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. A32به 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 performance را 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 می-انجامد. 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.

```
START

    ;Initialize R0 by 0 (summation result will be stored in R1)
    MOV R1, #0

    ;Rename R0 To COUNT and initialize by 8
COUNT RN R0
    MOV COUNT, #8

    ;Rename R2 To STEP and initialize by 1
STEP RN R2
    MOV STEP, #1
```

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)

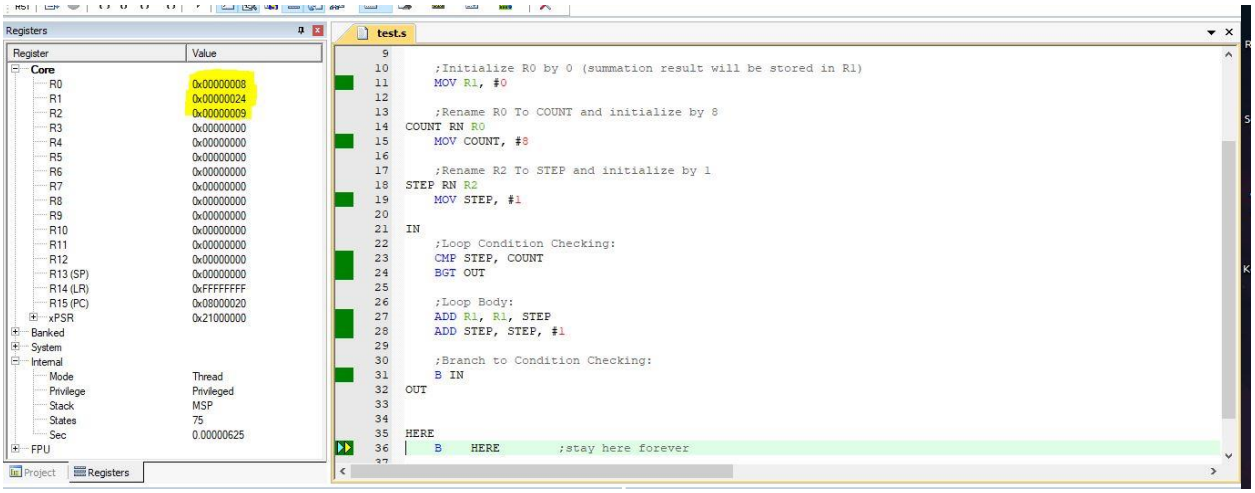
```
IN
    ;Loop Condition Checking:
    CMP STEP, COUNT
    BGT OUT

    ;Loop Body:
    ADD R1, R1, STEP
    ADD STEP, STEP, #1

    ;Branch to Condition Checking:
    B IN
OUT
```

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:

```
;make area for stack and allocate memory :
  AREA StackArea, DATA, READWRITE
Stack_Mem SPACE 0x18

```

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.

```
;initialize SP as a descending stack pointer:
    LDR R0, =Stack_Mem
    ADD R0, R0, 0x18
    MOV SP, R0
```

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.

```
;Renaming registers for simplicity :
INPUT RN R0
    MOV INPUT, #0xFFFFFFF0

ONES RN R1
    MOV ONES, #0

ZEROS RN R2
    MOV ZEROS, #0

STEP RN R3
    MOV STEP, #1

RESULT RN R4
    MOV RESULT, #0

```

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.

```
;Counting count of ones and zeros:
IN_COUNTER

    ;Loop Condition Checking:
    CMP STEP, #32                    ; Compare STEP and 32
    BGT OUT_COUNTER                  ; if STEP > 32 then go to OUT_COUNTER


    ANDS RESULT, INPUT, #0x00000001 ; RESULT = INPUT & 1

    ADDEQ ZEROS, ZEROS, #1           ; if result==0 then zeros++
    ADDNE ONES, ONES, #1             ; if result!=0 then ones++

    LSR INPUT, INPUT, #1             ; input = input >> 1

    ADD STEP, STEP, #1               ; step++
    B IN_COUNTER                     ; go to IN_COUNTER

OUT_COUNTER
```

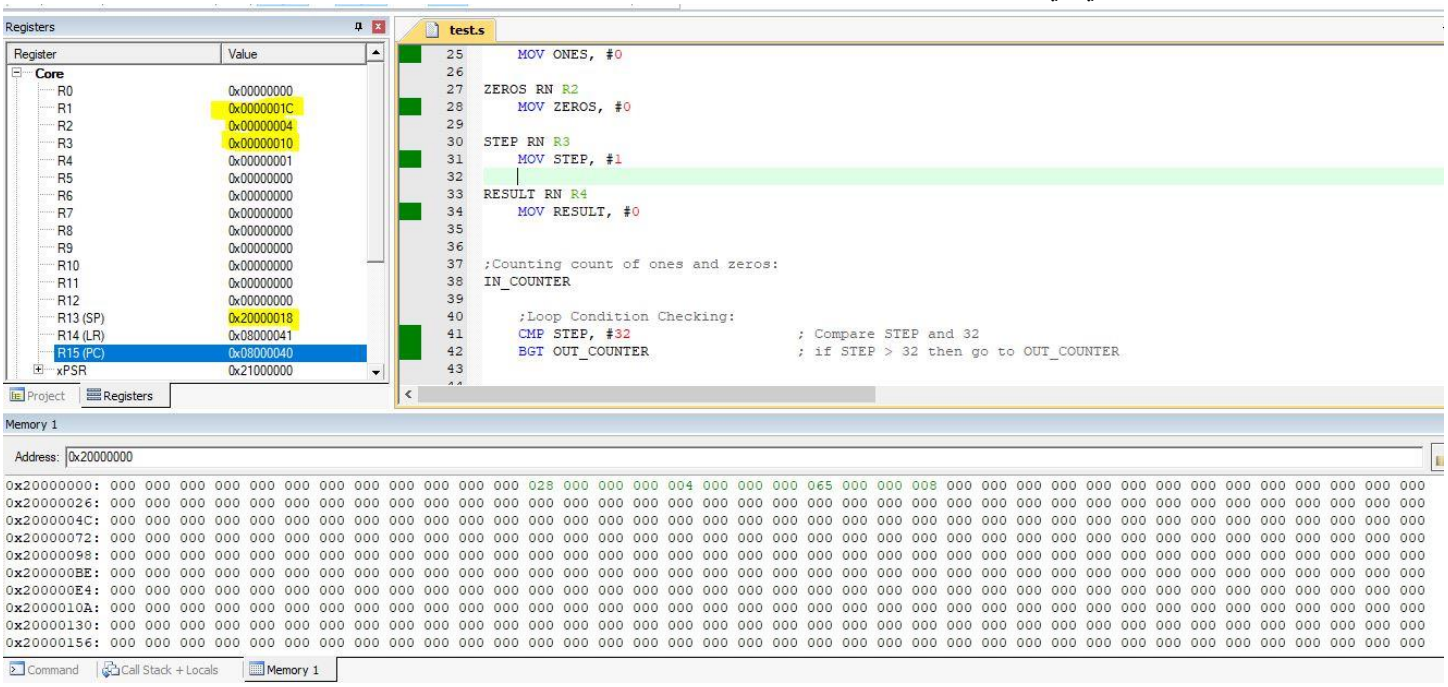
بدین ترتیب تعداد صفرها در ZEROS و تعداد یک ها در ONES قرار می‌گیرد. سپس از طریق BL به لیبل FUNC که پیاده‌سازی زیربرنامه ذکر شده در صورت سوال قرار دارد برنچ می‌کنیم. در این زیربرنامه در ابتدا ONES و ZEROS و نیز LR را که آدرس بازگشت در آن است را در استک پوش کرده، عملیات گفته شده در صورت سوال را که واضحا 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.

```
FUNC
; ONES  : R1
; ZEROS : R2

    PUSH {ONES, ZEROS, LR}

    MOV R3, #3
    MUL R2, R1, R3

    MOV R3, #100
    SUB R3, R3, R2

    POP {ONES, ZEROS, LR}
    MOV PC, LR
```

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: