Mc Graw Hill Education

Because learning changes everything.®

From Bits and Gates to C and Beyond

Data Structures

Chapter 8

Data Structures

Basic data types (Chapter 2): integer, floating-point number, character

Each represents a single value, typically using 1 to 2 words of memory.

Often directly supported by machine instructions.

Data structure = a more complex data item, or a collection of items, composed of the basic types

- Examples: company organization chart, music playlist, schedule of courses, etc.
- Array = sequence of values, like the character file in earlier chapters
- Stack = Last-in, First-out (LIFO) sequence
- Queue = First-in, First-out (FIFO) sequence
- String = sequence of characters

Subroutine: Reusable Module

Before implementing data structures, we introduce the **subroutine --** a mechanism for writing reusable code modules. These will be useful for implementing abstract operations on data structures.

There are different names for this concept in various languages, including **functions** (C) and **methods** (C++, Java).

Main idea:

- We want to write code once, and use it many times.
- We need a way to transfer control to the subroutine, possibly giving it some data to work on **--** this is a **subroutine call**.
- When the subroutine has finished its work, we need a way to get back to the program who called it **--** this is the **return**.

We will show the call and return mechanisms provided by the LC-3.

Example: Convert Repeated Code to a Subroutine


```
01  START   ST      R1,SaveR1   ; Save registers needed
02          ST      R2,SaveR2   ; by this routine
03          ST      R3,SaveR3
04  ;
05          LD      R2,Newline
06  L1      LDI     R3,DSR
07          BRzp    L1          ; Loop until monitor is ready
08          STI     R2,DDR      ; Move cursor to new clean line
09  ;
0A          LEA     R1,Prompt   ; Starting address of prompt string
0B  Loop    LDR     R0,R1,#0    ; Write the input prompt
0C          BRz     Input       ; End of prompt string
0D  L2      LDI     R3,DSR
0E          BRzp    L2          ; Loop until monitor is ready
0F          STI     R0,DDR      ; Write next prompt character
10          ADD     R1,R1,#1    ; Increment prompt pointer
11          BRnzp   Loop        ; Get next prompt character
12  ;
13  Input   LDI     R3,KBSR
14          BRzp    Input       ; Poll until a character is typed
15          LDI     R0,KBDR     ; Load input character into R0
16  L3      LDI     R3,DSR
17          BRzp    L3          ; Loop until monitor is ready
18          STI     R0,DDR      ; Echo input character
19  ;
1A  L4      LDI     R3,DSR
1B          BRzp    L4          ; Loop until monitor is ready
1C          STI     R2,DDR      ; Move cursor to new clean line
1D          LD      R1,SaveR1   ; Restore registers
1E          LD      R2,SaveR2   ; to original values
1F          LD      R3,SaveR3
20          JMP     R7          ; Do the program's next task
21  ;
22  SaveR1  .BLKW   1           ; Memory for registers saved
23  SaveR2  .BLKW   1
24  SaveR3  .BLKW   1
25  DSR     .FILL   xFE04
26  DDR     .FILL   xFE06
27  KBSR    .FILL   xFE00
28  KBDR    .FILL   xFE02
29  Newline .FILL   x000A       ; ASCII code for newline
2A  Prompt  .STRINGZ ''Input a character>''
```

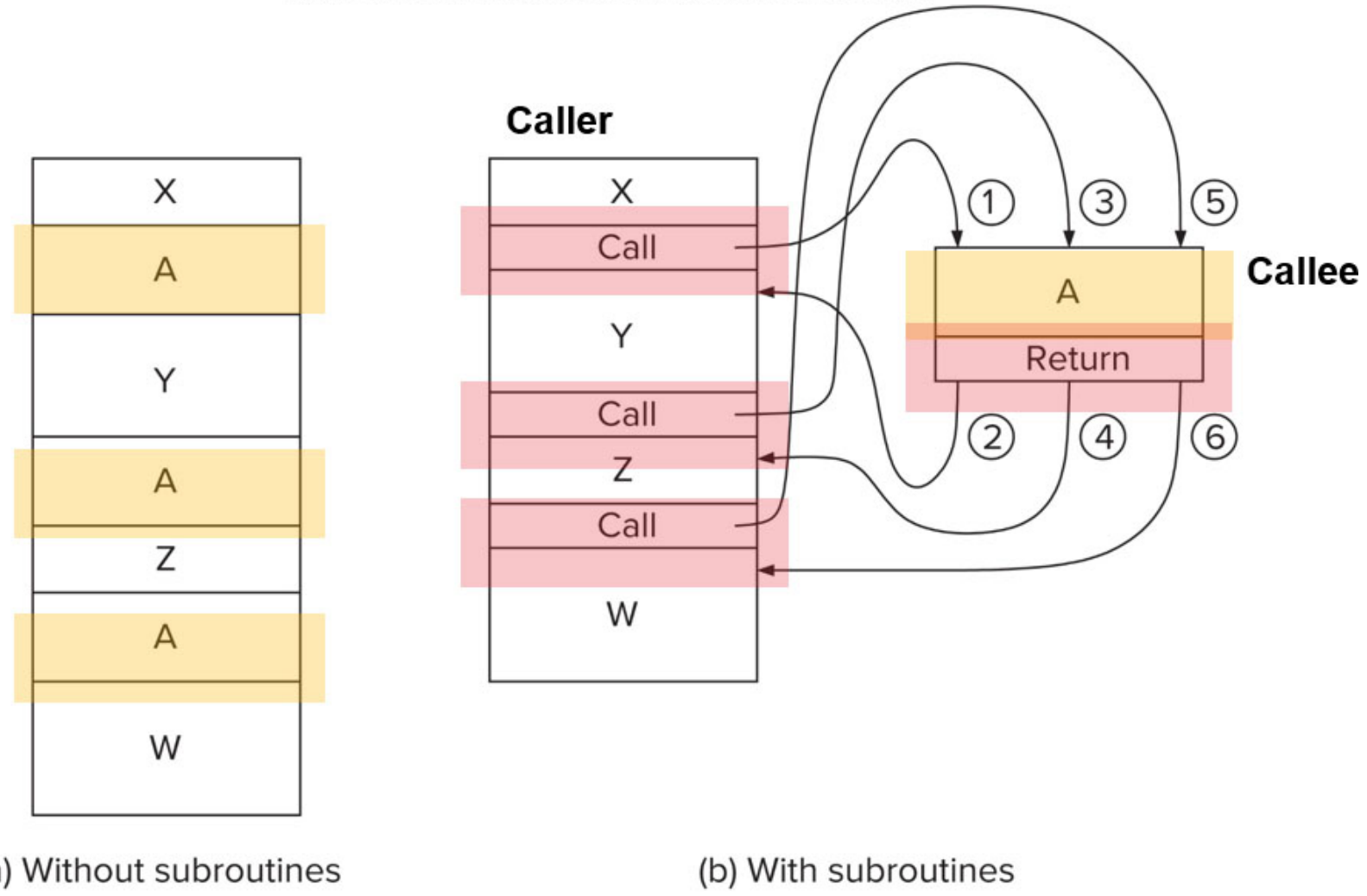
Here is some code from Figure 8.1. Will be discussed in Chapter 9, so don't worry about what it does, but notice the repetitive highlighted code.

Each uses a different label and different registers, but can be generalized as:

```
label   LDI   R3,DSR
        BRzp  label
        STI   Reg,DDR
```

Wouldn't it be nice to write that code once and use it whenever we need it, rather than writing it four times?

Convert Repeated Code to a Subroutine



(a) Without subroutines

(b) With subroutines

Call + return allow the same code to be executed from multiple places in the program.

Access the text alternative for slide images.

Advantages of Subroutines

Reusability

- Only have to write (and debug!) the code once.
- If you find an error, you can fix it in one place, rather than looking for all the places in the code where you did the same thing.

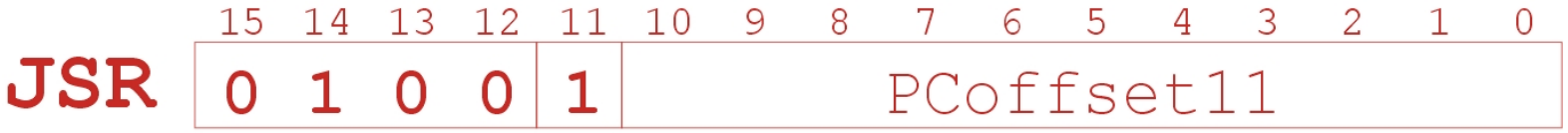
Abstraction

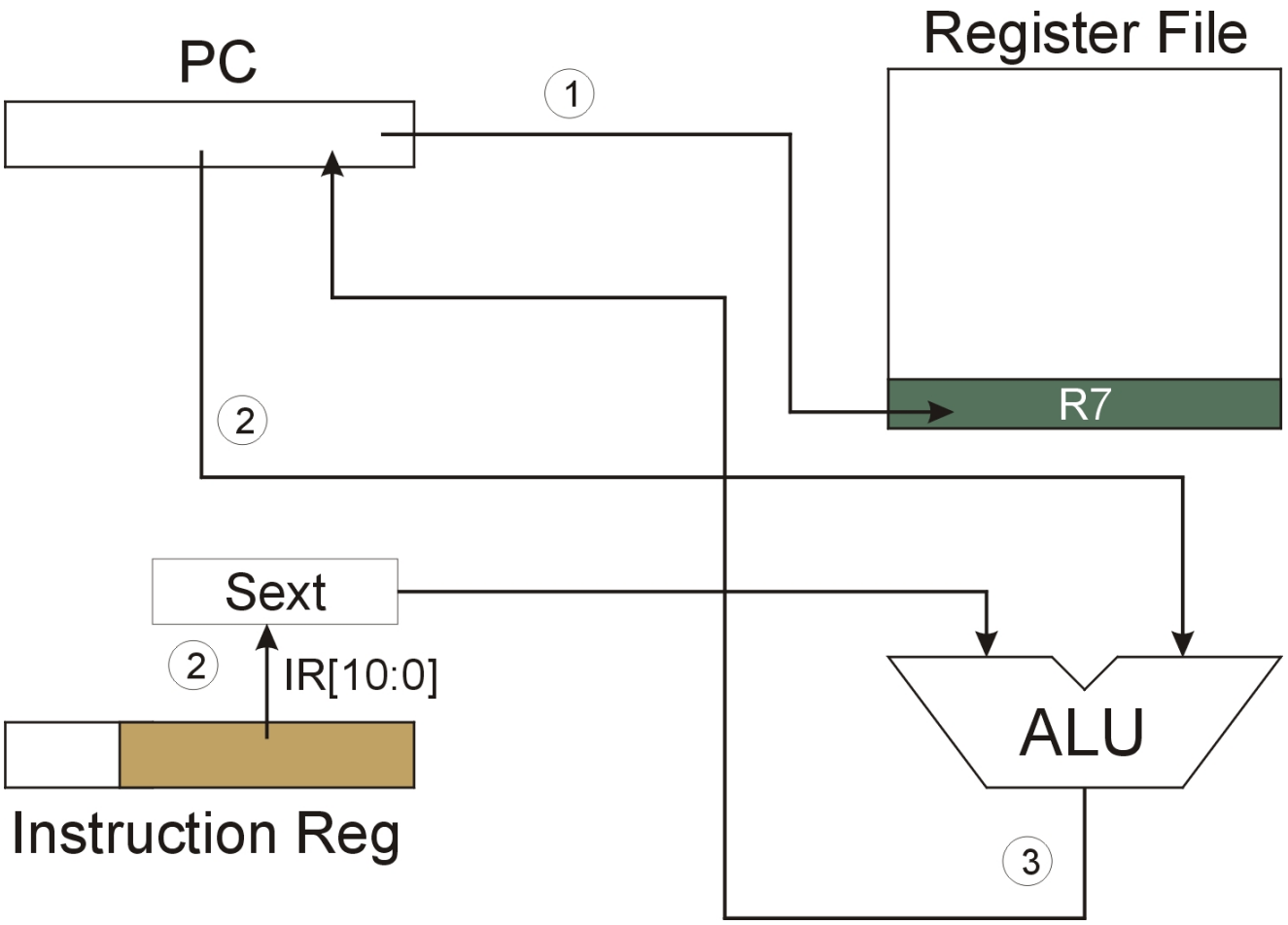
- Encapsulate a complex task (for example, "square root") into a subroutine with a symbolic name (SQRT).
- Long sequence of instructions is replaced with CALL SQRT.
- Program becomes easier to read and understand, and the reader does not get lost in the details of how to compute a square root.

Modularity

- Can divide program up into smaller pieces and assign to different programmers, or solve on different days.

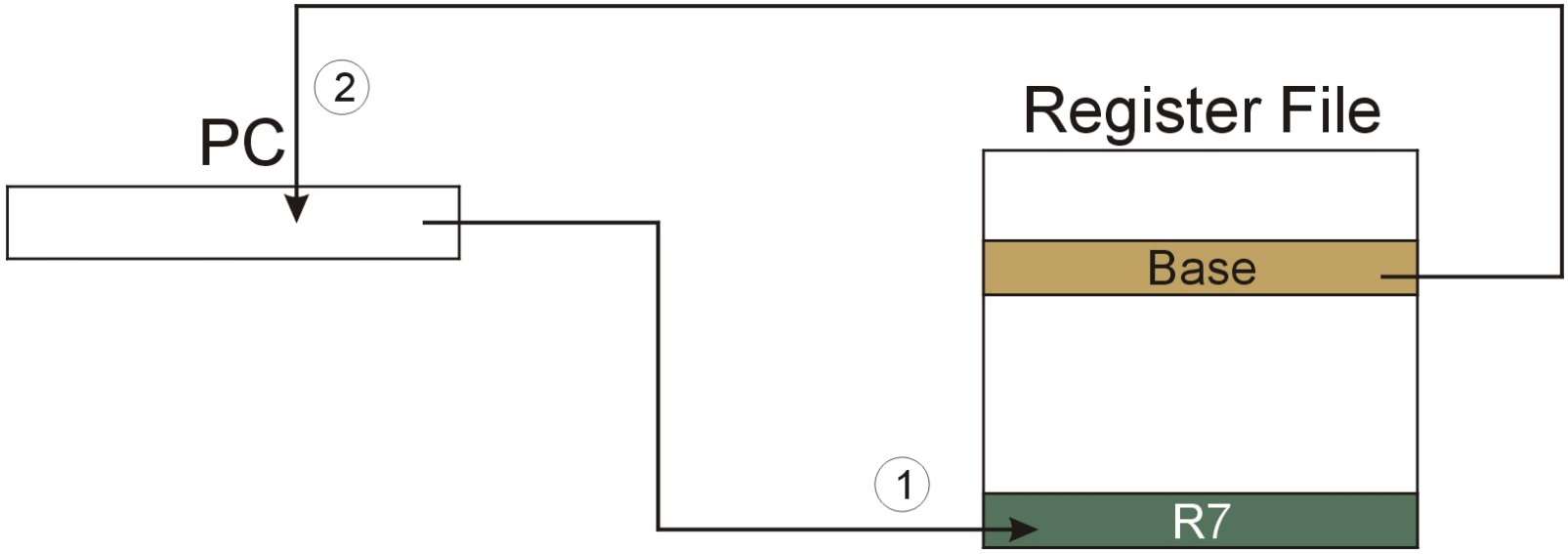
JSR (PC-Relative mode)





Access the text alternative for slide images.

JSRR (Base+offset (zero) mode)



Using a base register allows subroutine to be anywhere in memory.
Same opcode as JSR, but different assembly mnemonic (JSRR).
Bit 11 distinguishes between the two addressing modes.

Access the text alternative for slide images.

Returning from a Subroutine

We must be careful to not change R7 during the execution of the subroutine. When we are ready to return, we want to put R7's value (the saved PC) back into the PC. We already have an instruction for that!

```
JMP R7
```

Because this occurs a lot, and because using R7 as the base register has a special meaning for subroutines, the assembler provides an alternate syntax:

```
RET
```

NOTE that RET is listed in the instruction set in Chapter 5, but it's not a separate instruction. It's just a special case of JMP.

Calling a Subroutine

For the LC-3, we already know how to transfer control to another piece of code. We could use BR (if close) or JMP (if far away).

But how do we get back?

Once we've changed the PC, we've lost track of where in memory the calling code resides.

Solution: A new instruction **JSR** that saves a copy of the PC before changing it.

- If we save a copy of the PC, we can simply write that value back into the PC to get back to the calling code. Remember: PC has already been incremented (during Fetch), so we are returning to the instruction after the JSR.
- LC-3 uses R7 to store the saved PC.

Saving and Restoring Registers

When we write to a register, its previous value is gone forever.

If we need that value later, we must **save** it somewhere else, before overwriting its register. Once we've saved it, we can **restore** it (put it back into the register) when we need it again.

Where do we save it?

- Copy it to another register.
  OK, but there are only a few registers in the LC-3, so that's not likely to be a general solution.
- Store it to a memory location.
  Yes -- use .BLKW to reserve a memory location in the program where we can store a register's value to save it, and we can load it when we need to restore.

Saving and Restore Registers in Subroutines

A subroutine is separate and independent from the caller code.
The subroutine will need to use registers to perform its job, but it does not know which registers have information needed by the caller.

Therefore...

> A subroutine should **save** any register that it changes, and **restore** that register to the caller's value before it returns.

This is known as a callee-save strategy, because it's the callee code that is responsible for saving/restoring registers.

One exception: The *caller* must save R7 (if needed) before the call, because JSR changes R7 before the callee code is executed.

Interface: Passing Data Into and Out of a Subroutine

We often need to give information to the subroutine.

> Example: If we have a square root subroutine (SQRT), how does the subroutine know what value to take the square root of?

This is known as "passing" data into a subroutine.

Likewise, there is often information that we want back from the subroutine.

> Example: If we call SQRT, how do we get the computed value?

This is known as returning information from the subroutine, or passing data out.

The scheme for passing data into and out of a subroutine is known as its **interface**. It must be documented and followed by every caller to this subroutine.

Passing Data in the LC-3

The JSR(R) instruction only provides a way to transfer control to the subroutine -- it does not provide any means of transferring data in and out.

In the LC-3, there are only two places where data can be stored:

- Registers (R0 to R6) -- can't use R7, because it will contain the saved PC.
- Memory.

Either will work, and there are pros and cons to each approach. Generally, for a small amount of data, registers are easy to use. For large collections of data, we can pass a pointer (address) that gives the location of the data.

The **interface** tells how the transfer of data will work, and both the caller and the callee must follow the interface.

Example Subroutine: Summing an Array of Integers 1

```
; Subroutine: ArraySum
; Computes the sum of a sequence of integers
; Inputs:
;    R0 = pointer to the starting address of the array
;    R1 = number of integers in the array (size)
; Output:
;    R2 = computed sum

ArraySum   ST   R0,ASumR0     ; save registers
           ST   R1,ASumR1
           ST   R3,ASumR3

           AND  R2,R2,#0      ; initialize sum to zero

           ADD  R1,R1,#0      ; check size -- if <= 0, exit
           BRnz ASDone
```

Example Subroutine: Summing an Array of Integers 2

```
ASLoop      LDR   R3,R0,#0     ; load value from array
            ADD   R2,R2,R3     ; add to computed sum
            ADD   R0,R0,#1     ; point to next value
            ADD   R1,R1,#-1    ; until counter is zero
            BRp   ASLoop

; output value (sum) is in R2, ready to return

ASDone      LD    R3,ASumR3    ; restore registers
            LD    R1,ASumR1
            LD    R0,ASumR0
            RET

ASumR0      .BLKW 1            ; space for saved registers
ASumR1      .BLKW 1
ASumR3      .BLKW 1
```

Library Subroutines

It's common for a collection of related subroutines to be packaged together into a **library**. The programmer can then use the subroutines, according to the documented interface, without concern about the details of implementation.

When symbols (for example, subroutine names) are defined in modules, the EXTERNAL mechanism discussed in Chapter 7 can be used.


```
01              ...
02              ...
03              .EXTERNAL SQRT
04              ...
05              ...
06              LD      R0,SIDE1
07              BRz     S1
08              JSR     SQUARE
09      S1      ADD     R1,R0,#0
0A              LD      R0,SIDE2
0B              BRz     S2
0C              JSR     SQUARE
0D      S2      ADD     R0,R0,R1 ; R0 contains argument x
0E              LD      R4,BASE  ; BASE contains starting address of SQRT routine
0F              JSRR    R4
10              ST      R0,HYPOT
11              BRnzp   NEXT_TASK
12      SQUARE  ADD     R2,R0,#0
13              ADD     R3,R0,#0
14      AGAIN   ADD     R2,R2,#-1
15              BRz     DONE
16              ADD     R0,R0,R3
17              BRnzp   AGAIN
18      DONE    RET
19      BASE    .FILL   SQRT
1A      SIDE1   .BLKW   1
1B      SIDE2   .BLKW   1
1C      HYPOT   .BLKW   1
1D              ...
1E              ...
```

Abstract Data Types

Now, we're ready to talk about data structures.

Each data structure is described as an abstract data type.

Instead of simply describing how data is stored in memory, we specify **operations** on the data structure, implemented as subroutines.

By using the subroutines, the caller program is separated from the detailed implementation and storage details. This provides a level of abstraction -- if the implementation changes, the caller does not need to be changed.

We will show an implementation of each data structure, but keep in mind that there are alternatives -- a topic for another class.

Stack

A stack is a last-in, first-out (LIFO) sequence of data items.

The **last** thing you put in the stack will be the **first** thing removed.

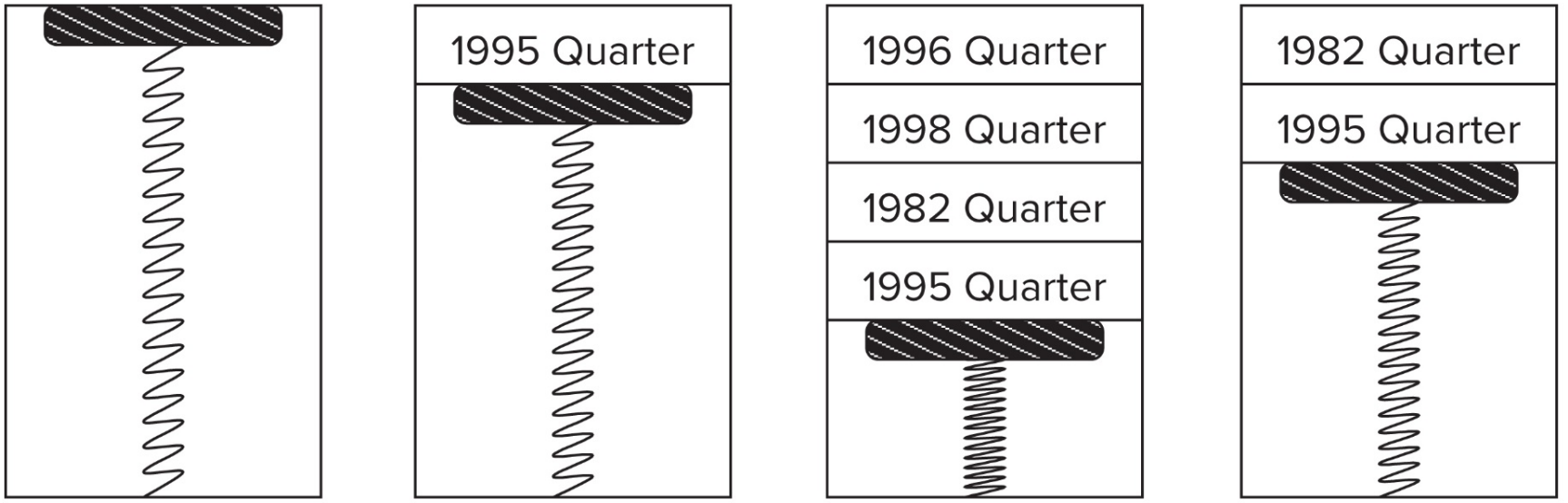
The operations defined for a stack are:

push an item onto the stack, and

pop an item from the stack.

Stack: Coin Holder Analogy

Consider a coin holder that contains quarters. We identify each coin by the year it was minted.



(a) Initial state (Empty) (b) After one push (c) After three pushes (d) After two pops

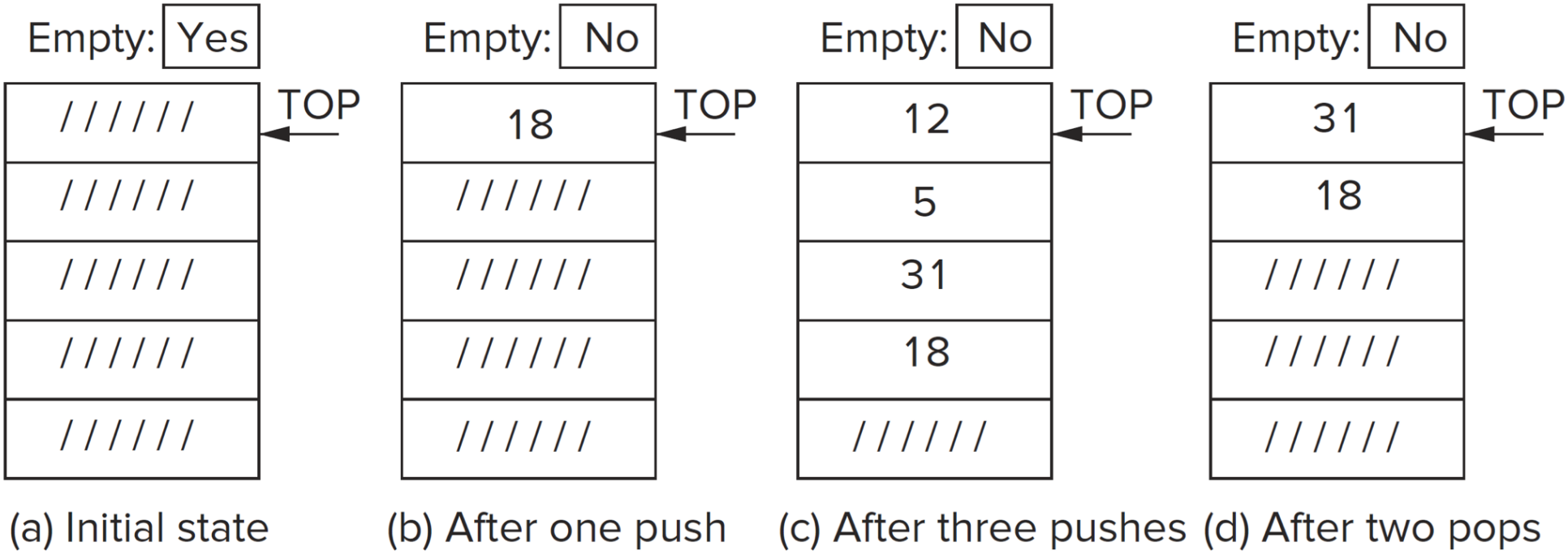
Coins are popped in the opposite order from when they are pushed: Last-in, First-out.

Access the text alternative for slide images.

Digital Hardware Stack

If we want to implement a stack of integers using digital hardware, we can provide a number of registers and move data between the registers on each push/pop operation.

We use an additional register to keep track of whether the stack is empty.

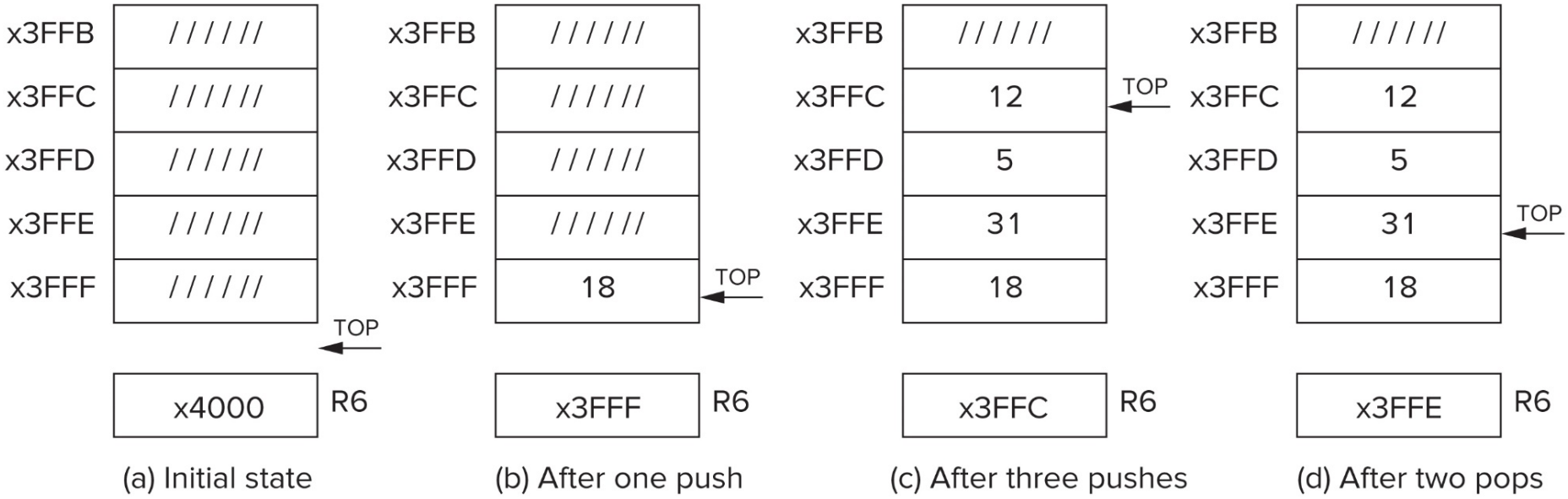


Access the text alternative for slide images.

Software Stack in Memory

If we use the same approach in software, where the stack data is stored in memory locations, it requires a lot of loads and stores to move all of the data for each push/pop.

Instead, we move a pointer to the top of stack, and leave the data in place. In the figure, we use R6 to hold the address of the top of stack. By convention, we grow the stack downward, to lower addresses.



Access the text alternative for slide images.

LC-3 Implementation 1

If R6 holds the top-of-stack (TOS) pointer, the following code is used to push the value in R0 onto the stack.

```
PUSH    ADD R6,R6,#-1   ; move ptr to make room
        STR R0,R6,#0
        RET
```

The following code will pop the value at the TOS to R0. The stack pointer (R6) is incremented to deallocate that memory location.

```
POP     LDR R0,R6,#0
        ADD R6,R6,#1    ; move ptr to remove item
        RET
```

By adding RET to the end, we've made subroutines that can abstract the stack operations for users. It may be silly to have such a small subroutine, but we're about to make them more sophisticated...

Stack Underflow

What happens if we pop more times than we have pushed?

In the previous code, R6 keeps getting larger and we load garbage data that was never pushed on the stack.

You could say that the user of the stack shouldn't do that, and you would be right, but we can also make the subroutine a little friendlier by detecting the underflow condition.

In the version of POP on the next slide, we use R0 to get the value, and we use R5 to indicate whether underflow occurs. If R5=0, there's no overflow and the popped data is valid. If R5=1, then underflow was detected, and the value in R0 is unchanged from before.

NOTE: We must predetermine the memory location of the stack, and some code must initialize the stack pointer (R6) to point at the next highest memory location.

POP subroutine with Underflow Detection

```
POP         ST   R1,POP_SR1     ; save registers

            AND  R5,R5,#0       ; assume no overflow
            LD   R1,EMPTY       ; check whether R6 is at bottom
            ADD  R1,R6,R1
            BRz  POP_Fail

            LDR  R0,R6,#0       ; if no underflow, get data
            ADD  R6,R6,#1       ; increment stack pointer
            LD   R1,POP_SR1     ; restore registers
            RET                 ; return with R5=0

POP_Fail    ADD  R5,R5,#1       ; set R5 to indicate underflow
            LD   R1,POP_SR1     ; restore registers
            RET

EMPTY       .FILL xC000         ; negative of x4000
POP_SR1     .BLKW 1
```

Stack Overflow

If we keep pushing items, we may overrun the storage that is allocated for the stack -- this is called "overflow."

Similar to the underflow case, the next slide shows a subroutine that checks for overflow and uses R5 to indicate when failure has occurred.

In this case, we assume that the stack is allocated five memory locations, from x3FFB to x3FFF. R6 should never be allowed to go lower than x3FFB.

PUSH subroutine with Overflow Detection

```
PUSH        ST   R1,POP_SR1     ; save registers

            AND  R5,R5,#0       ; assume no overflow
            LD   R1,FULL        ; check whether R6 is at top
            ADD  R1,R6,R1
            BRz  PUSH_Fail

            ADD  R6,R6,#-1      ; if no overflow, push
            STR  R0,R6,#0
            LD   R1,PUSH_SR1    ; restore registers
            RET                 ; return with R5=0

PUSH_Fail   ADD  R5,R5,#1       ; set R5 to indicate underflow
            LD   R1,PUSH_SR1     ; restore registers
            RET

FULL        .FILL xC005         ; negative of x3FFB
POP_SR1     .BLKW 1
```

Combined PUSH/POP Implementation

Figure 8.11 shows an implementation of both PUSH and POP where code and storage locations are shared for efficiency.

```
POP             AND     R5,R5,#0        ; R5 <-- success
                ST      R1,Save1        ; Save registers that
                ST      R2,Save2        ; are needed by POP
                LD      R1,EMPTY        ; EMPTY contains -x4000
                ADD     R2,R6,R1        ; Compare stack pointer to x4000
                BRz     fail_exit       ; Branch if stack is empty
;
                LDR     R0,R6,#0        ; The actual "pop"
                ADD     R6,R6,#1        ; Adjust stack pointer
                BRnzp   success_exit
;
PUSH            AND     R5,R5,#0
                ST      R1,Save1        ; Save registers that
                ST      R2,Save2        ; are needed by PUSH
                LD      R1,FULL         ; FULL contains -x3FFB
                ADD     R2,R6,R1        ; Compare stack pointer to x3FFB
                BRz     fail_exit       ; Branch if stack is full
;
                ADD     R6,R6,#-1       ; Adjust stack pointer
                STR     R0,R6,#0        ; The actual "push"
success_exit    LD      R2,Save2        ; Restore original
                LD      R1,Save1        ; register values
                RET
;
fail_exit       LD      R2,Save2        ; Restore original
                LD      R1,Save1        ; register values
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
;
EMPTY           .FILL   xC000           ; EMPTY contains -x4000
FULL            .FILL   xC005           ; FULL contains  -x3FFB
Save1           .FILL   x0000
Save2           .FILL   x0000
```

Recursion: Can a Subroutine Call Itself?

Recursion is a technique in which a subroutine (function) can call itself.

Advantage: Can be an elegant and efficient expression of a problem.

Disadvantage: When used improperly, can introduce significant run-time overhead in execution time and memory usage.

Why discuss now? A **stack** is a natural data structure to use for recursion.

Will show two bad examples (but useful for explaining the concepts) and one good example of using recursion to solve problems.

Bad Example #1: Factorial

Factorial is a mathematical function that computes the produce of the first n positive integers. The symbol for "n factorial" is n!

$$n! = n \times (n-1) \times (n-2) \times \boxed{?} \times 2 \times 1$$

A simpler (recursive) way to define it:

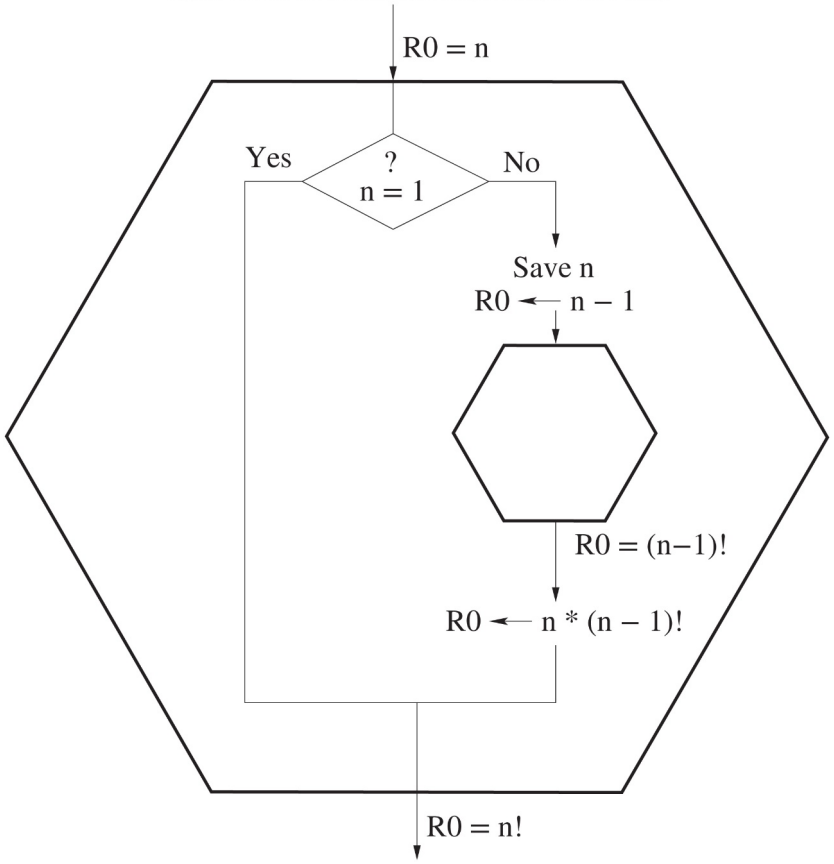
$$n! = n \times (n-1)!$$

We also need to note that $1! = 1$ That's what allows us to multiply only the numbers > 0.

Now factorial is defined in terms of itself.

A Factorial Subroutine

If we have a factorial subroutine FACT that takes a input n, then we could call FACT to compute the factorial of (n − 1) and multiply by n. If n = 1, we just return 1 without calling FACT.




```
FACT    ST    R1, Save1
        ADD   R1,R0,#-1
        BRz   DONE
        ADD   R1,R0,#0
        ADD   R0,R1, #-1
B       JSR   FACT
        MUL   R0,R0,R1
DONE    LD    R1, Save1
        RET
Save1   .BLKW 1
```

For the purposes of this example, we are pretending that the LC-3 has a multiply (MUL) opcode.

Access the text alternative for slide images.

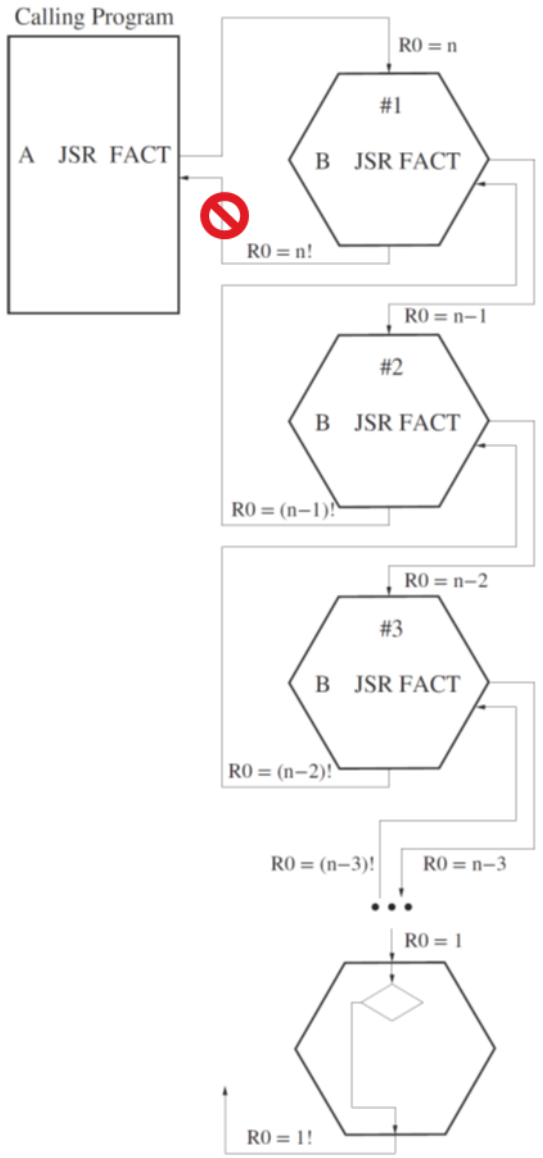
Problem: Overwritten Return Address (R7)

Unfortunately, the code does not work.

When original caller calls FACT, it sets the return address (R7) to A+1.

When FACT calls FACT, it sets the return address to B+1, overwriting the old value of R7. Therefore, when we are ready to return the original caller, we have lost the return address information and there's no way to get back.

Solution: Push R7 onto a **stack** before calling, and pop R7 from the stack before returning.



Access the text alternative for slide images.

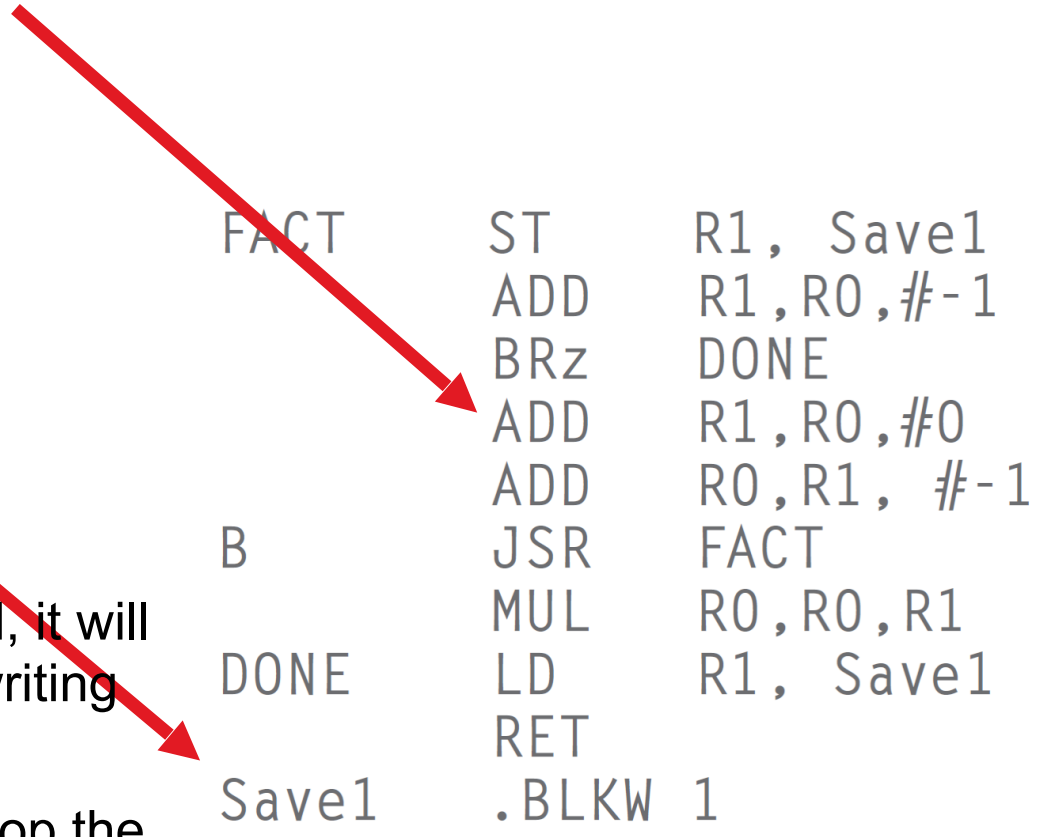
Still a Problem: Overwriting Local Data

Recall that we save R0 (n) into R1, so that we can multiply by n when FACT(n − 1) is returned.

Because the subroutine modifies R1, we are careful to save/restore it using this memory location.

However, the next time FACT is called, it will use the same memory location, overwriting the old data.

Solution: Push R0 onto a stack, and pop the stack before the multiply.



```
FACT    ST    R1, Save1
        ADD   R1,R0,#-1
        BRz   DONE
        ADD   R1,R0,#0
        ADD   R0,R1, #-1
B       JSR   FACT
        MUL   R0,R0,R1
DONE    LD    R1, Save1
        RET
Save1   .BLKW 1
```

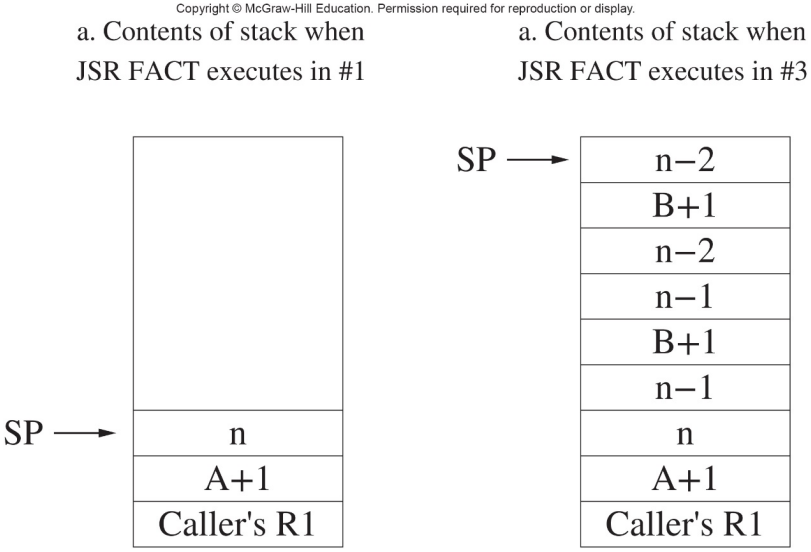
Correct Recursive Subroutine: Factorial

Figure 8.14 shows a correct implementation of the FACT subroutine, incorporating the stack solution of the previous problems.


```
FACT        ADD  R6,R6,#-1
            STR  R1,R6,#0   ; Push Caller's R1 on the stack, so we can use R1.
;
            ADD  R1,R0,#-1  ; If n=1, we are done since 1! = 1
            BRz  NO_RECURSE
;
            ADD  R6,R6,#-1
            STR  R7,R6,#0   ; Push return linkage onto stack
            ADD  R6,R6,#-1
            STR  R0,R6,#0   ; Push n on the stack
;
            ADD  R0,R0,#-1   ; Form n-1, argument of JSR
B           JSR  FACT
            LDR  R1,R6,#0   ; Pop n from the stack
            ADD  R6,R6,#1
            MUL  R0,R0,R1   ; form n*(n-1)!
;
            LDR  R7,R6,#0   ; Pop return linkage into R7
            ADD  R6,R6,#1
NO_RECURSE  LDR  R1,R6,#0   ; Pop caller's R1 back into R1
            ADD  R6,R6,#1
            RET
```

Correct, but not Efficient

So why do we call this a bad example? Because the space and time required to push and pop data from the stack is large. The space required to the stack increases linearly with the input variable (n).



Stack for n = 3.

If we implement the algorithm with a loop (iteration), the number of instructions is smaller and there is no need for a stack.


```
FACT     ST   R1,SAVE_R1
         ADD  R1,R0,#0
         ADD  R0,R0, #-1
         BRz  DONE
AGAIN    MUL  R1,R1,R0
         ADD  R0,R0,#-1   ; R0 gets next integer for MUL
         BRnp AGAIN
DONE     ADD  R0,R1,#0    ; Move n! to R0
         LD   R1,SAVE_R1
         RET
SAVE_R1  .BLKW 1
```

Access the text alternative for slide images.

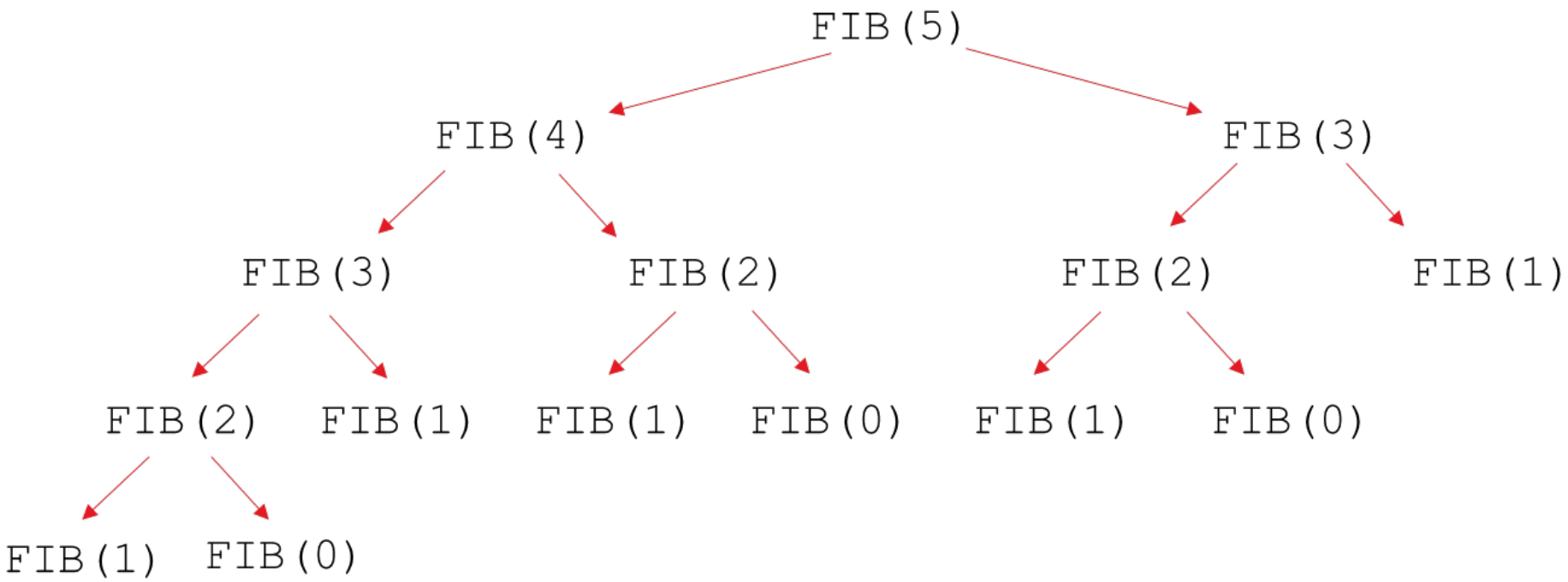
Bad Example #2: Fibonacci

Another popular example of recursion is calculating the n-th number in the Fibonacci series. Each number is equal to the sum of the previous two numbers in the series. Mathematically:

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-2) + f(n-1), & n \geq 2 \end{cases}$$

Based on the previous example, you can probably guess that FIB(*n*) would call FIB(*n* − 2) and FIB(*n* − 1), and that we can use a stack to keep track of return addresses, local temporary values, etc.

Problem: Repeated Computation



FIB(3) is called 2 times
FIB(2) is called 3 times
FIB(1) is called 5 times
FIB(0) is called 3 times

Iterative approach (next slide) does not repeat and is **much** faster!

Access the text alternative for slide images.

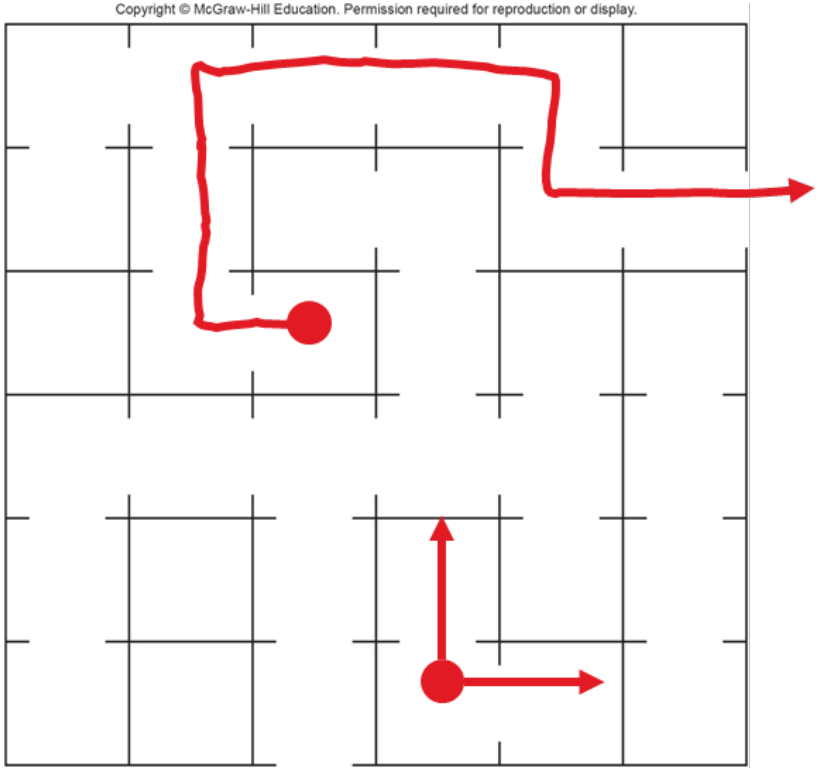
Iterative Fibonacci


```
FIB     ST    R1,SaveR1
        ST    R2,SaveR2
        ST    R3,SaveR3
        ST    R4,SaveR4
        ST    R5,SaveR5
;
    NOT   R0,R0
    ADD   R0,R0,#1  ; R0 contains -n
    AND   R1,R1,#0  ; Suppose n=0
    ADD   R5,R1,R0  ; R5 = 0 -n
    BRz   DONE      ; if n=0, done almost
    AND   R3,R2,#0  ; if n>0, set up R3 = FIB(0) = 0
    ADD   R1,R3,#1  ; Suppose n=1
    ADD   R5,R1,R0  ; R5 = 1-n
    BRz   DONE      ; if n=1, done almost
    ADD   R4,R1,#0  ; if n>1, set up R4 = FIB(1) = 1
;
AGAIN   ADD   R1,R1,#1  ; We begin the iteration of FIB(i)
    ADD   R2,R3,#0  : R2= FIB(i-2)
    ADD   R3,R4,#0  : R3= FIB(i-1)
    ADD   R4,R2,R3  ; R4 = FIB(i)
    ADD   R5,R1,R0  ; is R1=n ?
    BRn   AGAIN
;
    ADD   R0,R4,#0  ; if n>1, R0=FIB(n)
    BRnzp RESTORE
DONE    ADD   R0,R1,#0  ; if n=0,1, FIB(n)=n
RESTORE LD    R1,SaveR1
    LD    R2,SaveR2
    LD    R3,SaveR3
    LD    R4,SaveR4
    LD    R5,SaveR5
    RET
```

Good Example: Maze

Sometimes, the elegance of recursion provides a natural way to solve a complicated problem.

Example: Given a starting point in a maze, determine whether there is a path to exit the maze.



Access the text alternative for slide images.

Data Representation

In the 6×6 maze shown, there are 36 starting points.

For each point, use four bits to indicate whether there is a wall in each of the four directions. Use another bit to indicate whether this is an opening to the outside world.

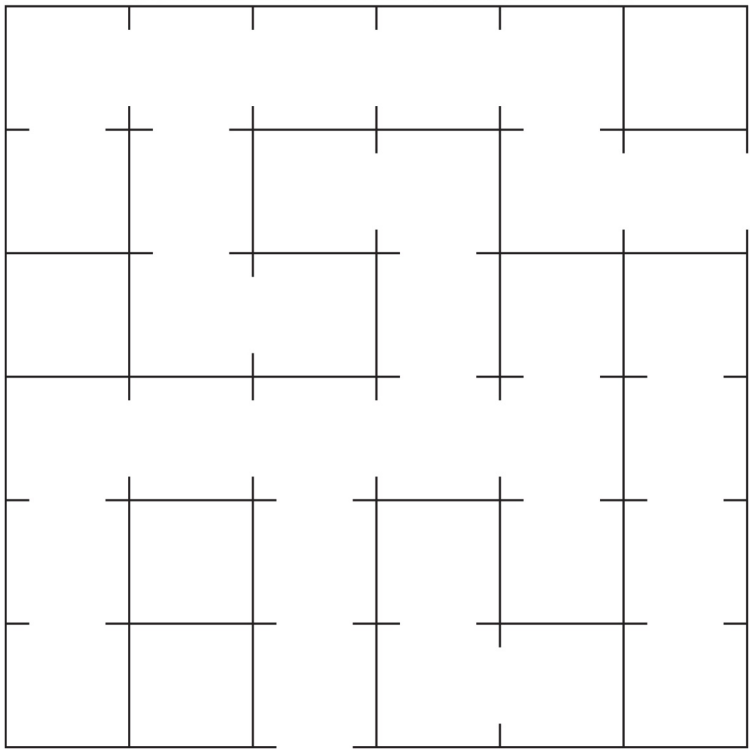
out	N	E	S	W

1 = door
0 = no door

Store data in "row-major" order:

All cells in row 1 (top), left to right.
All cells in row 2, left to right.
Etc.

Maze Data




```
00              .ORIG x5000
01 MAZE         .FILL x0006
02              .FILL x0007
03              .FILL x0005
04              .FILL x0005
05              .FILL x0003
06              .FILL x0000
07 ; second row: indices 6 to 11
08              .FILL x0008
09              .FILL x000A
0A              .FILL x0004
0B              .FILL x0003
0C              .FILL x000C
0D              .FILL x0015
0E ; third row: indices 12 to 17
0F              .FILL x0000
10              .FILL x000C
11              .FILL x0001
12              .FILL x000A
13              .FILL x0002
14              .FILL x0002
15 ; fourth row: indices 18 to 23
16              .FILL x0006
17              .FILL x0005
18              .FILL x0007
19              .FILL x000D
1A              .FILL x000B
1B              .FILL x000A
1C ; fifth row: indices 24 to 29
1D              .FILL x000A
1E              .FILL x0000
1F              .FILL x000A
20              .FILL x0002
21              .FILL x0008
22              .FILL x000A
23 ; sixth row: indices 30 to 35
24              .FILL x0008
25              .FILL x0000
26              .FILL x001A
27              .FILL x000C
28              .FILL x0001
29              .FILL x0008
2A              .END
```

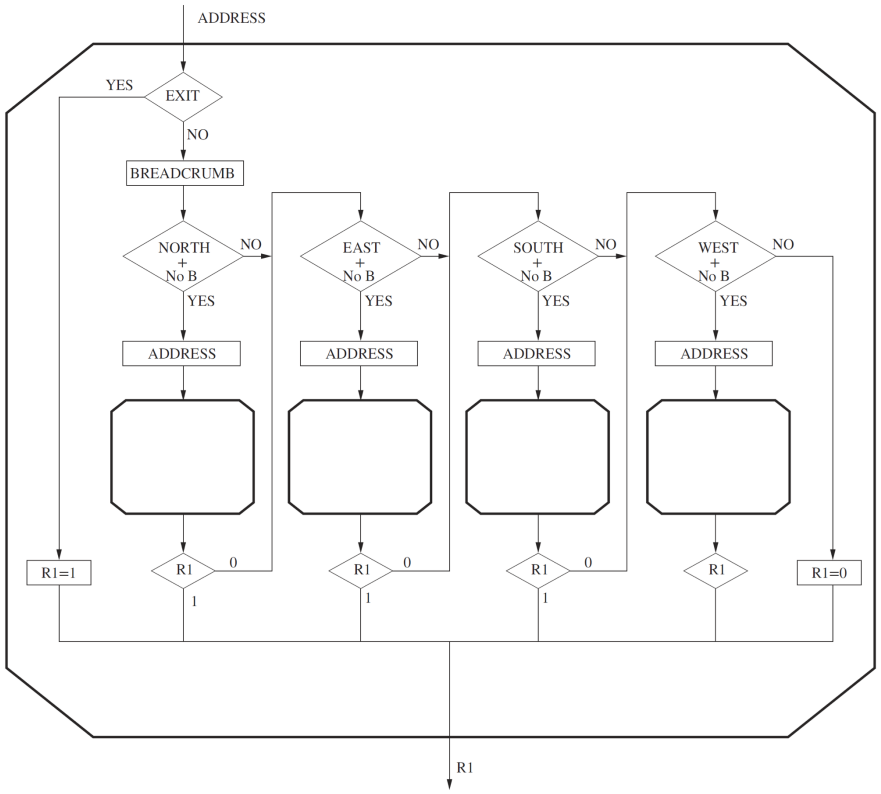
Access the text alternative for slide images.

Recursive Algorithm for Maze Traversal

A sketch of the recursive algorithm is shown in Figure 8.23.

If the current cell has an exit door, then we are done: success!

If not, we first "drop a breadcrumb" for this cell, so that we don't try to exit from here again. We can set bit 15 of this cell's data to show we've been here.



Next, if there's a door to the North, we ask whether there's an exit path from that cell -- this is the recursive call. If the answer is no, we try East, etc. If there is no path in any direction, we know that we are done.

See the implementation in Figure 8.24.

Access the text alternative for slide images.

Why is Maze a Good Example of Recursion?

The recursion greatly simplifies the description of the problem.

- Don't have to keep track of all the paths we've tried.
- Each cell "remembers" whether it has been visited, avoiding the possibility of infinite loops, etc.
- The problem is localized to a single cell. Instead of tracing out all the paths from here, we simply ask the neighbors.

This problem is simplified, because we just want a YES/NO answer. It's a little more complicated to actually determine what the exit path is, or what the shortest exit path would be, but recursion would still be useful in those cases.

An iterative version of this code would be much more complicated.

Stack and Recursion

The **stack** is a good data structure for storing state in a recursive algorithm.

- Return address
- Local state

The subroutine call/return mechanism is last-in, first-out:

Returns are done in the opposite order of calls.

The subroutine returning first is the last subroutine that was called.

This will be discussed again in Chapter 14, for C programs and other high-level languages.

Queue

A queue is a data structure that contains an sequence of items, accessed in first-in, first-out (FIFO) order.

The **first** thing you put in the queue will be the **first** thing removed.

The operations defined for a stack are:

insert an item at the back (rear) of the queue, and

remove an item from front of the queue.

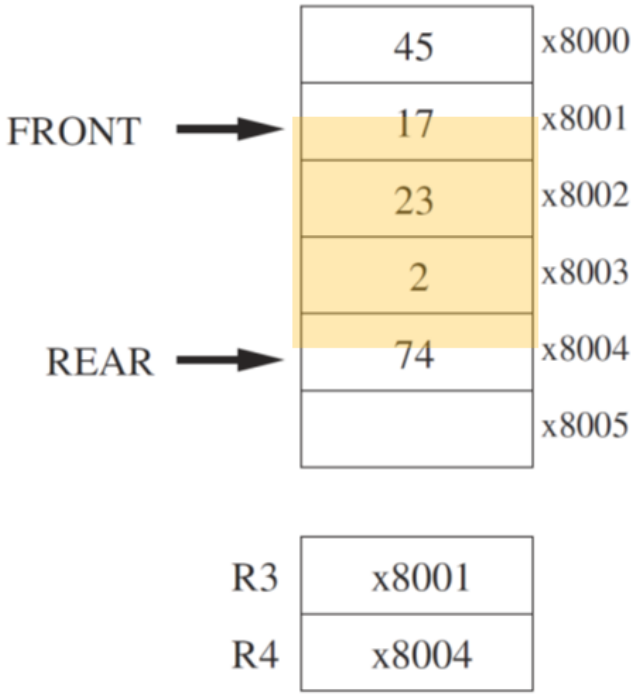
Examples:

a (polite) line at the grocery store,
a sequence of songs in a playlist,
a line of cars at a gas pump

Queue Implementation

To implement a queue, we allocate a region of memory to store data items, and we use **two pointers** to indicate the front and rear. Like the stack, the pointers move only one address at a time.

In the example below, memory x8000 to x8005 is allocated for a six queue.



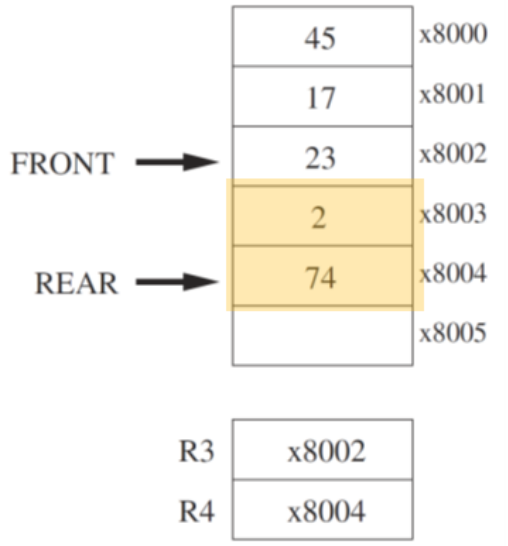
23, 2, and 74 are the current items in the queue. (45 and 17 have been removed.)

R3 is the front pointer, which is the address just before the first item.

R4 is the rear pointer, which refers to the last item inserted.

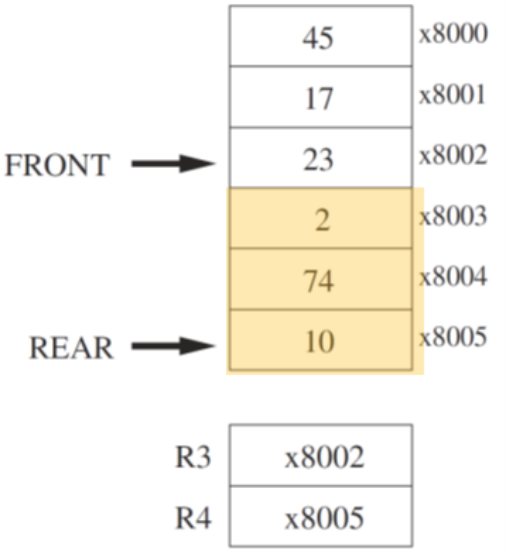
Access the text alternative for slide images.

Queue: LC-3 Implementation



To remove, we move the front pointer ahead and load the data item.

```
ADD R3,R3,#1
LDR R0,R3,#0
```



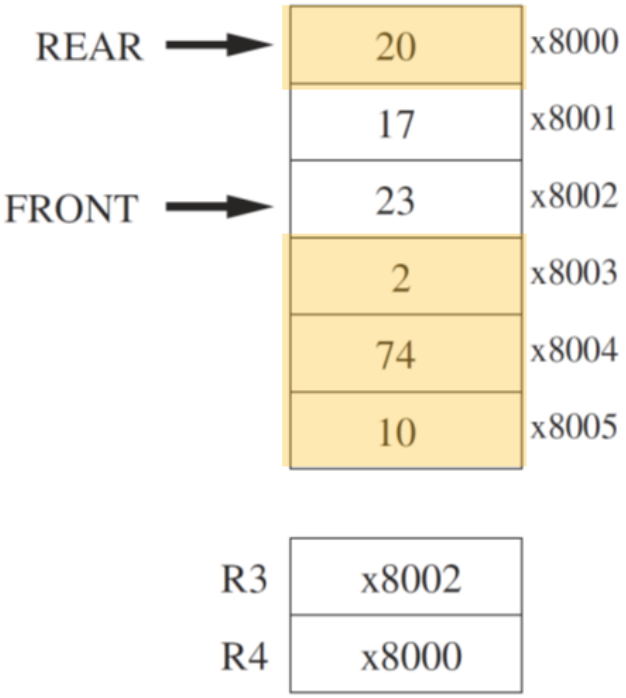
To insert, we move the rear pointer ahead and store the data item.

```
ADD R4,R4,#1
STR R0,R4,#0
```

Access the text alternative for slide images.

Wrap-Around 1

It now appears that we cannot insert more data, because we've reached the end of our allocated storage. But there are only three elements in the queue, and three unused storage locations.

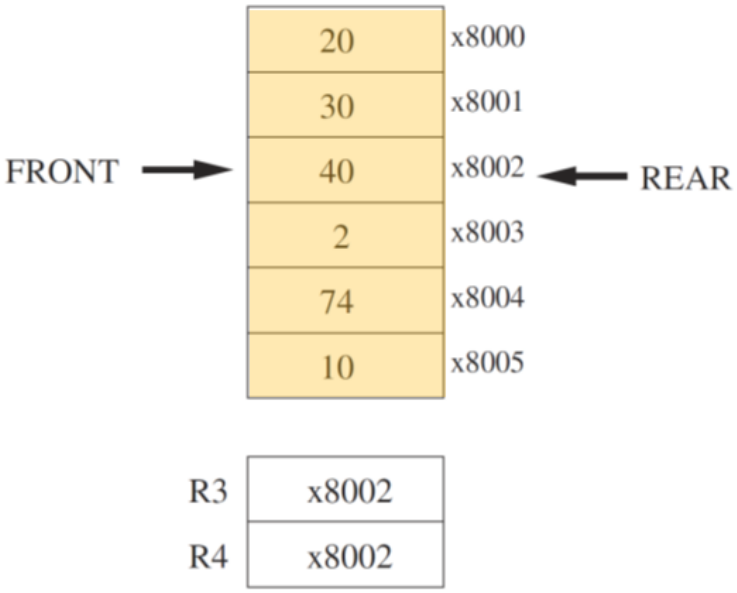


Since location x8000 is unused, we can wrap around the rear pointer and put a new element (20) in that location.

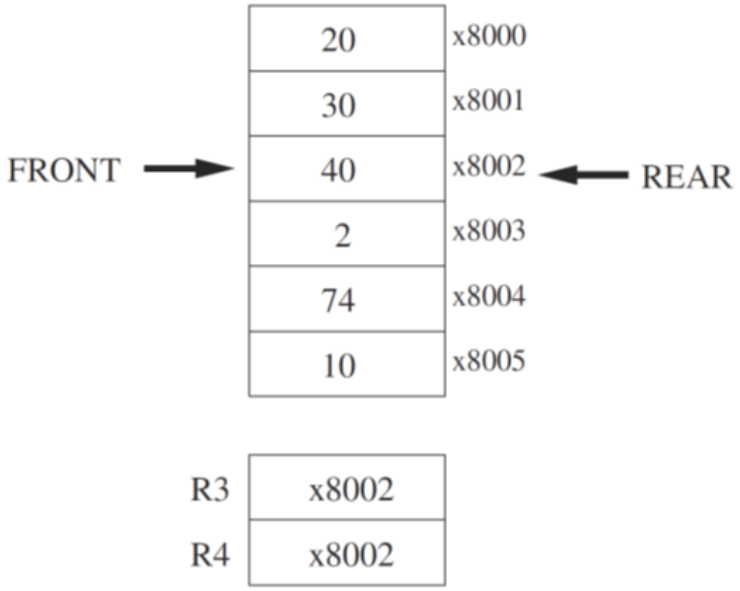
Access the text alternative for slide images.

Wrap-Around 2

If we add two more items, all of our queue slots are full, and the front and rear pointers are equal.



Suppose we now start removing items. Moving the front pointer forward and wrapping around at x8005. After six removals:

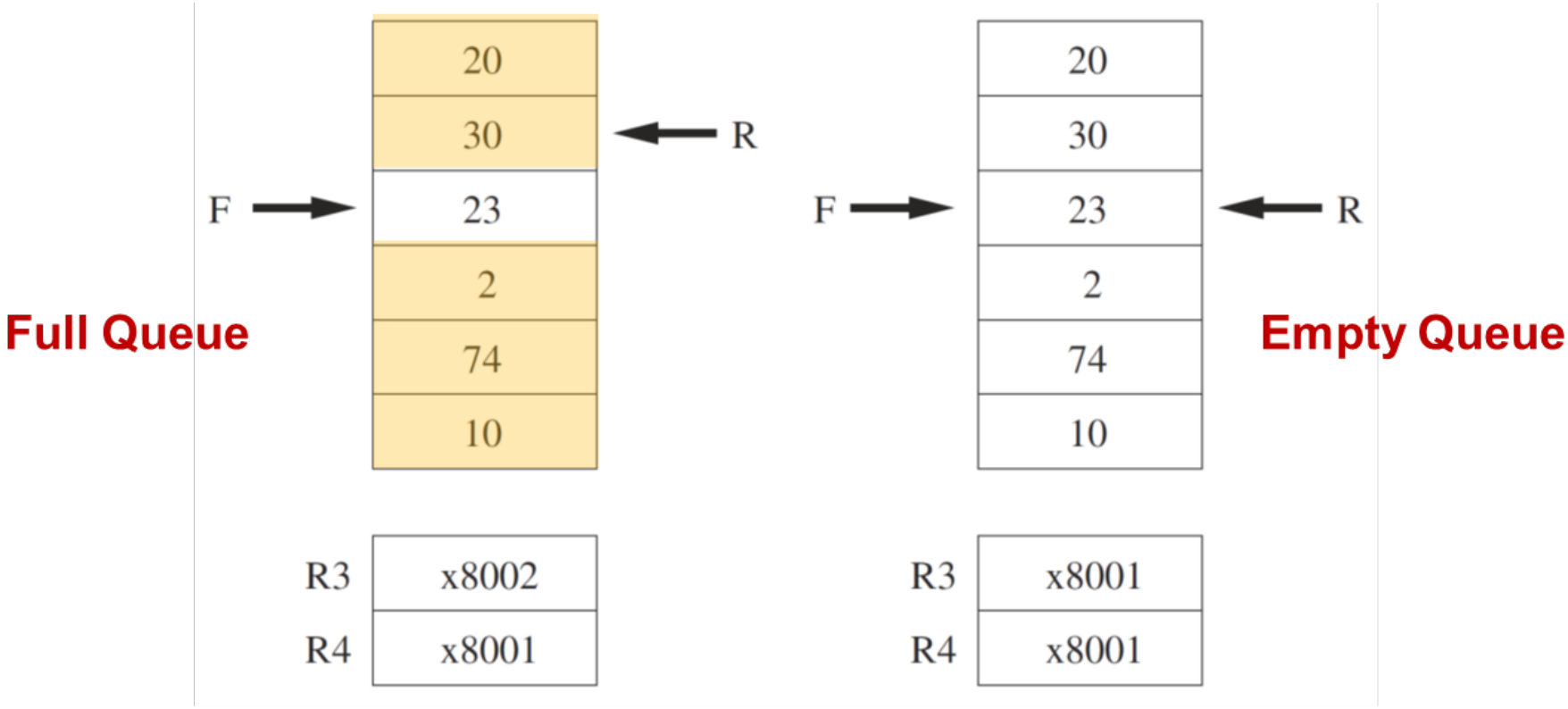


The full queue and the empty queue look exactly the same -- not good!!

Access the text alternative for slide images.

Solution to the Wrap-Around Problem

Our solution is to only store *n* − 1 items in an queue with n memory locations.



Thus, front = rear only if the queue is empty.

Access the text alternative for slide images.

Overflow and Underflow

As with the stack, we want our subroutines to recognize overflow and underflow scenarios.

- If the queue is full, an insert operation will fail.
- If the queue is empty, a remove operation will fail.

On the next slide, we show an INSERT subroutine with overflow detection. The REMOVE subroutine with underflow detection is given in Figure 8.27.

Queue Insertion with Overflow Detection

```
INSERT      ST    R1,SaveR1     ; save registers
            AND   R5,R5,#0      ; initialize for no overflow
            LD    R1,NEG_LAST
            ADD   R1,R1,R4      ; R1 = rear - x8005
            BRnp  SKIP1
            LD    R4,FIRST      ; wraparound
            BR    SKIP2         ; BR or BRnzp is unconditional branch
SKIP1       ADD   R4,R4,#1      ; if no wrap, increment rear ptr
SKIP2       NOT   R1,R4         ; compare front to rear
            ADD   R1,R1,#1
            ADD   R1,R1,R3
            BRz   FULL
            STR   R0,R4,#0      ; not full, store data
            BR    DONE
FULL        LD    R1,NEG_FIRST  ; if full, undo rear ptr move
            ADD   R1,R1,R4      ; R1 = rear - x8000
            BRnp  SKIP3
            LD    R4,LAST       ; undo wraparound
            BR    SKIP4
SKIP3       ADD   R4,R4,#-1     ; if no wrap, undo increment
SKIP4       ADD   R5,R5,#1      ; return error value
DONE        LD    R1,SaveR1
            RET
SaveR1      .BLKW 1       ; space to save reg
FIRST       .FILL x8000   ; starting address of queue storage
NEG_FIRST   .FILL x8000   ; negative of FIRST
LAST        .FILL x8005   ; ending address of queue storage
NEG_LAST    .FILL x7FFB   ; negative of LAST
```

String

A string is a sequence of characters.

- With the ASCII code, characters include letters, digits, punctuation, and other characters. (See Chapter 2.)
- A string can be of any length. The null character (ASCII x00) denotes the end of the string.

In most scenarios, we are given a **pointer** to the first character in the string. The code that processes the string proceeds until the null character is found.

LC-3 Implementation 2

Even though an ASCII character only requires 7 bits of information, we will usually store one character per (16-bit) word to simplify the code needed to process a string.

In the example below, the string `"Time flies."` is stored at location x4000.

This string has 11 characters, stored in 12 memory locations.

x4000	x0054	'T'
x4001	x0069	'i'
x4002	x006D	'm'
x4003	x0065	'e'
x4004	x0020	' '
x4005	x0066	'f'
x4006	x006C	'l'
x4007	x0069	'i'
x4008	x0065	'e'
x4009	x0073	's'
x400A	x002E	'.'
x400B	x0000	null

The .STRINGZ Directive

LC-3 Assembly Language has a directive that makes it easy to specify string data for a program.

```
.STRINGZ "Time flies."
```

Like .FILL, the directive will initialize data at the specified location. In this case, however, the number of memory locations is determined by the length of the string.

As shown on the previous slide, the directive above will allocate 12 memory locations and initialize each with a character. The null character is always in the last memory location.

The Z at the end is meant to remind you of that trailing zero.

.STRINGZ Example

```
; classic "Hello, World!" program in LC-3 assembly language

            .ORIG x3000
            LEA  R1, HELLO  ; get address of string
LOOP        LDR  R0,R1,#0   ; load character using pointer (R1)
            BRz  DONE       ; if null character, end of string
            TRAP x21        ; print the character
            ADD  R1,R1,#1   ; move ptr to next char in string
            BR   LOOP
DONE        TRAP x25        ; HALT
;
HELLO       .STRINGZ "Hello, World!\n"
            ; '\n' is a special character code that
            ; represents the linefeed character
            ; It is one character, not two.
            .END
```

Character Operations

You will need several common operations in your bag of tricks to process ASCII characters. Here are a few:

Compare two characters in "alphabetical order"

Same as comparing numerically -- subtract one from the other

Check if a character is a decimal digit

Digit characters go from x30 ('0') to x39 ('9')

If you know a character is a letter, check if it's upper-case

Look at bit 5 -- 1 if lower-case ('a' = x61), 0 if upper-case ('A' = x41)
(Look at? AND with x20. If zero, bit is zero. It nonzero, bit is 1.)

Convert letter to uppercase

AND with xFFDF. Clears bit 5.

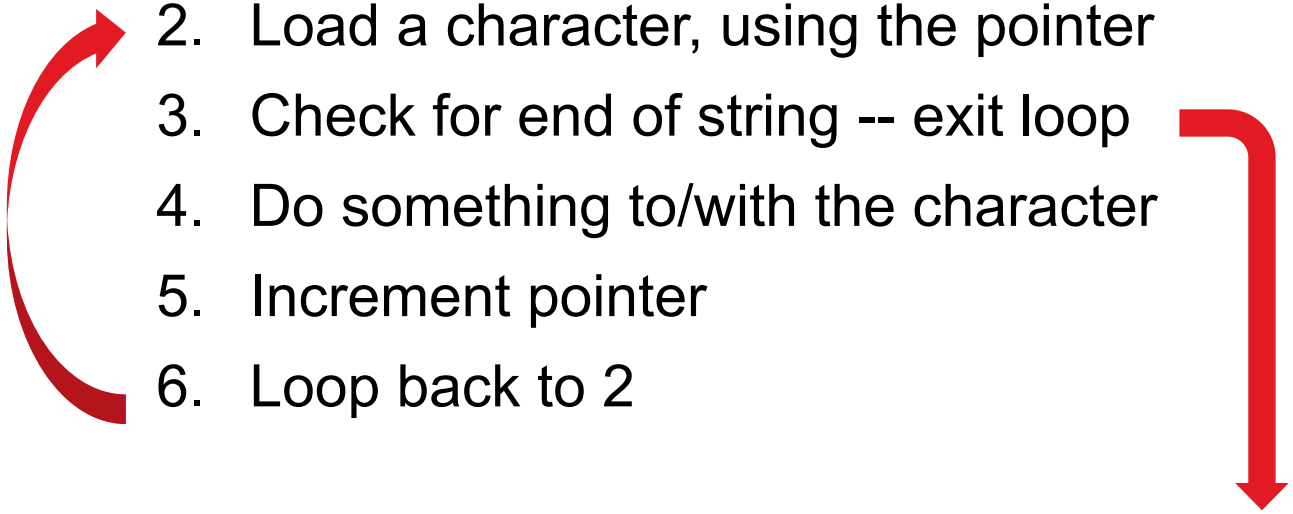
Convert from decimal digit to corresponding integer (for example, from '4' to 4)

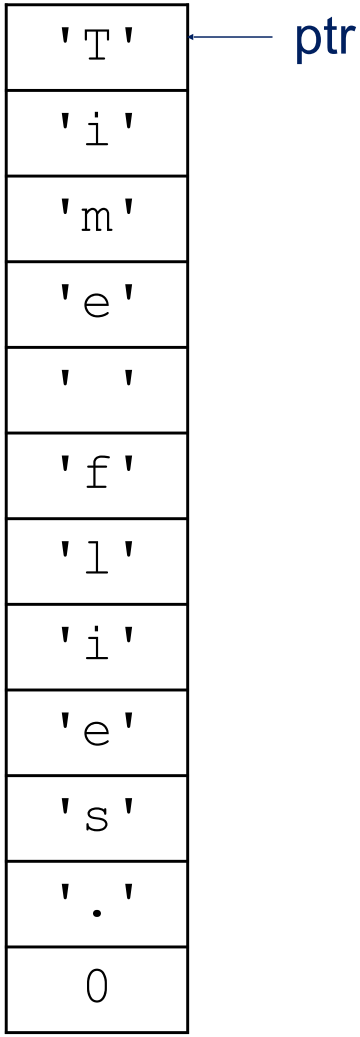
Subtract x30

String Processing

A typical pattern when processing strings

1. Point to the start of the string
2. Load a character, using the pointer
3. Check for end of string -- exit loop
4. Do something to/with the character
5. Increment pointer
6. Loop back to 2





Examples:
How many characters in the string?
How many uppercase letters?
How many digits?

String Subroutine

Problem:

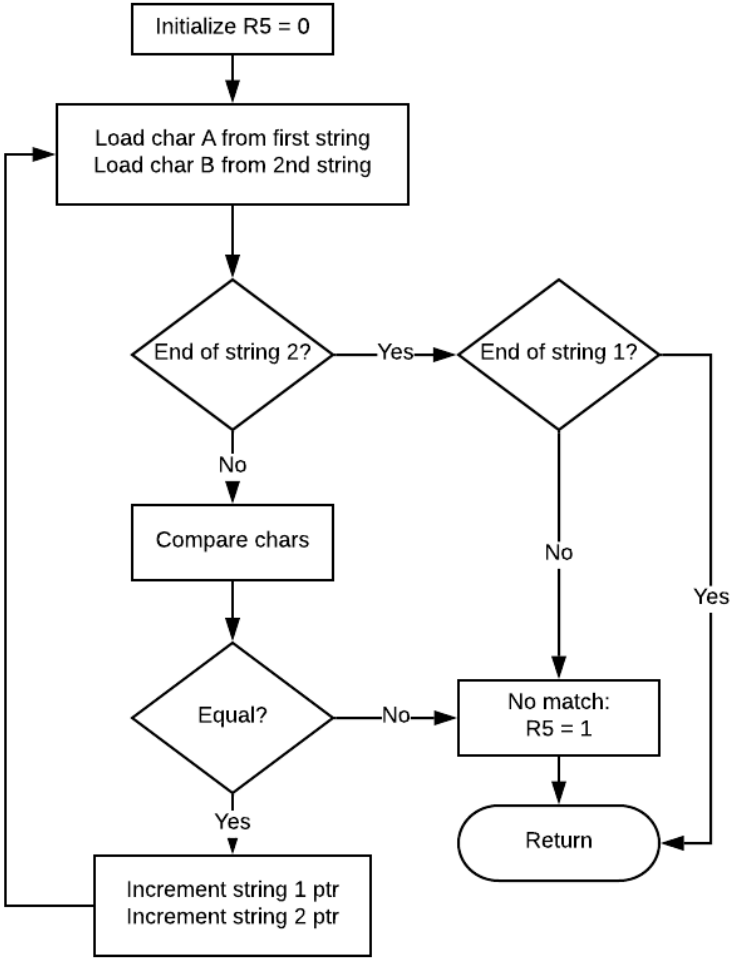
Write a subroutine to **compare two strings**.

Inputs:
R0 = address of string 1
R1 = address of string 2

Output:
R5 = 0 if strings equal, 1 otherwise

Algorithm:

Move two pointers through strings until (a) characters are not equal or (b) reach the end of one of the strings. If both strings end at the same time, then all characters match.



Access the text alternative for slide images.

String Compare Subroutine

```
STRCMP      ST     R0,SaveR0     ; save registers
            ST     R1,SaveR1
            ST     R2,SaveR2
            ST     R3,SaveR3
            AND    R5,R5,#0      ; assume a match

NEXTCHAR    LDR    R2,R0,#0      ; char from string 1
            LDR    R3,R1,#0      ; char from string 2
            BRnp   COMPARE       ; not end of string 2 -- compare
            ADD    R2,R2,#0
            BRz    DONE          ; end of both strings -- match!

COMPARE     NOT    R2,R2
            ADD    R2,R2,#1
            ADD    R2,R2,R3      ; if zero, chars are equal
            BRnp   FAIL
            ADD    R0,R0,#1      ; increment pointers to check next chars
            ADD    R1,R1,#1
            BR     NEXTCHAR

FAIL        ADD    R5,R5,#1      ; return error value
DONE        LD     R0,SaveR0
            LD     R1,SaveR1
            LD     R2,SaveR2
            LD     R3,SaveR3
            RET

SaveR0      .BLKW 1       ; space to save reg
SaveR1      .BLKW 1       ; space to save reg
SaveR2      .BLKW 1       ; space to save reg
SaveR3      .BLKW 1       ; space to save reg
```

Because learning changes everything.®

www.mheducation.com