نیمسال دوم **99-98**	**عنوان:** **برنامه‌سازی و I/O** در LC3	**استاد : دکتر مرادی**

در آخرین جلسه آزمایشگاه شما چند برنامه ی ساده با ماشین LC3 می نویسید و با ورودی و خروجی (I/O) آن آشنا می شوید.

1. انجام دهید!

1. از پوشه LC3 اسمبلر ماشین یا همان LC3Edit را باز کنید.
2. اکنون با فرض این که مقدار X در رجیستر R0 و مقدار Y در رجیستر R1 قرار دارد(مقدار X و مقدار Y را به دلخواه بدهید. نحوه انجام این کار در قسمت راهنمایی سوال توضیح داده شده است)، برنامه ای بنویسید که به ترتیب:
   - حاصل جمع X + Y را در رجیستر R2 ذخیره کند.
   - حاصل X AND Y را در رجیستر R3 ذخیره کند.
   - حاصل X OR Y را در رجیستر R4 ذخیره کند. (به دنبال دستور OR نگردید. چنین دستوری مستقیماً برای ماشین LC3 وجود ندارد!)
   - اگر x زوج بود آن را در رجیستر R7 و اگر فرد بود در رجیستر R6 ذخیره کند.

قسمت 1: کد مربوطه را در قسمت زیر قرار دهید.

```
.ORIG x3000
LD                  R0, X
LD                  R1, Y

ADD                 R2, R1 , R0

AND                 R3 , R1 , R0

NOT                 R0 , R0
NOT                 R1 , R1
AND                 R4 , R1 , R0
NOT                 R4 , R4
NOT                 R0 , R0
NOT                 R1 , R1

AND                 R5 , R0 , #1
BRz                 EVEN
LD                  R6 , X
EVEN LD                     R7 , X

HALT

X                   .FILL #4
Y                   .FILL #2

.END
```

دقت کنید که تمامی عملیات فوق باید در یک برنامه انجام شود و نه چهار برنامه ی مجزا، یعنی پس از این که حاصل جمع X + Y در R2 ذخیره شد، دیگر نباید تا انتهای برنامه مقدار R2 تغییر کند.

راهنمایی:

- در بخش سوم سوال بدیهی است که باید عملیات OR را با ترکیبی از عملیات AND و NOT پیاده سازی کنید. همچنین در بخش چهارم نیاز دارید که از دستور پرش شرطی یا BR استفاده کنید. برای بررسی این مسأله که آیا متغیر X زوج است یا نه کافی است سایر بیت های آن را به جز بیت کم ارزش صفر کنید. (به کمک دستور

AND)سپس مقدار موجود در X اگرصفر شده باشد، دستور BRz به خانه ای که در مقابل این دستور ذکر شده باشد پرش خواهد کرد.

دقت کنید!

احتمالاً در ابتدای راه اندازی ماشین LC3 تمام رجیسترها صفر هستند. پس برای این که نتایج برنامه ی فوق را به درستی مشاهده کنید، پیش از اجرای برنامه مقدار رجیسترها را به کمک کلیک راست و انتخاب Set Value تغییر دهید.

2. انجام دهید!

1. یک فایل جدید ایجاد کنید.
2. اکنون با فرض این که مقدارX در رجیستر R1 و مقدار Y در رجیستر R2 قرار دارد، برنامه ای بنویسید که ابتدا قدر مطلق X را محاسبه کرده و در R1 قرار دهد و سپس قدر مطلق Y را حساب کرده و در R2 قرار دهد و در انتها عددی که قدر مطلق آن بزرگتر است را در خانه ی R2 قرار دهد.

راهنمایی: برای فهمیدن این مطلب که مقدار موجود در یک رجیستر مثبت است یا منفی، کافی است تا یک عملیات ریاضی که حاصل آن رجیستر را تغییر نمی دهد انجام دهید و نتیجه را در همان رجیستر ذخیره کنید. با انجام این عملیات flag

```
.ORIG x3000
LD      R1, X
BRzp POSX
NOT     R1 , R1
ADD     R1 , R1 , #1
POSX
LD      R2, Y
BRzp POSY
NOT     R2 , R2
ADD     R2 , R2 , #1
POSY
NOT     R3, R2
ADD     R3, R3, #1
ADD     R3, R1, R3
BRzp BIG
BIG ADD  R2, R1, #0
HALT
X       .FILL #-4
Y       .FILL #2
.END
```

آشنایی با I/O:

برای این که برنامه ی ما بتواند به خوبی با کاربر ارتباط برقرار کند، باید بتوانیم ورودی را به آسانی از کاربر بگیریم و خروجی را در فرمتی مناسب به او نشان دهیم. برای این منظور در LC3 از LC3 Console استفاده می کنیم. در این قسمت می توان به برنامه ورودی داد یا چیزی روی آن برای کاربر چاپ کرد.

به جدول زیر توجه کنید:

دستور	توضیح
IN	با رسیدن برنامه به این خط اجرای برنامه متوقف می شود و برنامه منتظر دادن ورودی از طریق LC3 Console توسط کاربر میشود. دستور IN یک کاراکتر اسکی را از کاربر می گیرد و آن را در 8 بیت کم ارزش R0 ذخیره می کند. **دقت کنید که** پس از اجرای دستور IN محتوای سابق R0 از بین می رود, پس اگر در آن دیتای خاصی دارید، آن را در جای دیگری ذخیره کنید.
OUT	با اجرای این دستور محتوای R0 (هشت بیت کم ارزش آن) در خروجی چاپ می شود.
PUTS	این دستور که مخفف PUT String است, از آدرسی که R0 به آن اشاره می کند شروع به چاپ خانه های حافظه (هشت بیت کم ارزش آن ها) می کند تا زمانی که به خانه ای با محتوای x0000 (NULL) برسد. به عبارت دیگر یک رشته از کاراکتر ها که از خانه ای با آدرسی که در R0 ذخیره شده است را چاپ می کند.
.STRINGZ	این دستور یکی دیگر از دستوراتی است که با اسمبلر صحبت می کند. (مثل .ORIG, .END و...). این دستور همانند چند تا دستور .FILL پشت سرهم می ماند. با این دستور می توانید یک رشته از کاراکتر ها را در خانه های پشت سر هم در حافظه قرار دهید. (این دستور به صورت پیش فرض در انتهای رشته ی شما NULL را اضافه می کند.)

✓ برای درک بهتر دستورات فوق به مثال زیر توجه کنید:

```
.ORIG   x3050
LEA R0, HELLO ; put address of this label in R0
```

```
PUTS ; print the string starting from R0 until x0000 appears
AND R0,R0,0;
IN ; read a character from console (and store it in R0)
ADD R0,R0,1;
OUT   ; print R0[7:0] on console
HALT
HELLO .STRINGZ "HELLO WORLD" ; place this string in memory starting with
;address of x3057 (why?)
.END
```

3. انجام دهید! (خواندن از ورودی)

یک برنامه ی ساده بنویسید که عددی را از ورودی خوانده و در یک رجیستر بریزد.

قسمت 3 : کد مربوطه را در قسمت زیر قرار دهید.

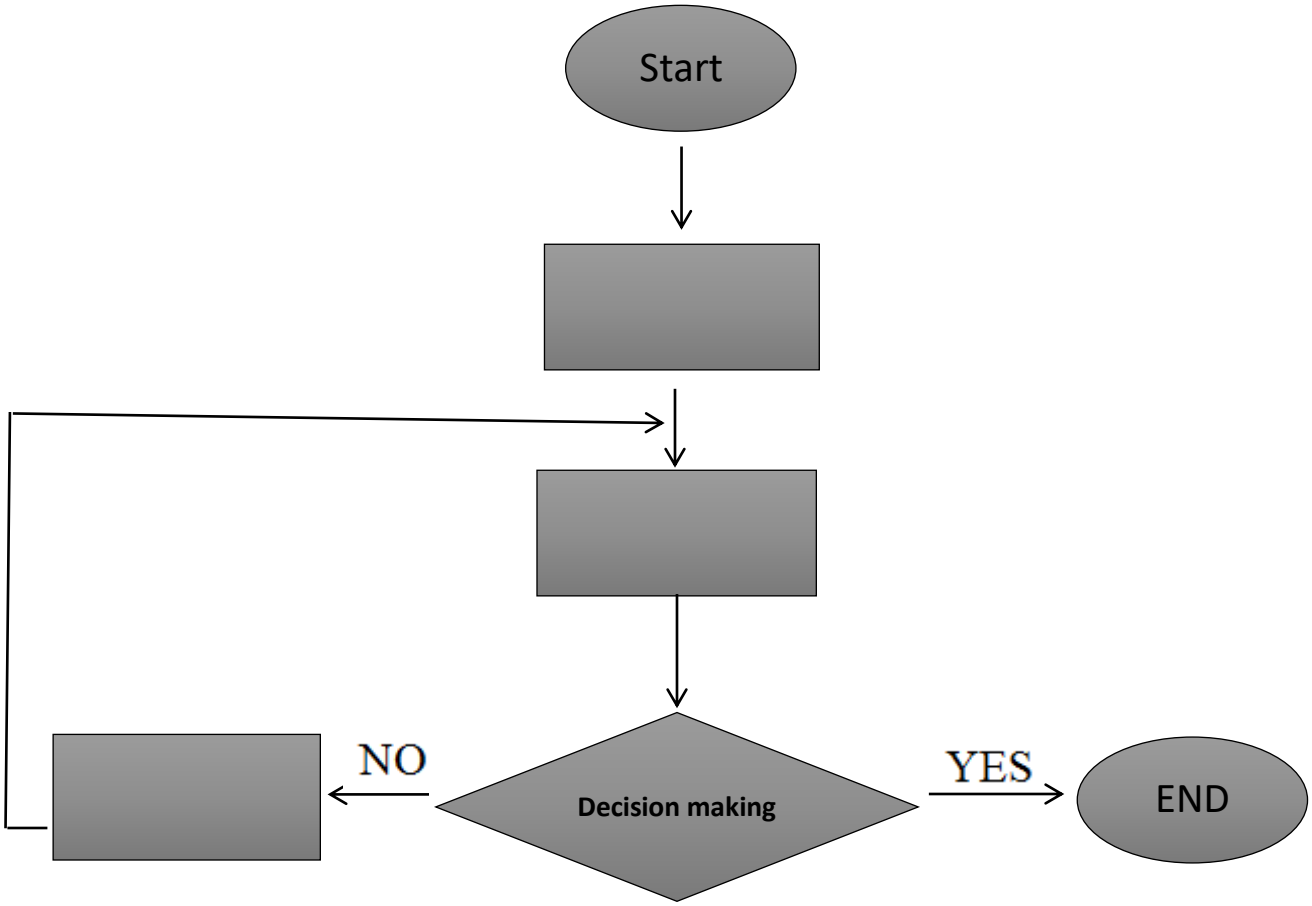
```
.ORIG x3000
LD R2, CONVERT
IN
ADD      R2,R0,R2

HALT

CONVERT .fill #-48

.END
```

در نهایت عدد یک رقمی در R2 ذخیره خواهد شد.

4. انجام دهید! (محاسبه ی حاصل ضرب مقادیر دو رجیستر)

در این قسمت می خواهیم برنامه ای بنویسیم که محتوای دو رجیستر R3 و R4 را در هم ضرب کرده و در رجیستر R5 ذخیره کند. احتمالا تا حالا متوجه شده اید که شبیه ساز LC3 دستور ضرب ندارد. برای پیاده سازی دستور ضرب از مفهوم

جمع استفاده میکنیم. به سادگی برای این کار عدد اول را به اندازه عدد دوم با خودش جمع می کنیم. برای انجام این قسمت می‌توانید ابتدا فلوچارت[1] زیر را کامل کنید(تحویل این فلوچارت الزامی نیست):



فلوچارت حاصل ضرب دو عدد صحیح نامنفی

حال به کمک فلوچارتی که تکمیل کردید، کد اسمبلی این قسمت را بزنید.

[1] Flow Chart

قسمت 4 : کد خود را در کادر زیر بنویسید.

```
.ORIG x3000
LD      R3, X
LD      R4, Y
AND     R5, R5, #0

MULTIPLY ADD  R5, R5, R3
 ADD    R4, R4, #-1
BRnp MULTIPLY

HALT

X .fill #8
Y .fill #5
.END
```

اگر R5 در ابتدا صفر بود نیازی به دستور خط چهارم نخواهد بود ولی برای اطمینان مقدار R5 در ابتدا صفر شده است.

تبدیل یک قطعه کد به تابع:

برای تبدیل یک قطعه کد به تابع کارهای زیر را انجام دهید:

- یک برچسب (Label) به عنوان نام تابع در نظر بگیرید.
- قطعه کد مورد نظر را زیر این برچسب قرار دهید.
- در پایان قطعه کد، آدرس بازگشت (Return Address) را که در رجیستر R7 ذخیره شده است، در رجیستر PC قرار دهید. (این کار را با اجرای دستور RET به عنوان آخرین دستور تابع خود می توانید انجام دهید.)

توجه: آدرس بازگشت از تابع که گفته شده در R7 قرار دارد را خودتان باید به هنگام فراخوانی تابع در R7 قرار دهید. در قسمت بعدی (صدا زدن یک تابع) نحوه ی انجام این کار توضیح داده شده است.

راهنمایی: به هنگام می‌توانید از نمونه کد زیر برای نوشتن یک تابع کمک بگیرید.

```
1          LDI R0, N     ; Argument N is now in R0
2          JSR F         ; Jump to subroutine F.
3          STI R1, FN
4          HALT
5  N       .FILL 3120    ; Address where n is located
6  FN      .FILL 3121    ; Address where fn will be stored.
7                        ; Subroutine F begins
8  F       AND R1, R1, x0 ; Clear R1
9          ADD R1, R0, x0 ; R1 ← R0
10         ADD R1, R1, R1 ; R1 ← R1 + R1
11         ADD R1, R1, x3 ; R1 ← R1 + 3.  Result is in R1
12         RET            ; Return from subroutine
13         END
```

Listing 5.1: A subroutine for the function $f(n) = 2n + 3$.

صدا زدن (فراخوانی) یک تابع[2]:

برای اجرای یک تابع لازم است که آدرس سر تابع را در رجیستر PC قرار دهیم. در این صورت ادامه ی اجرای برنامه از سر تابع خواهد بود. بعد از آن که کار تابع تمام شد، مقدار رجیستر PC باید به مقدار قبل از صدا زدن تابع به اضافه 1 (چرا؟) تغییر کند. مسئله ای که مطرح می شود این است که آدرس بازگشت را چگونه به دست بیاوریم، چون یک تابع ممکن است چندین بار و از آدرس های مختلف صدا زده شود. بنابر این باید قبل از صدا زدن تابع، آدرس فعلی را در جایی که تابع هم از آن با خبر است ذخیره کنیم. برای این کار می توانیم کارهای زیر را انجام دهیم:

JSR MyFunc آدرس دستور بعد از JSR (یا در واقع PC یک واحد اضافه شده) را در R7 ذخیره کن و به MyFunc پرش کن.

توجه کنید که MyFunc یک label است.

حال همانطور که در قسمت قبل توضیح داده شد در پایان اجرای تابع به عنوان آخرین دستور آن باید دستور زیر را بنویسید:

RET

این دستور محتوای رجیستر R7 (که به هنگام فراخوانی با دستور JSR تعیین شده بود) را در PC قرار می دهد.

پس از اجرای این دستور، به دستور بعد از فراخوانی تابع پرش می کنیم و اجرای برنامه ادامه می یابد.

توجه: دستور RET مشابه دستور JMP بوده، با این تفاوت که همواره به آدرسی که در R7 وجود دارد پرش می کند.

5. انجام دهید! (تبدیل برنامه های قبلی به دو تابع مجزا)

[2] Function Call

حال برنامه های قبلی را تبدیل به دو تابع مجزا کنید. یعنی تابع هایی بسازید که یکی عملیات ضرب و دیگری عملیات خواندن از ورودی را انجام دهد.

دقت کنید: مطمئن شوید که برنامه ی شما درست کار می کند.

نکته: در زمان صدا زده شدن تابع، محتوای رجیستر R7 تغییری می کند؟ (چرا؟)

قسمت 5 : کد و پاسخ خود را در کادر زیر بنویسید

موفق باشید

کد قسمت 5 در صفحه بعد

وقتی که تابع را با دستور JSR صدا می زنیم این دستور، آدرس دستور بعد JSR را در R7 قرار می دهد تا وقتی که با دستور RET از تابع برمی گردیم بدانیم از کجا باید به اجرا ادامه دهیم برای همین در هنگام صدا زده شدن تابع محتوای R7 تغییر می کند و برابر آدرس دستور بعد JSR می شود تا در بازگشت از تابع مشکلی پیش نیاید.

فقط باید به این نکته توجه شود که در تابعی که ورودی می گیریم چون بعد از دستور IN محتوای R7 تغییر کرده و برابر آدرس بعد از دستور IN می شود (تا در بازگشت از IN بدانیم از کجای INPUT باید ادامه دهیم) باید آدرس R7 قبل از دستور IN را در جایی مثل ADR7 ذخیره کرد تا هنگام برگشت از تابع INPUT به درستی مقدار R7 را آپدیت کرده و آن را خوانده و در نهایت با دستور RET برگشته و به ادامه برنامه بپردازیم و دیگر به آدرس بعد از IN برنگردیم و به جای درست برویم.

```
.ORIG x3000
AND     R7, R7, #0
LD      R3, X
LD      R4, Y

JSR MULTIPLY

JSR INPUT



HALT

X .fill #8
Y .fill #5
CONVERT .fill #-48
ADR7 .fill #0

MULTIPLY
AND     R5, R5, #0
MULT
ADD     R5, R5, R3
ADD     R4, R4, #-1
BRnp MULT
RET

INPUT
ST      R7, ADR7
LD      R2, CONVERT
IN
ADD     R2,R0,R2
LD      R7, ADR7
RET

.END
```