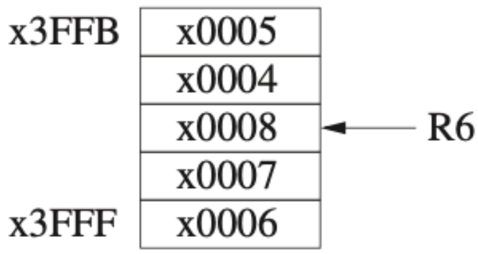
Homework 6

> 2023.07.19 沈韵沨 3200104392

Chap 7

★7.30 There are times when one wants to implement a stack in memory, but cannot provide enough memory to be sure there will always be plenty of space to push values on the stack. Furthermore, there are times (beyond EE 306) when it is OK to lose some of the oldest values pushed on the stack. We can save that discussion for the last class if you like.
In such situations, a reasonable technique is to specify a circular stack as shown below. In this case, the stack occupies five locations x3FFB to x3FFF. Initially, the stack is empty, with R6 = x4000. The figure shows

the result of successively pushing the values 1, 2, 3, 4, 5, 6, 7, 8 on the stack.



That is, the 1 was written into x3FFF, the 2 was written into x3FFE, etc. When the time came to push the 6, the stack was full, so R6 was set to x3FFF, and the 6 was written into x3FFF, clobbering the 1 which was originally pushed.
If we now pop five elements off the stack, we get 8, 7, 6, 5, and 4, AND we have an empty stack, even though R6 contains x3FFD. Why? Because 3, 2, and 1 have been lost. That is, even though we have pushed eight values, there can be at most only five values actually available on the stack for popping. We keep track of the number of actual values on the stack in R5.
Note that R5 and R6 are known to the calling routine, so a test for underflow can be made by the calling program using R5. Furthermore, the calling program puts the value to be pushed in R0 before calling PUSH.

Your job: Complete the assembly language code shown below to implement the PUSH routine of the circular stack by filling in each of the lines: (a), (b), (c), and (d) with a missing instruction.

```
PUSH     ST R1, SAVER
         LD R1, NEGFULL
         ADD R1, R6, R1
         --------------(a)

         LD R6, BASE
SKIP     ADD R6, R6, #-1
         LD R1, MINUS5
         ADD R1, R5, R1
         BRz END
         --------------(b)

END      --------------(c)
         --------------(d)

         RET
NEGFULL .FILL xC005      ; x-3FFB
MINUS5  .FILL xFFFB      ; #-5
BASE    .FILL x4000
SAVER   .BLKW #1
```

PUSH	ST	R1,	SAVER			SAVER = R1 (Callee - Save)
	LD	R1,	NEGFULL			R1 = x-3FFB (Min Addr)
	ADD	R1,	R6,	R1		R1 = sp - Min Addr
	BRnp	SKIP			(a)	if (R1 == Min Addr) reload
	LD	R6,	BASE			sp = BASE (MaxAddr +1)
SKIP	ADD	R6,	R6,	#-1		sp --
	LD	R1,	MINUS5			R1 = -5
	ADD	R1,	R5,	R1		R1 = size -5
	BRz	END				} if (size < Max Size) size ++
	ADD	R5,	R5,	#1	(b)	
END	STR	R0,	R6,	#0	(c)	Push (R0)
	LD	R1,	SAVER		(d)	Callee - Restore
	RET					

Chap 8

8.2 What is an advantage to using the model in Figure 8.9 to implement a stack vs. the model in Figure 8.8?

- On each Push / Pop action, **every value already in the stack MOVES** in the model of Figure 8.8, while they don't in the model of Figure 8.9.
- Since load & save data from memory takes a lot of time, it's wiser to move stack pointer(in Figure 8.9) than move all the other values already in the stack(in Figure 8.8).

8.8 The following operations are performed on a stack:

`PUSH A, PUSH B, POP, PUSH C, PUSH D, POP, PUSH E, POP, POP, PUSH F`

a. What does the stack contain after the `PUSH F`?

b. At which point does the stack contain the most elements? Without removing the elements left on the stack from the previous operations, we perform:

`PUSH G, PUSH H, PUSH I, PUSH J, POP, PUSH K, POP, POP, POP, PUSH L, POP, POP, PUSH M`

c. What does the stack contain now?

a. The stack contains [F, A] after `PUSH F`.

b. The stack contains the most elements after perform `PUSH J / PUSH K`:

- After `PUSH J`: [J, I, H, G, F A]
- After `PUSH K`: [K, I, H, G, F, A]

c. Now it contains [M, F, A] (from the top to the bottom).

★8.10 It is easier to identify borders between cities on a map if adjacent cities are colored with different colors. For example, in a map of Texas, one would not color Austin and Pflugerville with the same color, since doing so would obscure the border between the two cities.
Shown next is the recursive subroutine EXAMINE. EXAMINE examines the data structure representing a map to see if any pair of adjacent cities have the same color. Each node in the data structure contains the city's color and the addresses of the cities it borders. If no pair of adjacent cities have the same color, EXAMINE returns the value 0 in R1. If at least one pair of adjacent cities have the same color, EXAMINE returns the value 1 in R1. The main program supplies the address of a node representing one of the cities in R0 before executing JSR EXAMINE.

```
        .ORIG x4000
EXAMINE ADD R6, R6, #-1
        STR R0, R6, #0
        ADD R6, R6, #-1
        STR R2, R6, #0
        ADD R6, R6, #-1
        STR R3, R6, #0
        ADD R6, R6, #-1
        STR R7, R6, #0

        AND R1, R1, #0  ; Initialize output R1 to 0
        LDR R7, R0, #0
        BRn RESTORE     ; Skip this node if it has already been visited

        LD  R7, BREADCRUMB
        STR R7, R0, #0  ; Mark this node as visited
        LDR R2, R0, #1  ; R2 = color of current node
        ADD R3, R0, #2

AGAIN   LDR R0, R3, #0  ; R0 = neighbor node address
        BRz RESTOR
        LDR R7, R0, #1
        NOT R7, R7      ; <-- Breakpoint here
        ADD R7, R7, #1
        ADD R7, R2, R7  ; Compare current color to neighbor's color
        BRz BAD
        JSR EXAMINE     ; Recursively examine the coloring of next neighbor
        ADD R1, R1, #0
        BRp RESTORE     ; If neighbor returns R1=1, this node should return R1=1
        ADD R3, R3, #1
        BR  AGAIN       ; Try next neighbor

BAD     ADD R1, R1, #1
RESTORE LDR R7, R6, #0
        ADD R6, R6, #1
        LDR R3, R6, #0
        ADD R6, R6, #1
        LDR R2, R6, #0
        ADD R6, R6, #1
        LDR R0, R6, #0
        ADD R6, R6, #1
        RET

BREADCRUMB .FILL x8000
        .END
```

Your job is to construct the data structure representing a particular map. Before executing JSR EXAMINE, R0 is set to x6100 (the address of one of the nodes), and a breakpoint is set at x4012. The following table shows relevant information collected each time the breakpoint was encountered during the running of EXAMINE.

PC	R0	R2	R7
x4012	x6200	x0042	x0052
x4012	x6100	x0052	x0042
x4012	x6300	x0052	x0047
x4012	x6200	x0047	x0052
x4012	x6400	x0047	x0052
x4012	x6100	x0052	x0042
x4012	x6300	x0052	x0047
x4012	x6500	x0052	x0047
x4012	x6100	x0047	x0042
x4012	x6200	x0047	x0052
x4012	x6400	x0047	x0052
x4012	x6500	x0052	x0047
x4012	x6400	x0042	x0052
x4012	x6500	x0042	x0047

Construct the data structure for the particular map that corresponds to the relevant information obtained from the breakpoints. *Note:* We are asking you to construct the data structure as it exists AFTER the recursive subroutine has executed.

x6100	x8000
x6101	x0042
x6102	x6200
x6103	x6400
x6104	x6500
x6105	x0000
x6106	

x6200	x8000
x6201	x0052
x6202	x6100
x6203	x6300
x6204	x6500
x6205	x0000
x6206	

x6300	x8000
x6301	x0047
x6302	x6200
x6303	x6400
x6304	x0000
x6305	
x6306	

x6400	x8000
x6401	x0052
x6402	x6100
x6403	x6300
x6404	x6500
x6405	x0000
x6406	

x6500	x8000
x6501	x0047
x6502	x6100
x6503	x6200
x6504	x6400
x6505	x0000
x6506	

data structure

① flag = x8000 → visited
② Color of node
③ List of neighbor's Address
④ x0000 End of List

regs

R0 : neighbor's address
R2 : current node's color
R7 : neighbor's color