**به نام خدا**

اعضای گروه: زهرا حیدری 98243020 - امیرحسین ادواری 98243004

**بخش تحلیلی**

1. جهت برنامه نویسی میکرو های آرم چهار نوع مجموعه دستورالعمل : ( ویژگی­های هر مجموعه دستور نیز در کنار کاربرد آن آورده شده است )

* Arm64 :

این مجموعه دستورالعمل توسط Armv8-A ساپورت می شود. به چند ویژگی در این مجموعه دستور العمل ها اشاره میکنیم: سمنتیک دستورالعمل ها تا مقدار زیادی شبیه به thumb32 و arm32 می باشد .دارای 31 رجیستر 64 بیتی که همیشه در دسترس هستند می باشد.Program Counter و Stack Pointer آن از رجیستر های general purpose نیستند . دستور العمل های جدید برای پشتیبانی از عملوندهای 64 بیتی دارد. اندازه ی همه آدرس ها 64 بیتی در نظر گرفته می شوند. مجموعه دستورالعمل‌های شرطی نیز کاهش یافته اند.

این اینستراکشن ست در سرورها، کامپیوترهای شخصی، و کاربردهای مربوط به IoT استفاده می­شود. به طور کلی برای نیازمندی­هایی که توان پردازشی زیاد میطلبند همانند نمایشگرهای high-resolution، گیمینگ سه بعدی و کاربردهایی ازین قبیل استفاده می­شود.

* Arm32 :

دستورات این اینستراکشن ست 32 بیت عرض دارند. A32 به طور عمده در برنامه های که به **performance** **بالا** نیاز دارند یا برای رسیدگی به استثناهای سخت افزاری مانند **وقفه­ها** و **processor start-up** استفاده میشوند. بیشتر دستورالعمل های A32 فقط زمانی اجرا می شوند که دستورالعمل های قبلی یک کد شرط خاص را تنظیم کرده باشند. این بدان معنی است که دستورالعمل‌ها تنها در صورتی تأثیر عادی خود را بر عملکرد برنامه دارند که فلگ­های N، Z، C و V شرایط مشخص‌شده در دستورالعمل را برآورده کنند. اگر فلگ­ها این شرط را برآورده نکنند، دستورالعمل به عنوان یک NOP عمل می کند. اجرای مشروط دستورالعمل ها اجازه می دهد تا بخش های کوچکی از دستورات if و while بدون استفاده از دستورالعمل های برنچ استفاده شوند.

* Thumb:

در این اینستراکشن ست دستورالعمل­ها 16 بیتی هستند (برخلاف A32 که 32 بیتی بودند ) برخی محدودیت­ها در این اینستراکشن ست اضافه شده است مثلا الزام به استفاده از 8 رجیستر اول (موسوم به low register) در برخی از دستورات. فرمت تغییریافته­ای نیز دارند برای نمونه در دستورات حسابی همانند تفریق، Rd و Rn یکسانند یعنی رجیستر مقصد و عملوند اول یکسانند.

متعاقب همین ویژگی­ها از این مجموعه دستور در جاهایی که **حجم کد** و نیز میزان **توان مصرفی** حائز اهمیت است (همانند کار با **میکروکنترلر و امبدد**) استفاده می­شود این کاهش منجر افت performance می­شود اما این افت در کاربردهای این اینستراکشن ست قابل توجه نیست و قابل تحمل است.

* Thumb32 یا Thumb2 :

یک مجموعه دستورالعمل ترکیبی 32 و 16 بیتی است که چگالی کد مناسبی را به منظور استفاده حداقل از حافظه سیستم به طراح ارائه می دهد. Thumb32 سطوح بهبود یافته performance، توان و چگالی کد را برای طیف گسترده ای از برنامه های کاربردی امبدد ارائه می دهد. طراحان می‌توانند از مجموعه دستورالعمل‌های T32 و A32 استفاده کنند و بنابراین انعطاف‌پذیری لازم برای تمرکز بر عملکرد یا اندازه کد را دارند.

در مجموع این ایسنتراکشن ست با انعطافی که ارائه می­دهد موجب می­شود تا علاوه **کم کردن حجم کد و سربار حافظه** (طی استفاده از دستورات 16 بیتی ) بتوان از برخی دستورات 32 بیت نیز استفاده کرد و **سرعت** را نیز تا حد مناسبی بهبود داد.

<https://developer.arm.com/documentation/ddi0210/c/CACBCAAE#:~:text=Thumb%20instructions%20are%20each%2016,between%20ARM%20and%20Thumb%20states>.

<https://developer.arm.com/architectures/instruction-sets/base-isas/t32>

<https://jumpcloud.com/blog/why-should-you-use-arm64#:~:text=An%20ARM64%20processor%20is%20an,3D%20gaming%2C%20and%20voice%20recognition>.

+اسلایدها و توضیحات استاد.

پاسخ سوال دوم)

زبان اسمبلی به برنامه‌نویس‌ها کمک می‌کند تا کدهای قابل خواندن توسط انسان را بنویسند که بیشترین شباهت را به زبان ماشین دارد. در زبان اسمبلی بر خلاف زبان سطح بالا توسعه دهنده **امکان دسترسی Register ها** را داراست. این کار به بهینه‌سازی سرعت کمک می‌کند و performanceرا افزایش می‌دهد.زبان اسمبلی به **تماس مستقیم با** **سخت‌افزار** کمک می‌کند. زبان اسمبلی در مقایسه با دیگر زبان‌های سطح بالا از **شفافیت** زیادی در بحث عملکردها و دستورات موجود برای آن برخوردار است و تعداد کمی عملیات برای آن وجود دارد. این زبان برای درک **جریان‌های کنترلی** مفید است.توسعه با زبان اسمبلی (درصورت وجود تخصص کافی در توسعه دهنده) از جهت سرعت و دقت یا بعبارتی **performance** و نیز **حافظه مصرفی** خروجی مناسب­تری نسبت به کد سطح بالا خواهد داشت.

استفاده از **منابع سیستم** در توسعه توسط زبان اسمبلی بهینه­تر خواهد بود مخصوصا اینکه در میکروکنترلرها محدودیت منابع در اختیار توسعه دهنده شدیدتر و جدی­تر است، لذا استفاده درست از زبان اسمبلی به **استفاده بهینه­تر و متعاقبا وسیعتر** از میکروکنترلر می­انجامد. عملا این امکان وجود دارد تا توسعه دهنده با **optimize کردن کد** خود بتواند استفاده بیشتری از منابع کند. قیدهای مربوط به زبان اسمبلی کمتر از زبان­های سطح بالاتر است. برای یک برنامه یکسان، کد اسمبلی که توسط کامپایلر یک زبان سطح بالا تولید می­شود حجم بیشتری از توسعه مستقیم به زبان اسمبلی خواهد داشت. در نهایت توسعه مستقیم به زبان اسمبلی اتکا به کامپایلرها برای اپتیمایز کردن کد را از بین می­برد.

<https://cpentalk.com/998/list-advantages-assembly-language-compared-level-languages>

<https://www.geeksforgeeks.org/difference-between-assembly-language-and-high-level-language/>

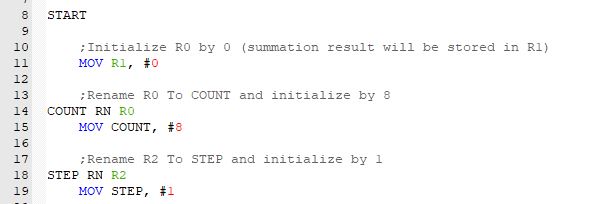
<https://www.gadgetronicx.com/assembly-language-features-uses-advantages-disadvantages/>

+اسلایدها و توضیحات استاد.

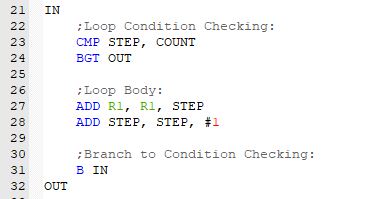
**بخش عملی**

**سوال اول)**

نتیجه محاسبه در رجیستر R1 نگه­داری می­شود.برای سهولت در خوانایی R0 را به COUNT و R2 را به STEP که همان گام حلقه است تغییر نام می­دهیم. COUNT را برابر 8 و STEP را برابر 1 قرار می­دهیم.

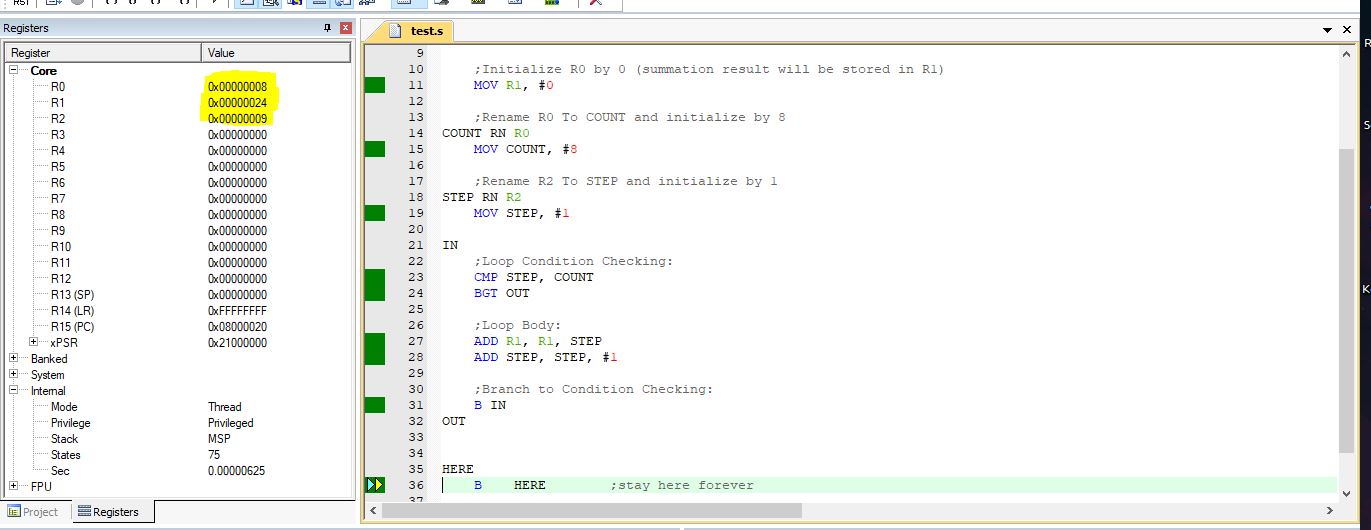


سپس در یک حلقه در هرمرحله R1 را با STEP جمع کرده و آنرا increment می­کنیم. و به قسمت بررسی شرط حلقه جامپ می­کنیم. این روال را COUNT بار انجام می­دهیم (‌ مقایسه در ابتدای حلقه صورت می­گیرد اگر STEP از COUNT‌ بیشتر شد به بیرون حلقه جامپ می­کنیم)



در نهایت نتیجه سیگما در R1 قرار خواهد گرفت.

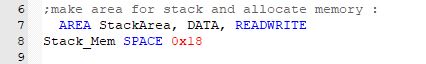
این برنامه را برای COUNT = 8 اجرا میکنیم نتیجه به صورت زیر است:‌



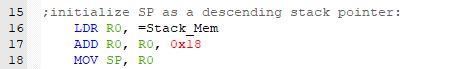
می­دانیم نتیجه سیگما برای COUNT = 8 بایستی 36 شود همانطور که قسمت هایلایت شده نشان می­دهد، رجیستر R1 حاوی عدد 0x24 است که به دسیمال 36 می­شود.

**سوال دوم)**

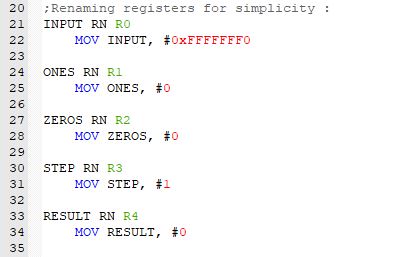
در این سوال به منظور ایجاد فضای استک، یک AREA از نوع دیتا با دسترسی READWRITE تعریف می­کنیم (خط اول) و در آن با دستوری که در خط دوم کد زیر آمده 24 بایت فضا (معادل 6 ورد) به استک برنامه اختصاص می­دهیم:



اکنون Stack\_Mem به ابتدای مموری 6 وردی استک ما اشاره می­کند، میدانیم که برای استک­های Descending عکس روال مذکور صادق است. لذا این آدرس را لود کرده، آنرا با 0x18 که معادل 24 بایت است جمع کرده و مقدار بدست آمده را که عملا آدرس انتهای فضای اختصاص شده است را در SP قرار می­دهیم.



حال می­توانیم عملیات ­های PUSH و POP را با توجه به اینکه یک استک Descending در اختیار داریم استفاده کنیم. سپس برای سهولت رجیسترهای R0 تا R4 را به ترتیب به INPUT، ONES، ZEROS و STEP و RESULT تغییرنام می­دهیم. نام آنها واضحا کاربرد آن­ها را نشان می­دهد.

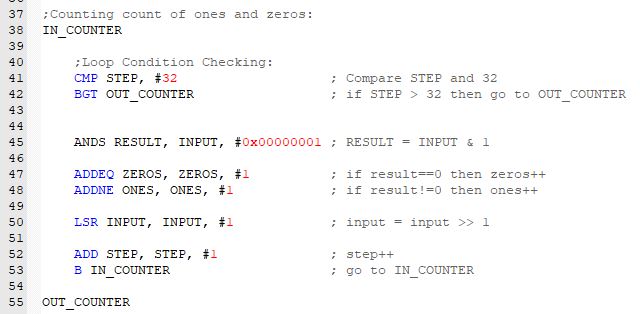


سپس برای شمارش تعداد صفرها و یک­ها روال زیر را در نظر می­گیریم :

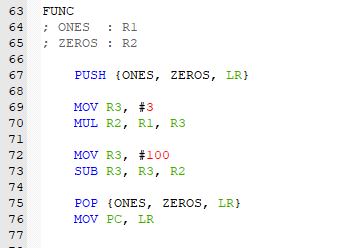
می­دانیم اگر یک رشته بیتی با 1 اند منطقی شود، اگر نتیجه 1 شود بدین معناست که بیت کم ارزش آن رشته بیتی 1 بوده و در غیراینصورت صفر بوده است.

در یک حلقه که 32 بار تکرار می­شود (شرط حلقه در ابتدای آن بررسی میشود و در صورتی که STEP از 32 بیشتر شود به بیرون حلقه برنچ می­کنیم) روال زیر را پیش می­گیریم :

1. INPUT را با 1 اند می­کنیم و نتیجه را در RESULT قرار می­دهیم، از دستور ANDS استفاده می­کنیم تا فلگ­ها ست شوند.
2. اگر حاصل عملیات قبل صفر شده بود یعنی Z==1 بود ZEROS را از طریق ADDEQ (که فقط زمانی که اجرا می­شود که Z==1 باشد) INCREMENT می­کنیم.
3. مشابه مرحله قبل اگر حاصل عملیات قبلی صفر نشده بود یعنی Z==0 بود ONES را از طریق ADDNE (که فقط زمانی که اجرا می­شود که Z==0 باشد) INCREMENT می­کنیم.
4. INPUT را یک واحد به راست شیفت می­دهیم.
5. گام حلقه را INCREMENT می­کنیم.
6. به قسمت بررسی شرط حلقه برنچ می­کنیم.



بدین ترتیب تعداد صفرها در ZEROS و تعداد یک ها در ONES قرار می­گیرد. سپس از طریق BL به لیبل FUNC که پیاده­سازی زیربرنامه ذکر شده در صورت سوال قرار دارد برنچ می­کنیم. در این زیربرنامه در ابتدا ONES و ZEROS و نیز LR را که آدرس بازگشت در آن است را در استک پوش کرده، عملیات گفته شده در صورت سوال را که واضحا مشخص است انجام می­دهیم، در نهایت 3 مقدار که در ابتدا پوش کردیم، پاپ می­کنیم و PC را برابر LR قرار می­دهیم تا به مکان فراخوانی زیربرنامه بازگردیم.



به ازای INPUT=0xffffff0 تعداد صفرها 0x4 و تعداد یک­ها 0x1C می­باشد، و پس از اجرای عملیات تابع مقدار R3 در نهایت بایستی 0x10 باشد که طبق تصویر زیر که مقادیر رجیسترها بعد از اجرا را نشان می­دهد. همینطور شده است:

