فاطمه سادات سیفی – 98243035

پارسا محمدپور – 98243050

سوالات تحلیلی:

شماره اول:

چهار مجموعه درستور العملی arm 32 بیتی و thumb و thumb2 و arm 64 بیتی هستند که هر کدام را به اختصار توضیح می دهیم :

**Arm 32 بیتی**: این مجموعه دستورات طولشان 32 بیت می باشد. عموما performance بالایی دارند یا در جاهایی که نیازمند هندل کردن hardware exception ها (مانند interrupt ها و processor start-up ) هستیم یا . به عنوان arm در pre-Armv8 شناخته می شوند. بیشتر دستورات این نوع معماری تنها در حالتی اجرا می شوند که اینستراکشن قبلی یک کد شرطی خاص را اجرا کرده باشد.

**Arm 64 بیتی**: این نوع مجموعه دستورات به وسیله معماری Armv8-A ساپورت می شوند. در این معماری 31 رجیستر 64 بیتی general-purpose در تمامی مدت قابلدسترسی خواهند بود. در این مجموعه دستورات، pc و sp به عنوان رجیستر های general-purpose نیستند. در این مجموعه دستورات تمامی ادرس ها 64 بیتی هستند. در این نوع مجموعه دستورات جدید برای پشتیبانی از operand های 64 بیتی دارند. همچنین در اینجا مجموعه دستور های شرطی نیز کاهش یافته است. در این مجموعه دستورات، semantic دستورات تا حد زیادی مشابه arm32 بیتی و thumb2 می باشد.

**Thumb**: این مجموعه دستورات، یک زیر مجموعه از دستوراتی که به مقدار زیاد بوسیله arm 32 بیتی استفاده می شوند است. این اینستراکشن ها 16 بیتی هستند و دستورات متناظر با دستورات arm 32 بیتی دارند که تاثییر یکسانی بر مدل پردازنده دارند. در اینجا مزیت هایی از جمله فضای حافظه 32 بیتی را داریم همچنین رجیستر های 32 بیتی داریم و shifter عه 32 بیتی و alu 32 بیتی نیز داریم. کد های که به این شکل نوشته شوند، عموما حجمی معادل 65 درصد حجم کد های arm را دارند و همچنین برایمان performance عه 160 % نسبت به کد های arm را فراهم میکنند در شرایطی که روی memory-system عه 16 بیتی اجرا شوند. در دسترس بودن هر دو مجموعه دستورات thumb 16 بیتی و arm 32 بیتی، به طراحان این انعطاف پذیری را می دهد که تاکیید بر روی اجرا و سایز کد روی لول سابروتین را داشته باشند.

**Thumb 32 (thumb2)**: به عنوان thumb در معماری های pre-Armv8 شناخته می شود. ترکیبی از اینستراکشن های 32 و 16 بیتی هستند که همین امر برای طراحان یک تراکم زیادی را بوجود می آورد برای مینیمم مموری سیستم و هزینه سیستم. این مجموعه دستورات یک لول بالاتر و پیشرفته تری از performance ، انرژی مفید و تراکم کد بسیار بالا برای طیف گسترده ای از embedded applications فراهم می کند. طراحان می توانند هر دو مجموعه دستورات t32 و a32 را اجرا استفاده کنند. از این رو بنابر نیاز application، می توانند انعطاف بر روی تاکیید کردن روی performance و یا سایز کد داشته باشند. این مجموعه دستورات بر روی تمامی معماری ها قابل اجرا هستند و تنها مجموعه دستوراتی هستند که بر روی معماری های M-Profile قابل پیاده سازی هستند.

شماره دوم:

زبان اسمبلی شباهت بسیار زیادی به کد های سطح ماشین دارد. در کامپیوتر ها، برای تبدیل کد اسمبلی به کد ماشین قابل اجرا یک اسمبلر وجود دارد که این کار را برایمان انجام می دهد. زبان اسمبلی کلیه کار هایی که یک کامپیوتر توانایی انجام آنها را دارد برای ما فراهم می کند. در زبان اسمبلی ما دسترسی مستقیم به رجیستر ها را داریم ولی این زبان برای ما تعداد عملیات های زیادی را فراهم نمی کند. در زبان اسمبلی دسترسی به قطعه های سخت افزاری و همچنین I/O ها آسان تر می باشد و همچنین به همین خاطر (نزدیک بودن به زبان ماشین و راحت بودن دسترسی به سخت افزار و دسترسی به رجیستر ها) برای ما امکاناتی را بوجود می آورد که بعضا در برخی از زبان سطح بالا نمی توان از آنها استفاده کرد.

در زبان اسمبلی همچنین بسیاری از ساده سازی ها برایمان آسان تر است زیرا دیگر عواملی مثل کامپایلر ها وجود ندارند از طرفی هم این امر باعث می شود که کد زدن به این زبان سخت تر باشد ولی باز هم در فهمیدن الگوریتم ها موثر است. همانطور که در پاراگراف قبلی اشاره شد، در زبان اسمبلی می توان کنترل کامل سیستم و منابع آن را بدست آورد. زبان اسمبلی سرعت را بهبود می بخشد و کارایی را افزایش می دهد و به عبارتی باعث بهبود performance می شود.

دستور کارها:

شماره اول:

برای این قسمت کد assembly را در قرار دادیم و اسکرین شات آن را در انتهای همین گزارش پس از توضیح دادن قسمت های مختلف کد، قرار داده ایم.  
برای این کار ساز و کار کلی یک حلقه را پیاده کردیم که در آن یک متغییر (i) از 1 تا مقدار count (که برابر با ثابت 8 در نظر گرفته شده است در این سوال) جلو می رویم و در هر سری این متغییر را با مقدار SUM که تا کنون وجود داشته است، جمع می کنیم. همچنین در ابتدا هم مقدار متغییر SUM صفر است.  
آنقدر جلو میرویم که مقدار متغییر i مان، بزرگتر از count شود، سپس وقتی مقدار متغییر i بزرگتر از count شد، سپس به اتمام اجرای حلقه می رسیم و باید از حلقه بیرون برویم و اجرای برنامه تمام می شود.

با توجه به مطلب بیان بده در کانال درسی، برای اینکه برای اجرا، مقدار رجیستر PC را دستی صفر نکنیم، ابتدا یک AREA تعریف میکنیم که به صورت زیر می باشد و این باعث می شود که در هنگام اجرا، به صورت خودکار، مقدار رجیستر PC به جایی اشاره کند که در آن کد های قسمت Example وجود دار. تیکه کد زیر برای همین منظور است :



سپس باید AREA ایی برای کدمان بسازیم (با نام Example) و کدی که میزنیم را در آن قرار دهیم. تیکه کد زیر برای همین منظور است:



حال نیز باید برای این procedure ای که در قسمت بالا ساخته ایم، یک ENDP قرار دهیم تا بسته شود.(که نشان دهنده اتمام این procedure نیز می باشد.) تیکه کد زیر برای همین منظور است(در اینجا چون فقط همین procedure را داریم و بعد از آن کدهایمان به پایان میرسد، پس END اخر برنامه را هم پس از آن قرار داده ایم.):



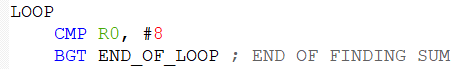
حال مطابق با توضیحات فوق، ابتدا متغییر SUM را مقداردهی میکنیم (در این کد همانطور که صورت سوال از ما خواسته، رجیستر R1 را برای متغییر SUM قرار می دهیم.) که مقدار اولیه آن هم برابر با صفر است. تیکه کد زیر، برای همین منظور است:



سپس بنابر توضیحات فوق باید ساز و کار یک حلقه را پیاده کنیم که از عدد 1 تا عدد count (که در اینجا برابر با ثابت 8 در نظر گرفته شده) جلو برویم و هرسری این عدد را با SUM جمع کنیم. پس برای این کار ابتدا یک متغییر (متغییر i) را مقداردهی اولیه کردیم که در این کد، رجیستر R0 را برای متغییر i قرار می دهیم. برای مقدار دهی کردن هم چون شروع کار (شروع عدد هایی که باید با هم جمع شوند، از یک است، پس مقدار اولیه آن را 1 گذاشتیم.) تیکه کد زیر برای همین منظور است:



سپس باید یک حلقه را پیاده سازی کنیم، به همین دلیل برای اینکه در هر سری این مقدار R0 (همان i) آپدیت می شود، پس باید چک کنیم که آیا مقدار آن از عدد count (در این سوال، عدد 8) بیشتر شده است یا نه و اگر بیشتر شده باشد، باید ادامه اجرای حلقه را متوقف کنیم و به انتهای برنامه برویم، وگرنه (اگر بیشتر نشده باشد، به عبارتی مقدار R0 کوچکتر مساوی 8 باشد)، به اجرای حلقه ادامه دهیم. تیکه کد زیر برای همین کار ابتدا یک label قرار داده است (LOOP) که نشانه شروع حلقه است. (و هر سری پس از یک دور اجرای کامل حلقه به آنجا میاییم.) سپس در ادامه یک مقایسه گذاشتیم که بیایم و ببینیم که آیا مقدار رجیستر R0 (همان i) از مقدار count (اینجا همان ثابت 8) بیشتر نشده باشد. سپس پس از اینکه مقایسه انجام شد، یک دستور branch (دستور branch شرطی) قرار دادیم که اگر در این نتیجه این مقدار مقایسه، مقدار (i) R0 از مقدار 8 بیشتر شده بود، به انتهای برنامه برویم. تیکه کد زیر برای همین منظور قرار داده شده است:



سپس باید بدنه حلقه و قسمت آپدیت شدن (i) R0 را پیاده سازی کنیم. در بدنه حلقه، ابتدا باید مقدار قبلی رجیستر R1 (SUM) را با مقدار رجیستر R0 (i) جمع کنیم و حاصل را در رجیستر R1 (SUM) بریزیم. سپس پس از آن باید مقدار رجیستر R0 (i) را آپدیت کنیم و به مقدار آن یکی اضافه کنیم.(و حاصل را در خودش یعنی رجیستر R0 بریزیم.) تیکه کد زیر برای همین کار است:



سپس پس از انجام بدنه حلقه و آپدیت کردن مقدار R0 (i) باید با استفاده از یک دستور branch به ابتدای حلقه برگردیم و شرط را چک کنیم. تیکه کد زیر برای همین منظور است:



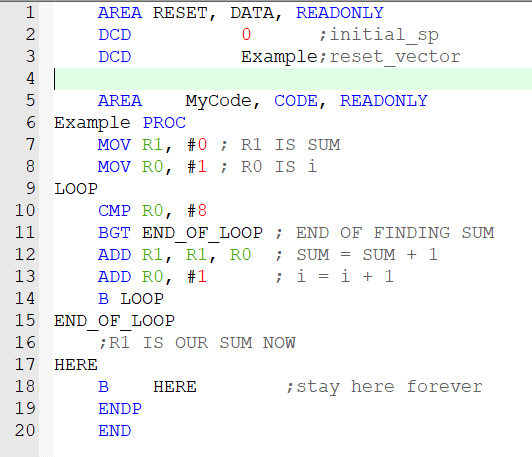
اکنون که موارد مربوط به حلقه تمام شده است، طبق مورد گفته شده در 3 قسمت بالاتر (جایی که در آن توضیح دادیم که باید مقایسه صورت بگیرد و اگر بزرگتر شد یک دستور branch (دستور branch شرطی)قرار میدهیم و ... )که این دستور branch شرطی، به انتهای حلقه اشاره میکرد، اکنون آن label را قرار میدهیم تا آن دستور branch شرطی به این label که بتواند به اینجا که انتهای حلقه است، اشاره کند. تیکه کد زیر مربوط به همین تیکه میباشد:



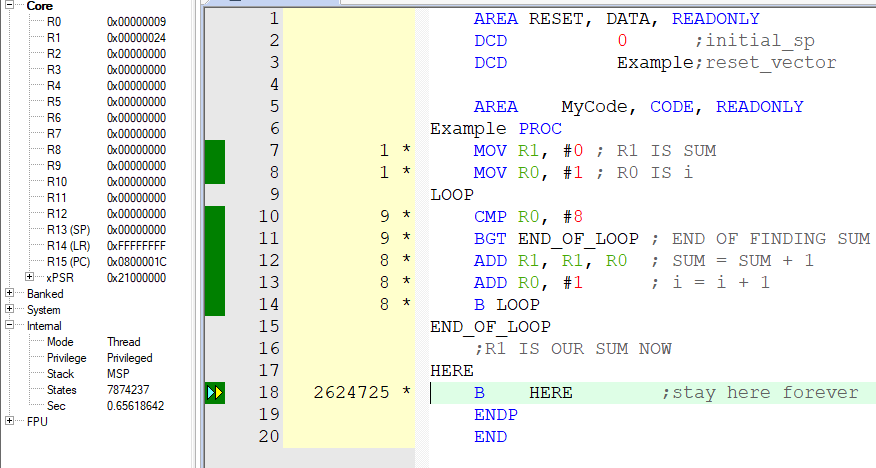
سپس در این تیکه، کد های مورد نیاز به پایان رسیده، اما پس از یک دور اجرا کد و رسیدن به این قسمت، اگر تیکه زیر را قرار ندهیم که در آن گیر کند و هیچ وقت از این خارج نشود، دوباره کد از اول اجرا می شود و ممکن است مشکلاتی بوجود بیاورد. به همین دلیل ما این تیکه کد زیر را قرار دادیم که در اصل باعث می شود که هنگام اجرا، وقتی که به این تیکه رسیدیم، همیشه در آن گیر کنیم و دیگر مشکل گفته شده (اجرای دوباره کد و ...) پیش نیاید. (که چون این دستور branch همیشه خودش را روی خودی می اندازد، باعث بوجود آمدن یک حلقه بی نهایت می شود و برنامه همیشه در این تیکه گیر میکند.) تیکه کد زیر که صرفا یک label است که در خط بعد یک دستور branch به همان label را قرار داده ایم:



پس در مجموع، کد ما به شکل زیر در میاید:



حال قسمت های مربوط به اجرای کد را قرار می دهیم:



همانطور که میبینیم، مقدار رجیستر R1 که همان مقدار SUM است 0x00000024 شده است. با توجه به اینکه این عدد در مبنای 16 است و معادل مبنای 10 آن می شود 2 \* 16 + 4 \* 1 = 36 که این جواب درست است، زیرا :

SUM = = 1 + 2 + 3 + … + 8 = 36

و همچنین مقدار رجیستر R0 ، 9 شده است که معادل مبنای 10 آن هم می شود 9 که درست است، زیرا وقتی که (در حلقه) عدد داخل این رجیستر بزرگتر از 8 شد، از حلقه بیرون آمده است.  
اعدادی که با ستاره در کنار خط های کد نشان داده شده اند، نشانه تعداد دفعات اجرای هر خط از کد است که این مقدار برای آن دو خطی که صرفا برای مقدار دهی به R0 و R1 بوده اند، یک است و برای آن خط هایی که مقایسه انجام می شد و آن دستور branch شرطی چک می شد، هم 9 است که یکی بیشتر از تعداد دفعات اجرای دستورات بدنه حلقه و آپدیت (چون این دو دستور برای اعداد 1 تا 8 همانند بدنه حلقه و آپدیت اجرا می شوند، اما یک بار اضافه تر هم برای عدد 9 اجرا می شوند چون این دو دستور چک می کنند که باید ادامه داد یا نه و ... به همین دلیل یک بار بیشتر از خود بدنه حلقه اجرا شده اند) که درست است.  
دستورات بدنه حلقه و آپدیت هم هر کدام 8 بار اجرا شده اند که یعنی هر کدام یک بار به ازای اعداد 1 تا 8 اجرا شده اند.  
آن دستور حلقه بی نهایت انتهای کار هم تعداد دفعات زیادی اجرا شد تا نشان دهیم برنامه در آن تیکه گیر میکند.

همچنین عکس های اجرا شدن مرحله به مرحله کد هم در کنار فایل اسمبلی همین کد قرار داده شده است.

شماره دوم:

در این قسمت ما بدین شکل عمل کردیم که ابتدا کد مربوط به قسمت های ساخت SP و همانطور که در قسمت قبل توضیح داده شد، برای اینکه به صورت خودکار مقدار رجیستر PC در همان ابتدا درست مقدار دهی شده باشد، را قرار دادیم. سپس تک عدد تصادفی (در این سوال از عدد 429 که باینری آن به صورت 110101101 می باشد) استفاده کردیم. برای پیدا کردن تعداد صفر ها و یک های این عدد، آمدیم و در طی یک حلقه تمام 32 بیت عدد را بررسی کردیم و اگر کم ارزش ترین بیت آن (LSB) صفر بود به تعداد صفر ها یکی اضافه کردیم و اگر یک بود، به تعداد یک ها یکی اضافه کردیم. سپس وقتی تمام 32 بیت آن را بررسی کردیم، تابع ذکر شده در صورت سوال را فراخوانی می کنیم که این تابع نیز صرفا دستورات و کار های گفته شده را انجام می دهد.(در این تابع، ورودی ها R1, R0 هستند و خروجی نیز در رجیستر R3 قرار می گیرد.) سپس مانند دستور کار اول، یک حلقه بی نهایت قرار دادیم تا پس از اتمام اجرای برنامه در آنجا بمانیم.

همانطور که گفته شد در ابتدا یک AREA قرار دادیم (مانند دستور کار اول) که باعث شود مقدار PC در همان ابتدا به صورت خودکار تعین شود.(و به همان جایی اشاره می کند که procedure عه Main در آن قرار دارد.) تیکه کد مربوط به این قسمت به صورت زیر است:



سپس برای اینکه SP را به صورت دستی تعیین کنیم و مقدار آن به اندازه 6 word باشد، باید یک ناحیه جدید ایجاد کنیم که این ناحیه باید هم قابلیت خواندن و هم قابلیت نوشتن را داشته باشیم و نمیتواند به صورت READONLY باشد و حتما باید به صورت READWRITE باشد. این ناحیه از نوع DATA می باشد زیرا در آن در حال مشخص کردن SP هستیم و در این ناحیه قرار است یک سری دیتا نوشته و خوانده شوند.(در این ناحیه قرار نیست کدی قرار بگیرد که بخواهیم این ناحیه را از نوع CODE قرار دهیم.) می خواهیم پشته مان به طول 6 word باشد و برای این کار باید از دستور SPACE به همراه یک اسم که در آن برای ما آدرس شروع آن ناحیه را قرار می دهد استفاده کنیم.(ما در اینجا از اسم STACK\_POINTER استفاده کردیم.) سپس در جلوی آن باید یک عدد می نوشتیم که نشان دهنده تعداد بایت های این پشته مان است می باشد، همچنین می دانیم که سایز هر word ، در اصل 4 بایت است. پس برای اینکه بتوانیم 6 word فضا در پشته داشته باشیم، باید 6 \* 4 = 24 بایت فضا برای این ناحیه (پشته مان) اختصاص دهیم. تیکه کد زیر برای همین منظور است:



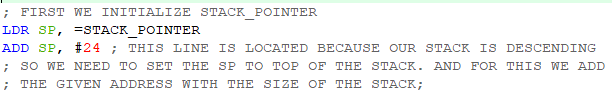
حال می توانیم کد های قسمت اصلی مان را بگذاریم (منظور از قسما اصلی همان procedure عه در Main است.) برای اینکار ابتدا ناحیه مورد نظر را به صورت READONLY و CODE قرار دادیم. تیکه کد زیر برای همین منظور است:



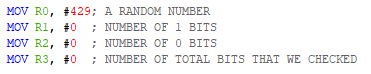
چون در این قسمت از یک procedure استفاده کردیم، باید در انتها آن را ببندیم. پس تیکه کد زیر را برای همین منظور قرار دادیم(چون انتهای procedure در اصل در اینجا می شود همان انتهای برنامه، پس باز هم همانند قسمت قبل آن را در بالای END قرار می دهیم.):



حال به سراغ procedure عه Main می رویم. در اینجا ابتدا باید آن SP را تنظیم کنیم. با توجه به اینکه در قسمت های قبلی توضیح داده شد که برای ساخت STACK\_POINTER چه اقداماتی صورت گرفت، حال میایمم و از آدرسی که در STACK\_POINTER برایمان ساخته شده است، استفاده می کنیم و با دستور LDR مقدار SP حال به همان آدرس شروع پشته ای که ساختیم می کند. اما ما پشته مان به صورت Descending می باشد و برای داشتن پشته ای که به این صورت کار کند، باید مقدار SP به بالاترین آدرس(بزرگترین آدرس) اشاره کند تا بتواند مقدار آن کم شود. پس برای اینکه این را هم درست کنیم، باید مقدار کنونی SP را (که آدرس شروع پشته است) با مقدار فضای کل موجود در پشته جمع کنیم تا مقدار SP درست شود.(تا به بزرگترین آدرس اشاره کند.) پس برای همین منظور، SP را با سایز پشته (که قبلا محاسبه کردیم و مقدار آن 24 بایت شد) جمع کنیم و حاصل را در داخل پشته بریزیم. تیکه کد زیر برای همین منظور است:



حال به سراغ مراحل محاسبه تعداد صفر ها و تعداد یک های عددمان میرویم. برای اینکار، ابتدا مقدار دهی های اولیه را انجام می دهیم، که برای اینکار باید در رجیستر های R1, R2و R3 مقدار صفر را قرار دهیم.(رجیستر R1 در اصل تعداد یک هاست که مقدار اولیه آن صفر است و به مرور با بررسی ها در حلقه در صورت نیاز زیاد می شوند. در رجیستر R2 تعداد صفر ها را نگه می داریم که دقیقا مشابه با تعداد یک ها است. در رجیستر R3 هم تعداد بیت هایی که آن لحظه بررسی کردیمشان را قرار می دهیم.) مقدار اولیه رجیستر R0 هم که همان عددی است که می خواهسم عملیات را روی آن انجام دهیم و در اینجا مقدار آن را همانطور که گفته شد 429 (که باینری آن به صورت 110101101 می باشد) قرار می دهیم. تیکه کد زیر برای همین منظور می باشد:

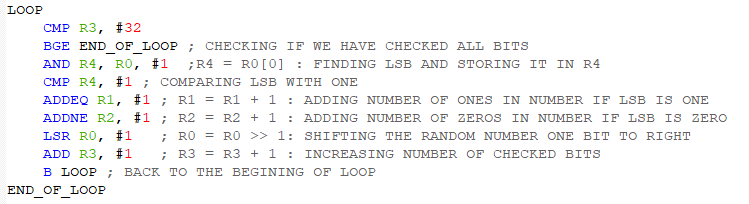


در ادامه کار می توانستیم از همان رجیستر R0 بدون اینکه برای آن از دستور PUSH استفاده کنیم هم جلو برویم ولی در این صورت در انتها مقدار اولیه موجود در آن را از دست می دادیم برای اینکه این اتفاق نیفتد، یک عمل PUSH برای آن انجام دادیم که پس از اجرای حلقه و اتمام کار، با اجرای دستور POP مقدار آن (که پس از اجرای کامل حلقه صفر شده بود) به همان مقدار اولیه اش باز میگردد. تیکه کد های زیر، عملیات های PUSH و POP مربوط به همین تیکه ها هستند:





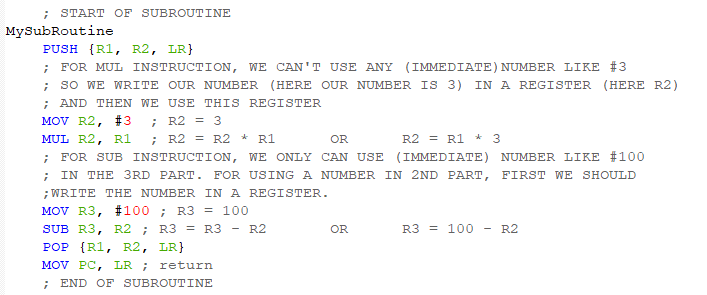
حال به سراغ کد های قسمت حلقه می رویم. در این حلقه ما میاییم و تمام بیت های عدد رندوم ورودی که در رجیستر R0 قرار دارد را بررسی می کنیم. در ابتدا چک میکنیم که آیا تمام بیت های عدد را بررسی کردیم یا نه که برای این کار تعداد بیت هایی که تا آن لحظه بررسی شده اند را با عدد 32 که تعداد کل بیت های عدد است مقایسه می کنیم و اگر این مقدار (تعداد بیت های بررسی شده که مقدار آن در رجیستر R3 است) بزرگتر یا مساوی عدد 32 باشد، یعنی ما تمام بیت ها را بررسی کردیم و باید از این حلقه خارج شویم و به LABEL موجود در انتهای حلقه برویم وگرنه، بدنه حلقه را اجرا کنیم. در هر مرحله از این حلقه کم ارزش ترین بیت (LSB) رجیستر R0 را پیدا میکنیم. برای پیدا کردن این بیت کافی است رجیستر R0 را با عدد 1 AND کنیم و مقدار را در رجیستر R4 بریزیم.(چون در عدد یک تمامی بیت ها جز اولین بیت سمت راست، صفر است، پس حاصل این AND همواره یا صفر است یا یک که دقیقا همان کم ارزش ترین بیت رجیستر R0 می شود.) حال این مقدار موجود در رجیستر R4 را با عدد 1 مقایسه می کنیم و اگر با یکدیگر برابر بودند، باید به تعداد یک های موجود در این عدد (همان مقدار موجود در رجیستر R1) یکی اضافه کنیم، و اگر حاصل این مقایسه برابر نبود، باید به تعداد صفر های موجود در این عدد (که مقدار آن در رجیستر R2 است) یکی اضافه کنیم. سپس پس از اسنکه اسن کار ها را انجام دادیم، عدد رندوم را (که مقدار آن در در رجیستر R0 است) یک بیت به سمت راست شیفت بدهیم.(به صورت منطقی یعنی همان LSR) حال باید تعداد بیت هایی که بررسی کردیم را یکی زیاد کنیم و سپس به LABEL موجود در ابتدای حلقه برگردیم و این روند را ادامه دهیم. در انتهای این حلقه نیز، LABEL مورد نیاز برای انتهای این حلقه موجود است. تیکه کد زیر برای همین قسمت است:



حال که اجرای بدنه حلقه به اتمام رسید، تعداد صفر ها و یک های عدد محاسبه شده است، پس باید به سابروتین گفته شده برویم و کد های مربوط به آن قسمت را اجرا کنیم. برای اینکار باید از دستور BL استفاده کنیم. کد آن به صورت زیر است(اسم که در ادامه دستور BL میاید، دقیقا همان اسم سابروتینمان است.):



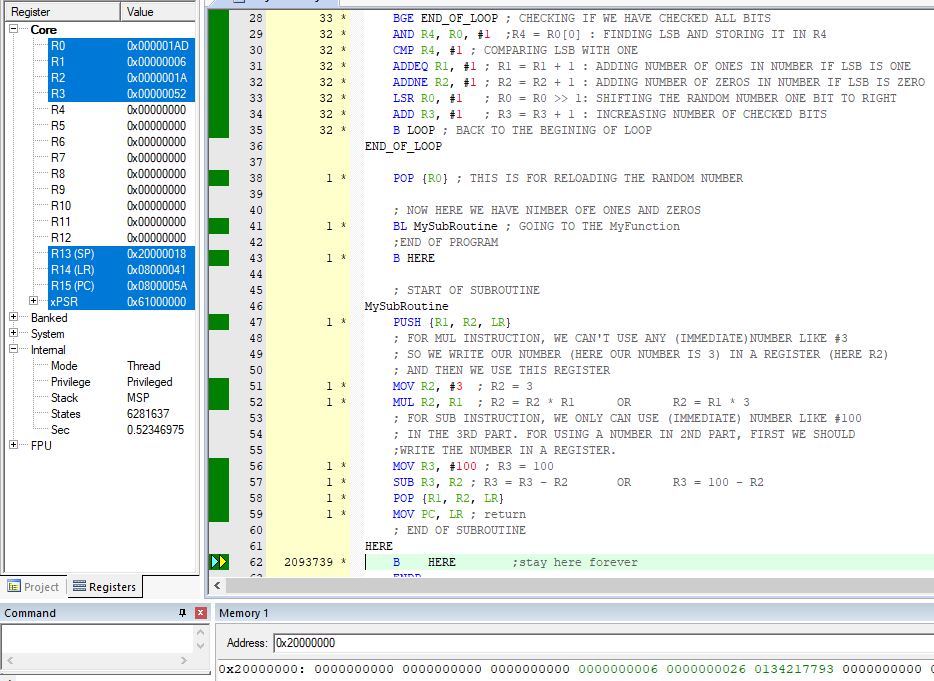
حال به سراغ قسمت سابروتین می رویم. در این قسمت که با استفاده از یک LABEL به نام MySubRoutine مشخص شده است، در ابتدا همانطور که در صورت سوال از ما خواسته شده است، ابتدا در پشته، رجیستر های R1, R2وLR را PUSH می کنیم. حال برای اینکه بتوانیم تعداد یک های موجود در عدد را (که مقدار رجیستر R1 است) در 3 ضرب کنیم و حاصل را در رجیستر R2 بریزیم، ابتدا مقدار 3 را در رجیستر R2 قرار دادیم و سپس رجیستر R2 را (که مقدار آن 3 است) با رجیستر R1 در هم ضرب کردیم و حاصل را در رجیستر R2 ریختیم.(علت اینکه ابتدا عدد 3 را در رجیستر R2 ریختیم و سپس عملیات ضرب را انجام دادیم این است که در عملیات ضرب، نمی شود از مقدار immediate استفاده کرد و باید از رجیستر ها استفاده کرد وگرنه به ارور میخوریم.) حال برای اینکه حاصلضرب را از 100 کم کنیم، باید ابتدا مقدار 100 را در رجیستر بریزیم و سپس از دستور SUB استفاده کنیم. زیرا نمی توان یک مقدار immediate را در operand دوم در یک عملیات منها قرار داد.(فقط می توان آن را در operand سوم قرا داد.) سپس مراحل اجرای این سابروتین به اتمام می رسد و همانطور که در صورت سوال از ما خواسته شده است، مقدار موجود در R1, R2 و LR را POP می کنیم و سپس باید به جای قبلی که در آن بودیم برگردیم.(اجرای این سابروتین تمام شده است پس باید return کنیم) برای اینکار از دستور MOV استفاده می کنیم و مقدار LR را در PC میریزیم.(چون مقدار LR به جایی اشاره میکرد که از آن آمده بودیم به سابروتین، پس با استفاده از این دستور، به همان جا برمیگردیم.) تیکه کد زیر برای هیمن قسمت می باشد:



حال که اجرای که تمامی کار ها را انجام دادیم، مانند دستور کار اول همانطور که گفته شد، باید به نوعی یک حلقه بی نهایت ایجاد کنیم که پس از اتمام اجرای دستورات در آن گیر کند و به اجرا کردن ادامه ندهد و مشکلی ایجاد نشود. این تیکه کد برای همین است:



حال عکس های مربوط به قسمت ران گرفتن کلی از کد را قرار می دهیم(ران گرفتن قسمت به قسمت، در کنار فایل کدی همین قسمت می باشد):



همانطور که در شکل مشخص است، مقدار ورودی (مقدار موجود در رجیستر R0) 0x000001AD می باشد که مقدار دسیمال آن می شود عدد 429 که همان ورودی است. مقدار موجود در رجیستر R1 نیز 0x00000006 است که در اصل می شود همان عدد 6 در مبنای 10 که تعداد یک های عدد 429 می باشد.(عدد 429 به شکل باینری برابر است با 110101101 که تعداد یک ها در آن 6 تاست.) مقدار رجیستر R2 برابر 0x00000052 است که عدد دسیمال 82 است. حال اگر مقدار 82 را با خروجی ای که باید بدست می آمد مقایسه کنیم، میبینیم که برابر است:

R2 = R1 \* 3 = 6 \* 3 = 18

R3 = 100 – R2 = 100 – 18 = 82

عکس های شامل ران گرفتن قسمت به قسمت نیز در کنار کد همین فایل می باشد.

منابع:

* <https://www.rapidtables.com>
* <https://www.keil.com>
* <https://developer.arm.com>
* <https://stackoverflow.com>
* <https://www.sciencedirect.com>
* <https://manualzz.com>
* <https://devzone.nordicsemi.com>
* <https://www.cs.uregina.ca>
* <https://ece353.engr.wisc.edu>
* <https://renesasrulz.com>
* <https://www.educba.com>
* <https://www.arm.com>
* <https://iq.opengenus.org>
* <https://en.wikipedia.org>
* https://developer.arm.com
* اسلاید های درس و مطالب گفته شده در کانال درس