LAB 4

1. INTRODUCTION

This lab requires to write a LC-3 assembly program to help Professor Patt, that is, to read a map and tells the longest distance. The map is a *N* X *M* matrix, and Patt can only ski to the adjacent vertex only when the height of the adjacent vertex is lower.

To achieve the goal, the program should include a recursive structure. When Patt reaches a vertex, the program should grope the adjacent four vertices to see if there is a path and save the length. The program should repeat this process until every vertex has been set as the beginning once.

This lab is meant to let us fully understand recursion and pay attention to the stack.

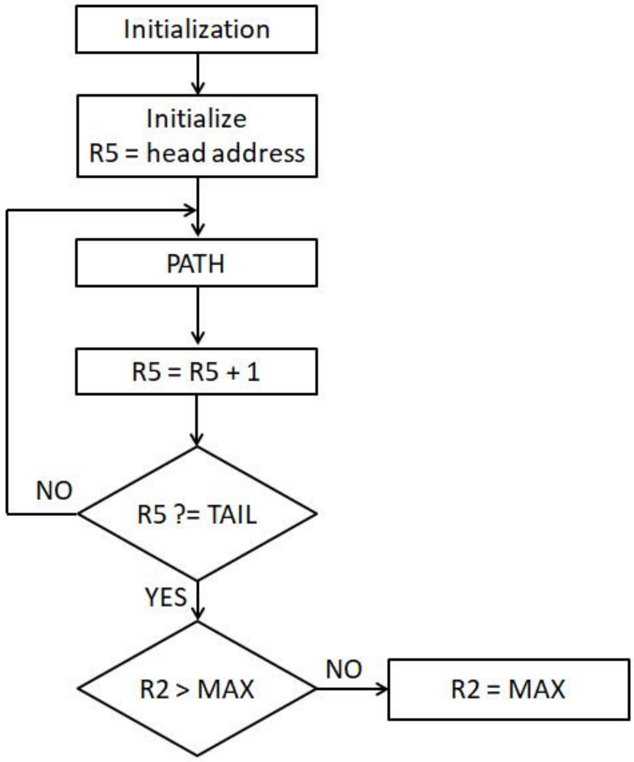
2. ALGORITHM

To finish the tasks, the algorithm should be like:

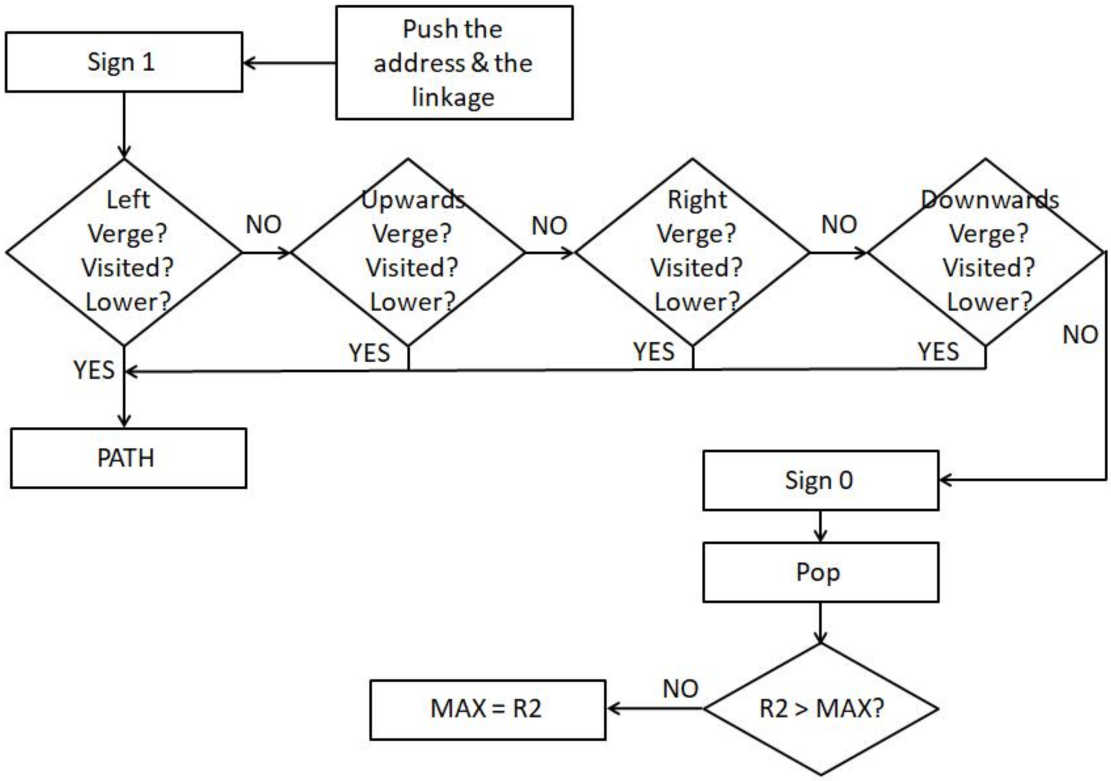
1. Do the initialization;
2. Execute the main codes and start the recursion;
3. Test the longest path started from a certain vertex and store the length;
4. Change the starting point and loop until every vertex has been tested;
5. Store the max length in R2;

In the program, R0 stores the value of the current vertex. R1 stores the value of the adjacent vertex. R2 stores the length and will be loaded with the max length at last. R3 and R4 are used for computation. R5 stores the address of the current vertex. R6 acts as a stack pointer. The run-time stack saves the address of the visited vertices and the linkages.

The diagram of the main codes is shown as follow:



The diagram of the PATH subroutine is shown as follow:

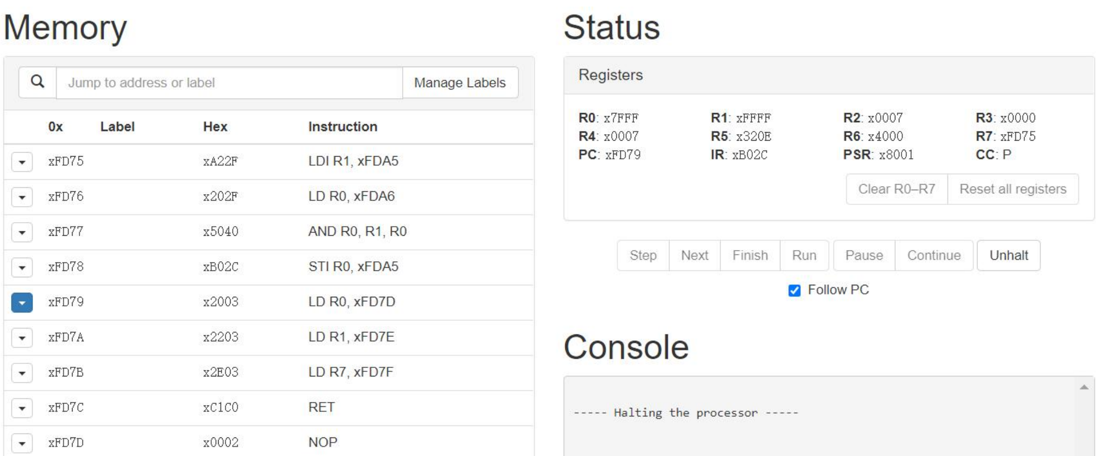


3. Testing Result

The tables represent the samples and the screenshots are the results.

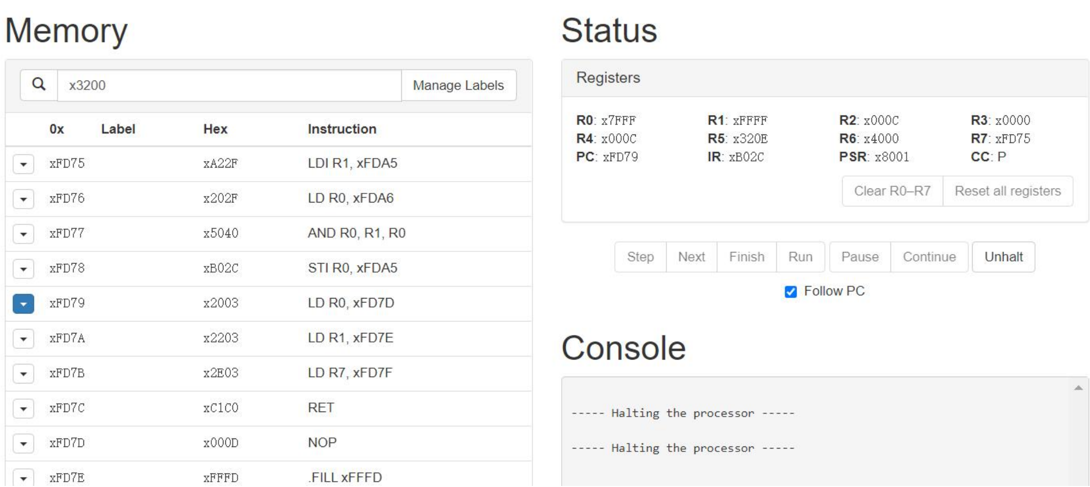
Test 1: x0007

1	7	6	5
2	9	8	3
3	4	5	2



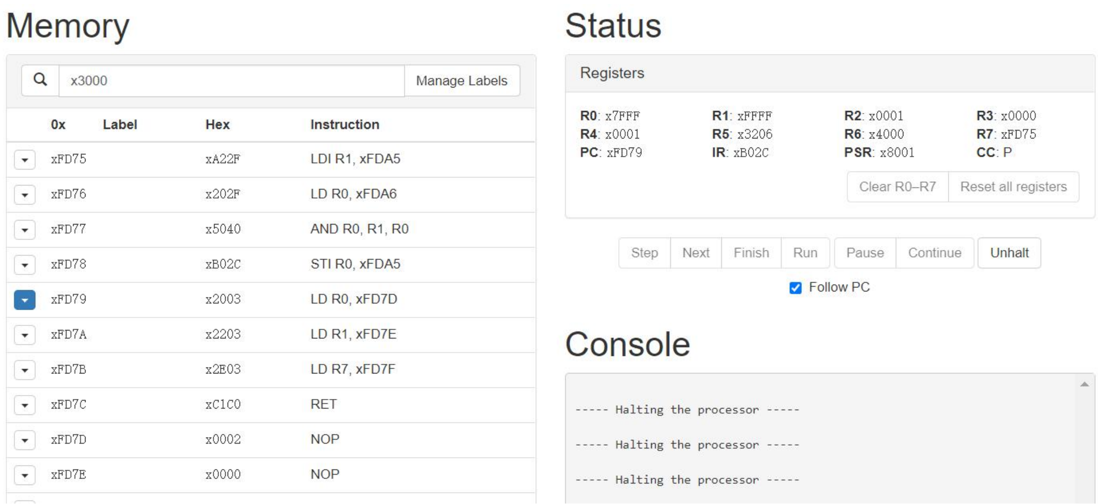
Test 2: x000C

11	12	13	14
18	17	16	15
19	20	21	22



Test 3: x0001

```
2    2
2    2
```



4. Discussion and Experience

When writing the program, I found the recursive structure is hard to comprehend. So at first I drew a diagram to help me to clarify my program. Then when debugging, I chose some special samples and set breakpoints at the beginning of each subroutine. I learn that appropriate breakpoints do help a lot and have a better comprehension on recursion.

In fact, I realized that to check if the adjacent vertex has been visited is not necessary, because a visited adjacent vertex will never pass the test that tests the height. Besides, I can use more registers next time, which can save the time that the program runs.

Appendix: Source Code

```
        .ORIG   x3000
        LEA     R6, VISIT
        AND     R3, R3, #0
        LD      R5, LENG
INIVI   STR     R3, R6, #0      ; clear VISIT
        ADD     R6, R6, #1
        ADD     R5, R5, #-1
        BRp     INIVI           ; none vertices has been visited
```

```
        AND     R2, R2, #0
        ST      R3, MAX
        LDI     R4, ROW
        BRz     STOP
        LDI     R5, COLUMN
        BRz     STOP
LOOP    ADD     R3, R3, R4
        ADD     R5, R5, #-1
        BRp     LOOP
        LD      R6, StaR6
        LD      R5, HEAD
        ADD     R3, R3, R5
        NOT     R3, R3
        ADD     R3, R3, #1
        ST      R3, TAIL        ; minus tail address
Main    ADD     R3, R5, #0
        AND     R2, R2, #0
        JSR     PATH
        ADD     R5, R5, #1
        LD      R3, TAIL
        ADD     R3, R3, R5
        BRn     Main
        LD      R2, MAX         ; load the result into R2
STOP    TRAP    x25
TAIL    .BLKW   #1
HEAD    .FILL   x3202
LENG    .FILL   x0034
StaR6   .FILL   x4000          ; activation record

PATH    STR     R3, R6, #0
        ADD     R6, R6, #1
        STR     R7, R6, #0
        ADD     R6, R6, #1      ; push
        ADD     R2, R2, #1      ; the current vertex
        JSR     SIGN1
CLEFT   JSR     VLEFT           ; check the verge
        BRz     CUP            ; verge, check upwards
        LDR     R3, R6, #-2
        ADD     R3, R3, #-1
        JSR     CVISIT          ; check if visited
        BRp     CUP            ; visited already
        LDR     R0, R6, #-2
        ADD     R3, R0, #-1
        JSR     COMPU
```

```
        BRnz    CUP
        JSR     PATH
CUP     JSR     VUPPER          ; check the verge
        BRn     CRIGHT
        LDR     R3, R6, #-2
        LDI     R4, COLUMN
        NOT     R4, R4
        ADD     R4, R4, #1
        ADD     R3, R3, R4
        JSR     CVISIT          ; check if visited
        BRp     CRIGHT
        LDR     R0, R6, #-2
        LDI     R4, COLUMN
        NOT     R4, R4
        ADD     R4, R4, #1
        ADD     R3, R0, R4
        JSR     COMPU
        BRnz    CRIGHT
        JSR     PATH
CRIGHT  JSR     VRIGHT          ; check the verge
        BRz     CDOWN
        LDR     R3, R6, #-2
        ADD     R3, R3, #1
        JSR     CVISIT          ; check if visited
        BRp     CDOWN
        LDR     R0, R6, #-2
        ADD     R3, R0, #1
        JSR     COMPU
        BRnz    CDOWN
        JSR     PATH
CDOWN   JSR     VLOWER          ; check the verge
        BRzp    ENDNO
        LDR     R3, R6, #-2
        LDI     R7, COLUMN
        ADD     R3, R3, R7
        JSR     CVISIT          ; check if visited
        BRp     ENDNO
        LDR     R0, R6, #-2
        LDI     R7, COLUMN
        ADD     R3, R0, R7
        JSR     COMPU
        BRnz    ENDNO
        JSR     PATH
ENDNO   JSR     SIGN0           ; cancel the signal
```

```
        ADD     R6, R6, #-1
        LDR     R7, R6, #0
        ADD     R6, R6, #-1
        NOT     R3, R2          ; pop
        ADD     R3, R3, #1
        LD      R4, MAX
        ADD     R3, R3, R4      ; compare R2 with the MAX length
        BRzp    Return
        ST      R2, MAX
Return  ADD     R2, R2, #-1
        RET
MAX     .FILL   x0001

COMPU   LDR     R0, R0, #0      ; the current value
        LDR     R1, R3, #0      ; the adjacent value
        NOT     R1, R1
        ADD     R1, R1, #1
        ADD     R1, R1, R0      ; R1 <- R0-R1
        RET
SIGN1   LDR     R3, R6, #-2     ; visited
        LD      R4, MASK
        ADD     R3, R3, R4
        LEA     R4, VISIT
        ADD     R4, R4, R3
        AND     R3, R3, #0
        ADD     R3, R3, #1
        STR     R3, R4, #0      ; signal a vertex has been passed
        RET
SIGN0   LDR     R3, R6, #-2     ; not visited
        LD      R4, MASK
        ADD     R3, R3, R4
        LEA     R4, VISIT
        ADD     R4, R4, R3
        AND     R3, R3, #0
        STR     R3, R4, #0      ; signal a vertex has not been passed
        RET

VLEFT   LDR     R3, R6, #-2     ; the left verge
        LD      R4, MASK
        ADD     R3, R3, R4
        LDI     R4, COLUMN
        ADD     R3, R3, R4
        NOT     R4, R4
        ADD     R4, R4, #1
```

```
Lleft   ADD     R3, R3, R4
        BRp     Lleft
        RET
MASK    .FILL  xCDFE          ; -x3202
VUPPER  LDR     R3, R6, #-2     ; the upper verge
        LD      R4, MASK
        ADD     R3, R3, R4
        LDI     R4, COLUMN
        NOT     R4, R4
        ADD     R4, R4, #1
        ADD     R3, R3, R4
        RET
VRIGHT  LDR     R3, R6, #-2     ; the right verge
        LD      R4, MASK
        ADD     R3, R3, R4
        ADD     R3, R3, #1
        LDI     R4, COLUMN
        NOT     R4, R4
        ADD     R4, R4, #1
Lright  ADD     R3, R3, R4
        BRp     Lright
        RET
VLOWER  LDR     R3, R6, #-2     ; the lower verge
        LDI     R4, COLUMN
        ADD     R3, R3, R4
        LD      R4, TAIL
        ADD     R3, R3, R4
        RET
CVISIT  LD      R4, MASK        ; check if the adjacent is visited
        ADD     R3, R3, R4
        LEA     R4, VISIT
        ADD     R4, R4, R3
        LDR     R4, R4, #0
        RET
ROW     .FILL  x3200
COLUMN  .FILL  x3201
VISIT   .BLKW  #52
        .END
```