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?

*Answer:*

When values are pushed and popped to and from a stack implemented in sequential memory locations, the data already stored on the stack does not physically move.

In Figure 8.8, as each value is added to the stack or removed from the stack, the values already on the stack move. But in Figure 8.9, each value is added to the stack or removed from the stack by moving the stack pointer.

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?

*Answer:*

|  |
| --- |
|  |
|  |
|  |
|  |
| B |
| A |

|  |
| --- |
|  |
|  |
|  |
|  |
| B |
| A |

|  |
| --- |
|  |
|  |
|  |
| D |
| C |
| A |

|  |
| --- |
|  |
|  |
|  |
| D |
| C |
| A |

|  |
| --- |
|  |
|  |
|  |
| E |
| C |
| A |

|  |
| --- |
|  |
|  |
|  |
| E |
| C |
| A |

|  |
| --- |
|  |
|  |
|  |
| E |
| C |
| A |

|  |
| --- |
|  |
|  |
|  |
| E |
| F |
| A |

1. A and F are contained in the stack.
2. PUSH J or PUSH K has the most elements in the stack, with 6 elements.

|  |
| --- |
|  |
|  |
|  |
| E |
| F |
| A |

|  |
| --- |
| J |
| I |
| H |
| G |
| F |
| A |

|  |
| --- |
| J |
| I |
| H |
| G |
| F |
| A |

|  |
| --- |
| K |
| I |
| H |
| G |
| F |
| A |

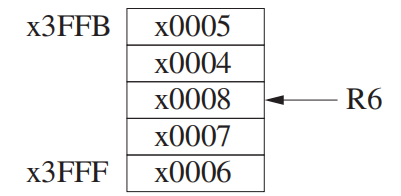
|  |
| --- |
| K |
| I |
| H |
| G |
| F |
| A |

|  |
| --- |
| K |
| I |
| L |
| G |
| F |
| A |

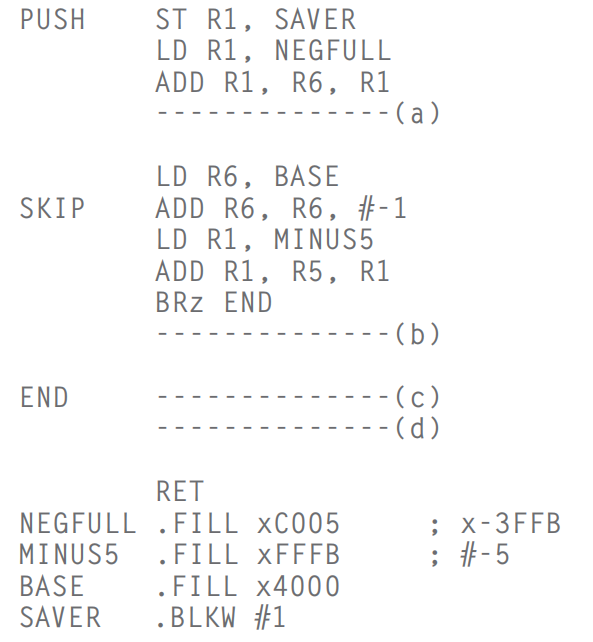
|  |
| --- |
| K |
| I |
| L |
| G |
| F |
| A |

|  |
| --- |
| K |
| I |
| L |
| **M** |
| **F** |
| **A** |

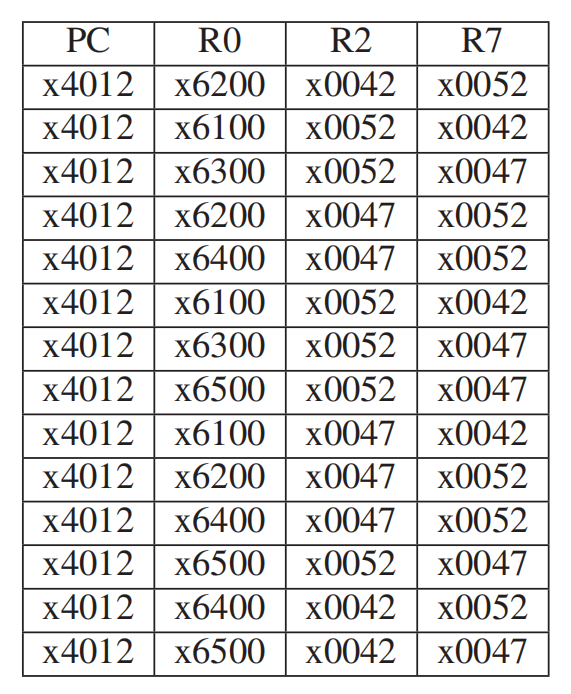
1. A and F and M are contained in the stack.

7.30 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.

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.



1. BRnp SKIP
2. ADD R5, R5, #1
3. STR R0, R6, #0
4. LD R1, SAVER

\*8.10 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.

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.

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.

*Answer:*

First: R2=MEM[x6101], R3=x6102, R0=MEM[x6102], R7=MEM[MEM[x6102]+1]

Second: R2=MEM[x6201], R3=x6202, R0=MEM[x6202], R7=MEM[MEM[x6202]+1]

Third: restore. R3=x6203, R0=MEM[x6203], R7=MEM[MEM[x6203]+1]

Fourth: R2=MEM[x6301], R3=x6302, R0=MEM[x6302], R7=MEM[MEM[x6302]+1]

Fifth: restore. R3=x6303, R0=MEM[x6303], R7=MEM[MEM[x6303]+1]

Sixth: R2=MEM[x6401], R3=x6402, R0=MEM[x6402], R7=MEM[MEM[x6402]+1]

Seventh: restore. R3=x6403, R0=MEM[x6403], R7=MEM[MEM[x6403]+1]

Eighth: restore. R3=x6404, R0=MEM[x6404], R7=MEM[MEM[x6404]+1]

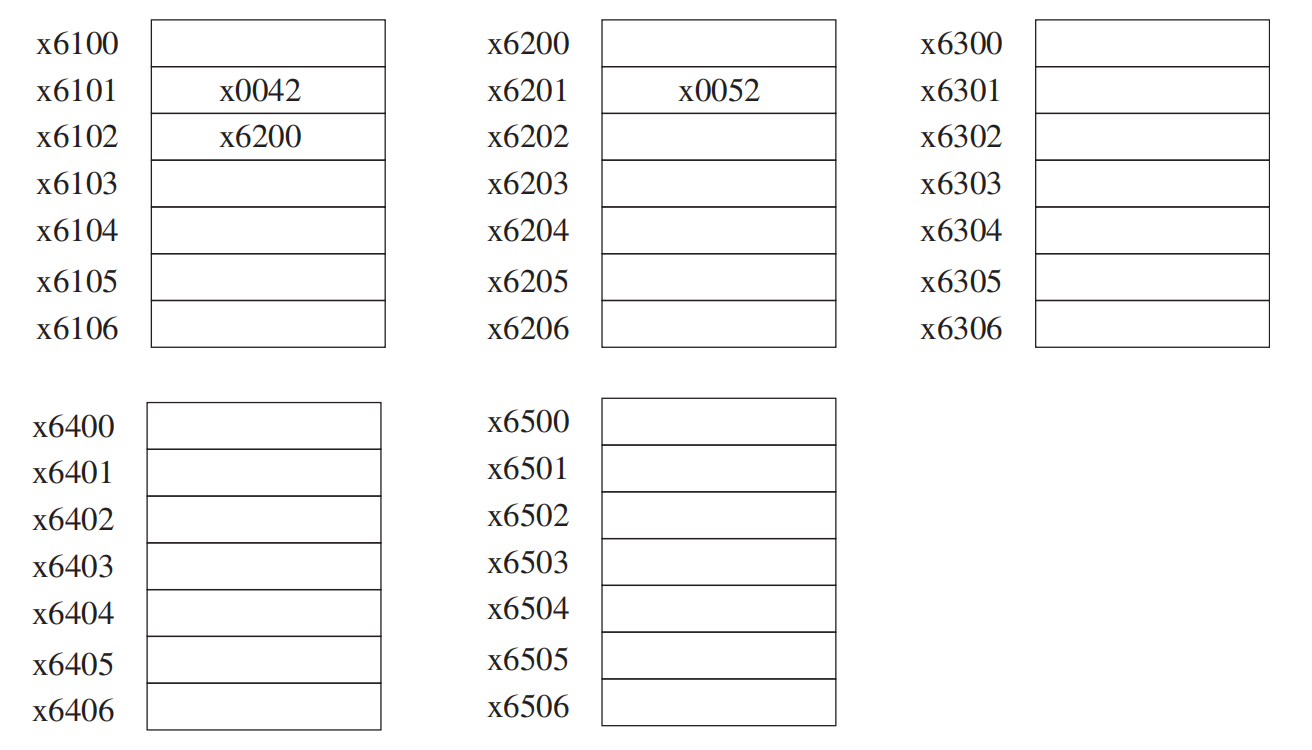
Ninth: R2=MEM[x6501], R3=x6502, R0=MEM[x6502], R7=MEM[MEM[x6502]+1]

Tenth: restore. R3=x6503, R0=MEM[x6503], R7=MEM[MEM[x6503]+1]

Eleventh: restore. R3=x6504, R0=MEM[x6504], R7=MEM[MEM[x6504]+1]

Twelfth: restore. R3=x6505, R0=MEM[x6505], R7=MEM[MEM[x6505]+1]

This time, R7==R2, so R1=1.



x6100

x6300

x0047

x6200

x6400

x0052

x6100

x6300

x6500

x0047

x6200

x6100

x6500

x6400