معماری کامپیوتر

به نام خدا

طراحی کامپیوتر پایه

1400/9/1

رضا صومی
طراحی کامپیوتر پایه

.1

آدرس دهی شبه مستقیم به طور کلی به معنای این است که نمی توان در یک دستورالعمل که در اینجا 32 بیت و در کل می تواند بیشتر باشد کل اطلاعات را قرار داد و متوسل به PC برای اینکه بخشی از اطلاعات را درون آن قرار داد می شویم.

فرض کنید پردازنده ای در اختیار داریم که طول دستورات 32 بیتی است و همچنین برای آدرس دهی به خانه های حافظه نیاز به 32 بیت داریم حال ما که نمی توانیم کل 32 بیت دستورالعمل را برای آدرس دهی صرف کنیم چرا که باید یک opcode نیز داشته باشیم تا تشخیص دهیم چه نوع دستوری باید اجرا شود لذا فرض کنید طول opcode نیز برابر 6 باشد حال 26 بیت جای خالی برای آدرس دهی داریم و ما 32 بیت نیاز داریم. یک راه حل (تف و سیریش) این است که 26 بیت آدرس را درون همان دستورالعمل قرار داده و 4 بیت آن را درون pc قرار دهیم (لازم به ذکر است 4 بیت پر ارزش pc برای این کار انتخاب می شود) و دو بیت کم ارزش را همیشه صفر بگذاریم چرا که در اینجا مقادیر ذخیره شده در حافظه 4 بایتی است و هر خانه حافظه 8 بیت. (جای دیگر می تواند متفاوت باشد) لذا در این پردازنده که به دستور برای مثال jump نیاز داریم از این طریق می توانیم این دستور را هندل کنیم در صورتی که آدرس دهی مستقیم جواب گوی این کار نیست.

.2

واحد ALU یک واحد محاسبه است که تعدادی محاسبات پایه را انجام می دهد در صورتی که در ISA موجود در یک پردازنده ممکن است انواع مختلفی از دستورات وجود داشته باشد برای مثال دستور MOV به طور مستقیم در ALU وجود ندارد اما همین دستور MOV را می توان با ADD با مقدار صفر پیاده سازی کرد لذا دستورات سطح بالاتر نیاز به محاسبات سطح پایین تری دارند حال آنکه بر اساس نوع دستور، کار انجام شده توسط ALU متفاوت است و بسته به نوع پیاده سازی پردازنده مورد نظر است. در یک پردازنده چندین نوع دستور ممکن است وجود داشته باشند برای مثال دستورات R-type یا J-type یا ... در دستورات R-type، 6 بیت انتهایی برای function در نظر گرفته شده است و بسته به opcode موجود در این قسمت function اطلاعات اضافه تری قرار داده می شود تا به طور مستقیم به ALU Control متصل شود و بر اساس این بیت ها این واحد کنترلی تصمیم بگیرد چه دستوری به واحد ALU ارسال کند. اما این پایان ماجرا نیست چراکه ما فقط دستورات R-type نداریم و ممکن است از نوع دیگر دستورات اجرا شود برای مثال دستور jump که در این قسمت آن 5 بیت آخر بخشی از آدرس هستند و هیچ ربطی به function ندارند لذا یک اطلاعات دیگری نیز باید داشته باشیم که این اطلاعات را Control unit با 2 بیت در اختیار واحد کنترلی ALU قرار می دهد. حال سوال می شود چرا 2 بیت و نه بیشتر. خب در اینجا باید گفت حداکثر 4 دسته یا رده از دستورات موجود بوده اند که باعث شده با همین 2 بیت کار جمع شود ولی ممکن بود با داشتن ISA متفاوت این تعداد

بیت خیلی بیشتر نیز باشد. برای پیاده سازی این واحد نیز نیاز است کلیه دستورات را بررسی کرده و اینکه در هر کدام ALU چه کاری باید انجام دهد را بر اساس 4 بیت خروجی دسته بندی می کنیم و یک جدول کارنو بزرگ تشکیل می شود که بر اساس آن طراحی این واحد کنترلی انجام می پذیرد.

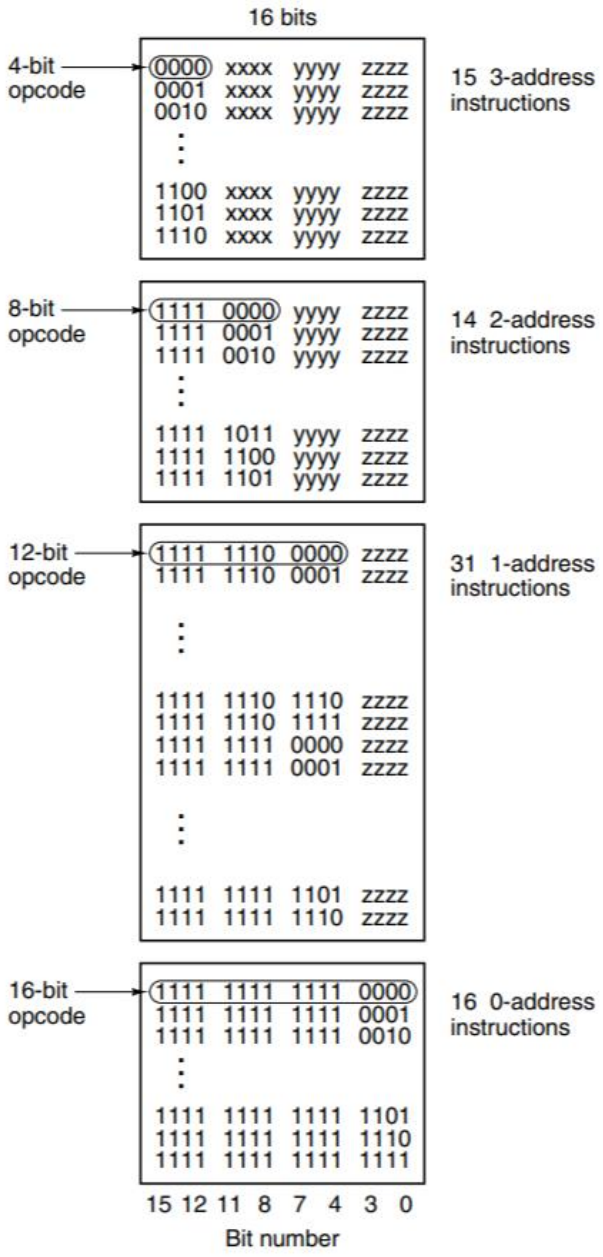
از مزیت این روش می توان به این اشاره کرد که دیگر مغز کامپیوتر که همان control unit می شود را خسته نمی کنیم به نوعی آن function که 6 بیت بود را به طور مستقیم به این واحد کنترلی مجزا وصل می کنیم و به control unit کاری نداریم و همین کار باعث می شود سرعت control unit نیز بالا رود و به نوعی خسته نشود. از کاستی این روش هم می توان به این اشاره کرد که باید منتظر control unit ماند تا 2 بیت را به واحد کنترلی ALU منتقل کند تا بقیه پردازش ها بتواند صورت گیرد. اما کاستی کلی تر این است که در اینجا ما 4 تایپ دستور داشتیم که توانستیم با گرفتن 2 بیت از control unit تشخیص دهیم ALU چه کاری باید انجام دهد و اگر تایپ دستورات بیشتر شود لذا 2 بیت کارا نیست و باید تعداد بیت بالاتری از control unit دریافت شود که این به این معنی ست که یک بار پردازش روی opcode صورت می گیرد و این خطوط توسط control unit تولید می شود و یک بار روی این بیت ها پردازش انجام می شود به عبارتی اگر تعداد بیت بالا باشد بهتر است در همان یک مرحله این کار صورت گیرد و این کار توسط دو سیستم ناهماهنگ انجام نپذیرد.

.3

یک نمونه کدگذاری ساده می تواند این باشد که کل تعداد دستورات را حساب کرده که در اینجا برابر 76 است و برای این 76 دستور به 7 بیت نیاز است و در 9 بیت باقی مانده آدرس ها را قرار می دهیم اما این روش اصلا بهینه نیست چرا که برای دستورات 0 آدرسه اصلا این 9 بیت نیاز نمی شود و در کل با توجه به اینکه دستور سه آدرسه نیز موجود است و 9 بیت کل ظرفیت باقی مانده است 3 بیت برای هر آدرس می توان اختصاص داد و این کار باعث می شود حافظه کمتری را بتوان اختیار کرد.

یک راه حل بهتر می تواند این باشد که با تمایز 4 بیت اول به 3 آدرسه بودن دستور و با تمایز 8 بیت اول به دو آدرسه بودن دستور و با تمایز 12 بیت اول به 1 آدرسه بودن دستور پی ببریم به این شیوه که 4 بیت اول از 0 تا 14 اگر باشند آنگاه این دستور سه آدرسه است و باقی 12 بیت می تواند محتوای عملوند ها باشد که هر کدام می توانند 4 بیت را به خود اختصاص دهند که از روش قبل یک بیت بیشتر است. حال اگر 4 بیت اول عدد 15 را نشان دهد می فهمیم این دستور سه آدرسه نیست و سراغ 4 بیت دوم می رویم که اگر بین 0 تا 13 باشد به این معنی است که دستور موجود دو آدرسه است و 8 بیت بعدی آدرس های موجود هستند که هر کدام 4 بیت به خود اختصاص می دهد. حال اگر 4 بیت اول عدد 15 و 4 بیت دوم یا 14 یا 15 را نمایش دهند آنگاه این دستور تک آدرسه است و بر اساس 4 بیت سوم با توجه به اینکه 4 بیت دوم یا 14 است یا 15 در نتیجه در این حالت می توانیم برای هر کدام 16 حالت و در مجموع 32 حالت در نظر بگیریم اما یکی از آنها یعنی هنگامی که 4 بیت اول و دوم و سوم همگی نشان دهنده عدد 15 باشند را به حالت چهارم و بدون آدرسه اختصاص می دهیم که در این حالت میز با توجه به اینکه 4 بیت آخر در موجموع 16 حالت می توانند اختیار کنند در نتیجه 16 حالت نیز برای دستورات بدون آدرس داریم.

کدگذاری شرح داده شده را در تصویر پایین مشاهده می کنید:



.4

(آ): دستور LdI به معنای این است که مقدار $immediate$ را $load$ کرده و در ثبات یا حافظه مورد نظر قرار داده می شود لذا در اینجا مقدار 20 در حافظه قرار می گیرد.

(ب): دستور LdA به معنای این است که مقدار موجود در $address$ مورد نظر را $load$ کرده و در ثبات یا حافظه مورد نظر قرار داده می شود لذا در اینجا مقدار موجود در آدرس خانه 20 که برابر 40 است در حافظه قرار می گیرد.

(پ): دستور $Ldind$ به معنای $load\ indirect$ است و مقدار موجود در $address$ مورد نظر خود آدرسی است که باید به آن رجوع کنیم. در اینجا ابتدا به خانه 20 حافظه رفته سپس مقدار آن را خوانده که برابر 40 است سپس مقدار موجود در حافظه با آدرس 40 که برابر 60 است را در ثبات یا حافظه مورد نظر load می کنیم.

(ج): همانند مورد بالا است خانه 30 حافظه دارای مقدار 50 است و خانه 50 حافظه دارای مقدار 70 است لذا مقدار 70 در ثبات یا حافظه مورد نظر قرار داده می شود.

.5

$$T_0.(R_C\ !=\ 0): R_C = R_C + R_B$$

$$T_0.(R_C\ ==\ 0): R_C = \sim R_A + \sim R_B + 1 = \sim R_A + \sim R_B + 1 + 1 - 1 = (\sim R_A + 1) + (\sim R_B + 1) - 1$$
$$= -(R_A + R_B + 1) \rightarrow R_C = -(R_A + R_B + 1)$$

$$T_1.(R_C\ !=\ 0): R_A = R_B$$

$$T_1.(R_C\ ==\ 0): R_A = 0$$

$$T_2: R_B = R_B - 1$$

.6

(الف):

تمامی دستورات نیاز به fetch & decode دارند. برای این کار 3 کلاک نیاز است ابتدا در کلاک اول pc باید در MAR گذارده شود برای این کار $S_1S_0 = 00$ باید باشد و MAR load فعال باشد. در کلاک دوم خانه با آدرس موجود در MAR محتوای آن در MBR قرار گیرد و لود شود لذا $S_4S_3S_2 = 001$ باید باشد و MBR load نیز فعال باشد و در کلاک آخر و سوم نیز مقدار MBR در IR لود می شود. چون PC باید همیشه به دستور بعدی اشاره کند در همین حین و به طور موازی نیز می توان عملیات INC را روی آن پذیرا شد.

با توجه به اینکه operand مستقیما به MAR وصل شده است لذا مقادیر immediate و دستورات immediate اعم از addi نمی تواند روی این سیستم انجام پذیرد.

دستور load را بررسی می کنیم. از آدرس موجود در operand می خواهیم مقدار آدرس موجود را خوانده و آن را در یکی از رجیستر ها load کنیم. در کلاک اول مقدار آدرس موجود که در operand قرار دارد در MAR باید قرار گیرد لذا $S_1S_0 = 01$ و MAR load باید فعال باشد. در کلاک بعدی خانه حافظه با آدرس موجود در MAR باید در MBR لود شود لذا $S_4S_3S_2 = 001$ باید باشد و MBR load نیز فعال باشد. در کلاک بعدی مقدار موجود در MBR باید در ACC ذخیره شود. برای این کار می توان از عملیات های مختلفی از ALU کمک گرفت برای مثال می توان آن را با صفر XOR کرد یا با صفر ADD/SUB کرد یا با یک OR کرد اما چون اینجا ورودی دیگر هم خود MBR باید باشد لذا تنها می توان از and یا OR استفاده کرد و مقدارش را

Pass

با خودش and/OR کرد. لذا در این کلاک $S_5 = 1$ و $F_2F_1 = 01$ باید باشد و همچنین ACC load فعال شود. و در کلاک نهایی لود همان رجیستری که مقدار موجود باید در آن ذخیره گردد فعال می شود تا نتیجه حاصله در آن قرار بگیرد.

دستور store را نیز بررسی می کنیم. برای اجرای این دستور کافیست مقدار موجود در یکی از رجیستر ها در یکی از خانه های حافظه قرار گیرد. ابتدا خانه حافظه موجود باید در MAR بنشیند لذا همانند قبل $S_1S_0 = 01$ و MAR load نیز باید فعال باشد. در همین کلاک می توان مقدار رجیستر مورد نظر را در MBR لود کرد لذا همانند دستور load باید لذا $S_4S_3S_2 = 001$ باشد و MBR load نیز فعال باشد. حال در کلاک بعد کافیست مقدار موجود در MBR در خانه حافظه با آدرس موجود در MAR ذخیره گردد که ورودی D_{in} باید فعال گردد و همان مقدار MBR را دارا شوند و این عملیات را انجام دهند.

دستور JUMP پیچیدگی خاصی ندارد و تنها باید مقدار موجود در operand در pc لود شود لذا یک کلاک نیاز است.

همچنین می توان مقدار موجود در stack pointer را به عنوان آدرسی در نظر گرفت و مقدار خانه حافظه با این آدرس را لود کرده و در رجیستر مورد نظر آن را ذخیره کنیم. برای این کار در کلاک اول باید $S_1S_0 = 10$ و MAR load نیز باید فعال باشد و در ادامه تمامی مراحل دقیقا مطابق دستور load خواهد بود.

در این نوع سیستم می توان دستورات غیر مستقیم حافظه ای را نیز اجرا کرد چرا که از MBR به طور مستقیم می توان داده به مالتی پلکسر موجود برای MAR ارسال کرد. فرض کنید دستوری وجود دارد که یک آدرس حافظه داده و مقدار موجود در آن آدرس حافظه خود یک آدرس حافظه است که مقدار آن باید برای مثال در یک رجیستر قرار گیرد. برای اجرای این دستور ابتدا آدرس حافظه در MAR لود می شود و در کلاک بعد مقدار موجود در این آدرس حافظه در MBR قرار می گیرد. سپس در کلاک بعد مقدار موجود در MBR در MAR لود می شود (با خطوط کنترلی $S_1S_0 = 11$) و سپس دوباره مقدار موجود در این آدرس حافظه در MBR لود می شود و در ادامه مانند دستور load می توان مقدار نهایی را در یکی از رجیستر های موجود قرار داد. همچنین می توان دستور مورد نظر این گونه باشد که آدرس خانه حافظه در یکی از رجیستر ها موجود باشد و در یک کلاک ابتدا مقدار آن در MBR لود شود سپس مقدار آن در اختیار MAR قرار گیرد تا در نهایت مقدار خانه با آن آدرس در MBR لود شود . در آخر در یکی از سه رجیستر متصل به ACC مقدار آن لود شود. (جالب است که خروجی ALU تنها می تواند در یکی از این سه رجیستر قرار گیرد و به هیچ عنصر دیگری وصل نیست و برای اینکه جواب ALU در اختیار قسمت های دیگر قرار گیرد ابتدا باید در یکی از رجیستر ها ذخیره شود سپس با استفاده از MBR در اختیار MAR قرار گیرد)

تمامی عملیات های منطقی که در جدول موجود برای خطوط کنترلی ALU نشان داده شده است را می توان روی یکی از رجیستر ها اعمال کرد و حاصل را در همان رجیستر یا رجیستر دیگر (R_T یا R_0 یا R_1) قرار داد که برای این کار کافیست یک بار MBR لود شود و سپس نتیجه حاصله در ACC قرار گیرد و سپس در رجیستر موجود لود شود که سه کلاک زمان می برد.

با توجه به معماری موجود نمی توان دستورات Base + Offset addressing mode یا conditional branch را در این سیستم اجرا کرد.

(ب):

دستور در این حالت fetch و decode شده است و در IR قرار دارد.

هر وقت که نیاز به load شدن رجیستر ها باشد کلاک زدن نیاز است اما می توان همزمان در یک کلاک دو عملیات یا چند عملیات load انجام شود. با توجه به اینکه عملیات در ALU انجام می پذیرد و تنها ACC و MBR است که پر کننده ورودی های ALU هستند لذا ابتدا باید یکی از دو ورودی در ACC ذخیره شود و سپس ورودی دیگر نیز از طریق MBR به ALU وصل شود تا عملیات جمع صورت پذیرد.

کلاک اول : لذا در ابتدا که مقدار R_0 آماده است را در MBR لود کرده و در همین زمان آدرس حافظه موجود در دستور را در MAR لود می کنیم لذا در کلاک اول باید $S_4S_3S_2 = 010$ باشد و MBR load فعال شود تا مقدار این رجیستر در MBR قرار گیرد همچنین در این کلاک باید $S_1S_0 = 01$ باشد و MAR load نیز فعال باشد تا آدرس حافظه موجود در MAR لود شود.

کلاک دوم : در این کلاک مقدار MBR باید در ACC ذخیره شود برای این کار می توان از عملیات های مختلفی از ALU کمک گرفت برای مثال می توان آن را با صفر XOR کرد یا با صفر ADD/SUB کرد یا با یک OR کرد اما چون اینجا ورودی دیگر هم خود MBR باید باشد لذا تنها می توان از and یا OR استفاده کرد و مقدارش را با خودش and/OR کرد. در نتیجه در این حالت $S_5 = 1$ و $F_2F_1 = 01$ باید باشد و در این کلاک ACC load فعال شود و همچنین در این کلاک باید از حافظه مقدار موجود در MAR گرفته شود و در MBR گذارده شود لذا $S_4S_3S_2 = 001$ باید باشد و همچنین MBR load فعال شود البته ممکن است سوال شود اگر MBR زودتر load شود آنگاه این مقدار به جای مقدار رجیستر R_0 که از قبل در MBR بود در ACC ذخیره می شود اما چون در اینجا فرض بر این است در یک کلاک تمامی دستورات همزمان اجرا می شوند لذا این مشکل در اینجا نادیده گرفته می شود (نمی دانم در واقعیت هم نادیده گرفته می شود یا یک کلاک اضافه تر داریم!) **بله یک کلاک دیگر میخواهیم**

چون حافظه بسیار کند است و ما نمیتوانیم انتظار داشته باشیم که در همان لحظه داده مورد نظرمان را تحویل دهید به همین دلیل باید در کلاک بعدی از آن بخوانیم

کلاک سوم : مقدار R_0 در ACC موجود است و مقدار موجود در آدرس حافظه در MBR قرار گرفته است حال در این کلاک $S_2 = 0$ و $F_2F_1F_0 = 000$ باید باشد تا این دو مقدار با هم جمع شوند و در این کلاک ACC load باید فعال باشد.

کلاک چهارم : و در این کلاک $R_0 load$ باید فعال شود تا مقدار نهایی در این رجیستر قرار گیرد. در نتیجه 4 کلاک نیاز است.

(آ) در این کامپیوتر طول هر دستورالعمل 16 بیت است و 4 بیت ابتدایی آن به opcode اختصاص یافته است پس بر اساس این 4 بیت حداکثر 16 بیت تعداد دستورات مان خواهد بود. این دستورات را در جدول زیر مشاهده می کنید.

Type	Instruction	Opcode	Def
Arithmetic	Add X	$0011 = 3$	$AC \leftarrow AC + mem[X]$
	Subt X	$0100 = 4$	$AC \leftarrow AC - mem[X]$
	AddI X	$1011 = 11$	$Add\ indirect: AC \leftarrow AC + mem[mem[X]]$
	Clear	$1010 = 10$	$AC \leftarrow 0$
Data Transfer	Load X	$0001 = 1$	$AC \leftarrow mem[X]$
	Store X	$0010 = 2$	$mem[X] \leftarrow AC$
I/O	Input	$0101 = 5$	$Request\ user\ to\ input\ a\ value\ and\ put\ it\ in\ AC$
	Output	$0110 = 6$	$print\ value\ form\ AC$
Branch	Jump X	$1001 = 9$	$Jump\ to\ address\ X$
	Skipcond	$1000 = 8$	$skips\ the\ next\ instruction\ based\ on\ a\ condition$
Subroutine	JnS X	$0000 = 0$	$store\ PC\ at\ address\ X\ and\ jump\ to\ X + 1$
	Jumpl X	$1100 = 12$	$use\ the\ value\ at\ X\ as\ the\ address\ to\ jump\ to$
Indirect Addressing	Storel X	$1110 = 14$	$Store\ indirect: mem[mem[X]] \leftarrow AC$
	Loadl X	$1101 = 13$	$Load\ indirect: AC \leftarrow mem[mem[X]]$
	Halt	$0111 = 7$	$End\ the\ program$

با توجه به جدول بالا و دستورات ممکن دو نوع مد آدرس دهی در این نوع کامپیوتر وجود دارد. دستوراتی مانند add sub load store jump از مد آدرس دهی direct addressing استفاده می کنند و دستوراتی مانند LoadI jumpl از مد آدرس دهی indirect addressing استفاده می کنند. لازم به ذکر است این direct و indirect با توجه به نوع کامپیوتر موجود از نوع حافظه ای است.

از دیگر ویژگی های این کامپیوتر می توان به 16-bit bus آن اشاره کرده که در هر لحظه تنها یکی از ثبات ها یا حافظه موجود قابلیت نوشتن روی آن را دارند و این bus به نوعی به ورودی های رجیستر ها و حافظه نیز متصل است و تمامی انتقال ها از طریق این bus صورت می گیرد. همچنین علاوه بر این bus راه ارتباطی مجزا برای ارتباط memory address register و main memory در نظر گرفته شده است که به دلیل کارایی و مفهوم make common case faster این کار صورت گرفته است چرا که آدرسی که باید اطلاعات آن خانه حافظه fetch شود در این رجیستر قرار دارد و با این کار دیگر نیازی نیست MAR مقدار خود را در bus قرار دهد و سپس main memory آن را از bus دریافت کند همان جا و به صورت مجزا می توان مقدار MAR را به memory رساند. همچنین در راستای این کارایی memory buffer register که حاوی مقدار 16 بیتی موجود در حافظه است با AC که آن نیز 16 بیتی است رابطه دوسویه برقرار کرده اند و می توانند به صورت مجزا انتقال اطلاعات داشته باشند. همچنین لازم به ذکر است ورودی های ALU همان MBR و AC هستند و خروجی آن نیز در AC قرار می گیرد.

(ب):

$MAR \leftarrow PC$

$IR \leftarrow mem[MAR], PC \leftarrow PC + 1$

$MAR \leftarrow IR[11:0]$

$if\ instruction\ require\ operand \rightarrow MBR \leftarrow mem[MAR]$

$execute$

(پ):

$Load\ X$:

$MAR \leftarrow X$

$MBR \leftarrow mem[MAR]$

$AC \leftarrow MBR$

$Store\ X$:

$MAR \leftarrow X, MBR \leftarrow AC$

$mem[MAR] \leftarrow MBR$

$Add\ X$:

$MAR \leftarrow X$

$MBR \leftarrow mem[MAR]$

$AC \leftarrow AC + MBR$

دقت کنید که از دستور $AC \leftarrow AC + mem[MAR]$ نمی توان استفاده کرد چرا که همان طور که در بالا اشاره کردیم ورودی های ALU دو رجیستر MBR و AC هستند لذا این کار امکان پذیر نیست.

$Subt\ X$:

$MAR \leftarrow X$

$MBR \leftarrow mem[MAR]$

$AC \leftarrow AC - MBR$

$Input$:

$AC \leftarrow InREG$

چرا که مقدار دریافت شده از input device در InREG قرار می گیرد.

$Output$:

$OutREG \leftarrow AC$

$Halt$:

$end\ of\ program\ and\ no\ operation\ are\ performed\ on\ registers$

$Skipcond$:

$if\ IR[11:10] == 00\ then$

$\quad if\ AC < 0\ then\ PC \leftarrow PC + 1$

$else\ if\ IR[11:10] == 01\ then$

$if\ AC == 0\ then\ PC \leftarrow PC + 1$

$else\ if\ IR[11:10] == 10\ then$

$AC > 0\ then\ PC \leftarrow PC + 1$

$jump\ X$:

$PC \leftarrow X$