و ۴ بیت آزاد داریم برای تعیین دستورات که خوب ۱۶ حالت را شکل میدهند پس کدگذاری ما به درستی به پایان یافت.

سوال ۴

(a) LdI 20

Acc = 20

(b) LdA 20

Acc = 40

(c) LdInd 20

Acc = 60

(d) LdInd 30

Acc = 70

سوال ۵

$$T_2: R_B \leftarrow R_B - 1$$

$$T_0.NOR(R_C): R_C \leftarrow \overline{R_A} + \overline{R_B} + 1$$

$$T_0.OR(R_C): R_C \leftarrow R_C + R_B$$

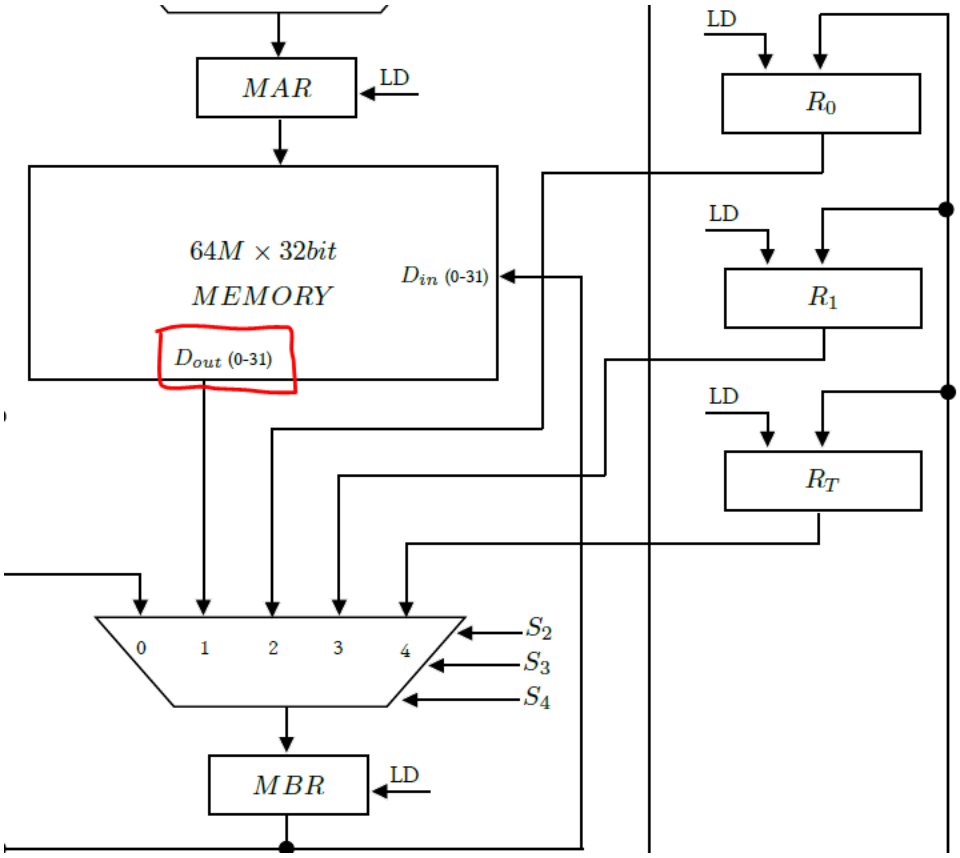
$$T_1.NOR(R_C): R_A \leftarrow R_C$$

$$T_1.OR(R_C): R_A \leftarrow R_B$$

سوال ۶

توضیح مختصر درباره اندازه هر کدام از ثبات ها و مالتی پلکسر هایی که در تصویر استفاده شده است:

همانطور که در این تصویر میبینید مالتی پلکسر ورودی ۳۲ بیتی از حافظه دارد پس میتوان گفت که تمام ورودی های این مالتی پلکسر ۳۲ بیت دیتا دارند و خروجی آن نیز ۳۲ بیتی است پس ثبات MBR نیز ۳۲ بیتی است و ثبات های R_0, R_1, R_T نیز ثبات هایی ۳۲ بیتی هستند.



هر خانه حافظه بنابر اندازه ای که بر روی خودش ثبت شده است ۳۲ بیت میباشد و تعداد خانه های حافظه 64M است پس 64×2^{20} که همان 2^{26} خانه در حافظه موجود است پس نتیجتا به ۲۶ بیت نیاز است تا یک خانه مشخص را نشان داد.

این طراحی برخلاف بیشتر کامپیوتر های حال حاضر که از مدل ها و طراحی های چارلز بابیج استفاده میکنند و حافظه دستور و دیتای آنها از یکدیگر جدا است پیروی نمیکند.

یک نکته ای که باید برای طراحی این تصویر در نظر گرفت اختلاف ۶ بیتی است که ثبات ها و آدرس مورد نظر دارند که خوب به عنوان طراحی از این قضیه نمی توان ناراحت بود زیرا ما نه تنها برای نشان دادن خانه های حافظه بیت کم نداریم بلکه اضافه نیز داریم و هرگاه که خواستیم میتوانیم حجم حافظه را افزایش بدهیم.

در این طراحی فرض را بر این گذاشته ام که در ثبات هایی که قرار است آدرس خانه ای از حافظه را تامین کنند ۶ بیت پرارزش ۰ را دارا باشند(به عبارتی ۶ بیت پرارزش آنها اصلا مورد توجه قرار نخواهد گرفت).

پس دستوراتی که آدرس خانه ای از حافظه را در ثبات گذاشته اند و میخواهند ارجاع دهند باید این را در نظر داشته باشیم که ۶ بیت پرارزش آنها ارزش ندارند.

بدین ترتیب یک اختلاف ۶ بیتی بین طول دیتایی که در ثبات MAR هست و دیتا های ورودی مالتی پلکسر بالایی بوجود آمده است که با استفاده از تکنیک zero filling این اختلاف را جبران میکنیم بدین صورت که ۶ بیت اضافه 0 را برای ثبات PC و ورودی بخش operand ثبات IR در نظر میگیریم تا آنها را نیز به طول ۳۲ بیت برسانیم.

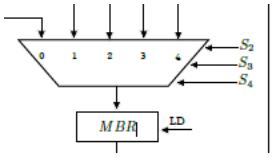
(آ)

دستور اول: $R_T <- R_1 + R_0$

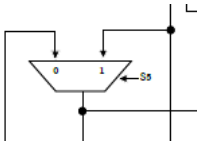
همانطور که مشخص است این دستور یک دستور سه ادرسه است که شناسه های سه ثبات را در خود دارد.

در این شکل واحد کنترل موجود نیست پس دستورات کنترلی را خودمان فرض میکنیم که بطوری وارد میکنیم تا دستور های مورد نظر را ایجاد کنیم.

برای این دستور ابتدا (S2S3S4) را ۰۱۰ ست میکنیم تا از این مالتی پلکسر با گذشت یک کلاک ساعت مقدار دیتای ۳۲ بیتی ثبات R0 در MBR قرار بگیرد.



حال s5 را نیز ۱ در نظر میگیریم تا خروجی این بخش نیز برابر بشود با دیتای ۳۲ بیتی ثبات R0



کاری که این مالتی پلکسر میکند در واقع این است که یا یک عملیات جدید با ورودی جدید و خروجی عملیات قبلی را مهیا میکند یا اینکه یک عملیات جدید بین دو ورودی یکسان را فراهم میسازد.

پس با ست کردن سه بیت کنترلی F2F1F0 به ۰۰۱ میتوان عملیات تفریق با دو ورودی یکسان را عملی کرد و خروجی واحد محاسبات را به ۰ رساند آنگاه با فعال کردن لود ثبات ACC این مقدار را در آن ثبات نیز قرار میدهیم.

حال که مقدار خروجی عملیات قبلی مان ۰ است پس S5 را ۰ میکنیم و فرمان عملیات واحد محاسبات را هم به جمع تغییر میدهیم تا دوباره با فعال شدن لود ثبات ACC مقدار دیتای R0 در آن قرار بگیرد.

حال در مالتی پلکسر اول تغییراتی در دستور(بیت های سلکت) ایجاد میکنیم تا با فعال شدن دوباره لود ثبات MBR دیتای R1 در آن قرار بگیرد.

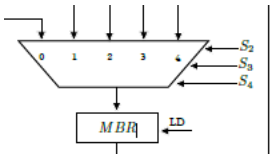
حال در این لحظه یکی از ورودی های واحد محاسبات R1 و دیگر ورودی واحد محاسبات نیز R0 میباشد و با تغییر دستور ورودی واحد محاسبات به عملیات جمع میتوانیم حاصل جمع این دو دیتا را در خروجی واحد محاسبات داشته باشیم و با فعال سازی لود این مقدار در ثبات ACC قرار میگیرد.

تنها بخش باقی مانده این دستور این است که مقدار لود ثبات مقصد را فعال کنیم تا دیتای ثبات ACC در آن ذخیره شود.

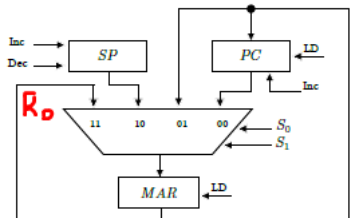
دستور دوم: $load\ R_T\ ,\ (R_0)$ Or $R_T <- M[R_0]$

برای اجرای این دستور باید به نحوی مقدار دیتای ۳۲ بیتی ثبات R_0 را به عنوان ادرس operand در ثبات MAR قرار دهیم تا بتوانیم operand را از حافظه استخراج کنیم و مورد استفاده قرار بدهیم.

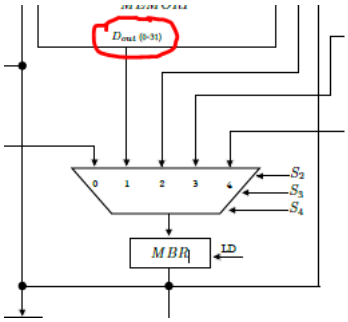
پس ابتدا کافیست که در این مالتی پلکسر بیت های سلکت طوری وارد شوند که مقدار دیتای R_0 به عنوان خروجی از این مالتی پلکسر خارج شود و سپس باید ورودی لود ثبات MBR را فعال کنیم.



حال باید ورودی مالتی پلکسر پایین را ۱۱ ست کنیم تا مقدار خروجی ثبات MBR را به خروجی خود ببرد و سپس مقدار ورودی لود ثبات MAR را فعال میکنیم تا این دیتا در آن ثبات قرار بگیرد.



حال مقداری که در D_{out} نشان داده میشود مقدار $M[R_0]$ میباشد پس کافیست که در مالتی پلکسر پایین بیت های سلکت را به گونه ای انتخاب کنیم که پس از فعال کردن ورودی لود ثبات MBR مقدار D_{out} در آن قرار بگیرد.



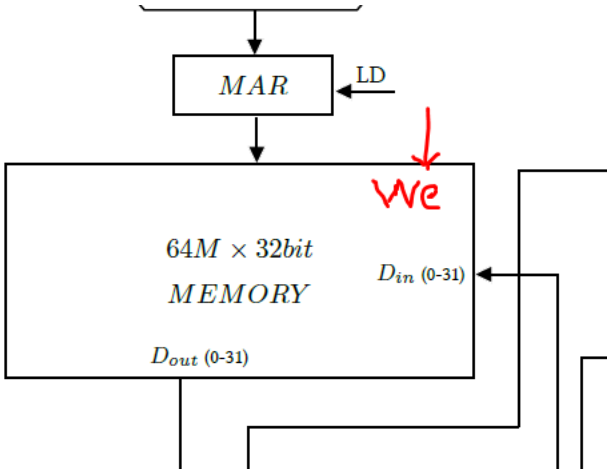
حال تنها کاری که کافیست انجام دهیم این است که ورودی سلکت این مالتی پلکسر را ۱ دهیم تا هر دو ورودی واحد محاسبات D_{out} باشد و سپس دستور های ورودی واحد محاسبات را بطوری ست میکنیم تا عملیات منطقی and بین دو ورودی یکسان شکل بگیرد.

پس مقدار خروجی ALU همان دیتای D_{out} است و با فعال کردن لود ثبات ACC این مقدار در آن ثبات قرار داده میشود و تنها با فعال کردن لود R_T میتوانیم این دیتا را در ثبات مورد نظر بریزیم.

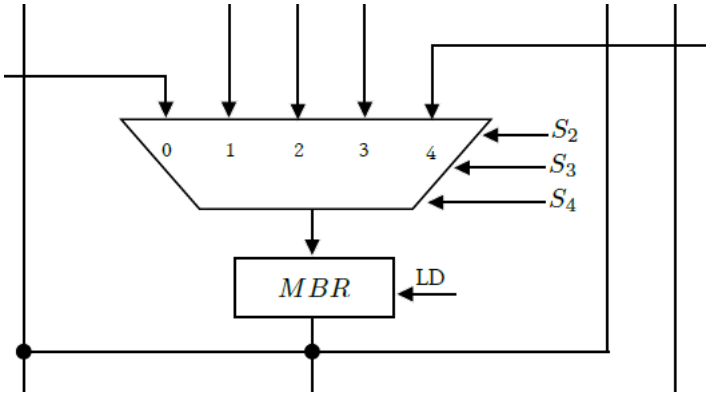
دستور سوم: str R_0 , (R_1)

این دستور برای ذخیره کردن دیتای درون ثبات ۰ در خانه ای از حافظه است که ادرس آن همان دیتای درون ثبات یک میباشد.

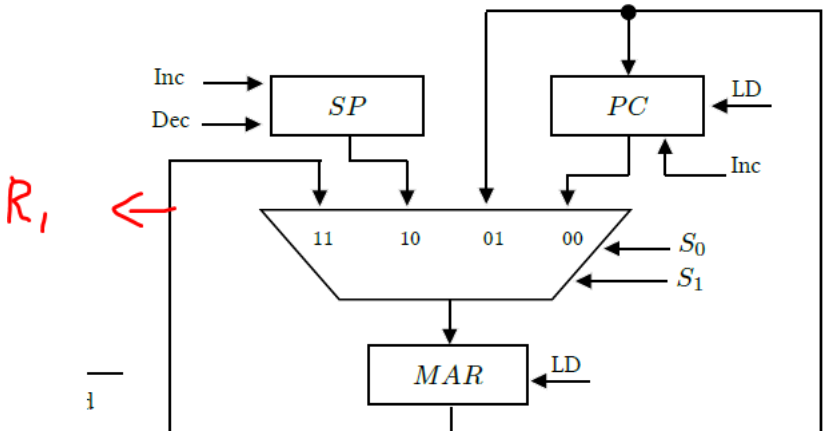
برای این دستور نیاز داریم که یک سیگنال کنترلی به تصویر اضافه کنیم در بخش زیر تا هنگامی که فعال شد دیتای ورودی را درون آدرسی از حافظه که در MAR هست ذخیره کند.



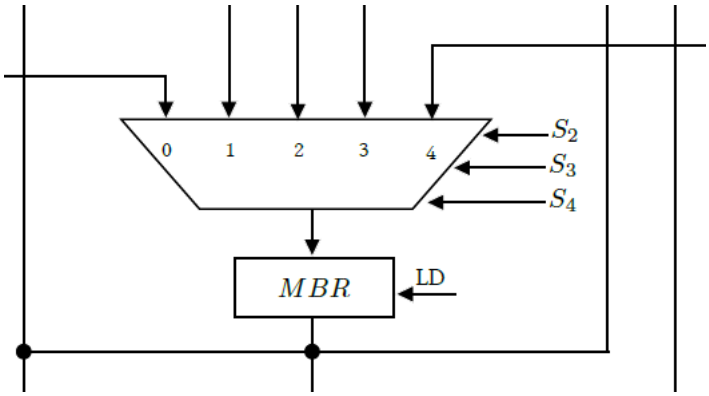
برای اجرای این دستور ابتدا باید مقدار دیتای درون ثبات یک را درون ثبات MAR بریزیم پس ابتدا در این مالتی پلکسر به نحوی ورودی های سلکت را وارد میکنیم تا خروجی آن مقدار دیتای ثبات یک باشد و با فعال شدن لود ثبات MBR این مقدار درون آن ثبات قرار بگیرد.



حال باید در مالتی پلکسر پایین طوری ورودی های سلکت را معین کنیم تا این دیتا که در ثبات MBR موجود است به عنوان خروجی مالتی پلکسر انتخاب شود و سپس با فعال شدن لود ثبات MAR درون این ثبات قرار گیرد.



حال گام اول اجرای این دستور با موفقیت به پایان رسید و آدرسی که میخواستیم را در ثبات MAR قرار دادیم و تنها کاری که باقیمانده است این است که دیتای ثبات ۰ را در مقام D_{in} قرار دهیم.



این کار بسیار ساده است و تنها کافیست که در همان مالتی پلکسر اول طوری ورودی های سلکت انتخاب شوند که خروجی مالتی پلکسر برابر باشد با دیتای ثبات ۰ و سپس ورودی لود ثبات MBR را فعال میکنیم و در این هنگام است که ورودی D_{in} را به درستی ست کردیم و تنها کاری که کافیست انجام دهیم این است که ورودی We حافظه را فعال کنیم تا بتوان دیتای ثبات ۰ را در ادرسی که در ثبات ۱ هست قرار دهیم.

دستور چهارم: J L_1

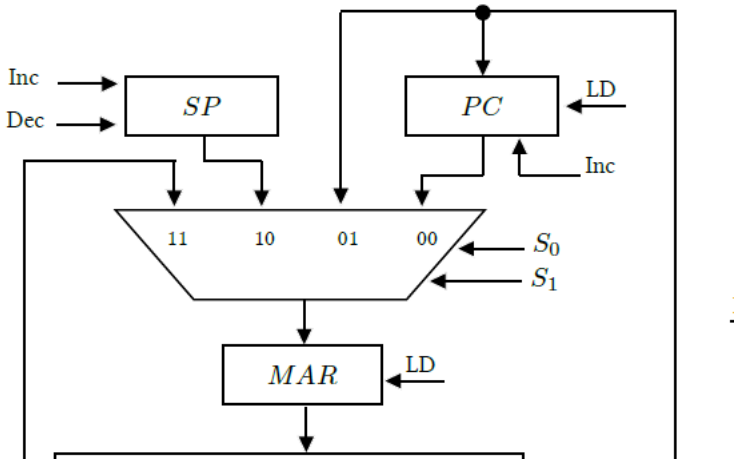
این دستور از دستور های یک آدرسه است و بصورت مستقیم و بدون هیچ حالت نسبی بصورت مطلق به آدرسی که در دستور آمده است میرود و آن خط را اجرا میکند.

برای اجرای این دستور باید مقدار PC را تغییر دهیم و دیتایی (آدرسی) که در دستور آمده است را در آن قرار دهیم.

میدانیم که دستور مورد نظر الان در حافظه است و PC نیز به آن اشاره میکند پس ۳۲ بیت دستور را در جایگاه D_{out} داریم. در این مالتی پلکسر به نحوی ورودی های سلکت را انتخاب میکنیم تا D_{out} را به خروجی مالتی پلکسر برسانیم.

با فعال شدن ورودی لود ثبات MBR این دیتا را در آن ثبات میریزیم. سپس ورودی لود ثبات IR را فعال میکنیم تا این دیتا در ثبات IR قرار گیرد.

حال همانطور که در شکل نیز مشخص است ۲۶ بیت کم ارزش تر را به عنوان operand از ثبات میگیریم و به عنوان ورودی به ثبات PC به عنوان ورودی میدهیم. ورودی لود را فعال میکنیم تا این ۲۶ بیت دیتا درون این ثبات قرار گیرد و ورودی سلکت مالتی پلکسر زیر را طوری مشخص میکنیم تا خروجی ثبات PC را به خروجی مالتی پلکسر برسانیم و با فعال کردن ورودی لود ثبات MAR نیز میتوان این ادرس را در آن ثبات لود کرد.



بدین صورت این دستور اجرا شد. البته باید در نظر داشته باشیم که در این تصویر واحد کنترل نشان داده نشده است ولی شکل کاملتر این است که این ورودی های سلکت و دستور را واحد کنترل با استفاده از دستوراتی که میگیرد مشخص میکند.

دستور پنجم: push R_0

این دستور در واقع دسترسی به بخشی از حافظه که آنرا تحت عنوان استک میدانیم دارد.

برای اجرای این دستور باید ابتدا D_{in} را به مقدار دیتای درون ثبات ۰ ست کنیم. سپس بدنبال این باشیم که راهی بیابیم تا آدرس ثبات SP را درون ثبات MAR قرار دهیم و پس از آن ورودی We حافظه را فعال کنیم تا دیتای D_{in} درون آدرس مشخص شده در حافظه نوشته شود.

پس ابتدا در مالتی پلکسر ورودی های سلکت را به گونه ای وارد میکنیم تا دیتای ثبات SP در خروجی مالتی پلکسر ظاهر شود و سپس ورودی لود ثبات MAR را فعال میکنیم و در همین زمان بصورت همزمان ورودی Inc ثبات SP را نیز فعال میکنیم تا آدرسی که در ثبات SP هست دوباره به پایین ترین خانه خالی در استک اشاره کند.

دستور ششم: pop R_0

در این دستور ابتدا باید سیگنال ورودی Dec ثبات SP را فعال کند (به اندازه یک کلاک) تا آدرسی که درون این ثبات موجود است یک واحد کاهش یابد و آدرس درون این ثبات به بالاترین خانه پر استک اشاره کند. سپس بیت های ورودی سلکت مالتی پلکسر را طوری وارد میکنیم تا دیتای ثبات SP را به خروجی مالتی پلکسر ببرد و با فعال کردن لود برای یک کلاک این دیتا را در ثبات MAR قرار میدهیم.

با ست شدن دیتای این ثبات در همان کلاک مقدار پورت خروجی D_{out} نیز مقدار دیتای بالاترین خانه استک هست. حال D_{out} را با استفاده از بیت های مشخص سلکت برای مالتی پلکسر پایین تصویر به عنوان ورودی به واحد محاسبات میدهیم.

نکته طراحی:

با توجه به اینکه عملیات mov از عملیات های بسیار پرکاربرد است و به این صورت است که دیتای یک ثبات را به ثبات دیگر منتقل میکنیم پس تصمیم بر این گرفتم تا دستور pass در واحد محاسبات به این گونه باشد که ورودی راست را مستقیم به خروجی ببرد.

با استفاده از دستور pass در واحد محاسبات این دیتا را به خروجی ALU میبریم و با یک کلاک و فعال کردن ورودی لود ثبات ACC دیتا را در آن ثبات ذخیره میکنیم.

حال با یک کلاک دیگر و فعال کردن ورودی لود ثبات مقصد میتوانیم اجرای این دستور را به پایان برسانیم.

فقط دو بخش دیگر از تصویر باقیمانده است که توضیح داده نشده اند که یکی از آنها ورودی Inc ثبات PC است که واحد کنترل با قرار دادن PC در ثبات MAR بلافاصله این ورودی را فعال میکند تا به دستور بعدی ارجاع دهد.

بخش دیگری که توضیح داده نشده است یکی از ورودی های مالتی پلکسر بالایی است که از خروجی ثبات IR تامین میشود.

از انجایی که با استفاده از این ورودی در واقع یک آدرس مطلق که در دستور آمده است صرفا به عنوان یک آدرس استفاده میکنیم و به عنوان مقصد PC آنرا نمی بینیم میتوان به دستوراتی فکر کرد که صرفا مقصد مشخصی مثل ثبات ACC دارند و مبدا آنها حافظه میباشد.

برای مثال دستور هفتم: $Id\ L_2$

در این دستور L_2 یک آدرس ۲۶ بیتی مطلق است. وقتی PC به این دستور از حافظه اشاره میکند و با ست کردن انواع سلکت ها و دستورات آنرا در ثبات IR قرار میدهد آدرس L_2 را با استفاده از سلکت های مالتی پلکسر دومی بصورت مستقیم به ثبات MAR میرسانیم. در نتیجه دیتای آن آدرس را به پورت D_{out} منتقل میکنیم و با ست کردن بیت های سلکت مالتی پلکسر پایین و ست کردن دستور pass برای واحد محاسبات این دیتا را در ثبات ACC قرار میدهیم.

(ب)

خوب در ابتدا این دستور صرفا به شکل یک رشته ۳۲ بیتی در حافظه است تا اینکه PC به آدرس خانه این دستور میرسد. در پالس اول ورودی لود ثبات MAR فعال میشود و ورودی های سلکت مالتی پلکسر بالا نیز به نحوی هستند که خروجی ثبات PC را به خروجی مالتی پلکسر منتقل کنند پس در پالس اول آدرس دستور در ثبات MAR قرار میگیرد.

پالس دوم: پس از تغییراتی که در پالس قبلی داشتیم الان رشته باینری ۳۲ بیتی دستور ما در پورت D_{out} قرار دارد. پس با یک کلاک و فعال کردن لود ثبات MBR و البته ورودی های مشخص مالتی پلکسر پایین این دیتا در ثبات معین قرار میگیرد.

پالس سوم: در این پالس این دستور در ثبات IR قرار میگیرد و بدین صورت از مرحله fetch به مرحله decode میرسد. دستور در این زمان دیکود میشود و عملگر های آن مشخص میشوند.

پالس چهارم: واحد کنترل که حال میداند باید چه دستوری را اجرا کند و سیگنال کنترلی S_0 را 1 و سیگنال کنترلی S_1 را 0 ست میکند تا در مالتی پلکسر بالایی دیتای operand ثبات IR به خروجی این مالتی پلکسر برسد. از طرفی واحد کنترل پس از دیکود دستور حال میداند که باید ثبات R_0 را نیز در ثبات ACC قرار دهد.

پس در همین زمان بصورت همزمان ($S_4S_3S_2$) را بصورت 010 وارد میکند تا دیتای ثبات R_0 را به خروجی مالتی پلکسر پایینی برساند.

وقتی کلاک چهارم فرامیرسد بدین ترتیب دیتای operand در ثبات MAR و دیتای R_0 در ثبات MBR قرار میگیرد. واحد کنترل کد های عملیات واحد محاسبات را هم مطابق دستور pass وارد میکند تا مقدار دیتای ثبات R_0 به خروجی واحد محاسبات برسد.

پالس پنجم: در این زمان با فرارسیدن لبه بالا رونده کلاک مقدار دیتای ثبات R_0 در ثبات ACC قرار خواهد گرفت. و از طرف دیگر ورودی های ($S_4S_3S_2$) به نحوی از سمت واحد کنترل وارد میشوند که دیتای پورت خروجی حافظه را به خروجی مالتی پلکسر پایین برساند یعنی 001 وارد شود. وقتی که این کلاک فرا میرسد دیتای بخش آدرس دستور به ثبات MAR منتقل میشود در نتیجه خروجی حافظه میشود [Mem_Address] که مد نظر ما است.

پالس ششم: با این کلاک [Mem_Address] در ثبات MBR قرار میگیرد و واحد کنترل فرمان عملیات واحد محاسبات را به جمع تغییر میدهد و s_5 هم 0 میکند تا در خروجی واحد محاسبات داشته باشیم R_0 + [Mem_Address].

پالس هفتم: در این پالس حاصل محاسبات با فعال شدن ورودی لود ثبات ACC در این ثبات قرار میگیرد و اماده انتقال به مقصد است.

پالس هشتم: دیتای درون ثبات ACC با فعال شدن ورودی لود در ثبات R_0 درون این ثبات قرار میگیرد.

سوال ۷

(الف)

همانطور که گفته شده است اندازه ثبات IR برابر ۱۶ بیت است پس طول دستورات در این طراحی ۱۶ بیت است و همانطور که قالب دستورات گفته شده است که بصورت یک آدرسه است با 4 بیت برای opcode که خوب به این معنی است که 2^4 حداکثر تعداد دستوراتی است که در این طراحی میتوانیم داشته باشیم.

با توجه به ISA ای که برای پردازنده داریم میدانیم که این پردازنده ۹ دستور دارد.

مد های آدرس دهی ممکن در این پردازنده :

Memory direct addressing mode – implicit addressing mode – immediate addressing mode

هر کلمه در حافظه این پردازنده ۲ بایت است و حجم این حافظه 4096 × 16 bit میباشد که در واقع 64K bit میباشد.

تنها راه فرستادن دیتای یک خانه از حافظه به پورت خروجی حافظه این است که آدرس آن خانه را در ثبات MAR قرار دهیم. فکر میکنم قسمت ALU شکل پردازنده نیز دارای مشکل میباشد زیرا همانطور که در این پردازنده خروجی ALU همواره به ورودی ثبات ACC متصل است یکی از ورودی های ALU نیز به این ثبات باید متصل باشد که هست اما جهت فلش این اتصل اشتباه است(البته میتواند اتصال پایینی نیز دیگر ورودی واحد محاسبات باشد که خوب در این صورت نیز فلش اصلا ندارد).

ثبات های OutReg, InReg ۸ بیتی هستند که به این دلیل است که سرعت عمل سمت کاربر بسیار پایین تر از سرعت پردازنده هست پس به همین دلیل پردازنده ترجیح میدهد که کاراکتر به کاراکتر از input , output buffer بگیرد و در ثبات های خود قرار دهد.

اما در مورد ورودی های ALU گفتیم یکی از آنها لزوما دیتای درون ثبات ACC است و خروجی این واحد نیز در همین ثبات قرار خواهد گرفت ولی ورودی دیگر این واحد از ثبات MBR است که در واقع به پورت خروجی حافظه متصل است ولی در این پردازنده میتوانیم حتی دیتای ACC را نیز در این ثبات بریزیم.

این معماری از یک گذرگاه ۱۶ بیتی مشترک استفاده میکند و حافظه آن نیز word addressable است. از انجایی که هم مموری دارد و هم تعدادی ثبات و واحد محاسبات پس میتوان گفت که این معماری تمام نیاز های ضروری برای داشتن عملکردی همچون یک کامپیوتر حقیقی را دارا است. در هر فاز نیز گذرگاه آن تنها از یکی از این بخش ها دیتا را برداشته و به مقصد منتقل میکند در نتیجه انتقال دیتای همزمان بین چند ثبات و مموری نخواهیم داشت.

یک نکته ای که در تصویر گفته شده ذخیره آدرس بازگشت در اولین خانه زیر برنامه هست که خوب این امکان را به ما میدهد تا زیربرنامه در این پردازنده بتوانیم تعریف کنیم و هرگاه به پایان یک زیربرنامه رسیدیم با مراجعه به خانه اول زیربرنامه آدرس دستوری که باید پس از آن اجرا کنیم را داریم.

(ب)

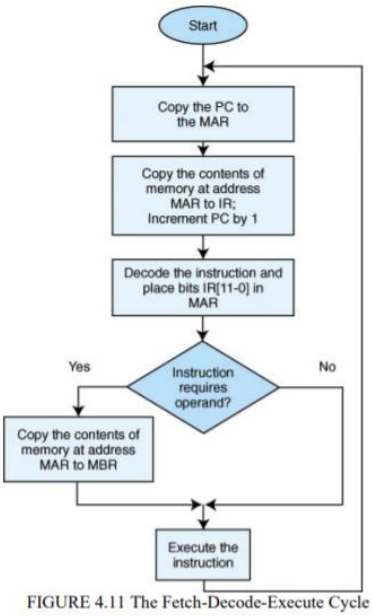


FIGURE 4.11 The Fetch-Decode-Execute Cycle

در گام اول باید مقدار دیتای درون ثبات PC را به ثبات MAR منتقل کنیم.

در گام بعدی باید دیتای خروجی حافظه را به ثبات IR منتقل کنیم و در همان زمان دیتای درون ثبات PC را نیز یکی افزایش دهیم.

گام بعد برای دیکود کردن دستور است و از انجایی که قالب دستورات را میدانیم پس قسمت ادرس انرا در ثبات MAR قرار میدهیم.

و حال باید تصمیم بگیریم بنابر opcode آیا دستور برای اجرا نیازمند عملگری از حافظه است یا نه.

هنگامی که این سوال را پاسخ دهیم دیگر فرایند fetch, decode به پایان رسیده است و به مرحله اجرای دستور رسیدیم.

یک فلگ (F) را ست میکنیم به محض ورودی به سلسله دستورات تا مشخص کند که این سلسله دستورات مربوط به fetch و دیکود است.

با یک جدول کارنو میتوان شرط مربوط به بخش بررسی شرط در الگوریتم را یافت.

یک فلگ هم برای هنگامی ست میکنیم که کار این دو پروسه تمام میشود و باید برویم برای اجرای دستور.(E)

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow IR[11-0], F_2 \leftarrow 1$$

$$T_0.F_2.Or(IR[14]'.IR[12], IR[15]'.IR[14]', IR[14]'.IR[13]'.IR[12]'): MBR \leftarrow M[MAR], E \leftarrow 1$$

$$F_2.NOR(IR[14]'.IR[12], IR[15]'.IR[14]', IR[14]'.IR[13]'.IR[12]'): E \leftarrow 1$$

(ج)

دستور اول

Load X

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow X, F_2 \leftarrow 1$$

$$T_0.F_2: MBR \leftarrow M[MAR], E \leftarrow 1$$

$$T_3.E: ACC \leftarrow MBR, S \leftarrow 1$$

دستور دوم

Store X

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow X, F_2 \leftarrow 1, E \leftarrow 1$$

$$T_4.E: M[MAR] \leftarrow ACC, S \leftarrow 1$$

دستور سوم

ADD X

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow X, F_2 \leftarrow 1$$

$$T_0.F_2: MBR \leftarrow M[MAR], E \leftarrow 1$$

$$T_3.E: ACC \leftarrow ACC + MBR, S \leftarrow 1$$

دستور چهارم

Subt X

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow X, F_2 \leftarrow 1$$

$$T_0.F_2: MBR \leftarrow M[MAR], E \leftarrow 1$$

$$T_3.E: ACC \leftarrow ACC - MBR, S \leftarrow 1$$

دستور پنجم

Input

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow X, E \leftarrow 1$$

$$T_5.E: ACC \leftarrow InREG, S \leftarrow 1$$

دستور ششم

Output

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow X, E \leftarrow 1$$

$$T_4.E: OutREG \leftarrow ACC, S \leftarrow 1$$

دستور هفتم

Halt

$$T_2.S: MAR \leftarrow PC, F \leftarrow 1, F_2 \leftarrow 0, F_3 \leftarrow 0$$

$$T_0.F: IR \leftarrow M[MAR], PC \leftarrow PC + 1$$

$$T_7.F: MAR \leftarrow IR[11-0], F_2 \leftarrow 1$$

$$F.F_2: MBR \leftarrow ACC, F_3 \leftarrow 1, F_2 \leftarrow 0$$

$$F.F_3: ACC \leftarrow ACC - MBR, F_3 \leftarrow 0, E \leftarrow 1$$

$$F.E.T_4: PC \leftarrow ACC, S \leftarrow 1$$

سوال ۸ به پاسخ اقای احدی نیا مراجعه شود

راهی که بنظرم میرسه این است که یک حافظه در نظر بگیریم که ۱۰۱ خانه دارد که هر خانه آن $\lfloor \log_2 n \rfloor$ بیت دارد. بدین صورت عمل میکنیم که از یکی یکی از اساتید نمره ای که مدنظرشان هست را دریافت میکنیم و میبینیم که نمره چند است اگر مثلا i بود به دیتای خانه ای که ادرسش برابر با i هست یکی اضافه میکنیم و پردازش میکنیم که آیا تعداد آن خانه برابر n/2 شده است یا نه اگر شده بود که کار تمام است و اگر نشده بود به این رونده ادامه میدهیم.

در این الگوریتم در بهترین شرایط نیاز به $\frac{n}{2} + 1$ تعداد پرسش و در بدترین شرایط نیاز به n تا پرسش داریم.