**تمرین اول درس ریزپردازنده و زبان اسمبلی**

نگار موقتیان، 9831062

**1.**

صفات قابل استفاده برای directive AREA به شرح زیر می­باشند:

* **READONLY**: مشخص می­کند که در این محدوده نباید چیزی نوشته شود. تمام قسمت­هایی از برنامه که با صفت READLONY مشخص شده­اند توسط Linker در حافظۀ flash و در مجاورت هم قرار خواهند گرفت.
* **READWRITE**: مشخص می­کند که این محدوده ممکن است خوانده شده یا چیزی در آن نوشته شود. تمام قسمت­هایی از برنامه که با صفت READWRITE مشخص شده­اند توسط Linker در حافظۀ SRAM و در مجاورت هم قرار خواهند گرفت.
* **CODE**: مشخص می­کند که دستورات در این محدوده قرار خواهند گرفت. به صورت پیش­فرض محدوده­ای که با این صفت مشخص می­شود به صورت READLONY می­باشد.
* **DATA**: مشخص می­کند که داده­ها، و نه دستورات، در این محدوده قرار خواهند گرفت. به صورت پیش­فرض محدوده­ای که با این صفت مشخص می­شود به صورت READWRITE می­باشد.
* **ALIGN=expression**: به صورت پیش­فرض نواحی با یک مرز 4 بایتی align شده­اند. با استفاده از این دستور ناحیۀ فوق با یک مرز 2expression بایتی align خواهد شد که در آن expression می­تواند عدد صحیحی میان 2 تا 31 باشد.
* **COMDEF**: برای تعریف نواحی مشترکی به کار می­رود که می­توانند حاوی داده یا دستورات باشند. این محدوده باید دقیقاً مشابه با هر محدودۀ دیگر با همین نام که در source file های دیگر نوشته شده­اند تعریف شود. این نواحی مشابه توسط Linker در یک قسمت مشترک از حافظه قرار می­گیرند.
* **COMMON**: یک محدوده مشترک برای نگهداری داده­هاست. در این قسمت نباید هیچ دستور یا داده­ای نوشته شود و توسط Linker به صفر مقداردهی می­شود. مانند قسمت قبل این محدوده­های مشترک با نام یکسان توسط Linker در یک قسمت مشترک از حافظه نگهداری می­شوند، اما اینبار نیازی نیست که اندازۀ آن­ها یکسان باشد و به اندازۀ بزرگ­ترین آن­ها حافظه به این نواحی تخصیص داده می­شود.
* **INTERWORK**: مشخص می­کند که این محدوده برای نوشتن ترکیبی از دستورات ARM و Thumb (که یک instruction set از دستوراتی که توسط ARM پشتیبانی می­شوند است) استفاده خواهد شد.
* **NOINIT**: مشخص می­کند که این محدوده حاوی داده­هایی است که با صفر مقداردهی شده­اند. بنابراین تنها حاوی directive هایی است که برای رزور کردن حافظه، بدون مقداردهی اولیه، استفاده می­شوند.
* **PIC**: مشخص می­کند که دستورات این ناحیه مستقل از موقعیت (position-independent) هستند، بنابراین می­توانند در هر آدرسی بدون نیاز به تغییر اجرا شوند.

**2.**

برنامۀ داده شده را خط به خط بررسی می­کنیم.

LDR R1, =0x11121314

در خط اول برنامه مقدار 0x11121314 توسط شبیه دستور LDR درون رجیستر R1 ذخیره می­شود.

LDR R2, =0x40000

در خط دوم برنامه نیز مقدار 0x40000 توسط شبیه دستور LDR درون رجیستر R2 ذخیره می­شود.

STR R1, [R2]

پس از آن توسط دستور STR مقدار درون رجیستر R1 که برابر است با 0x11121314 درون خانه­ای از حافظه که R2 به آن اشاره می­کند (یعنی خانۀ شماره 0x40000 حافظه) قرار می­گیرد.

LDRB R3, [R2]

در نهایت نیز توسط دستور LDRB یک بایت از خانه­ای از حافظه که R2 به آن اشاره می­کند (یعنی خانۀ شماره 0x40000 حافظه) درون رجیستر R3 ذخیره می­شود.

پس از انجام این عملیات خانه­های حافظه شکل زیر را خواهند داشت:

|  |  |
| --- | --- |
| 0x40003 | 0x11 |
| 0x40002 | 0x12 |
| 0x40001 | 0x13 |
| 0x40000 | 0x14 |

بنابراین مقدار درون خانۀ 0x40000 حافظه برابر با 0x14 و مقدار

درون رجیستر R3 برابر با 0x00000014 خواهد بود.

**3.**

ابتدا بررسی می­کنیم که به طور کلی حافظه­های SRAM و EEPROM چه ویژگی­هایی دارند.

حافظه­های SRAM داده­ها را در مدارهایی دارای latch، یا در انواع ساده­تر، ترانزیستورهایی که به شیوه­ای خاص به یکدیگر متصل شده­اند نگهداری می­کنند. بنابراین بدیهیست که داده­های ذخیره شده در آن­ها با قطع شدن تغذیۀ مدار از دست می­رود. از طرفی آپدیت کردن و خواندن این داده­ها به سادگی و با سرعت بسیار بالایی قابل انجام است.

در مقابل EEPROM یک حافظۀ غیرفرّار و read-only است که برای تغییر دادن داده­های آن نیاز به اعمال جریان الکتریکی (بالاتر از حد معمول) برای پاک کردن داده­های ذخیره شده و نوشتن دوبارۀ داده­ها داریم. بنابراین آپدیت کردن داده­ها در آن زمانبر است و نمی­تواند به صورت مکرر انجام شود.

در نتیجه می­توان گفت نوشتن داده بر EEPROM زمان و انرژی زیادی می برد، به علاوه این حافظه در طول زمان با نوشتن زیاد بر روی آن دچار فرسودگی می شود. بنابراین برای ذخیره سازی داده­هایی که مدام می­خواهیم مقدار آن­ها را تغییر دهیم (مانند متغیرهای برنامه) به SRAM نیاز داریم که سریع است و محدودیتی برای تعداد دفعات آپدیت شدن ندارد. در عوض حافظه­های SRAM به نسبت قیمت بالایی دارند و فرّار هستند. بنابراین برای داده­هایی که آن­ها را به ندرت آپدیت می­کنیم و نیاز داریم که با خاموش شدن سیستم هم آن­ها را از دست ندهیم (مانند تنظیمات کاربر) استفاده از EEPROM ارجحیت دارد.

**4.**

با توجه به قطعه کد داده شده داده­ها به صورت زیر در حافظه قرار خواهند گرفت.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00 | 00 | 00 | 01 | 00 | 00 | 00 | 01 | 0x02000000 |
| 00 | 00 | 02 | 00 | 00 | 00 | 00 | 01 | 0x02000008 |

**5.**

طبق جدول اعداد ASCII، عدد داده شده (69) مربوط به کاراکتر ‘E’ بوده و لذا می­توان آن را برابر با عدد 14 در نظر گرفت. بنابراین خواهیم داشت:

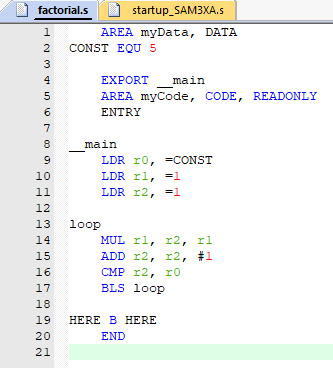
(0100 0101)ASCII = (14)DEC = (0000 0001 0000 0100)unpacked BCD = (0001 0100)packed BCD

**6.**

در این برنامه فاکتوریل با استفاده از یک حلقه حساب شده است.

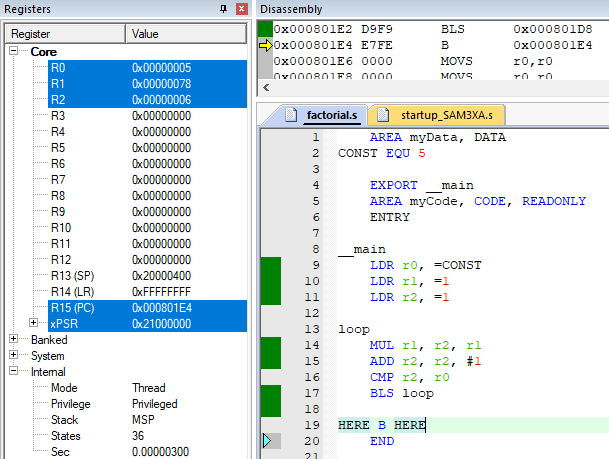
ابتدا یک مقدار ثابت که می­خواهیم فاکتوریل آن را حساب کنیم تعریف کرده و در رجیستر R0 ذخیره می­کنیم. همچنین رجیسترهای R1 و R2 را با مقدار اولیه 1 مقداردهی می کنیم. R1 نتیجۀ نهایی را ذخیره کرده و R2 نقش شمارندۀ حلقه را خواهد داشت (که از 1 تا مقدار ثابتی که می­خواهیم فاکتوریل آن را حساب کنیم تغییر می­کند).

حال در هر مرحله نتیجۀ مرحلۀ قبل (R1) را در شمارندۀ حلقه (R2) ضرب کرده و به عنوان نتیجۀ این مرحله ذخیره می­کنیم. با این کار با خارج شدن از حلقه، نتیجۀ نهایی درون رجیستر R1 خواهد بود.



خروجی برنامه به ازای مقدار ثابت 5 به صورت زیر خواهد بود.

(78)16 = 7 × 16 + 8 = 120 = 5!



**7.**

\* برای اینکه بتوانیم در این برنامه مقادیر آرایۀ داده شده را بخوانیم ابتدا یک فایل با پسوند .ini ساخته شده­است که در آن دستور زیر وجود دارد:

MAP 0x4000D00, 0x4000FFF READ WRITE EXEC

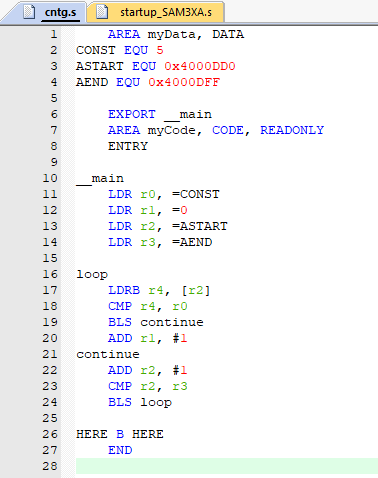
این فایل به تنظیمات دیباگر Keil اضافه شده­است تا در هر شبیه­سازی دسترسی خواندن، نوشتن و اجرای این قسمت از حافظه داشته باشیم.

در برنامۀ نوشته شده ابتدا یک مقدار ثابت (که مقداریست که قرار است با عناصر آرایه مقایسه شود)، آدرس ابتدای آرایه و آدرس انتهای آرایه را تعریف می­کنیم.

سپس در رجیستر R0 مقدار ثابت تعریف شده، در رجیستر R1 عدد صفر (در نهایت نتیجۀ نهایی در این رجیستر قرار خواهد گرفت)، در رجیستر R2 آدرس شروع آرایه (این رجیستر به عنوان شمارندۀ حلقه نیز عمل خواهد کرد و در هر مرحله آدرس خانه­ای از حافظه که در حال بررسی آن هستیم را در خود ذخیره می­کند) و در رجیستر R3 آدرس پایان آرایه را قرار می­دهیم.

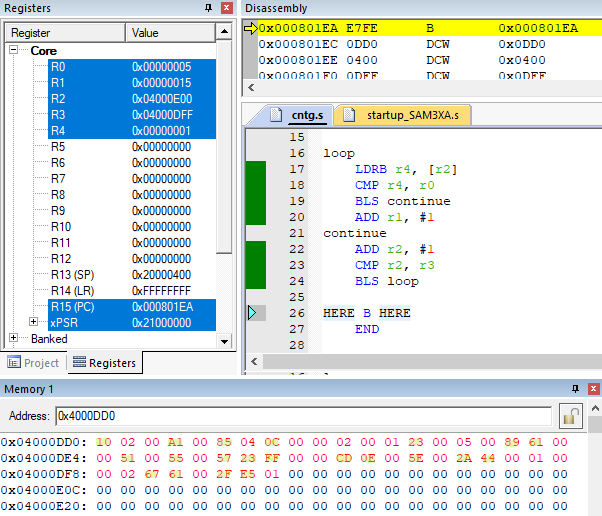
حال در یک حلقه تمام اعضاء آرایه را بررسی می­کنیم. به این صورت که یک بایت خانه­ای از حافظه که آدرس آن برابر با R2 است را در یک رجیستر موقت (R4) لود می­کنیم و آن را با مقدار ثابتی که داشتیم (R0) مقایسه می­کنیم. اگر R4 از R0 بزرگ­تر بود مقدار R1 را یکی افزایش می­دهیم و اگر نه ادامه می­دهیم. بررسی اعضاء آرایه را تا جایی ادامه می­دهیم که به پایان آرایه برسیم.

در نهایت با پایان حلقه تعداد اعداد بزرگ­تر از ثابت در R1 قرار خواهد گرفت.



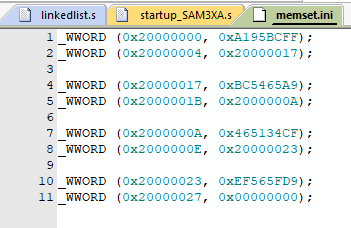
برای بررسی برنامه در قسمت دیباگ به این آرایه مقدارهای تصادفی داده­شده است. همچنین مقدار ثابت برابر با 5 در نظر گرفته شده بود. همانطور که دیده می­شود تعداد اعداد بزرگ­تر از 5 در آرایه (اعداد هایلایت شده با رنگ سبز) برابر است با:

(15)16 = 1 × 16 + 5 = 21



**8.**

\* در این سوال مانند سوال قبل یک فایل با پسوند .ini ساخته شده و به تنظیمات دیباگر Keil اضافه شده­است تا در هر شبیه­سازی خانه­های حافظه مقدار اولیه داشته باشند و نیاز نباشد آن­ها را به صورت دستی تنظیم کنیم. کد موجود در این فایل مانند زیر می­باشد.



در ابتدا یک مقدار ثابت که آدرس اولین خانۀ Linked List است (اشاره­گر head) را تعریف کرده و آن را درون رجیستر R4 ذخیره می­کنیم. همچنین از مقدار آن درون رجیستر R1 یک کپی می­گیریم. این رجیستر ورودی تابع REVERSE در هر مرحله و گره­ای خواهد بود که در حال انجام عملیات بر روی آن هستیم. همچنین توافق می­کنیم که در هر مرحله خروجی تابع را در رجیستر R0 قرار دهیم. حال می­توانیم تابع بازگشتی REVERSE را صدا بزنیم. معادل این تابع در زبان C به صورت زیر خواهد بود:

Node\* reverse(Node\* head) {

if (head == NULL || head -> next == NULL)

return head;

Node\* rest = reverse(head -> next);

Head -> next -> next = head;

Head -> next = NULL;

return rest;

}

طبق صورت سوال می­توان گفت که آدرس 0 برای ما نقش NULL را خواهد داشت. بنابراین در ابتدا بررسی می­کنیم که مقدار R1 (که در کد اسمبلی نوشته شده معادل ورودی head تابع بالا است) مقدار 0 دارد یا خیر. اگر داشت طبق توافقی که درمورد خروجی تابع داشتیم مقدار R1 را درون R0 می­ریزیم و با دستور BX، return می­کنیم.

اگر نه باید بررسی کنیم که قسمتی از گره که به گره بعدی اشاره می­کند NULL است یا خیر (که در این صورت گره آخرین گره خواهد بود). برای این کار ابتدا 4 بایت از جایی که R1 به آن اشاره می­کند را از حافظه خوانده و درون R2 قرار می­دهیم (داده درون گره فعلی). سپس 4 بایت بعدی را نیز خوانده و درون رجیستر R3 قرار می­دهیم (آدرس گره بعدی). حال می­توانیم R3 را با 0 مقایسه کنیم. اگر مقدار آن برابر با صفر بود باز هم مقدار R1 را درون R0 می­ریزیم و با دستور BX، return می­کنیم.

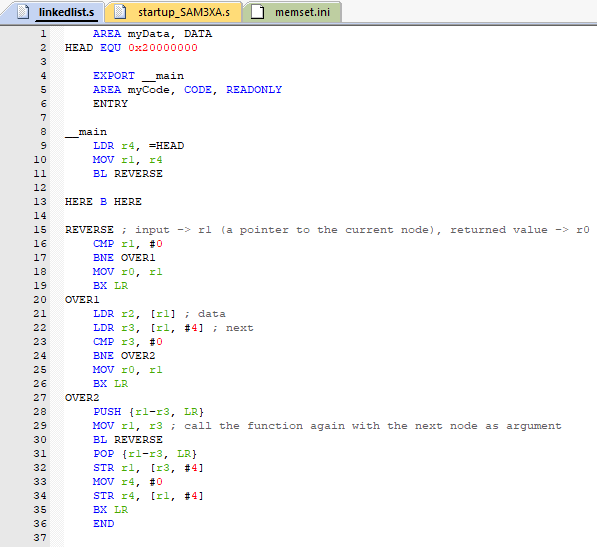
حال باید به صورت بازگشتی دوباره تابع را صدا بزنیم. برای این کار ابتدا مقدار رجیسترهایی که ست کرده بودیم را داخل استک PUSH می­کنیم تا پس از برگشتن از تابع بتوانیم مقادیر آن­ها را بازیابی کنیم.

سپس طبق قرارداد ورودی­ای که تابع باید داشته باشد را درون رجیستر R0 می­ریزیم. این ورودی آدرس گره بعدی است که پیش از این آن را درون رجیستر R3 ذخیره کرده بودیم.

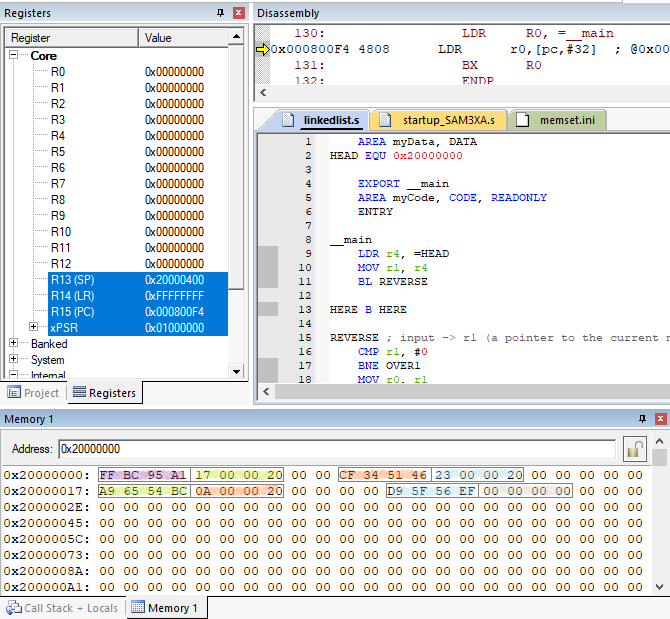
حال که از تابع برگشتیم مقدار رجیسترهایی را که PUSH کرده بودیم با دستور POP بازیابی می­کنیم. در این لحظه مقداری که تابع برگردانده (معادل با متغیر rest در تابع بالا) در رجیستر R0 قرار دارد.

گفتیم R3 حاوی آدرس گره بعدی است، بنابراین خانۀ شماره R3 + 4 معادل با head -> next -> next خواهد بود. بنابراین مقدار R1 که گره فعلی است را در این خانه ذخیره می­کنیم. به علاوه خانۀ شماره R1 + 4 معادل با head -> next را برابر با 0 قرار می­دهیم و با دستور BX، return می­کنیم.

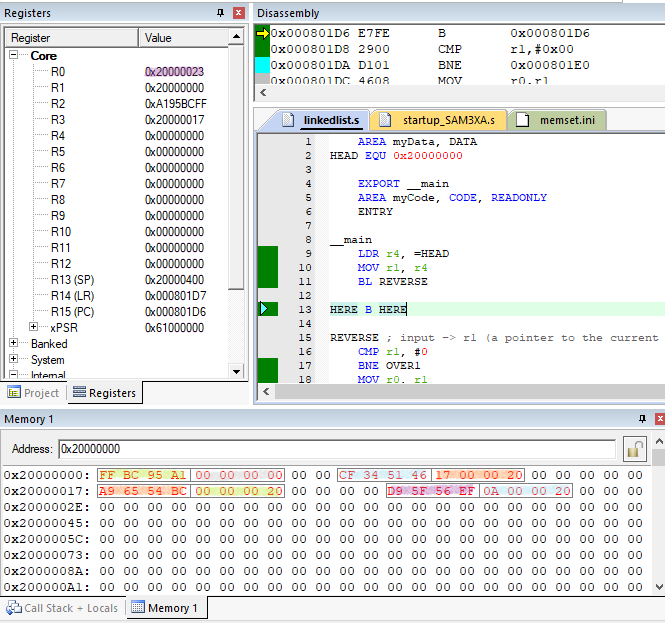
در نهایت رجیستر R0 مقدار اولین خانۀ Linked List برعکس شده را در خود خواهد داشت.



با توجه به مقادیر پیش فرض حافظه که در فایل memset.ini تعریف کرده بودیم در ابتدای برنامه شکل حافظه مانند زیر خواهد بود (4 بایت دوم هر گره به 4 بایت اول گره همرنگ خودش اشاره می­کند. همچنین رنگ بنفش نشان­دهندۀ شروع و رنگ خاکستری نشان­دهندۀ پایان لیست است).



و در پایان برنامه شکل حافظه مانند زیر خواهد بود. همانطور که مشاهده می­شود ترتیب گره­های Linked List برعکس شده­است و آدرس اولین گره لیست برعکس شده در رجیستر R0 قرار دارد.



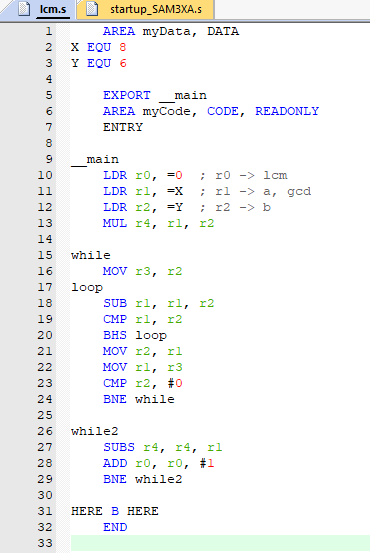
**9.**

کد نوشته شده دقیقاً مشابه کد موجود در متن تمرین می­باشد.

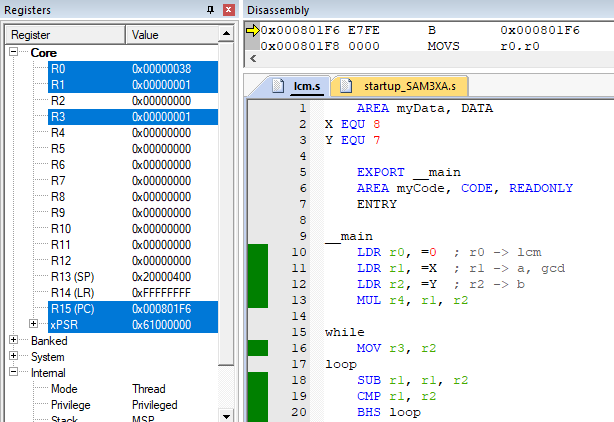
تنها نکتۀ آن محاسبه باقی­مانده است که از طریق تفریق­های متوالی در یک حلقه (با برچسب loop) انجام شده­است. انگار که هر بار از a (معادل R1) به اندازۀ b (معادل R2) کم می­کنیم (اهمیتی ندارد که مقدار موجود در a خراب شود، زیرا قرار است در خط بعد آن را با مقدار t (معادل R3) جایگزین کنیم). این تفریق­ها را تا جایی تکرار می­کنیم که مقدار موجود در a کمتر از مقدار b شود.

برای تقسیم در آخر برنامه نیز از همین روش استفاده می­کنیم (مطمئن هستیم که تقسیم باقی مانده ندارد). با این تفاوت که هر بار مقدار یک متغیر شمارنده (R0) را با هر تفریق افزایش می­دهیم تا خارج قسمت تقسیم را محاسبه کنیم.

در نهایت R0 دربردارندۀ مقدار ک.م.م. خواهد بود. همچنین R1 نیز مقدار ب.م.م. را ذخیره خواهد کرد.



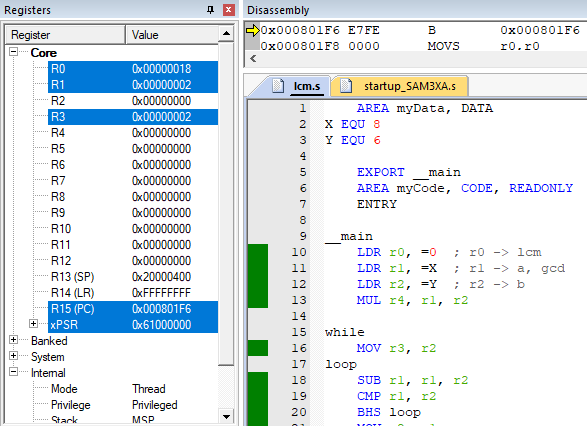
نمونۀ خروجی برنامه مانند زیر خواهد بود.



در نمونۀ بالا دو عدد نسبت به هم اول هستند. بنابراین ب.م.م. آن­ها برابر با 1 و ک.م.م. آن­ها برابر است با:

7 × 8 = 56 = 3 × 16 + 8 = (38)16

نمونۀ دیگری از خروجی برنامه مانند زیر خواهد بود.



در این نمونه ب.م.م. دو عدد برابر با 2 و ک.م.م. آن­ها برابر است با:

6 × 8 / 2 = 24 = 1 × 16 + 8 = (18)16

**10.**

الف) با توجه به متن این قسمت از تمرین می­توان گفت که مقادیر درون رجیسترها می­توانند منفی باشند. بنابراین برای دستورات branch این قسمت از دستوراتی که برای اعداد علامتدار وجود دارند (مانند BLT) استفاده می­کنیم.

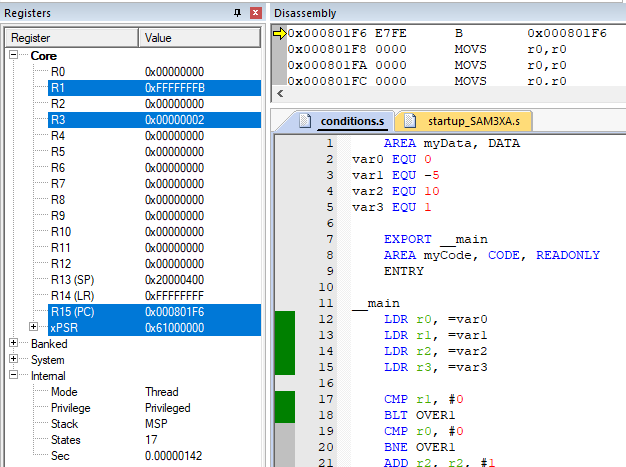
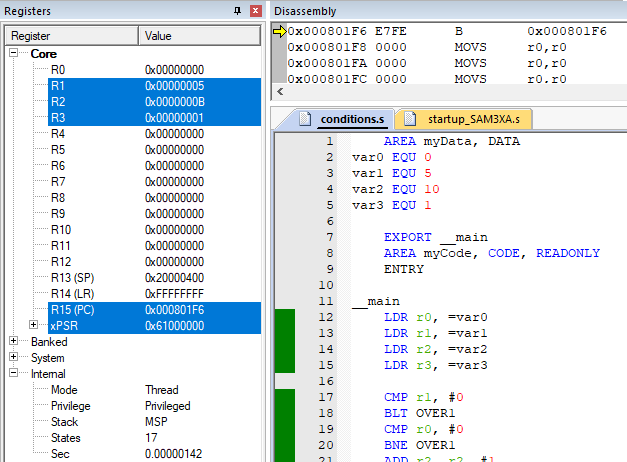
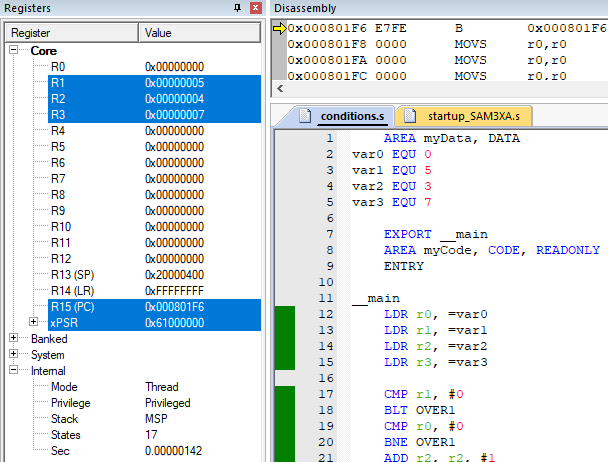
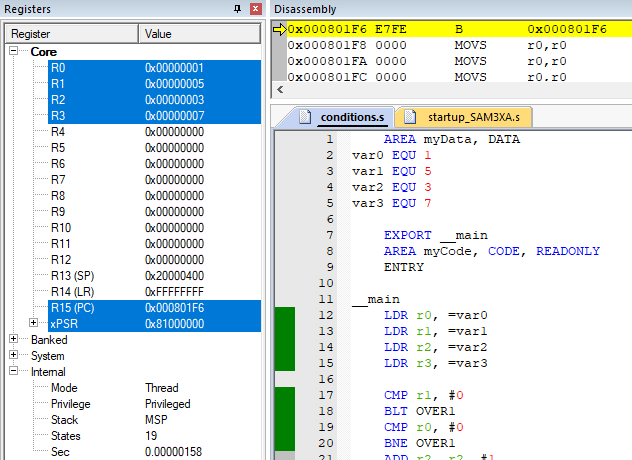
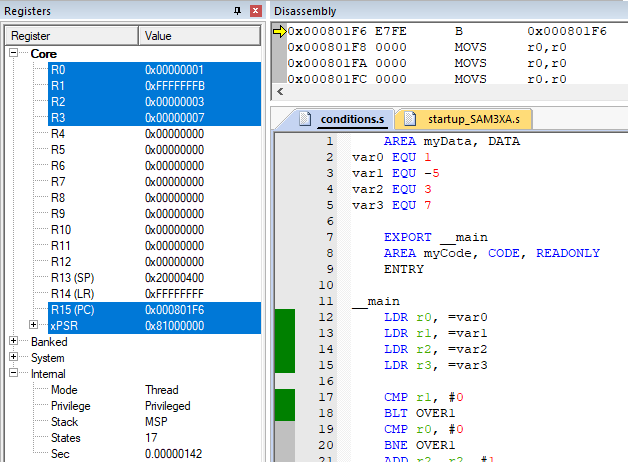
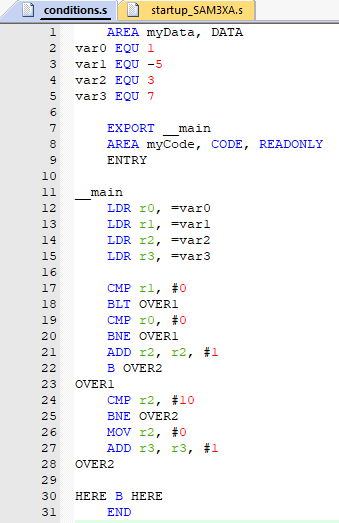
در ادامه کد مربوط به این قسمت و خروجی برنامه به ازای مقادیر مختلف موجود در رجیسترها آمده­است.

در نمونۀ اول هیچ یک از شرط­ها برقرار نیستند.

در نمونه­های دوم و سوم یکی از شرط­های if اول برقرار بوده و دیگری برقرار نیست پس هیچ یک از شرط­ها اجرا نمی­شوند.

در نمونۀ چهارم هر دو شرط برقراراند اما به دلیل وجود دستور else if (و نه if) تنها کد مربوط به شرط اول اجرا می­شود.

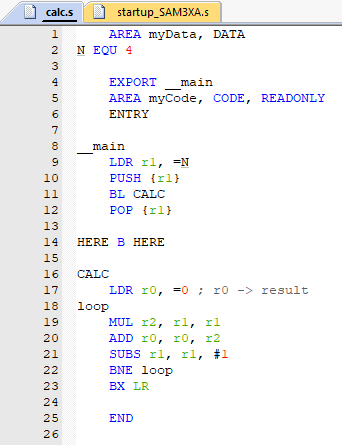
در نمونۀ پنجم شرط اول برقرار نیست و کد مربوط به شرط دوم اجرا می­شود.



ب) در این قسمت یک تابع با نام CALC داریم که مربع تمام اعداد 1 تا n را جمع می­کند.

در برنامۀ زیر مقدار ثابت N را تعریف کرده و آن را درون رجیستر R1 قرار داده­ایم. به دلیل این که مقدار موجود در R1 در تابع CALC خراب می­شود پیش از صدا زدن این تابع آن را درون استک PUSH کرده و در نهایت آن را POP می­کنیم.

درون تابع CALC نیز مقدار رجیستر R0 را در ابتدا برابر با صفر قرار داده و در یک حلقه هر بار مربع مقدار شمارنده (R1 که از N تا 1 تغییر می­کند) را با آن جمع می­کنیم. در انتهای این حلقه مقدار R0 برابر با result نهایی است، بنابراین آن را با دستور BX برمی­گردانیم.



مقدار خروجی برنامه نیز به ازای n = 4 مانند زیر خواهد بود.

12 + 22 + 32 + 42 = 1 + 4 + 9 + 16 = 30 = 1 × 16 + 14 = (1E)16

