تمرین اول درس ریزپردازنده و زبان اسمبلی

نگار موقتیان، ۹۸۳۱۰۶۲

۱.

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 حافظهٔ 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 حافظهٔ 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 ۴ بایتی align شده‌اند. با استفاده از این دستور ناحیهٔ فوق با یک مرز $2^{expression}$ بایتی align خواهد شد که در آن expression می‌تواند عدد صحیحی میان ۲ تا ۳۱ باشد.
- **COMDEF**: برای تعریف نواحی مشترکی به کار می‌رود که می‌توانند حاوی داده یا دستورات باشند. این محدوده باید دقیقاً مشابه با هر محدودهٔ 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 اندازهٔ آن‌ها یکسان باشد و به اندازهٔ 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.

۲.

برنامهٔ 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 خانهٔ 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 خانهٔ شماره 0x40000 حافظه) درون رجیستر R3 ذخیره می‌شود.

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

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

بنابراین مقدار درون خانهٔ 0x40000 حافظه برابر با 0x14 و مقدار درون رجیستر R3 برابر با 0x00000014 خواهد بود.

.۳

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 تغذیهٔ 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 حافظهٔ 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 دوبارهٔ 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.

.۴

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

0x02000000	01	00	00	00	01	00	00	00
0x02000008	01	00	00	00	00	02	00	00

.۵

طبق جدول اعداد ASCII، عدد داده شده (69) مربوط به کاراکتر ’E‘ بوده و لذا می‌توان آن را برابر با عدد ۱۴ در نظر گرفت. بنابراین خواهیم داشت:

$$(0100\ 0101)_{ASCII} = (14)_{DEC} = (0000\ 0001\ \ 0000\ 0100)_{unpacked\ BCD} = (0001\ 0100)_{packed\ BCD}$$

۶.

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 ۱ مقداردهی می کنیم. R1 نتیجهٔ نهایی را ذخیره کرده و R2 نقش شمارندهٔ حلقه را خواهد داشت (که از ۱ تا مقدار ثابتی که می‌خواهیم فاکتوریل آن را حساب کنیم تغییر می‌کند).

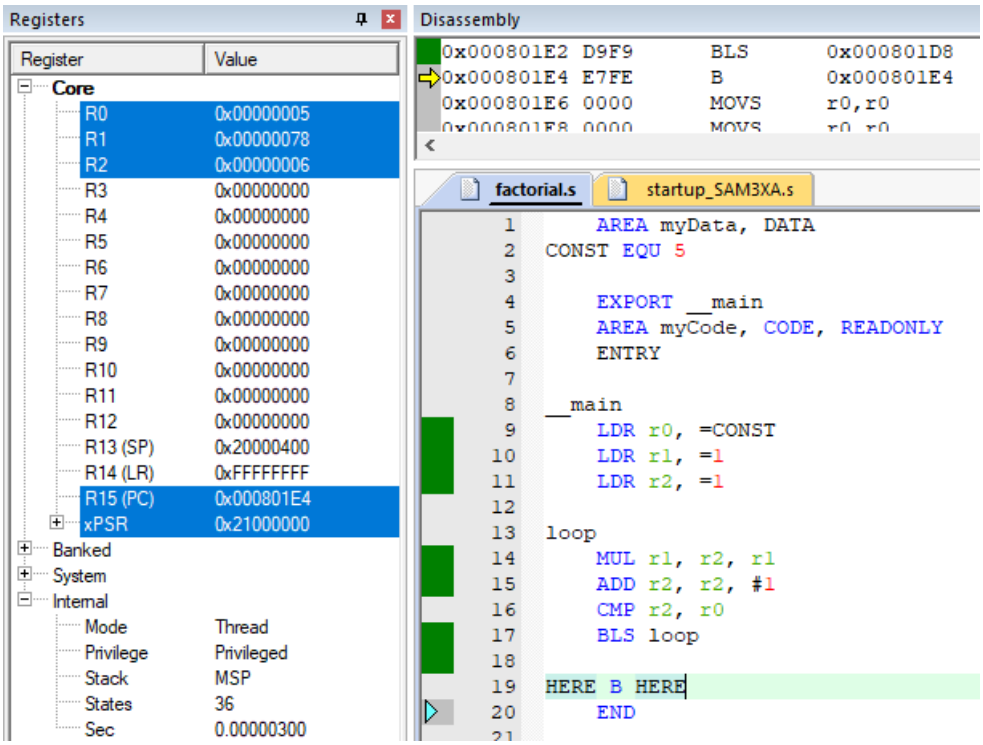
حال در هر مرحله نتیجهٔ مرحلهٔ قبل (R1) را در شمارندهٔ حلقه (R2) ضرب کرده و به عنوان نتیجهٔ این مرحله ذخیره می‌کنیم. با این کار با خارج شدن از حلقه، نتیجهٔ نهایی درون رجیستر R1 خواهد بود.

factorial.s | startup_SAM3XA.s

```
1     AREA myData, DATA
2  CONST EQU 5
3
4     EXPORT __main
5     AREA myCode, CODE, READONLY
6     ENTRY
7
8  __main
9     LDR r0, =CONST
10    LDR r1, =1
11    LDR r2, =1
12
13 loop
14    MUL r1, r2, r1
15    ADD r2, r2, #1
16    CMP r2, r0
17    BLS loop
18
19 HERE B HERE
20    END
21
```

خروجی برنامه به ازای مقدار ثابت ۵ به صورت زیر خواهد بود.

$(78)_{16} = 7 \times 16 + 8 = 120 = 5!$



۷.

* برای اینکه بتوانیم در این برنامه مقادیر آرایهٔ 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 برنامهٔ 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 نتیجهٔ نهایی در این رجیستر قرار خواهد گرفت)، در رجیستر R2 آدرس شروع آرایه (این رجیستر به عنوان شمارندهٔ 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.

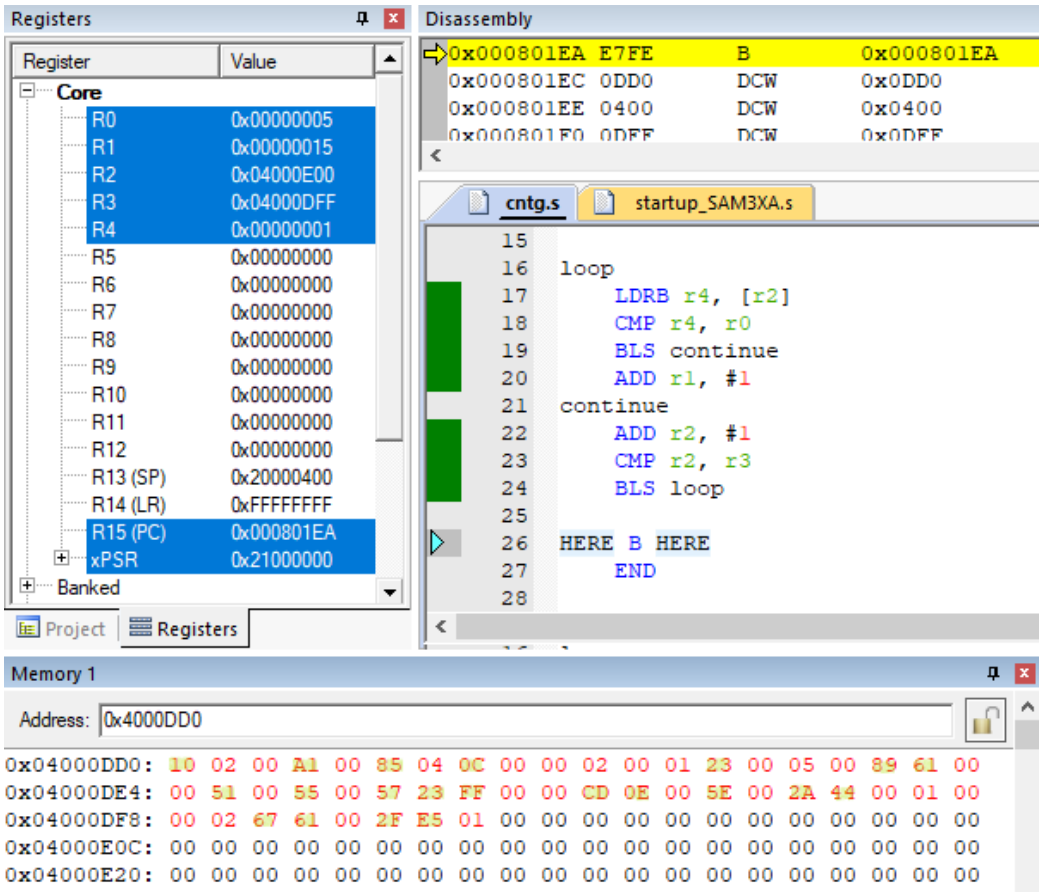
در نهایت با پایان حلقه تعداد اعداد بزرگ‌تر از ثابت در R1 قرار خواهد گرفت.

cntg.s | startup_SAM3XA.s

```
1     AREA myData, DATA
2  CONST EQU 5
3  ASTART EQU 0x4000DD0
4  AEND EQU 0x4000DFF
5
6     EXPORT __main
7     AREA myCode, CODE, READONLY
8     ENTRY
9
10 __main
11    LDR r0, =CONST
12    LDR r1, =0
13    LDR r2, =ASTART
14    LDR r3, =AEND
15
16 loop
17    LDRB r4, [r2]
18    CMP r4, r0
19    BLS continue
20    ADD r1, #1
21 continue
22    ADD r2, #1
23    CMP r2, r3
24    BLS loop
25
26 HERE B HERE
27    END
28
```

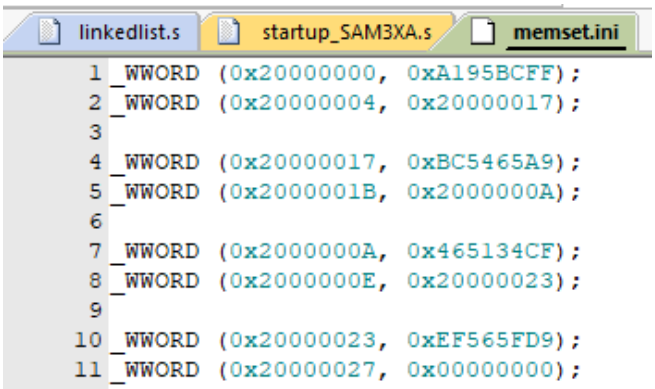
برای بررسی برنامه در قسمت دیباگ به این آرایه مقدارهای تصادفی داده‌شده است. همچنین مقدار ثابت برابر با ۵ در نظر گرفته شده بود. همانطور که دیده می‌شود تعداد اعداد بزرگ‌تر از ۵ در آرایه (اعداد هایلایت شده با رنگ سبز) برابر است با:

$(15)_{16} = 1 \times 16 + 5 = 21$



۸

* 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 خانهٔ 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 ۴ بایت از جایی که R1 به آن اشاره می‌کند را از حافظه خوانده و درون R2 قرار می‌دهیم (داده درون گره فعلی). سپس ۴ 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.

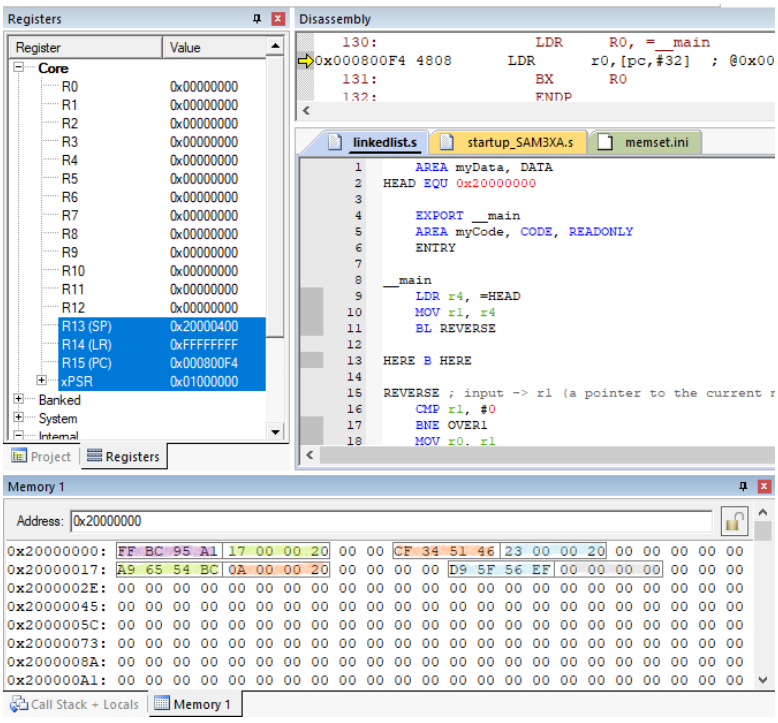
گفتیم R3 حاوی آدرس گره بعدی است، بنابراین خانهٔ شماره R3 + 4 معادل با head -> next -> next خواهد بود. بنابراین مقدار R1 که گره فعلی است را در این خانه ذخیره می‌کنیم. به علاوه خانهٔ شماره R1 + 4 معادل با head -> next را برابر با 0 قرار می‌دهیم و با دستور BX، return می‌کنیم.

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

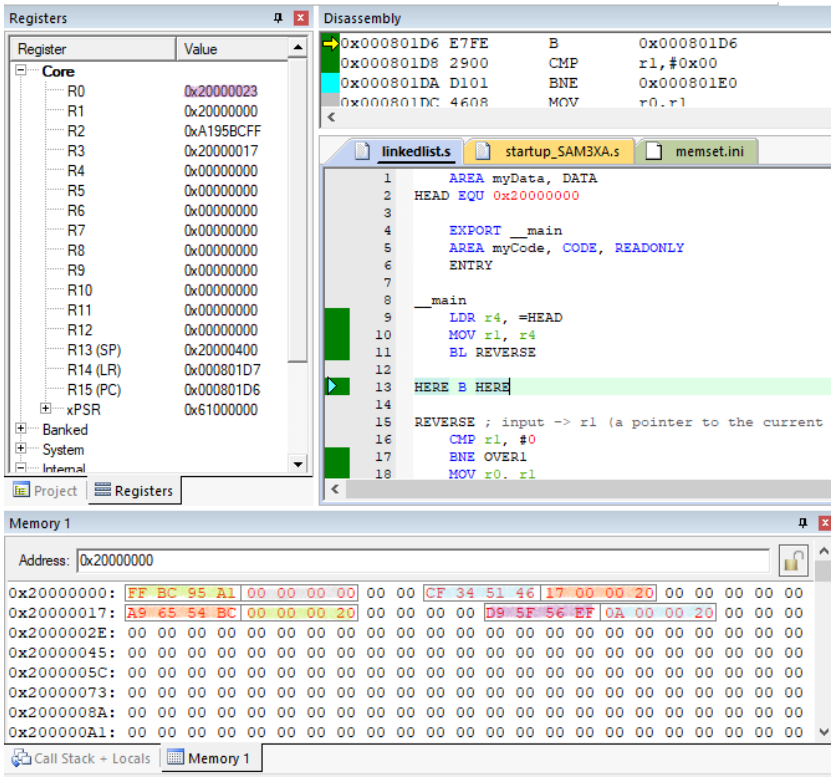
linkedlist.s | startup_SAM3XA.s | memset.ini

```
1     AREA myData, DATA
2  HEAD EQU 0x20000000
3
4     EXPORT __main
5     AREA myCode, CODE, READONLY
6     ENTRY
7
8  __main
9     LDR r4, =HEAD
10    MOV r1, r4
11    BL REVERSE
12
13 HERE B HERE
14
15 REVERSE ; input -> r1 (a pointer to the current node), returned value -> r0
16    CMP r1, #0
17    BNE OVER1
18    MOV r0, r1
19    BX LR
20 OVER1
21    LDR r2, [r1] ; data
22    LDR r3, [r1, #4] ; next
23    CMP r3, #0
24    BNE OVER2
25    MOV r0, r1
26    BX LR
27 OVER2
28    PUSH {r1-r3, LR}
29    MOV r1, r3 ; call the function again with the next node as argument
30    BL REVERSE
31    POP {r1-r3, LR}
32    STR r1, [r3, #4]
33    MOV r4, #0
34    STR r4, [r1, #4]
35    BX LR
36    END
37
```

با توجه به مقادیر پیش فرض حافظه که در فایل memset.ini تعریف کرده بودیم در ابتدای برنامه شکل حافظه مانند زیر خواهد بود (۴ بایت دوم هر گره به ۴ بایت اول گره همرنگ خودش اشاره می‌کند. همچنین رنگ بنفش نشان‌دهندهٔ شروع و رنگ خاکستری نشان‌دهندهٔ 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.



۹.

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

تنها نکتهٔ 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 اندازهٔ 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.

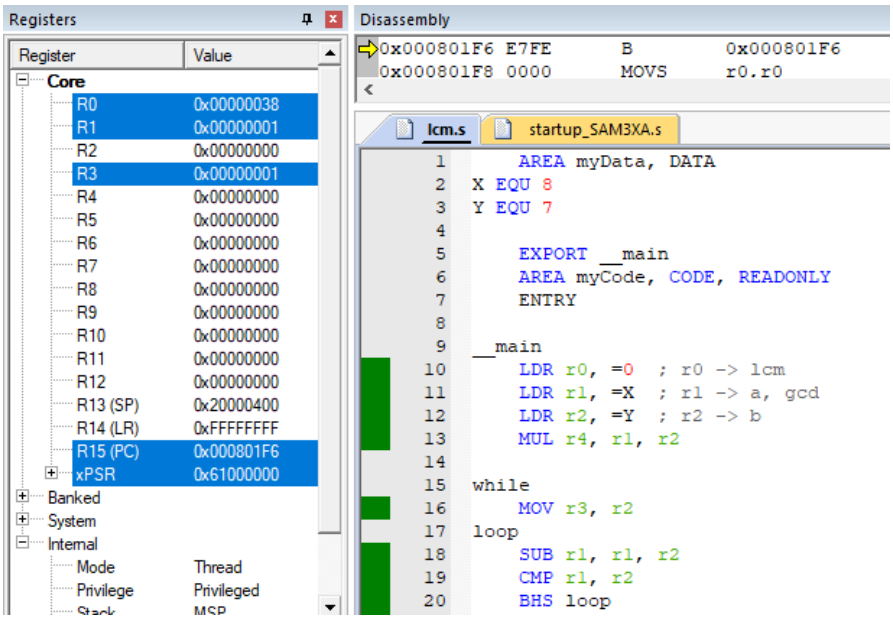
برای تقسیم در آخر برنامه نیز از همین روش استفاده می‌کنیم (مطمئن هستیم که تقسیم باقی مانده ندارد). با این تفاوت که هر بار مقدار یک متغیر شمارنده (R0) را با هر تفریق افزایش می‌دهیم تا خارج قسمت تقسیم را محاسبه کنیم.

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

lcm.s | startup_SAM3XA.s

```
1   AREA myData, DATA
2   X EQU 8
3   Y EQU 6
4
5       EXPORT __main
6       AREA myCode, CODE, READONLY
7       ENTRY
8
9   __main
10      LDR r0, =0  ; r0 -> lcm
11      LDR r1, =X  ; r1 -> a, gcd
12      LDR r2, =Y  ; r2 -> b
13      MUL r4, r1, r2
14
15  while
16      MOV r3, r2
17  loop
18      SUB r1, r1, r2
19      CMP r1, r2
20      BHS loop
21      MOV r2, r1
22      MOV r1, r3
23      CMP r2, #0
24      BNE while
25
26  while2
27      SUBS r4, r4, r1
28      ADD r0, r0, #1
29      BNE while2
30
31  HERE B HERE
32      END
33
```

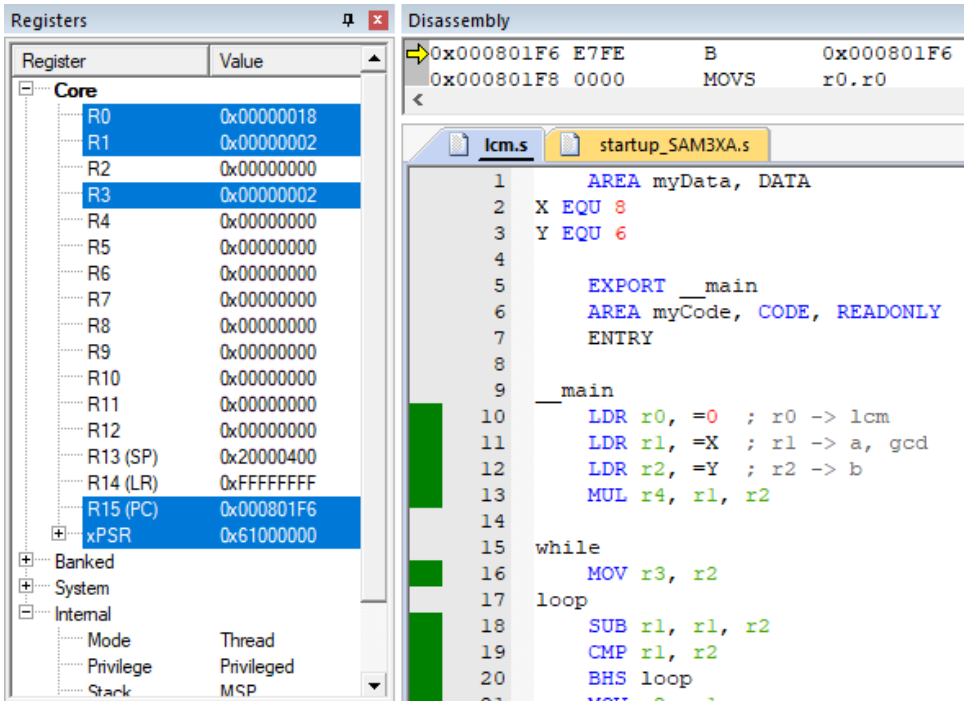
نمونهٔ خروجی برنامه مانند زیر خواهد بود.



در نمونهٔ بالا دو عدد نسبت به هم اول هستند. بنابراین ب.م.م. آن‌ها برابر با ۱ و ک.م.م. آن‌ها برابر است با:

$7 \times 8 = 56 = 3 \times 16 + 8 = (38)_{16}$

نمونهٔ دیگری از خروجی برنامه مانند زیر خواهد بود.



در این نمونه ب.م.م. دو عدد برابر با ۲ و ک.م.م. آن‌ها برابر است با:

$6 \times 8 / 2 = 24 = 1 \times 16 + 8 = (18)_{16}$

۱۰.

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

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

conditions.s | startup_SAM3XA.s

```
1     AREA myData, DATA
2  var0 EQU 1
3  var1 EQU -5
4  var2 EQU 3
5  var3 EQU 7
6
7     EXPORT __main
8     AREA myCode, CODE, READONLY
9     ENTRY
10
11 __main
12    LDR r0, =var0
13    LDR r1, =var1
14    LDR r2, =var2
15    LDR r3, =var3
16
17    CMP r1, #0
18    BLT OVER1
19    CMP r0, #0
20    BNE OVER1
21    ADD r2, r2, #1
22    B OVER2
23 OVER1
24    CMP r2, #10
25    BNE OVER2
26    MOV r2, #0
27    ADD r3, r3, #1
28 OVER2
29
30 HERE B HERE
31    END
```

Registers

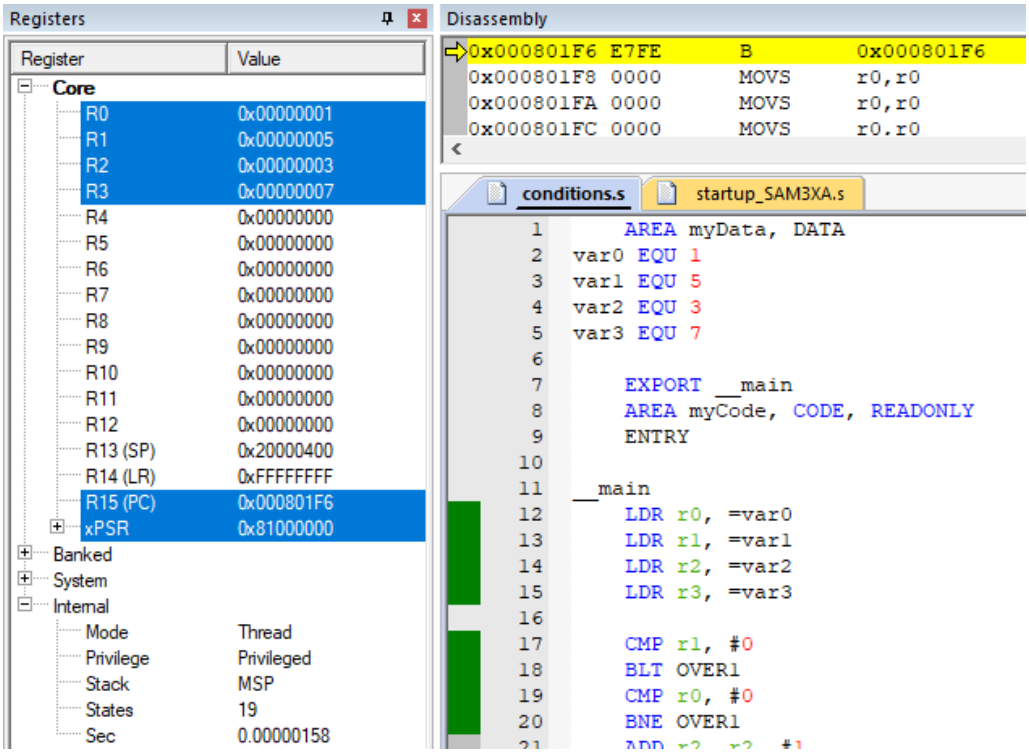
Register	Value
Core	
R0	0x00000001
R1	0xFFFFFFFB
R2	0x00000003
R3	0x00000007
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x000801F6
xPSR	0x81000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	17
Sec	0.00000142

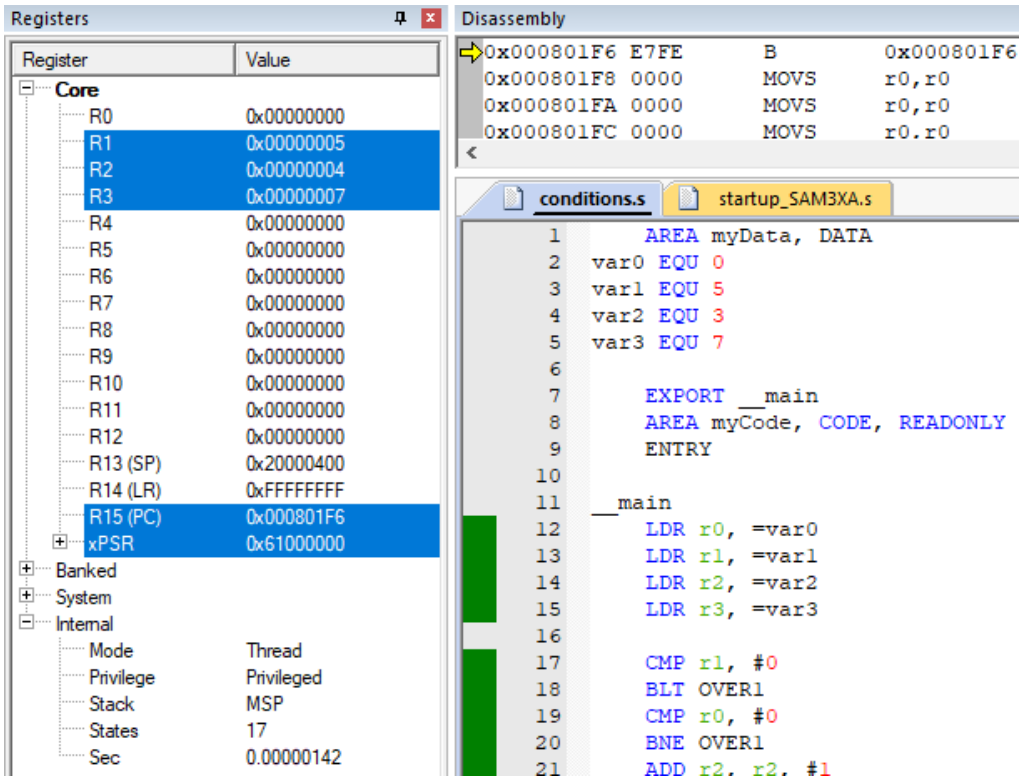
Disassembly

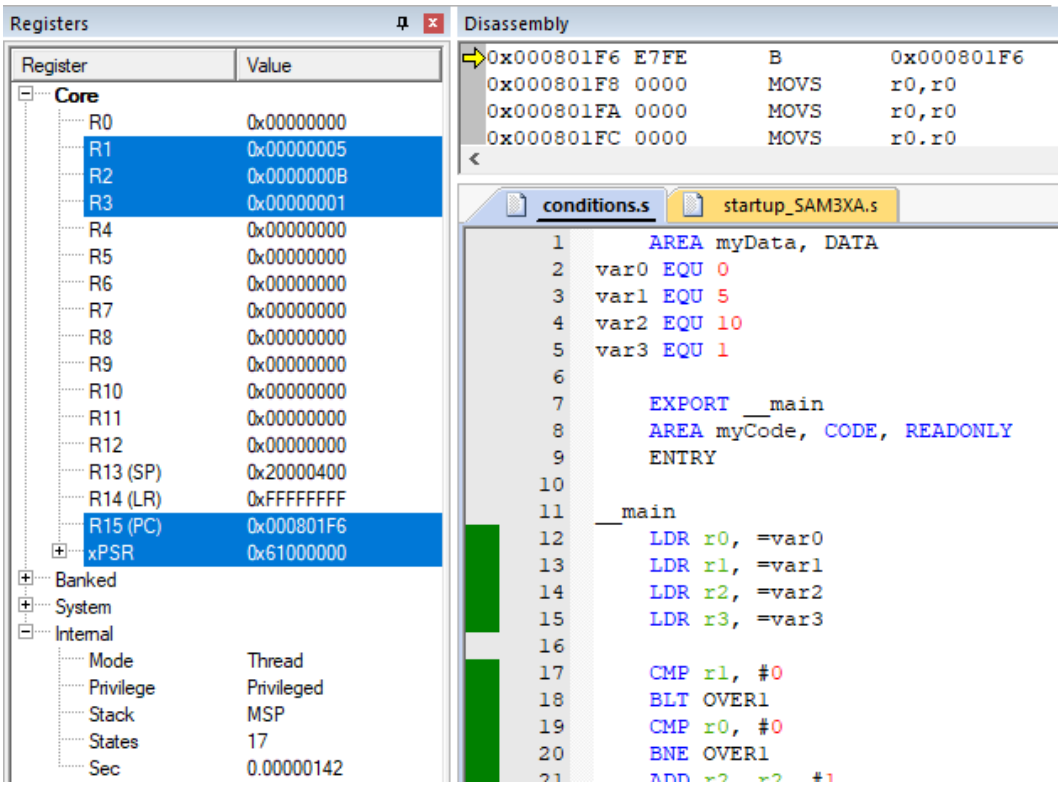
```
0x000801F6 E7FE      B       0x000801F6
0x000801F8 0000      MOVS    r0,r0
0x000801FA 0000      MOVS    r0,r0
0x000801FC 0000      MOVS    r0,r0
```

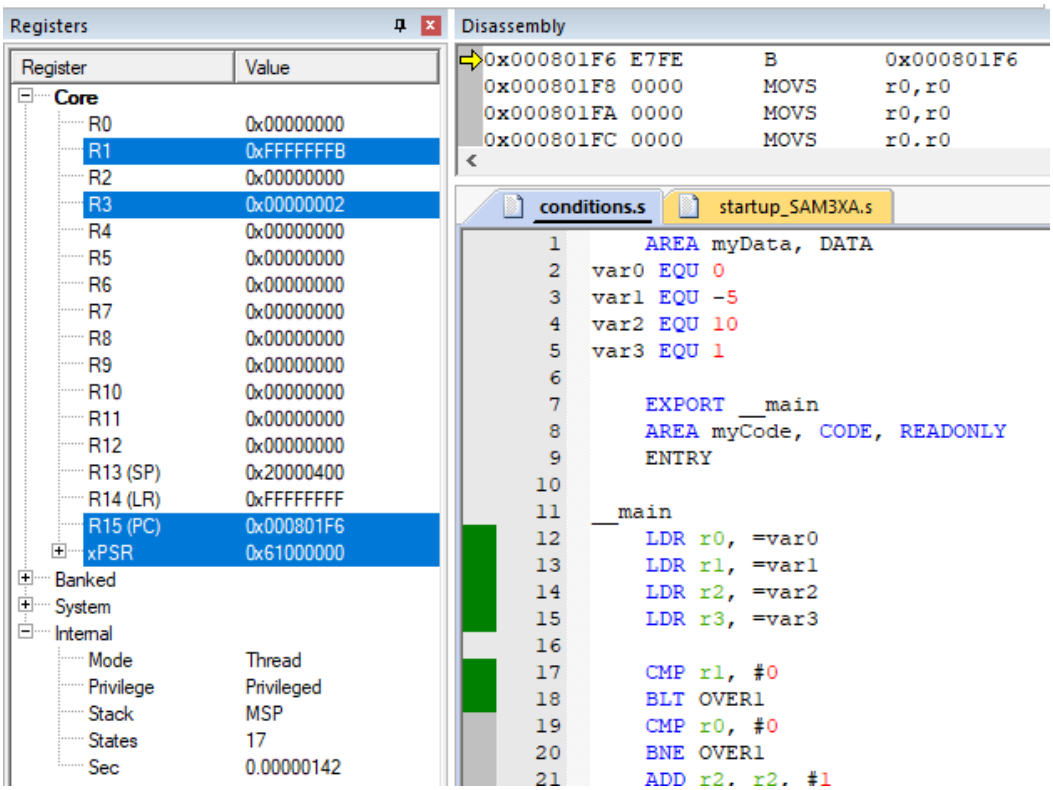
conditions.s | startup_SAM3XA.s

```
1     AREA myData, DATA
2  var0 EQU 1
3  var1 EQU -5
4  var2 EQU 3
5  var3 EQU 7
6
7     EXPORT __main
8     AREA myCode, CODE, READONLY
9     ENTRY
10
11 __main
12    LDR r0, =var0
13    LDR r1, =var1
14    LDR r2, =var2
15    LDR r3, =var3
16
17    CMP r1, #0
18    BLT OVER1
19    CMP r0, #0
20    BNE OVER1
```


Registers
Register
Value
Core
R0 0x00000001
R1 0x00000005
R2 0x00000003
R3 0x00000007
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20000400
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x000801F6
xPSR 0x81000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 19
Sec 0.00000158
Disassembly
0x000801F6 E7FE B 0x000801F6
0x000801F8 0000 MOVS r0,r0
0x000801FA 0000 MOVS r0,r0
0x000801FC 0000 MOVS r0,r0
conditions.s
startup_SAM3XA.s
1 AREA myData, DATA
2 var0 EQU 1
3 var1 EQU 5
4 var2 EQU 3
5 var3 EQU 7
6
7 EXPORT __main
8 AREA myCode, CODE, READONLY
9 ENTRY
10
11 __main
12 LDR r0, =var0
13 LDR r1, =var1
14 LDR r2, =var2
15 LDR r3, =var3
16
17 CMP r1, #0
18 BLT OVER1
19 CMP r0, #0
20 BNE OVER1


Registers
Register
Value
Core
R0 0x00000000
R1 0x00000005
R2 0x00000004
R3 0x00000007
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20000400
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x000801F6
xPSR 0x61000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 17
Sec 0.00000142
Disassembly
0x000801F6 E7FE B 0x000801F6
0x000801F8 0000 MOVS r0,r0
0x000801FA 0000 MOVS r0,r0
0x000801FC 0000 MOVS r0,r0
conditions.s
startup_SAM3XA.s
1 AREA myData, DATA
2 var0 EQU 0
3 var1 EQU 5
4 var2 EQU 3
5 var3 EQU 7
6
7 EXPORT __main
8 AREA myCode, CODE, READONLY
9 ENTRY
10
11 __main
12 LDR r0, =var0
13 LDR r1, =var1
14 LDR r2, =var2
15 LDR r3, =var3
16
17 CMP r1, #0
18 BLT OVER1
19 CMP r0, #0
20 BNE OVER1
21 ADD r2, r2, #1


Registers
Register
Value
Core
R0 0x00000000
R1 0x00000005
R2 0x0000000B
R3 0x00000001
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20000400
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x000801F6
xPSR 0x61000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 17
Sec 0.00000142
Disassembly
0x000801F6 E7FE B 0x000801F6
0x000801F8 0000 MOVS r0,r0
0x000801FA 0000 MOVS r0,r0
0x000801FC 0000 MOVS r0,r0
conditions.s
startup_SAM3XA.s
1 AREA myData, DATA
2 var0 EQU 0
3 var1 EQU 5
4 var2 EQU 10
5 var3 EQU 1
6
7 EXPORT __main
8 AREA myCode, CODE, READONLY
9 ENTRY
10
11 __main
12 LDR r0, =var0
13 LDR r1, =var1
14 LDR r2, =var2
15 LDR r3, =var3
16
17 CMP r1, #0
18 BLT OVER1
19 CMP r0, #0
20 BNE OVER1


Registers
Register
Value
Core
R0 0x00000000
R1 0xFFFFFFFB
R2 0x00000000
R3 0x00000002
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20000400
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x000801F6
xPSR 0x61000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 17
Sec 0.00000142
Disassembly
0x000801F6 E7FE B 0x000801F6
0x000801F8 0000 MOVS r0,r0
0x000801FA 0000 MOVS r0,r0
0x000801FC 0000 MOVS r0,r0
conditions.s
startup_SAM3XA.s
1 AREA myData, DATA
2 var0 EQU 0
3 var1 EQU -5
4 var2 EQU 10
5 var3 EQU 1
6
7 EXPORT __main
8 AREA myCode, CODE, READONLY
9 ENTRY
10
11 __main
12 LDR r0, =var0
13 LDR r1, =var1
14 LDR r2, =var2
15 LDR r3, =var3
16
17 CMP r1, #0
18 BLT OVER1
19 CMP r0, #0
20 BNE OVER1
21 ADD r2, r2, #1

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

در برنامهٔ 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.

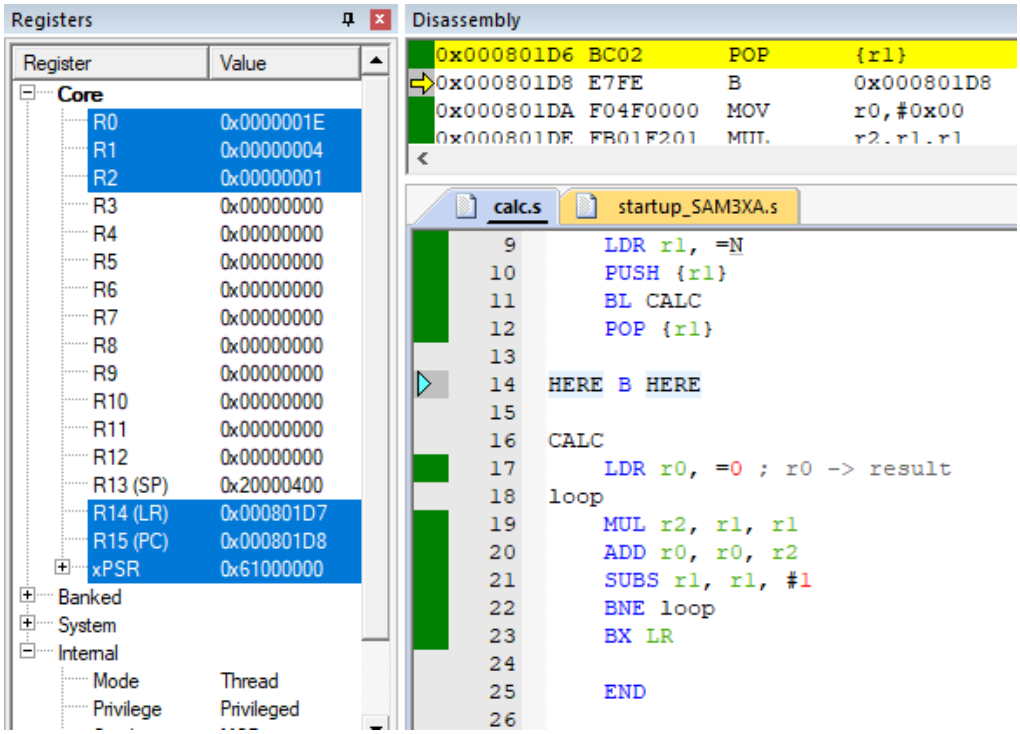
درون تابع CALC نیز مقدار رجیستر R0 را در ابتدا برابر با صفر قرار داده و در یک حلقه هر بار مربع مقدار شمارنده (R1 که از N تا 1 تغییر می‌کند) را با آن جمع می‌کنیم. در انتهای این حلقه مقدار R0 برابر با result نهایی است، بنابراین آن را با دستور BX برمی‌گردانیم.

calc.s | startup_SAM3XA.s

```
1      AREA myData, DATA
2  N EQU 4
3
4      EXPORT __main
5      AREA myCode, CODE, READONLY
6      ENTRY
7
8  __main
9      LDR r1, =N
10     PUSH {r1}
11     BL CALC
12     POP {r1}
13
14 HERE B HERE
15
16 CALC
17     LDR r0, =0 ; r0 -> result
18 loop
19     MUL r2, r1, r1
20     ADD r0, r0, r2
21     SUBS r1, r1, #1
22     BNE loop
23     BX LR
24
25     END
26
```

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

$12 + 22 + 32 + 42 = 1 + 4 + 9 + 16 = 30 = 1 \times 16 + 14 = (1E)_{16}$


Registers
Register
Value
Core
R0 0x0000001E
R1 0x00000004
R2 0x00000001
R3 0x00000000
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20000400
R14 (LR) 0x000801D7
R15 (PC) 0x000801D8
xPSR 0x61000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Disassembly
0x000801D6 BC02 POP {r1}
0x000801D8 E7FE B 0x000801D8
0x000801DA F04F0000 MOV r0,#0x00
calc.s
startup_SAM3XA.s
9 LDR r1, =N
10 PUSH {r1}
11 BL CALC
12 POP {r1}
13
14 HERE B HERE
15
16 CALC
17 LDR r0, =0 ; r0 -> result
18 loop
19 MUL r2, r1, r1
20 ADD r0, r0, r2
21 SUBS r1, r1, #1
22 BNE loop
23 BX LR
24
25 END
26