Lab04 2017/12/2

PB16030899 Zhu Heqin

PURPOSE

This lab is more difficult than the three formmer ones. It's a bit hard . Through this lab, we should get grape of the concept of subroutines and write more powerful programms that contains algorithm such as the `two-div find` , which is the represent of the thinking of divide and conquer.
By this way, we can get the root of a polynomial .

PRINCEPLE

Iinstructions I used are as follows

- LD
- LDI
- LDR
- ST
- STR
- ADD
- NOT
- AND
- BR
- JSR

- RET

and **`persudo-op`**

- .FILL
- .ORIG
- .END

I wrote two subroutines :

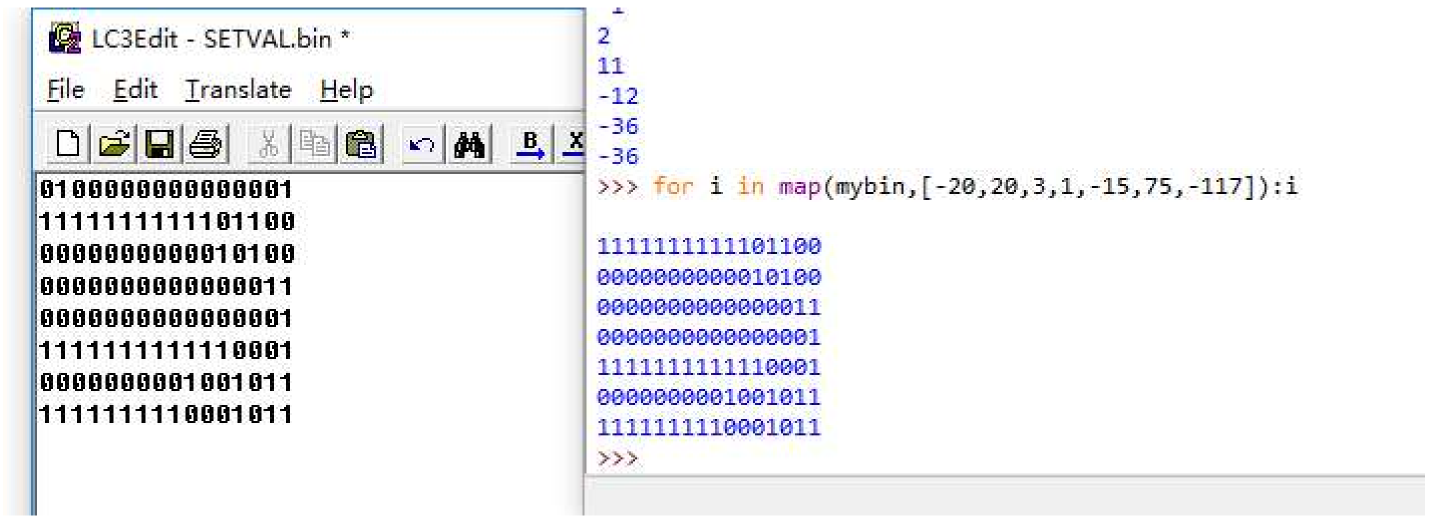
- `twodiv` :it takes two ends of an interval (stored in R1,R2) , and return the mid of them ,stored in R0
- `func` : it cal the fun with val stored in R0 , and stored the result in R4.
  in the function , I also write mul label to multiply two num

PROCEDURE

I get some bugs when dubugging:

1. in the function , the first step is to store the registers that will be used in the subrourine.
2. then clear the val of the registers
3. make sure the two ends of the interval changes correctly, that is , the left increase, and the right decrease.
4. in the subroutine, if you want to jump to another subroutine, save the val of R7, which contains the origin return-addr , otherwise it will be covered and the subroutine can't return the main routine.

When dubugging , to quickly set the test case, I write the val in binary and load it in the simulator. What's more ? to get a num's two's implement code , I use python scipt to generate it。 The script is attchted at the end of the file.
It looks like this



```
0100000000000001
1111111111101100
0000000000010100
0000000000000011
0000000000000001
1111111111110001
0000000001001011
1111111110001011
```

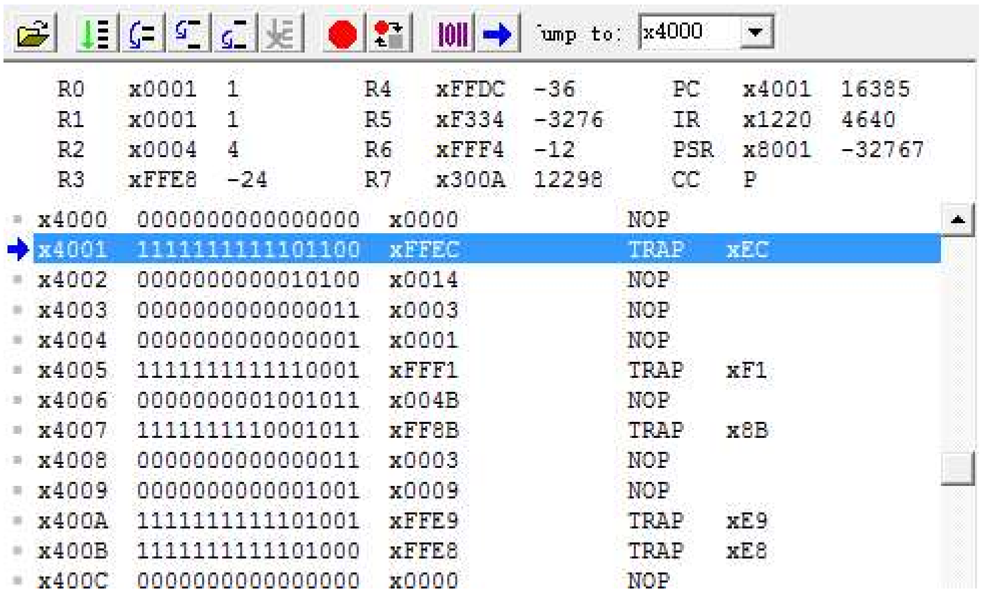
```
2
11
-12
-36
-36
>>> for i in map(mybin,[-20,20,3,1,-15,75,-117]):i

1111111111101100
0000000000010100
0000000000000011
0000000000000001
1111111111110001
0000000001001011
1111111110001011
>>>
```

RESULT

test case 1: $f(x) = x^3 - 15x^2 + 75x - 117$

Before executing the program:

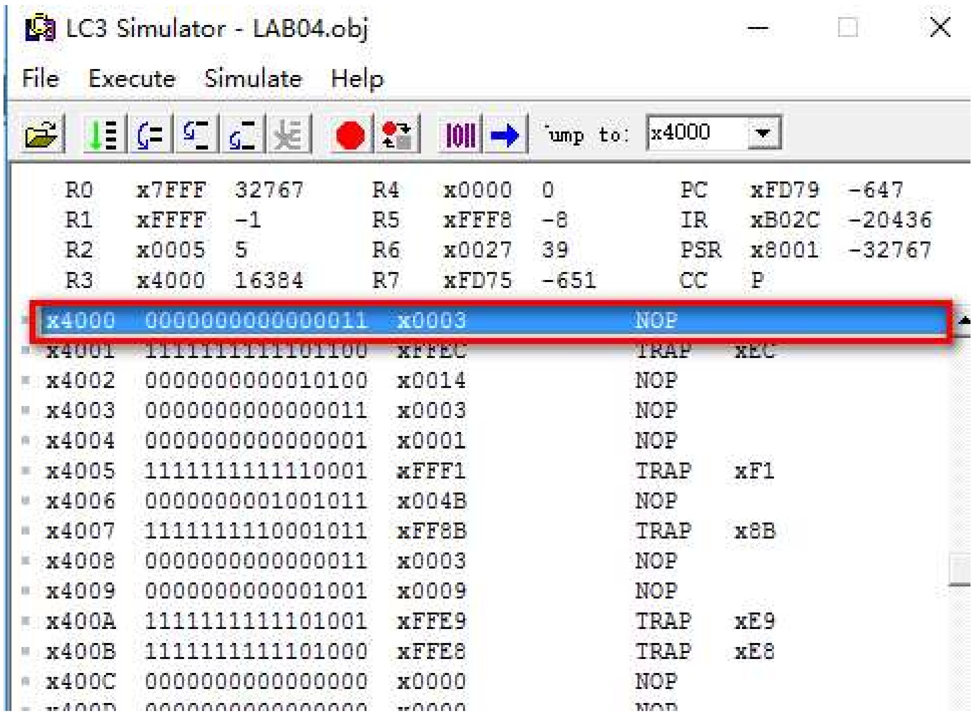


```
'ump to: x4000

R0   x0001  1          R4   xFFDC  -36        PC   x4001  16385
R1   x0001  1          R5   xF334  -3276      IR   x1220  4640
R2   x0004  4          R6   xFFF4  -12        PSR  x8001  -32767
R3   xFFE8  -24        R7   x300A  12298      CC   P

x4000  0000000000000000  x0000    NOP
x4001  1111111111101100  xFFEC    TRAP   xEC
x4002  0000000000010100  x0014    NOP
x4003  0000000000000011  x0003    NOP
x4004  0000000000000001  x0001    NOP
x4005  1111111111110001  xFFF1    TRAP   xF1
x4006  0000000001001011  x004B    NOP
x4007  1111111110001011  xFF8B    TRAP   x8B
x4008  0000000000000011  x0003    NOP
x4009  0000000000001001  x0009    NOP
x400A  1111111111101001  xFFE9    TRAP   xE9
x400B  1111111111101000  xFFE8    TRAP   xE8
x400C  0000000000000000  x0000    NOP
```

After executing the program:



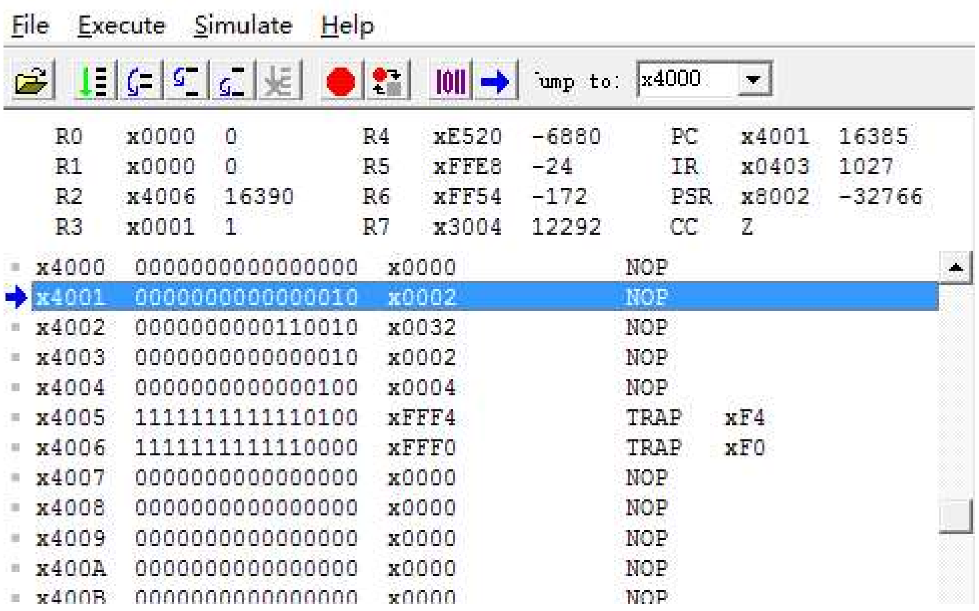
```
LC3 Simulator - LAB04.obj
File  Execute  Simulate  Help
'ump to: x4000

R0   x7FFF  32767      R4   x0000  0          PC   xFD79  -647
R1   xFFFF  -1         R5   xFFF8  -8         IR   xB02C  -20436
R2   x0005  5          R6   x0027  39         PSR  x8001  -32767
R3   x4000  16384      R7   xFD75  -651       CC   P

x4000  0000000000000011  x0003    NOP
x4001  1111111111101100  xFFEC    TRAP   xEC
x4002  0000000000010100  x0014    NOP
x4003  0000000000000011  x0003    NOP
x4004  0000000000000001  x0001    NOP
x4005  1111111111110001  xFFF1    TRAP   xF1
x4006  0000000001001011  x004B    NOP
x4007  1111111110001011  xFF8B    TRAP   x8B
x4008  0000000000000011  x0003    NOP
x4009  0000000000001001  x0009    NOP
x400A  1111111111101001  xFFE9    TRAP   xE9
x400B  1111111111101000  xFFE8    TRAP   xE8
x400C  0000000000000000  x0000    NOP
```

test case 2: $f(x) = 4x^2 - 12x - 16$

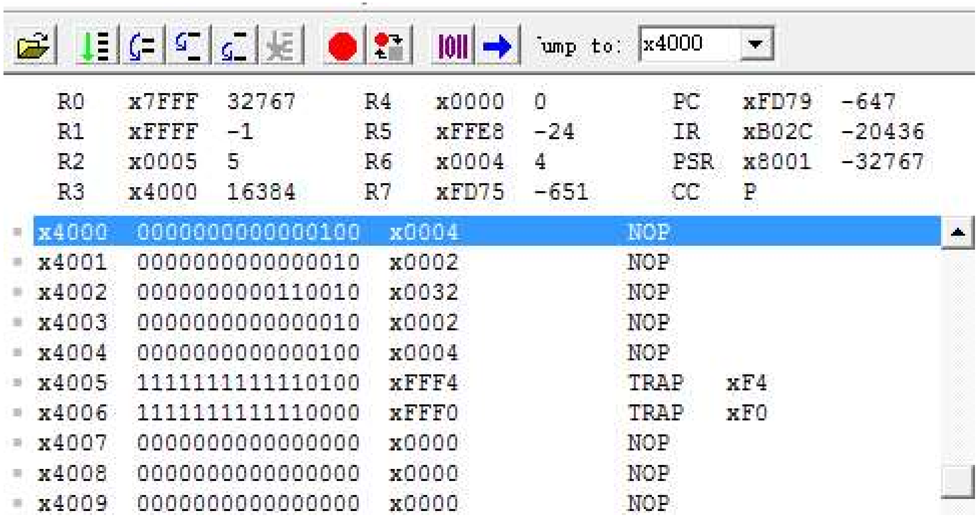
Before executing the program:



```
R0   x0000   0        R4   xE520   -6880     PC    x4001   16385
R1   x0000   0        R5   xFFE8   -24       IR    x0403   1027
R2   x4006   16390    R6   xFF54   -172      PSR   x8002   -32766
R3   x0001   1        R7   x3004   12292     CC    Z

x4000  0000000000000000  x0000         NOP
x4001  0000000000000010  x0002         NOP
x4002  0000000000110010  x0032         NOP
x4003  0000000000000010  x0002         NOP
x4004  0000000000000100  x0004         NOP
x4005  1111111111110100  xFFF4         TRAP   xF4
x4006  1111111111110000  xFFF0         TRAP   xF0
x4007  0000000000000000  x0000         NOP
x4008  0000000000000000  x0000         NOP
x4009  0000000000000000  x0000         NOP
x400A  0000000000000000  x0000         NOP
x400B  0000000000000000  x0000         NOP
```

After executing the program:



```
R0   x7FFF   32767    R4   x0000   0         PC    xFD79   -647
R1   xFFFF   -1       R5   xFFE8   -24       IR    xB02C   -20436
R2   x0005   5        R6   x0004   4         PSR   x8001   -32767
R3   x4000   16384    R7   xFD75   -651      CC    P

x4000  0000000000000100  x0004         NOP
x4001  0000000000000010  x0002         NOP
x4002  0000000000110010  x0032         NOP
x4003  0000000000000010  x0002         NOP
x4004  0000000000000100  x0004         NOP
x4005  1111111111110100  xFFF4         TRAP   xF4
x4006  1111111111110000  xFFF0         TRAP   xF0
x4007  0000000000000000  x0000         NOP
x4008  0000000000000000  x0000         NOP
x4009  0000000000000000  x0000         NOP
```

It works well. Through this lab , I learned a lot. Assemble language is so brief and useful.

ATTACHMENT

This is the small `python` script to get **two's implement code** of a num. So I can load this obj to the simulator to set the test case value.

```
def mybin(n,length=16):
    rst=0
    if n<0:
        n=-n
        if not n&(n-1):  # judge if n is 2^k
            ct=0
            while n!=1:
                ct+=1
                n=n>>1
            rst= '1'*(length-ct)+'0'*ct
        else:
            s=bin(n-1)[2:]
            sn=''.join(['1' if i=='0' else '0' for i in s ])
            rst='1'*(length-len(s))+sn
    else:
        s=bin(n)[2:]
        rst = '0'*(length-len(s))+s
    print(rst)
```