**Contents**

##### [Preface xv](#_bookmark0)

###### [Welcome Aboard 1](#_bookmark0)

[!.! What We Will Try to Do](#_bookmark0) !

[!." How We Will Get There !](#_bookmark0)

[!.# Two Recurring Themes #](#_bookmark0)

[!.#.! The Notion of Abstraction](#_bookmark0) #

[!.#." Hardware vs. Software 5](#_bookmark0)

[!.4 A Computer System 7](#_bookmark0)

[!.4.! A (Very) Little History for a (Lot) Better Perspective ’](#_bookmark0)

[!.4." The Parts of a Computer System !(](#_bookmark0)

[!.5 Two Very Important Ideas !!](#_bookmark0)

[!.) Computers as Universal Computational Devices !"](#_bookmark0)

[!.7 How Do We Get the Electrons to Do the Work? !4](#_bookmark0)

[!.7.! The Statement of the Problem !4](#_bookmark0)

[!.7." The Algorithm !)](#_bookmark0)

[!.7.# The Program !)](#_bookmark0)

[!.7.4 The ISA !7](#_bookmark0)

[!.7.5 The Microarchitecture !’](#_bookmark0)

[!.7.) The Logic Circuit !9](#_bookmark0)

[!.7.7 The Devices !9](#_bookmark0)

[Exercises "(](#_bookmark0)

###### [Bits, Data Types, and Operations 25](#_bookmark0)

[".! Bits and Data Types "5](#_bookmark0)

[".!.! The Bit as the Unit of Information "5](#_bookmark0) [".!." Data Types ")](#_bookmark0)

["." Integer Data Types ")](#_bookmark0)

[".".! Unsigned Integers ")](#_bookmark0) ["."." Signed Integers "7](#_bookmark0)

[".# "’s Complement Integers "9](#_bookmark0)

[".4 Conversion Between Binary and Decimal #!](#_bookmark0) [".4.! Binary to Decimal Conversion #!](#_bookmark0)  [".4." Decimal to Binary Conversion #" ".4.# Extending Conversion to Numbers with](#_bookmark0)

[Fractional Parts ##](#_bookmark0)

[".5 Operations on Bits—Part I: Arithmetic #4](#_bookmark0)

[".5.! Addition and Subtraction #4](#_bookmark0)

[".5." Sign-Extension #)](#_bookmark0)

[".5.# Overflow #)](#_bookmark0)

[".) Operations on Bits—Part II: Logical Operations #’](#_bookmark0)

[".).! A Logical Variable #’](#_bookmark0) [".)." The AND Function #](#_bookmark0)’ [".).# The OR Function #9](#_bookmark0) [".).4 The NOT Function 4(](#_bookmark0)

[".).5 The Exclusive-OR Function 4(](#_bookmark0) [".).) DeMorgan’s Laws 4!](#_bookmark0)

[".).7 The Bit Vector 4"](#_bookmark0)

[".7 Other Representations 4](#_bookmark0)#

[".7.! Floating Point Data Type (Greater Range](#_bookmark0), [Less Precision) 4#](#_bookmark0)

[".7." ASCII Codes 47](#_bookmark0)

[".7.# Hexadecimal Notation 4’](#_bookmark0)

[Exercises 49](#_bookmark0)

###### [Digital Logic Structures 59](#_bookmark0)

[#.! The Transistor 59](#_bookmark0) [#." Logic Gates )!](#_bookmark0)

[#.".! The NOT Gate (Inverter) )!](#_bookmark0) [#."." OR and NOR Gates )"](#_bookmark0)

[#.".# Why We Can’t Simply Connect P-Type to Ground )4](#_bookmark0)

[#.".4 AND and NAND Gates )](#_bookmark0)5

[#.".5 Gates with More Than Two Inputs ))](#_bookmark0)

[#.# Combinational Logic Circuits )7](#_bookmark0)

[#.#.! Decoder )7](#_bookmark0) [#.#." Mux )’](#_bookmark0)

[#.#.# A One-Bit Adder (a.k.a. a Full Adder) )9](#_bookmark0)



[!.!." The Programmable Logic Array (PLA) #1](#_bookmark0)

[!.!.5 Logical Completeness #2](#_bookmark0)

[!." Basic Storage Elements #!](#_bookmark0)

[!.".1 The R-S Latch #!](#_bookmark0)

[!.".2 The Gated D Latch #"](#_bookmark0)

[!.5 The Concept of Memory #5](#_bookmark0)

[!.5.1 Address Space #5](#_bookmark0)

[!.5.2 Addressability #’](#_bookmark0)

[!.5.! A 22-by-!-Bit Memory #’](#_bookmark0)

[!.’ Sequential Logic Circuits #(](#_bookmark0)

[!.’.1 A Simple Example: The Combination Lock #)](#_bookmark0)

[!.’.2 The Concept of State (0](#_bookmark0)

[!.’.! The Finite State Machine and Its State Diagram (2](#_bookmark0)

[!.’." The Synchronous Finite State Machine (5](#_bookmark0)

[!.’.5 The Clock (’](#_bookmark0)

[!.’.’ Example: A Danger Sign (#](#_bookmark0)

[!.# Preview of Coming Attractions: The Data Path of the LC-! )!](#_bookmark0)

[Exercises )5](#_bookmark0)

###### [The von Neumann Model 121](#_bookmark0)

[".1 Basic Components 121](#_bookmark0)

[".1.1 Memory 122](#_bookmark0)

[".1.2 Processing Unit 12!](#_bookmark0)

[".1.! Input and Output 12"](#_bookmark0)

[".1." Control Unit 125](#_bookmark0)

[".2 The LC-!: An Example von Neumann Machine 125](#_bookmark0)

[".! Instruction Processing 12#](#_bookmark0)

[".!.1 The Instruction 12#](#_bookmark0)

[".!.2 The Instruction Cycle (NOT the Clock Cycle!) 1!0](#_bookmark0)

[".!.! Changing the Sequence of Execution 1!2](#_bookmark0)

[".!." Control of the Instruction Cycle 1!"](#_bookmark0) [".!.5 Halting the Computer (the TRAP](#_bookmark0)

[Instruction) 1!’](#_bookmark0)

["." Our First Program: A Multiplication Algorithm 1!#](#_bookmark0)

[Exercises 1!)](#_bookmark0)

###### [The LC-3 145](#_bookmark0)

* 1. [The ISA: Overview 1"5](#_bookmark0)
     1. [Memory Organization 1"’](#_bookmark0)
     2. [Registers 1"’](#_bookmark0)

[5.1.! The Instruction Set 1"#](#_bookmark0)

[5.1." Opcodes 1")](#_bookmark0)

[5.1.5 Data Types 1")](#_bookmark0)

[5.1.’ Addressing Modes 150](#_bookmark0)

* 1. [# Condition Codes 150](#_bookmark0)
  2. [Operate Instructions 151](#_bookmark0)
     1. [ADD, AND, and NOT 151](#_bookmark0)
     2. [Immediates 152](#_bookmark0)
  3. [! The LEA Instruction (Although Not Really an Operate) 15"](#_bookmark0)

[5.! Data Movement Instructions 155](#_bookmark0)

[5.!.1 PC-Relative Mode 15’](#_bookmark0)

[5.!.2 Indirect Mode 15(](#_bookmark0)

[5.!.! Base+o+set Mode 15)](#_bookmark0)

[5.!." An Example 1’0](#_bookmark0)

[5." Control Instructions 1’1](#_bookmark0)

[5.".1 Conditional Branches 1’2](#_bookmark0)

[5.".2 Two Methods of Loop Control 1’5 5.".! The JMP Instruction 1’)](#_bookmark0)

[5."." The TRAP Instruction 1’)](#_bookmark0)

[5.5 Another Example: Counting Occurrences of a Character 1#0](#_bookmark0)

[5.’ The Data Path Revisited 1#!](#_bookmark0)

[5.’.1 Basic Components of the Data Path 1#5](#_bookmark0)

[5.’.2 The Instruction Cycle Specific to the LC-! 1#’](#_bookmark0)

[Exercises 1##](#_bookmark0)

###### [Programming 2’3](#_bookmark1)

[’.1 Problem Solving 20!](#_bookmark1)

[’.1.1 Systematic Decomposition 20!](#_bookmark1) [’.1.2 The Three Constructs: Sequential,](#_bookmark2)

[Conditional, Iterative 20"](#_bookmark2)

[’.1.! LC-! Control Instructions to Implement the Three Constructs 205](#_bookmark3)

[’.1." The Character Count Example from Chapter 5, Revisited 20’](#_bookmark4)

[’.2 Debugging 210](#_bookmark5)

[’.2.1 Debugging Operations 211](#_bookmark6)

[’.2.2 Use of an Interactive Debugger 212](#_bookmark7)

[Exercises 220](#_bookmark8)

###### [( Assembly Language 231](#_bookmark9)

[#.1 Assembly Language Programming—Moving Up a Level 2!1](#_bookmark9)

[#.2 An Assembly Language Program 2!2](#_bookmark10)

[#.2.1 Instructions 2!!](#_bookmark11)

[#.2.2 Pseudo-Ops (Assembler Directives) 2!’](#_bookmark12)

[!.".# Example: The Character Count Example of Section 5.5, Revisited Again! "#8](#_bookmark13)

[!.# The Assembly Process "4’](#_bookmark14)

[!.#.( Introduction "4’](#_bookmark14)

[!.#." A Two-Pass Process "4’](#_bookmark14)

[!.#.# The First Pass: Creating the Symbol Table "4(](#_bookmark15)

[!.#.4 The Second Pass: Generating the Machine Language Program "4"](#_bookmark16)

[!.4 Beyond the Assembly of a Single Assembly Language Program "4#](#_bookmark17)

[!.4.( The Executable Image "44](#_bookmark18)

[!.4." More than One Object File "44](#_bookmark18)

[Exercises "45](#_bookmark19)

###### [8 Data Structures 263](#_bookmark20)

[8.( Subroutines ")#](#_bookmark20)

[8.(.( The Call/Return Mechanism ")5](#_bookmark21) [8.(." JSR(R)—The Instruction That Calls](#_bookmark22)

[the Subroutine "))](#_bookmark22)

[8.(.# Saving and Restoring Registers ")!](#_bookmark23) [8.(.4 Library Routines ")9](#_bookmark24)

[8." The Stack "!#](#_bookmark25)

[8.".( The Stack—An Abstract Data Type "!#](#_bookmark25) [8."." Two Example Implementations "!#](#_bookmark25) [8.".# Implementation in Memory "!4](#_bookmark26)

[8.".4 The Complete Picture "!8](#_bookmark27)

[8.# Recursion, a Powerful Technique When Used Appropriately "8’](#_bookmark28)

[8.#.( Bad Example Number (: Factorial "8’](#_bookmark28) [8.#." Fibonacci, an Even Worse](#_bookmark29)

[Example "85](#_bookmark29)

1. [#.# The Maze, a Good Example "88](#_bookmark30)
   1. [The Queue "94](#_bookmark31)

[8.4.( The Basic Operations: Remove from Front, Insert at Rear "95](#_bookmark32)

[8.4." Wrap-Around "95](#_bookmark32)

[8.4.# How Many Elements Can We Store in a Queue? "9)](#_bookmark33)

* + 1. [Tests for Underflow, Overflow "9!](#_bookmark34)
    2. [The Complete Story "98](#_bookmark35)
  1. [Character Strings #’’](#_bookmark36)

[Exercises #’4](#_bookmark37)

###### [9 I/O 313](#_bookmark38)

[9.( Privilege, Priority, and the Memory Address Space #(4](#_bookmark39)

[9.(.( Privilege and Priority #(4](#_bookmark39) [9.(." Organization of Memory #()](#_bookmark40)

Contents **ix**

1. [" Input/Output #(!](#_bookmark41)

[9.".( Some Basic Characteristics of I/O #(!](#_bookmark41)

[9."." Input from the Keyboard #"’](#_bookmark42) [9.".# Output to the Monitor #""](#_bookmark43) [9.".4 A More Sophisticated Input](#_bookmark44)

[Routine #"5](#_bookmark44)

[9.".5 Implementation of Memory-Mapped I/O, Revisited #")](#_bookmark45)

[9.# Operating System Service Routines (LC-# Trap Routines) #"!](#_bookmark46)

[9.#.( Introduction #"!](#_bookmark46)

[9.#." The Trap Mechanism #"9](#_bookmark47) [9.#.# The TRAP Instruction ##’](#_bookmark48) [9.#.4 The RTI Instruction: To Return](#_bookmark49)

[Control to the Calling Program ##(](#_bookmark49)

[9.#.5 A Summary of the Trap Service Routine Process ##(](#_bookmark49)

[9.#.) Trap Routines for Handling I/O ###](#_bookmark50) [9.#.! A Trap Routine for Halting](#_bookmark51)

[the Computer ##5](#_bookmark51)

[9.#.8 The Trap Routine for Character Input (One Last Time) ##)](#_bookmark52)

[9.#.9 PUTS: Writing a Character String to the Monitor ##8](#_bookmark53)

* 1. [Interrupts and Interrupt-Driven I/O ##9 9.4.( What Is Interrupt-Driven I/O? ##9](#_bookmark54) [9.4." Why Have Interrupt-Driven I/O? #4’](#_bookmark55) [9.4.# Two Parts to the Process #4(](#_bookmark56)
     1. [Part I: Causing the Interrupt to Occur #4(](#_bookmark56)
     2. [Part II: Handling the Interrupt Request #44](#_bookmark57)

[9.4.) An Example #4!](#_bookmark58)

* 1. [! Not Just I/O Devices #49](#_bookmark59)
  2. [Polling Revisited, Now That We Know About Interrupts #5’](#_bookmark60)

[9.5.( The Problem #5’](#_bookmark60)

* 1. [" The Solution #5(](#_bookmark61)

[Exercises #5"](#_bookmark62)

###### [1’ A Calculator 3(9](#_bookmark63)

[(’.( Data Type Conversion #8’](#_bookmark64)

[(’.(.( Example: A Bogus Program:](#_bookmark65)

[" + # = e #8(](#_bookmark65)

[(’.(." Input Data (ASCII to Binary) #8(](#_bookmark65) [(’.(.# Display Result (Binary](#_bookmark66)

[to ASCII) #85](#_bookmark66)

[!".# Arithmetic Using a Stack 387](#_bookmark67)

[!".#.! The Stack as Temporary Storage 387](#_bookmark67)

[!".#.# An Example 388](#_bookmark68)

[!".#.3 OpAdd, OpMult, and OpNeg 38’](#_bookmark69)

[!".3 The Calculator 3’(](#_bookmark70)

[!".3.! Functionality 3’(](#_bookmark70)

[!".3.# Code 3’)](#_bookmark71)

[Exercises 4"#](#_bookmark72)

###### [Introduction to C/C++ Programming 405](#_bookmark0)

[!!.! Our Objective 4"(](#_bookmark0)

[!!.# Bridging the Gap 4")](#_bookmark0)

[!!.3 Translating High-Level Language Programs 4!"](#_bookmark0)

[!!.3.! Interpretation 4!"](#_bookmark0)

[!!.3.# Compilation 4!!](#_bookmark0)

[!!.3.3 Pros and Cons 4!!](#_bookmark0)

[!!.4 The C/C++ Programming Languages 4!!](#_bookmark0)

[!!.4.! The Origins of C and C++ 4!!](#_bookmark0)

[!!.4.# How We Will Approach C and C++ 4!#](#_bookmark0)

[!!.4.3 The Compilation Process 4!3](#_bookmark0)

[!!.4.4 Software Development](#_bookmark0)

[Environments 4!(](#_bookmark0)

[!!.( A Simple Example in C 4!(](#_bookmark0)

[!!.(.! The Function main 4!(](#_bookmark0)

[!!.(.# Formatting, Comments, and Style 4!7](#_bookmark0)

[!!.(.3 The C Preprocessor 4!8](#_bookmark0)

[!!.(.4 Input and Output 4!’](#_bookmark0)

[!!.) Summary 4##](#_bookmark0)

[Exercises 4##](#_bookmark0)

###### [Variables and Operators 425](#_bookmark0)

[!#.! Introduction 4#(](#_bookmark0)

[!#.# Variables 4#(](#_bookmark0)

[!#.#.! Four Basic Data Types 4#)](#_bookmark0)

[!#.#.# Choosing Identifiers 4#’](#_bookmark0)

[!#.#.3 Scope: Local vs. Global 4#’](#_bookmark0)

[!#.#.4 More Examples 43!](#_bookmark0)

[!#.3 Operators 43#](#_bookmark0)

[!#.3.! Expressions and Statements 433](#_bookmark0)

[!#.3.# The Assignment Operator 433](#_bookmark0)

[!#.3.3 Arithmetic Operators 434](#_bookmark0)

[!#.3.4 Order of Evaluation 43(](#_bookmark0)

[!#.3.( Bitwise Operators 43)](#_bookmark0)

[!#.3.) Relational Operators 437](#_bookmark0)

[!#.3.7 Logical Operators 438](#_bookmark0)

[!#.3.8 Increment /Decrement Operators 43’](#_bookmark0)

[!#.3.’ Expressions with Multiple Operators 44!](#_bookmark0)

[!#.4 Problem Solving Using Operators 44!](#_bookmark0)

[!#.( Tying It All Together 444](#_bookmark0)

[!#.(.! Symbol Table 444](#_bookmark0)

[!#.(.# Allocating Space for Variables 44(](#_bookmark0)

[!#.(.3 A Comprehensive Example 447](#_bookmark0)

[!#.) Additional Topics 44’](#_bookmark0)

[!#.).! Variations of the Basic Types 4("](#_bookmark0)

[!#.).# Literals, Constants, and Symbolic Values 4(!](#_bookmark0)

[!#.).3 Additional C Operators 4(#](#_bookmark0)

[!#.7 Summary 4(3](#_bookmark0)

[Exercises 4(3](#_bookmark0)

###### [Control Structures 45’](#_bookmark0)

[!3.! Introduction 4(7](#_bookmark0)

[!3.# Conditional Constructs 4(7](#_bookmark0)

[!3.#.! The if Statement 4(](#_bookmark0)8

[!3.#.# The if-else Statement 4)"](#_bookmark0)

[!3.3 Iteration Constructs 4)4](#_bookmark0)

[!3.3.! The while Statement 4)4](#_bookmark0)

[!3.3.# The for Statement 4))](#_bookmark0)

[!3.3.3 The do-while Statement 47!](#_bookmark0)

[!3.4 Problem Solving Using Control Structures 47#](#_bookmark0)

[!3.4.! Problem !: Approximating the Value of 兀 47#](#_bookmark0)

[!3.4.# Problem #: Finding Prime Numbers Less Than !"" 474](#_bookmark0)

[!3.4.3 Problem 3: Analyzing an E-mail](#_bookmark0)

[Address 477](#_bookmark0)

[!3.( Additional C Control Structures 48"](#_bookmark0)

[!3.(.! The switch Statement 48"](#_bookmark0)

[!3.(.# The break and continue](#_bookmark0)

[Statements 48#](#_bookmark0)

[!3.(.3 An Example: Simple Calculator 48#](#_bookmark0)

[!3.) Summary 484](#_bookmark0)

[Exercises 484](#_bookmark0)

###### [Functions 4(1](#_bookmark0)

[!4.! Introduction 4’!](#_bookmark0)

[!4.# Functions in C 4’#](#_bookmark0)

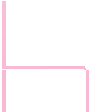
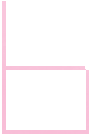
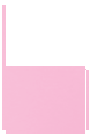
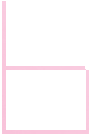
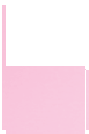
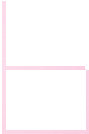
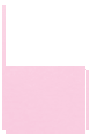
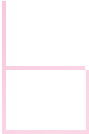
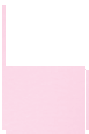
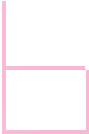
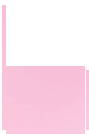
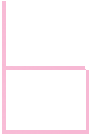
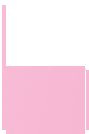
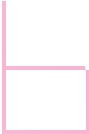
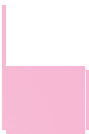
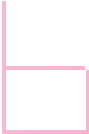
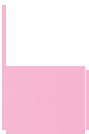
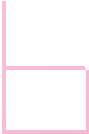
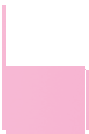
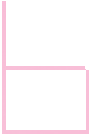
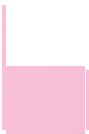
[!4.#.! A Function with a Parameter 4’#](#_bookmark0)

[!4.#.# Example: Area of a Ring 4’(](#_bookmark0)

### CHAPTER

6

Programming



e are now ready to develop programs to solve problems with the com- puter. In this chapter we attempt to do two things: first, we develop a methodology for constructing programs to solve problems (Section 6.1, Problem Solving), and second, we develop a methodology for fixing those programs (Sec- tion 6.2, Debugging) under the likely condition that we did not get everything

**W**

right the first time.

There is a long tradition that the errors present in programs are referred to as *bugs*, and the process of removing those errors is called *debugging*. The opportu- nities for introducing bugs into a complicated program are so great that it usually takes much more time to get the program to work correctly (debugging) than it does to create the program in the first place.

## !.1 Problem Solving

##### !.1.1 Systematic Decomposition

Recall from Chapter 1 that in order for electrons to solve a problem, we need to go through several levels of transformation to get from a natural language description of the problem (in our case English, although many of you might prefer Italian, Mandarin, Hindi, or something else) to something electrons can deal with. Once we have a natural language description of the problem, the next step is to trans- form the problem statement into an algorithm. That is, the next step is to transform the problem statement into a step-by-step procedure that has the properties of def- initeness (each step is precisely stated), effective computability (each step can be carried out by a computer), and finiteness (the procedure terminates).

In the late 1960s, the concept of *structured programming* emerged as a way to dramatically improve the ability of average programmers to take a complex

description of a problem and systematically decompose it into smaller and smaller manageable units so that they could ultimately write a program that executed cor- rectly. The methodology has also been called *systematic decomposition* because the larger tasks are systematically broken down into smaller ones.

We will find the systematic decomposition model a useful technique for designing computer programs to carry out complex tasks.

##### !.1.# The Three Constructs: Sequential, Conditional, Iterative

Systematic decomposition is the process of taking a task, that is, a unit of work (see Figure 6.1a), and breaking it into smaller units of work such that the collec- tion of smaller units carries out the same task as the one larger unit. The idea is that if one starts with a large, complex task and applies this process again and again, one will end up with very small units of work and consequently be able to easily write a program to carry out each of these small units of work. The process is also referred to as *stepwise refinement,* because the process is applied one step at a time, and each step refines one of the tasks that is still too complex into a collection of simpler subtasks.

The idea is to replace each larger unit of work with a construct that correctly decomposes it. There are basically three constructs for doing this: *sequential, conditional,* and *iterative*.

The **sequential** construct (Figure 6.1b) is the one to use if the designated task can be broken down into two subtasks, one following the other. That is, the computer is to carry out the first subtask completely, *then* go on and carry out the second subtask completely— *never* going back to the first subtask after starting the second subtask.

The **conditional** construct (Figure 6.1c) is the one to use if the task consists of doing one of two subtasks but not both, depending on some condition. If the

(a)

The task to be

decomposed

(b) Sequential

True Test False

(c) Conditional

(d) Iterative

|  |  |
| --- | --- |
|  |  |
| Do first part to completion | |
|  |  |
| Do second part to completion | |
|  |  |

Subtask 2

Subtask 1

cond.

Test False cond.

True

Subtask

**Figure !.1 The basic constructs of structured programming.**

condition is true, the computer is to carry out one subtask. If the condition is not true, the computer is to carry out a different subtask. Either subtask may be vacuous; that is, it may *do nothing*. Regardless, after the correct subtask is completed, the program moves onward. The program never goes back and retests the condition.

The **iterative** construct (Figure 6.1d) is the one to use if the task consists of doing a subtask a number of times, but only as long as some condition is true. If the condition is true, do the subtask. After the subtask is finished, go back and test the condition again. As long as the result of the condition tested is true, the program continues to carry out the same subtask again and again. The first time the test is not true, the program proceeds onward.

Note in Figure 6.1 that whatever the task of Figure 6.1a, work starts with the arrow into the top of the ‘‘box’’ representing the task and finishes with the arrow out of the bottom of the box. There is no mention of what goes on *inside* the box. In each of the three possible decompositions of Figure 6.1a (i.e., Figure 6.1b, c, and d), there is exactly *one entrance into the construct* and exactly *one exit out of the construct*. Thus, it is easy to replace any task of the form of Figure 6.1a with whichever of its three decompositions apply. We will see how with several examples.

##### !.1.$ LC-$ Control Instructions to Implement the Three Constructs

Before we move on to an example, we illustrate in Figure 6.2 the use of LC-3 control instructions to direct the program counter to carry out each of the three

(a)

A

The task to be

decomposed

A B1

B1 +1 D1

(b)

(c)

A

Second subtask

First subtask

|  |  |  |
| --- | --- | --- |
| Generate condition | | |
| 0000 | ? | *x* |
| Subtask 2 | | |
| 0000 | 111 | *y* |
| Subtask 1 | | |

B2

C2

(d)

A

|  |  |  |
| --- | --- | --- |
| Generate condition | | |
| 0000 | ? | *z* |
| Subtask | | |
| 0000 | 111 | *w* |

B3

D3

D2

**Figure !.# Use of LC-$ control instructions to implement structured programming.**

decomposition constructs. That is, Figure 6.2b, c, and d corresponds respectively to the three constructs shown in Figure 6.1b, c, and d.

We use the letters A, B, C, and D to represent addresses in memory containing LC-3 instructions. The letter A, for example, represents the address of the first LC-3 instruction to be executed in all three cases, since it is the starting address of the task to be decomposed (shown in Figure 6.2a).

Figure 6.2b illustrates the control flow of the sequential decomposition. Note that no control instructions are needed since the PC is incremented from Address B1 to Address B1+1. The program continues to execute instructions through address D1. It does not return to the first subtask.

Figure 6.2c illustrates the control flow of the conditional decomposition. First, a condition is generated, resulting in the setting of one of the condition codes. This condition is tested by the conditional branch instruction at Address B2. If the condition is true, the PC is set to Address C2+1, and subtask 1 is executed. (*Note: x* equals 1 + the number of instructions in subtask 2.) If the condition is false, the PC (which had been incremented during the FETCH phase of the branch instruction) fetches the instruction at Address B2+1, and subtask 2 is executed. Subtask 2 terminates in a branch instruction that at address C2 unconditionally branches to D2+1. (*Note: y* equals the number of instructions in subtask 1.)

Figure 6.2d illustrates the control flow of the iterative decomposition. As in the case of the conditional construct, first a condition is generated, a condi- tion code is set, and a conditional branch instruction is executed. In this case, the condition bits of the instruction at address B3 are set to cause a conditional branch if the condition generated is false. If the condition is false, the PC is set to address D3+1. (*Note: z* equals 1 + the number of instructions in the subtask in Figure 6.2d.) On the other hand, as long as the condition is true, the PC will be incremented to B3+1, and the subtask will be executed. The subtask termi- nates in an unconditional branch instruction at address D3, which sets the PC to A to again generate and test the condition. (*Note: w* equals the total number of instructions in the decomposition shown as Figure 6.2d.)

Now, we are ready to move on to an example.

##### !.1.4 The Character Count Example from Chapter &, Revisited

Recall the example of Section 5.5. The statement of the problem is as follows: “We wish to input a character from the keyboard, count the number of occurrences of that character in a file, and display that count on the monitor.”

The systematic decomposition of this English language statement of the problem to the final LC-3 implementation is shown in Figure 6.3. Figure 6.3a is a brief statement of the problem.

In order to solve the problem, it is always a good idea first to examine exactly what is being asked for, and what is available to help solve the problem. In this case, the statement of the problem says that we will get the character of inter- est from the keyboard, and that we must examine all the characters in a file and determine how many are identical to the character obtained from the keyboard. Finally, we must output the result.

* 1. (b)

Start

A

B

C

Stop

**Figure !.3 Stepwise refinement of the character count program (Fig. !.3 continued on next page.)**

To do this, we will need to examine in turn all the characters in a file, we will need to compare each to the character we input from the keyboard, and we will need a counter to increment each time we get a match.

Start

Stop

Input a character. Then scan a file, counting occurrences of that character. Finally, display on the monitor the number of occurrences of that character (up to 9).

|  |  |
| --- | --- |
|  |  |
| Initialize: Put initial values into all locations that will be needed to carry out this task.   * Input a character. * Set up the pointer to the first location in the file that will be scanned. * Get the first character from the file. * Zero the register that holds the count. | |
|  |  |
| Scan the file, location by location, incrementing the counter if the character matches. | |
|  |  |
| Display the count on the monitor. | |
|  |  |

We will need registers to hold all these pieces of information:

1. The character input from the keyboard.
2. Where we are (a pointer) in our scan of the file.
3. The character in the file that is currently being examined.
4. The count of the number of occurrences.

We will also need to know when we have reached the end of the file.

(c) (d)

B

Start

A! A2 A3

A4

Yes

Done

?

B!

No

B2

B3

C

C!

C2

Stop

Display output

Prepare to display

Get next character

Test character. If “match,” increment counter.

R! <- first char in the file

R3 <- address of first char in the file

R0 <- input from the keyboard

R2 <- 0 (count)

Start

A

Yes

Done

?

B!

No

Test character. If “match,” increment counter. Get the next character.

R2 <- 0 (count)

R0 <- input from the keyboard

R3 <- address of first char in the file R! <- first char in the file

C

Stop

Display the count on the monitor

**Figure !.3 Stepwise refinement of the character count program (Fig. !.3 continued on next page.)**

The problem decomposes naturally (using the sequential construct) into three parts as shown in Figure 6.3b: (A) initialization, which includes keyboard input of the character to be “counted,” (B) the process of determining how many occur- rences of the character are present in the file, and (C) displaying the count on the monitor.

We have seen the importance of proper initialization in several examples already. Before a computer program can get to the crux of the problem, it must have the correct initial values. These initial values do not just show up in the GPRs by magic. They get there as a result of the first set of steps in every algorithm: the initialization of its variables.

In this particular algorithm, initialization (as we said in Chapter 5) consists of starting the counter at 0, setting the pointer to the address of the first character in the file to be examined, getting an input character from the keyboard, and get- ting the first character from the file. Collectively, these four steps comprise the initialization of the algorithm shown in Figure 6.3b as A.

(e)

Start

A1 A2 A3

A4

Yes

Done

?

No

B2

Yes

R1 = R0

?

No

B3

C1 Form ASCII output

C2 Display output

Stop

R1 <– next character

R3 <– R3 + 1

R2 <– R2 + 1

R1 <– first character

R3 <– starting address

R0 <– input

R2 <– 0 (count)

**Figure !.3 Stepwise refinement of the character count program (continued Fig. !.3**

**from previous page.)**

Figure 6.3c decomposes B into an iteration construct, such that as long as there are characters in the file to examine, the loop iterates. B1 shows what gets accomplished in each iteration. The character is tested and the count incremented if there is a match. Then the next character is prepared for examination. Recall from Chapter 5 that there are two basic techniques for controlling the number

of iterations of a loop: the sentinel method and the use of a counter. Since we are unlikely to know how many characters there are in a random file, and since each file ends with an end of text (EOT) character, our choice is obvious. We use the sentinel method, that is, testing each character to see if we are examining a character in the file or the EOT character.

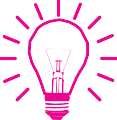
Figure 6.3c also shows the initialization step in greater detail. Four LC-3 registers (R0, R1, R2, and R3) have been specified to handle the four requirements of the algorithm: the input character from the keyboard, the current character being tested, the counter, and the pointer to the next character to be tested.

Figure 6.3d decomposes both B1 and C using the sequential construct in both cases. In the case of B1, first the current character is tested (B2), and the counter incremented if we have a match, and then the next character is fetched (B3). In the case of C, first the count is prepared for display by converting it from a 2’s complement integer to an ASCII code (C1), and then the actual character output is performed (C2).

Finally, Figure 6.3e completes the decomposition, replacing B2 with the ele- ments of the condition construct and B3 with the sequential construct (first the pointer is incremented, and then the next character to be scanned is loaded).

The last step (and usually the easiest part) is to write the LC-3 code corre- sponding to each box in Figure 6.3e. Note that Figure 6.3e is essentially identical to Figure 5.16 of Chapter 5 (except now you know where it all came from!).

Before leaving this topic, it is worth pointing out that it is not always pos- sible to understand everything at the outset. When you find that to be the case, it is not a signal simply to throw up your hands and quit. In such cases (which realistically are most cases), you should see if you can make sense of a piece of the problem and expand from there. Problems are like puzzles; initially they can be opaque, but the more you work at it, the more they yield under your attack. Once you do understand what is given, what is being asked for, and how to pro- ceed, you are ready to return to square one (Figure 6.3a) and restart the process of systematically decomposing the problem.



## !.2 Debugging

Debugging a program is pretty much applied common sense. A simple example comes to mind: You are driving to a place you have never visited, and somewhere along the way you made a wrong turn. What do you do now? One common “driv- ing debugging” technique is to wander aimlessly, hoping to find your way back. When that does not work, and you are finally willing to listen to the person sitting next to you, you turn around and return to some “known” position on the route. Then, using a map (very difficult for some people), you follow the directions pro- vided, periodically comparing where you are (from landmarks you see out the window) with where the map says you should be, until you reach your desired destination.

Debugging is somewhat like that. A logical error in a program can make take a wrong turn. The simplest way to keep track of where you are as

you

compared to where you want to be is to *trace* the program. This consists of keeping track of the **sequence** of instructions that have been executed and the **results** produced by each instruction executed. When you examine the sequence of instructions executed, you can detect errors in the flow of the program. When you compare what each instruction has done to what it is supposed to do, you can detect logical errors in the program. In short, when the behavior of the pro- gram as it is executing is different from what it should be doing, you know there is a bug.

A useful technique is to partition the program into parts, often referred to as *modules*, and examine the results that have been computed at the end of execu- tion of each module. In fact, the structured programming approach discussed in Section 6.1 can help you determine where in the program’s execution you should examine results. This allows you to systematically get to the point where you are focusing your attention on the instruction or instructions that are causing the problem.

##### !.#.1 Debugging Operations

Many sophisticated debugging tools are offered in the marketplace, and undoubtedly you will use many of them in the years ahead. In Chapter 15, for example, we will examine debugging techniques using a source-level debugger for C.

Right now, however, we wish to stay at the level of the machine architecture, so we will see what we can accomplish with a few very elementary interactive debugging operations. We will set breakpoints, single-step, and examine the state of a program written in the LC-3 ISA.

In Chapter 15, we will see these same concepts again: breakpoints, single- stepping, and examining program state that we are introducing here, but applied to a C program, instead of the 0s and 1s of a program written in the LC-3 ISA.

When debugging interactively, the user sits in front of the keyboard and mon- itor and issues commands to the computer. In our case, this means operating an LC-3 simulator, using the menu available with the simulator. It is important to be able to:

1. Write values into memory locations and into registers.
2. Execute instruction sequences in a program.
3. Stop execution when desired.
4. Examine what is in memory and registers at any point in the program. These few simple operations will go a long way toward debugging programs.

!.#.1.1 Set Values

In order to test the execution of a part of a program in isolation without having to worry about parts of the program that come before it, it is useful to first write values in memory and in registers that would have been written by earlier parts of the program. For example, suppose one module in your program supplies input from a keyboard, and a subsequent module operates on that input. Suppose you

want to test the second module before you have finished debugging the first mod- ule. If you know that the keyboard input module ends up with an ASCII code in R0, you can test the module that operates on that input by first writing an ASCII code into R0.

!.#.1.# Execute Sequences

It is important to be able to execute a sequence of instructions and then stop execution in order to examine the values that the program has computed as a result of executing that sequence. Three simple mechanisms are usually available for doing this: run, step, and set breakpoints.

The **Run** command causes the program to execute until something makes it stop. This can be either a HALT instruction or a breakpoint.

The **Step** command causes the program to execute a fixed number of instruc- tions and then stop. The interactive user enters the number of instructions he/she wishes the simulator to execute before it stops. When that number is 1, the com- puter executes one instruction, then stops. Executing one instruction and then stopping is called *single-stepping*. It allows the person debugging the program to examine the individual results of each instruction executed.

The **Set Breakpoint** command causes the program to stop execution at a specific instruction in a program. Executing the debugging command Set Break- point consists of adding an address to a list maintained by the simulator. During the FETCH phase of each instruction, the simulator compares the PC with the addresses in that list. If there is a match, execution stops. Thus, the effect of setting a breakpoint is to allow execution to proceed until the PC contains an address that has been set as a breakpoint. This is useful if one wishes to know what has been computed up to a particular point in the program. One sets a breakpoint at that address in the program and executes the Run command. The program executes until that point and then stops so the user can examine what has been computed up to that point. (When one no longer wishes to have the program stop execution at that point, the breakpoint can be removed by executing the Clear Breakpoint command.)

!.#.1.$ Display Values

Finally, it is useful to examine the results of execution when the simulator has stopped execution. The Display command allows the user to examine the contents of any memory location or any register.

##### !.#.# Use of an Interactive Debugger

We conclude this chapter with four examples, showing how the use of interactive debugging operations can help us find errors in a program. We have chosen the following four errors: (1) incorrectly setting the loop control so that the loop exe- cutes an incorrect number of times, (2) confusing the load instruction 0010, which loads a register with the *contents* of a memory location, with the load effective address instruction 1110, which loads a register with the *address* of a memory location, (3) forgetting which instructions set the condition codes, resulting in

a branch instruction testing the wrong condition, and (4) not covering all possible cases of input values.

!.#.#.1 Example 1: Multiplying Without a Multiply Instruction

Let’s start with an example we have seen before, multiplying two positive integers when the computer (such as the LC-3) does not have a multiply instruction. This time we will assume the two integers to be multiplied are in R4 and R5, and the result of execution (the product of those two integers) will be put in R2. Figure 6.4 shows the program we have written to do the job.

Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

x3200 x3201 x3202 x3203 x3204

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 1 0 1  0 0 0 1 | 0 1 0  0 1 0 | 0 1 0  0 1 0 | | 1  0 | 0 0 0 0 0  0 0 1 0 0 | |
| 0 0 0 1 | 1 0 1 | 1 0 1 | | 1 | 1 1 | 1 1 1 |
| 0 0 0 0 | 0 1 1 | 1 1 1 | | 1 | 1 1 1 0 1 | |
| 1 1 1 1 | 0 0 0 | 0 0 0 1 0 0 1 0 1 | | | | |
|  |  | |  | | | |

**R2 <- 0**

**R2 <- R2 + R4 R5 <- R5 - 1**

**BRzp x3201 HALT**

**Figure !.4 Debugging Example 1. An LC-$ program to multiply (without a Multiply instruction).**

If we examine the program instruction by instruction, we note that the pro- gram first clears R2 (i.e., initializes R2 to 0) and then attempts to perform the multiplication by adding R4 to itself a number of times equal to the initial value in R5. Each time an add is performed, R5 is decremented. When R5 = 0, the program terminates.

It looks like the program should work! Upon execution, however, we find that if R4 initially contains the integer 10 and R5 initially contains the integer 3, the program produces the result 40. What went wrong?

Our first thought is to trace the program. Before we do that, we note that the program assumes positive integers in R4 and R5. Using the Set Values command, we put the value 10 in R4 and the value 3 in R5.

It is also useful to annotate each instruction with some algorithmic descrip- tion of **exactly** what each instruction is doing. While this can be very tedious and not very helpful in a 10,000-instruction program, it often can be very helpful after one has isolated a bug to within a few instructions. There is a big difference between quickly eyeballing a sequence of instructions and stating precisely what each instruction is doing. Quickly eyeballing often results in mistaking what one eyeballs! Stating precisely usually does not. We have included in Figure 6.4, next to each instruction, such an annotation.

Figure 6.5a shows a trace of the program, which we can obtain by single- stepping. The column labeled *PC* shows the contents of the PC at the start of each instruction. R2, R4, and R5 show the values in those three registers at the start of each instruction.

A quick look at the trace shows that the loop body was executed four times, rather than three. That suggests that the condition codes for our branch instruction could have been set incorrectly. From there it is a short step to noting that the branch should have been taken only when R5 was positive, and not when R5 is 0. That is, bit [10]=1 in the branch instruction caused the extra iteration of the loop.

(a)

|  |  |  |  |
| --- | --- | --- | --- |
| PC  x3201 | R2  0 | R4  10 | R5  3 |
| x3202 x3203 x3201 x3202 x3203 x3201 x3202 x3203 x3201 x3202 x3203 x3204 | 10  10  10  20  20  20  30  30  30  40  40  40  40 | 10  10  10  10  10  10  10  10  10  10  10  10  10 | 3  2  2  2  1  1  1  0  0  0  −1  −1  −1 |
|  |  |  |  |

(b)

|  |  |  |  |
| --- | --- | --- | --- |
| PC  x3203 | R2  10 | R4  10 | R5  2 |
| x3203 | 20 | 10 | 1 |
| x3203 | 30 | 10 | 0 |
| x3203 | 40 | 10 | −1 |

**Figure !.5 Debugging Example #. (a) A trace of the Multiply program. (b) Tracing with breakpoints.**

The program can be corrected by simply replacing the instruction at x3203 with

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |
| --- | --- | --- |
| 0 0 BR 0 0 | 0 0 1  n z p | 1 1 1 1 1 1 1 0 1  −3 |

We should also note that we could have saved a lot of the work of tracing the program by using a breakpoint. That is, instead of examining the results of **each instruction,** if we set a breakpoint at x3203, we would examine the results of of the loop. Setting a breakpoint to stop the program after each

**e**it**a**e**c**ra**h**ti**i**o**t**n**er**o**a**f**t**t**i**h**o**e**n**

loop is often enough to have us see the problem (and debug the pro- the tedium of single-stepping each iteration of the loop. Figure 6.5b esults of tracing the program, where each step is one iteration of the see that the loop executed four times instead of three, immediately

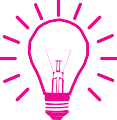
gram) without

shows the r

loop. We

identifying the bug.

One last comment before we leave this example. Before we started tracing program, we initialized R4 and R5 with values 10 and 3. When testing a ogram, it is important to judiciously choose the initial values for the test. Here, e program stated that the program had to work only for positive integers. So, 10 nd 3 are probably OK. What if a (different) multiply program had been written



the

pr

th

a

to work for all integers? Then we could have tried initial values of −6 and 3, 4 and

−12, and perhaps −5 and −7. The problem with this set of tests is that we have left out one of the most important initial values of all: 0. For the program to work for “all” integers, it has to work for 0 as well. The point is that, for a program to work, it must work for all valid source operands, and a good test of such a program is to set source operands to the unusual values, the ones the programmer may have failed to consider. These values are often referred to colloquially as *corner*

*cases*, and more often that not, they are the values for which the program does not operate correctly.

!.#.#.# Example #: Adding a Column of Numbers

The program of Figure 6.6 is supposed to add the numbers stored in the ten locations starting with x3100, and leave the result in R1.

Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

x3000 x3001 x3002 x3003 x3004 x3005 x3006 x3007 x3008 x3009

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 1 0 1  0 1 0 1 | 0 0 1  1 0 0 | 0 0 1  1 0 0 | | 1  1 | 0 0 0 0 0  0 0 0 0 0 | |
| 0 0 0 1 | 1 0 0 | 1 0 0 | | 1 | 0 1 0 1 0 | |
| 0 0 1 0 | 0 1 0 | 0 1 1 | | 1 | 1 1 1 0 0 | |
| 0 1 1 0 | 0 1 1 | 0 1 0 0 0 0 0 0 0 | | | | |
| 0 0 0 1 | 0 1 0 | 0 1 0 | | 1 0 0 0 0 1 | | |
| 0 0 0 1 | 0 0 1 | 0 0 1 | | 0 | 0 0 0 1 1 | |
| 0 0 0 1 | 1 0 0 | 1 0 0 | | 1 | 1 1 | 1 1 1 |
| 0 0 0 0 | 0 0 1 | 1 1 1 | | 1 | 1 1 0 1 1 | |
| 1 1 1 1 | 0 0 0 | 0 0 0 1 0 0 1 0 1 | | | | |
|  |  | |  | | | |

**Figure !.! Debugging Example #. An LC-$ program to add 1’ integers.**

**R1 <- 0**

**R4 <- 0**

**R4 <- R4 + 10 R2 <- M[x3100] R3 <- M[R2] R2 <- R2 + 1 R1 <- R1 + R3 R4 <- R4 - 1 BRp x3004**

**HALT**

The contents of the 20 memory locations starting at location x3100 are shown in Figure 6.7.

The program should work as follows: The instructions in x3000 to x3003 initialize the variables. In x3000, the sum (R1) is initialized to 0. In x3001 and

|  |  |
| --- | --- |
| Address  x3100 | Contents  x3107 |
| x3101 | x2819 |
| x3102 | x0110 |
| x3103 | x0310 |
| x3104 | x0110 |
| x3105 | x1110 |
| x3106 | x11B1 |
| x3107 | x0019 |
| x3108 | x0007 |
| x3109 | x0004 |
| x310A | x0000 |
| x310B | x0000 |
| x310C | x0000 |
| x310D | x0000 |
| x310E | x0000 |
| x310F | x0000 |
| x3110 | x0000 |
| x3111 | x0000 |
| x3112 | x0000 |
| x3113 | x0000 |

**Figure !.( Contents of memory locations x$1’’ to x$11$ for debugging Example #.**

|  |  |  |  |
| --- | --- | --- | --- |
| PC  x3001 | R1  0 | R2  x | R4  x |
| x3002 | 0 | x | 0 |
| x3003 | 0 | x | #10 |
| x3004 | 0 | x3107 | #10 |

**Figure !.) Debugging Example #. A trace of the first four instructions of the Add program.**

x3002, the loop control (R4), which counts the number of values added to R1, is initialized to #10. The program subtracts 1 each time through the loop and repeats until R4 contains 0. In x3003, the base register (R2) is initialized to the starting location of the values to be added: x3100.

From there, each time through the loop, one value is loaded into R3 (in x3004), the base register is incremented to get ready for the next iteration (x3005), the value in R3 is added to R1, which contains the running sum (x3006), the counter is decremented (x3007), the P bit is tested, and if true, the PC is set to x3004 to begin the next iteration of the loop body (x3008). After ten times through the loop, R4 contains 0, the P bit is 0, the branch is not taken, and the program terminates (x3009).

It looks like the program should work. However, when we execute the pro- gram and then check the value in R1, we find the number x0024, which is not x8135, the sum of the numbers stored in locations x3100 to x3109. What went wrong?

We turn to the debugger and trace the program. Figure 6.8 shows a trace of the first four instructions executed. Note that after the instruction at x3003 has executed, R2 contains x3107, not x3100 as we had expected. The problem is that the opcode 0010 loaded the **contents** of M[x3100] (i.e., x3107) into R2, not the **address** x3100. The result was to add the ten numbers starting at M[x3107] instead of the ten numbers starting at M[x3100].

Our mistake: We used the wrong opcode. We should have used the opcode 1110, which would have loaded R2 with the address x3100. We correct the bug by replacing the opcode 0010 with 1110, and the program runs correctly.

!.#.#.$ Example $: Does a Sequence of Memory Locations Contain a &?

The program of Figure 6.9 has been written to examine the contents of the ten memory locations starting at address x3100 and to store a 1 in R0 if any of them contains a 5 and a 0 in R0 if none of them contains a 5.

The program is supposed to do the following: The first six instructions (at x3000 to x3005) initialize R0 to 1, R1 to −5, and R3 to 10. The instruction at x3006 initializes R4 to the address (x3100) of the first location to be tested, and x3007 loads the contents of x3100 into R2.

The instructions at x3008 and x3009 determine if R2 contains the value 5 by adding −5 to R2 and branching to x300F if the result is 0. Since R0 is initialized to 1, the program terminates with R0 reporting the presence of a 5 among the locations tested.

Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 x3000

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 1 0 1  0 0 0 1 | 0 0 0  0 0 0 | 0 0 0  0 0 0 | | 1  1 | 0 0 0 0 0  0 0 0 0 1 | |
| 0 1 0 1 | 0 0 1 | 0 0 1 | | 1 | 0 0 0 0 0 | |
| 0 0 0 1 | 0 0 1 | 0 0 1 | | 1 | 1 1 0 1 1 | |
| 0 1 0 1 | 0 1 1 | 0 1 1 | | 1 | 0 0 0 0 0 | |
| 0 0 0 1 | 0 1 1 | 0 1 1 | | 1 | 0 1 0 1 0 | |
| 0 0 1 0 | 1 0 0 | 0 0 0 | | 0 | 0 1 0 0 1 | |
| 0 1 1 0 | 0 1 0 | 1 0 0 0 0 0 0 0 0 | | | | |
| 0 0 0 1 | 0 1 0 | 0 1 0 | | 0 0 0 0 0 1 | | |
| 0 0 0 0 | 0 1 0 | 0 0 0 | | 0 | 0 0 | 1 0 1 |
| 0 0 0 1 | 1 0 0 | 1 0 0 1 0 0 0 0 1 | | | | |
| 0 0 0 1 | 0 1 1 | 0 1 1 | | 1 | 1 1 1 1 1 | |
| 0 1 1 0 | 0 1 0 | 1 0 0 | | 0 | 0 0 0 0 0 | |
| 0 0 0 0 | 0 0 1 | 1 1 1 | | 1 1 1 0 1 0 | | |
| 0 1 0 1 | 0 0 0 | 0 0 0 1 0 0 0 0 0 | | | | |
| 1 1 1 1 | 0 0 0 | 0 0 0 | | 1 | 0 0 1 0 1 | |
| 0 0 1 1 | 0 0 0 1 | | 0 0 0 0 0 0 0 0 | | | |
|  | | | | | | |

x3001 x3002 x3003 x3004 x3005 x3006 x3007 x3008 x3009 x300A x300B x300C x300D x300E x300F x3010

**Figure !.9 Debugging Example $. An LC-$ program to detect the presence of a &.**

**R0 <- 0**

**R0 <- R0 + 1 R1 <- 0**

**R1 <- R1 - 5 R3 <- 0**

**R3 <- R3 + 10 R4 <- M[x3010] R2 <- M[R4] R2 <- R2 + R1 BRz x300F**

**R4 <- R4 + 1 R3 <- R3 - 1 R2 <- M[R4] BRp x3008 R0 <- 0**

**HALT x3100**

x300A increments R4, preparing to load the next value. x300B decrements R3, indicating the number of values remaining to be tested. x300C loads the next value into R2. x300D branches back to x3008 to repeat the process if R3 still indicates more values to be tested. If R3 = 0, we have exhausted our tests, so R0 is set to 0 (x300E), and the program terminates (x300F).

When we run the program for some sample data that contains a 5 in one of the memory locations, the program terminates with R0 = 0, indicating there were no 5s in locations x3100 to x310A.

What went wrong? We examine a trace of the program, with a breakpoint set at x300D. The results are shown in Figure 6.10.

The first time the PC is at x300D, we have already tested the value stored in x3100, we have loaded 7 (the contents of x3101) into R2, and R3 indicates there are still nine values to be tested. R4 contains the address from which we most recently loaded R2.

The second time the PC is at x300D, we have loaded 32 (the contents of x3102) into R2, and R3 indicates there are eight values still to be tested. The third time the PC is at x300D, we have loaded 0 (the contents of x3103) into R2, and R3 indicates seven values still to be tested. The value 0 loaded into R2 causes the branch instruction at x300D to be not taken, R0 is set to 0 (x300E), and the program terminates (x300F) before the locations containing a 5 are tested.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PC  x300D | R1  5 | R2  7 | R3  9 | R4  3101 |
| x300D x300D | −  −5  −5 | 32  0 | 8  7 | 3102  3013 |

**Figure !.1’ Debugging Example $. Tracing Example $ with a breakpoint at x$’’D.**

The error in the program occurred because the branch instruction immedi- ately followed the load instruction that set the condition codes based on what was loaded. That wiped out the condition codes set by the iteration control instruction at x300B, which was keeping track of the number of iterations left to do. Since the branch instruction should branch if there are still more memory locations to examine, the branch instruction should have immediately followed the iteration control instruction and NOT the load instruction which also sets condition codes.

A conditional branch instruction should be considered the second instruction in a pair of instructions.

Instruction A ; sets the condition codes

BR instruction ; branches based on the condition codes

The first instruction in the pair (Instruction A) sets the condition codes. The second instruction (BR) branches or not, depending on the condition codes set by instruction A. It is important to never insert any instruction that sets condition codes between instruction A and the branch instruction, since doing so will wipe out the condition codes set by instruction A that are needed by the branch instruction.

Since the branch at x300D was based on the value loaded into R2, instead of how many values remained to be tested, the third time the branch instruction was executed, it was not taken when it should have been. If we interchange the instructions at x300B and x300C, the branch instruction at x300D immediately follows the iteration control instruction, and the program executes correctly.

It is also worth noting that the branch at x300D *coincidentally* behaved cor- rectly the first two times it executed because the load instruction at x300C loaded positive values into R2. The bug did not produce incorrect behavior until the third iteration. It would be nice if bugs would manifest themselves the first time they are encountered, but that is often not the case. Coincidences do occur, which adds to the challenges of debugging.

!.#.#.4 Example 4: Finding the First 1 in a Word

Our last example contains an error that is usually one of the hardest to find, as we will see. The program of Figure 6.11 has been written to examine the contents of a memory location, find the first bit (reading from left to right) that is set, and store its bit position into R1. If no bit is set, the program is to store −1 in R1. For

Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

x3000 x3001 x3002 x3003 x3004 x3005 x3006 x3007 x3008 x3009

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 1 0 1  0 0 0 1 | 0 0 1  0 0 1 | 0 0 1  0 0 1 | 1  1 | 0 0 0 0 0  0 1 1 1 1 | |
| 1 0 1 0 | 0 1 0 | 0 0 0 | 0 | 0 0 1 1 0 | |
| 0 0 0 0 | 1 0 0 | 0 0 0 0 0 0 1 0 0 | | | |
| 0 0 0 1 | 0 0 1 | 0 0 1 1 1 1 1 1 1 | | | |
| 0 0 0 1 | 0 1 0 | 0 1 0 | 0 | 0 0 0 1 0 | |
| 0 0 0 0 | 1 0 0 | 0 0 0 | 0 | 0 0 | 0 0 1 |
| 0 0 0 0 | 1 1 1 | 1 1 1 1 1 1 1 0 0 | | | |
| 1 1 1 1 | 0 0 0 | 0 0 0 1 0 0 1 0 1 | | | |
| 0 0 1 1 | 0 1 0 0 0 0 0 0 0 0 0 0 | | | | |
|  | | | | | |

**Figure !.11 Debugging Example 4. An LC-$ program to find the first 1 in a word.**

**R1 <- 0**

**R1 <- R1 + 15**

**R2 <- M[M[x3009]] BRn x3008**

**R1 <- R1 - 1 R2 <- R2 + R2 BRn x3008**

**BRnzp x3004 HALT**

**x3400**

example, if the location examined contained 0010000110000000, the program would terminate with R1 = 13. If the location contained 0000000000000110, the program would terminate with R1 = 2.

The program Figure 6.11 is supposed to work as follows (and it usually does): x3000 and x3001 initialize R1 to 15, the bit number of the leftmost bit.

x3002 loads R2 with the contents of x3400, the bit pattern to be exam- ined. Since x3400 is too far from x3000 for a LD instruction, the load indirect instruction is used, obtaining the location of the bit pattern in x3009.

x3003 tests the most significant bit of the bit pattern (bit [15]), and if it is a 1, branches to x3008, where the program terminates with R1=15. If the most significant bit is 0, the branch is not taken, and processing continues at x3004.

The loop body, locations x3004 to x3007, does two things. First (x3004), it subtracts 1 from R1, yielding the bit number of the next bit to the right. Second (x3005), it adds R2 to itself, resulting in the contents of R2 shifting left one bit, resulting in the next bit to the right being shifted into the bit [15] position. Third (x3006), the BR instruction tests the “new” bit [15], and if it is a 1, branches to x3008, where the program halts with R1 containing the actual bit number of the current leftmost bit. If the new bit [15] is 0, x3007 is an unconditional branch to x3004 for the next iteration of the loop body.

The process continues until the first 1 is found. The program works correctly almost all the time. However, when we ran the program on our data, the program failed to terminate. What went wrong?

A trace of the program, with a breakpoint set at x3007, is shown in Figure 6.12.

|  |  |
| --- | --- |
| PC  x3007 | R1  14 |
| x3007 | 13 |
| x3007 | 12 |
| x3007 | 11 |
| x3007 | 10 |
| x3007 | 9 |
| x3007 | 8 |
| x3007 | 7 |
| x3007 | 6 |
| x3007 | 5 |
| x3007 | 4 |
| x3007 | 3 |
| x3007 | 2 |
| x3007 | 1 |
| x3007 | 0 |
| x3007 | −1 |
| x3007 | −2 |
| x3007 | −3 |
| x3007 | −4 |

**Figure !.1# Debugging Example 4. A Trace of Debugging Example 4 with a breakpoint at x$’’(.**

Each time the PC contained the address x3007, R1 contained a value smaller by 1 than the previous time. The reason is as follows: After R1 was decremented and the value in R2 shifted left, the bit tested was a 0, and so the program did not terminate. This continued for values in R1 equal to 14, 13, 12, 11, 10, 9, 8, 7, 6,

5, 4, 3, 2, 1, 0, −1, −2, −3, −4, and so forth.

The problem was that the initial value in x3400 was x0000. The program worked fine as long as there was at least one 1 present. For the case where x3400 contained all zeros, the conditional branch at x3006 was never taken, and so the program continued with execution of x3007, then x3004, x3005, x3006, x3007, and then back again to x3004. There was no way to break out of the sequence x3004, x3005, x3006, x3007, and back again to x3004. We call the sequence x3004 to x3007 a loop. Because there is no way for the program execution to break out of this loop, we call it an *infinite loop*. Thus, the program never terminates, and so we can never get the correct answer.

Again, we emphasize that this is often the hardest error to detect because it is as we said earlier a corner case. The programmer assumed that at least one bit was set. What if no bits are set? That is, it is not enough for a program to execute correctly most of the time; it must execute correctly all the time, independent of the data that the program is asked to process.

**Exercises**

**!.1** Can a procedure that is *not* an algorithm be constructed from the three basic constructs of structured programming? If so, demonstrate through an example.

**!.#** The LC-3 has no Subtract instruction. If a programmer needed to subtract two numbers, he/she would have to write a routine to handle it. Show the systematic decomposition of the process of subtracting two integers.

**!.$** Recall the machine busy example from previous chapters. Suppose memory location x4000 contains an integer between 0 and 15 identifying a particular machine that has just become busy. Suppose further that the value in memory location x4001 tells which machines are busy and which machines are idle. Write an LC-3 machine language program that sets the appropriate bit in x4001 indicating that the machine in x4000 is busy.

For example, if x4000 contains x0005 and x4001 contains x3101 at the start of execution, x4001 should contain x3121 after your program terminates.

**!.4** Write a short LC-3 program that compares the two numbers in R1 and R2 and puts the value 0 in R0 if R1 = R2, 1 if R1 > R2, and −1 if

R1 < R2.

**!.&** Which of the two algorithms for multiplying two numbers is preferable and why? 88 · 3 = 88 + 88 + 88 OR 3 + 3 + 3 + 3 +…+ 3?

**!.!** Use your answers from Exercises 6.4 and 6.5 to develop a program that efficiently multiplies two integers and places the result in R3. Show the

complete systematic decomposition, from the problem statement to the final program.

**!.(** What does the following LC-3 program do?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x3001 | 1110 | 0000 | 0000 | 1100 |
| x3002 | 1110 | 0010 | 0001 | 0000 |
| x3003 | 0101 | 0100 | 1010 | 0000 |
| x3004 | 0010 | 0100 | 0001 | 0011 |
| x3005 | 0110 | 0110 | 0000 | 0000 |
| x3006 | 0110 | 1000 | 0100 | 0000 |
| x3007 | 0001 | 0110 | 1100 | 0100 |
| x3008 | 0111 | 0110 | 0000 | 0000 |
| x3009 | 0001 | 0000 | 0010 | 0001 |
| x300A | 0001 | 0010 | 0110 | 0001 |
| x300B | 0001 | 0100 | 1011 | 1111 |
| x300C | 0000 | 0011 | 1111 | 1000 |
| x300D | 1111 | 0000 | 0010 | 0101 |
| x300E | 0000 | 0000 | 0000 | 0101 |
| x300F | 0000 | 0000 | 0000 | 0100 |
| x3010 | 0000 | 0000 | 0000 | 0011 |
| x3011 | 0000 | 0000 | 0000 | 0110 |
| x3012 | 0000 | 0000 | 0000 | 0010 |
| x3013 | 0000 | 0000 | 0000 | 0100 |
| x3014 | 0000 | 0000 | 0000 | 0111 |
| x3015 | 0000 | 0000 | 0000 | 0110 |
| x3016 | 0000 | 0000 | 0000 | 1000 |
| x3017 | 0000 | 0000 | 0000 | 0111 |
| x3018 | 0000 | 0000 | 0000 | 0101 |

**!.)** Why is it necessary to initialize R2 in the character counting example in Section 6.1.4? In other words, in what manner might the program behave incorrectly if the R2 +- 0 step were removed from the routine?

**!.9** Using the iteration construct, write an LC-3 machine language routine that displays exactly 100 Zs on the screen.

**!.1’** Using the conditional construct, write an LC-3 machine language routine that determines if a number stored in R2 is odd.

**!.11** Write an LC-3 machine language routine to increment each of the numbers stored in memory location A through memory location B. Assume these locations have already been initialized with meaningful numbers. The addresses A and B can be found in memory locations x3100 and x3101.

**!.1#** *a.* Write an LC-3 machine language routine that echoes the last character typed at the keyboard. If the user types an *R,* the program then immediately outputs an *R* on the screen.

1. Expand the routine from part *a* such that it echoes a line at a time. For

example, if the user types:

The quick brown fox jumps over the lazy dog.

then the program waits for the user to press the Enter key (the ASCII code for which is x0A) and then outputs the same line.

**!.1$** Notice that we can shift a number to the left by one bit position by adding it to itself. For example, when the binary number 0011 is added to itself, the result is 0110. Shifting a number one bit pattern to the right is not as easy. Devise a routine in LC-3 machine code to shift the contents of memory location x3100 to the right by one bit.

**!.14** Consider the following machine language program:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x3000 | 0101 | 0100 | 1010 | 0000 |
| x3001 | 0001 | 0010 | 0111 | 1111 |
| x3002 | 0001 | 0010 | 0111 | 1111 |
| x3003 | 0001 | 0010 | 0111 | 1111 |
| x3004 | 0000 | 1000 | 0000 | 0010 |
| x3005 | 0001 | 0100 | 1010 | 0001 |
| x3006 | 0000 | 1111 | 1111 | 1010 |
| x3007 | 1111 | 0000 | 0010 | 0101 |

What are the possible initial values of R1 that cause the final value R2 to be 3?

in

**!.1&** Shown below are the contents of memory and registers **before** and **after** the LC-3 instruction at location x3010 is executed. Your job: Identify the instruction stored in x3010. *Note:* There is enough information below to uniquely specify the instruction at x3010.

|  |  |  |
| --- | --- | --- |
|  | Before | After |
| R0:  R1:  R2:  R3:  R4:  R5:  R6:  R7: | x3208  x2d7c xe373 x2053  x33ff x3f1f xf4a2 x5220 | x3208  x2d7c xe373 x2053  x33ff x3f1f xf4a2 x5220 |
| ...  x3400: x3401: x3402: x3403: x3404: x3405: x3406: x3407: x3408:  ... | x3001  x7a00 x7a2b xa700 xf011 x2003  x31ba xc100 xefef | x3001  x7a00 x7a2b xa700 xf011 x2003  xe373 xc100 xefef |

**!.1!** An LC-3 program is located in memory locations x3000 to x3006. It starts executing at x3000. If we keep track of all values loaded into the MAR as the program executes, we will get a sequence that starts as follows. Such a sequence of values is referred to as a trace.

MAR Trace

x3000 x3005 x3001 x3002 x3006 x4001 x3003 x0021

We have shown below some of the bits stored in locations x3000 to x3006. Your job is to fill in each blank space with a 0 or a 1, as appropriate.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x3000 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |
| x300l | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| x3002 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |
| x3003 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| x3004 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| x3005 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| x3006 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**!.1(** Shown below are the contents of registers before and after the LC-3 instruction at location x3210 is executed. Your job: Identify the instruction stored in x3210. *Note:* There is enough information below to uniquely specify the instruction at x3210.

|  |  |  |
| --- | --- | --- |
|  | Before | After |
| R0: | xFF1D | xFF1D |
| R1: | x301C | x301C |
| R2: | x2F11 | x2F11 |
| R3: | x5321 | x5321 |
| R4: | x331F | x331F |
| R5: | x1F22 | x1F22 |
| R6: | x01FF | x01FF |
| R7: | x341F | x3211 |
| PC: | x3210 | x3220 |
| N: | 0 | 0 |
| Z: | 1 | 1 |
| P: | 0 | 0 |

**!.1)** The LC-3 has no Divide instruction. A programmer needing to divide two numbers would have to write a routine to handle it. Show the systematic decomposition of the process of dividing two positive integers. Write an LC-3 machine language program starting at location x3000 that divides the number in memory location x4000 by the number in memory location x4001 and stores the quotient at x5000 and the remainder at x500l.

**!.19** It is often necessary to encrypt messages to keep them away from prying eyes. A message can be represented as a string of ASCII characters, one per memory location, in consecutive memory locations. Bits [15:8] of each location contain 0, and the location immediately following the string contains x0000.

A student who has not taken this course has written the following LC-3 machine language program to encrypt the message starting at location x4000 by adding 4 to each character and storing the resulting message at x5000. For example, if the message at x4000 is “Matt,” then the encrypted message at x5000 is “Qeyy.” However, there are four bugs in his code. Find and correct these errors so that the program works correctly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x3000 | 1110 | 0000 | 0000 | 1010 |
| x3001 | 0010 | 0010 | 0000 | 1010 |
| x3002 | 0110 | 0100 | 0000 | 0000 |
| x3003 | 0000 | 0100 | 0000 | 0101 |
| x3004 | 0001 | 0100 | 1010 | 0101 |
| x3005 | 0111 | 0100 | 0100 | 0000 |
| x3006 | 0001 | 0000 | 0010 | 0001 |
| x3007 | 0001 | 0010 | 0110 | 0001 |
| x3008 | 0000 | 1001 | 1111 | 1001 |
| x3009 | 0110 | 0100 | 0100 | 0000 |
| x300A | 1111 | 0000 | 0010 | 0101 |
| x300B | 0100 | 0000 | 0000 | 0000 |
| x300C | 0101 | 0000 | 0000 | 0000 |

**!.#’** Redo Exercise 6.18 for all integers, not just positive integers.

**!.#1** You have been asked to design the volume control system in a stereo. The user controls the volume by using Volume Up and Volume Down buttons on the stereo. When the user presses the Volume Up button, the volume should increase by 1; when the user presses the Volume Down button, the volume should decrease by 1. The volume level is represented as a four-bit unsigned value, ranging from 0 to 15. If the user presses Volume Up when the volume is already at the maximum level of 15, the volume should remain at 15; similarly, if the user presses Volume Down when the volume is already at the minimum level of 0, the volume should remain at 0. The memory location x3100 has been directly hooked up to the speakers so that reading bits 3 through 0 from that memory location will give the current speaker volume, while writing bits [3:0] of that memory location will set the new speaker volume.

When the user presses one of the volume buttons, the stereo hardware will reset the PC of the processor to x3000 and begin execution. If the user presses Volume Up, then memory location x3101 will be set to 1; otherwise, if the user presses Volume Down, then the memory location x3101 will be set to 0.

Below is the program that controls the volume on the stereo. Two of the instructions in the program have been left out. Your job: Fill in the missing instructions so that the program controls the volume correctly as specified.

|  |  |  |
| --- | --- | --- |
| Address  x3000 | Contents  0010000011111111 | Description  R0 +- M[x3100] |
| x3001 | 0010001011111111 | R1 +- M[x3101] |
| x3002 | 0000010000000100 | Branch to x3007 if Z is set |
| x3003 |  |  |
| x3004 | 0000010000000101 | Branch to x300A if Z is set |
| x3005 | 0001000000100001 | R0 +- R0 + x0001 |
| x3006 | 0000111000000011 | Branch always to x300A |
| x3007 | 0001001000100000 | R1 +- R0 + x0000 |
| x3008 | 0000010000000001 | Branch to x300A if Z is set |
| x3009 |  |  |
| x300A | 0011000011110101 | M[x3100] +- R0 |
| x300B | 1111000000100101 | TRAP x25 |
|  |  |  |

\***!.##** A warehouse is controlled by an electronic lock having an *n*-digit combination. The electronic lock has ten buttons labeled 0 to 9 on its face. To open the lock, a user presses a sequence of *n* buttons. The corresponding ASCII characters get loaded into sequential locations of memory, starting at location x3150. After *n* buttons have been pressed, the null character x00 is loaded into the next sequential memory location.

The following program determines whether or not the lock should open, depending on whether the combination entered agrees with the combination stored in the *n* memory locations starting at x3100. If the lock should open, the program stores a 1 in location x3050. If the lock should not open, the program stores a 0 in location x3050.

Note that some of the instructions are missing.

Complete the program by filling in the missing instructions.

|  |  |  |
| --- | --- | --- |
| x3000  x3001 | 0101 101 101 1 00000  0010 000 000001111 | ; R5 *<*– x0000  ; R0 *<*– M[x3011] |
| x3002 | 0010 001 000001101 | ; R1 *<*– M[x3010] |
| x3003 | 0110 010 000 000000 | ; R2 *<*– M[R0] |
| x3004 |  |  |
| x3005 | 0110 011 001 000000 | ; R3 *<*– M[R1] |
| x3006 | 1001 011 011 111111 | ; NOT R3 |
| x3007 | 0001 011 011 1 00001 | ; R3 *<*– R3 + 1 |
| x3008 |  |  |
| x3009 | 0000 101 000000100 | ; Branch to x300E if N or P is set |
| x300A |  |  |
| x300B |  |  |
| x300C | 0000 111 111110110 | ; Branch always to x3003 |
| x300D |  |  |
| x300E | 0011 101 001000001 | ; Store R5 in x3050 |
| x300F | 1111 0000 0010 0101 | ; HALT |
| x3010 | 0011 0001 0000 0000 | ; x3100 |
| x3011 | 0011 0001 0101 0000 | ; x3150 |
|  |  |  |

A simple change to the contents of memory will allow us to eliminate the instructions at memory locations x3006 and x3007 in our program. What is the change?

\***!.#$** The PC is loaded with x3000, and the instruction at address x3000 is executed. In fact, execution continues and four more instructions are executed. The table below contains the contents of various registers at the end of execution for each of the five (total) instructions.

Your job: Complete the table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Before execution starts | x3**P**0**C**00 | **M**—**A**—**R** | **M**—**D**—**R** | —**IR**— | x0**R**0**0**00 | x0**R**0**1**00 |
| After the first finishes |  |  | xB333 | x2005 |  |  |
| After the 2nd finishes |  |  |  | x0601 |  |  |
| After the 3rd finishes |  |  | x1 |  |  | x0001 |
| After the 4th finishes |  |  | x1 |  | x6666 |  |
| After the 5th finishes |  |  |  | x0BFC |  |  |
|  |  |  |  |  |  |  |

Let’s start execution again, starting with PC = x3000. First, we re-initialize R0 and R1 to 0, and set a breakpoint at x3004. We press RUN eleven times, and each time the program executes until the breakpoint. What are the final values of R0 and R1?

**!.#4** A student is debugging his program. His program does not have access to memory locations x0000 to x2FFF. Why that is the case we will discuss before the end of the book. The term is “privileged memory” but not something for you to worry about today.

He sets a breakpoint at x3050, and then starts executing the program. When the program stops, he examines the contents of several memory locations and registers, then hits single step. The simulator executes one instruction and then stops. He again examines the contents of the memory locations and registers. They are as follows:

|  |  |  |
| --- | --- | --- |
|  | Before | After |

|  |  |  |
| --- | --- | --- |
| PC  R0 | x3050  x2F5F | x3051  xFFFF |
| R1 | x4200 | x4200 |
| R2 | x0123 | x0123 |
| R3 | x2323 | x2323 |
| R4 | x0010 | x0010 |
| R5 | x0000 | x0000 |
| R6 | x1000 | x1000 |
| R7 | x0522 | x0522 |
|  |  |  |
| **M**M[**[**x**x**4**3**2**0**0**5**0**0**]**]** | **x**x**6**5**?**5**?**5**?**5 | **x**x**6**5**?**5**?**5**?**5 |
| M[x4201] | xFFFF | xFFFF |
|  |  |  |

Complete the contents of location x3050

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |

**!.#&** A student is writing a program and needs to subtract the contents of R1 from the contents of R2 and put the result in R3. Instead of writing:

NOT R3,R1

ADD R3,R3,#1

ADD R3,R3,R2

she writes:

NOT R3,R1

.FILL x16E1 ADD R3,R3,R2

She assembles the program and attempts to execute it. Does the subtract execute correctly? Why or why not?

\***!.#!** During the execution of an LC-3 program, an instruction in the program starts executing at clock cycle T and requires 15 cycles to complete.

The table lists **ALL** five clock cycles during the processing of this instruction, which require use of the bus. The table shows for each of those clock cycles: which clock cycle, the state of the state machine, the value on the bus, and the important control signals that are active during that clock cycle.

|  |  |  |  |
| --- | --- | --- | --- |
| Cycle | State | Bus | Important Control Signals for This Cycle |

|  |  |  |  |
| --- | --- | --- | --- |
| T  T + 4 | 18 | x3010 | LD.MAR = 1, LD.PC =1, PCMux = PC + 1, GatePC = 1 |
| T + 6 |  | x3013 |  |
| T + 10 |  | x4567 |  |
| T + 14 |  | x0000 | LD.REG = 1, LD.CC = 1, GateMDR = 1, DR = 001 |
|  |  |  |  |

* 1. Fill in the missing entries in the table.
  2. What is the instruction being processed?
  3. Where in memory is that instruction?
  4. How many clock cycles does it take memory to read or write?
  5. There is enough information above for you to know the contents of three memory locations. What are they, and what are their contents?

\***!.#(** An LC-3 program starts execution at x3000. During the execution of the program, a snapshot of all eight registers was taken at six different times as shown below: before the program executes, after execution of instruction 1, after execution of instruction 2, after execution of instruction 3, after execution of instruction 4, after execution of instruction 5, and after execution of instruction 6.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Registers** | **Initial**  **Value** | **After 1st**  **Instruction** | **After 2nd**  **Instruction** | **After 3rd**  **Instruction** | **After 4th**  **Instruction** | **After 5th**  **Instruction** | **After 6th**  **Instruction** |
| **R0** | x4006 | x4050 | x4050 | x4050 | x4050 | x4050 | x4050 |
| **R1** | x5009 | x5009 | x5009 | x5009 | x5009 | x5009 | x5009 |
| **R2** | x4008 | x4008 | x4008 | x4008 | x4008 | x4008 | xC055 |
| **R3** | x4002 |  |  | x8005 | x8005 | x8005 | x8005 |
| **R4** | x4003 | x4003 | x4003 | x4003 |  |  | x4003 |
| **R5** | x400D | x400D |  |  | x400D | x400D | x400D |
| **R6** | x400C | x400C | x400C | x400C | x400C | x400C | x400C |
| **R7** | x6001 | x6001 | x6001 | x6001 |  |  | x400E |

Also, during the execution of the program, the PC trace, the MAR trace, and the MDR trace were recorded as shown below. Note that a PC trace records the addresses of the instructions executed in sequence by the program.

|  |
| --- |
| **PC Trace** |
|  |
|  |
| x400D |
|  |
| x400E |
|  |

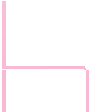
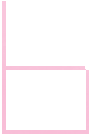
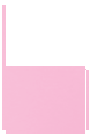
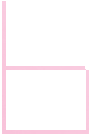
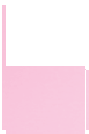
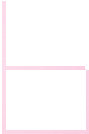
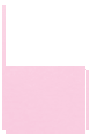
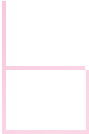
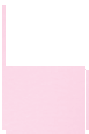
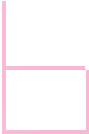
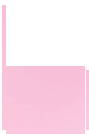
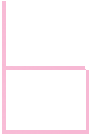
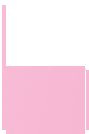
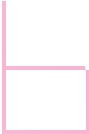
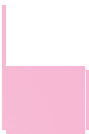
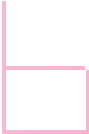
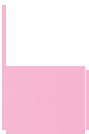
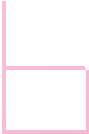
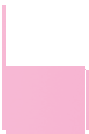
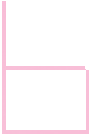
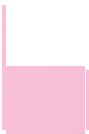
|  |  |
| --- | --- |
| **MAR Trace** | **MD**x**R**A0**T**0**r**9**ace** |
|  |  |
| x3025 |  |
|  | x1703 |
|  |  |
|  | x4040 |
|  |  |
| x400E | x1403 |
|  |  |

Your job: Fill in the missing entries in the three tables above.

CHAPTER

7

Assembly Language



y now, you are probablya little tired of 1s and 0s and keeping track of 0001 meaning ADD and 1001 meaning NOT. Also, wouldn’t it be nice if we could refer to a memory location by some meaningful symbolic name instead of memorizing its 16-bit address? And wouldn’t it be nice if we could represent each instruction in some more easily comprehensible way, instead of having to keep track of which bit of an instruction conveys which individual piece of information

**B**

about that instruction? It turns out that help is on the way.

In this chapter, we introduce assembly language, a mechanism that does all of the above, and more.

## !.1 Assembly Language Programming— Moving Up a Level

Recall the levels of transformation identified in Figure 1.9 of Chapter 1. Algo- rithms are transformed into programs described in some mechanical language. This mechanical language can be, as it is in Chapter 5, the machine language of a particular computer. Recall that a program is in a computer’s machine language if every instruction in the program is from the ISA of that computer.

On the other hand, the mechanical language can be more user-friendly. We erally partition mechanical languages into two classes, high-level and low- el. Of the two, high-level languages are much more user-friendly. Examples

gen

lev

are C, C++, Java, Fortran, COBOL, Python, plus more than a thousand others. Instructions in a high-level language almost (but not quite) resemble statements in a natural language such as English. High-level languages tend to be ISA inde- pendent. That is, once you learn how to program in C (or Fortran or Python) for one ISA, it is a small step to write programs in C (or Fortran or Python) for another ISA.

Before a program written in a high-level language can be executed, it must be translated into a program in the ISA of the computer on which it is expected to execute. It is often the case that each statement in the high-level language specifies several instructions in the ISA of the computer. In Chapter 11, we will introduce the high-level language C, and in Chapters 12 through 19, we will show the rela- tionship between various statements in C and their corresponding translations to LC-3 code. In this chapter, however, we will only move up a small step from the ISA we dealt with in Chapter 5.

A small step up from the ISA of a machine is that ISA’s assembly language. Assembly language is a low-level language. There is no confusing an instruc- tion in a low-level language with a statement in English. Each assembly language instruction usually specifies a single instruction in the ISA. Unlike high-level lan- guages, which are usually ISA independent, low-level languages are very much ISA dependent. In fact, it is usually the case that each ISA has only one assembly language.

The purpose of assembly language is to make the programming process more user-friendly than programming in machine language (i.e., in the ISA of the com- puter with which we are dealing), while still providing the programmer with detailed control over the instructions that the computer can execute. So, for exam- ple, while still retaining control over the detailed instructions the computer is to carry out, we are freed from having to remember what opcode is 0001 and what opcode is 1001, or what is being stored in memory location 0011111100001010 and what is being stored in location 0011111100000101. Assembly languages let us use mnemonic devices for opcodes, such as ADD for 0001 and NOT for 1001, and they let us give meaningful symbolic names to memory locations, such as SUM or PRODUCT, rather than use the memory locations’ 16-bit addresses. This makes it easier to differentiate which memory location is keeping track of a SUM and which memory location is keeping track of a PRODUCT. We call these names *symbolic addresses*.

We will see, starting in Chapter 11, that when we take the larger step of

moving up to a higher-level language (such as C), programming will be even more user-friendly, but in doing so, we will relinquish some control over exactly which detailed ISA instructions are to be carried out to accomplish the work specified by a high-level language statement.

## !.# An Assembly Language Program

We will begin our study of the LC-3 assembly language by means of an example. The program in Figure 7.1 multiplies the integer initially stored in NUMBER by 6 by adding the integer to itself six times. For example, if the integer is 123, the program computes the product by adding 123 + 123 + 123 + 123 + 123 + 123. Where have you seen that before? :-)

The program consists of 21 lines of code. We have added a *line number* to each line of the program in order to be able to refer to individual lines easily. This is a common practice. These line numbers are not part of the program. Ten lines start with a semicolon, designating that they are strictly for the benefit of

01 ;

1. ; Program to multiply an integer by the constant 6.
2. ; Before execution, an integer must be stored in NUMBER. 04 ;
3. .ORIG x3050
4. LD R1,SIX
5. LD R2,NUMBER
6. AND R3,R3,#0 ; Clear R3. It will 09 ; contain the product.

0A ; The inner loop 0B ;

0C AGAIN ADD R3,R3,R2

0D ADD R1,R1,#-1 ; R1 keeps track of 0E BRp AGAIN ; the iterations 0F ;

10 HALT

11 ;

1. NUMBER .BLKW 1
2. SIX .FILL x0006 14 ;

15 .END

**Figure !.1 An assembly language program.**

the human reader. More on this momentarily. Seven lines (06, 07, 08, 0C, 0D, 0E, and 10) specify assembly language instructions to be translated into machine lan- guage instructions of the LC-3, which will be executed when the program runs. The remaining four lines (05, 12, 13, and 15) contain pseudo-ops, which are mes- sages from the programmer to the translation program to help in the translation process. The translation program is called an *assembler* (in this case the LC-3 assembler), and the translation process is called *assembly*.

##### !.#.1 Instructions

Instead of an instruction being 16 0s and 1s, as is the case in the LC-3 ISA, an instruction in assembly language consists of four parts, as follows:

Label Opcode Operands ; Comment

Two of the parts (Label and Comment) are optional. More on that momentarily.

!.#.1.1 Opcodes and Operands

Two of the parts (Opcode and Operands) are **mandatory**. For an assembly lan- guage instruction to correspond to an instruction in the LC-3 ISA, it must have an Opcode (the thing the instruction is to do), and the appropriate number of Operands (the things it is supposed to do it to). Not surprisingly, this was exactly what we encountered in Chapter 5 when we studied the LC-3 ISA.

The Opcode is a symbolic name for the opcode of the corresponding LC-3 truction. The idea is that it is easier to remember an operation by the symbolic

ins

name ADD, AND, or LDR than by the four-bit quantity 0001, 0101, or 0110. Figure 5.3 (also Figure A.2) lists the Opcodes of the 15 LC-3 instructions. Pages 658 through 673 show the assembly language representations for the 15 LC-3 instructions.

The number of operands depends on the operation being performed. For example, the ADD instruction (line 0C in the program of Figure 7.1) requires three operands (two sources to obtain the numbers to be added, and one desti- nation to designate where the result is to be stored). All three operands must be explicitly identified in the instruction.

AGAIN ADD R3,R3,R2

In this case the operands to be added are obtained from register 2 and from register

3. The result is to be placed in register 3. We represent each of the registers 0 through 7 as R0, R1, R2, … , R7, rather than 000, 001, 010, … , 111.

The LD instruction (line 07 of the program in Figure 7.1) requires two operands (the memory location from which the value is to be read and the destina- tion register that is to contain the value after the instruction finishes execution). In LC-3 assembly language, we assign symbolic names called *labels* to the memory locations so we will not have to remember their explicit 16-bit addresses. In this case, the location from which the value is to be read is given the label *NUMBER*. The destination (i.e., where the value is to be loaded) is register 2.

LD R2, NUMBER

As we discussed in Section 5.1.6, operands can be obtained from registers, from memory, or they may be literal (i.e., immediate) values in the instruction. In the case of register operands, the registers are explicitly represented (such as R2 and R3 in line 0C). In the case of memory operands, the symbolic name of the mem- ory location is explicitly represented (such as NUMBER in line 07 and SIX in line 06). In the case of immediate operands, the actual value is explicitly represented (such as the value 0 in line 08).

AND R3, R3, #0 ; Clear R3. It will contain the product.

A literal value must contain a symbol identifying the representation base of the number. We use # for decimal, x for hexadecimal, and b for binary. Sometimes there is no ambiguity, such as in the case 3F0A, which is a hex number. Nonethe- less, we write it as x3F0A. Sometimes there is ambiguity, such as in the case 1000. x1000 represents the decimal number 4096, b1000 represents the decimal number 8, and #1000 represents the decimal number 1000.

!.2.#.2 Labels

Labels are symbolic names that are used to identify memory locations that are referred to explicitly in the program. In LC-3 assembly language, a label consists of from 1 to 20 alphanumeric characters (i.e., each character is a capital or lower- case letter of the English alphabet, or a decimal digit), starting with a letter of the alphabet.

However, not all sequences of characters that follow these rules can be used labels. You know that computer programs cannot tolerate ambiguity. So ADD,

as

NOT, x1000, R4, and other character strings that have specific meanings in an LC-3 program cannot be used as labels. They could confuse the LC-3 assembler as it tries to translate the LC-3 assembly language program into a program in the LC-3 ISA. Such not-allowed character strings are often referred to as *reserved words*.

NOW, Under21, R2D2, R785, and C3PO are all examples of legitimate LC-3 assembly language labels.

We said we give a label (i.e., a symbolic name) to a memory location if we explicitly refer to it in the program. There are two reasons for explicitly referring to a memory location.

1. The location is the target of a branch instruction (e.g., AGAIN in line 0C). That is, the label AGAIN identifies the location of the instruction that will be executed next if the branch is taken.
2. The location contains a value that is loaded or stored (e.g., NUMBER in line 12, and SIX in line 13).

Note the location AGAIN (identified in line 0C) is specifically referenced by the branch instruction in line 0E.

BRp AGAIN

If the result of ADD R1,R1,#–1 is positive (which results in the P bit being set), then the program branches to the location explicitly referenced as AGAIN to perform another iteration.

The location NUMBER is specifically referenced by the load instruction in line 07. The value stored in the memory location explicitly referenced as NUMBER is loaded into R2.

If a location in the program is not explicitly referenced, then there is no need to give it a label.

!.#.1.$ Comments

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the LC-3 assem- bler. They are identified in the program by semicolons. A semicolon signifies that the rest of the line is a comment and is to be ignored by the assembler. If the semicolon is the first nonblank character on the line, the entire line is ignored. If the semicolon follows the operands of an instruction, then only the comment is ignored by the assembler.

The purpose of comments is to make the program more comprehensible to human reader. Comments help explain a nonintuitive aspect of an instruction a set of instructions. In lines 08 and 09, the comment “Clear R3; it will contain e product” lets the reader know that the instruction on line 08 is initializing

the

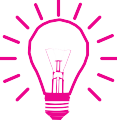
or

th

R3 prior to accumulating the product of the two numbers. While the purpose of line 08 may be obvious to the programmer today, it may not be the case two years from now, after the programmer has written an additional 30,000 instructions and cannot remember why he/she wrote AND R3,R3,#0. It may also be the case that two years from now, the programmer no longer works for the company, and the

company needs to modify the program in response to a product update. If the task is assigned to someone who has never seen the program before, comments go a long way toward helping that person understand the program.

It is important to make comments that provide additional insight and do not just restate the obvious. There are two reasons for this. First, comments that restate the obvious are a waste of everyone’s time. Second, they tend to obscure the comments that say something important because they add clutter to the pro- gram. For example, in line 0D, the comment “Decrement R1” would be a bad idea. It would provide no additional insight to the instruction, and it would add clutter to the page.



Another purpose of comments is to make the visual presentation of a program easier to understand. That is, comments are used to separate pieces of a program from each other to make the program more readable. Lines of code that work together to compute a single result are placed on successive lines, but they are separated from the rest of the program by blank lines. For example, note that lines 0C through 0E, which together form the loop body that is the crux of this computer program, are separated from the rest of the code by lines 0B and 0F. There is nothing on lines 0B and 0F other than the semicolons in the first column. Incidentally, another opportunity to make a program easier to read is the judi- cious use of white space, accomplished by adding extra spaces to a line that are ignored by the assembler—for example, having all the opcodes start in the same

column on the page, whether or not the instruction has a label.

##### !.2.2 Pseudo-Ops (Assembler Directives)

The LC-3 assembler is a program that takes as input a string of characters repre- senting a computer program written in LC-3 assembly language and translates it into a program in the ISA of the LC-3. Pseudo-ops help the assembler perform that task.

The more formal name for a pseudo-op is *assembler directive*. It is called a pseudo-op because, like its Greek root “pseudes” (which means “false”), it does not refer to an operation that will be performed by the program during execution. Rather, the pseudo-op is strictly a message from the assembly language program- mer to the assembler to help the assembler in the assembly process. Once the assembler handles the message, the pseudo-op is discarded. The LC-3 assem- bly language contains five pseudo-ops that we will find useful in our assembly language programming: .ORIG, .FILL, .BLKW, .STRINGZ, and .END. All are easily recognizable by the dot as their first character.

!.2.2.# .ORIG

.ORIG tells the assembler where in memory to place the LC-3 program. In line 05,

.ORIG x3050 says, place the first LC-3 ISA instruction in location x3050. As a result, 0010001000001100 (the translated LD R1,SIX instruction) is put in loca- tion x3050, and the rest of the translated LC-3 program is placed in the subsequent sequential locations in memory. For example, if the program consists of x100 LC- 3 instructions, and .ORIG says to put the first instruction in x3050, the remaining xFF instructions are placed in locations x3051 to x314F.

!.#.#.# .FILL

.FILL tells the assembler to set aside the next location in the program and initial- ize it with the value of the operand. The value can be either a number or a label. In line 13, the ninth location in the resulting LC-3 program is initialized to the value x0006.

!.#.#.$ .BLKW

.BLKW tells the assembler to set aside some number of sequential memory loca- tions (i.e., a **BL**oc**K** of **W**ords) in the program. The actual number is the operand of the .BLKW pseudo-op. In line 12, the pseudo-op instructs the assembler to set aside one location in memory (and, incidentally, to label it NUMBER).

The pseudo-op .BLKW is particularly useful when the actual value of the rand is not yet known. In our example we assumed the number in loca ion NUMBER was 123. How did it get there? A common use of .BLKW is to set aside a location in the program, as we did here, and have another section of code produce the number, perhaps from input from a keyboard (which we cannot know at the time we write the program), and store that value into NUMBER before we

ope t

execute the code in Figure 7.1.

!.#.#.4 .STRINGZ

.STRINGZ tells the assembler to initialize a sequence of *n* + 1 memory locations. The argument is a sequence of *n* characters inside double quotation marks. The first *n* words of memory are initialized with the zero-extended ASCII codes of the corresponding characters in the string. The final word of memory is initialized to

0. The last word, containing x0000, provides a convenient sentinel for processing the string of ASCII codes.

For example, the code fragment

.ORIG x3010

HELLO .STRINGZ "Hello, World!"

would result in the assembler initializing locations x3010 through x301D to the following values:

x3010: x0048 x3011: x0065 x3012: x006C x3013: x006C x3014: x006F x3015: x002C x3016: x0020 x3017: x0057 x3018: x006F x3019: x0072 x301A: x006C x301B: x0064 x301C: x0021 x301D: x0000

!.#.#.& .END

.END tells the assembler it has reached the end of the program and need not even look at anything after it. That is, any characters that come after .END will not be processed by the assembler. *Note:* .END does not stop the program during execution. In fact, .END does not even **exist** at the time of execution. It is simply a delimiter—it marks the end of the program. It is a message from the programmer, telling the assembler where the assembly language program ends.

##### !.#.$ Example: The Character Count Example of Section &.&, Revisited Again!

Now we are ready for a complete example. Let’s consider again the problem of Section 5.5. We wish to write a program that will take a character that is input from the keyboard and count the number of occurrences of that character in a file. As before, we first develop the algorithm by constructing the flowchart. Recall that in Section 6.1, we showed how to decompose the problem system- atically so as to generate the flowchart of Figure 5.16. In fact, the final step of that process in Chapter 6 is the flowchart of Figure 6.3e, which is essentially identical to Figure 5.16. Next, we use the flowchart to write the actual program. This time, however, we enjoy the luxury of not worrying about 0s and 1s and instead write the program in LC-3 assembly language. The program is shown in Figure 7.2.

A few comments about this program: Three times during this program, assis- e in the form of a service call is required of the operating system. In each case, RAP instruction is used. TRAP x23 causes a character to be input from the yboard and placed in R0 (line 0D). TRAP x21 causes the ASCII code in R0 be displayed on the monitor (line 28). TRAP x25 causes the machine to be halted (line 29). As we said before, we will leave the details of how the TRAP

tanc

a T

ke

to

instruction is carried out until Chapter 9.

The ASCII codes for the decimal digits 0 to 9 (0000 to 1001) are x30 to x39. conversion from binary to ASCII is done simply by adding x30 to the binary e of the decimal digit. Line 2D shows the label ASCII used to identify the

The

valu

memory location containing x0030. The LD instruction in line 26 uses it to load x30 into R0, so it can convert the count that is in R2 from a binary value to an ASCII code. That is done by the ADD instruction in line 27. TRAP x21 in line 28 prints the ASCII code to the monitor.

The file that is to be examined starts at address x4000 (see line 2E). Usually, starting address would not be known to the programmer who is writing this ogram since we would want the program to work on many files, not just the one arting at x4000. To accomplish that, line 2E would be replaced with .BLKW 1 and be filled in by some other piece of code that knew the starting address of the desired file before executing the program of Figure 7.2. That situation will be

this

pr

st

discussed in Section 7.4.

01 ;

02 ; Program to count occurrences of a character in a file. 03 ; Character to be input from the keyboard.

1. ; Result to be displayed on the monitor.
2. ; Program works only if no more than 9 occurrences are found. 06 ;

07 ;

08 ; Initialization

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 09  0A | ; | .ORIG | x3000 | |  | |
| 0B |  | AND | R2,R2,#0 | | ; | R2 is counter, initialize to 0 |
| 0C |  | LD | R3,PTR | | ; | R3 is pointer to characters |
| 0D |  | TRAP | x23 | | ; | R0 gets character input |
| 0E  0F | ; | LDR | R1,R3,#0 | | ; | R1 gets the next character |
| 10 | ; Test | character | for end of | | file |  |
| 11 | ; |  |  | |  |  |
| 12 | ; |  |  | |  |  |
| 13 | TEST | ADD | R4,R1,#-4 | | ; | Test for EOT |
| 14 |  | BRz | OUTPUT | | ; | If done, prepare the output |
| 15 | ; |  |  | |  |  |
| 16 | ; Test | character | for match. | | If | a match, increment count. |
| 17 | ; |  |  | |  |  |
| 18 |  | NOT | R1,R1 | |  |  |
| 19 |  | ADD | R1,R1,#1 | | ; | R1 <-- -R1 |
| 1A  1B  1C  1D | ; | ADD  BRnp ADD | R1,R1,R0 GETCHAR R2,R2,#1 | | ;  ; | R1 <-- R0-R1. If R1=0, a match!  If no match, do not increment |
| 1E  1F  20 | ; Get next character from  ;  GETCHAR ADD R3,R3,#1 | | | the file  ; Increment the pointer | | |
| 21 | LDR R1,R3,#0 | | | ; R1 gets the next character to test | | |
| 22 | BRnzp TEST | | |  | | |
| 23 | ; | | |  | | |
| 24 | ; Output the count. | | |  | | |
| 25 | ; | | |  | | |
| 26 | OUTPUT LD R0,ASCII | | | ; Load the ASCII template | | |
| 27 | ADD R0,R0,R2 | | | ; Convert binary to ASCII | | |
| 28 | TRAP x21 | | | ; ASCII code in R0 is displayed | | |
| 29  2A  2B  2C  2D  2E  2F | TRAP x25  ;  ; Storage for pointer and  ;  ASCII .FILL x0030 PTR .FILL x4000  .END | | | ; Halt machine ASCII template | | |

**Figure !.# The assembly language program to count occurrences of a character.**

## !.$ The Assembly Process

##### !.$.1 Introduction

Before an LC-3 assembly language program can be executed, it must first be translated into a machine language program, that is, one in which each instruction is in the LC-3 ISA. It is the job of the LC-3 assembler to perform that translation. If you have available an LC-3 assembler, you can cause it to translate your assembly language program into a machine language program by executing an appropriate command. In the LC-3 assembler that is generally available via the web, that command is *assemble*, and it requires as an argument the filename of your assembly language program. For example, if the filename is solution1.asm,

then

assemble solution1.asm outfile

produces the file outfile, which is in the ISA of the LC-3. It is necessary to check with your instructor for the correct command line to cause the LC-3 assembler to produce a file of 0s and 1s in the ISA of the LC-3.

##### !.$.# A Two-Pass Process

In this section, we will see how the assembler goes through the process of trans- lating an assembly language program into a machine language program. We will use as our input to the process the assembly language program of Figure 7.2.

You remember that there is in general a one-to-one correspondence between instructions in an assembly language program and instructions in the final machine language program. We could try to perform this translation in one pass through the assembly language program. Starting from the top of Figure 7.2, the assembler discards lines 01 to 09, since they contain only comments. Comments are strictly for human consumption; they have no bearing on the translation pro- cess. The assembler then moves on to line 0A. Line 0A is a pseudo-op; it tells the assembler that the machine language program is to start at location x3000. The assembler then moves on to line 0B, which it can easily translate into LC-3 machine code. At this point, we have

x3000: 0101010010100000

The LC-3 assembler moves on to translate the next instruction (line 0C). Unfor- tunately, it is unable to do so since it does not know the meaning of the symbolic address PTR. At this point the assembler is stuck, and the assembly process fails. To prevent this from occurring, the assembly process is done in two com- plete passes (from beginning to .END) through the entire assembly language program. The objective of the first pass is to identify the actual binary addresses corresponding to the symbolic names (or labels). This set of correspondences is known as the *symbol table*. In pass 1, we construct the symbol table. In pass 2, we translate the individual assembly language instructions into their corresponding

machine language instructions.

Thus, when the assembler examines line 0C for the purpose of translating

LD R3,PTR

* 1. The Assembly Process **241**

during the second pass, it already knows that PTR is the symbolic address of memory location x3013 (from the first pass). Thus, it can easily translate line 0C to

x3001: 0010011000010001

The problem of not knowing the 16-bit address corresponding to PTR no longer exists.

##### !.$.$ The First Pass: Creating the Symbol Table

For our purposes, the symbol table is simply a correspondence of symbolic names with their 16-bit memory addresses. We obtain these correspondences by pass- ing through the assembly language program once, noting which instruction is assigned to which memory location, and identifying each label with the memory address of its assigned entry.

Recall that we provide labels in those cases where we have to refer to a loca- tion, either because it is the target of a branch instruction or because it contains data that must be loaded or stored. Consequently, if we have not made any pro- gramming mistakes, and if we identify all the labels, we will have identified all the symbolic addresses used in the program.

The preceding paragraph assumes that our entire program exists between our

.ORIG and .END pseudo-ops. This is true for the assembly language program of Figure 7.2. In Section 7.4, we will consider programs that consist of multi- ple parts, each with its own .ORIG and .END, wherein each part is assembled separately.

The first pass starts, after discarding the comments on lines 01 to 09, by ting (line 0A) that the first instruction will be assigned to address x3000. We ep track of the location assigned to each instruction by means of a location counter (LC). The LC is initialized to the address specified in .ORIG, that is,

no

ke

x3000.

The assembler examines each instruction in sequence and increments the LC once for each assembly language instruction. If the instruction examined contains a label, a symbol table entry is made for that label, specifying the current con- tents of LC as its address. The first pass terminates when the .END pseudo-op is reached.

The first instruction that has a label is at line 13. Since it is the fifth instruction he program and since the LC at that point contains x3004, a symbol table entry constructed thus:

Address

Symbol

in t

is

x3004

TEST

The second instruction that has a label is at line 20. At this point, the LC has been incremented to x300B. Thus, a symbol table entry is constructed, as follows:

Address

Symbol

x300B

GETCHAR

At the conclusion of the first pass, the symbol table has the following entries:

Address

Symbol

|  |  |
| --- | --- |
| TEST | x3004 |
| GETCHAR | x300B |
| OUTPUT | x300E |
| ASCII | x3012 |
| PTR | x3013 |

##### !.$.4 The Second Pass: Generating the Machine Language Program

The second pass consists of going through the assembly language program a sec- ond time, line by line, this time with the help of the symbol table. At each line, the assembly language instruction is translated into an LC-3 machine language instruction.

Starting again at the top, the assembler again discards lines 01 through 09 because they contain only comments. Line 0A is the .ORIG pseudo-op, which the assembler uses to initialize LC to x3000. The assembler moves on to line 0B and produces the machine language instruction 0101010010100000. Then the assembler moves on to line 0C.

This time, when the assembler gets to line 0C, it can completely assemble the instruction since it knows that PTR corresponds to x3013. The instruction is LD, which has an opcode encoding of 0010. The destination register (DR) is R3, that is, 011.

The only part of the LD instruction left to do is the PCoffset. It is computed as follows: The assembler knows that PTR is the label for address x3013 and that the incremented PC is LC+1, in this case x3002. Since PTR (x3013) must be the sum of the incremented PC (x3002) and the sign-extended PCoffset, PCoffset must be x0011. Putting this all together, the assembler sets x3001 to 0010011000010001 and increments the LC to x3002.

*Note:* In order to use the LD instruction, it is necessary that the source of the load, in this case the address whose label is PTR, is not more than +256 or

−255 memory locations from the LD instruction itself. If the address of PTR had been greater than LC+1+255 or less than LC+1−256, then the offset would not fit in bits [8:0] of the instruction. In such a case, an assembly error would have occurred, preventing the assembly process from finishing successfully. For- tunately, PTR is close enough to the LD instruction, so the instruction assembled correctly.

The second pass continues. At each step, the LC is incremented and the location specified by LC is assigned the translated LC-3 instruction or, in the case of .FILL, the value specified. When the second pass encounters the .END pseudo-op, assembly terminates.

The resulting translated program is shown in Figure 7.3.

That process was, on a good day, merely tedious. Fortunately, you do not have to do it for a living—the LC-3 assembler does that. And, since you now know the

* 1. Beyond the Assembly of a Single Assembly Language Program **243**

Binary

Address

|  |  |
| --- | --- |
| x3000 | 0011000000000000  0101010010100000 |
| x3001 | 0010011000010001 |
| x3002 | 1111000000100011 |
| x3003 | 0110001011000000 |
| x3004 | 0001100001111100 |
| x3005 | 0000010000001000 |
| x3006 | 1001001001111111 |
| x3007 | 0001001001000000 |
| x3008 | 1001001001111111 |
| x3009 | 0000101000000001 |
| x300A | 0001010010100001 |
| x300B | 0001011011100001 |
| x300C | 0110001011000000 |
| x300D | 0000111111110110 |
| x300E | 0010000000000011 |
| x300F | 0001000000000010 |
| x3010 | 1111000000100001 |
| x3011 | 1111000000100101 |
| x3012 | 0000000000110000 |
| x3013 | 0100000000000000 |
|  |  |

**Figure !.$ The machine language program for the assembly language program of Figure !.#.**

LC-3 assembly language, there is no need to program in machine language. Now we can write our programs symbolically in LC-3 assembly language and invoke the LC-3 assembler to create the machine language versions that can execute on an LC-3 computer.

## !.4 Beyond the Assembly of a Single Assembly Language Program

Our purpose in this chapter has been to take you up one more step from the ISA of the computer and introduce assembly language. Although it is still quite a large step from C or C++, assembly language does, in fact, save us a good deal of pain. We have also shown how a rudimentary two-pass assembler actually works to translate an assembly language program into the machine language of the LC-3 ISA.

There are many more aspects to sophisticated assembly language program- ming that go well beyond an introductory course. However, our reason for teaching assembly language is not to deal with its sophistication, but rather to show its innate simplicity. Before we leave this chapter, however, there are a few additional highlights we should explore.

##### !.4.1 The Executable Image

When a computer begins execution of a program, the entity being executed is called an *executable image*. The executable image is created from modules often created independently by several different programmers. Each module is trans- lated separately into an object file. We have just gone through the process of performing that translation ourselves by mimicking the LC-3 assembler. Other modules, some written in C perhaps, are translated by the C compiler. Some mod- ules are written by users, and some modules are supplied as library routines by the operating system. Each object file consists of instructions in the ISA of the computer being used, along with its associated data. The final step is to combine (i.e., *link*) all the object modules together into one executable image. During exe- cution of the program, the FETCH, DECODE, … instruction cycle is applied to instructions in the executable image.

##### !.4.# More than One Object File

It is very common to form an executable image from more than one object file. In fact, in the real world, where most programs invoke libraries provided by the operating system as well as modules generated by other programmers, it is much more common to have multiple object files than a single one.

A case in point is our example character count program. The program counts number of occurrences of a character in a file. A typical application could ly have the program as one module and the input data file as another. If this e the case, then the starting address of the file, shown as x4000 in line 2E of ure 7.2, would not be known when the program was written. If we replace line with

the

easi

wer

Fig

2E

PTR .FILL STARTofFILE

then the program of Figure 7.2 will not assemble because there will be no symbol table entry for STARTofFILE. What can we do?

If the LC-3 assembly language, on the other hand, contained the pseudo-op

.EXTERNAL, we could identify STARTofFILE as the symbolic name of an address that is not known at the time the program of Figure 7.2 is assembled. This would be done by the following line

.EXTERNAL STARTofFILE,

which would send a message to the LC-3 assembler that the absence of label STARTofFILE is not an error in the program. Rather, STARTofFILE is a label in some other module that will be translated independently. In fact, in our case, it will be the label of the location of the first character in the file to be examined by our character count program.

If the LC-3 assembly language had the pseudo-op .EXTERNAL, and if we had designated STARTofFILE as .EXTERNAL, the LC-3 assembler would be able to create a symbol table entry for STARTofFILE, and instead of assigning it an address, it would mark the symbol as belonging to another module. At *link*

*time,* when all the modules are combined, the linker (the program that manages the “combining” process) would use the symbol table entry for STARTofFILE in another module to complete the translation of our revised line 2E.

In this way, the .EXTERNAL pseudo-op allows references by one module to symbolic locations in another module without a problem. The proper translations are resolved by the linker.

**Exercises**

**!.1** An assembly language program contains the following two instructions.

The assembler puts the translated version of the LDI instruction that follows into location x3025 of the object module. After assembly is complete, what is in location x3025?

PLACE .FILL x45A7

LDI R3, PLACE

**!.#** An LC-3 assembly language program contains the instruction:

ASCII LD R1, ASCII

The symbol table entry for ASCII is x4F08. If this instruction is executed during the running of the program, what will be contained in R1 immediately after the instruction is executed?

**!.$** What is the problem with using the string AND as a label?

**!.4** Create the symbol table entries generated by the assembler when translating the following routine into machine code:

|  |  |  |
| --- | --- | --- |
|  | .ORIG ST  ST | x301C R3, SAVE3  R2, SAVE2 |
| TEST | AND IN  BRz | R2, R2, #0  TEST |
|  | ADD  BRn | R1, R0, #-10 FINISH |
|  | ADD | R1, R0, #-15 |
|  | NOT  BRn | R1, R1 FINISH |
|  | HALT |  |
| FINISH | ADD | R2, R2, #1 |
|  | HALT |  |
| SAVE3 | .FILL | X0000 |
| SAVE2 | .FILL  .END | X0000 |

**!.&** *a.* What does the following program do?

|  |  |  |
| --- | --- | --- |
|  | .ORIG | x3000 |
|  | LD | R2, ZERO |
|  | LD | R0, M0 |
|  | LD | R1, M1 |
| LOOP | BRz | DONE |
|  | ADD | R2, R2, R0 |
|  | ADD | R1, R1, -1 |
|  | BR | LOOP |
| DONE | ST | R2, RESULT |
|  | HALT |  |
| RESULT | .FILL | x0000 |
| ZERO | .FILL | x0000 |
| M0 | .FILL | x0004 |
| M1 | .FILL | x0803 |
|  | .END |  |

*b.* What value will be contained in RESULT after the program runs to completion?

**!.’** Our assembler has crashed, and we need your help! Create a symbol table for the following program, and assemble the instructions at labels A, B, and D.

|  |  |  |  |
| --- | --- | --- | --- |
| A | .ORIG AND LD | x3000 R0, R0, R1, E | #0 |
|  | AND | R2, R1, | #1 |
|  | BRp | C |  |
| B | ADD | R1, R1, | #-1 |
| C | ADD | R0, R0, | R1 |
|  | ADD | R1, R1, | #-2 |
| D | BRp | C |  |
|  | ST | R0, F |  |
|  | TRAP | x25 |  |
| E | .BLKW | 1 |  |
| F | .BLKW | 1 |  |
|  | .END |  |  |

You may assume another module deposits a positive value into E before the module executes. In 15 words or fewer, what does this program do?

**!.!** Write an LC-3 assembly language program that counts the number of 1s in the value stored in R0 and stores the result into R1. For example, if R0 contains 0001001101110000, then after the program executes, the result stored in R1 would be 0000 0000 0000 0110.

**!.(** An engineer is in the process of debugging a program she has written. She is looking at the following segment of the program and decides to place a breakpoint in memory at location 0xA404. Starting with the PC = 0xA400, she initializes all the registers to zero and runs the program until the breakpoint is encountered.

Code Segment:

...

|  |  |  |
| --- | --- | --- |
| 0xA400 THIS1  0xA401 THIS2  0xA402 THIS3 | LEA LD  LDI | R0, THIS1 R1, THIS2  R2, THIS5 |
| 0xA403 THIS4 | LDR | R3, R0, #2 |
| 0xA404 THIS5  ... | .FILL | xA400 |

Show the contents of the register file (in hexadecimal) when the breakpoint is encountered.

**!.)** What is the purpose of the .END pseudo-op? How does it differ from the HALT instruction?

**!.10** The following program fragment has an error in it. Identify the error and explain how to fix it.

ADD R3, R3, #30

ST R3, A HALT

A .FILL #0

Will this error be detected when this code is assembled or when this code is run on the LC-3?

**!.11** The LC-3 assembler must be able to convert constants represented in ASCII into their appropriate binary values. For instance, x2A translates into 00101010 and #12 translates into 00001100. Write an LC-3 assembly language program that reads a decimal or hexadecimal constant from the keyboard (i.e., it is preceded by a # character signifying it is a decimal, or x signifying it is hex) and prints out the binary representation. Assume the constants can be expressed with no more than two decimal or hex digits.

**!.1#** What does the following LC-3 program do?

|  |  |  |
| --- | --- | --- |
|  | .ORIG AND | x3000  R5, R5, #0 |
| AND | R3, R3, #0 |
| ADD | R3, R3, #8 |
| LDI | R1, A |
| ADD | R2, R1, #0 |
| AG | ADD | R2, R2, R2 |
|  | ADD | R3, R3, #-1 |
|  | BRnp LD | AG R4, B |
|  | AND | R1, R1, R4 |
|  | NOT | R1, R1 |
|  | ADD | R1, R1, #1 |
|  | ADD  BRnp | R2, R2, R1 NO |
|  | ADD | R5, R5, #1 |
| NO | HALT |  |
| B | .FILL | xFF00 |
| A | .FILL  .END | x4000 |

**!.1$** The following program adds the values stored in memory locations A, B, and C and stores the result into memory. There are two errors in the code. For each, describe the error and indicate whether it will be detected at assembly time or at run time.

Line No.

|  |  |  |
| --- | --- | --- |
| 1 |  | .ORIG x3000 |
| 2 | ONE | LD R0, A |
| 3 |  | ADD R1, R1, R0 |
| 4 | TWO | LD R0, B |
| 5 |  | ADD R1, R1, R0 |
| 6 | THREE | LD R0, C |
| 7 |  | ADD R1, R1, R0 |
| 8 |  | ST R1, SUM |
| 9 |  | TRAP x25 |
| 10 | A | .FILL x0001 |
| 11 | B | .FILL x0002 |
| 12 | C | .FILL x0003 |
| 13 | D | .FILL x0004 |
| 14 |  | .END |

**!.14** *a.* Assemble the following program:

.ORIG x3000

STI R0, LABEL OUT

HALT

LABEL .STRINGZ "%"

.END

1. The programmer intended the program to output a % to the monitor and then halt. Unfortunately, the programmer got confused about the semantics of each of the opcodes (i.e., exactly what function is

carried out by the LC-3 in response to each opcode). Replace exactly **one** opcode in this program with the correct opcode to make the program work as intended.

1. The original program from part *a* was executed. However, execution exhibited some very strange behavior. The strange behavior was in part due to the programming error and in part due to the fact that the value in R0 when the program started executing was x3000. Explain what the strange behavior was and why the program behaved that way.

**!.1&** The following is an LC-3 program that performs a function. Assume a sequence of integers is stored in consecutive memory locations, one integer per memory location, starting at the location x4000. The sequence terminates with the value x0000. What does the following program do?

|  |  |  |
| --- | --- | --- |
|  | .ORIG | x3000 |
|  | LD | R0, NUMBERS |
|  | LD | R2, MASK |
| LOOP | LDR | R1, R0, #0 |
|  | BRz | DONE |
|  | AND | R5, R1, R2 |
|  | BRz | L1 |
|  | BRnzp | NEXT |
| L1 | ADD | R1, R1, R1 |
|  | STR | R1, R0, #0 |
| NEXT | ADD | R0, R0, #1 |
|  | BRnzp | LOOP |
| DONE | HALT |  |
| NUMBERS | .FILL | x4000 |
| MASK | .FILL | x8000 |
| .END |  |  |

**!.1’** Assume a sequence of nonnegative integers is stored in consecutive memory locations, one integer per memory location, starting at location x4000. Each integer has a value between 0 and 30,000 (decimal). The sequence terminates with the value −1 (i.e., xFFFF).

What does the following program do?

|  |  |  |
| --- | --- | --- |
|  | .ORIG AND | x3000  R4, R4, #0 |
|  | AND | R3, R3, #0 |
|  | LD | R0, NUMBERS |
| LOOP | LDR | R1, R0, #0 |
|  | NOT | R2, R1 |
|  | BRz | DONE |
|  | AND | R2, R1, #1 |
|  | BRz | L1 |
|  | ADD | R4, R4, #1 |
|  | BRnzp | NEXT |
| L1 | ADD | R3, R3, #1 |
| NEXT | ADD | R0, R0, #1 |
|  | BRnzp | LOOP |
| DONE | TRAP | x25 |
| NUMBERS | .FILL | x4000 |
|  | .END |  |

**!.1!** Suppose you write two separate assembly language modules that you expect to be combined by the linker. Each module uses the label AGAIN, and neither module contains the pseudo-op .EXTERNAL AGAIN.

Is there a problem using the label AGAIN in both modules? Why or why not?

**!.1(** The following LC-3 program compares two character strings of the same length. The source strings are in the .STRINGZ form. The first string starts at memory location x4000, and the second string starts at memory location x4100. If the strings are the same, the program terminates with the value 1 in R5. Insert instructions at (a), (b), and (c) that will complete the program.

.ORIG x3000

LD R1, FIRST

LD R2, SECOND

AND R0, R0, #0

LOOP (a)

LDR R4, R2, #0

BRz NEXT

ADD R1, R1, #1

ADD R2, R2, #1

(b)

(c) ADD R3, R3, R4

BRz LOOP

AND R5, R5, #0

BRnzp DONE

NEXT AND R5, R5, #0 ADD R5, R5, #1

DONE TRAP x25 FIRST .FILL x4000 SECOND .FILL x4100

.END

**!.1)** When the following LC-3 program is executed, how many times will the instruction at the memory address labeled LOOP execute?

|  |  |  |
| --- | --- | --- |
|  | .ORIG LEA | x3005 R2, DATA |
|  | LDR | R4, R2, #0 |
| LOOP | ADD  BRzp | R4, R4, #-3 LOOP |
|  | TRAP | x25 |
| DATA | .FILL  .END | x000B |

**!.#0** LC-3 assembly language modules (a) and (b) have been written by different programmers to store x0015 into memory location x4000. What is fundamentally different about their approaches?

|  |  |  |
| --- | --- | --- |
| *a.* | .ORIG AND | x5000  R0, R0, #0 |
|  | ADD | R0, R0, #15 |
|  | ADD | R0, R0, #6 |
|  | STI | R0, PTR |
| PTR | HALT  .FILL | x4000 |
|  | .END |  |
| *b.* | .ORIG  .FILL  .END | x4000 x0015 |

**!.#1** Assemble the following LC-3 assembly language program.

.ORIG x3000

|  |  |  |
| --- | --- | --- |
|  | AND | R0, R0, #0 |
| ADD | R2, R0, #10 |
| LD | R1, MASK |
| LD | R3, PTR1 |
| LOOP | LDR | R4, R3, #0 |
|  | AND  BRz | R4, R4, R1 NEXT |
|  | ADD | R0, R0, #1 |
| NEXT | ADD | R3, R3, #1 |
|  | ADD | R2, R2, #-1 |
|  | BRp STI | LOOP R0, PTR2 |
|  | HALT |  |
| MASK | .FILL | x8000 |
| PTR1 | .FILL | x4000 |
| PTR2 | .FILL  .END | x5000 |

What does the program do (in no more than 20 words)?

**!.##** The LC-3 assembler must be able to map an instruction’s mnemonic opcode into its binary opcode. For instance, given an ADD, it must generate the binary pattern 0001. Write an LC-3 assembly language

program that prompts the user to type in an LC-3 assembly language opcode and then displays its binary opcode. If the assembly language opcode is invalid, it displays an error message.

**!.#$** The following LC-3 program determines whether a character string is a palindrome or not. A palindrome is a string that reads the same

backwards as forwards. For example, the string “racecar” is a palindrome. Suppose a string starts at memory location x4000 and is in the .STRINGZ format. If the string is a palindrome, the program terminates with the value 1 in R5. If not, the program terminates with the value 0 in R5.

Insert instructions at (a)–(e) that will complete the program.

.ORIG x3000 LD R0, PTR

ADD R1, R0, #0

AGAIN LDR R2, R1, #0 BRz CONT

ADD R1, R1, #1 BRnzp AGAIN

CONT (a) LOOP LDR R3, R0, #0

(b) NOT R4, R4

ADD R4, R4, #1

ADD R3, R3, R4

BRnp NO

(c)

(d) NOT R2, R0

ADD R2, R2, #1

ADD R2, R1, R2

BRnz YES

(e)

YES AND R5, R5, #0 ADD R5, R5, #1

BRnzp DONE

NO AND R5, R5, #0

DONE HALT

PTR .FILL x4000

.END

**!.#4** We want the following program fragment to shift R3 to the left by four bits, but it has an error in it. Identify the error and explain how to fix it.

.ORIG x3000

AND R2, R2, #0

ADD R2, R2, #4 LOOP BRz DONE

ADD R2, R2, #-1

ADD R3, R3, R3

BR LOOP

DONE HALT

.END

**!.#&** What does the pseudo-op .FILL xFF004 do? Why?

**!.#’** Recall the assembly language program of Exercise 7.6. Consider the following program:

.ORIG x3000

AND R0, R0, #0

|  |  |  |
| --- | --- | --- |
| D | LD | R1, A |
|  | AND  BRp | R2, R1, #1 B |
| E | ADD | R1, R1, #-1 |
| B | ADD | R0, R0, R1 |
| F | ADD  BRp ST | R1, R1, #-2 B  R0, C |
|  | TRAP | x25 |
| A | .BLKW | 1 |
| C | .BLKW  .END | 1 |

The assembler translates both assembly language programs into machine language programs. What can you say about the two resulting machine language programs?

**!.#!** Consider the following LC-3 assembly language program:

.ORIG x3000 AND R2, R2, #0 AND R6, R6, #0 ADD R2, R2, #1

TOP ADD R3, R2, #0 ADD R4, R1, #0

SEARCH ADD R3, R3, R3

ADD R4, R4, #-1 BRp SEARCH

AND R5, R3, R0 BRz NEXT

ADD R6, R6, R2

NEXT ADD R2, R2, R2 BRzp TOP

END ST R6, RESULT HALT

RESULT .BLKW 1

.END

What does it do (in 20 words or fewer)? Please be BRIEF but PRECISE. You can assume that some of the registers will already contain numbers that are relevant to the program.

What is the function of R0? For what range of input values does the program function as you’ve described above?

What is the function of R1? For what range of input values does the program function as you’ve described above?

What is the function of R6? For what range of input values does the program function as you’ve described above?

\***!.#(** Consider the following program:

.ORIG x3000 LD R0, A

LD R1, B BRz DONE

(a)

(b) BRnzp AGAIN

DONE ST R0, A

HALT

* 1. .FILL x0--- (c)
  2. .FILL x0001

.END

The program uses only R0 and R1. Note lines (a) and (b) indicate two missing instructions. Complete line (c). Note also that one of the instructions in the program must be labeled AGAIN, and that label is missing.

After execution of the program, the contents of A is x1800.

During execution, we examined the computer during each clock cycle and recorded some information for certain clock cycles, producing the table shown below. The table is ordered by the cycle number in which the information was collected. Note that each memory access takes five clock cycles.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Cycle Number** | **State Number** | **Control Signals** | | | |
| 1 | 18 | LD.MAR: LD.PC: |  | LD.REG: PCMUX: | GateMDR: GatePC: |
|  | 0 | LD.MAR:  LD.PC: |  | LD.REG:  LD.CC: | BEN |
|  |  | LD.REG:  GateALU: | 1 | DR: 000  GateMARMUX: | GateMDR: |
| 57 | 1 | LD.MAR: LD.REG: |  | ALUK: DR: | GateALU: GatePC: |
| 77 | 22 | ADDR1MUX: LD.PC: |  | ADDR2MUX: LD.MAR: | PCMUX: |
| 101 | 15 |  | | | |

Fill in the missing instructions in the program, and complete the program by labeling the appropriate instruction AGAIN. Also, fill in the missing information in the table.

Given values for A and B, what does the program do?

\***!.#)** An LC-3 program is executing on the LC-3 simulator when a breakpoint is encountered, and the simulator stops. At that point, **the contents of several registers are as shown in the first row of the table**. After the

run button is subsequently pushed, the next four instructions that are executed, none of which are an STI or LDI, produce the values shown in the table, **two rows of the table per instruction executed**. The first row of each pair shows the contents after the fetch phase of the corresponding instruction, and the second row of each pair after that instruction completes.

Note that some values are missing and are presented by letters A, B, C, D, E, F, G, H, I, and J.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PC | MAR | MDR | IR | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x1800 | x7FFF | x2211 | xBFFE | x31FF | x2233 | x5177 | x3211 | x21FF | x5233 | x3177 | x2211 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A  A | x1800  x1800 | B  B | B  B | x31FF  x31FF | x2233  x2233 | x5177  x5177 | x3211  x3211 | x21FF  x21FF | x5233  C | x3177  x3177 | x2211  x2211 |
|  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| D  D | A  F | E  G | E  E | x31FF  x31FF | x2233  x2233 | x5177  x5177 | x3211  x3211 | x21FF  x21FF | C  C | x3177  x3177 | x2211  x2211 |
|  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| H  F | D  D | I  I | I  I | x31FF  x31FF | x2233  x2233 | x5177  x5177 | x3211  x3211 | x21FF  x21FF | C  C | x3177  x3177 | x2211  x2211 |
|  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A  A | F  F | J  J | J  J | x31FF  x31FF | x2233  x2233 | x5177  x5177 | x3211  x3211 | x21FF  x223A | C  C | x3177  x3177 | x2211  x2211 |
|  |  |  |  |  |  |  |  |  |  |  |  |

Your job: Determine the values of A, B, C, D, E, F, G, H, I, and J. Note that some of the values may be identical.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | E |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| F | G | H | I | J |
|  |  |  | x .F.. |  |

\***!.$0** 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.

x3FFB

|  |  |
| --- | --- |
| x0005 |  |
| x0004 |
| x0008 |
|  |
| x0007 |
| x0006 |

R6

x3FFF

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

**!.$1** Memory locations x5000 to x5FFF contain 2’s complement integers.

What does the following program do?

.ORIG x3000 LD R1, ARRAY LD R2, LENGTH AND R3, R3, #0

AGAIN LDR R0, R1, #0

AND R0, R0, #1 BRz SKIP

ADD R3, R3, #1 SKIP ADD R1, R1, #1

ADD R2, R2, #-1 BRp AGAIN

HALT

ARRAY .FILL x5000 LENGTH .FILL x1000

.END

**!.$#** Consider the following semi-nonsense assembly language program:

line 1: .ORIG x8003

line 2: AND R1,R1,#0

line 3: ADD R0,R1,#5

line 4: ST R1,B

line 5: LD R1,A

line 6: BRz SKIP

**line 7: ST R0,B** line 8: SKIP TRAP x25 line 9: A .BLKW #7

**line 10: B .FILL #5**

line 11: BANNER .STRINGZ "We are done!" line 12: C .FILL x0

line 13: .END

A separate module will store a value in A before this program executes. Construct the symbol table.

Show the result of assembly of lines 5 through 7 above. *Note:* the instruction at line 8 has already been assembled for you.

x8006 x8007 x8008 x8009

1 1 1 1 0 0 0 0 0 0 1 0 0 1 0

1

Note that two different things could cause location B to contain the value 5: the contents of line 7 or the contents of line 10. Explain the difference

between line 7 causing the value 5 to be in location B and line 10 causing the value 5 to be in location B.

\***!.$$** We have a program with some missing instructions, and we have a table consisting of some information and some missing information associated with five specific clock cycles of the program’s execution. Your job is to complete both!

Insert the missing instructions in the program and the missing information in the table. Cycle numbering starts at 1. That is, cycle 1 is the first clock cycle of the processing of LD R0,A. Note that we have not said anything about the number of clock cycles a memory access takes. You do have enough information to figure that out for yourself. Note that we are asking for the value of the registers DURING each clock cycle.

.ORIG x3000 LD R0, A

LD R1, B NOT R1, R1

ADD R1, R1, #1 AND R2, R2, #0

AGAIN (a)

(b) BRnzp AGAIN

DONE ST R2, C

HALT

* + 1. .FILL #5
    2. .FILL (c)
    3. .BLKW #1

.END

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Cycle Number** | **State Number** | **Information** | | | | | |
|  |  |  | | | | | |
|  |  | LD.REG:  LD.CC: | 1 | DRMUX:  GateALU: |  | GateMDR:  GatePC: |  |
|  |
|  |  |  | | | | | |
| 16 | 35 |  | | | | | |
| LD.MDR:  LD.IR: |  | MDR: |  | IR: |  |
|  | | | | | |
| 50 |  |  | | | | | |
|  | LD.REG:  BUS: | 1  x0001 | MDR: | x\_4A\_ | DRMUX:  GateMDR: |  |
|  |
|  |  | | | | | |
| 57 | 1 |  | | | | | |
| PC: IR: x\_040 GateALU:  BUS: x0003 GatePC: | | | | |  |
|  |
|  | | | | | |
|  | 23 | ADDR1MUX: ADDR2MUX: | | | | | |
|  |
| LD.PC: 1 PC: x3008 PCMUX: | | | | | ADDER |
|  |  |

What is stored in C at the end of execution for the specific operands given in memory locations A and B?

Actually, the program was written by a student, so as expected, he did not get it quite right. Almost, but not quite! Your final task on this problem is to examine the code, figure out what the student was trying to

do, and point out where he messed up and how you would fix it. It is not necessary to write any code, just explain briefly how you would fix it.

What was the student trying to do?

How did the student mess up? How would you fix his program?

**!.$4** It is often useful to find the midpoint between two values. For this problem, assume A and B are both even numbers, and A is less than B. For example, if A = 2 and B = 8, the midpoint is 5. The following program finds the midpoint of two even numbers A and B by continually incrementing the smaller number and decrementing the larger number.

You can assume that A and B have been loaded with values before this program starts execution.

Your job: Insert the missing instructions.

.ORIG x3000 LD R0,A

LD R1,B

X (a)

(b) ADD R2,R2,R1

(c) ADD R1,R1,#-1

(d) BRnzp X

DONE ST R1,C TRAP x25

1. .BLKW 1
2. .BLKW 1
3. .BLKW 1

.END

\***!.$&** The program stored in memory locations x3000 to x3007 loads a value from memory location x3100, then does some processing, and then stores a result in memory location x3101. Following is an incomplete specification of the program. Your job: Complete the specification of the program.

|  |  |  |
| --- | --- | --- |
| Address  x3000 | Contents  0101 001 001 1 00000 | Assembly code  AND R1, R1, #0 |
| x3001 | 0010 000 | LD R0, x3100 |
| x3002 | 0000 110 000000011 | BRnz x3006 |
| x3003 | 0001 | ADD |
| x3004 |  |  |
| x3005 | 0000 111 |  |
| x3006 | 0011 001 | ST R1, x3101 |
| x3007 | 1111 0000 0010 0101 | HALT |
|  |  |  |

To help you in this process, we have taken a snapshot of part of the state of the machine before the first instruction executes and at several

instruction boundaries thereafter, that is, after a number of instructions executed. Part of the snapshot is shown below. Your job is to complete the snapshot. Note that the program enters the TRAP x25 service routine after executing 17 instructions. Therefore, some instructions must execute more than once.

Note that in the following table, some entries are designated xxxx. You do not have to fill in those entries. Also, you can ignore snapshots for any instructions that are not listed in the table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction #  Initial | PC  x3000 | MAR  xxxx | MDR  xxxx | R0  xxxx | R1  xxxx |
| 1 | x3001 | xxxx | xxxx | xxxx |  |
| 2 | x3002 |  |  |  |  |
| 3 | x3003 | xxxx | xxxx |  |  |
| 4 | x3004 |  | x1240 |  |  |
| 5 | x3005 | xxxx | xxxx | x0002 |  |
| 9 | x3005 | xxxx | xxxx | x0001 |  |
| 13 | x3005 | xxxx | xxxx | x0000 |  |
| 14 | x3002 | xxxx | xxxx |  |  |
| 15 | x3006 | xxxx | xxxx |  |  |
| 16 | x3007 |  |  |  |  |
| 17 | xxxx | xxxx | xxxx |  |  |
|  |  |  |  |  |  |

**!.$’** The modulo operator (A mod B) is the remainder one gets when dividing A by B. For example, 10 mod 5 is 0, 12 mod 7 is 5.

The following program is supposed to perform A mod B, where A is in x3100 and B is in x3101. The result should be stored at location x3200. However, the programmer made a serious mistake, so the program does not work. You can assume that A and B are both positive integers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | .ORIG x3000 LD R3, L2 |  | ;  ; | Line 1  2 |
| LDR R0, R3, | #0 | ; | 3 |
| LDR R1, R3, | #1 | ; | 4 |
| NOT R2, R1 |  | ; | 5 |
| ADD R2, R2, | #1 | ; | 6 |
| L1 | ADD R0, R0, | R2 | ; | 7 |
|  | BRzp L1 |  | ; | 8 |
|  | ADD R0, R0, | R1 | ; | 9 |
|  | ST R0, L3 |  | ; | 10 |
|  | HALT |  | ; | 11 |
| L2 | .FILL x3100 |  | ; | 12 |
| L3 | .FILL x3200 |  | ; | 13 |
|  | .END |  | ; | 14 |

After the instruction at line 6 has executed, what are the contents of R0, R1, and R2? *Note:* The correct answer in each case is one of the following: A, −A, B, −B, 0, 1, −1.

There is a bug in the program. What line is it in, and what should the correct instruction be?

\***!.$!** During the processing of an LC-3 program by the data path we have been using in class, the computer stops due to a breakpoint set at x3000. The contents of certain registers and memory locations at that time are as follows:

R2 through R7: x0000 M[x3000]: x1263 M[x3003]: x0000

The LC-3 is restarted and executes exactly four instructions. To accomplish this, a number of clock cycles are required. In 15 of those clock cycles, the bus must be utilized. The following table lists those 15 clock cycles in sequential order, along with the values that are gated onto the LC-3 bus in each.

1st:

|  |
| --- |
| BUS  x3000 |
| x1263 |
| x009A |
| x3001 |
| xA000 |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| x3003 |
| x1263 |
| x009D |
|  |

2nd:

3rd:

4th:

5th:

6th:

7th:

8th:

9th:

10th:

11th:

12th:

13th:

14th:

15th:

Fill in the missing entries above.

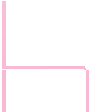
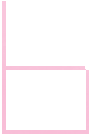
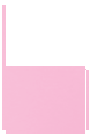
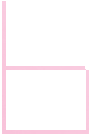
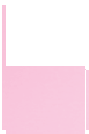
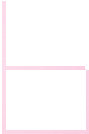
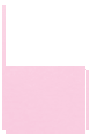
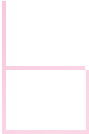
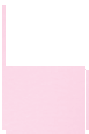
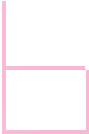
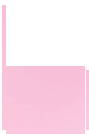
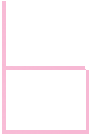
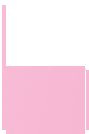
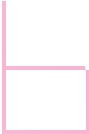
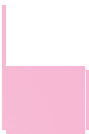
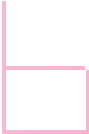
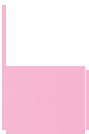
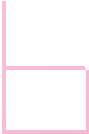
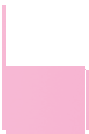
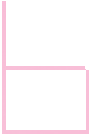
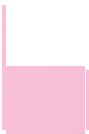
What are the four instructions that were executed?

What are the contents of R0 and R1 after the four instructions execute?

CHAPTER

8

Data Structures



p to now, each item of information we have processed with the computer has been a single value—either an integer, a floating point number, or an ASCII character. The real world is filled with items of information far more com- plex than simple, single numbers. A company’s organization chart and a list of items arranged in alphabetical order are two examples. We call these complex items of information *abstract data types*, or more colloquially *data structures*. In this chapter, we will study three abstract data types: stacks, queues, and char- acter strings. We will write programs to solve problems that require expressing information according to its structure. There are other abstract data types to be sure, but we will leave those for Chapter 15, after we have introduced you to the

**U**

C programming language.

Before we get to stacks, queues, and character strings, however, we intro- duce a new concept that will prove very useful in manipulating data structures: *subroutines*, or what is also called *functions*.

## !.1 Subroutines

It is often useful to be able to invoke a program fragment multiple times within the same program without having to specify its details in the source program each time it is needed. Also, in the real world of computer software development, it is often (usually?) the case that one software engineer writes a program that requires such fragments and another software engineer writes the fragments.

Or, one might require a fragment that has been supplied by the manufacturer y some independent software supplier. It is almost always the case that col- ions of such fragments are available to user programmers to free them from ving to write their own. These collections are referred to as *libraries*. An exam- e is the Math Library, which consists of fragments that compute such functions

or b

lect

ha

pl

as **square root, sine,** and **arctangent**. In fact, because the Math Library exists, user programmers can get the computer to compute those functions without even having to know how to write a program fragment to do it!

For all of these reasons, it is good to have a way to use program frag- ments efficiently. Such program fragments are called *subroutines,* or alternatively, *procedures*, or in C terminology, *functions*.

Figure 8.1 provides a simple illustration of a part of a program—call it “piece- of-code-A”—containing fragments that must be executed multiple times within piece-of-code-A. Figure 8.1 will be studied in detail in Chapter 9, but for now,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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>’’

**Figure !.1 Instruction sequence (Piece-of-code-A) we will study in detail in Chapter #.**

let’s ignore everything about it except the three-instruction sequences starting at symbolic addresses L1, L2, L3, and L4.

Each of these four 3-instruction sequences does the following:

|  |  |  |
| --- | --- | --- |
| label | LDI  BRzp | R3,DSR  label |
|  | STI | Reg,DDR |

Each of the four instances uses a different label (L1, L2, L3, L4), but that is a problem since in each instance the only purpose of the label is to branch k from the BRzp instruction to the LDI instruction.

not

bac

Two of the four program fragments store the contents of R0 and the other two e the contents of R2, but that is easy to take care of, as we will see. The main point is that, aside from the small nuisance of which register is being used for the source of the STI instruction, the four program fragments do exactly the same thing, and it is wasteful to require the programmer to write the code four times.

stor

The subroutine call/return mechanism enables the programmer to write the code only once.

##### !.1.1 The Call/Return Mechanism

The call/return mechanism allows us to execute this one three-instruction sequence multiple times by requiring us to include it as a subroutine in our program only once.

Figure 8.2 shows the instruction execution flow for a program with and hout subroutines.

wit

Note in Figure 8.2 that without subroutines, the programmer has to provide same code A after X, after Y, and after Z. With subroutines, the programmer

the

|  |
| --- |
| X |
| A |
| Y |
| A |
| Z |
| A |
| W |



1

3

5

2

4

6

Return

A

W

Call

Z

Call

Y

Call

X

(a) Without subroutines (b) With subroutines

**Figure !.$ Instruction execution flow with/without subroutines.**

has to provide the code A only once. The programmer uses the call/return mech- anism to direct the computer each time via the **call instruction** to the code A, and after the computer has executed the code A, to the **return instruction** to the proper next instruction to be executed in the program.

We refer to the program that contains the call as the *caller*, and the subroutine that contains the return as the *callee*.

The call/return mechanism consists of two instructions. The first instruction **JSR(R)** is in the caller program and does two things: It loads the PC with the starting address of the subroutine and it loads R7 with the address immediately after the address of the JSR(R) instruction. The address immediately after the address of the JSR(R) instruction is the address to come back to after execut- ing the subroutine. We call the address we come back to the *return linkage*. The second instruction **JMP R7** is the last instruction in the subroutine (i.e., in the callee program). It loads the PC with the contents of R7, the address just after the address of the JSR instruction, completing the round trip flow of control from the caller to the callee and back.

##### !.1.$ JSR(R)—The Instruction That Calls the Subroutine

The LC-3 specifies one control instruction for calling subroutines; its opcode is

**0**in**1**g**00**

. The instruction uses one of two addressing modes for computing the start- address of the subroutine, PC-relative addressing or Base Register addressing.

The LC-3 assembly language provides two different mnemonic names for the opcode, JSR and JSRR, depending on which addressing mode is used.

The JSR(R) instruction does two things. Like all control instructions, it loads PC, overwriting the incremented PC that was loaded during the FETCH phase the JSR(R) instruction. In this case the starting address of the subroutine is computed and loaded into the PC. The second thing the JSR(R) instruction does is save the return address in R7. The return address is the incremented PC, which is the address of the instruction following the JSR(R) instruction in the calling

the

of

program.

The JSR(R) instruction consists of three parts.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |
| --- | --- | --- |
| Opcode | A | Address evaluation bits |

Bits [15:12] contain the opcode, 0100. Bit [11] specifies the addressing mode, value 1 if the addressing mode is PC-relative, and the value 0 if the addressing mode is Base Register addressing. Bits [10:0] contain information that is used to obtain the starting address of the subroutine. The only difference between JSR and JSRR is the addressing mode that is used for evaluating the starting address

the

of the subroutine.

***JSR*** The JSR instruction computes the target address of the subroutine by sign- extending the 11-bit offset (bits [10:0]) of the instruction to 16 bits and adding that to the incremented PC. This addressing mode is almost identical to the addressing

mode of another control instruction, the BR instruction, except eleven bits of PCoffset are used, rather than nine bits as is the case for BR.

If the following JSR instruction is stored in location x4200, its execution will cause the PC to be loaded with x3E05 (i.e., xFC04 + x4201) and R7 to be loaded with x4201.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |
| --- | --- | --- |
| 0 1 0 0  JSR | 1  A | 1 0 0 0 0 0 0 0 1 0 0  PCoffset11 |

***JSRR*** The JSRR instruction is exactly like the JSR instruction except for the addressing mode. **JSRR** obtains the starting address of the subroutine in exactly the same way the JMP instruction does; that is, bits [8:6] identify the Base Register, that contains the address to be loaded into the PC.

If the following JSRR instruction is stored in location x420A, and if R5 con- s x3002, the execution of the JSRR will cause R7 to be loaded with x420B the PC to be loaded with x3002.

tain

and

*Question:* What important feature does the JSRR instruction provide that the instruction does not provide?

JSR

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 1 0 0  JSRR | 0  A | 0 0 | 1 0 1  BaseR | 0 0 0 0 0 0 |

##### !.1.# Saving and Restoring Registers

We have known for a long time that every time an instruction loads a value into a register, the value that was previously in that register is lost. Thus, we need to save the value in a register

* if that value will be destroyed by some subsequent instruction, and
* if we will need it after that subsequent instruction.

This can be a problem when dealing with subroutines. s examine again the piece of code in Figure 8.1. Suppose

Let’ this piece of code is

a su ller program, which

we broutine called by the instruction JSR START in some ca

will call CALLER.

Suppose before CALLER executes JSR START, it computes values that it ds into R1, R2, and R3. In our subroutine starting at START, the instruction line 05 loads a value into R2, the instruction on line 06 loads a value into R3, and the instruction on line 0A loads a value into R1. What would happen if CALLER needed those values after returning from the subroutine that begins at START? Too bad! Since the subroutine destroyed the values in R1, R2, and R3 by executing the instructions in lines 05, 06, and 0A, those values are lost to CALLER when it resumes execution after the JMP R7 instruction on line 20 of

loa

on

the subroutine. Of course, this is unacceptable.

We prevent it from happening during the first part of our subroutine, that during initialization. In lines 01, 02, and 03, the contents of R1, R2, and R3

is,

are stored in memory locations SaveR1, SaveR2, and SaveR3. Three locations in the subroutine (lines 22, 23, and 24) have been set aside for the purpose of saving those register values. And, in lines 1D, 1E, and 1F (just before the JMP R7 instruction), the values stored there are put back into R1, R2, and R3. That is, before the subroutine uses R1, R2, and R3 for its own use, the subroutine **saves** the values put there by the calling program. And, before the subroutine returns to the calling program, those values are put back (i.e., **restored**) where the calling program has a right to expect them.

We call this technique *callee save* because the subroutine (i.e., the callee) saves and restores the registers. It makes sense to have the subroutine save the registers because the subroutine knows which registers it needs to do the work of the subroutine. There really is no reason to burden the person writing the caller program to know which registers the subroutine needs.

We could of course have the caller program save all the registers before JSR START, and then the subroutine would not have to bother saving any of them. Some programs do that, and in fact, some ISAs have JSR instructions that do that as part of the execution of the JSR instruction. But if we wish to eliminate unnecessary saves and restores, we can do so in this case by having the callee save only the registers it needs.

We should also point out that since JMP START loads the return linkage in R7, whatever was in R7 is destroyed by the execution of the JMP START instruction. Therefore, if the calling program had stored a value in R7 before calling the subroutine at START, and it needed that value after returning from the subroutine, the caller program would have to save and restore R7. Why should the caller program save and restore R7? Because the caller program knows that the contents of R7 will be destroyed by execution of JMP START. We call this *caller save* because the calling program saves and restores the register value.

The message is this: If a value in a register will be needed after something else is stored in that register, we must *save* it before something else happens and *restore* it before we can subsequently use it. We save a register value by storing it in memory; we restore it by loading it back into the register.

The save/restore problem can be handled either by the calling program before the JSR occurs or by the subroutine. We will see in Section 9.3 that the same prob- lem exists for another class of calling/called routines, those due to system calls.

In summary, we use the term *caller save* if the calling program handles the problem. We use the term *callee save* if the called program handles the problem. The appropriate one to handle the problem is the one that knows which registers will be destroyed by subsequent actions.

The callee knows which registers it needs to do the job of the called program. Therefore, before it starts, it saves those registers with a sequence of stores. After it finishes, it restores those registers with a sequence of loads. And it sets aside memory locations to save those register values.

The caller knows what damage will be done by instructions under its con- trol. It knows that each instance of a JSR instruction will destroy what is in R7. So, before the JSR instruction is executed, R7 is saved. After the caller program resumes execution (upon completion of the subroutine), R7 is restored.

##### !.1.4 Library Routines

We noted early in this section that there are many uses for the call/return mech- anism, among them the ability of a user program to call library subroutines that are usually delivered as part of the computer system. Libraries are provided as a convenience to the user programmer. They are legitimately advertised as *pro- ductivity enhancers* since they allow the application programmer to use them without having to know or learn much of their inner details. For example, it is often the case that a programmer knows what a square root is (we abbreviate **SQRT**), may need to use sqrt(x) for some value x, but does not have a clue as to how to write a program to perform sqrt, and probably would rather not have to learn how.

A simple example illustrates the point: We have lost our key and need to get into our apartment. We can lean a ladder up against the wall so that the ladder touches the bottom of our open window, 24 feet above the ground. There is a 10-foot flower bed on the ground along the edge of the wall, so we need to keep the base of the ladder outside the flower bed. How big a ladder do we need so that we can lean it against the wall and climb through the window? Or, stated less colorfully: If the sides of a right triangle are 24 feet and 10 feet, how big is the hypotenuse (see Figure 8.3)?

24 feet

Ladder

10 feet

**Figure !.& Solving for the length of the hypotenuse.**

We remember from high school that Pythagoras answered that one for us:

*c*2 = *a*2 + *b*2

Kno2wing *a* and *b*, we can easily solve for *c* by taking the square root of the sum

of *a* and *b*2. Taking the sum is not hard—the LC-3 ADD instruction can do that

job. The square is also not hard; we can multiply two numbers by a sequence of additions. But how does one get the square root? The structure of our solution is shown in Figure 8.4.

The subroutine SQRT has yet to be written. If it were not for the Math Library, the programmer would have to pick up a math book (or get someone to do it for him/her), check out the Newton-Raphson method, and produce the missing subroutine.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 |  | ... |  |  |
| 02 |  | ... |  |
| 03 |  | LD | R0,SIDE1 |
| 04 |  | BRz | S1 |
| 05 |  | JSR | SQUARE |
| 06 | S1 | ADD | R1,R0,#0 |
| 07 |  | LD | R0,SIDE2 |
| 08 |  | BRz | S2 |
| 09 |  | JSR | SQUARE |
| 0A | S2 | ADD | R0,R0,R1 |
| 0B |  | JSR | SQRT |
| 0C |  | ST | R0,HYPOT |
| 0D |  | BRnzp | NEXT\_TASK |
| 0E | SQUARE | ADD | R2,R0,#0 |
| 0F |  | ADD | R3,R0,#0 |
| 10 | AGAIN | ADD | R2,R2,#-1 |
| 11 |  | BRz | DONE |
| 12 |  | ADD | R0,R0,R3 |
| 13 |  | BRnzp | AGAIN |
| 14 | DONE | RET |  |
| 15 | SQRT | ... | ; | R0 <-- SQRT(R0) |
| 16 |  | ... | ; |  |
| 17 |  | ... | ; | How do we write this subroutine? |
| 18 |  | ... | ; |  |
| 19 |  | ... | ; |  |
| 1A |  | RET |  |  |
| 1B | SIDE1 | .BLKW | 1 |  |
| 1C | SIDE2 | .BLKW | 1 |  |
| 1D | HYPOT | .BLKW | 1 |  |
| 1E |  | ... |  |  |
| 1F |  | ... |  |  |

**Figure !.4 A program fragment to compute the hypotenuse of a right triangle.**

However, with the Math Library, the problem pretty much goes away. Since the Math Library supplies a number of subroutines (including SQRT), the user programmer can continue to be ignorant of the likes of Newton-Raphson. The user still needs to know the label of the target address of the library routine that performs the square root function, where to put the argument x, and where to expect the result SQRT(x). But these are easy conventions that can be obtained from the documentation associated with the Math Library.

If the library routine starts at address SQRT, and the argument is provided in R0 to the library routine, and the result is obtained in R0 from the library routine, Figure 8.4 reduces to Figure 8.5.

Two things are worth noting:

* + *Thing 1*—The programmer no longer has to worry about how to compute the square root function. The library routine does that for us.
  + *Thing 2*—The pseudo-op .EXTERNAL. We already saw in Section 7.4.2 that this pseudo-op tells the assembler that the label (SQRT), which is needed to assemble the .FILL pseudo-op in line 19, will be supplied by some other program fragment (i.e., module) and will be combined with this program

01 ...

02 ...

03 .EXTERNAL SQRT

04 ...

05 ...

1. LD R0,SIDE1
2. BRz S1
3. 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

1. ST R0,HYPOT
2. BRnzp NEXT\_TASK
3. SQUARE ADD R2,R0,#0 13 ADD R3,R0,#0
4. AGAIN ADD R2,R2,#-1
5. BRz DONE
6. ADD R0,R0,R3
7. BRnzp AGAIN
8. DONE RET
9. BASE .FILL SQRT 1A SIDE1 .BLKW 1

1B SIDE2 .BLKW 1

1C HYPOT .BLKW 1

1D ...

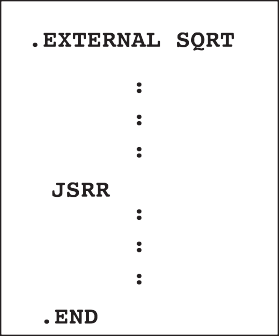
1E ...

**Figure !.’ The program fragment of Figure !.4, using a library routine.**

fragment (i.e., module) when the *executable image* is produced. The exe- cutable image is the binary module that actually executes. The executable image is produced at *link* time.

This notion of combining multiple modules at link time to produce an exe- cutable image is the normal case. Figure 8.6 illustrates the process. You will see concrete examples of this when we work with the programming language C in the second half of this course.

Most application software requires library routines from various libraries. It would be very inefficient for the typical programmer to produce all of them— assuming the typical programmer were able to produce such routines in the first place. We have mentioned routines from the Math Library. There are also a num- ber of preprocessing routines for producing *pretty* graphic images. There are other routines for a number of other tasks where it would make no sense at all to have the programmer write them from scratch. It is much easier to require only (1) appro- priate documentation so that the interface between the library routine and the program that calls that routine is clear, and (2) the use of the proper pseudo-ops such as .EXTERNAL in the source program. The linker can then produce an executable image at link time from the separately assembled modules.

Source module A

Link

Object module for math library

Some other separately assembled module

Symbol table for math library

Symbol table for A

Assemble

Object module for A

Executable image

|  |
| --- |
|  |
|  |
|  |
|  |

Symbol table

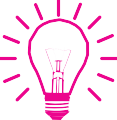
**Figure !.( An executable image constructed from multiple files.**

## !.2 The Stack

Now we are ready to study some data structures. The first and most important data structure is the stack.

##### !.2.# The Stack—An Abstract Data Type

Throughout your future interaction with computers (whether writing software or designing hardware), you will encounter again and again the storage mechanism known as a *stack*. Stacks can be implemented in many different ways, and we will get to that momentarily. But first, it is important to know that the concept of a stack has nothing to do with how it is implemented. The concept of a stack is the specification of how it is to be *accessed*. That is, the defining notion ofa stack is that the **last** thing you stored in the stack is the **first** thing you remove from it. That is what makes a stack different from everything else in the world. Simply put: Last In, First Out, or LIFO.



In the terminology of computer programming languages, we say the stack is xample of an *abstract data type*. That is, an abstract data type is a storage hanism that is defined by the operations performed on it and not at all by the specific manner in which it is implemented. In this section, you will see stacks

an e

mec

implemented as sequential locations in memory.

##### !.2.2 Two Example Implementations

A coin holder in the armrest next to the driver of an automobile is an example of a stack. The first quarter you take to pay the highway toll is the last quarter you added to the stack of quarters. As you add quarters, you push the earlier quarters down into the coin holder.

Figure 8.7 shows the behavior of a coin holder. Initially, as shown in ure 8.7a, th coin holder is emp y. The firs highway toll is 75 cents, and u give the tol collector a dol ar. He gives you 25 cents change, a 1995 quar-

Fig e t t

yo l l

ter, which you insert into the coin holder. The coin holder appears as shown in Figure 8.7b.



1995 Quarter

|  |
| --- |
| 1996 Quarter |
| 1998 Quarter |
| 1982 Quarter |
| 1995 Quarter |
|  |

|  |
| --- |
| 1982 Quarter |
| 1995 Quarter |
|  |

* 1. Initial state (Empty)
  2. After one push
  3. After three pushes
  4. After two pops

**Figure !.$ A coin holder in an automobile armrest—example of a stack.**

There are special terms for the insertion and removal of elements from a stack. We say we *push* an element onto the stack when we insert it. We say we *pop* an element from the stack when we remove it.

The second highway toll is $4.25, and you give the toll collector $5.00. She gives you 75 cents change, which you insert into the coin holder: first a 1982 quarter, then a 1998 quarter, and finally, a 1996 quarter. Now the coin holder is as shown in Figure 8.7c. The third toll is 50 cents, and you remove (pop) the top two quarters from the coin holder: the 1996 quarter first and then the 1998 quarter. The coin holder is then as shown in Figure 8.7d.

The coin holder is an example of a stack, **precisely** because it obeys the LIFO requirement. Each time you insert a quarter, you do so at the top. Each time you remove a quarter, you do so from the top. The last coin you inserted is the first coin you remove. Therefore, it is a stack.

Another implementation of a stack, sometimes referred to as a computer hardware stack, is shown in Figure 8.8. Its behavior resembles that of the coin holder we just described. It consists of some number of hardware registers, each of which can store a value. The example of Figure 8.8 contains five registers. As each value is added to the stack or removed from the stack, the values **already** on the stack **move.**

Empty:

Empty:

TOP

/ / / / / /

/ / / / / /

/ / / / / /

/ / / / / /

/ / / / / /

Yes

Empty:

TOP

/ / / / / /

18

31

5

12

No

Empty:

TOP

/ / / / / /

/ / / / / /

/ / / / / /

/ / / / / /

18

No

No

TOP

/ / / / / /

/ / / / / /

/ / / / / /

18

31

* + 1. Initial state (b) After one push (c) After three pushes (d) After two pops

**Figure !.! A stack, implemented in hardware—data entries move.**

In Figure 8.8a, the stack is initially shown as empty. Access is always via the first element, which is labeled TOP. If the value 18 is pushed onto the stack, we have Figure 8.8b. If the three values 31, 5, and 12 are pushed (in that order), the result is as shown in Figure 8.8c. Finally, if two values are popped from the stack, we have Figure 8.8d. A distinguishing feature of the stack of Figure 8.8 is that, like the quarters in the coin holder, as each value is added or removed, **all the other values already on the stack move**.

##### !.$.& Implementation in Memory

By far the most common implementation of a stack in a computer is as shown in Figure 8.9. This stack consists of a sequence of memory locations along with a mechanism, called the *stack pointer*, which keeps track of the **top** of the stack. We use R6 to contain the address of the top of the stack. That is, in the LC-3, R6 is the stack pointer.

x3FFB x3FFC x3FFD x3FFE x3FFF

TOP

|  |
| --- |
| / / / / / / |
| / / / / / / |
| / / / / / / |
| / / / / / / |
| / / / / / / |

R6

x4000

x3FFB x3FFC x3FFD x3FFE x3FFF

x3FFB x3FFC x3FFD x3FFE x3FFF

R6

/ / / / / /

/ / / / / /

/ / / / / /

/ / / / / /

18

TOP

x3FFF

x3FFB x3FFC x3FFD x3FFE x3FFF

R6 R6

/ / / / / /

12

5

31

18

TOP

/ / / / / /

12

5

31

18

TOP

x3FFE

x3FFC

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

**Figure !.# A stack, implemented in memory—data entries do not move.**

In Figure 8.9, five memory locations (x3FFF to x3FFB) are provided for the stack. The actual locations comprising the stack at any single instant of time are the consecutive locations from x3FFF to the location specified in R6, that is, the top of the stack. For example, in Figure 8.9c, the stack consists of the contents of locations x3FFF, x3FFE, x3FFD, and x3FFC.

Figure 8.9a shows an initially empty stack. Since there are no values on the stack, the stack pointer contains the address x4000, the address of the memory location just after the memory locations reserved for the stack. Why this makes sense will be clear after we show the actual code for pushing values onto and popping values off of the stack. Figure 8.9b shows the stack after pushing the value 18. Note that the stack pointer contains the address x3FFF, which is the new top of the stack.

Figure 8.9c shows the stack after pushing the values 31, 5, and 12, in that order. Note that the values inserted into the stack are stored in memory loca- tions having decreasing addresses. We say the stack *grows toward zero*. Finally, Figure 8.9d shows the stack after popping the top two elements off the stack.

Note that those two elements (the values 5 and 12) that were popped are still present in memory locations x3FFD and x3FFC. However, as we will see momentarily, those values 5 and 12 cannot be accessed from memory, as long as **every** access to memory is controlled by the stack mechanism.

Note also that, unlike the coin holder and computer hardware stack imple- mentations discussed in the previous section, 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**.

***Push*** We push a value onto the stack by executing the two-instruction sequence

PUSH ADD R6,R6,#-1 STR R0,R6,#0

In Figure 8.9a, R6 contains x4000, indicating that the stack is empty. To push the value 18 onto the stack, we decrement R6, the stack pointer, so the address in R6 (i.e., address x3FFF) corresponds to the location where we want to store the value we are pushing onto the stack. The actual push is done by first loading 18 into R0, and then executing STR R0,R6,#0. This stores the contents of R0 into memory location x3FFF.

That is, to push a value onto the stack, we first load that value into R0. Then we decrement R6, which contained the previous top of the stack. Then we execute STR R0,R6,#0, which stores the contents of R0 into the memory location whose address is in R6.

The three values 31, 5, and 12 are pushed onto the stack by loading each in turn into R0 and then executing the two-instruction sequence. In Figure 8.9c, R6 (the stack pointer) contains x3FFC, indicating that the top of the stack is location x3FFC and that 12 was the last value pushed.

***Pop*** To pop a value from the stack, the value is read and the stack pointer is incremented. The following two-instruction sequence

POP LDR R0,R6,#0 ADD R6,R6,#1

pops the value contained in the top of the stack and loads it into R0. The stack pointer (R6) is incremented to indicate that the old value at the top of the stack has been popped and is no longer on the stack, and we have a new value at the top of the stack.

If the stack were as shown in Figure 8.9c and we executed the sequence twice, we would pop two values from the stack. In this case, we would first remove the 12, and then the 5. Assuming the purpose of popping two values is to use those two values, we would, of course, have to move the 12 from R0 to some other location before calling POP a second time.

Note that after 12 and 5 are popped, R6 contains x3FFE, indicating that 12 and 5 are no longer on the stack and that the top of the stack is 31. Figure 8.9d shows the stack after that sequence of operations.

Note that the values 12 and 5 are still stored in memory locations x3FFD and x3FFC, respectively. However, since the stack requires that we push by executing the PUSH sequence and pop by executing the POP sequence, we cannot read the values 12 and 5 if we obey the rules. The fancy name for “the rules” is the *stack protocol*.

***Underflow*** What happens if we now attempt to pop three values from the stack? Since only two values remain on the stack, we would have a problem. Attempting to pop items that have not been previously pushed results in an *underflow* situ- ation. In our example, we can test for underflow by comparing the stack pointer with x4000, which would be the contents of R6 if there were nothing left on the stack to pop. If UNDERFLOW is the label of a routine that handles the underflow condition, our resulting POP sequence would be

POP LD R1,EMPTY

ADD R2,R6,R1 ; Compare stack

BRz UNDERFLOW ; pointer with x4000.

;

LDR R0,R6,#0

ADD R6,R6,#1

;

RET

EMPTY .FILL xC000 ; EMPTY <-- negative of x4000



Yes

Underflow

?

No

R5 <-- 0

R5 <-- 1

R0 <-- Value popped

**Figure !.1) POP routine, including test for underflow.**

Rather than have the POP routine immediately jump to the UNDERFLOW routine if the POP is unsuccessful, it is often useful to have the POP routine return to the calling program with the underflow information contained in a reg- ister. We will use R5 to provide success/failure information. Figure 8.10 is a flowchart showing how the POP routine could be augmented, using R5 to report this success/failure information.

Upon return from the POP routine, the calling program would examine R5 to determine whether the POP completed successfully (R5 = 0), or not (R5 = 1). Note that since the POP routine reports success or failure in R5, whatever was stored in R5 **before** the POP routine was called is lost. Thus, it is the job of the calling program to save the contents of R5 before the JSR instruction is executed if the value stored there will be needed later. Recall from Section 8.1.3

that this is an example of a caller-save situation.

The resulting POP routine is shown in the following instruction sequence.

|  |  |  |
| --- | --- | --- |
| POP | AND LD ADD | R5,R5,#0 R1,EMPTY R2,R6,R1 |
|  | BRz | Failure |
|  | LDR | R0,R6,#0 |
|  | ADD | R6,R6,#1 |
|  | RET |  |
| Failure | ADD | R5,R5,#1 |
|  | RET |  |
| EMPTY | .FILL | xC000 ; EMPTY <-- -x4000 |

***Overflow*** What happens when we run out of available space and we try to push a value onto the stack? Since we cannot store values where there is no space, we have an *overflow* situation. We can test for overflow by comparing the stack pointer with (in the example of Figure 8.9) x3FFB. If they are equal, we have no place to push another value onto the stack. If OVERFLOW is the label

of a routine that handles the overflow condition, our resulting PUSH sequence would be

|  |  |  |
| --- | --- | --- |
| PUSH | LD ADD  BRz | R1,MAX R2,R6,R1 OVERFLOW |
| ; |  |  |
|  | ADD | R6,R6,#-1 |
|  | STR | R0,R6,#0 |
| ; |  |  |
|  | RET |  |
| MAX | .FILL | xC005 ; MAX <-- negative of x3FFB |

In the same way that it is useful to have the POP routine return to the calling ogram with success/failure information, rather than immediately jumping to the

pr

UNDERFLOW routine, it is useful to have the PUSH routine act similarly.

We augment the PUSH routine with instructions to store 0 (success) or ailu e) in R5 depending on whether or not the push completed success-

1 (f r ,

full

y. Upon return from the PUSH routine, the calling program would examine (R5to determine whether the PUSH completed successfully (R5 = 0) or not Note again that since the PUSH routine reports success or failure in R5, we

hav

= 1).

R5

e another example of a caller-save situation. That is, since whatever was stored R5 before the PUSH routine was called is lost, it is the job of the calling pro- am to save the contents of R5 before the JSR instruction is executed if the value ored in R5 will be needed later.

in

gr

st

The resulting PUSH routine is shown in the following instruction sequence.

|  |  |  |
| --- | --- | --- |
| PUSH | AND LD ADD | R5,R5,#0 R1,MAX R2,R6,R1 |
|  | BRz | Failure |
|  | ADD | R6,R6,#-1 |
|  | STR | R0,R6,#0 |
|  | RET |  |
| Failure | ADD | R5,R5,#1 |
|  | RET |  |
| MAX | .FILL | xC005 ; MAX <-- -x3FFB |

##### !.$.4 The Complete Picture

The POP and PUSH routines allow us to use memory locations x3FFF through x3FFB as a five-entry stack. If we wish to push a value onto the stack, we simply load that value into R0 and execute JSR PUSH. To pop a value from the stack into R0, we simply execute JSR POP. If we wish to change the location or the size of the stack, we adjust BASE and MAX accordingly.

Before leaving this topic, we should be careful to clean up an important detail we discussed in Section 8.1.3. The subroutines PUSH and POP make use of and R2, and there is no reason why the calling program would know that.

that

R1

Therefore, it is the job of the subroutine (callee save) to save R1 and R2 before using them, and to restore them before returning to the calling program.

The PUSH and POP routines also write to R5. But, as we have already pointed out, the calling program knows that the subroutine will report success or failure in R5, so it is the job of the calling program to save R5 before executing the JSR instruction if the value stored in R5 will be needed later. As discussed in Section 8.1.3, this is an example of caller save.

The final code for our PUSH and POP operations is shown in Figure 8.11.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 01 | ; | | | | | |  |
| 02 | ; Subroutines for carrying out the PUSH and POP functions. This | | | | | |
| 03 | ; program works with a stack consisting of memory locations x3FFF | | | | | |
| 04 | ; through x3FFB. R6 is the stack pointer. | | | | | |
| 05 | ; | | | | | |
| 06 | POP AND R5,R5,#0 ; R5 <-- success | | | | | |
| 07 | ST R1,Save1 ; Save registers that | | | | | |
| 08 | ST R2,Save2 ; are needed by POP | | | | | |
| 09  0B  0C  0D  0E  0F  10 | LD R1,EMPTY ; EMPTY contains -x4000 ADD R2,R6,R1 ; Compare stack pointer to 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 | | | | | | x4000 |
| 11 | ; | | | | | |  |
| 12 | PUSH AND R5,R5,#0 | | | | | |  |
| 13 | ST R1,Save1 ; Save registers that | | | | | |  |
| 14 | ST R2,Save2 ; are needed by PUSH | | | | | |  |
| 15 | LD R1,FULL ; FULL contains -x3FFB | | | | | |  |
| 16 | ADD R2,R6,R1 ; Compare stack pointer to | | | | | | x3FFB |
| 17 | BRz fail\_exit ; Branch if stack is full | | | | | |  |
| 18 | ; | | | | | |  |
| 19  1A  1B  1C  1D  1E  1F  20 | 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 | | | | | |  |
| 21 | ADD R5,R5,#1 ; R5 <-- failure | | | | | |  |
| 22 | RET | | | | | |  |
| 23 | ; | | | | | |  |
| 24 | EMPTY .FILL xC000 ; EMPTY contains -x4000 | | | | | |  |
| 25 | FULL | .FILL | xC005 | ; | FULL contains | -x3FFB | |
| 26 | Save1 | .FILL | x0000 |  |  |  | |
| 27 | Save2 | .FILL | x0000 |  |  |  | |

**Figure !.11 The stack protocol.**

## !.& Recursion, a Powerful Technique When Used Appropriately

Recursion is a mechanism for expressing a function *in terms of itself*. Some have referred to it as picking oneself up by one’s bootstraps, since at first blush, it looks like magic—which, of course, it isn’t.

When used appropriately, the expressive power of recursion is going to save us a lot of headaches. When used whimsically, recursion is going to require unnecessary activity, resulting in longer execution time and wasted energy.

The mechanism is so important that we will study it in greater detail later in the book after we have raised the level of abstraction to programming in a high-level language. However, since a critical concept needed to understand the implementation of recursion is the stack, which we have just studied, it is useful to show by means of examples just when using recursion is warranted and when using it is not a good idea.

We will examine two ill-advised uses of recursion. We will also examine a problem where using the expressive power of recursion is very helpful.

##### !.&.1 Bad Example Number 1: Factorial

The simplest example to illustrate recursion is the function **factorial**. The equation

n! = n \* (n-1)!

says it all. We are expressing factorial in terms of factorial! How we can write a program to do this we will see momentarily.

Assume the subroutine FACT (Factorial) is supplied with a positive integer n in R0 and returns with the value n! in R0. (We will save 0! for an exercise at the end of the chapter.)

Figure 8.12 shows a pictorial view of the recursive subroutine. We rep- resent the subroutine FACT as a hexagon, and inside the hexagon is another instance of the hexagon! We call the subroutine recursive because inside the FACT subroutine is an instruction JSR FACT.

The subroutine first tests to see if n = 1. If so, we are done, since (1)!

= 1. It is important to emphasize that every recursive subroutine must have such an initial test to see if we should execute the recursive call. Without this test, the subroutine would call itself (JSR FACT) an infinite number of times! Clearly, that cannot be correct. The answer is to provide a test before the recur- sive JSR instruction. In the case of the subroutine FACT, if R0 is 1, we are done, since 1! = 1.

If n does not equal 1, we save the value in R1, so we can store n in R1, load R0 with n-1 and JSR FACT. When FACT returns with (n-1)! in R0, we multiply it by n (which was stored in R1), producing n!, which we load into R0, restore R1 to the value expected by the calling program, and RET.

If we assume the LC-3 has a MUL instruction, the basic structure of the FACT subroutine takes the following form:



R0 = n

Yes

n = 1

?

No

Save n

R0 n − 1

R0 = (n−1)!

R0 n \* (n − 1)!

R0 = n!

**Figure !.1$ Flowchart for a recursive FACTORIAL subroutine.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| FACT | ST | R1, Save1 ; Callee save R1 | | | | |
|  | ADD  BRz ADD | R1,R0,#-1 ; Test if R0=1  DONE ; If R0=1, R0 also contains (1)!, so we are done R1,R0,#0 ; Save n in R1, to be used after we compute (n-1)! | | | | |
|  | ADD | R0,R1, #-1 | ; | Set R0 to n-1, and then call FACT |  |  |
| B | JSR | FACT | ; | On RET, R0 will contain (n-1)! |  |  |
|  | MUL | R0,R0,R1 | ; | Multiply n times (n-1)!, yielding n! | in | R0 |
| DONE  Save1 | LD RET  .BLKW | R1, Save1  1 | ; | Callee restore R1 |  |  |

Since the LC-3 does not have a MUL instruction, this will require another subroutine call, but we are ignoring that here in order to focus on the essence of recursion.

Unfortunately, the code we have written will not work. To see why it will not work, Figure 8.13 shows the flow of instruction execution as we would like it to be. The main program calls the subroutine with a JSR instruction at address A. This causes the code labeled #1 to execute. At address B, the subroutine FACT calls itself with the instruction JSR FACT. This causes the code labeled #2 to execute, and so forth.

Note that when the main program executes the instruction JSR FACT, the return linkage A+1 is saved in R7. In the block of code labeled #1, the instruction at address B (JSR FACT) stores its return linkage B+1 in R7, destroying A+1, so there is no way to get back to the main program. Bad! In fact, very, very bad! We can solve this problem by pushing the address A+1 onto a stack before executing JSR FACT at address B. After we subsequently return to address B+1, we can then pop the stack and load the address A+1 into R7 before we execute

the instruction RET back to the main program.

Calling Program

R0 = n

A JSR FACT

#1

B JSR FACT

R0 = n!

R0 = n−1

#2

B JSR FACT

R0 = (n−1)!

R0 = n−2

#3

B JSR FACT

R0 = (n−2)!

R0 = (n−3).! .. R0 = n−3

R0 = 1

R0 = 1!

**Figure !.1& Execution flow for recursive FACTORIAL subroutines.**

Also, note that the instruction ADD R1,R0,#0 in #1 loads the value n into R1, and in #2, the instruction ADD R1,R0,#0 loads the value n-1 into R1, thereby wiping out the value n that had been put there by the code in #1. Thus, when the instruction flow gets back to #1), where the value n is needed by the instruction MUL R0,R0,R1, it is no longer there. It was previously wiped out. Again, very, very bad!

We can solve this problem with a stack also. That is, instead of moving the value n to R1 before loading n-1 into R0, we push n onto the stack and then pop it when we need it after returning from the subroutine with (n-1)! in R0.

Finally, we note that the first instruction in our subroutine saves R1 in Save1 and the last instruction before the RET restores it to R1. We do this so that from the standpoint of the calling program, the value in R1 before the subroutine is the same as the value in R1 after the subroutine, even though the subroutine used R1 in performing its job. However, since our subroutine is recursive, when FACT is called by the JSR instruction at address B, R1 does not contain the value it had in the main program, but instead it has the value last stored in R1 by the ADD R1,R0,#0 instruction. Thus after the JSR FACT instruction is executed, the first instruction of the recursively called subroutine FACT will save that value, wiping out the value that the main program had stored in R1 when it called FACT.

We can solve this problem with a stack also. We simply replace the ST R1,Save1 with a push and LD R1,Save1 with a pop.

If we make these changes (and if the LC-3 had a MUL opcode), the recursive subroutine works as we would like it to. The resulting subroutine is shown in Figure 8.14 (with almost all instructions explained via comments):

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

**Figure !.14 The recursive subroutine FACT.**

a. Contents of stack when JSR FACT executes in #1

|  |
| --- |
|  |
| n |
| A+1 |
| Caller's R1 |

SP

a. Contents of stack when JSR FACT executes in #3

#### SP

|  |
| --- |
| n−2 |
| B+1 |
| n−2 |
| n−1 |
| B+1 |
| n−1 |
| n |
| A+1 |
| Caller's R1 |

**Figure !.1’ The stack during two instances of executing the FACTORIAL subroutine.**

The main program calls FACT with R0 = n. The code in #1 executes, with FACT being called with R0 = n-1. At this point, the stack contains the e entries pushed, as shown in Figure 8.15a. When the JSR FACT instruc- in #3 executes, with R0 = n-3, the stack contains the nine entries as shown igure 8.15b.

JSR

thre

tion

in F

The obvious question you should ask at this point is, “Why is this such ad use of recursion, particularly when ts representa ion n! = n \* (n-1)! is elegant?” To answer this question, we first note how many instructions are ecuted and how much time is wasted pushing and popping elements off the ack. AND, the second question you should ask is, “Is there a better way to

a b i t

so

ex

st

compute n!?”

Consider the alternative shown in Figure 8.16:

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

**Figure !.1( Implementing FACT iteratively (i.e., without recursion).**

##### !.&.$ Fibonacci, an Even Worse Example

Another bad use of recursion is to evaluate the Fibonacci number FIB(n). The

Fibonacci numbers are defined for all non-negative integers as follows: FIB(0)=0, FIB(1)=1, and if n > 1, FIB(n) = FIB(n-1) + FIB(n-2). The expression is beautifully elegant, but the execution time is horrendous.

Figure 8.17 shows a pictorial view of the recursive subroutine FIB. Note that the subroutine FIB is represented as a “capital F,” and inside the capital F there are two more instances of the capital F.

The recursive subroutine in Figure 8.18 computes FIB(n).

R0 = n

Test

R0

n−1

R1 = FIB(n−1)

SAVE FIB(n−1) R0 n−2

R1

R1 = FIB(n−2)

FIB(n−1) + FIB(n−2)

R1 = FIB(n)

Restore R2 Restore n Restore Linkage

Save Linkage Save n Save R2

**Figure !.17 Pictorial representation of the recursive FIB subroutine.**

;FIB subroutine

; + FIB(0) = 0

; + FIB(1) = 1

; + FIB(n) = FIB(n-1) + FIB(n-1)

;

; Input is in R0

; Return answer in R1

;

FIB ADD R6, R6, #-1

STR R7, R6, #0 ; Push R7, the return linkage ADD R6, R6, #-1

STR R0, R6, #0 ; Push R0, the value of n ADD R6, R6, #-1

STR R2, R6, #0 ; Push R2, which is needed in the subroutine

; Check for base case

AND R2, R0, #-2

BRnp SKIP ; Z=0 if R0=0,1

ADD R1, R0, #0 ; R0 is the answer BRnzp DONE

; Not a base case, do the recursion

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SKIP | ADD  JSR | R0,  FIB | R0, | #-1 | ; | R1 = | FIB(n-1) |
|  | ADD | R2, | R1, | #0 | ; | Move | result before calling FIB again |
|  | ADD | R0, | R0, | #-1 |  |  |  |
|  | JSR | FIB |  |  | ; | R1 = | FIB(n-2) |
|  | ADD | R1, | R2, | R1 | ; | R1 = | FIB(n-1) + FIB(n-2) |

; Restore registers and return DONE LDR R2, R6, #0

ADD R6, R6, #1 LDR R0, R6, #0 ADD R6, R6, #1 LDR R7, R6, #0 ADD R6, R6, #1 RET

**Figure !.1! A recursive implementation of Fibonacci.**

As with all recursive subroutines, we first need to test for the base cases. his case, we AND n with xFFFE, which produces a non-zero result for all xcept n = 1 and n = 0. If n = 0 or 1, we are effectively done We move n into R1, restore R2, R0, and R7 (actually, only R2 needs to be restored), and

In t

n e .

return.

If n is not 0 or 1, we need to recursively call FIB twice, once with argument and once with argument n-2. Finally we add FIB(n-1) to FIB(n-2), put the

n-1

result in R1, restore R2, R0, and R7, and return.

Note that the recursive subroutine FIB(n) calls FIB twice: once for FIB(n-1) and once for FIB(n-2). FIB(n-1) must call FIB(n-2) and FIB(n-3), and FIB(n-2) must call FIB(n-3) and FIB(n-4). That means FIB(n-2) must be evaluated twice and FIB(n-3) will have to be evaluated three times.

Question: Suppose n = 10. How many times must this recursive algorithm compute the same function FIB(5)?

Compare the recursive algorithm for Fibonacci (Figure 8.18) with a non- recursive algorithm, as shown in Figure 8.19. Much, much faster execution time!

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

**Figure !.1# An iterative solution to Fibonacci.**

##### !.3.3 The Maze, a Good Example

The reason for shying away from using recursion to compute factorial or Fibonacci is simply that the iterative algorithms are simple enough to understand without the horrendous execution time penalty of recursion. However, it is impor- tant to point out that there are times when the expressive beauty of recursion is useful to attack a complicated problem. Such is the case with the following prob- lem, involving a maze: Given a maze and a starting position within the maze, write a program that determines whether or not there is a way out of the maze from your starting position.

***A Maze*** A maze can be any size, n by m. For example, Figure 8.20 illustrates a 6x6 maze.

**Figure !.#$ Example of a maze.**

Each of the 36 cells of the maze can be characterized by whether there is a door to the north, east, south, or west, and whether there is a door from the cell to the outside world. Each cell is represented by one word of memory (Figure 8.21), as follows:

Bit[4]=1 if there is a door to the outside world; Bit[4]=0 if no door. Bit[3]=1 if there is a door to the cell to the north; Bit[3]=0 if no door. Bit[2]=1 if there is a door to the cell to the east; Bit[2]=0 if no door. Bit[1]=1 if there is a door to the cell to the south; Bit[1]=0 if no door. Bit[0]=1 if there is a door to the cell to the west; Bit[0]=0 if no door.

**Figure !.#1 Specification of each cell in the maze.**

The words are stored in what we call *row major* order; that is, row 1 is stored, then row 2, then row 3, etc. The complete specification of the 6 by 6 maze is shown in Figure 8.22.

1. .ORIG x5000
2. MAZE .FILL x0006
3. .FILL x0007
4. .FILL x0005
5. .FILL x0005
6. .FILL x0003
7. .FILL x0000
8. ; second row: indices 6 to 11
9. .FILL x0008
10. .FILL x000A

0A .FILL x0004

0B .FILL x0003

0C .FILL x000C

0D .FILL x0015

0E ; third row: indices 12 to 17 0F .FILL x0000

1. .FILL x000C
2. .FILL x0001
3. .FILL x000A
4. .FILL x0002
5. .FILL x0002
6. ; fourth row: indices 18 to 23
7. .FILL x0006
8. .FILL x0005
9. .FILL x0007
10. .FILL x000D

1A .FILL x000B

1B .FILL x000A

1C ; fifth row: indices 24 to 29 1D .FILL x000A

1E .FILL x0000

1F .FILL x000A

1. .FILL x0002
2. .FILL x0008
3. .FILL x000A
4. ; sixth row: indices 30 to 35
5. .FILL x0008
6. .FILL x0000
7. .FILL x001A
8. .FILL x000C
9. .FILL x0001
10. .FILL x0008

2A .END

**Figure !.$$ Specification of the maze of Figure !.$).**

***A Recursive Subroutine to Exit the Maze*** Our job is to develop an algorithm to determine whether we can exit a maze from a given starting position within the maze. With all the intricate paths that our attempts can take, keeping track of all that bookkeeping looks daunting. Recursion allows us to not have to keep track of the paths at all! Figure 8.23 shows a pictorial view of a recursive sub- routine FIND EXIT, an algorithm for determining whether or not we can exit the maze. Note that the subroutine FIND EXIT is shown as an octagon, and inside the octagon there are four more instances of octagons, indicating recursive calls to FIND EXIT. If we can exit the maze, we will return from the subroutine with R1=1; if not, we will return with R1=0.

|  |  |  |
| --- | --- | --- |
| NO | |  |
|  |
| 0 |  |

|  |  |  |
| --- | --- | --- |
| NO | |  |
|  |
| 0 |  |

|  |  |  |
| --- | --- | --- |
| NO | |  |
|  |
| 0 |  |

**Figure !.2# Pictorial representation of the recursive subroutine to exit the maze.**

ADDRESS

YES EXIT

NO

NORTH

EAST

SOUTH

NO

No B

YES

+

No B

YES

+

No B

YES

+

WEST

No B

YES

+

R1

1

R1

1

R1

1

R1

R1

R1=0

R1=1

ADDRESS

ADDRESS

ADDRESS

ADDRESS

BREADCRUMB

The algorithm works as follows: In each cell, we first ask if there is an exit from this cell to the outside world. If yes, we return the value 1 and return. If not, we ask whether we should try the cell to the north, the east, the south, or the west. In order to try a cell in any direction, clearly there must be a door to the cell in that direction. Furthermore, we want to be sure we do not end up in an infinite loop where for example, there are doors that allow us to go north one cell, and from there east one cell, and from there south one cell, and from there west one cell, putting us right back where we started. To prevent situations like that from happening, we put a “breadcrumb” in each cell we visit, and we only go to a cell and JSR FIND EXIT if we have not visited that cell before.

Thus, our algorithm:

* 1. From our cell, we ask if we can exit. If yes, we are done. We exit with R1=1.
  2. If not, we put a breadcrumb in our cell. Our breadcrumb is bit [15] of the word corresponding to our current cell. We set it to 1.
  3. We ask two questions: Is there a door to the north, and have we never visited the cell to the north before? If the answer to both is yes, we set the address to the cell to the north, and JSR FIND EXIT. We set the address to the cell to the north by simply subtracting 6 from the address of the current cell. Why 6? Because the cells are stored in row major order, and the number of columns in the maze is 6.
  4. If the answer to either question is no, or if going north resulted in failure, we ask: Is there a door to the east, and have we never visited that cell before? If the answer to both is yes, we set the address to the address of the cell to the east (by adding 1 to the address) and JSR FIND EXIT.
  5. If going east does not get us out, we repeat the question for south, and if that does not work, then for west.
  6. If we end up with no door to the west to a cell we have not visited, or if there is a door and we haven’t visited, but it results in failure, we are done. We cannot exit the maze from our starting position. We set R1=0 and return.

Figure 8.24 shows a recursive algorithm that determines if we can exit the maze, given our starting address.

; Recursive subroutine that determines if there is a path from current cell

; to the outside world.

; input: R0, current cell address

; output: R1, YES (1) or NO (0)

.ORIG x4000

01 FIND\_EXIT ; save modified registers into the stack. 02 ADD R6, R6, #-1

03 STR R2, R6, #0 ; R2 holds the cell data of the caller 04 ADD R6, R6, #-1

05 STR R3, R6, #0 ; R3 holds the cell address of the caller 06 ADD R6, R6, #-1

07 STR R7, R6, #0 ; R7 holds the PC of the caller 08

09 ; Move cell address to R3, since we need to use R0

0A ; as the input to recursive subroutine calls.

0B ADD R3, R0, #0

0C

0D ; If the exit is in this cell, return YES

0E LDR R2, R0, #0 ; R2 now holds the current cell data 0F LD R7, EXIT\_MASK

1. AND R7, R2, R7
2. BRnp DONE\_YES 12
3. ; Put breadcrumb in the current cell.
4. LD R7, BREADCRUMB
5. ADD R2, R2, R7

16 STR R2, R0, #0

17

1. ; check the north cell for a path to exit
2. CHECK\_NORTH LD R7, NORTH\_MASK 1A AND R7, R2, R7

1B BRz CHECK\_EAST ; If north is blocked, check east 1C LDR R7, R3, #-6

1D BRn CHECK\_EAST ; If a breadcrumb in the north cell, check east 1E ADD R0, R3, #-6

1F JSR FIND\_EXIT ; Recursively check the north cell 20 ADD R1, R1, #0

21 BRp DONE\_YES ; If a path from north cell found, return YES 22

1. ; check the north cell for a path to exit
2. CHECK\_EAST LD R7, EAST\_MASK
3. AND R7, R2, R7
4. BRz CHECK\_SOUTH ; If the way to east is blocked, check south 27 LDR R7, R3, #1

28 BRn CHECK\_SOUTH ; If a breadcrumb in the east cell, check south 29 ADD R0, R3, #1

2A JSR FIND\_EXIT ; Recursively check the east cell 2B ADD R1, R1, #0

2C BRp DONE\_YES ; If a path from east cell found, return YES 2D

**Figure !.2# A recursive subroutine to determine if there is an exit from the maze (Fig. !.2# continued on next page.)**

2E ; check the south cell for a path to exit 2F CHECK\_SOUTH LD R7, SOUTH\_MASK

1. AND R7, R2, R7
2. BRz CHECK\_WEST ; If the way to south is blocked, check west 32 LDR R7, R3, #6

33 BRn CHECK\_WEST ; If a breadcrumb in the south cell, check west 34 ADD R0, R3, #6

35 JSR FIND\_EXIT ; Recursively check the south cell 36 ADD R1, R1, #0

37 BRp DONE\_YES ; If a path from south cell found, return YES 38

39 ; check the west cell for a path to exit 3A CHECK\_WEST LD R7, WEST\_MASK

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3B  3C  3D  3E  3F | AND  BRz LDR  BRn  ADD | | R7, R2, DONE\_NO R7, R3, DONE\_NO  R0, R3, | R7 #-1  #-1 | ;  ; | If the way to west is blocked, return NO  If a breadcrumb in the west cell, return NO |
| 40 | JSR | | FIND\_EXIT | | ; | Recursively check the west cell |
| 41 | ADD | | R1, R1, #0 | |  |  |
| 42 | BRp | | DONE\_YES | | ; | If a path from west cell found, return YES |
| 43 |  | |  | |  |  |
| 44 | DONE\_NO | AND R1, R1, #0 | | | | |
| 45 |  | BR RESTORE | | | | |
| 46 |  |  | | | | |
| 47 | DONE\_YES | AND R1, R1, #0 | | | | |
| 48 |  | ADD R1, R1, #1 | | | | |
| 49  4A | RESTORE | ADD R0, R3, #0 ; restore R0 from R3 | | | | |
| 4B  4C  4D  4E  4F  50 |  | ; restore the rest of the modified registers from the stack. LDR R7, R6, #0  ADD R6, R6, #1 LDR R3, R6, #0 ADD R6, R6, #1  LDR R2, R6, #0 | | | | |
| 51 |  | ADD R6, R6, #1 | | | | |
| 52 |  | RET | | | | |
| 53 |  |  | | | | |
| 54 | BREADCRUMB | .FILL x8000 | | | | |
| 55 | EXIT\_MASK | .FILL x0010 | | | | |
| 56 | NORTH\_MASK | .FILL x0008 | | | | |
| 57 | EAST\_MASK | .FILL x0004 | | | | |
| 58 | SOUTH\_MASK | .FILL x0002 | | | | |
| 59  5A | WEST\_MASK | .FILL x0001  .END | | | | |

**Figure !.2# A recursive subroutine to determine if there is an exit from the maze (continued Fig. !.2# from previous page.)**

## !.4 The Queue

Our next data structure is the *queue*. Recall that the property that defined the concept of “stack” was LIFO, the last thing we pushed onto the stack is the first thing we pop off the stack. The defining property of the abstract data type *queue* is **FIFO**. FIFO stands for “First in First out.” The data structure “queue” is likea queue in a polite supermarket, or a polite ticket counter. That is, the first person in line is the first person serviced. In the context of the data structure, this means we need to keep track of two ends of the storage structure: a FRONT pointer for servicing (i.e., removing elements from the front of the queue) and a REAR pointer for entering (i.e., inserting into the rear of the queue).

Figure 8.25 shows a block of six sequential memory locations that have been allocated for storing elements in the queue. The queue grows from x8000 to x8005. We arbitrarily assign the FRONT pointer to the location just before the first element of the queue. We assign the REAR pointer to the location containing the most recent element that was added to the queue. Let’s use R3 as our FRONT pointer and R4 as our REAR pointer.

Figure 8.25a shows a queue in which five values were entered into the queue. Since FRONT = x8001, the values 45 in memory location x8000 and 17 in x8001 must have been removed, and the front element of the queue is 23, the value contained in x8002.

FRONT

REAR

x8000 x8001 x8002 x8003 x8004 x8005

FRONT

REAR

x8000 x8001 x8002 x8003 x8004 x8005

FRONT

REAR

x8000 x8001 x8002 x8003 x8004 x8005

R3 R4

x8004

x8001

|  |
| --- |
| 45 |
| 17 |
| 23 |
| 2 |
| 74 |
|  |

|  |
| --- |
| 45 |
| 17 |
| 23 |
| 2 |
| 74 |
|  |

|  |
| --- |
| 45 |
| 17 |
| 23 |
| 2 |
| 74 |
| 10 |

(a)

|  |
| --- |
| 20 |
| 17 |
| 23 |
| 2 |
| 74 |
| 10 |

R3 R4

(b)

x8004

x8002

|  |
| --- |
| 20 |
| 30 |
| 40 |
| 2 |
| 74 |
| 10 |

R3 R4

(c)

x8005

x8002

|  |
| --- |
| 20 |
| 30 |
| 40 |
| 2 |
| 74 |
| 10 |

REAR

FRONT

R3 R4

x8000

x8002

(d)

x8000 x8001 x8002 x8003 x8004 x8005

FRONT

R3 R4

(e)

x8002

x8002

x8000 x8001 x8002 x8003 x8004 x8005

REAR

FRONT

R3 R4

(f)

x8002

x8002

x8000 x8001 x8002 x8003 x8004 x8005

REAR

**Figure !.$’ A queue allocated to memory locations x!))) to x!))’.**

Note that the values 45 and 17 are still contained in memory locations x8000 and x8001, even though they have been removed. Like the stack, studied already, that is the nature of load instructions. When a value is removed by means of a load instruction, what is stored in the memory location is not erased. The contents of the memory location is simply copied into the destination register. However, since FRONT contains the address x8001, there is no way to load from locations x8000 and x8001 as long as locations x8000 to x8005 behave like a queue—i.e., as long as the accesses are FIFO.

##### !.4.1 The Basic Operations: Remove from Front, Insert at Rear

Since FRONT points to the location just in front of the first element in the queue, we remove a value by first incrementing FRONT and then loading the value stored at that incremented address. In our example, the next value to be removed is the value 23, which is at the front of the queue, in memory location x8002. The following code *removes* 23 from the queue:

ADD R3,R3,#1

LDR R0,R3,#0

yielding the structure in Figure 8.25b.

Since REAR = x8004, the last value to enter the queue is 74. The values in the queue in Figure 8.25b are 2 and 74. To *insert* another element (e.g., 10) at the back of the queue, the following code is executed:

ADD R4,R4,#1

STR R0,R4,#0

resulting in Figure 8.25c.

##### !.4.$ Wrap-Around

At first blush, it looks like we cannot insert any more elements into the queue. Not so! When we remove a value from the queue, that location becomes available for storing another element. We do that by allowing the available storage locations to *wrap around*. For example, suppose we want to add 20 to the queue. Since there is nothing stored in x8000 (recall 45 had been previously removed), we can store 20 in x8000. The result is shown in Figure 8.25d.

“Wrap-around” works by having our removal and insertion algorithms test the contents of FRONT and REAR for the value x8005. If we wish to insert, and REAR contains x8005, we know we have reached the end of our available storage and we must see if x8000 is available. If we wish to remove, we must first see if FRONT contains the address x8005. If it does, the front of the queue is in x8000.

Thus, our code for remove and insert has to include a test for wrap-around. The code for remove becomes:

LD R2, LAST ADD R2,R3,R2 BRnp SKIP\_1 LD R3,FIRST

BR SKIP\_2 SKIP\_1 ADD R3,R3,#1

SKIP\_2 LDR R0,R3,#0 ; R0 gets the front of the queue RET

LAST .FILL x7FFB ; LAST contains the negative of 8005 FIRST .FILL x8000

The code for insert is similar. If REAR contains x8005, we need to set R4 to x8000 before we can insert an element at the rear of the queue. The code to insert is as follows:

LD R2, LAST ADD R2,R4,R2 BRnp SKIP\_1 LD R4,FIRST

BR SKIP\_2 SKIP\_1 ADD R4,R4,#1

SKIP\_2 STR R0,R4,#0 ; R0 gets the front of the queue RET

LAST .FILL 7FFB ; LAST contains the negative of 8005 FIRST .FILL x8000

##### !.4.& How Many Elements Can We Store in a Queue?

Let’s look again at Figure 8.25d. There are four values in the queue: 2, 74, 10, and

20. Suppose we insert 30 and 40 at the rear of the queue, producing Figure 8.25e. Both R3 and R4 contain the same address (x8002), and the queue is full. Now suppose we start removing elements from the front of the queue. If we remove 2, which is at the front of the queue, R3 will contain the address x8003. If we remove the remaining five elements in the queue, we will have what is shown in Figure 8.25f. Note that the FRONT and REAR pointers for e and f are identi- cal, yet Figure 8.25e describes a full queue and Figure 8.25f describes an empty queue! Clearly that is not acceptable.

Our answer is to allow a queue to store only n-1 elements if space for n ele- ments has been allocated. That is, if inserting an nth element into the queue would cause FRONT to equal REAR, we do not allow that insertion. We declare the queue full when there are n-1 elements in the queue.

Let’s look again at the queue in Figure 8.25d. There are four elements in queue, from front to rear: 2, 74, 10, and 20, and two empty slots, x8001 and x8002. We can insert 30 in x8001, producing Figure 8.26a. That is, 30 is the fifth element inserted in the queue. Since six words have been allocated for the queue, and we now have five elements in the queue, we declare the queue full and do not allow a sixth element to be inserted. Suppose we now start removing elements

the

R



|  |
| --- |
| 20 |
| 30 |
| 23 |
| 2 |
| 74 |
| 10 |

|  |
| --- |
| 20 |
| 30 |
| 23 |
| 2 |
| 74 |
| 10 |

F F R



R3 R3

x8001

x8002

x8001

x8001

R4 R4

1. A full queue (b) An empty queue

**Figure !.$( A full queue and an empty queue.**

from the queue until the queue is empty, as shown in Figure 8.26b. Now there is no ambiguity between a full and an empty queue since if the queue is empty, FRONT = REAR.

##### !.4.4 Tests for Underflow, Overflow

As was the case with the stack, we can only remove an element from a queue if there are elements in the queue. Likewise, we can only insert elements in the queue if it is not full. If the queue is empty and we try to remove an element, we have an *underflow* condition. If the queue is full and we try to insert an element, we have an overflow condition. In both cases, if we are using a subroutine to manage the queue, we need to report success or failure to the calling program. As with the stack, we will use R5 for this purpose.

The test for underflow is straightforward. We saw from Figure 8.26 that FRONT = REAR, the queue is empty. Our code to test for underflow is erefore

if

th

AND R5,R5,#0 ; Initialize R5 to 0 NOT R2,R3

ADD R2,R2,#1 ; R2 contains negative of R3 ADD R2,R2,R4

BRz UNDERFLOW

; code to remove the front of the queue and return success.

UNDERFLOW ADD R5,R5,#1 RET

That is, we first check to see if the queue is empty, that is, if R3 = R4. If so, we branch to UNDERFLOW, where we set R5 to failure, restore R1, and return. If not, carry out the code to remove the front of the queue.

The test for overflow is similar. To insert an element to the back of the queue, we first increment the REAR pointer. If that causes FRONT = REAR, then the queue already contains n-1 elements, which means it is full so we can- not insert any more elements. We decrement the REAR pointer, set R5 to 1, and return.

##### !.4.’ The Complete Story

We conclude our attention to queues with a subroutine that allows elements to be removed from the front or inserted into the rear of the queue, wraps around when one of the pointers reaches the last element, and returns with a report of success (R5 = 0) or failure (R5 = 1) depending on whether the access succeeds or the access fails due to an underflow or overflow condition.

To make this concrete, we will tie this subroutine to the queue of Figure 8.25, where we have allocated locations x8000 to x8005 for our queue, x8000 being the FIRST location and x8005 being the LAST location.

To insert, we first have to make sure the queue is not full. To do that, we increment the REAR pointer (R4) and then test REAR=FRONT. If the REAR pointer was initially x8005, we increment REAR by setting it to x8000; that is, we need to wrap around. If the queue is full, we need to set REAR back to its original value, and return, reporting failure (R5 = 1). If the queue is not full, we store the item we wish to insert (which is in R0) in REAR, and return, reporting success (R5 = 0).

To remove, we first make sure the queue is not empty by testing whether REAR=FRONT. If REAR=FRONT, the queue is empty, so we return, reporting failure. If REAR is not the same as FRONT, the queue is not empty, so we can remove the front element. To do this, we first test to see if FRONT=x8005. If it is, we set FRONT=x8000. If it isn’t, we increment FRONT. In both cases, we then load the value from that memory location into R0, and return, reporting success.

Figure 8.27 shows the complete subroutine.

00 ;Input: R0 for item to be inserted, R3 is FRONT, R4 is REAR 01 ;Output: R0 for item to be removed

02 ;

03 INSERT ST R1,SaveR1 ; Save register we need 04 AND R5,R5,#0 ; Set R5 to success code

1. ; Initialization complete
2. LD R1,NEG\_LAST
3. ADD R1,R1,R4 ; R1 = REAR MINUS x8005
4. BRnp SKIP1 ; SKIP WRAP AROUND
5. LD R4,FIRST ; WRAP AROUND, R4=x8000

0A BR SKIP2

0B SKIP1 ADD R4,R4,#1 ; NO WRAP AROUND, R4=R4+1

0C SKIP2 NOT R1,R4

0D ADD R1,R1,#1 ; R1= NEG REAR

0E ADD R1,R1,R3 ; R1= FRONT-REAR

0F BRz FULL

1. STR R0,R4,#0 ; DO THE INSERT
2. BR DONE
3. FULL LD R1,NEG\_FIRST
4. ADD R1,R1,R4 ; R1 = REAR MINUS x8000
5. BRnp SKIP3
6. LD R4,LAST ; UNDO WRAP AROUND, REAR=x8005
7. BR SKIP4
8. SKIP3 ADD R4,R4,#-1 ; NO WRAP AROUND, R4=R4-1
9. SKIP4 ADD R5,R5,#1 ; R5=FAILURE
10. BR DONE

1A ;

1B REMOVE ST R1,SaveR1 ; Save register we need 1C AND R5,R5,#0 ; Set R5 to success code

1D ; Initialization complete

1E NOT R1,R4

1F ADD R1,R1,#1 ; R1= NEG REAR

1. ADD R1,R1,R3 ; R1= FRONT-REAR
2. BRz EMPTY
3. LD R1, NEG\_LAST
4. ADD R1,R1,R3 ; R1= FRONT MINUS x8005
5. BRnp SKIP5
6. LD R3, FIRST ; R3=x8000
7. BR SKIP6

27 SKIP5 ADD R3,R3,#1 ; R3=R3+1

1. SKIP6 LDR R0,R3,#0 ; DO THE REMOVE
2. BR DONE

2A EMPTY ADD R5,R5.#1 ; R5=FAILURE

2B DONE LD R1,SaveR1 ; Restore register 2C RET

2D FIRST .FILL x8000

2E NEG\_FIRST .FILL x8000

2F LAST .FILL x8005

1. NEG\_LAST .FILL x7FFB
2. SaveR1 .BLKW 1

**Figure !.$7 The complete queue subroutine.**

## !.’ Character Strings

Our final data structure: the character string!

The last data structure we will study in this chapter is the character string, where a sequence of keyboard characters (letters, digits, and other symbols) is organized as a one-dimensional array of ASCII codes, usually representing a person’s name, address, or some other alphanumeric string. Figure 8.28 shows a character string representing the name of the famous late Stanford professor Bill Linvill, stored in 13 consecutive words of memory, starting at location x5000. The ASCII code for each letter of his name is stored in a separate word of mem- ory. Since an ASCII code consists of one byte of memory, we add a leading x00 to each location. For example, x5000 contains x0042 since the ASCII code for a capital B is x42. We need 13 memory locations, one word for each of the 11 letters in his name, one word for the ASCII code x20 representing the space between his first and last names, and finally the null character x0000 to indicate that we have reached the end of the character string. Different alphanumeric strings require character strings of different lengths, but that is no problem since we allocate as many words of memory as are needed, followed by the null character x0000 to indicate the end of the character string.

x5000

|  |
| --- |
| x0042 |
| x0069 |
| x006C |
| x006C |
| x0020 |
| x004C |
| x0069 |
| x006E |
| x0076 |
| x0069 |
| x006C |
| x006C |
| x0000 |

**Figure !.$! Character string representing the name “Bill Linvill.”**

A common use ofa character string is to identifya body of information asso- ciated with a particular person. Figure 8.29 shows such a body of information (often called a personnel record) associated with an employee of a company.

|  |
| --- |
| x004A |
| x006F |
| x006E |
| x0065 |
| x0073 |
| x0000 |

|  |
| --- |
| x004D |
| x0061 |
| x0072 |
| x0079 |
| x0000 |

x4000 x4001 x4002 x4003 x4004 x4005

|  |
| --- |
| x6000 |
| x4508 |
| xCA9B |
| $84,000 |
| 4 |
| x8E25 |

x6000 x6001 x6002 x6003 x6004 x6005

x4508 x4509 x450A x450B x450C

xCA9B xCA9C xCA9D xCA9E xCA9F xCAA0 xCAA1 xCAA2 xCAA3

x8E25 x8E26 x8E27 x8E28 x8E29 x8E2A x8E2B x8E2C x8E2D

**Figure !.$# Mary Jones’ personnel record.**

|  |
| --- |
| x0030 |
| x0031 |
| x0032 |
| x0036 |
| x0035 |
| x0034 |
| x0036 |
| x0032 |
| x0031 |

|  |
| --- |
| x0045 |
| x006E |
| x0067 |
| x0069 |
| x006E |
| x0065 |
| x0065 |
| x0072 |
| x0000 |

Our example personnel record consists of six words of sequential memory, starting at location x4000, as follows:

* 1. The first word contains the starting address of a character string containing the person’s last name. The pointer in location x4000 is the address x6000. The six-word character string, starting at location x6000, contains the ASCII code for “Jones,” terminated with the null character.
  2. The second word, at x4001, contains a pointer to the character string of the person’s first name, in this case “Mary,” starting at location x4508.
  3. The third word, at x4002, contains a pointer (xCA9B) to her nine-digit social security number, the unique identifier for all persons working in the United States.
  4. The fourth word, at x4003, contains her salary (in thousands of dollars).
  5. The fifth word contains how long she has worked for the company.
  6. The sixth word is a pointer (x8E25) to the character string identifying her job title, in this case “Engineer.”

In summary, an employee named Mary Jones, social security number 012654621, an Engineer, has been with the company four years and earns

$84,000/year salary.

One can write computer programs that examine employee records looking for various personnel information. For example, if one wanted to know an employee’s salary, a program could examine employee records, looking for that employee. The program would call a subroutine that compares the character string repre- senting an employee’s social security number with the characters of the social security number of the person the subroutine is searching for. If all the characters match, the subroutine would return a success code (R5 = 0), and the program

STRCMP ST R0,SaveR0

ST R1,SaveR1

ST R2,SaveR2

ST R3,SaveR3

;

AND R5,R5,#0 ; R5 <-- Match

;

NEXTCHAR LDR R2,R0,#0 ; R2 contains character from 1st string LDR R3,R1,#0 ; R3 contains character from 2nd string BRnp COMPARE ; String is not done, continue comparing ADD R2,R2,#0

BRz DONE ; If both strings done, match found COMPARE NOT R2,R2

ADD R2,R2,#1 ; R2 contains negative of character ADD R2,R2,R3 ; Compare the 2 characters

BRnp FAIL ; Not equal, no match ADD R0,R0,#1

ADD R1,R1,#1

BRnzp NEXTCHAR ; Move on to next pair of characters

;

FAIL ADD R5,R5,#1 ; R5 <-- No match

;

DONE LD R0,SaveR0 LD R1,SaveR1

LD R2,SaveR2

LD R3,SaveR3 RET

;

SaveR0 .BLKW 1

SaveR1 .BLKW 1

SaveR2 .BLKW 1

SaveR3 .BLKW 1

**Figure !.&) Subroutine to compare two character strings.**

would go on to read the salary information in the fourth word of the personnel record. If all the characters do not match, the subroutine would return a failure code (R5 = 1), and the program would call the subroutine with the starting address of another employee’s social security number.

Figure 8.30 is a subroutine that compares two character strings to see if they identical.

are

***Another Example: A Character String Representing an “Integer.”*** We can also represent arbitrarily long integers by means of character strings. For example, Figure 8.31 is a character string representing the integer 79,245.

Figure 8.32 is a subroutine that examines such a character string to be sure in fact all ASCII codes represent decimal digits. If all the entries in the charac- tring are ASCII codes of decimal digits (between x30 and x39), the subroutine rns success (R = 0). If not, the subroutine returns failure (R5 = 1).

that

ter s

retu

|  |
| --- |
| x0037 |
| x0039 |
| x0032 |
| x0034 |
| x0035 |
| x0000 |

**Figure !.&1 A character string representing the integer 7#,$4’, with one ASCII code per decimal digit.**

; Input: R0 contains the starting address of the character string

; Output: R5=0, success; R5=1, failure.

;

TEST\_INTEGER ST R1,SaveR1 ; Save registers needed by subroutine ST R2,SaveR2

ST R3,SaveR3

ST R4,SaveR4

;

AND R5,R5,#0 ; Initialize success code to R5=0, success LD R2,ASCII\_0 ; R2=xFFD0, the negative of ASCII code x30 LD R3,ASCII\_9 ; R3=xFFC7, the negative of ASCII code x39

;

NEXT\_CHAR LDR R1,R0,#0 ; Load next character BRz SUCCESS

ADD R4,R1,R2

BRn BAD ; R1 is less than x30, not a decimal digit ADD R4,R1,R3

BRp BAD ; R1 is greater than x39, not a decimal digit ADD R0,R0,#1 ; Character good! Prepare for next character BR NEXT\_CHAR

;

BAD ADD R5,R5,#1 ; R5 contains failure code SUCCESS LD R4,SaveR4 ; Restore registers

LD R3,SaveR3

LD R2,SaveR2

LD R1,SaveR1 RET

ASCII\_0 .FILL xFFD0

ASCII\_9 .FILL xFFC7 SaveR1 .BLKW 1

SaveR2 .BLKW 1

SaveR3 .BLKW 1

SaveR4 .BLKW 1

**Figure !.&$ Subroutine to determine if a character string represents an integer.**

**Exercises**

**!.1** What are the defining characteristics of a stack?

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

**!.$** The LC-3 ISA has been augmented with the following push and pop instructions. Push Rn pushes the value in Register n onto the stack. Pop Rn removes a value from the stack and loads it into Rn. The following figure shows a snapshot of the eight registers of the LC-3 BEFORE and AFTER the following six stack operations are performed. Identify (a)–(d).

**BEFORE AFTER**

RO PUSH R4 RO

|  |
| --- |
| xOOOO |
| x1111 |
| x2222 |
| x3333 |
| x4444 |
| x5555 |
| x’’’’ |
| x(((( |

|  |
| --- |
| x1111 |
| x1111 |
| x3333 |
| x3333 |
| x4444 |
| x5555 |
| x’’’’ |
| x4444 |

R1 PUSH (a) R1

R2 POP (b) R2

R3 PUSH (c) R3

R4 POP R2 R4

R5 POP (d) R5

R’ R’

R( R(

**!.4** Write a function that implements another stack function, peek. Peek returns the value of the first element on the stack without removing the element from the stack. Peek should also do underflow error checking. (Why is overflow error checking unnecessary?)

**!.&** How would you check for underflow and overflow conditions if you implemented a stack using the model in Figure 8.8? Rewrite the PUSH and POP routines to model a stack implemented as in Figure 8.8, that is, one in which the data entries move with each operation.

**!.’** Rewrite the PUSH and POP routines such that the stack on which they operate holds elements that take up two memory locations each.

**!.(** Rewrite the PUSH and POP routines to handle stack elements of arbitrary sizes.

**!.!** The following operations are performed on a stack:

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

* + 1. What does the stack contain after the PUSH F?
    2. 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

* + 1. What does the stack contain now?

**!.)** The input stream of a stack is a list of all the elements we pushed onto the stack, in the order that we pushed them. The input stream from Exercise 8.8 was ABCDEFGHIJKLM

The output stream is a list of all the elements that are popped off the stack, in the order that they are popped off.

1. What is the output stream from Exercise 8.8?

*Hint:* BDE …

1. If the input stream is ZYXWVUTSR, create a sequence of pushes and pops such that the output stream is YXVUWZSRT.
2. If the input stream is ZYXW, how many different output streams can be created?

\***!.1)** 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 LDR NOT  ADD | | RESTOR R7, R0, R7, R7  R7, R7, | | #1  #1 | ; | <-- Breakpoint here | |
| ADD  BRz JSR ADD  BRp ADD  BR | | R7, R2, BAD EXAMINE R1, R1, RESTORE R3, R3, AGAIN | | R7  #0  #1 | ;  ;  ;  ; | Compare current color to neighbor’s color Recursively examine the coloring of next neighbor  If neighbor returns R1=1, this node should return R1=1  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 RET | R6, | R6, | #1 | | | |

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  x4012 | R0  x6200 | R2  x0042 | R7  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.

|  |
| --- |
|  |
| x0042 |
| x6200 |
|  |
|  |
|  |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x6100 |  | x6200 |  | x6300 |
| x6101 |  | x6201 | x0052 | x6301 |
| x6102 |  | x6202 |  | x6302 |
| x6103 |  | x6203 |  | x6303 |
| x6104 |  | x6204 |  | x6304 |
| x6105 |  | x6205 |  | x6305 |
| x6106 |  | x6206 |  | x6306 |
|  |  |  |  |  |
| x6400 |  | x6500 |  |  |
| x6401 |  | x6501 |  |  |
| x6402 |  | x6502 |  |  |
| x6403 |  | x6503 |  |  |
| x6404 |  | x6504 |  |  |
| x6405 |  | x6505 |  |  |
| x6406 |  | x6506 |  |  |

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

**!.11** The following program needs to be assembled and stored in LC-3 memory. How many LC-3 memory locations are required to store the assembled program?

.ORIG x4000 AND R0,R0,#0 ADD R1,R0,#0 ADD R0,R0,#4 LD R2,B

* 1. LDR R3,R2,#0 ADD R1,R1,R3 ADD R2,R2,#1 ADD R0,R0,#-1

BRnp A

JSR SHIFTR ADD R1,R4,#0 JSR SHIFTR ST R4,C TRAP x25

* 1. .BLKW 1
  2. .BLKW 1

.END

How many memory locations are required to store the assembled program?

What is the address of the location labeled C?

Before the program can execute, the location labeled B must be loaded by some external means. You can assume that happens before this program starts executing. You can also assume that the subroutine starting at location SHIFTR is available for this program to use. SHIFTR takes the value in R1, shifts it right one bit, and stores the result in R4.

After the program executes, what is in location C?

\***!.1$** Many cities, like New York City, Stockholm, Konigsberg, etc., consist of several areas connected by bridges. The following figure shows a map of FiveParts, a city made up of five areas A, B, C, D, E, with the areas connected by nine bridges as shown.

C

E

B

D

A

The following program prompts the user to enter two areas and then stores the number of bridges from the first area to the second in location x4500. Your job: On the next page, design the data structure for the city of FiveParts that the following program will use to count the number of bridges between two areas.

.ORIG x3000 LEA R0, FROM TRAP x22

TRAP x20 ; Inputs a char without banner NOT R1, R0

ADD R1, R1, #1 LEA R0, TO TRAP x22

TRAP x20 NOT R0, R0

ADD R0, R0, #1 AND R5, R5, #0 LDI R2, HEAD

SEARCH BRz DONE

LDR R3, R2, #0 ADD R7, R1, R3 BRz FOUND\_FROM LDR R2, R2, #1 BRnzp SEARCH

FOUND\_FROM ADD R2, R2, #2 NEXT\_BRIDGE LDR R3, R2, #0

BRz DONE

LDR R4, R3, #0 ADD R7, R0, R4 BRnp SKIP

ADD R5, R5, #1 ; Increment Counter

SKIP ADD R2, R2, #1 BRnzp NEXT\_BRIDGE

DONE STI R5, ANSWER HALT

HEAD .FILL x3050

ANSWER .FILL x4500

FROM .STRINGZ "FROM: "

TO .STRINGZ "TO: "

.END

Your job is to provide the contents of the memory locations that are needed to specify the data structure for the city of FiveParts, which is needed by the program on the previous page. We have given you the HEAD pointer for the data structure and in addition, five memory locations and the contents of those five locations. We have also supplied more than enough sequential memory locations after each of the five to enable you to finish the job. Use as many of these memory locations as you need.

**!.1&** Our code to compute n factorial worked for all positive integers n. As promised in the text, your assignment here: Augment the iterative solution to FACT to also work for 0!.

**!.14** As you know, the LC-3 ADD instruction adds 16-bit 2’s complement integers. If we wanted to add 32-bit 2’s complement integers, we could do that with the program shown next. Note that the program requires calling subroutine X, which stores into R0 the carry that results from adding R1 and R2.

Fill in the missing pieces of both the program and the subroutine X, as identified by the empty boxes. Each empty box corresponds to **one** instruction or the operands of **one** instruction.

x4000 x4001 x4002 x4003 x4004 x4005 x4006

x4100 x4101 x4102 x4103 x4104 x4105 x4106

x3100 x3101 x3102 x3103 x3104 x3105 x3106

x3050

xA243 xA244 xA245 xA246 xA247 xA248 xA249

|  |
| --- |
| x0041 |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| x0042 |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| x0043 |
|  |
|  |
|  |
|  |
|  |
|  |

xBBBB xBBBC xBBBD xBBBE xBBBF xBBC0 xBBC1

|  |
| --- |
| x0044 |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| x0045 |
|  |
|  |
|  |
|  |
|  |
|  |

Note that a 32-bit operand requires two 16-bit memory locations. A

32-bit operand Y has Y[15:0] stored in address A, and Y[31:16] stored in address A+1.

.ORIG x3000

LEA R3, NUM1 LEA R4, NUM2 LEA R5, RESULT LDR R1, R3, #0 LDR R2, R4, #0 ADD R0, R1, R2 STR R0, R5, #0

(a)

LDR LDR ADD R0, R1, R2

(d) TRAP x25

X ST R4, SAVER4 AND R0, R0, #0 AND R4, R1, R2

BRn (e)

ADD R1, R1, #0

BRn (f)

ADD BRn ADDING

BRnzp EXIT ADDING ADD R4, R1, R2

BRn EXIT

(g)

LABEL ADD R0, R0, #1 EXIT LD R4, SAVER4

RET

NUM1 .BLKW 2

NUM2 .BLKW 2

RESULT .BLKW 2

SAVER4 .BLKW 1

.END

\***!.1’** A program encounters a breakpoint and halts. The computer operator does not change the state of the computer in any way but immediately presses the **run** button to resume execution.

The following table shows the contents of MAR and MDR for the first nine memory accesses that the LC-3 performs after resuming execution. Your job: Fill in the missing entries.

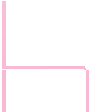
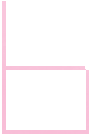
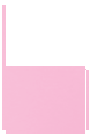
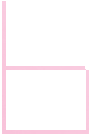
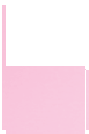
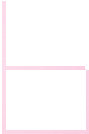
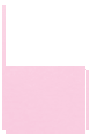
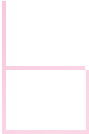
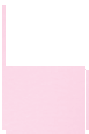
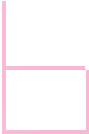
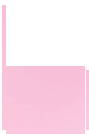
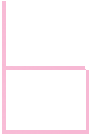
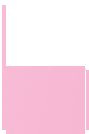
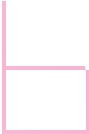
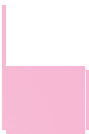
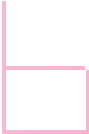
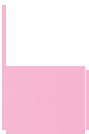
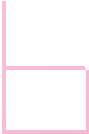
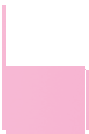
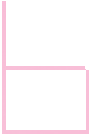
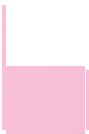
|  |  |
| --- | --- |
| MAR | MDR |
|  | x5020 |
|  | xF0F0 |
|  |  |
| x2000 | x020A |
|  | x040A |
|  | x61FE |
|  |  |
|  | xC1C0 |
| x4002 | xF025 |

1st: 2nd: 3rd: 4th: 5th: 6th: 7th: 8th: 9th:

CHAPTER

9

I/O



p to now we have completely ignored the details of input and output, that is, how the computer actually gets information from the keyboard (input), and

**U**

how the computer actually delivers information to the monitor (output). Instead we have relied on the TRAP instruction (e.g., TRAP x23 for input and TRAP x21 for output) to accomplish these tasks. The TRAP instruction enables us to tell the operating system what we need done by means of a trap vector, and we trust the operating system to do it for us.

The more generic term for our TRAP instruction is *system call* because the TRAP instruction is calling on the operating system to do something for us while allowing us to remain completely clueless as to how it gets done. Now we are ready to examine how input and output actually work in the LC-3, what happens when the user program makes a system call by invoking the TRAP instruction, and how it all works under the control of the operating system.

We will start with the actual physical structures that are required to cause input and output to occur. But before we do that, it is useful to say a few words about the operating system and understand a few basic concepts that have not been important so far but become very important when considering what the operating system needs to do its job.

You may be familiar with Microsoft’s various flavors of Windows, Apple’s MacOS, and Linux. These are all examples of operating systems. They all have the same goal: to optimize the use of all the resources of the computer system while making sure that no software does harmful things to any program or data that it has no right to mess with. To better understand their job, we need to under- stand the notions of privilege and priority and the layout of the memory address space (i.e., the regions of memory and the purpose of each).

## !.1 Privilege, Priority, and the Memory Address Space

##### !.1.1 Privilege and Priority

Two very different (we often say orthogonal) concepts associated with computer processing are *privilege* and *priority*.

!.1.1.1 Privilege

Privilege is all about the right to do something, such as execute a particular instruction or access a particular memory location. Not all computer programs have the right to execute all instructions. For example, if a computer system is shared among many users and the ISA contains a HALT instruction, we would not want any random program to execute that HALT instruction and stop the computer. If we did, we would have some pretty disgruntled users on our hands. Similarly, some memory locations are only available to the operating system. We would not want some random program to interfere with the data structures or code that is part of the operating system, which would in all likelihood cause the entire system to crash. In order to make sure neither of these two things happens, we designate every computer program as either privileged or unprivileged. We often say *supervisor privilege* to indicate privileged. We say a program is executing in Supervisor mode to indicate privileged, or User mode to indicate unprivileged. If a program is executing in Supervisor mode, it can execute all instructions and access all of memory. If a program is executing in User mode, it cannot. If a program executing in User mode tries to execute an instruction or access a memory location that requires being in Supervisor mode, the computer will not allow it.

!.1.1.# Priority

Priority is all about the urgency of a program to execute. Every program is assigned a priority, specifying its urgency as compared to all other programs. This allows programs of greater urgency to interrupt programs of lesser urgency. For example, programs written by random users may be assigned a priority of 0. The keyboard may be asigned a priority of 4, and the fact that the computer is plugged into a source of energy like a wall outlet may be assigned a priority of 6. If that is the case, a random user program would be interrupted if someone sitting at a keyboard wanted to execute a program that caused data to be input into the computer. And that program would be interrupted if someone pulled the power cord out of the wall outlet, causing the computer to quickly lose its source of energy. In such an event, we would want the computer to execute some operating system program that is provided specifically to handle that situation.

!.1.1.$ Two Orthogonal Notions

We said privilege and priority are two orthogonal notions, meaning they have nothing to do with each other. We humans sometimes have a problem with that as we think of fire trucks that have the privilege of ignoring traffic lights because

**9.1** Privilege, Priority, and the Memory Address Space **315**

they must quickly reach the fire. In our daily lives, we often are given privileges because of our greater sense of urgency. Not the case with computer systems.

For example, we can have a user program that is tied to a physics experiment that needs to interrupt the computer at a specific instance of time to record infor- mation being generated by the physics experiment. If the user program does not pre-empt the program running at that instant of time, the data generated by the experiment may be lost. This is a user program, so it does not have supervisor privilege. But it does have a greater urgency, so it does have a higher priority.

Another example: The system administrator wants to execute diagnostic pro- grams that access all memory locations and execute all instructions as part of some standard preventive maintenance. The diagnostic program needs supervi- sor privilege to execute all instructions and access all memory locations. But it has no sense of urgency. Whether this happens at 1 a.m. or 2 a.m. is irrelevant, compared to the urgency of other programs that need access to the computer system exactly when they need it. The diagnostic program has privilege but no priority.

Finally, an example showing that even in human activity one can have priority but not privilege. Our friend Bob works in the basement of one of those New York City skyscrapers. He is about to go to the men’s room when his manager tells him to take a message immediately to the vice president on the 88th floor, and bring back a response. So Bob delays his visit to the men’s room and takes the elevator to the 88th floor. The vice president keeps him waiting, causing Bob to be concerned he might have an accident. Finally, the vice president gives his response, and Bob pushes the button to summon the elevator to take him back to the basement, in pain because he needs to go to the men’s room. While waiting for the elevator, another vice president appears, unlocks the executive men’s room, and enters. Bob is in pain, but he cannot enter the executive men’s room. Although he certainly has the priority, he does not have the privilege!

!.1.1.4 The Processor Status Register (PSR)

Each program executing on the computer has associated with it two very impor- tant registers. The Program Counter (PC) you are very familiar with. The other register, the Processor Status Register (PSR), is shown in Figure 9.1. It contains the privilege and priority assigned to that program.

Bit [15] specifies the privilege, where PSR[15]=0 means supervisor privi- e, and PSR[15]=1 means unprivileged. Bits [10:8] specify the priority level

leg

(P

1. of the program. The highest priority level is 7 (PL7), the lowest is PL0. The PSR also contains the current values of the condition codes, as shown igure 9.1. We will see in Section 9.4 why it is important that the condition

in F

codes are included in the PSR.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

PSR

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Pr  Priv |  | PL n  Priority |  | N  con | Z  d co | P  des |

**Figure !.1 Processor status register (PSR).**

##### !.1.# Organization of Memory

Figure 9.2 shows the layout of the LC-3 memory.

You know that the LC-3 has a 16-bit address space; ergo, memory locations from x0000 to xFFFF. Locations x0000 to x2FFF are privileged memory loca- tions. They contain the various data structures and code of the operating system. They require supervisor privilege to access. They are referred to as *system space*. Locations x3000 to xFDFF are unprivileged memory locations. Supervisor privilege is not required to access these memory locations. All user programs and

data use this region of memory. The region is often referred to as *user space*.

Addresses xFE00 to xFFFF do not correspond to memory locations at all. That is, the last address of a memory location is xFDFF. Addresses xFE00 to xFFFF are used to identify registers that take part in input and output functions and some special registers associated with the processor. For example, the PSR is assigned address xFFFC, and the processor’s Master Control Register (MCR) is assigned address xFFFE. The benefit of assigning addresses from the memory address space will be discussed in Section 9.2.1.2. The set of addresses from xFE00 to xFFFF is usually referred to as the I/O page since most of the addresses are used for identifying registers that take part in input or output functions. Access to those registers requires supervisor privilege.

Finally, note that Figure 9.2 shows two stacks, a *supervisor stack* in system space and a *user stack* in user space. The supervisor stack is controlled by the



###### x0000

|  |
| --- |
| System Space |
| Supervisor Stack |
| User Space |
| User Stack |
| I/O Page |

SSP

x3000

USP

xFE00 xFFFF

**Figure !.# Regions of memory.**

###### Privileged Memory

operating system and requires supervisor privilege to access. The user stack is controlled by the user program and does not require privilege to access.

Each has a stack pointer, Supervisor Stack Pointer (SSP) and User Stack Pointer (USP), to indicate the top of the stack. Since a program can only execute in Supervisor mode or User mode at any one time, only one of the two stacks is active at any one time. Register 6 is generally used as the stack pointer (SP) for the active stack. Two registers, Saved SSP and Saved USP, are provided to save the SP not in use. When privilege changes, for example, from Supervisor mode to User mode, the SP is stored in Saved SSP, and the SP is loaded from Saved USP.

## !.# Input/Output

Input and output devices (keyboards, monitors, disks, or kiosks at the shopping mall) all handle input or output data using registers that are tailored to the needs of each particular input or output device. Even the simplest I/O devices usually need at least two registers: one to hold the data being transferred between the device and the computer, and one to indicate status information about the device. An example of status information is whether the device is available or is it still busy processing the most recent I/O task.

##### !.#.1 Some Basic Characteristics of I/O

All I/O activity is controlled by instructions in the computer’s ISA. Does the ISA need special instructions for dealing with I/O? Does the I/O device execute at the same speed as the computer, and if not, what manages the difference in speeds? Is the transfer of information between the computer and the I/O device initiated by a program executing in the computer, or is it initiated by the I/O device? Answers to these questions form some of the basic characteristics of I/O activity.

!.#.1.1 Memory-Mapped I/O vs. Special I/O Instructions

An instruction that interacts with an input or output device register must identify the particular input or output device register with which it is interacting. Two schemes have been used in the past. Some computers use special input and output instructions. Most computers prefer to use the same data movement instructions that are used to move data in and out of memory.

The very old PDP-8 (from Digital Equipment Corporation, more than 50 years ago—1965) is an example of a computer that used special input and output instructions. The 12-bit PDP-8 instruction contained a three-bit opcode. If the opcode was 110, an I/O instruction was indicated. The remaining nine bits of the PDP-8 instruction identified which I/O device register and what operation was to be performed.

Most computer designers prefer not to specify an additional set of instructions for dealing with input and output. They use the same data movement instructions that are used for loading and storing data between memory and the general pur- pose registers. For example, a load instruction (LD, LDI, or LDR), in which the

source address is that of an input device register, is an input instruction. Similarly, a store instruction (ST, STI, or STR) in which the destination address is that of an output device register is an output instruction.

Since programmers use the same data movement instructions that are used for memory, every input device register and every output device register must be uniquely identified in the same way that memory locations are uniquely identified. Therefore, each device register is assigned an address from the memory address space of the ISA. That is, the I/O device registers are *mapped* to a set of addresses that are allocated to I/O device registers rather than to memory locations. Hence the name *memory-mapped I/O*.

The original PDP-11 ISA had a 16-bit address space. All addresses wherein bits [15:13] = 111 were allocated to I/O device registers. That is, of the 2 addresses, only 57,344 corresponded to memory locations. The remaining 2 were memory-mapped I/O addresses.

16

13

The LC-3 uses memory-mapped I/O. As we discussed in Section 9.1.2, addresses x0000 to xFDFF refer to actual memory locations. Addresses xFE00 to xFFFF are reserved for input/output device registers. Table A.3 lists the memory- mapped addresses of the LC-3 device registers that have been assigned so far. Future uses and future sales of LC-3 microprocessors may require the expansion of device register address assignments as new and exciting applications emerge!

!.#.1.# Asynchronous vs. Synchronous

Most I/O is carried out at speeds very much slower than the speed of the processor. A typist, typing on a keyboard, loads an input device register with one ASCII code every time he/she types a character. A computer can read the contents of that device register every time it executes a load instruction, where the operand address is the memory-mapped address of that input device register.

Many of today’s microprocessors execute instructions under the control of a clock that operates well in excess of 2 GHz. Even for a microprocessor operating at only 2 GHz, a clock cycle lasts only 0.5 nanoseconds. Suppose a processor executed one instruction at a time, and it took the processor ten clock cycles to execute the instruction that reads the input device register and stores its contents. At that rate, the processor could read the contents of the input device register once every 5 nanoseconds. Unfortunately, people do not type fast enough to keep this processor busy full-time reading characters. *Question:* How fast would a person have to type to supply input characters to the processor at the maximum rate the processor can receive them?

We could mitigate this speed disparity by designing hardware that would accept typed characters at some slower fixed rate. For example, we could design a piece of hardware that accepts one character every 200 million cycles. This would require a typing speed of 100 words/minute, assuming words on average consisted of five letters, which is certainly doable. Unfortunately, it would also require that the typist work in lockstep with the computer’s clock. That is not acceptable since the typing speed (even of the same typist) varies from moment to moment.

What’s the point? The point is that I/O devices usually operate at speeds very different from that of a microprocessor, and not in lockstep. We call this

latter characteristic *asynchronous*. Most interaction between a processor and I/O is asynchronous. To control processing in an asynchronous world requires some protocol or *handshaking* mechanism. So it is with our keyboard and monitor. In the case of the keyboard, we will need a one-bit status register, called a *flag,* to indicate if someone has or has not typed a character. In the case of the monitor, we will need a one-bit status register to indicate whether or not the most recent character sent to the monitor has been displayed, and so the monitor can be given another character to display.

the

These flags are the simplest form of *synchronization*. A single flag, called

*ready bit*, is enough to synchronize the output of the typist who can type

characters at the rate of 100 words/minute with the input to a processor that can accept these characters at the rate of 200 million characters/second. Each time the typist types a character, the ready bit is set to 1. Each time the computer reads a character, it clears the ready bit. By examining the ready bit before reading a character, the computer can tell whether it has already read the last character typed. If the ready bit is clear, no characters have been typed since the last time the computer read a character, and so no additional read would take place. When the computer detects that the ready bit is set, it could only have been caused by a **new** character being typed, so the computer would know to again read a character. The single ready bit provides enough handshaking to ensure that the asyn- chronous transfer of information between the typist and the microprocessor can

be carried out accurately.

If the typist could type at a constant speed, and we did have a piece of hard- ware that would accept typed characters at precise intervals (e.g., one character every 200 million cycles), then we would not need the ready bit. The computer would simply know, after 200 million cycles of doing other stuff, that the typist had typed exactly one more character, and the computer would read that charac- ter. In this hypothetical situation, the typist would be typing in lockstep with the processor, and no additional synchronization would be needed. We would say the computer and typist were operating *synchronously*. That is, the input activity was synchronous.

!.2.#.$ Interrupt-Driven vs. Polling

The processor, which is computing, and the typist, who is typing, are two separate entities. Each is doing its own thing. Still, they need to interact; that is, the data that is typed has to get into the computer. The issue of *interrupt-driven* vs. *polling* is the issue of who controls the interaction. Does the processor do its own thing until being interrupted by an announcement from the keyboard, “Hey, a key has been struck. The ASCII code is in the input device register. You need to read it.” This is called *interrupt-driven I/O*, where the keyboard controls the interaction. Or, does the processor control the interaction, specifically by interrogating (usually, again and again) the ready bit until it (the processor) detects that the ready bit is set. At that point, the processor knows it is time to read the device register. This second type of interaction when the processor is in charge is called *polling*, since the ready bit is polled by the processor, asking if any key has been struck.

Section 9.2.2.2 describes how polling works. Section 9.4 explains interrupt- driven I/O.

##### !.#.# Input from the Keyboard

!.#.#.1 Basic Input Registers (KBDR and KBSR)

We have already noted that in order to handle character input from the keyboard, we need two things: a data register that contains the character to be input and a synchronization mechanism to let the processor know that input has occurred. The synchronization mechanism is contained in the status register associated with the keyboard.

These two registers are called the *keyboard data register* (KBDR) and the *keyboard status register* (KBSR). They are assigned addresses from the memory address space. As shown in Table A.3, address xFE02 is assigned to the KBDR; address xFE00 is assigned to the KBSR.

1# 8 7

0

KBDR

1# 14

0

KBSR

**Figure !.$ Keyboard device registers.**

Even though a character needs only 8 bits and the synchronization mecha- nism needs only 1 bit, it is easier to assign 16 bits (like all memory addresses in the LC-3) to each. In the case of KBDR, bits [7:0] are used for the data, and bits [15:8] contain x00. In the case of KBSR, bit [15] contains the synchroniza- tion mechanism, that is, the ready bit. Figure 9.3 shows the two device registers needed by the keyboard.

!.#.#.# The Basic Input Service Routine

KBSR[15] controls the synchronization of the slow keyboard and the fast pro- cessor. When a key on the keyboard is struck, the ASCII code for that key is loaded into KBDR[7:0], and the electronic circuits associated with the keyboard automatically set KBSR[15] to 1. When the LC-3 reads KBDR, the electronic circuits associated with the keyboard automatically clear KBSR[15], allowing another key to be struck. If KBSR[15] = 1, the ASCII code corresponding to the last key struck has not yet been read, and so the keyboard is disabled; that is, no key can be struck until the last key is read.

If input/output is controlled by the processor (i.e., via polling), then a pro- m can repeatedly test KBSR[15] unt l it notes that the bit is set. At that point, processor can load the ASCII code contained in KBDR into one of the LC-3 registers. Since the processor only loads the ASCII code if KBSR[15] is 1, there is no danger of reading a single typed character multiple times. Furthermore, since the keyboard is disabled until the previous code is read, there is no danger of the processor missing characters that were typed. In this way, KBSR[15] provides

gra i

the

the mechanism to guarantee that each key typed will be loaded exactly once.

The following input routine loads R0 with the ASCII code that has been entered through the keyboard and then moves on to the NEXT TASK in the program.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 | START | LDI | R1, A | ; Test for |
| 02 |  | BRzp | START | ; character input |
| 03 |  | LDI | R0, B |  |
| 04 |  | BRnzp | NEXT\_TASK | ; Go to the next task |
| 05 | A | .FILL | xFE00 | ; Address of KBSR |
| 06 | B | .FILL | xFE02 | ; Address of KBDR |

As long as KBSR[15] is 0, no key has been struck since the last time the processor read the data register. Lines 01 and 02 comprise a loop that tests bit [15] of KBSR. Note the use of the LDI instruction, which loads R1 with the contents of xFE00, the memory-mapped address of KBSR. If the ready bit, bit [15], is clear, BRzp will branch to START and another iteration of the loop. When someone strikes a key, KBDR will be loaded with the ASCII code of that key, and the ready bit of KBSR will be set. This will cause the branch to fall through, and the instruction at line 03 will be executed. Again, note the use of the LDI instruction, which this time loads R0 with the contents of xFE02, the memory-mapped address of KBDR. The input routine is now done, so the program branches unconditionally to its NEXT TASK.

!.#.#.$ Implementation of Memory-Mapped Input

Figure 9.4 shows the additional data path required to implement memory-mapped input. You are already familiar, from Chapter 5, with the data path required to



GateMDR

16

16

LD.MDR

LD.MAR

R.W / READ

16

INPUT

16

2 2

MEM.EN, READ

INMUX

KBDR

KBSR

MEMORY

ADDR CONTROL LOGIC

MAR

MDR

**Figure !.4 Memory-mapped input.**

carry out the EXECUTE phase of the load instructions. Essentially three steps are required:

* 1. The MAR is loaded with the address of the memory location to be read.
  2. Memory is read, resulting in MDR being loaded with the contents at the specified memory location.
  3. The destination register (DR) is loaded with the contents of MDR.

In the case of memory-mapped input, the same steps are carried out, **except** instead of MAR being loaded with the address of a memory location, MAR is loaded with the address of a device register. Instead of the address control logic enabling memory to read, the address control logic selects the corresponding device register to provide input to the MDR.

##### !.#.$ Output to the Monitor

!.#.$.1 Basic Output Registers (DDR and DSR)

Output works in a way very similar to input, with DDR and DSR replacing the roles of KBDR and KBSR, respectively. DDR stands for Display Data Register, which drives the monitor display. DSR stands for Display Status Register. In the LC-3, DDR is assigned address xFE06. DSR is assigned address xFE04.

1# 8 7

0

DDR

1# 14

**Figure !.& Monitor device registers.**

0

DSR

As is the case with input, even though an output character needs only 8 bits and the synchronization mechanism needs only one bit, it is easier to assign 16 bits (like all memory addresses in the LC-3) to each output device register. In the case of DDR, bits [7:0] are used for data, and bits [15:8] contain x00. In the case of DSR, bit [15] contains the synchronization mechanism, that is, the ready bit. Figure 9.5 shows the two device registers needed by the monitor.

!.#.$.# The Basic Output Service Routine

DSR[15] controls the synchronization of the fast processor and the slow monitor display. When the LC-3 transfers an ASCII code to DDR[7:0] for outputting, the electronics of the monitor automatically clear DSR[15] as the processing of the contents of DDR[7:0] begins. When the monitor finishes processing the character on the screen, it (the monitor) automatically sets DSR[15]. This is a signal to the processor that it (the processor) can transfer another ASCII code to DDR for outputting. As long as DSR[15] is clear, the monitor is still processing the previous character, so the monitor is disabled as far as additional output from the processor is concerned.

If input/output is controlled by the processor (i.e., via polling), a program can repeatedly test DSR[15] until it notes that the bit is set, indicating that it is OK to write a character to the screen. At that point, the processor can store the ASCII code for the character it wishes to write into DDR[7:0], setting up the transfer of that character to the monitor’s display.

The following routine causes the ASCII code contained in R0 to be displayed on the monitor:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 | START | LDI | R1, A | ; Test to see if |
| 02 |  | BRzp | START | ; output register is ready |
| 03 |  | STI | R0, B |  |
| 04 |  | BRnzp | NEXT\_TASK |  |
| 05 | A | .FILL | xFE04 | ; Address of DSR |
| 06 | B | .FILL | xFE06 | ; Address of DDR |

Like the routine for KBDR and KBSR in Section 9.2.2.2, lines 01 and 02 repeat- edly poll DSR[15] to see if the monitor electronics is finished with the last character shipped by the processor. Note the use of LDI and the indirect access to xFE04, the memory-mapped address of DSR. As long as DSR[15] is clear, the monitor electronics is still processing this character, and BRzp branches to START for another iteration of the loop. When the monitor electronics finishes with the last character shipped by the processor, it automatically sets DSR[15] to 1, which causes the branch to fall through and the instruction at line 03 to be executed. Note the use of the STI instruction, which stores R0 into xFE06, the memory-mapped address of DDR. The write to DDR also clears DSR[15], dis- abling for the moment DDR from further output. The monitor electronics takes over and writes the character to the screen. Since the output routine is now done, the program unconditionally branches (line 04) to its NEXT TASK.

!.#.$.$ Implementation of Memory-Mapped Output

Figure 9.6 shows the additional data path required to implement memory-mapped output. As we discussed previously with respect to memory-mapped input, the mechanisms for handling the device registers provide very little additional complexity to what already exists for handling memory accesses.

In Chapter 5, you became familiar with the process of carrying out the EXECUTE phase of the store instructions.

* 1. The MAR is loaded with the address of the memory location to be written.
  2. The MDR is loaded with the data to be written to memory.
  3. Memory is written, resulting in the contents of MDR being stored in the specified memory location.

In the case of memory-mapped output, the same steps are carried out, tead of MAR being loaded with the address of a memory location, MAR

ins **excep**is**t**

lo ic

aded with the address of a device register. Instead of the address control log enabling memory to write, the address control logic asserts the load enable signal of DDR.

is **read**

Memory-mapped output also requires the ability to output device reg- ters. You saw in Section 9.2.3.2 that before the DDR could be loaded, the ready



GateMDR

16

16

LD.MDR

LD.MAR

16

R.W / WRITE

16

OUTPUT

16

2

MEM.EN, WRITE

LD.DDR

INMUX

DSR

DDR

MEMORY

ADDR CONTROL LOGIC

MAR

MDR

**Figure !.’ Memory-mapped output.**

bit had to be in state 1, indicating that the previous character had already fin- ished being written to the screen. The LDI and BRzp instructions on lines 01 and 02 perform that test. To do this, the LDI reads the output device register DSR, and BRzp tests bit [15]. If the MAR is loaded with xFE04 (the memory- mapped address of the DSR), the address control logic selects DSR as the input to the MDR, where it is subsequently loaded into R1, and the condition codes are set.

!.#.$.4 Example: Keyboard Echo

When we type at the keyboard, it is helpful to know exactly what characters we have typed. We can get this echo capability easily (without any sophisticated elec- tronics) by simply combining the two routines we have discussed. The result: The key typed at the keyboard is displayed on the monitor.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 01 | START | LDI | R1, KBSR | ; | Test for character input |
| 02 |  | BRzp | START |  |  |
| 03 |  | LDI | R0, KBDR |  |  |
| 04 | ECHO | LDI | R1, DSR | ; | Test output register ready |
| 05 |  | BRzp | ECHO |  |  |
| 06 |  | STI | R0, DDR |  |  |
| 07 |  | BRnzp | NEXT\_TASK |  |  |
| 08 | KBSR | .FILL | xFE00 | ; | Address of KBSR |
| 09 | KBDR | .FILL | xFE02 | ; | Address of KBDR |
| 0A | DSR | .FILL | xFE04 | ; | Address of DSR |
| 0B | DDR | .FILL | xFE06 | ; | Address of DDR |

##### !.#.4 A More Sophisticated Input Routine

In the example of Section 9.2.2.2, the input routine would be a part of a program being executed by the computer. Presumably, the program requires character input from the keyboard. But how does the person sitting at the keyboard know when to type a character? Sitting there, the person may wonder whether or not the program is actually running, or if perhaps the computer is busy doing something else.

To let the person sitting at the keyboard know that the program is waiting for input from the keyboard, the computer typically prints a message on the monitor. Such a message is often referred to as a *prompt*. The symbols that are displayed by your operating system (e.g., **%** or **C:**) or by your editor (e.g., **:**) are examples of prompts.

The program fragment shown in Figure 9.7 obtains keyboard input via polling as we have shown in Section 9.2.2.2. It also includes a prompt to let the person sitting at the keyboard know when it is time to type a key. Let’s examine this program fragment.

You are already familiar with lines 13 through 19 and lines 25 through 28, which correspond to the code in Section 9.2.3.4 for inputting a character via the keyboard and echoing it on the monitor.

You are also familiar with the need to save and restore registers if those reg- isters are needed by instructions in the input routine. Lines 01 through 03 save R1, R2, and R3, lines 1D through 1F restore R1, R2, and R3, and lines 22 through 24 set aside memory locations for those register values.

This leaves lines 05 through 08, 0A through 11, 1A through 1C, 29 and 2A. These lines serve to alert the person sitting at the keyboard that it is time to type a character.

Lines 05 through 08 write the ASCII code x0A to the monitor. This is the ASCII code for a *new line*. Most ASCII codes correspond to characters that are visible on the screen. A few, like x0A, are control characters. They cause an action to occur. Specifically, the ASCII code x0A causes the cursor to move to the far left of the next line on the screen. Thus, the name *Newline*. Before attempting to write x0A, however, as is always the case, DSR[15] is tested (line 6) to see if DDR can accept a character. If DSR[15] is clear, the monitor is busy, and the loop (lines 06 and 07) is repeated. When DSR[15] is 1, the conditional branch (line 7) is not taken, and (line 8) x0A is written to DDR for outputting.

Lines 0A through 11 cause the prompt Input a character> to be written to the screen. The prompt is specified by the .STRINGZ pseudo-op on line 2A and is stored in 19 memory locations—18 ASCII codes, one per memory location, corresponding to the 18 characters in the prompt, and the terminating sentinel x0000.

Line 0C iteratively tests to see if the end of the string has been reached (by detecting x0000), and if not, once DDR is free, line 0F writes the next character in the input prompt into DDR. When x0000 is detected, the entire input prompt has been written to the screen, and the program branches to the code that handles the actual keyboard input (starting at line 13).

After the person at the keyboard types a character and it has been echoed (lines 13 to 19), the program writes one more new line (lines 1A through 1C) before branching to its NEXT TASK.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |  | BRnzp | NEXT\_TASK | ; | 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>''

**Figure !.7 The more sophisticated input routine.**

##### !.#.$ Implementation of Memory-Mapped I/O, Revisited

We showed in Figures 9.4 and 9.6 partial implementations of the data path to handle (separately) memory-mapped input and memory-mapped output. We have also learned that in order to support interrupt-driven I/O, the two status registers must be writeable as well as readable.



GateMDR

MDR

LD.MDR MA R LD.MAR

MIO.EN

R.W

MI O.EN

INPUT

KBDR

OUTPUT

R

MEMORY

MEM.EN

ADDR.CTL. LOGIC

2

KBSR

DSR

INMUX

DDR

**Figure !.8 Relevant data path implementation of memory-mapped I/O.**

Figure 9.8 (also shown as Figure C.3 of Appendix C) shows the data path necessary to support the full range of features we have discussed for the I/O device registers. The Address Control Logic Block controls the input or output operation. Note that there are three inputs to this block. MIO.EN indicates whether a data movement from/to memory or I/O is to take place this clock cycle. MAR contains the address of the memory location or the memory-mapped address of an I/O device register. R.W indicates whether a load or a store is to take place. Depending on the values of these three inputs, the address control logic does nothing (MIO.EN = 0), or it provides the control signals to direct the transfer of data between the MDR and the memory or between the MDR and one of the I/O registers.

If R.W indicates a load, the transfer is from memory or I/O device to the MDR. The Address Control Logic Block provides the select lines to INMUX to source the appropriate I/O device register or memory (depending on MAR) and also enables the memory if MAR contains the address of a memory location.

If R.W indicates a store, the contents of the MDR is written either to memory or to one of the device registers. The address control logic either enables a write to memory or asserts the load enable line of the device register specified by the contents of the MAR.

## !.# Operating System Service Routines (LC-# Trap Routines)

##### !.#.$ Introduction

Recall Figure 9.7 of the previous section. In order for the program to successfully obtain input from the keyboard, it was necessary for the programmer to know several things:

1. The hardware data registers for both the monitor and the keyboard: the monitor so a prompt could be displayed, and the keyboard so the program would know where to get the input character.
2. The hardware status registers for both the monitor and the keyboard: the monitor so the program would know when it was OK to display the next character in the input prompt, and the keyboard so the program would know when someone had struck a key.
3. The asynchronous nature of keyboard input relative to the executing program.

This is beyond the knowledge of most application programmers. In fact, in the real world, if application programmers (or user programmers, as they are some- times called) had to understand I/O at this level, there would be much less I/O and far fewer programmers in the business.

There is another problem with allowing user programs to perform I/O activity by directly accessing KBDR and KBSR. I/O activity involves the use of device registers that are shared by many programs. This means that if a user programmer were allowed to access the hardware registers, and he/she messed up, it could create havoc for other user programs. Thus, in general it is ill-advised to give user programmers access to these registers. That is why the addresses of hardware registers are part of the privileged memory address space and accessible only to programs that have supervisor privilege.

The simpler solution, as well as the safer solution to the problem of user programs requiring I/O, involves the TRAP instruction and the operating system, which of course has supervisor privilege.

We were first introduced to the TRAP instruction in Chapter 4 as a way to get the operating system to halt the computer. In Chapter 5 we saw that a user program could use the TRAP instruction to get the operating system to do I/O tasks for it (the user program). In fact a great benefit of the TRAP instruction, which we have already pointed out, is that it allows the user programmer to not have to know the gory details of I/O discussed earlier in this chapter. In addition, it protects user programs from the consequences of other inept user programmers.

Figure 9.9 shows a user program that, upon reaching location x4000, needs an I/O task performed. The user program uses the TRAP instruction to request the

User Program

Operating system service routine for handling the I/O request

|  |
| --- |
|  |
| TRAP |
|  |

x4000

**Figure !.! Invoking an OS service routine using the TRAP instruction.**

operating system to perform the task on behalf of the user program. The operating system takes control of the computer, handles the request specified by the TRAP instruction, and then returns control back to the user program at location x4001. As we said at the start of this chapter, we usually refer to the request made by the user program as a *system call* or a *service call*.

##### !.3.# The Trap Mechanism

The trap mechanism involves several elements:

1. **A**op**s**e**e**ra**t** t**o**in**f**g**se**s**r**y**v**s**i**t**c**e**e**m**r**.**o**T**u**h**t**e**in**se**es** executed on behalf of user programs by the arbitrary addresses in sy to

256 service routines canstem space. The LC-3 was designed so that up the

are part of the operating system and start at

be specified. Table A.2 in Appendix A contains list of operating system service routines.

LC-3’s current complete

1. **A**is s**t**t**a**o**b**r**l**e**e**d**o**i**f**n**t**m**he**em**st**o**a**r**r**y**ti**l**n**o**g**ca**a**t**d**io**d**n**r**s**e**x**ss**0**e**0**s** of these 256 service routines. This table

various names by various compa 00 to x00FF. The table is referred to by

nies. One company calls this table the company calls it the Trap Vector Table. tor Table of the LC-3, with specific start he starting addresses are the one for the (memory location x0420), which is stor one for the keyboard input service routine

System Control Block. Another

Figure 9.10 shows the Trap Vec ing

addresses highlighted. Among t

character output service routine ed

in memory location x0021, the

(location x04A0), stored in

service routine (location location x0023, and the one for the machine halt

x0520), stored in location x0025.

x0000

|  |
| --- |
|  |
| x03E0 |
| x0420 |
| x0460 |
| x04A0 |
| x04E0 |
| x0520 |
|  |

x0020 x0021 x0022 x0023 x0024 x0025



x00FF

**Figure !.$0 The Trap Vector Table.**

1. **The TRAP instruction.** When a user program wishes to have the operating system execute a specific service routine on behalf of the user program, and then return control to the user program, the user program uses the TRAP instruction (as we have been doing since Chapter 4).
2. **A linkage** back to the user program. The service routine must have a mechanism for returning control to the user program.

##### !.$.$ The TRAP Instruction

The TRAP instruction causes the service routine to execute by (1) changing the PC to the starting address of the relevant service routine on the basis of its trap vector, and (2) providing a way to get back to the program that executed the TRAP instruction. The “way back” is referred to as a *linkage*.

As you know, the **TRAP** instruction is made up of two parts: the TRAP

opcode 1111 and the trap vector (bits [7:0]), which identifies the service routine the user program wants the operating system to execute on its behalf. Bits [11:8] must be zero.

In the following example, the trap vector is x23.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |
| --- | --- | --- |
| 1 1 1 1  TRAP | 0 0 0 0 | 0 0 1 0 0 0 1 1  trap vector |

The EXECUTE phase of the TRAP instruction’s instruction cycle does e things:

thre

1. The PSR and PC are both pushed onto the system stack. Since the PC was incremented during the FETCH phase of the TRAP instruction’s instruction cycle, the return linkage is automatically saved in the PC. When control returns to the user program, the PC will automatically be pointing to the instruction following the TRAP instruction.

Note that the program requesting the trap service routine can be running either in Supervisor mode or in User mode. If in User mode, R6, the stack pointer, is pointing to the user stack. Before the PSR and PC can be pushed onto the system stack, the current contents of R6 must be stored in Saved USP, and the contents of Saved SSP loaded into R6.

1. PSR[15] is set to 0, since the service routine is going to require supervisor privilege to execute. PSR[10:8] are left unchanged since the priority of the TRAP routine is the same as the priority of the program that requested it.
2. The 8-bit trap vector is zero-extended to 16 bits to form an address that corresponds to a location in the Trap Vector Table. For the trap vector x23, that address is x0023. Memory location x0023 contains x04A0, the starting address of the TRAP x23 service routine. The PC is loaded with x04A0, completing the instruction cycle.

Since the PC contains x04A0, processing continues at memory address x04A0.

Location x04A0 is the starting address of the operating system service rou- tine to input a character from the keyboard. We say the trap vector “points” to the starting address of the TRAP routine. Thus, TRAP x23 causes the operating system to start executing the keyboard input service routine.

##### !.$.4 The RTI Instruction: To Return Control to the Calling Program

The only thing left to show is a mechanism for returning control to the calling program, once the trap service routine has finished execution.

This is accomplished by the Return from Trap or Interrupt (RTI) instruction:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

RTI

The RTI instruction (opcode = **1000**, with no operands) pops the top two values on the system stack into the PC and PSR. Since the PC contains the address following the address of the TRAP instruction, control returns to the user program at the correct address.

Finally, once the PSR has been popped off the system stack, PSR[15] must be examined to see whether the processor was running in User mode or Super- visor mode when the TRAP instruction was executed. If in User mode, the stack pointers need to be adjusted to reflect that now back in User mode, the relevant stack in use is the user stack. This is done by loading the Saved SSP with the current contents of R6, and loading R6 with the contents of Saved USP.

##### !.$.& A Summary of the Trap Service Routine Process

Figure 9.11 shows the LC-3 using the TRAP instruction and the RTI instruction to implement the example of Figure 9.9. The flow of control goes from (A) within a user program that needs a character input from the keyboard, to (B) the operating system service routine that performs that task on behalf of the user program, back to the user program (C) that presumably uses the information contained in the input character.

As we know, the computer continually executes its instruction cycle (FETCH, DECODE, etc.) on sequentially located instructions until the flow of control is changed by changing the contents of the PC during the EXECUTE phase of the current instruction. In that way, the next FETCH will be at a redirected address.

The TRAP instruction with trap vector x23 in our user program does exactly that. Execution of TRAP x23 causes the PSR and incremented PC to be pushed onto the system stack and the contents of memory location x0023 (which, in this case, contains x04A0) to be loaded into the PC. The dashed line on Figure 9.11 shows the use of the trap vector x23 to obtain the starting address of the trap service routine from the Trap Vector Table.

The next instruction cycle starts with the FETCH of the contents of x04A0, which is the first instruction of the relevant operating system service routine. The trap service routine executes to completion, ending with the RTI instruction,

User program

x0000

|  |
| --- |
| A |
| 1111 00000010 0011 |
| C |

|  |
| --- |
|  |
| 0000 0100 1010 0000 |
|  |

Trap Vector Table

x0023

x00FF

x04A0

1000 000000000000

B

Character input service routine

**Figure !.11 Flow of control from a user program to an OS service routine and back.**

which loads the PC and PSR with the top two elements on the system stack, that is, the PSR and incremented PC that were pushed during execution of the TRAP instruction. Since the PC was incremented prior to being pushed onto the system stack, it contains the address of the instruction following the TRAP instruction in the calling program, and the user program resumes execution by fetching the instruction following the TRAP instruction.

The following program is provided to illustrate the use of the TRAP instruc- tion. It can also be used to amuse the average four-year-old!

**Example !.1** Write a game program to do the following: A person is sitting at a keyboard. Each time the person types a capital letter, the program outputs the lowercase version of that letter. If the person types a 7, the program terminates.

The following LC-3 assembly language program will do the job.

* 1. .ORIG x3000
  2. LD R2,TERM ; Load -7
  3. LD R3,ASCII ; Load ASCII difference 04 AGAIN TRAP x23 ; Request keyboard input 05 ADD R1,R2,R0 ; Test for terminating

1. BRz EXIT ; character
2. ADD R0,R0,R3 ; Change to lowercase
3. TRAP x21 ; Output to the monitor
4. BRnzp AGAIN ; ... and do it again!

0A TERM .FILL xFFC9 ; FFC9 is negative of ASCII 7 0B ASCII .FILL x0020

0C EXIT TRAP x25 ; Halt

0D .END

appears.

an unconditional BR to AGAIN is executed, and another request for keyboard input

the monitor. When control returns to the application program (this time at line 09),

called. This causes the lowercase representation of the same letter to be displayed on

code, storing the result in R0. Then a TRAP to the monitor output service routine is

7, the ASCII uppercase/lowercase difference (x0020) is added to the input ASCII

BRz instructions test for the terminating character 7. If the character typed is not a

line 05), and R0 contains the ASCII code of the character typed. The ADD and

Then TRAP x23 is executed, which invokes the keyboard input service routine.

When the service routine is finished, control returns to the application program (at

x61. The ASCII codes for ***Z*** and ***z*** are x5A and x7A, respectively.

representation. For example, the ASCII code for ***A*** is x41; the ASCII code for ***a*** is

ASCII code for a capital letter and the ASCII code for that same letter’s lowercase

to continue playing. The constant x0020 is the zero-extended difference between the

for 7, is used to test the character typed at the keyboard to see if the four-year-old wants

The program executes as follows: The program first loads constants xFFC9 and

x0020 into R2 and R3. The constant xFFC9, which is the negative of the ASCII code

The correct operation of the program in this example assumes that the per- son sitting at the keyboard only types capital letters and the value 7. What if the person types a $? A better solution to Example 9.1 would be a program that tests the character typed to be sure it really is a capital letter from among the 26 cap- ital letters in the alphabet or the single digit 7, and if it is not, takes corrective action.

*Question:* Augment this program to add the test for bad data. That is, write a program that will type the lowercase representation of any capital letter typed and will terminate if anything other than a capital letter is typed. See Exercise 9.20.

##### !.3.# Trap Routines for Handling I/O

With the constructs just provided, the input routine described in Figure 9.7 can be slightly modified to be the input service routine shown in Figure 9.12. Two changes are needed: (1) We add the appropriate .ORIG and .END pseudo-ops.

.ORIG specifies the starting address of the input service routine—the address found at location x0023 in the Trap Vector Table. And (2) we terminate the input service routine with the RTI instruction rather than the BR NEXT TASK, as is done on line 20 in Figure 9.7. We use RTI because the service routine is invoked by TRAP x23. It is not part of the user program, as was the case in Figure 9.7.

The output routine of Section 9.2.3.2 can be modified in a similar way, as wn in Figure 9.13. The results are input (Figure 9.12) and output (Figure 9.13) vice routines that can be invoked simply and safely by the TRAP instruction th the appropriate trap vector. In the case of input, upon completion of TRAP

sho

ser

wi

x23, R0 contains the ASCII code of the keyboard character typed. In the case of output, the initiating program must load R0 with the ASCII code of the character it wishes displayed on the monitor and then invoke TRAP x21.

01 ; Service Routine for Keyboard Input 02 ;

1. .ORIG x04A0
2. START ST R1,SaveR1 ; Save the values in the registers 05 ST R2,SaveR2 ; that are used so that they

06 ST R3,SaveR3 ; can be restored before RET

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 07 | ; |  |  |  | | |
| 08 |  | LD | R2,Newline |
| 09  0A | L1 | LDI  BRzp | R3,DSR  L1 | ; | Check DDR -- | is it free? |
| 0B  0C  0D | ; | STI  LEA | R2,DDR  R1,Prompt | ;  ; | Move cursor to new clean line  Prompt is starting address | |
| 0E  1F | Loop | LDR | R0,R1,#0 | ;  ; | of prompt string  Get next prompt character | |
| 10 |  | BRz | Input | ; | Check for end of prompt string | |
| 11 | L2 | LDI | R3,DSR |  |  | |
| 12 |  | BRzp | L2 |  |  | |
| 13 |  | STI | R0,DDR | ; | Write next character of | |
| 14 |  |  |  | ; | prompt string | |
| 15 |  | ADD | R1,R1,#1 | ; | Increment prompt pointer | |
| 16 |  | BRnzp | Loop |  |  | |
| 17 | ; |  |  |  |  | |
| 18 | Input | LDI | R3,KBSR | ; | Has a character been typed? | |
| 19  1A  1B  1C  1D | L3 | BRzp LDI LDI  BRzp  STI | Input R0,KBDR R3,DSR L3  R0,DDR | ;  ; | Load it into R0  Echo input character | |
| 1E  1F | ; |  |  | ; | to the monitor | |
| 20 | L4 | LDI | R3,DSR |  |  | |
| 21 |  | BRzp | L4 |  |  | |
| 22 |  | STI | R2,DDR | ; | Move cursor to new clean line | |
| 23 |  | LD | R1,SaveR1 | ; | Service routine done, restore | |
| 24 |  | LD | R2,SaveR2 | ; | original values in registers. | |
| 25 |  | LD | R3,SaveR3 |  |  | |
| 26 |  | RTI |  | ; | Return from Trap | |
| 27 | ; |  |  |  |  | |
| 28 | SaveR1 | .BLKW | 1 |  |  | |
| 29 | SaveR2 | .BLKW | 1 |  |  | |
| 2A | SaveR3 | .BLKW | 1 |  |  | |
| 2B | DSR | .FILL | xFE04 |  |  | |
| 2C | DDR | .FILL | xFE06 |  |  | |
| 2D | KBSR | .FILL | xFE00 |  |  | |
| 2E  2F | KBDR  Newline | .FILL  .FILL | xFE02  x000A | ; | ASCII code for newline | |
| 30 | Prompt | .STRINGZ "Input a | | character>" | | |
| 31 |  | .END | |  | | |

**Figure !.1# Character input service routine.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 | .ORIG | | x0420 | ; System call starting address |
| 02 | ST | | R1, SaveR1 | ; R1 will be used to poll the DSR |
| 03 |  | |  | ; hardware |
| 1. ; Write the character 2. TryWrite LDI R1, DSR ; Get status | | | | |
| 06 |  | BRzp | TryWrite | ; Bit 15 on says display is ready |
| 07 | WriteIt | STI | R0, DDR | ; Write character |
| 08 |  |  |  |  |

09 ; return from trap

0A Return LD R1, SaveR1 ; Restore registers 0B RTI ; Return from trap

0C DSR .FILL xFE04 ; Address of display status register 0D DDR .FILL xFE06 ; Address of display data register 0E SaveR1 .BLKW 1

0F .END

**Figure !.1# Character output service routine.**

##### !.#.$ A Trap Routine for Halting the Computer

Recall from Section 4.5 that the RUN latch is ANDed with the crystal oscillator to produce the clock that controls the operation of the computer. We noted that if that one-bit latch was cleared, the output of the AND gate would be 0, stopping the clock.

Years ago, most ISAs had a HALT instruction for stopping the clock. Given infrequently that instruction is executed, it seems wasteful to devote an opcode to it. In many modern computers, the RUN latch is cleared by a TRAP routine. In the LC-3, the RUN latch is bit [15] of the Master Control Register (MCR), which is memory-mapped to location xFFFE. Figure 9.14 shows the trap

how

service routine for halting the processor, that is, for stopping the clock.

01 .ORIG x0520 ; Where this routine resides

|  |  |  |
| --- | --- | --- |
| 02 | ST | R1, SaveR1 ; R1: a temp for MC register |
| 03 | ST | R0, SaveR0 ; R0 is used as working space |
| 04 |  |  |
| 05 ; print message that machine is halting 06 | | |
| 07 | LD | R0, ASCIINewLine |
| 08 | TRAP | x21 |
| 09 | LEA | R0, Message |
| 0A | TRAP | x22 |
| 0B | LD | R0, ASCIINewLine |
| 0C | TRAP | x21 |
| 0D | ; |  |
| 0E | ; clear bit 15 at xFFFE | to stop the machine |
| 0F | ; |  |
| 10 | LDI | R1, MCR ; Load MC register into R1 |
| 11 | LD | R0, MASK ; R0 = x7FFF |
| 12 | AND | R0, R1, R0 ; Mask to clear the top bit |
| 13 | STI | R0, MCR ; Store R0 into MC register |

**Figure !.14 HALT service routine for the LC-# (Fig. !.14 continued on next page.)**

14 ;

1. ; return from HALT routine.
2. ; (how can this routine return if the machine is halted above?) 17 ;
3. LD R1, SaveR1 ; Restore registers
4. LD R0, SaveR0

1A RTI

1B ;

1C ; Some constants 1D ;

1E ASCIINewLine .FILL x000A

1F SaveR0 .BLKW 1

1. SaveR1 .BLKW 1
2. Message .STRINGZ "Halting the machine."
3. MCR .FILL xFFFE ; Address of MCR
4. MASK .FILL x7FFF ; Mask to clear the top bit
5. .END

**Figure !.1# HALT service routine for the LC-$ (continued Fig. !.1# from previous page.)**

First (lines 02 and 03), registers R1 and R0 are saved. R1 and R0 are saved because they are needed by the service routine. Then (lines 07 through 0C), the banner *Halting the machine* is displayed on the monitor. Finally (lines 10 through 13), the RUN latch (MCR[15]) is cleared by ANDing the MCR with 0111111111111111. That is, MCR[14:0] remains unchanged, but MCR[15] is cleared. *Question*: What instruction (or trap service routine) can be used to start the clock? *Hint:* This is a trick question! :-)

##### !.$.8 The Trap Routine for Character Input (One Last Time)

Let’s look again at the keyboard input service routine of Figure 9.12. In particular, let’s look at the three-line sequence that occurs at symbolic addresses L1, L2, L3, and L4:

|  |  |  |
| --- | --- | --- |
| LABEL | LDI | R3,DSR |
|  | BRzp | LABEL |
|  | STI | Reg,DDR |

Can the JSR/RET mechanism enable us to replace these four occurrences of the same sequence with a single subroutine? *Answer:* Yes, **almost.**

Figure 9.15, our “improved” keyboard input service routine, contains

JSR WriteChar

at lines 04, 0A, 10, and 13, and the four-instruction subroutine

WriteChar LDI R3,DSR

BRzp WriteChar

STI R2,DDR RET

at lines 1A through 1D. Note the RET instruction (a.k.a. JMP R7) that is needed to terminate the subroutine.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 |  | .ORIG | x04A0 |  |
| 02 | START | JSR | SaveReg |
| 03 |  | LD | R2,Newline |
| 04 |  | JSR | WriteChar |
| 05 |  | LEA | R1,PROMPT |
| 06 | ; |  |  |
| 07 | ; |  |  |
| 08 | Loop | LDR | R2,R1,#0 | ; Get next prompt char |
| 09 |  | BRz | Input |  |
| 0A |  | JSR | WriteChar |  |
| 0B  0C  0D | ; | ADD  BRnzp | R1,R1,#1  Loop |  |
| 0E | Input | JSR | ReadChar |  |
| 0F |  | ADD | R2,R0,#0 | ; Move char to R2 for writing |
| 10 |  | JSR | WriteChar | ; Echo to monitor |
| 11 | ; |  |  |  |
| 12 |  | LD | R2, Newline |  |
| 13 |  | JSR | WriteChar |  |
| 14 |  | JSR | RestoreReg |  |
| 15 |  | RTI |  | ; RTI terminates the trap routine |
| 16 | ; |  |  |  |
| 17 | Newline | .FILL | x000A |  |
| 18 | PROMPT | .STRINGZ "Input a | | character>" |
| 19 | ; |  | |  |
| 1A  1B  1C  1D  1E | WriteChar  DSR | LDI  BRzp STI RET  .FILL | R3,DSR  WriteChar R2,DDR  ; JMP R7 terminates subroutine  xFE04 | |
| 1F  20 | DDR  ; | .FILL | xFE06 | |
| 21 | ReadChar | LDI | R3,KBSR | |
| 22 |  | BRzp | ReadChar | |
| 23 |  | LDI | R0,KBDR | |
| 24 |  | RET |  | |
| 25 | KBSR | .FILL | xFE00 | |
| 26 | KBDR | .FILL | xFE02 | |
| 27 | ; |  |  | |
| 28 | SaveReg | ST | R1,SaveR1 | |
| 29 |  | ST | R2,SaveR2 | |
| 2A |  | ST | R3,SaveR3 | |
| 2B |  | ST | R4,SaveR4 | |
| 2C |  | ST | R5,SaveR5 | |
| 2D |  | ST | R6,SaveR6 | |
| 2E  2F | ; | RET |  | |
| 30 | RestoreReg | LD | R1,SaveR1 | |
| 31 |  | LD | R2,SaveR2 | |
| 32 |  | LD | R3,SaveR3 | |
| 33 |  | LD | R4,SaveR4 | |
| 34 |  | LD | R5,SaveR5 | |
| 35 |  | LD | R6,SaveR6 | |
| 36 |  | RET |  | |
| 37 | SaveR1 | .FILL | x0000 | |
| 38 | SaveR2 | .FILL | x0000 | |
| 39 | SaveR3 | .FILL | x0000 | |
| 3A | SaveR4 | .FILL | x0000 | |
| 3B | SaveR5 | .FILL | x0000 | |
| 3C  3D | SaveR6 | .FILL  .END | x0000 | |

**Figure !.1& The LC-$ trap service routine for character input (our final answer!).**

Note the hedging: *almost*. In the original sequences starting at L2 and L3, the STI instruction forwards the contents of R0 (not R2) to the DDR. We can fix that easily enough, as follows: In line 08 of Figure 9.15, we use

LDR R2,R1,#0

instead of

LDR R0,R1,#0

This causes each character in the prompt to be loaded into R2. The subroutine Writechar forwards each character from R2 to the DDR.

In line 0F of Figure 9.15, we insert the instruction

ADD R2,R0,#0

in order to move the keyboard input (which is in R0) into R2. The subroutine Writechar forwards it from R2 to the DDR. Note that R0 still contains the key- board input. Furthermore, since no subsequent instruction in the service routine loads R0, R0 still contains the keyboard input after control returns to the user program.

In line 12 of Figure 9.15, we insert the instruction

LD R2,Newline

in order to move the “newline” character into R2. The subroutine Writechar forwards it from R2 to the DDR.

Figure 9.15 is the actual LC-3 trap service routine provided for keyboard input.

##### !.$.! PUTS: Writing a Character String to the Monitor

Before we leave the example of Figure 9.15, note the code on lines 08 through 0C. This fragment of the service routine is used to write the sequence of characters *Input a character* to the monitor. A sequence of characters is often referred to as a *string of characters* or a *character string*. This fragment is also present in Figure 9.14, with the result that *Halting the machine* is written to the monitor. In fact, it is so often the case that a user program needs to write a string of characters to the monitor that this function is given its own trap service routine in the LC-3 operating system. Thus, if a user program requires a character string to be written to the monitor, it need only provide (in R0) the starting address of the character string, and then invoke TRAP x22. In LC-3 assembly language this TRAP is called *PUTS*.

PUTS (or TRAP x22) causes control to be passed to the operating system, the trap routine shown in Figure 9.16 is executed. Note that PUTS is the code lines 08 through 0C of Figure 9.15, with a few minor adjustments.

and

of

01 ; This service routine writes a NULL-terminated string to the console. 02 ; It services the PUTS service call (TRAP x22).

03 ; Inputs: R0 is a pointer to the string to print.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 04  05 | ; | .ORIG | x0460 |  | |
| 06 |  | ST | R0, SaveR0 |  | ; Save registers that |
| 07 |  | ST | R1, SaveR1 |  | ; are needed by this |
| 08 |  | ST | R3, SaveR3 |  | ; trap service routine |
| 09 | ; |  |  |  |  |
| 0A | ; Loop | through each | character in | the | array |
| 0B | ; |  |  |  |  |
| 0C | Loop | LDR | R1, R0, #0 |  | ; Retrieve the character(s) |
| 0D |  | BRz | Return |  | ; If it is 0, done |
| 0E | L2 | LDI | R3,DSR |  |  |
| 0F |  | BRzp | L2 |  |  |
| 10 |  | STI | R1, DDR |  | ; Write the character |
| 11 |  | ADD | R0, R0, #1 |  | ; Increment pointer |
| 12 |  | BRnzp | Loop |  | ; Do it all over again |
| 13 | ; |  |  |  |  |
| 14 ; Return from the request for service call | | | | | |
| 15 | Return | LD | R3, SaveR3 | | |
| 16 |  | LD | R1, SaveR1 | | |
| 17 |  | LD | R0, SaveR0 | | |
| 18 |  | RTI |  | | |
| 19 | ; |  |  | | |
| 1A ; Register locations  1B DSR .FILL xFE04 | | | | | |
| 1C | DDR | .FILL | xFE06 | | |
| 1D | SaveR0 | .FILL | x0000 | | |
| 1E | SaveR1 | .FILL | x0000 | | |
| 1F | SaveR3 | .FILL | x0000 | | |
| 20 |  | .END |  | | |

**Figure !.1’ The LC-$ PUTS service routine.**

## !.4 Interrupts and Interrupt- Driven I/O

In Section 9.2.1.3, we noted that interaction between the processor and an I/O device can be controlled by the processor (i.e., polling) or it can be controlled by the I/O device (i.e., interrupt driven). In Sections 9.2.2, 9.2.3, and 9.2.4, we have studied several examples of polling. In each case, the processor tested the ready bit of the status register again and again, and when the ready bit was finally 1, the processor branched to the instruction that did the input or output operation.

We are now ready to study the case where the interaction is controlled by the device.

I/O

##### !.4.1 What Is Interrupt-Driven I/O?

The essence of interrupt-driven I/O is the notion that an I/O device that may or may not have anything to do with the program that is running can (1) force the

.

.

.

Program A is executing instruction n Program A is executing instruction n+1 Program A is executing instruction n+2

1: Interrupt signal is detected

1: Program A is put into suspended animation

1: PC is loaded with the starting address of Program B 2: Program B starts satisying I/O device’s needs

2: Program B continues satisfying I/O device’s needs 2: Program B continues satisfying I/O device’s needs 2: Program B finishes satisfying I/O device’s needs 3: Program A is brought back to life

Program A is executing instruction n+3 Program A is executing instruction n+4

.

.

.

**Figure !.1( Instruction execution flow for interrupt-driven I/O.**

running program to stop, (2) have the processor execute a program that carries out the needs of the I/O device, and then (3) have the stopped program resume exe- cution as if nothing had happened. These three stages of the instruction execution flow are shown in Figure 9.17.

As far as Program A is concerned, the work carried out and the results computed are no different from what would have been the case if the interrupt had never happened; that is, as if the instruction execution flow had been the following:

.

.

.

Program A is executing instruction n Program A is executing instruction n+1 Program A is executing instruction n+2 Program A is executing instruction n+3 Program A is executing instruction n+4

.

.

.

##### !.4.# Why Have Interrupt-Driven I/O?

As is undoubtedly clear, polling requires the processor to waste a lot of time spin- ning its wheels, re-executing again and again the LDI and BR instructions until the ready bit is set. With interrupt-driven I/O, none of that testing and branching has to go on. Interrupt-driven I/O allows the processor to spend its time doing

what is hopefully useful work, executing some other program perhaps, until it is notified that some I/O device needs attention.

Suppose we are asked to write a program that takes a sequence of 100 characters typed on a keyboard and processes the information contained in those 100 characters. Assume the characters are typed at the rate of 80 words/minute, which corresponds to one character every 0.125 seconds. Assume the processing of the 100-character sequence takes 12.49999 seconds, and that our program is to perform this process on 1000 consecutive sequences. How long will it take our program to complete the task? (Why did we pick 12.49999? To make the numbers come out nice, of course.) :-)

**Example !.#**

we We could obtain each character input by polling, as in Section 9.2.2. If we did,

would waste a lot of time waiting for the “next” character to be typed. It would

take 100 • 0.125 or 12.5 seconds to get a 100-character sequence.

any be

On the other hand, if we use interrupt-driven I/O, the processor does not waste time re-executing the LDI and BR instructions while waiting for a character to

typed. Rather, the processor can be busy working on the previous 100-character

sequence that was typed, **except** for those very small fractions of time when it is inter- rupted by the I/O device to read the next character typed. Let’s say that to read the next character typed requires executing a ten-instruction program that takes on the aver- age 0.00000001 seconds to execute each instruction. That means 0.0000001 seconds for each character typed, or 0.00001 seconds for the entire 100-character sequence. That is, with interrupt-driven I/O, since the processor is only needed when characters are actually being read, the time required for each 100-character sequence is 0.00001 seconds, instead of 12.50000 seconds. The remaining 12.49999 of every 12.50000 seconds, the processor is available to do useful work. For example, it can process the previous 100-character sequence.

The bottom line: With polling, the time to complete the entire task for each sequence is 24.9999 seconds, 12.5 seconds to obtain the 100 characters + 12.49999 seconds to process them. With interrupt-driven I/O, the time to complete the entire task for each sequence after the first is 12.5 seconds, 0.00001 seconds to obtain the characters + 12.49999 seconds to process them. For 1000 sequences, that is the difference between 7 hours and 3 1 hours.

2

##### !.4.$ Two Parts to the Process

There are two parts to interrupt-driven I/O:

1. the mechanism that enables an I/O device to interrupt the processor, and
2. the mechanism that handles the interrupt request.

##### !.4.4 Part I: Causing the Interrupt to Occur

Several things must be true for an I/O device to actually interrupt the program that is running:

1. The I/O device **must want** service.
2. The device **must have the right** to request the service.
3. The device request **must be more urgent** than what the processor is currently doing.

If all three elements are present, the processor stops executing the program that is running and takes care of the interrupt.

!.4.4.1 The Interrupt Signal from the Device

For an I/O device to generate an interrupt request, the device must want service, and it must have the right to request that service.

***The Device Must Want Service*** We have discussed that already in the study of polling. It is the ready bit of the KBSR or the DSR. That is, if the I/O device is the keyboard, it wants service if someone has typed a character. If the I/O device is the monitor, it wants service (i.e., the next character to output) if the associated electronic circuits have successfully completed the display of the last character. In both cases, the I/O device wants service when the corresponding ready bit is set.

***The Device Must Have the Right to Request That Service*** This is the interrupt enable bit, which can be set or cleared by the processor (usually by the operating system), depending on whether or not the processor wants to give the I/O device the right to request service. In most I/O devices, this interrupt enable (IE) bit is part of the device status register. In the KBSR and DSR shown in Figure 9.18, the IE bit is bit [14]. The **interrupt request signal from the I/O device** is the logical AND of the IE bit and the ready bit, as is also shown in Figure 9.18.

1# 1413

0

KBSR

Interrupt signal to the processor

1# 1413

0

DSR

Interrupt signal to the processor

**Figure !.1) Interrupt enable bits and their use.**

If the interrupt enable bit (bit [14]) is clear, it does not matter whether the ready bit is set; the I/O device will not be able to interrupt the processor because it (the I/O device) has not been given the right to interrupt the processor. In that case, the program will have to poll the I/O device to determine if it is ready.

If bit [14] is set, then interrupt-driven I/O is enabled. In that case, as soon as someone types a key (or as soon as the monitor has finished processing the last character), bit [15] is set. In this case, the device wants service, and it has been given the right to request service. The AND gate is asserted, causing an interrupt request to be generated from the I/O device.

!.4.4.# The Urgency of the Request

The third element in the list of things that must be true for an I/O device to actually interrupt the processor is that the request must be more urgent than the program

that is currently executing. Recall from Section 9.1.1.2 that each program runs at a specified level of urgency called its priority level. To interrupt the running program, the device must have a higher priority than the program that is currently running. Actually, there may be many devices that want to interrupt the processor at a specific time. To succeed, the device must have a higher priority level than all other demands for use of the processor.

Almost all computers have a set of priority levels that programs can run at. As we have already noted, the LC-3 has eight priority levels, PL0 to PL7. The higher the number, the more urgent the program. The PL of a program is usually the same as the PL (i.e., urgency) of the request to run that program. If a program is running at one PL, and a higher-level PL request wants the computer, the lower- priority program suspends processing until the higher-PL program executes and satisfies its more urgent request. For example, a computer’s payroll program may run overnight, and at PL0. It has all night to finish—not terribly urgent. A program that corrects for a nuclear plant current surge may run at PL6. We are perfectly happy to let the payroll wait while the nuclear power correction keeps us from being blown to bits.

For our I/O device to successfully stop the processor and start an interrupt- driven I/O request, the priority of the request must be higher than the priority of the program it wishes to interrupt. For example, we would not normally want to allow a keyboard interrupt from a professor checking e-mail to interrupt the nuclear power correction program.

!.4.4.# The INT Signal

To stop the processor from continuing execution of its currently running program and service an interrupt request, the INT signal must be asserted. Figure 9.19 shows what is required to assert the INT signal. Figure 9.19 shows the status registers of several devices operating at various priority levels (PL). Any device that has bits [14] and [15] both set asserts its interrupt request signal. The interrupt request signals are input to a priority encoder, a combinational logic structure that selects the highest priority request from all those asserted. If the PL of that request is higher than the PL of the currently executing program, the INT signal is asserted.

!.4.4.4 The Test for INT

Finally, the test to enable the processor to stop and handle the interrupt. Recall from Chapter 4 that the instruction cycle continually sequences through the phases of the instruction cycle (FETCH, DECODE, EVALUATE ADDRESS, FETCH OPERAND, EXECUTE, and STORE RESULT). Each instruction changes the state of the computer, and that change is completed at the end of the instruction cycle for that instruction. That is, in the last clock cycle before the computer returns to the FETCH phase for the next instruction, the computer is put in the state caused by the complete execution of the current instruction.

Interrupts can happen at any time. They are asynchronous to the synchronous finite state machine controlling the computer. For example, the interrupt signal could occur when the instruction cycle is in its FETCH OPERAND phase. If we stopped the currently executing program when the instruction cycle was in

PL0 device PL1 device PL7 device

1# 14

1# 14

1# 14

1

1

1

3

3 PL of current program

B

A

? A > B

Priority encoder

INT

**Figure !.1! Generation of the INT signal.**

its FETCH OPERAND phase, we would have to keep track of what part of the current instruction has executed and what part of the current instruction still has work to do. It makes much more sense to ignore interrupt signals except when we are at an instruction boundary; that is, the current instruction has completed, and the next instruction has not yet started. Doing that means we do not have to worry about partially executed instructions, since the state of the computer is the state created by the completion of the current instruction, period!

The additional logic to test for the interrupt signal is to augment the last state of the instruction cycle for each instruction with a test. Instead of **always** going from the last state of one instruction cycle to the first state of the FETCH phase of the next instruction, the next state depends on the INT signal. If INT is not asserted, then it is business as usual, with the control unit returning to the FETCH phase to start processing the next instruction. If INT is asserted, then the next state is the first state of Part II, handling the interrupt request.

##### !.4.& Part II: Handling the Interrupt Request

Handling the interrupt request goes through three stages, as shown in Figure 9.17:

* 1. Initiate the interrupt (three lines numbered 1 in Figure 9.17).
  2. Service the interrupt (four lines numbered 2 in Figure 9.17).
  3. Return from the interrupt (one line numbered 3 in Figure 9.17). We will discuss each.

!.4.&.1 Initiate the Interrupt

Since the INT signal was asserted, the processor does not return to the first state of the FETCH phase of the next instruction cycle, but rather begins a sequence of actions to initiate the interrupt. The processor must do two things: (1) save the state of the interrupted program so it can pick up where it left off after the require- ments of the interrupt have been completed, and (2) load the state of the higher priority interrupting program so it can start satisfying its request.

***Save the State of the Interrupted Program*** The state of a program is a snap- shot of the contents of all the program’s resources. It includes the contents of the memory locations that are part of the program and the contents of all the general purpose registers. It also includes the PC and PSR.

Recall from Figure 9.1 in Section 9.1.1.4 that a program’s PSR specifies the privilege level and priority level of that program. PSR[15] indicates whether the program is running in privileged (Supervisor) or unprivileged (User) mode. PSR[10:8] specifies the program’s priority level (PL), from PL0 (lowest) to PL7 (highest). Also, PSR[2:0] is used to store the condition codes. PSR[2] is the N bit, PSR[1] is the Z bit, and PSR[0] is the P bit.

The first step in initiating the interrupt is to save enough of the state of the program that is running so that it can continue where it left off after the I/O device request has been satisfied. That means, in the case of the LC-3, saving the PC and the PSR. The PC must be saved since it knows which instruction should be exe- cuted next when the interrupted program resumes execution. The condition codes (the N, Z, and P flags) must be saved since they may be needed by a subsequent conditional branch instruction after the program resumes execution. The priority level of the interrupted program must be saved because it specifies the urgency of the interrupted program with respect to all other programs. When the interrupted program resumes execution, it is important to know what priority level programs can interrupt it and which ones cannot. Finally, the privilege level of the program must be saved since it specifies what processor resources the interrupted program can and cannot access.

Although many computers save the contents of the general purpose registers, we will not since we will assume that the service routine will always save the contents of any general purpose register that it needs before using it, and then restore it before returning to the interrupted program. The only state information the LC-3 saves are the PC and PSR.

The LC-3 saves this state information on the supervisor stack in the same way the PC and PSR are saved when a TRAP instruction is executed. That is, before the interrupt service routine starts, if the interrupted program is in User mode, the User Stack Pointer (USP) is stored in Saved USP, and R6 is loaded with the Supervisor Stack Pointer (SSP) from Saved SSP. Then the PSR and PC of the interrupted program are pushed onto the supervisor stack, where they remain unmolested while the service routine executes.

***Load the State of the Interrupt Service Routine*** Once the state of the inter- rupted program has been safely saved on the supervisor stack, the second step

is to load the PC and PSR of the interrupt service routine. Interrupt service rou- tines are similar to the trap service routines we have already discussed. They are program fragments stored in system space. They service interrupt requests.

Most processors use the mechanism of *vectored interrupts*. You are famil- iar with this notion from your study of the trap vector contained in the TRAP instruction. In the case of interrupts, the eight-bit vector is provided by the device that is requesting the processor be interrupted. That is, the I/O device trans- mits to the processor an eight-bit interrupt vector along with its interrupt request signal and its priority level. The interrupt vector corresponding to the highest priority interrupt request is the one supplied to the processor. It is designated INTV.

If the interrupt is taken, the processor expands the 8-bit interrupt vector (INTV) to form a 16-bit address, which is an entry into the Interrupt Vector Table. You know that the Trap Vector Table consists of memory locations x0000 to x00FF, each containing the starting address of a trap service routine. The Inter- rupt Vector Table consists of memory locations x0100 to x01FF, each containing the starting address of an interrupt service routine. The processor loads the PC with the contents of the location in the Interrupt Vector Table corresponding to the address formed by expanding the interrupt vector INTV.

For example, the LC-3 keyboard could interrupt the processor every time a key is pressed by someone sitting at the keyboard. The keyboard interrupt vector would indicate the location in the interrupt vector table that contains the starting address of the keyboard interrupt service routine.

The PSR is loaded as follows: Since no instructions in the service routine have yet executed, PSR[2:0] contains no meaningful information. We arbitrarily load it initially with 010. Since the interrupt service routine runs in privileged mode, PSR[15] is set to 0. PSR[10:8] is set to the priority level associated with the interrupt request.

This completes the initiation phase, and the interrupt service routine is ready to execute.

!.4.&.# Service the Interrupt

Since the PC contains the starting address of the interrupt service routine, the service routine will execute, and the requirements of the I/O device will be serviced.

!.4.&.$ Return from the Interrupt

The last instruction in every interrupt service routine is RTI, return from trap or interrupt. When the processor finally accesses the RTI instruction, all the requirements of the I/O device have been taken care of.

the

Like the return from a trap routine discussed in Section 9.3.4, execution of

**RTI** instruction (opcode = 1000) for an interrupt service routine consists

simply of popping the PC and the PSR from the supervisor stack (where they have been resting peacefully) and restoring them to their rightful places in the

processor. The condition codes are now restored to what they were when the program was interrupted, in case they are needed by a subsequent BR instruction in the interrupted program. PSR[15] and PSR[10:8] now reflect the privilege level and priority level of the about-to-be-resumed program. If the privilege level of the interrupted program is unprivileged, the stack pointers must be adjusted, that is, the Supervisor Stack Pointer saved, and the User Stack Pointer loaded into R6. The PC is restored to the address of the instruction that would have been executed next if the program had not been interrupted.

With all these things as they were before the interrupt occurred, the program can resume as if nothing had happened.

##### !.4.’ An Example

We complete the discussion of interrupt-driven I/O with an example.

Suppose program A is executing when I/O device B, having a PL higher n that of A, requests service. During the execution of the service routine for

tha

I/O device B, a still more urgent device C requests service.

Figure 9.20 shows the execution flow that must take place.

x3000

x3006

x3(1(

Program A

|  |
| --- |
|  |
| ADD |
|  |

Service routine for device B

x6300

x6’((

Service routine for device C

x6’(’

x6’1(

RTI

RTI

AND

x631#

**Figure !.#0 Execution flow for interrupt-driven I/O.**

Program A consists of instructions in locations x3000 to x3010 and was in the middle of executing the ADD instruction at x3006 when device B sent its interrupt request signal and accompanying interrupt vector xF1, causing INT to be asserted.

Note that the interrupt service routine for device B is stored in locations x6200 to x6210; x6210 contains the RTI instruction. Note that the service routine

for B was in the middle of executing the AND instruction at x6202 when device C sent its interrupt request signal and accompanying interrupt vector xF2. Since the request associated with device C is of a higher priority than that of device B, INT is again asserted.

Note that the interrupt service routine for device C is stored in locations x6300 to x6315; x6315 contains the RTI instruction.

Let us examine the order of execution by the processor. Figure 9.21 shows several snapshots of the contents of the supervisor stack and the PC during the execution of this example.

The processor executes as follows: Figure 9.21a shows the supervisor stack and the PC before program A fetches the instruction at x3006. Note that the stack pointer is shown as Saved SSP, not R6. Since the interrupt has not yet occurred, R6 is pointing to the current contents of the user stack, which are not shown! The INT signal (caused by an interrupt from device B) is detected at the end of execution of the instruction in x3006. Since the state of program A must be

R6

|  |  |
| --- | --- |
|  |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |  |
| --- | --- |
|  |  |
|  |
|  |
|  |
| x3007 |
|  |
| PSR of program A |
|  |

|  |  |
| --- | --- |
|  |  |
|  |
| x6203 |
|  |
| PSR for device B |
| x3007 |
| PSR of program A |
|  |

R6

Saved. SSP

PC PC PC

x6300

x6200

x3006

* + 1. (b) (c)

|  |  |
| --- | --- |
|  |  |
|  |
| x6203 |
| PSR for device B |
| x3007 |
|  |
| PSR of program A |
|  |

|  |  |
| --- | --- |
|  |  |
|  |
| x6203 |
| PSR for device B |
| x3007 |
| PSR of program A |
|  |
|  |

R6

Saved.SSP

PC PC

x3007

x6203

(d) (e)

**Figure !.#1 Snapshots of the contents of the supervisor stack and the PC during interrupt-driven I/O.**

saved on the supervisor stack, the first step is to start using the supervisor stack. This is done by saving R6 in the Saved.UPC register and loading R6 with the contents of the Saved SSP register. The PSR of program A, which includes the condition codes produced by the ADD instruction, is pushed on the supervisor stack. Then the address x3007, the PC for the next instruction to be executed in program A is pushed on the stack. The interrupt vector associated with device B is expanded to 16 bits x01F1, and the contents of x01F1 (x6200) is loaded into the PC. Figure 9.21b shows the stack and PC at this point.

The service routine for device B executes until a higher priority interrupt is detected at the end of execution of the instruction at x6202. The PSR of the service routine for B, which includes the condition codes produced by the AND instruction at x6202, and the address x6203 are pushed on the stack. The interrupt vector associated with device C is expanded to 16 bits (x01F2), and the contents of x01F2 (x6300) is loaded into the PC. Figure 9.21c shows the supervisor stack and PC at this point.

Assume the interrupt service routine for device C executes to completion, fin- ishing with the RTI instruction in x6315. The supervisor stack is popped twice, restoring the PC to x6203 and the PSR of the service routine for device B, includ- ing the condition codes produced by the AND instruction in x6202. Figure 9.21d shows the stack and PC at this point.

The interrupt service routine for device B resumes execution at x6203 and runs to completion, finishing with the RTI instruction in x6210. The supervisor stack is popped twice, restoring the PC to x3007 and the PSR of program A, including the condition codes produced by the ADD instruction in x3006. Finally, since program A is in User mode, the contents of R6 is stored in Saved SSP and R6 is loaded with the contents of Saved USP. Figure 9.21e shows the supervisor stack and PC at this point.

Program A resumes execution with the instruction at x3007.

##### !.4.( Not Just I/O Devices

We have discussed the processing of interrupts in the context of I/O devices that have higher priority than the program that is running and therefore can stop that program to enable its interrupt service routine to execute.

We must point out that not all interrupts deal with I/O devices. Any event that a higher priority and is external to the program that is running can interrupt computer. It does so by supplying its INT signal its INTV vector, and its pri- ity level. If it is the highest priority event that wishes to interrupt the computer,

has

the ,

or

it

does so in the same way that I/O devices do as described above.

There are many examples of such events that have nothing to do with I/O vices. For example, a *timer interrupt* interrupts the program that is running in order to note the passage of a unit of time. The *machine check* interrupt calls atten- tion to the fact that some part of the computer system is not functioning properly. The *power failure* interrupt notifies the computer that, for example, someone has yanked the power cord out of its receptacle. Unfortunately, we will have to put

de

off dealing with all of these until later in your coursework.

## !.& Polling Revisited, Now That We Know About Interrupts

##### !.&.1 The Problem

Recall our discussion of polling: We continually test the ready bit in the relevant status register, and if it is not set, we branch back to again test the ready bit. For example, suppose we are writing a character string to the monitor, and we are using polling to determine when the monitor has successfully written the current character so we can dispatch the next character. We take it for granted that the three-instruction sequence LDI (to load the ready bit of the DSR), BRzp (to test it and fall through if the device is ready), and STI (to store the next character in the DDR) acts as an atomic unit. But what if we had interrupts enabled at the same time? That is, if an interrupt occurred **within** that LDI, BRzp, STI sequence (say, just before the STI instruction), it could easily be the case that the LDI instruction indicated the DDR was ready, the BRzp instruction did not branch back, but by the time the interrupt service routine completed so the STI could write to the DDR, the DDR may no longer be ready. The computer would execute the STI, but the write would not happen.

A simple, but somewhat contrived example :-), will illustrate the problem. Suppose you are executing a “for” loop ten times, where each time the loop body prints to the monitor a particular character. Polling is used to determine that the monitor is ready before writing the next character to the DDR. Since the loop body executes ten times, this should result in the character being printed on the monitor ten times. Suppose you also have keyboard interrupts enabled, and the keyboard service routine echoes the character typed.

Suppose the loop body executes as follows: LDI loads the ready bit, BRzp falls through since the monitor is ready, and STI stores the character in DDR. In the middle of this sequence, before the STI can execute, someone types a key. The keyboard interrupt occurs, the character typed is echoed, i.e., written to the DDR, and the keyboard interrupt service routine completes.

The interrupted loop body then takes over and “knows” the monitor is ready, so it executes the STI. except the monitor is not ready because it has not com-

pleted the write of the keyboard service routine! The STI of the loop body writes, but since DDR is not ready, the write does not occur. The final result: Only nine characters get written, not ten.

The problem becomes more serious if the string written is in code, and the missing write prevents the code from being deciphered.

A simple way to handle this would be to disable all interrupts while polling was going on. But consider the consequences. Suppose the polling was required for a long time. If we disable interrupts while polling is going on, interrupts would be disabled for that very long time, unacceptable in an environment where one is concerned about the time between a higher priority interrupt occurring and the interrupt getting service.

##### !.&.# The Solution

**9.5** Polling Revisited, Now That We Know About Interrupts **351**

A better solution is shown in Figure 9.22.

The sequence we want to make noninterruptable is shown on lines 0F to 11. We accomplish this by first loading R1 with the PSR in line 09 and R2 with the PSR having interrupts disabled in line 0A. PSR[14] is the interrupt enable bit for all interrupts associated with this program. Note that PSR is memory mapped to xFFFC. We enable interrupts by storing R1 in PSR (line 0D), followed imme- diately by disabling interrupts by storing R2 in PSR (line 0E). With interrupts disabled, we execute the three-instruction sequence LDI, BRzp, and LDI (lines 0F, 10, and 11) if the status register indicates that the device is ready. If the device is not ready, BRzp (line 10) takes the computer back to line 0D where interrupts are again enabled.

01 .ORIG x0420

02 ADD R6,R6,#-1

03 STR R1,R6,#0

04 ADD R6,R6,#-1

05 STR R2,R6,#0

06 ADD R6,R6,#-1

07 STR R3,R6,#0 ; Save R1,R2,R3 on the stack 08 ;

09 LDI R1, PSR

0A LD R2,INTMASK

0B AND R2,R1,R2 ; R1=original PSR, R2=PSR with interrupts disabled 0C

0D POLL STI R1,PSR ; enable interrupts (if they were enabled to begin) 0E STI R2,PSR ; disable interrupts

0F LDI R3,DSR

1. BRzp POLL ; Poll the DSR
2. STI R0,DDR ; Store the character into the DDR
3. STI R1,PSR ; Restore original PSR 13

14 LDR R3,R6,#0

15 ADD R6,R6,#1

16 LDR R2,R6,#0

17 ADD R6,R6,#1

18 LDR R1,R6,#0

1. ADD R6,R6,#1 ; Restore R3,R2,and R1 from the stack 1A

1B RTI

1C

1D INTMASK .FILL xBFFF

1E PSR .FILL xFFFC

1F DSR .FILL xFE04

1. DDR .FILL xFE06 21

22 .END

**Figure !.## Polling AND allowing interrupts.**

In this way, interrupts are disabled again and again, but each time only long enough to execute the three-instruction sequence LDI, BRzp, STI (in lines 0F, 10, 0D), after which interrupts are enabled again. The result: An interrupt would have to wait for the three-instruction sequence LDI, BRzp, STI to execute, rather than for the entire polling process to complete.

**Exercises**

**!.1** *a.* What is a device register?

1. What is a device data register?
2. What is a device status register?

**!.#** Why is a ready bit not needed if synchronous I/O is used?

**!.$** In Section 9.2.1.3, the statement is made that a typist would have trouble supplying keyboard input to a 300-MHz processor at the maximum rate (one character every 33 nanoseconds) that the processor can accept it.

Assume an average word (including spaces between words) consists of six characters. How many words/minute would the typist have to type in order to exceed the processor’s ability to handle the input?

**!.4** Are the following interactions usually synchronous or asynchronous?

1. Between a remote control and a television set
2. Between the mail carrier and you, via a mailbox
3. Between a mouse and your PC

Under what conditions would each of them be synchronous? Under what conditions would each of them be asynchronous?

**!.&** What is the purpose of bit [15] in the KBSR?

**!.’** What problem could occur if a program does not check the ready bit of the KBSR before reading the KBDR?

**!.(** Which of the following combinations describe the system described in Section 9.2.2.2?

1. Memory mapped and interrupt driven
2. Memory mapped and polling
3. Special opcode for I/O and interrupt driven
4. Special opcode for I/O and polling

**!.)** Write a program that checks the initial value in memory location x4000 to see if it is a valid ASCII code, and if it is a valid ASCII code, prints the character. If the value in x4000 is not a valid ASCII code, the program prints nothing.

**!.!** What problem is likely to occur if the keyboard hardware does not check the KBSR before writing to the KBDR?

**!.10** What problem could occur if the display hardware does not check the DSR before writing to the DDR?

**!.11** Which is more efficient, interrupt-driven I/O or polling? Explain.

**!.1#** Adam H. decided to design a variant of the LC-3 that did not need a keyboard status register. Instead, he created a readable/writable keyboard data and status register (KBDSR), which contains the same data as the KBDR. With the KBDSR, a program requiring keyboard input would wait until a nonzero value appeared in the KBDSR. The nonzero value would be the ASCII value of the last key press. Then the program would write a zero into the KBDSR, indicating that it had read the key press.

Modify the basic input service of Section 8.2.2 to implement Adam’s scheme.

**!.1$** Some computer engineering students decided to revise the LC-3 for their senior project. In designing the LC-4, they decided to conserve on device registers by combining the KBSR and the DSR into one status register: the IOSR (the input/output status register). IOSR[15] is the keyboard device ready bit and IOSR[14] is the display device ready bit. What are the implications for programs wishing to do I/O? Is this a poor design decision?

**!.14** An LC-3 Load instruction specifies the address xFE02. How do we know whether to load from the KBDR or from memory location xFE02?

**!.1&** Name some of the advantages of doing I/O through a TRAP routine instead of writing the routine yourself each time you would like your program to perform I/O.

**!.1’** *a.* How many trap service routines can be implemented in the LC-3?

Why?

1. Why must a RET instruction be used to return from a TRAP routine? Why won’t a BR (Unconditional Branch) instruction work instead?
2. How many accesses to memory are made during the processing of a TRAP instruction? Assume the TRAP is already in the IR.

**!.1(** Refer to Figure 9.14, the HALT service routine.

1. What starts the clock after the machine is HALTed? *Hint:* How can the HALT service routine return after bit [15] of the Master Control Register is cleared?
2. Which instruction actually halts the machine?
3. What is the first instruction executed when the machine is started again?
4. Where will the RET of the HALT routine return to?

**!.1)** Consider the following LC-3 assembly language program:

|  |  |  |  |
| --- | --- | --- | --- |
| L1 | .ORIG LEA | x3000 R1, L1 |  |
|  | AND | R2, R2, | x0 |
|  | ADD | R2, R2, | x2 |
|  | LD | R3, P1 |  |
| L2 | LDR | R0, R1, | xC |
|  | OUT |  |  |
|  | ADD | R3, R3, | #-1 |
|  | BRz | GLUE |  |
|  | ADD | R1, R1, | R2 |
|  | BR | L2 |  |
| GLUE | HALT |  |  |
| P1 | .FILL | xB |  |

.STRINGZ "HBoeoakteSmtHaotren!s"

.END

* 1. After this program is assembled and loaded, what binary pattern is stored in memory location x3005?
  2. Which instruction (provide a memory address) is executed after instruction x3005 is executed?
  3. Which instruction (provide a memory address) is executed prior to instruction x3006?
  4. What is the output of this program?

**!.1!** The following LC-3 program is assembled and then executed. There are no assemble time or run-time errors. What is the output of this program? Assume all registers are initialized to 0 before the program executes.

|  |  |  |
| --- | --- | --- |
|  | .ORIG LEA | x3000  R0, LABEL |
| STR | R1, R0, #3 |
| TRAP | x22 |
| TRAP | x25 |
| LABEL | .STRINGZ | "FUNKY" |
| LABEL2 | .STRINGZ | "HELLO WORLD" |
|  | .END |  |

**!.#0** The correct operation of the program in Example 9.1 assumes that the person sitting at the keyboard only types capital letters and the value 7. What if the person types a $? A better program would be one that tests the character typed to be sure it really is a capital letter from among the 26 capital letters in the alphabet, and if it is not, takes corrective action. Your job: Augment the program of Example 9.1 to add a test for bad data. That is, write a program that will type the lowercase representation of any capital letter typed and will terminate if anything other than a capital letter is typed.

**!.#1** Assume that an integer greater than 2 and less than 32,768 is deposited in memory location A by another module before the program below is executed.

|  |  |  |
| --- | --- | --- |
|  | .ORIG AND LD NOT | x3000  R4, R4, #0 R0, A  R5, R0 |
|  | ADD | R5, R5, #2 |
|  | ADD | R1, R4, #2 |
|  | ; |  |
| REMOD | JSR | MOD |
|  | BRz | STORE0 |
|  | ; |  |
|  | ADD | R7, R1, R5 |
|  | BRz | STORE1 |
|  | ADD | R1, R1, #1 |
|  | BR | REMOD |
|  | ; |  |
| STORE1 | ADD | R4, R4, #1 |
| STORE0 | ST | R4, RESULT |
|  | TRAP | x25 |
|  | ; |  |
| MOD | ADD | R2, R0, #0 |
|  | NOT | R3, R1 |
|  | ADD | R3, R3, #1 |
| DEC | ADD | R2, R2, R3 |
|  | BRp | DEC |
|  | RET |  |
|  | ; |  |
| A | .BLKW | 1 |
| RESULT | .BLKW | 1 |
|  | .END |  |

In 20 words or fewer, what does the above program do?

**!.##** Recall the machine busy example. Suppose the bit pattern indicating which machines are busy and which are free is stored in memory location x4001. Write subroutines that do the following:

1. Check if no machines are busy, and return 1 if none are busy.
2. Check if all machines are busy, and return 1 if all are busy.
3. Check how many machines are busy, and return the number of busy machines.
4. Check how many machines are free, and return the number of free machines.
5. Check if a certain machine number, passed as an argument in R5, is busy, and return 1 if that machine is busy.
6. Return the number of a machine that is not busy.

**!.#$** The starting address of the trap routine is stored at the address specified in the TRAP instruction. Why isn’t the first instruction of the trap routine stored at that address instead? Assume each trap service routine requires at most 16 instructions. Modify the semantics of the LC-3 TRAP

instruction so that the trap vector provides the starting address of the service routine.

**!.#4** Following is part of a program that was fed to the LC-3 assembler. The program is supposed to read a series of input lines from the console into a buffer, search for a particular character, and output the number of times that character occurs in the text. The input text is terminated by an EOT and is guaranteed to be no more than 1000 characters in length. After the text has been input, the program reads the character to count.

The subroutine labeled COUNT that actually does the counting was written by another person and is located at address x3500. When called, the subroutine expects the address of the buffer to be in R5 and the address of the character to count to be in R6. The buffer should have a NULL to mark the end of the text. It returns the count in R6.

The OUTPUT subroutine that converts the binary count to ASCII digits and displays them was also written by another person and is at address x3600. It expects the number to print to be in R6.

Here is the code that reads the input and calls COUNT:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | .ORIG LEA | x3000  R1, BUFFER |  | |
| G\_TEXT | TRAP | x20 | ; | Get input text |
|  | ADD | R2, R0, x-4 |  |  |
|  | BRz | G\_CHAR |  |  |
|  | STR | R0, R1, #0 |  |  |
|  | ADD | R1, R1, #1 |  |  |
|  | BRnzp | G\_TEXT |  |  |
| G\_CHAR | STR | R2, R1, #0 | ; | x0000 terminates buffer |
|  | TRAP | x20 | ; | Get character to count |
|  | ST | R0, S\_CHAR |  |  |
|  | LEA | R5, BUFFER |  |  |
|  | LEA | R6, S\_CHAR |  |  |
|  | LD | R4, CADDR |  |  |
|  | JSRR | R4 | ; | Count character |
|  | LD | R4, OADDR |  |  |
|  | JSRR | R4 | ; | Convert R6 and display |
|  | TRAP | x25 |  |  |
| CADDR | .FILL | x3500 | ; | Address of COUNT |
| OADDR | .FILL | x3600 | ; | Address of OUTPUT |
| BUFFER | .BLKW | 1001 |  |  |
| S\_CHAR | .FILL | x0000 |  |  |
|  | .END |  |  |  |

There is a problem with this code. What is it, and how might it be fixed? (The problem is *not* that the code for COUNT and OUTPUT is missing.)

**!.#&** Consider the following LC-3 assembly language program:

|  |  |  |
| --- | --- | --- |
|  | .ORIG LEA | x3000 R0,DATA |
| AND | R1,R1,#0 |
| ADD | R1,R1,#9 |
| LOOP1 | ADD | R2,R0,#0 |
|  | ADD | R3,R1,#0 |
| LOOP2 | JSR | SUB1 |
|  | ADD | R4,R4,#0 |
|  | BRzp | LABEL |
|  | JSR | SUB2 |
| LABEL | ADD | R2,R2,#1 |
|  | ADD | R3,R3,#-1 |
|  | BRP | LOOP2 |
|  | ADD | R1,R1,#-1 |
|  | BRp | LOOP1 |
|  | HALT |  |
| DATA | .BLKW | 10 x0000 |
| SUB1 | LDR | R5,R2,#0 |
|  | NOT | R5,R5 |
|  | ADD | R5,R5,#1 |
|  | LDR | R6,R2,#1 |
|  | ADD | R4,R5,R6 |
|  | RET |  |
| SUB2 | LDR | R4,R2,#0 |
|  | LDR | R5,R2,#1 |
|  | STR | R4,R2,#1 |
|  | STR | R5,R2,#0 |
|  | RET |  |
|  | .END |  |

Assuming that the memory locations at DATA get filled in before program executes, what is the relationship between the final values TA and the initial values at DATA?

the at

DA

**!.#’** The following program is supposed to print the number 5 on the screen.

It does not work. Why? Answer in no more than ten words, please.

.ORIG x3000

|  |  |  |
| --- | --- | --- |
| A | JSR OUT  BRnzp  AND | A  DONE R0,R0,#0 |
|  | ADD | R0,R0,#5 |
|  | JSR | B |
|  | RET |  |
| DONE | HALT |  |
| ASCII | .FILL | x0030 |
| B | LD | R1,ASCII |
|  | ADD | R0,R0,R1 |
|  | RET |  |
|  | .END |  |

**!.#(** Figure 9.14 shows a service routine to stop the computer by clearing the RUN latch, bit [15] of the Master Control Register. The latch is cleared by the instruction in line 14, and the computer stops. What purpose is served by the instructions on lines 19 through 1C?

**!.#)** Suppose we define a new service routine starting at memory location x4000. This routine reads in a character and echoes it to the screen. Suppose memory location x0072 contains the value x4000. The service routine is shown below.

.ORIG x4000 ST R7, SaveR7 GETC

OUT

LD R7, SaveR7 RET

SaveR7 .FILL x0000

* 1. Identify the instruction that will invoke this routine.
  2. Will this service routine work? Explain.

**!.#!** The two code sequences *a* and *b* are assembled separately. There are two errors that will be caught at assemble time or at link time. Identify the bugs, and describe why the bug will cause an error, and whether it will be detected at assemble time or link time.

1. .ORIG x3500

SQRT ADD R0, R0, #0

; code to perform square

; root function and

; return the result in R0 RET

.END

1. .EXTERNAL SQRT

.ORIG x3000

LD R0,VALUE

JSR SQRT

ST R0,DEST HALT

VALUE .FILL x30000 DEST .FILL x0025

.END

**!.$0** Shown below is a partially constructed program. The program asks the user his/her name and stores the sentence “Hello, name” as a string starting from the memory location indicated by the symbol HELLO. The program then outputs that sentence to the screen. The program assumes that the user has finished entering his/her name when he/she presses the Enter key, whose ASCII code is x0A. The name is restricted to be not more than 25 characters.

Assuming that the user enters Onur followed by a carriage return when prompted to enter his/her name, the output of the program looks exactly like:

Please enter your name: Onur Hello, Onur

Insert instructions at (a)–(d) that will complete the program.

.ORIG x3000 LEA R1,HELLO

AGAIN LDR R2,R1,#0 BRz NEXT

ADD R1,R1,#1

BR AGAIN

NEXT LEA R0,PROMPT

TRAP x22 ; PUTS

(a)

AGAIN2 TRAP x20 ; GETC

TRAP x21 ; OUT ADD R2,R0,R3

BRz CONT

(b)

(c) BR AGAIN2

CONT AND R2,R2,#0

(d) LEA R0, HELLO

TRAP x22 ; PUTS

TRAP x25 ; HALT NEGENTER .FILL xFFF6 ; -x0A

PROMPT .STRINGZ "Please enter your name: " HELLO .STRINGZ "Hello, "

.BLKW #25

.END

**!.$1** The program below, when complete, should print the following to the monitor:

ABCFGH

Insert instructions at (a)–(d) that will complete the program.

.ORIG x3000

LEA R1, TESTOUT BACK\_1 LDR R0, R1, #0

BRz NEXT\_1

TRAP x21

(a) BRnzp BACK\_1

;

NEXT\_1 LEA R1, TESTOUT BACK\_2 LDR R0, R1, #0

BRz NEXT\_2

JSR SUB\_1

ADD R1, R1, #1

BRnzp BACK\_2

;

NEXT\_2 (b)

;

SUB\_1 (c)

K LDI R2, DSR

(d)

STI R0, DDR RET

DSR .FILL xFE04

DDR .FILL xFE06 TESTOUT .STRINGZ "ABC"

.END

**!.$#** A local company has decided to build a real LC-3 computer. In order to make the computer work in a network, four interrupt-driven I/O devices are connected. To request service, a device asserts its interrupt request signal (IRQ). This causes a bit to get set in a special LC-3

memory-mapped interrupt control register called INTCTL, which is mapped to address xFF00. The INTCTL register is shown below. When a device requests service, the INT signal in the LC-3 data path is asserted. The LC-3 interrupt service routine determines which device has requested service and calls the appropriate subroutine for that device. If more than one device asserts its IRQ signal at the same time, only the subroutine for the highest priority device is executed. During execution of the subroutine, the corresponding bit in INTCTL is cleared.

Hard disk

Ethernet card

Printer

CD-ROM

IRQH

IRQE

IRQP

IRQC

INTCTL

INT

The following labels are used to identify the first instruction of each vice subroutine:

de

HARDDISK ETHERNET PRINTER CDROM

For example, if the highest priority device requesting service is the inter, the interrupt service routine will call the printer subroutine with e following instruction:

pr

th

JSR PRINTER

Finish the code in the LC-3 interrupt service routine for the following priority scheme by filling in the spaces labeled (a)–(k). The lower the number, the higher the priority of the device.

1. Hard disk
2. Ethernet card
3. Printer
4. CD-ROM

LDI R1, INTCTL DEV0 LD R2, (a)

AND R2, R2, R1 BRnz DEV1

JSR (b)

(c)

;

DEV1 LD R2, (d)

AND R2, R2, R1 BRnz DEV2

JSR (e)

(f)

;

DEV2 LD R2, (g)

AND R2, R2, R1 BRnz DEV3

JSR (h)

(i)

;

DEV3 JSR (j)

;

END (k)

|  |  |  |
| --- | --- | --- |
| INTCTL | .FILL | xFF00 |
| MASK8 | .FILL | x0008 |
| MASK4 | .FILL | x0004 |
| MASK2 | .FILL | x0002 |
| MASK1 | .FILL | x0001 |

**!.$$** Interrupt-driven I/O:

* 1. What does the following LC-3 program do?

|  |  |  |
| --- | --- | --- |
|  | .ORIG | x3000 |
| LD | R3, A |
| STI | R3, KBSR |
| AGAIN | LD | R0, B |
|  | TRAP | x21 |
|  | BRnzp | AGAIN |
| A | .FILL | x4000 |
| B | .FILL | x0032 |
| KBSR | .FILL | xFE00 |
|  | .END |  |

* 1. If someone strikes a key, the program will be interrupted and the keyboard interrupt service routine will be executed as shown below. What does the keyboard interrupt service routine do?

.ORIG x1000

LDI R0, KBDR

TRAP x21

TRAP x21 RTI

KBDR .FILL xFE02

.END

*Note:* RTI is an instruction that enables the computer to return to executing the program that was interrupted. It will be studied in Chapter 10. The only thing you need to know about it now is that it loads the PC with the address of the instruction that was about to be fetched when the interrupt occurred.

* 1. Finally, suppose the program of part *a* started executing, and someone sitting at the keyboard struck a key. What would you see on the screen?
  2. In part c, how many times is the digit typed shown on the screen? Why is the correct answer: “I cannot say for sure.”

**!.$4** What does the following LC-3 program do?

|  |  |  |
| --- | --- | --- |
|  | .ORIG | x3000 |
| LD | R0,ASCII |
| LD | R1,NEG |
| AGAIN | LDI | R2,DSR |
|  | BRzp | AGAIN |
|  | STI | R0,DDR |
|  | ADD | R0,R0,#1 |
|  | ADD | R2,R0,R1 |
|  | BRnp | AGAIN |
|  | HALT |  |
| ASCII | .FILL | x0041 |
| NEG | .FILL | xFFB6 ; -x004A |
| DSR | .FILL | xFE04 |
| DDR | .FILL | xFE06 |
|  | .END |  |

**!.$&** During the initiation of the interrupt service routine, the N, Z, and P condition codes are saved on the stack. Show by means of a simple example how incorrect results would be generated if the condition codes were not saved.

**!.$’** In the example of Section 9.4.6, what are the contents of locations x01F1 and x01F2? They are part of a larger structure. Provide a name for that structure. (*Hint:* See Table A.3.)

**!.$(** Expand the example of Section 9.4.6 to include an interrupt by a still more urgent device D while the service routine of device C is executing the instruction at x6310. Assume device D’s interrupt vector is xF3.

Assume the interrupt service routine is stored in locations x6400 to

x6412. Show the contents of the stack and PC at each relevant point in the execution flow.

**!.$)** Suppose device D in Exercise 9.37 has a lower priority than device C but a higher priority than device B. Rework Exercise 9.37 with this new wrinkle.

**!.$!** Write an interrupt handler to accept keyboard input as follows: A buffer is allocated to memory locations x4000 through x40FE. The interrupt handler must accept the next character typed and store it in the next “empty” location in the buffer. Memory location x40FF is used as a pointer to the next available empty buffer location. If the buffer is full (i.e., if a character has been stored in location x40FE), the interrupt handler must display on the screen: “Character cannot be accepted; input buffer full.”

**!.40** Consider the interrupt handler of Exercise 9.39. The buffer is modified as follows: The buffer is allocated to memory locations x4000 through x40FC. Location x40FF contains, as before, the address of the next available empty location in the buffer. Location x40FE contains the address of the oldest character in the buffer. Location x40FD contains the number of characters in the buffer. Other programs can remove characters from the buffer. Modify the interrupt handler so that, after x40FC is filled, the next location filled is x4000, assuming the character in x4000 has been previously removed. As before, if the buffer is full, the interrupt handler must display on the screen: “Character cannot be accepted; input buffer full.”

**!.41** Consider the modified interrupt handler of Exercise 9.40, used in conjunction with a program that removes characters from the buffer. Can you think of any problem that might prevent the interrupt handler that is adding characters to the buffer and the program that is removing characters from the buffer from working correctly together?

**!.4#** Suppose the keyboard interrupt vector is x34 and the keyboard interrupt service routine starts at location x1000. What can you infer about the contents of any memory location from the above statement?

**!.4$** Two students wrote interrupt service routines for an assignment. Both service routines did exactly the same work, but the first student accidentally used RET at the end of his routine, while the second student correctly used RTI. There are three errors that arose in the first student’s program due to his mistake. Describe any two of them.

\***!.44** Since ASCII codes consist of eight bits each, we can store two ASCII codes in one word of LC-3 memory. If a user types 2*n* characters on the keyboard, followed by the Enter key, the subroutine PACK on the next page will store the corresponding ASCII codes into *n* sequential memory locations, two per memory location, starting at location A.

You may assume that a user never enters an odd number of characters. Your job: Fill in the blanks in the program.

If a user types the string **Please help!** followed by the Enter key, what does the program do?

.ORIG x7020 PACK ST R7, SAVER7

ST R6, SAVER6 ST R4, SAVER4 ST R3, SAVER3

LEA R6, A ; R6 is the pointer AND R4, R4, #0

ADD R4, R4, #8 ; R4 is our counter AND R3, R3, #0

LEA R0, PROMPT TRAP x22

POLL BRzp POLL

(a)

(b) LD R0, NEG\_LF

ADD R0, R7, R0

(c) ADD R4, R4, #0

BRz NOSHIFT SHIFT ADD R7, R7, R7

ADD R4, R4, #-1 BRp SHIFT

ADD R3, R7, #0

BRnzp POLL NOSHIFT ADD R3, R3, R7

(d) ADD R6, R6, #1

ADD R4, R4, #8

BRnzp POLL DONE LD R7, SAVER7

LD R6, SAVER6 LD R4, SAVER4 LD R3, SAVER3

LEA R0, A ; Returns a pointer to the characters RET

KBSR .FILL xFE00

KBDR .FILL xFE02 NEG\_LF .FILL xFFF6

PROMPT .STINGZ "Please enter a string: " A .BLKW #5

SAVER7 .BLKW #1

SAVER6 .BLKW #1

SAVER4 .BLKW #1

SAVER3 .BLKW #1

.END

\***!.4&** We want to support eight input keyboards instead of one. To do this, we need eight ready bits in KBSR, and eight separate KBDRs. We will use the eight odd-numbered bits in the KBSR as ready bits for the eight keyboards, as shown below. We will set the other eight bits in the KBSR to 0.

KBSR

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

15 13 11 9 7 5 3 1

The eight memory-mapped keyboard data registers and their corresponding ready bits are as follows:

FE04: KBSR

FE06: KBDR1, Ready bit is KBSR[1] FE08: KBDR2, Ready bit is KBSR[3] FE0A: KBDR3, Ready bit is KBSR[5] FE0C: KBDR4, Ready bit is KBSR[7] FE0E: KBDR5, Ready bit is KBSR[9] FE10: KBDR6, Ready bit is KBSR[11] FE12: KBDR7, Ready bit is KBSR[13] FE14: KBDR8, Ready bit is KBSR[15]

We wish to write a program that polls the keyboards and loads the ASCII code typed by the highest priority keyboard into R0. That is, if someone had previously typed a key on keyboard 1, we want to load the ASCII code in KBDR1 into R0. If no key was typed on keyboard 1, but a key had been typed on keyboard 2, we want to load the ASCII code in KBDR2 into R0. and so on. That is, KB1 has higher priority than

KB2, which has higher priority than KB3, which has higher priority than KB4, etc. KB8 has the lowest priority.

The following program will do the job AFTER you fill in the missing instructions. Your job: Fill in the missing instructions.

.ORIG X3000

LD R0, KBDR1 POLL LDI R1, KBSR

BRz POLL

AND R2, R2, #0 ADD R2, R2, #2

AGAIN

BRnp FOUND

ADD R0, R0, #2

(a)

(b)

(c) BRnp AGAIN

HALT

FOUND

HALT

KBSR .FILL xFE04 KBDR1 .FILL xFE06

.END

(d)

\***!.4’** The following program pushes elements onto a stack with JSR PUSH and pops elements off the stack with JSR POP.

.ORIG X3000

LEA R6, STACK\\_BASE

X TRAP x20 ;GETC

TRAP x21 ;OUT

ADD R1, R0, x-0A ;x0A is ASCII code for line feed,

;x-0A is the negative of x0A

BRz Y

JSR PUSH

BRnzp X

Y LEA R2, STACK\\_BASE NOT R2, R2

ADD R2, R2, #1 ADD R3, R2, R6 BRz DONE

JSR POP

TRAP x21 ;OUT

BRnzp Y

DONE TRAP x25 ;HALT

STACK .BLKW 5 STACK\\_BASE .FILL x0FFF

PUSH ADD R6, R6, #-1 STR R0, R6, #0 RET

POP LDR R0, R6, #0 ADD R6, R6, #1 RET

.END

What will appear on the screen if a user, sitting at a keyboard, typed the three keys a, b, c, followed by the Enter key?

What will happen if a user, sitting at a keyboard, typed the eight keys a, b, c, d, e, f, g, h, followed by the Enter key?

**!.4(** We wish to add a new TRAP service routine, which will be called by the instruction TRAP x9A. The new trap routine will wait for someone to type a lowercase letter, then echo on the screen the corresponding capital letter. Assume the user will not type anything except a lowercase letter. The assembly language code for this trap service routine is shown below:

.ORIG x2055

(a) ST R1, SaveR1

ST R0, SaveR0 TRAP x20

LD R1, A

(b) TRAP x21

(c) LD R1, SaveR1

LD R0, SaveR0 JMP R7

SaveR1 .BLKW 1

SaveR0 .BLKW 1

A .FILL (d)

.BLKW ; (e) a missing label

.END

In order for TRAP x9A to call this service routine, what must be contained in the memory location having what address?

Fill in the missing information in the assembly language program, that is, the three missing instructions, the one missing label, and the operand of the .FILL pseudo-op.

**!.4)** A programmer wrote the following program that was assembled and executed. Execution started with PC at x3000.

.ORIG x3000

LEA R0, Message TRAP x01

TRAP x22 ; What is the output here? TRAP x25

Message .STRINGZ "Cat in the hat."

.END

Assume that the Trap Vector Table includes the following entries in addition to the ones we have previously used:

Memory Address Memory Contents x0000 x0100

x0001 x0102

x0002 x0107

x0003 x010A

Assume further that additional trap service routines have been loaded previously in memory as specified below:

.ORIG x0100

LD R7, SaveR7 RET

ST R7, SaveR7 TRAP x02

AND R1, R1, #0 STR R1, R0, #3 RET

AND R1, R1, #0 STR R1, R0, #5 TRAP x00

RET SaveR7 .BLKW #1

.END

What is the result of execution of this program?

\***!.4!** The state machine shown below will produce an output sequence if it receives an input sequence. The initial state is S0.

S0

a

0

1

0

0

1

S2

c

S1

b

1

For example, the input sequence 100 produces the output sequence bac. We have written a program that simulates this state machine. Inputs are requested from the keyboard, and the corresponding outputs are shown on the screen. For example, for the input sequence shown above, the monitor would display

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| INPUT (either OUTPUT: b | 0 | or | 1): | 1 |
| INPUT (either | 0 | or | 1): | 0 |
| OUTPUT: a |  |  |  |  |
| INPUT (either | 0 | or | 1): | 0 |
| OUTPUT: c |  |  |  |  |

Complete the program that simulates the state machine by filling in each blank with **one** missing line of LC-3 assembly language code. You can assume the person at the keyboard can type a 1 or a 0 without error (i.e., you do not have to test for wrong input).

.ORIG x3000

LEA R6, S0

Loop

TRAP x22

(a)

TRAP x20 ; inputs a character TRAP x21

LD R1, NEGASCII

ADD R0, R0, R1

(b) LDR R6, R6,#0

LD R0, NEWLINE TRAP x21

LEA R0, OUTPUT

TRAP x22

(c)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | TRAP |  | x21 |  |
| LD | R0, | NEWLINE |
| TRAP  BRnzp |  | x21 LOOP |
| S0 | .FILL  .FILL  .FILL |  | S2 S1 x0061 |
| S1  S2 | .FILL  .FILL  .FILL  .FILL |  | S0 S2 x0062 | (d) |
|  | .FILL  .FILL |  |  | (e)  (f) |

NEGASCII .FILL xFFD0 ; the value -48 OUTPUT .STRINGZ "OUTPUT:"

INPUT .STRINGZ "INPUT (either 0 or 1):" NEWLINE .FILL x000A

.END

\***!.&0** Up to now, we have only had one output device, the monitor, with xFE04 and xFE06 used to address its two device registers. We now introduce a second output device, a light that requires a single device register, to which we assign the address xFE08. Storing a 1 in xFE08 turns the light on, storing a 0 in xFE08 turns the light off.

An Aggie decided to write a program that would control this light by a keyboard interrupt as follows: Pressing the key 0 would turn the light off. Pressing the key 1 would cause the light to flash on and off repeatedly.

Shown below is the Aggie’s code and his keyboard interrupt service routine.

The User Program:

.ORIG x3000

1. LEA R7, LOOP
2. LOOP LDI R0, ENABLE
3. LD R1, NEG\_OFF
4. ADD R0, R0, R1 ; check if switch is on
5. BRnp BLINK

;

5 AND R0, R0, #0

1. STI R0, LIGHT ; turn light off
2. RET

;

1. BLINK ST R7, SAVE\_R7 ; save linkage
2. LDI R0, LIGHT
   1. ADD R0, R0, #1
   2. AND R0, R0, #1 ; toggle LIGHT between 0 and 1 C STI R0, LIGHT

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| D E |  | JSR DELAY  LD R7, SAVE\_R7 | ; | 1 second delay |
| F |  | RET | ; | <-- Breakpoint here |
|  | ; |  |  |  |
|  | LIGHT | .FILL xFE08 |  |  |
|  | ENABLE | .FILL x4000 |  |  |
|  | NEG\_OFF | .FILL x-30 |  |  |
|  | SAVE\_R7 | .BLKW #1 |  |  |
|  |  | .END |  |  |

The Keyboard Interrupt Routine:

.ORIG x1500

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 |  | ADD R6, R6, #-1 | ; | <-- Breakpoint here |
| 1 |  | STR R0, R6, #0 | ; | save R0 on stack |
| 2 |  | ADD R6, R6, #-1 |  |  |
| 3 |  | STR R7, R6, #0 | ; | save R7 on stack |
|  | ; |  |  |  |
| 4 |  | TRAP x20 |  |  |
| 5 |  | STI R0, ENABLE2 |  |  |
|  | ; |  |  |  |
| 6 |  | RTI | ; | <-- Breakpoint here |
| 7 | ENABLE2 | .FILL x4000 |  |  |
|  |  | .END |  |  |

The DELAY subroutine was inserted in his program in order to separate the turning on and off of the light by one second in order to make the

on-off behavior visible to the naked eye. The DELAY subroutine does not modify any registers.

Unfortunately, per usual, the Aggie made a mistake in his program, and things do not work as he intended. So he decided to debug his program (see the next page).

He set three breakpoints, at x1500, at x1506, and at x300F. He initialized the PC to x3000, the keyboard IE bit to 1, and memory location x0180 to x1500.

Then he hit the Run button, which stopped executing when the PC reached x1500. He hit the Run button three more times, each time the computer stopping when the PC reached a breakpoint. While the program was running, he pressed a key on the keyboard EXACTLY ONCE.

The table below shows the data in various registers and memory locations each time a breakpoint was encountered. *Note:* Assume, when an interrupt is initiated, the PSR is pushed onto the system stack before the PC.

Complete the table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Initial** | **Breakpoint 1** | **Breakpoint 2** | **Breakpoint 3** | **Breakpoint 4** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **PC** | x3000  x1234 | x1500 | x1506  x0030 | x1506 | x300F |
| **R0** | x3000 |  |  |  |  |
| **R6** | x1234 |  |  |  |  |
| **R7** | x0000 |  |  |  |  |
| **M[x2FFC]** | x0000 |  |  |  |  |
| **M[x2FFD]** | x0000 | x300D |  |  |  |
| **M[x2FFE]** | x0000 | x8001 |  |  |  |
| **M[x2FFF]** | x0031 |  |  |  |  |
| **M[x4000]** | x4000 |  |  |  |  |
| **M[xFE00]** |  |  |  |  |  |

\***!.&1** The following user program (priority 0) is assembled and loaded into memory.

.ORIG x8000 LD R0, Z

AGAIN ADD R0, R0, #-1

BRnp AGAIN LD R0, W BRp L1

LD R0, X TRAP x21

BRnzp DONE

L1 LEA R0, Y TRAP x22

DONE HALT

1. .FILL x34
2. .STRINGZ "OOOOPS!"
3. .FILL x100

W .BLKW #1

.END

Before this code executes, two things happen: (a) another program loads a value into W, and (b) a breakpoint is set at the address DONE.

Then the run switch is hit and the program starts executing. Before the computer stops due to the breakpoint, several interrupts occur and their corresponding service routines are executed. Finally, the LC-3 stops due to the breakpoint. We examine the memory shown, and R6, the Supervisor Stack Pointer.

|  |
| --- |
| Memory  x0601 |
| x0601 |
| x0500 |
| x0504 |
| x0204 |
| x0201 |
| x8004 |
| x8002 |
| x8010 |
| x8012 |
|  |

x2FF8 x2FF9 x2FFA x2FFB x2FFC x2FFD x2FFE x2FFF x3000 x3001

R6 x3000

What does the user program write to the monitor? How do you know that?

\***!.&#** Your job in this problem will be to add the missing instructions to a program that detects palindromes. Recall that a palindrome is a string of characters that are identical when read from left to right or from right to left—for example, racecar and 112282211. In this program, we will have no spaces and no capital letters in our input string—just a string of lowercase letters.

The program will make use of both a stack and a queue. The subroutines for accessing the stack and queue are shown below. Recall that elements are PUSHed (added) and POPped (removed) from the stack. Elements are ENQUEUEd (added) to the back of a queue and DEQUEUEd (removed) from the front of the queue.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PUSH | .ORIG x3050 ADD R6, R6, | #-1 | ENQUEUE | .ORIG x3080 ADD R5, R5, | #1 |
|  | STR R0, R6, | #0 |  | STR R0, R5, | #0 |
|  | RET |  |  | RET |  |
| POP | LDR R0, R6, | #0 | DEQUEUE | LDR R0, R4, | #0 |
|  | ADD R6, R6, | #1 |  | ADD R4, R4, | #1 |
|  | RET |  |  | RET |  |
| STACK | .BLKW #20 |  | QUEUE | .BLKW #20 |  |
|  | .END |  |  | .END |  |

The program is carried out in two phases. Phase 1 enables a user to input a character string one keyboard character at a time. The character string is terminated when the user types the Enter key (line feed). In Phase 1, the ASCII code of each character input is pushed onto a stack, and its

negative value is inserted at the back of a queue. Inserting an element at the back of a queue we call enqueuing.

In Phase 2, the characters on the stack and in the queue are examined by removing them one by one from their respective data structures (i.e., stack and queue). If the string is a palindrome, the program stores a 1 in memory location RESULT. If not, the program stores a 0 in memory location RESULT. The PUSH and POP routines for the stack as well as the ENQUEUE and DEQUEUE routines for the queue are shown below. You may assume the user never inputs more than 20 characters.

.ORIG X3000 LEA R4, QUEUE LEA R5, QUEUE

ADD R5, R5, #-1

LEA R6, ENQUEUE ; Initialize SP LD R1, ENTER

AND R3, R3, #0

(a) TRAP x22

PHASE1 TRAP x20

(b) BRz PHASE2

JSR PUSH

(c)

(d) JSR ENQUEUE

ADD R3, R3, #1 BRnzp PHASE1

;

PHASE2 JSR POP

(e) JSR DEQUEUE

ADD R1, R0, R1 BRnp FALSE

(f)

(g) BRnzp PHASE2

;

TRUE AND R0, R0, #0

ADD R0, R0, #1 ST R0, RESULT HALT

FALSE AND R0, R0, #0

ST R0, RESULT HALT

RESULT .BLKW #1 ENTER .FILL x-0A

PROMPT .STRINGZ "Enter an input string: "

.END

\***!.&$** Now that the keyboard interrupt is old stuff for you, it is time to introduce two interrupts for the LC-3: INTA and INTB. The necessary hardware has been added to allow them to happen. INTA has priority 2

and an interrupt vector of x50. INTB has priority 4 and an interrupt vector of x60.

Recall that the priority is specified in bits [10:8] of the PSR. In fact, the full PSR specification is:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

PSR:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pr | 0 | 0 | 0 | 0 | Priority | 0 | 0 | 0 | 0 | 0 | N | Z | P |

where PSR[15] = 0 (Supervisor mode), 1 (User mode).

PSR[14:11] = 0000

PSR[10:8] = priority, 0 (lowest) to 7 (highest). PSR[7:3] = 00000

PSR[2:0] = condition codes for N,Z,P

In this problem, you are given the user program and the two interrupt service routines. The user program starts executing at cycle 1 and runs at priority 0.

User program:

.ORIG x3000 AND R0,R0,#0 ADD R0,R0,#5 LD R1,COUNT NOT R0,R0 ADD R0,R0,#1

AGAIN ADD R2,R0,R1

BRz DONE

ADD R1,R1,#-1 BRnzp AGAIN

DONE TRAP x25 COUNT .FILL x000F

.END

INTA service routine:

.ORIG x1000 AND R5,R4,#0 ADD R5,R5,#-1 LD R3,VAL

ADD R3,R3,R5 ST R3,VAL RTI

VAL .BLKW 1

.END

INTB service routine:

.ORIG x2000 LDI R4,VAL2 NOT R4,R4 ADD R4,R4,#1 STI R4,VAL2 RTI

VAL2 .FILL xFE08

.END

Assume both interrupts are enabled. Assume 22 cycles are needed to initiate an interrupt when you are in User mode, that is, from the time the test is taken until the interrupt service routine starts executing. Assume it takes 21 cycles if you are in privileged (Supervisor) mode. You already know from problem 1 the number of cycles individual instructions take. In order to support INTA and INTB, the interrupt vector table must have entries. Show the addresses of these entries and the contents of those memory locations.

Content

Memory address

|  |  |
| --- | --- |
|  |  |
|  |  |

Suppose INTA requests service at cycle 30 and INTB requests service at cycle 68. In which cycle does each **service routine** start executing?

The following table shows the contents of a section of memory (locations x2FFA to x3002) before the user program starts executing. Show the contents of these locations and the contents of the stack pointer in

cycle 100.

x2FFA x2FFB x2FFC x2FFD x2FFE x2FFF x3000 x3001 x3002

|  |  |
| --- | --- |
| Initial  x0001 | At the end of  cycle 100 |
| x0010 |  |
| x0100 |  |
| x1000 |  |
| x1100 |  |
| x1110 |  |
| x5020 |  |
| x1025 |  |
| x2207 |  |
| x3000 |  |
|  |  |

Stack Pointer

\***!.&4** Consider a two-player game where the players must think quickly each time it is their turn to make a move. Each player has a total allotted amount of time to make all his/her moves. Two clocks display the remaining time for each player. While a player is thinking of his/her move, his clock counts down. If time runs out, the other player wins. As soon as a player makes his/her move, he hits a button, which serves to stop counting down his clock and start counting down the other player’s clock.

The program on the next page implements this mechanism. The main program keeps track of the time remaining for each player by decrementing the proper counter once per second while the player is

thinking. When a player’s counter reaches zero, a message is printed on the screen declaring the winner. When a player hits the button, an interrupt is taken. The interrupt service routine takes such action as to enable the main program (after returning from the interrupt) to start decrementing the other counter.

The interrupt vector for the button is x35. The priority level of the button is #2. Assume that the operating system has set the interrupt enable bit of

the button to enable it to interrupt. Assume the main program runs at priority #1 and executes in User mode.

In order for the interrupt service routine to be executed when the button is pushed, what memory location must contain what value?

Assume a player hits the button while the instruction at line 16 is being executed. What two values (in hex) will be pushed onto the stack?

Fill in the missing instructions in the user program.

This program has a bug that will only occur if an interrupt is taken at an inappropriate time. Write down the line number of an instruction such that if the button is pressed while that instruction is executing, unintended behavior will result.

How could we fix this bug?

; Interrupt Service Routine

.ORIG x1550 NOT R0, R0 RTI

.END

; User Program

.ORIG x3000

AND R0, R0, #0 ; Line 1

LD R1, TIME ; Line 2

LD R2, TIME ; Line 3

NEXT (a)

(b)

BRn P2\_DEC ; Line 6 ADD R1, R1, #-1 ; Line 7

(c)

LEA R0, P2WINS ; Line 9

BRnzp END ; Line 10 P2\_DEC ADD R2, R2, #-1 ; Line 11

(d)

LEA R0, P1WINS ; Line 13 END PUTS ; Line 14

HALT ; Line 15

COUNT LD R3, SECOND ; Line 16 LOOP ADD R3, R3, #-1 ; Line 17 BRp LOOP ; Line 18

(e) TIME .FILL #300

SECOND .FILL #25000 ; 1 second P1WINS .STRINGZ "Player 1 Wins." P2WINS .STRINGZ "Player 2 Wins."

.END

\***!.55** A program is running in privilege mode (PSR[15] = 0). We set a breakpoint at location x2000. The operator immediately pushes the run button. What are the subsequent MAR/MDR values?

**MDR**

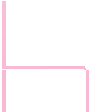
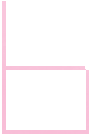
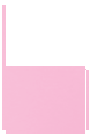
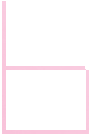
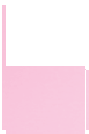
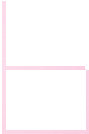
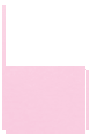
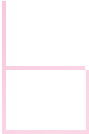
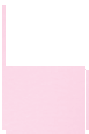
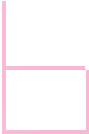
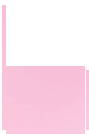
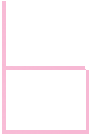
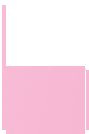
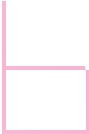
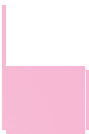
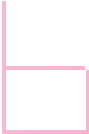
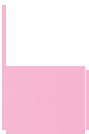
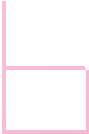
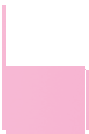
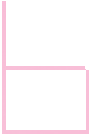
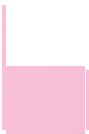
**MAR**

|  |  |
| --- | --- |
|  | x8000  x1050 |
|  | x0004 |
|  | xBCAE |
|  |  |
| x2800 | x2C04 |
|  |  |
| x1052 | x3C4D |
|  | x2C0A |
|  |  |

CHAPTER

10

# A Calculator



efore we leave LC-3 assembly language and raise the level of abstraction to the C programming language, it is useful to step back and pull together much of what we have learned with a comprehensive example. The intent is to demonstrate the use of many of the concepts discussed thus far, as well as to show an example of well-documented, clearly written code, where the example is much

**B**

more complicated than what can fit on one or two pages.

Our example is a program that simulates the actions of a calculator that a per- son can use to add, subtract, and multiply 2’s complement integers. The person will enter numbers into the calculator-simulator by typing keys on the keyboard. Results of a computation will be displayed on the monitor. The calculator sim- ulation consists of a main program and eleven separate subroutines. You are encouraged to study this example before moving on to Chapter 11 and high-level language programming.

Two topics we have not discussed thus far are needed to understand the work- ings of the calculator simulation: (1) the conversion of integers between ASCII strings and 2’s complement, and (2) arithmetic using a stack, the method most calculators use.

The reason for two data types and conversion between them: We need one data type for input/output and another for doing arithmetic. Numbers entered via the keyboard and displayed on the monitor use ASCII codes to represent the numbers. Arithmetic uses 2’s complement integers.

We will need to convert the number the person types from ASCII codes to a 2’s complement integer, and we will need to convert the result of the computa- tion from a 2’s complement integer to ASCII codes in order to display it on the monitor. Section 10.1 deals with data type conversion.

With respect to the way calculators perform arithmetic, the mechanism used by most calculators is very different from the way most desktop and laptop com- puters perform arithmetic. The ISAs of most desktops and laptops are like the LC-3, where arithmetic instructions get their source operands from general pur- pose registers and store the results of the arithmetic operations in general purpose registers. Our simulation of a calculator, like most calculators, does not use gen- eral purpose registers. Instead it uses a stack. Source operands are popped from

the stack, and the result of the operation is pushed back onto the stack. Section

10.2 deals with arithmetic using a stack instead of general purpose registers. Finally, Section 10.3 contains a full discussion of the calculator-simulator,

along with all the subroutines that are needed to make it work.

## !0.! Data Type Conversion

It has been a long time since we talked about data types. We have already been exposed to several data types: unsigned integers for address arithmetic, 2’s complement integers for integer arithmetic, 16-bit binary strings for logical oper- ations, floating point numbers for scientific computation, and ASCII codes for interaction with input and output devices.

It is important that every instruction be provided with source operands of the data type that the instruction requires. For example, an ALU requires operands that are 2’s complement integers to perform an **ADD**. If the ALU were supplied with floating point operands, the ALU would produce garbage results.

It is not uncommon in high-level language programs to find an instruction of the form *A* = *R* + *I* where *R* (floating point) and *I* (2’s complement integer) are represented in different data types.

If the operation is to be performed by a floating point adder, then we have a problem with *I*. To handle the problem, one must first convert the value *I* from its original data type (2’s complement integer) to the data type required by the functional unit performing the operation (floating point). For those programming in some high-level language, the compiler generally produces the code to do that conversion so the programmer does not even have to think about it.

Even in our “character count problem” way back in Chapter 5, we had to deal with data type conversion. Our program entered a character from the keyboard, scanned a file counting the number of occurrences of that character, and then displayed the count on the monitor. Recall that before we could display our final count on the monitor, we had to convert our 2’s complement integer to an ASCII code. Why? Because when we were counting, we were performing arithmetic on 2’s complement integers. But when we were displaying, we needed to represent our count as an ASCII code. You remember we restricted our program to work only on files where the total count was not greater than 9, so our conversion from a 2’s complement integer to an ASCII code could be obtained by simply adding x30 to the 2’s complement integer to get the ASCII code. For example, the 2’s complement representation for 6 (in one byte) is 00000110, or x06. The ASCII code for 6, on the other hand, is 00110110, or x36.

That was a severe limitation to put on our count, restricting it to a single decimal digit. But that was Chapter 5, and now we are in Chapter 10! If our number is represented by more than one decimal digit, simply adding x30 does not work. For example, consider the two decimal digit number 25. If we enter 25 via the keyboard, we input the ASCII code x32, followed by the ASCII code x35. The bit stream is 0011001000110101. To perform arithmetic on this integer, we must first convert it to 0000000000011001, the 2’s complement integer representation of 25. Displaying the result of some arithmetic computation on the monitor causes

a similar problem. To do that, we must first convert the result of the arithmetic (a 2’s complement integer) to an ASCII string.

In this section, we develop routines to convert integers consisting of more than one decimal digit from a string of ASCII codes to 2’s complement, and from 2’s complement to a string of ASCII codes.

##### !0.!.! Example: A Bogus Program: # + $ = e

Before we get into the actual conversion routines, it is worth looking at a simple, concrete example that illustrates their importance. Figure 10.1 shows how we can get into trouble if we do not pay attention to the data types that we are working with.

Suppose we want to enter two single-digit integers from the keyboard, add them, and display the result on the monitor. At first blush, we write the simple program of Figure 10.1. What happens?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 | TRAP | x23 | ; | Input from the keyboard. |
| 02 | ADD | R1,R0,#0 | ; | Make room for another input. |
| 03 | TRAP | x23 | ; | Input another character. |
| 04 | ADD | R0,R1,R0 | ; | Add the two inputs. |
| 05 | TRAP | x21 | ; | Display result on the monitor. |
| 06 | TRAP | x25 | ; | Halt. |

**Figure !0.! ADDITION without paying attention to data types.**

Suppose the first digit entered via the keyboard is a 2 and the second digit entered via the keyboard is a 3. What will be displayed on the monitor before the program terminates? The value loaded into R0 as a result of entering a 2 is the ASCII code for 2, which is x0032. When the 3 is entered, the ASCII code for 3, which is x0033, is loaded into R0 (after the ASCII code for 2 is moved to R1, of course). Thus, the ADD instruction adds the two binary strings x0032 and x0033, producing x0065. When that value is displayed on the monitor, it is treated as an ASCII code. Since x0065 is the ASCII code for a lowercase *e*, a lowercase e is displayed on the monitor.

The reason we did not get 5 (which, at last calculation, is the correct result when adding 2 + 3) is that (a) we didn’t convert the two input characters from ASCII to 2’s complement integers before performing the addition and (b) we didn’t convert the result back to ASCII before displaying it on the monitor.

*Exercise:* Correct Figure 10.1 so that it will add two single-digit positive integers and produce the correct single-digit positive sum. Assume that the two digits being added do in fact produce a single-digit sum.

##### !0.!.# Input Data (ASCII to Binary)

Figure 10.2 shows the ASCII representation of the three-decimal-digit integer 295, stored as an ASCII string in three consecutive LC-3 memory locations, starting at ASCIIBUFF. R1 contains the number of decimal digits in the positive integer. Our ASCII to binary subroutine restricts integers to the range 0 to 999.

ASCIIBUFF

|  |
| --- |
| x0032 |
| x0039 |
| x0035 |
|  |

R1

3

**Figure !0.# The ASCII representation of #$5 stored in consecutive memory locations.**

ASCIIBUFF is the address of the first memory location of a sequence of four memory locations that we have allocated (a) to store the ASCII codes of decimal digits entered from the keyboard, and (b) to store the ASCII codes corresponding to the result of arithmetic operations in preparation for writing it (the result) to the monitor.

You might ask why, in Figure 10.2, we used a whole 16-bit word to store ASCII code of each decimal digit when a byte would have been enough. In

the

fact

, typically, one does store each ASCII code in a single byte of memory. In this mple, we decided to give each ASCII character its own word of memory in

exa

order to simplify the algorithm.

Since we are restricting input to positive integers consisting of at most three decimal digits, you might also ask why we are allocating four words of memory to ASCIIBUFF. Wouldn’t three words be enough? For input yes, but you will see in Section 10.1.3 that in preparation for output, we will need one more word for the sign (positive or negative) of the result, since the result of the arithmetic could be negative.

Figure 10.3 shows the flowchart for a subroutine that converts the ASCII representation of an integer, stored in Figure 10.2, into a binary integer.

The subroutine systematically takes each digit, converts it from its ASCII code to its binary code by stripping away all but the last four bits, and then uses those four bits to index into a table of ten binary values. Since we are restrict- ing conversion to integers consisting of at most three decimal digits, only two tables are needed, one for the tens digit and one for the hundreds digit. Each entry in each table corresponds to the value of one of the ten digits. For example, the entry for index 6 in the hundreds table is the value #600, which is in binary 0000001001011000. That value is then added to R0. R0 is used to accumulate the contributions of all the digits. The result is returned in R0.

Question: If we wanted to be able to convert four decimal-digit integers, ld we need a table of *thousands digits*? Or, is there a way to convert larger numbers represented as larger decimal strings into their binary form without

wou

requiring a table of *thousands* digits, *ten-thousands* digits, etc.?

*Exercise:* [Challenging] Suppose the decimal number is arbitrarily long. her than store a table of 10 values for the thousands-place digit, another table the 10 ten-thousands-place digit, and so on, design an algorithm to do the version without requiring any tables at all. See Exercise 10.4.

Rat

for

con

Figure 10.4 shows the LC-3 code that implements this subroutine.

Yes

R1 ? = 0

(no digits left)

No

Yes R1 ? = 0

(no digits left)

No

Yes R1 ? = 0

(no digits left)

No

Done

R4 <— Hundreds digit R0 <— R0 + 100 \* R4

R4 <— Tens digit R0 <— R0 + 10 \* R4 R1 <— R – 1

R4 <— Units digit R0 <— R0 + R4 R1 <— R1 – 1

R0 < — 0

**Figure !0.$ Flowchart, subroutine for ASCII-to-binary conversion.**

There are two points that we need to make about the subroutines in Chapter 10, which are all part of our calculator simulation, described fully in Section 10.3. First, they cannot be assembled individually, and second (because of that) we need to be sure that no label is used more than once.

Why cannot the subroutine of Figure 10.4 be assembled by itself? Answer: Line 36 specifies a .FILL having the value ASCIIBUFF, but there is no location

01 ;

02 ; This subroutine takes an ASCII string of up to three decimal digits and 03 ; converts it into a binary number. R0 is used to collect the result.

04 ; R1 keeps track of how many digits are left to process. ASCIIBUFF 05 ; contains the most significant digit in the ASCII string.

06 ;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 07 | ASCIItoBinary | ST | R1,AtoB\_Save1 |  | |
| 08 |  | ST | R2,AtoB\_Save2 |
| 09 |  | ST | R3,AtoB\_Save3 |
| 0A |  | ST | R4,AtoB\_Save4 |
| 0B |  | AND | R0,R0,#0 | ; | R0 will be used for our result. |
| 0C  0D  0E | ; | ADD  BRz | R1,R1,#0  AtoB\_Done | ;  ; | Test number of digits.  There are no digits, result is 0. |
| 0F |  | LD | R2,AtoB\_ASCIIBUFF ; R2 points to ASCIIBUFF | | |
| 10 |  | ADD | R2,R2,R1 | | |
| 11 |  | ADD | R2,R2,#-1 ; R2 now points to "ones" digit. | | |
| 12 | ; |  |  | | |
| 13 |  | LDR | R4,R2,#0 ; R4 <-- "ones" digit | | |
| 14 |  | AND | R4,R4,x000F ; Strip off the ASCII template. | | |
| 15 |  | ADD | R0,R0,R4 ; Add ones contribution. | | |
| 16 | ; |  |  | | |
| 17 |  | ADD | R1,R1,#-1 | | |
| 18 |  | BRz | AtoB\_Done ; The original number had one digit. | | |
| 19  1A  1B | ; | ADD  LDR | R2,R2,#-1 ; R2 now points to "tens" digit.  R4,R2,#0 ; R4 <-- "tens" digit | | |
| 1C |  | AND | R4,R4,x000F ; Strip off ASCII template. | | |
| 1D |  | LEA | R3,LookUp10 ; LookUp10 is BASE of tens values. | | |
| 1E |  | ADD | R3,R3,R4 ; R3 points to the right tens value. | | |
| 1F |  | LDR | R4,R3,#0 | | |
| 20 |  | ADD | R0,R0,R4 ; Add tens contribution to total. | | |
| 21 | ; |  |  | | |
| 22 |  | ADD | R1,R1,#-1 | | |
| 23 |  | BRz | AtoB\_Done ; The original number had two digits. | | |
| 24 |  | ADD | R2,R2,#-1 ; R2 now points to "hundreds" digit. | | |
| 25 | ; |  |  | | |
| 26 |  | LDR | R4,R2,#0 ; R4 <-- "hundreds" digit | | |
| 27 |  | AND | R4,R4,x000F ; Strip off ASCII template. | | |
| 28 |  | LEA | R3,LookUp100 ; LookUp100 is hundreds BASE. | | |
| 29 |  | ADD | R3,R3,R4 ; R3 points to hundreds value. | | |
| 2A |  | LDR | R4,R3,#0 | | |
| 2B  2C | ; | ADD | R0,R0,R4 ; Add hundreds contribution to total. | | |
| 2D | AtoB\_Done | LD | R1,AtoB\_Save1 | | |
| 2E |  | LD | R2,AtoB\_Save2 | | |
| 2F |  | LD | R3,AtoB\_Save3 | | |

**Figure !0.# ASCII-to-binary conversion subroutine (Fig. !0.# continued on next page.)**

|  |  |  |  |
| --- | --- | --- | --- |
| 30 |  | LD | R4,AtoB\_Save4 |
| 31 |  | RET |  |
| 32 | ; |  |  |
| 33 | AtoB\_ASCIIBUFF | .FILL | ASCIIBUFF |
| 34 | AtoB\_Save1 | .BLKW | #1 |
| 35 | AtoB\_Save2 | .BLKW | #1 |
| 36 | AtoB\_Save3 | .BLKW | #1 |
| 37 | AtoB\_Save4 | .BLKW | #1 |
| 38 | LookUp10 | .FILL | #0 |
| 39 |  | .FILL | #10 |
| 3A |  | .FILL | #20 |
| 3B |  | .FILL | #30 |
| 3C |  | .FILL | #40 |
| 3D |  | .FILL | #50 |
| 3E |  | .FILL | #60 |
| 3F |  | .FILL | #70 |
| 40 |  | .FILL | #80 |
| 41 |  | .FILL | #90 |
| 42 | ; |  |  |
| 43 | LookUp100 | .FILL | #0 |
| 44 |  | .FILL | #100 |
| 45 |  | .FILL | #200 |
| 46 |  | .FILL | #300 |
| 47 |  | .FILL | #400 |
| 48 |  | .FILL | #500 |
| 49 |  | .FILL | #600 |
| 4A |  | .FILL | #700 |
| 4B |  | .FILL | #800 |
| 4C |  | .FILL | #900 |

**Figure !0.# ASCII-to-binary conversion subroutine (continued Fig. !0.# from previous page.)**

in the subroutine labeled ASCIIBUFF. Therefore, trying to assemble the sub- routine by itself will fail. We could have used .EXTERNAL, discussed briefly in Chapter 7, to enable the subroutines to be assembled individually, but we chose to not do that, preferring to assemble the entire calculator-simulator program includ- ing its eleven subroutines as a single entity. As you would expect, line 43 in the code of Figure 10.15 contains the label ASCIIBUFF.

Second, if we are to assemble the main program and all the subroutines as ngle unit, we need to be sure to not use the same label in more than one subroutine. Note that in Figure 10.4, most labels start with “AtoB .” As expected,

a si

the same pattern of labeling is used in the rest of the subroutines.

##### !0.!.$ Display Result (Binary to ASCII)

To display the result of a computation on the monitor, we must first convert the 2’s complement integer result into an ASCII string. Figure 10.5 shows the subroutine

1. ; This subroutine converts a 2’s complement integer within the range
2. ; -999 to +999 (located in R0) into an ASCII character string consisting 03 ; of a sign digit, followed by three decimal digits, and stores the

04 ; character string into the four memory locations starting at ASCIIBUFF 05 ; (see Figure 10.4).

06 ;

07 BinarytoASCII ST R0,BtoA\_Save0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 08 |  | ST | R1,BtoA\_Save1 | | |
| 09 |  | ST | R2,BtoA\_Save2 | | |
| 0A | ST | | R3,BtoA\_Save3 | | |
| 0B | LD | | R1,BtoA\_ASCIIBUFF | ; | R1 keeps track of output string. |
| 0C | ADD | | R0,R0,#0 | ; | R0 contains the binary value. |
| 0D | BRn | | NegSign | ; |  |
| 0E | LD | | R2,ASCIIplus | ; | First store the ASCII plus sign. |
| 0F | STR | | R2,R1,#0 |  |  |

1. BRnzp Begin100
2. NegSign LD R2,ASCIIminus ; First store ASCII minus sign. 12 STR R2,R1,#0

13 NOT R0,R0 ; Convert the number to absolute 14

|  |  |  |  |
| --- | --- | --- | --- |
| ADD | R0,R0,#1 | ; | value; it is easier to work with. |
| LD | R2,ASCIIoffset | ; | Prepare for "hundreds" digit. |

15 ;

16 Begin100

17 ;

1. LD R3,Neg100 ; Determine the hundreds digit.
2. Loop100 ADD R0,R0,R3

1A BRn End100

1B ADD R2,R2,#1

1C BRnzp Loop100

1D ;

1E End100 STR R2,R1,#1 ; Store ASCII code for hundreds digit. 1F LD R3,Pos100

1. ADD R0,R0,R3 ; Correct R0 for one-too-many subtracts. 21 ;

22 LD R2,ASCIIoffset ; Prepare for "tens" digit. 23 ;

1. Loop10 ADD R0,R0,#-10 ; Determine the tens digit.
2. BRn End10

26 ADD R2,R2,#1

27 BRnzp Loop10

28 ;

29 End10 STR R2,R1,#2 ; Store ASCII code for tens digit.

2A ADD R0,R0,#10 ; Correct R0 for one-too-many subtracts.

|  |  |  |  |
| --- | --- | --- | --- |
| 2B  2C | Begin1 | LD ADD | R2,ASCIIoffset ; Prepare for "ones" digit. R2,R2,R0 |
| 2D |  | STR | R2,R1,#3 |
| 2E |  | LD | R0,BtoA\_Save0 |
| 2F |  | LD | R1,BtoA\_Save1 |
| 30 |  | LD | R2,BtoA\_Save2 |
| 31  32  33 | ; | LD  RET | R3,BtoA\_Save3 |
| 34 | ASCIIplus | .FILL | x002B |
| 35 | ASCIIminus | .FILL | x002D |
| 36 | ASCIIoffset | .FILL | x0030 |
| 37 | Neg100 | .FILL | #-100 |
| 38 | Pos100 | .FILL | #100 |
| 39 | BtoA\_Save0 | .BLKW | #1 |
| 3A | BtoA\_Save1 | .BLKW | #1 |
| 3B | BtoA\_Save2 | .BLKW | #1 |
| 3C | BtoA\_Save3 | .BLKW | #1 |
| 3D | BtoA\_ASCIIBUFF | .FILL | ASCIIBUFF |

**Figure !0.5 Binary-to-ASCII conversion subroutine.**

for converting a 2’s complement integer stored in R0 into an ASCII string stored in the four consecutive memory locations starting at ASCIIBUFF. The value ini- tially in R0 is restricted to the range −999 to +999. After the subroutine completes execution, ASCIIBUFF contains the sign (+ or −) of the value initially stored in R0, followed by three locations that contain the ASCII codes corresponding to the decimal digits representing its magnitude.

The subroutine works as follows: First, the sign of the result to be displayed is determined, and the ASCII code for + or − is stored in ASCIIBUFF. The result (in R0) is replaced by its absolute value. The algorithm determines the hundreds- place digit by repeatedly subtracting #100 from R0 until the result goes negative. This is next repeated for the tens-place digit. The value left is the ones digit.

*Exercise:* This subroutine always produces a string of four characters inde- pendent of the sign and magnitude of the integer being converted. Devise an algorithm that eliminates unnecessary characters; that is, eliminate leading zeros and eliminate a leading + sign. See Exercise 10.6.

## !0.# Arithmetic Using a Stack

##### !0.#.! The Stack as Temporary Storage

You know that the LC-3 ADD instruction takes two source operands that are stored in registers, performs an addition, and stores the result into one of the LC- 3’s eight general purpose registers. We call the register where the result is stored the *destination register*. The eight general purpose registers R0 to R7 comprise the temporary storage that allows operate instructions like ADD to access both source registers and the destination register much more quickly than if the com- puter had to access memory for the operands. Because the three locations are specified explicitly,

ADD R0,R1,R2

we call the LC-3 a three-address machine. Most desktop and laptop computers are either *three-address machines* like the LC-3 or *two-address machines* like the x86 ISA that is implemented in many of your laptop and desktop computers. In a two-address machine, two locations are specified explicitly. An example of an x86 ADD instruction is

ADD EAX,EBX

where EAX and EBX are two of the eight general purpose registers in the x86 ISA. In this case, EAX serves as both the location of one of the source operands and the location of the destination operand. With a two-address machine, one of the source registers is overwritten with the result of the operation.

There are also ISAs that do not use general purpose registers at all to store either source operands or the results of operate instructions. The most common of these are called *stack machines* because a stack is used for temporary storage. Most calculators, including the one we will simulate in Section 10.3, use a stack for temporary storage rather than a set of general purpose registers.

Source operands are obtained by popping the top two elements from the stack. The result (i.e., the destination operand) is subsequently pushed onto the stack. Since the computer always pops and pushes operands from the stack, no addresses need to be specified explicitly. Therefore, stack machines are sometimes referred to as zero-address machines. The instruction would simply be

ADD

and the computer would know where to find the operands. For a calculator, that is convenient because a person can cause an ADD to be performed by simply press- ing the **+** button on the calculator. Note that the pop, push, and add are not part of the ISA of the computer, and therefore they are available to the programmer. They are control signals that the hardware uses to make the actual pop, push, and add occur. The control signals are part of the microarchitecture, similar to the load enable signals and mux select signals we discussed in Chapters 4 and 5. As is the case with LC-3 instructions LD and ST, and control signals PCMUX and LD.MDR, the programmer simply instructs the computer to ADD, and the microarchitecture does the rest.

##### !0.#.# An Example

Suppose we want to evaluate (*A* + *B*) • (*C* + *D*), where *A* contains 25, *B* contains 17, *C* contains 3, and *D* contains 2, and store the result in *E*. If the LC-3 had a multiply instruction (we would probably call it MUL), we could use the following program:

|  |  |
| --- | --- |
| LD | R0,A |
| LD | R1,B |
| ADD | R0,R0,R1 |
| LD | R2,C |
| LD | R3,D |
| ADD | R2,R2,R3 |
| MUL | R0,R0,R2 |
| ST | R0,E |

With a calculator, we would execute the following eight operations:

|  |  |  |
| --- | --- | --- |
| (1) | push | 25 |
| (2) | push | 17 |
| (3) | add |  |
| (4) | push | 3 |
| (5) | push | 2 |
| (6) | add |  |
| (7) | multiply |  |
| (8) | pop | E |

with the final result popped (i.e., 210) being the result of the computation. Figure 10.6 shows a snapshot of the stack after each of the eight operations. Note that in this example we have allocated memory locations x3FFB to x3FFF for our stack, and the stack pointer is initially at x4000, indicating that there is nothing initially on the stack.

In Section 10.3, we write a program that causes the LC-3 (with keyboard and monitor) to act like such a calculator. We say the LC-3 *simulates* the calculator

x3FFB x3FFC x3FFD x3FFE x3FFF

|  |
| --- |
| / / / / / |
| / / / / / |
| / / / / / |
| / / / / / |
| / / / / / |

Stack pointer

x4000

x3FFB x3FFC x3FFD x3FFE x3FFF

Stack pointer

x3FFF

|  |
| --- |
| / / / / / |
| / / / / / |
| / / / / / |
| / / / / / |
| 25 |

x3FFB x3FFC x3FFD x3FFE x3FFF

Stack pointer

x3FFE

|  |
| --- |
| / / / / / |
| / / / / / |
| / / / / / |
| 1# |
| 25 |

1. Before (b) After first push (c) After second push

x3FFB x3FFC x3FFD x3FFE x3FFF

|  |
| --- |
| / / / / / |
| / / / / / |
| / / / / / |
| 1# |
| 42 |

Stack pointer

x3FFF

x3FFB x3FFC x3FFD x3FFE x3FFF

Stack pointer

x3FFE

|  |
| --- |
| / / / / / |
| / / / / / |
| / / / / / |
| 3 |
| 42 |

x3FFB x3FFC x3FFD x3FFE x3FFF

Stack pointer

x3FFD

|  |
| --- |
| / / / / / |
| / / / / / |
| 2 |
| 3 |
| 42 |

(d) After first add (e) After third push (f ) After fourth push

x3FFB x3FFC x3FFD x3FFE x3FFF

|  |
| --- |
| / / / / / |
| / / / / / |
| 2 |
| 5 |
| 42 |

Stack pointer

x3FFE

x3FFB x3FFC x3FFD x3FFE x3FFF

Stack pointer

x3FFF

|  |
| --- |
| / / / / / |
| / / / / / |
| 2 |
| 5 |
| 210 |

x3FFB x3FFC x3FFD x3FFE x3FFF

Stack pointer

x4000

|  |
| --- |
| / / / / / |
| / / / / / |
| 2 |
| 5 |
| 210 |

1. After second add (h) After multiply (i) After pop

**Figure !0.& Stack usage during the computation of (#5** + **!’)** • **($** + **#).**

when it executes that program. To do this, our program will need subroutines to perform the various arithmetic operations.

##### !0.#.$ OpAdd, OpMult, and OpNeg

The program we write in Section 10.3 to simulate a calculator will need three subroutines to be able to perform addition, subtraction, and multiplication. They are:

* 1. OpAdd, which will pop two values from the stack, add them, and push the result onto the stack.
  2. OpMult, which will pop two values from the stack, multiply them, and push the result onto the stack.
  3. OpNeg, which will pop the top value, form its 2’s complement negative value, and push the result onto the stack. This will allow us to subtract two numbers A minus B by first forming −B and then adding the result to A.

***The OpAdd Subroutine*** Figure 10.7 shows the flowchart of the OpAdd sub- routine. Basically, it attempts to pop two values off the stack and, if successful, add them. If the result is within the range of acceptable values (i.e., an integer between −999 and +999), then the result is pushed onto the stack.

There are two things that could prevent OpAdd from completing success-

fully: Fewer than two values are available on the stack for source operands, or

Start

POP

OK?

No

Yes

OK?

No

Yes

Range OK?

No

Yes

Stop

Put back 1st POP

Put back both

PUSH

ADD

POP

**Figure !0.’ Flowchart for OpAdd algorithm.**

the result is out of range. In both cases, the stack is put back to the way it was at the start of the OpAdd subroutine. If the first pop is unsuccessful, the stack is not changed since the POP routine leaves the stack as it was. If the second of the two pops reports back unsuccessfully, the stack pointer is decremented, which effectively returns the first value popped to the top of the stack. If the result is out- side the range of acceptable values, then the stack pointer is decremented twice, returning both values to the top of the stack.

The OpAdd subroutine is shown in Figure 10.8.

Note that the OpAdd subroutine calls the RangeCheck subroutine. This is a simple test to be sure the result of the computation is within what can successfully be stored in a single stack location. For our purposes, we restrict values to integers in the range −999 to +999. This will come in handy in Section 10.3 when we design our home-brew calculator. The flowchart for the RangeCheck subroutine is shown in Figure 10.9. The LC-3 program that implements this subroutine is shown in Figure 10.10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 01 | ; |  | | | | |
| 02 | ; | Subroutine to pop the top two elements from the stack, | | | | |
| 03 | ; | add them, and push the sum onto the stack. R6 is | | | | |
| 04 | ; | the stack pointer. | | | | |
| 05 | ; |  | | | | |
| 06 | OpAdd | | ST | R0,OpAdd\_Save0 |  |  |
| 07 |  | | ST | R1,OpAdd\_Save1 |  |  |
| 08 |  | | ST | R5,OpAdd\_Save5 |  |  |
| 09 |  | | ST | R7,OpAdd\_Save7 |  |  |
| 0A |  | | JSR | POP | ; | Get first source operand. |
| 0B  0C  0D  0E  0F  10 |  | | ADD  BRp ADD JSR ADD  BRp | R5,R5,#0  OpAdd\_Exit R1,R0,#0 POP R5,R5,#0  OpAdd\_Restore1 | ;  ;  ;  ;  ;  ; | Test if POP was successful. Branch if not successful.  Make room for second operand. Get second source operand.  Test if POP was successful.  Not successful, put back first. |
| 11 |  | | ADD | R0,R0,R1 | ; | THE Add. |
| 12 |  | | JSR | RangeCheck | ; | Check size of result. |
| 13 |  | | ADD | R5,R5,#0 | ; | Check R5 for success/failure. |
| 14 |  | | BRp | OpAdd\_Restore2 | ; | Out of range, restore both. |
| 15 |  | | JSR | PUSH | ; | Push sum on the stack. |
| 16 |  | | BRnzp | OpAdd\_Exit | ; | On to the next task... |
| 17 | OpAdd\_Restore2 | | ADD | R6,R6,#-1 | ; | Decrement stack pointer. |
| 18 | OpAdd\_Restore1 | | ADD | R6,R6,#-1 | ; | Decrement stack pointer. |
| 19 | OpAdd\_Exit | | LD | R0,OpAdd\_Save0 |  |  |
| 1A |  | | LD | R1,OpAdd\_Save1 |  |  |
| 1B |  | | LD | R5,OpAdd\_Save5 |  |  |
| 1C |  | | LD | R7,OpAdd\_Save7 |  |  |
| 1D  1E | OpAdd\_Save0 | | RET  .BLKW | #1 |  |  |
| 1F | OpAdd\_Save1 | | .BLKW | #1 |  |  |
| 20 | OpAdd\_Save5 | | .BLKW | #1 |  |  |
| 21 | OpAdd\_Save7 | | .BLKW | #1 |  |  |

**Figure !0.( The OpAdd Subroutine.**



Yes

X > #999

?

No

Yes

X < #–999

?

No

R5 <-- 1

R5 <-- 0

Print (Number out of range)

**Figure !0.) The RangeCheck algorithm flowchart.**

01 ;

02 ; Subroutine to check that a value is 03 ; between -999 and +999.

04 ;

1. RangeCheck LD R5,Neg999
2. ADD R5,R0,R5 ; Recall that R0 contains the
3. BRp BadRange ; result being checked.
4. LD R5,Pos999
5. ADD R5,R0,R5

0A BRn BadRange

0B AND R5,R5,#0 ; R5 <-- success

0C RET

0D BadRange ST R0,RangeCheck\_Save0 0E LEA R0,RangeErrorMsg

0F TRAP x22 ; Output character string

10 AND R5,R5,#0 ;

1. ADD R5,R5,#1 ; R5 <-- failure
2. LD R0,RangeCheck\_Save0
3. RET

14 Neg999 .FILL #-999

15 Pos999 .FILL #999

1. RangeErrorMsg .FILL x000A
2. .STRINGZ "Error: Number is out of range."
3. RangeCheck\_Save0 .BLKW #1

**Figure !0.!0 The RangeCheck Subroutine.**

***The OpMult Subroutine*** Figure 10.11 shows the flowchart of the OpMult subroutine, and Figure 10.12 shows the LC-3 program that implements it. Similar to the OpAdd subroutine, the OpMult subroutine attempts to pop two values off the stack and, if successful, multiplies them. Since the LC-3 does not have a multiply

Start

POP

OK?

No

Yes

OK?

No

Yes

–

Sign of mult.

+

Range OK?

No

Yes

Yes

Flag?

No

Stop

Multiply

Set flag Negate multiplier

Put back 1st POP

Put back both

PUSH

Correct product

POP

**Figure !0.!! Flowchart for the OpMult subroutine.**

01 ;

02 ; Two values are popped from the stack, multiplied, and if 03 ; their product is within the acceptable range, the result 04 ; is pushed onto the stack. R6 is the stack pointer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 05 | ; |  |  |  | |
| 06 | OpMult | ST | R0,OpMult\_Save0 |
| 07 |  | ST | R1,OpMult\_Save1 |
| 08 |  | ST | R2,OpMult\_Save2 |
| 09 |  | ST | R3,OpMult\_Save3 |
| 0A |  | ST | R5,OpMult\_Save5 |
| 0B |  | ST | R7,OpMult\_Save7 |
| 0C |  | AND | R3,R3,#0 | ; | R3 holds sign of multiplier. |
| 0D |  | JSR | POP | ; | Get first source from stack. |
| 0E |  | ADD | R5,R5,#0 | ; | Test for successful POP. |
| 0F |  | BRp | OpMult\_Exit | ; | Failure |
| 10 |  | ADD | R1,R0,#0 | ; | Make room for next POP. |
| 11 |  | JSR | POP | ; | Get second source operand. |
| 12 |  | ADD | R5,R5,#0 | ; | Test for successful POP. |
| 13 |  | BRp | OpMult\_Restore1 | ; | Failure; restore first POP. |
| 14 |  | ADD | R2,R0,#0 | ; | Moves multiplier, tests sign. |
| 15 |  | BRzp | PosMultiplier |  |  |
| 16 |  | ADD | R3,R3,#1 | ; | Sets FLAG: Multiplier is neg. |
| 17 |  | NOT | R2,R2 |  |  |
| 18 |  | ADD | R2,R2,#1 | ; | R2 contains -(multiplier). |
| 19 | PosMultiplier | AND | R0,R0,#0 | ; | Clear product register. |
| 1A |  | ADD | R2,R2,#0 |  |  |
| 1B |  | BRz | PushMult | ; | Multiplier = 0, Done. |
| 1C | ; |  |  |  |  |
| 1D | MultLoop | ADD | R0,R0,R1 | ; | THE actual "multiply" |
| 1E |  | ADD | R2,R2,#-1 | ; | Iteration Control |
| 1F |  | BRp | MultLoop |  |  |
| 20 | ; |  |  |  |  |
| 21 |  | JSR | RangeCheck |  |  |
| 22 |  | ADD | R5,R5,#0 | ; | R5 contains success/failure. |
| 23 |  | BRp | OpMult\_Restore2 |  |  |
| 24 | ; |  |  |  |  |
| 25 |  | ADD | R3,R3,#0 | ; | Test for negative multiplier. |
| 26 |  | BRz | PushMult |  |  |
| 27 |  | NOT | R0,R0 | ; | Adjust for |
| 28 |  | ADD | R0,R0,#1 | ; | sign of result. |
| 29 | PushMult | JSR | PUSH | ; | Push product on the stack. |
| 2A |  | BRnzp | OpMult\_Exit |  |  |
| 2B | OpMult\_Restore2 | ADD | R6,R6,#-1 | ; | Adjust stack pointer. |
| 2C | OpMult\_Restore1 | ADD | R6,R6,#-1 | ; | Adjust stack pointer. |
| 2D | OpMult\_Exit | LD | R0,OpMult\_Save0 |  |  |
| 2E |  | LD | R1,OpMult\_Save1 |  |  |
| 2F |  | LD | R2,OpMult\_Save2 |  |  |
| 30 |  | LD | R3,OpMult\_Save3 |  |  |
| 31 |  | LD | R5,OpMult\_Save5 |  |  |
| 32 |  | LD | R7,OpMult\_Save7 |  |  |
| 33 |  | RET |  |  |  |
| 34 | OpMult\_Save0 | .BLKW | #1 |  |  |
| 35 | OpMult\_Save1 | .BLKW | #1 |  |  |
| 36 | OpMult\_Save2 | .BLKW | #1 |  |  |
| 37 | OpMult\_Save3 | .BLKW | #1 |  |  |
| 38 | OpMult\_Save5 | .BLKW | #1 |  |  |
| 39 | OpMult\_Save7 | .BLKW | #1 |  |  |

**Figure !0.!# The OpMult subroutine.**

instruction, multiplication is performed as we have done in the past as a sequence of adds. Lines 17 to 19 of Figure 10.12 contain the crux of the actual multiply. If the result is within the range of acceptable values, then the result is pushed onto the stack.

If the second of the two pops reports back unsuccessfully, the stack pointer is decremented, which effectively returns the first value popped to the top of the stack. If the result is outside the range of acceptable values, which as before will be indicated by a 1 in R5, then the stack pointer is decremented twice, returning both values to the top of the stack.

***The OpNeg Subroutine*** To perform subtraction with the top two elements on the stack, we first replace the top element on the stack with its negative and then use OpADD. That is, if the top of the stack contains A, and the second element on the stack contains B, we can push B−A on the stack by first negating the top of the stack and then performing OpAdd. The subroutine OpNeg for computing the negative of the element on the top of the stack is shown in Figure 10.13.

01 ; Subroutine to pop the top of the stack, form its negative, 02 ; and push the result onto the stack.

03 ;

1. OpNeg ST R0,OpNeg\_Save0
2. ST R5,OpNeg\_Save5
3. ST R7,OpNeg\_Save7
4. JSR POP ; Get the source operand.
5. ADD R5,R5,#0 ; Test for successful pop
6. BRp OpNeg\_Exit ; Branch if failure.

0A NOT R0,R0

0B ADD R0,R0,#1 ; Form the negative of source.

0C JSR PUSH ; Push result onto the stack. 0D OpNeg\_Exit LD R0,OpNeg\_Save0

0E LD R5,OpNeg\_Save5

0F LD R7,OpNeg\_Save7

* 1. RET
  2. OpNeg\_Save0 .BLKW #1
  3. OpNeg\_Save5 .BLKW #1
  4. OpNeg\_Save7 .BLKW #1

**Figure !0.!$ The OpNeg subroutine.**

## !0.$ The Calculator

##### !0.$.! Functionality

We are now ready to specify all the code for our calculator. As we already said, our calculator is not very sophisticated by today’s standards. It will allow a user to enter positive integers consisting of not more than three decimal digits, perform basic arithmetic (addition, subtraction, and multiplication) on these integers, and display the decimal result (which will also be limited to at most three decimal digits).

We will use the keyboard to tell the calculator what to do. We can enter positive integers having up to three decimal digits, the arithmetic operators + (for ADD), \* (for MUL), and − (for negative), and three additional commands D (to display the result of the calculation on the monitor), C (to erase all values entered), and X (to turn off the calculator).

The calculator algorithm works as follows: We use the keyboard to input commands and decimal values. We use the monitor to display results. We use a stack to hold source operands for performing arithmetic operations and the results of those arithmetic operations, as described in Section 10.2. Values entered and displayed are restricted to three decimal digits, that is, only values between −999 and +999, inclusive.

Figure 10.14 is a flowchart that provides an overview of our algorithm that simulates a calculator. Simulation of the calculator starts with initialization, which includes setting R6, the stack pointer, to an empty stack. Then the user sitting at the keyboard is prompted with: “Enter a Command.”

The following commands are available to the user.

**X** Exit the simulation.

**D** Display the value at the top of the stack.

**C** Clear all values from the stack.

+ Pop the top two elements A,B off the stack and push A+B.

**\*** Pop the top two elements A,B off the stack and push A\*B.

− Pop the top element A off the stack and push “minus” A.

**Enter** or **LF** Push the value typed on the keyboard onto the top of the stack.

If the user wants to enter a number, he/she types the number (up to three decimal digits) followed by <Enter> or <Line Feed (LF)>.

Input is echoed, and the calculator simulation systematically tests the char-

acter to identify the user’s command. Depending on the user’s command, the calculator calls the appropriate subroutine to carry out the work specified. After the work is carried out, the subroutine returns, followed by a prompt for another command. The calculator simulation continues in this way until the user presses X, signaling that the user is finished with the calculator.

For example, to calculate

(51 - 49) \* (172 + 205) - (17 \* 2)

and display the result 720 on the monitor, one types the following sequence of keys on the keyboard: 5,1,LF,4,9,LF,−,+,1,7,2,LF,2,0,5,LF,+,\*,1,7,LF,2,LF,\*,−,+,D.

##### !0.$.# Code

Twelve routines comprise the calculator simulation. Figure 10.15 is the main algorithm, supported by eleven subroutines. Note the three global labels, Stack- Max, StackBase, and ASCIIBUFF, are all part of the main algorithm, shown in Figure 10.15. They provide the symbol table entries needed by the subroutines that reference those locations. Note also that the stack has been allocated ten

Start

Initialize

(i.e., clear stack)

X

Yes

Exit

No

C

No

Yes

+

Yes

No

\*

Yes

No

−

Yes

No

D

Yes

No

Push value

(see Figure 10.16)

(uses ASCII−to−binary conversion)

Display

(see Figure 10.19)

(uses binary−to−ASCII conversion)

OpNeg

(see Figure 10.13)

OpMult

(see Figure 10.12)

OpAdd

(see Figure 10.8)

Clear stack

(see Figure 10.20)

Prompt user Get char

**Figure !0.!4 The calculator, overview.**

01 ;

02 ; The Calculator, Main Algorithm 03 ;

1. LEA R6,StackBase ; Initialize the Stack Pointer.
2. ADD R6,R6,#1 ; R6 = StackBase + 1 --> empty stack 06

07 NewCommand LEA R0,PromptMsg 08 PUTS

09 GETC

0A OUT

0B ;

0C ; Check the command 0D ;

0E TestX LD R1,NegX ; Check for X. 0F ADD R1,R1,R0

1. BRnp TestC
2. HALT

12 ;

1. TestC LD R1,NegC ; Check for C.
2. ADD R1,R1,R0
3. BRnp TestAdd
4. JSR OpClear ; See Figure 10.20
5. BRnzp NewCommand 18 ;
6. TestAdd LD R1,NegPlus ; Check for + 1A ADD R1,R1,R0

1B BRnp TestMult

1C JSR OpAdd ; See Figure 10.8

1D BRnzp NewCommand 1E ;

1F TestMult LD R1,NegMult ; Check for \*

1. ADD R1,R1,R0
2. BRnp TestMinus
3. JSR OpMult ; See Figure 10.12
4. BRnzp NewCommand 24 ;
5. TestMinus LD R1,NegMinus ; Check for -
6. ADD R1,R1,R0
7. BRnp TestD
8. JSR OpNeg ; See Figure 10.13
9. BRnzp NewCommand 2A ;

2B TestD LD R1,NegD ; Check for D 2C ADD R1,R1,R0

2D BRnp EnterNumber

2E JSR OpDisplay ; See Figure 10.19

2F BRnzp NewCommand 30 ;

31 ; Then we must be entering an integer 32 ;

1. EnterNumber JSR PushValue ; See Figure 10.16
2. BRnzp NewCommand

|  |  |  |  |
| --- | --- | --- | --- |
| 35 | ; | | |
| 36 | PromptMsg .FILL x000A | | |
| 37 | .STRINGZ "Enter a command:" | | |
| 38 | NegX | .FILL | xFFA8 |
| 39 | NegC | .FILL | xFFBD |
| 3A | NegPlus | .FILL | xFFD5 |
| 3B | NegMinus | .FILL | xFFD3 |
| 3C | NegMult | .FILL | xFFD6 |
| 3D | NegD | .FILL | xFFBC |
| 3E  3F | ; Globals |  |  |
| 40 | StackMax | .BLKW | #9 |
| 41 | StackBase | .BLKW | #1 |
| 42 | ASCIIBUFF | .BLKW | #4 |
| 43 |  | .FILL | x0000 ; ASCIIBUFF sentinel |

**Figure !0.!5 The calculator’s main algorithm.**

entries in the main algorithm, and R6, the stack pointer, is initialized to an empty stack in line 05.

Figure 10.16 takes an ASCII string of characters terminating by a LF, checks to be sure it corresponds to a string of not more than three decimal digits, and if so, converts it to a binary number, then pushes the binary number onto the top of the stack. Figure 10.4 provides the ASCII-to-binary conversion routine. Figure 10.19 pops the entry on the top of the stack, converts it to an ASCII character string, and displays the ASCII string on the monitor. Figure 10.5 provides the binary- to-ASCII conversion routine. Figures 10.8 (OpAdd), 10.12 (OpMult), and 10.13 (OpNeg) supply the basic arithmetic algorithms using a stack. Figures 10.17 and 10.18 contain the basic POP and PUSH routines Finally, Figure 10.20 clears the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 01 | ; This subroutine takes a sequence of not more than three decimal digits | | | | |  |
| 02 | ; typed by the user, converts its ASCII string to a binary value using the | | | | |
| 03 | ; ASCIItoBinary subroutine, and pushes the binary value onto the stack. | | | | |
| 04 | ; Anything else typed results in an error message. | | | | |
| 05 | ; | | | | |
| 06 | PushValue ST R0,PushValue\_Save0 | | | | |
| 07 | ST R1,PushValue\_Save1 | | | | |
| 08 | ST R2,PushValue\_Save2 | | | | |
| 09  0A | ST R7,PushValue\_Save7  LD R1,PushValue\_ASCIIBUFF ; R1 points to string | | | | | being |
| 0B  0C | ; | LD | R2,MaxDigits |  | ; generated. | |
| 0D  0E  0F  10 | ValueLoop | ADD  BRz ADD  BRz | R3,R0,x-0A  GoodInput R2,R2,#0  TooLargeInput | ; | Test for line feed, x0A | |
| 11 |  | LD | R3,NEGASCII0 |  |  | |
| 12 |  | ADD | R3,R0,R3 |  |  | |
| 13 |  | BRn | NotInteger |  |  | |
| 14 |  | LD | R3,NEGASCII9 |  |  | |
| 15 |  | ADD | R3,R0,R3 |  |  | |
| 16 |  | BRp | NotInteger |  |  | |
| 17 |  | ADD | R2,R2,#-1 | ; | Still room for more digits. | |
| 18 |  | STR | R0,R1,#0 | ; | Store last character read. | |
| 19 |  | ADD | R1,R1,#1 |  |  | |
| 1A |  | GETC |  |  |  | |
| 1B  1C  1D | ; | OUT  BRnzp | ValueLoop | ; | Echo it. | |
| 1E | GoodInput | LD | R2,PushValue\_ASCIIBUFF | | | |
| 1F |  | NOT | R2,R2 | | | |
| 20 |  | ADD | R2,R2,#1 | | | |
| 21 |  | ADD | R1,R1,R2 ; R1 now contains no. of char. | | | |
| 22 |  | BRz | NoDigit | | | |
| 23 |  | JSR | ASCIItoBinary | | | |
| 24 |  | JSR | PUSH | | | |
| 25 |  | BRnzp | PushValue\_Done | | | |

**Figure !0.!# The calculator’s PushValue routine (Fig. !0.!# continued on next page.)**

1. NoDigit LEA R0,NoDigitMsg
2. PUTS
3. BRnzp PushValue\_Done
4. NotInteger GETC ; Spin until carriage return. 2A OUT

2B ADD R3,R0,x-0A ; Test for line feed, x0A

2C BRnp NotInteger

2D LEA R0,NotIntegerMsg

2E PUTS

2F BRnzp PushValue\_Done

1. TooLargeInput GETC ; Spin until carriage return.
2. OUT
3. ADD R3,R0,x-0A ; Test for line feed, x0A
4. BRnp TooLargeInput
5. LEA R0,TooManyDigits
6. PUTS
7. PushValue\_Done LD R0,PushValue\_Save0
8. LD R1,PushValue\_Save1
9. LD R2,PushValue\_Save2
10. LD R7,PushValue\_Save7

3A RET

3B TooManyDigits .FILL x000A

3C .STRINGZ "Too many digits"

3D NoDigitMsg .FILL x000A

3E .STRINGZ "No number entered" 3F NotIntegerMsg .FILL x000A

1. .STRINGZ "Not an integer"
2. MaxDigits .FILL x0003
3. NegASCII0 .FILL x-30
4. NegASCII9 .FILL x-39
5. PushValue\_ASCIIBUFF .FILL ASCIIBUFF
6. PushValue\_Save0 .BLKW #1
7. PushValue\_Save1 .BLKW #1
8. PushValue\_Save2 .BLKW #1
9. PushValue\_Save7 .BLKW #1

**Figure !0.!# The calculator’s PushValue routine (continued Fig. !0.!# from previous page.)**

01 ; This subroutine POPs a value from the stack and puts it in 02 ; R0 before returning to the calling program. R5 is used to

03 ; report success (R5 = 0) or failure (R5 = 1) of the POP operation.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 04 | POP | LD | R0,POP\_StackBase |  | |
| 05 |  | NOT | R0,R0 | ; | R0 = -(addr. of StackBase + 1) |
| 06 |  | ADD | R0,R0,R6 | ; | R6 = StackPointer |
| 07 |  | BRz | Underflow |  |  |
| 08 |  | LDR | R0,R6,#0 | ; | The actual POP |
| 09 |  | ADD | R6,R6,#1 | ; | Adjust StackPointer |
| 0A |  | AND | R5,R5,#0 | ; | R5 <-- success |
| 0B |  | RET |  |  |  |
| 0C | Underflow | LEA | R0,UnderflowMsg |  |  |
| 0D |  | PUTS |  | ; | Print error message. |
| 0E |  | AND | R5,R5,#0 |  |  |
| 0F |  | ADD | R5,R5,#1 | ; | R5 <-- failure |
| 10 |  | RET |  |  |  |
| 11 | UnderflowMsg | .FILL | x000A |  |  |

* 1. .STRINGZ "Error: Too Few Values on the Stack."
  2. POP\_StackBase .FILL StackBase

**Figure !0.!’ The calculator’s POP routine.**

01 ; This subroutine PUSHes on the stack the value stored in R0. 02 ; R5 is used to report success (R5 = 0) or failure (R5 = 1) of 03 ; the PUSH operation.

04 PUSH ST R1,PUSH\_Save1 ; R1 is needed by this routine. 05 LD R1,PUSH\_StackMax

1. NOT R1,R1
2. ADD R1,R1,#1 ; R1 = - addr. of StackMax
3. ADD R1,R1,R6 ; R6 = StackPointer
4. BRz Overflow

0A ADD R6,R6,#-1 ; Adjust StackPointer for PUSH.

0B STR R0,R6,#0 ; The actual PUSH

0C LD R1,PUSH\_Save1 ; Restore R1.

0D AND R5,R5,#0 ; R5 <-- success

0E RET

0F Overflow LEA R0,OverflowMsg

1. PUTS
2. LD R1,PUSH\_Save1 ; Restore R1.

12 AND R5,R5,#0

1. ADD R5,R5,#1 ; R5 <-- failure
2. RET
3. PUSH\_Save1 .BLKW #1
4. OverflowMsg .FILL x000A
5. .STRINGZ "Error: Stack is Full."
6. PUSH\_StackMax .FILL StackMax

**Figure !0.!( The calculator’s PUSH routine.**

1. ; This subroutine calls BinarytoASCII to convert the 2’s complement 02 ; number on the top of the stack into an ASCII character string, and 03 ; then calls PUTS to display that number on the screen.

|  |  |  |
| --- | --- | --- |
| 04 | OpDisplay ST | R0,OpDisplay\_Save0 |
| 05 | ST | R5,OpDisplay\_Save5 |
| 06 | ST | R7,OpDisplay\_Save7 |
| 07 | JSR | POP ; R0 gets the value to be displayed. |
| 08 | ADD | R5,R5,#0 |
| 09 | BRp | OpDisplay\_DONE ; POP failed, nothing on the stack. |
| 0A | JSR | BinarytoASCII |
| 0B | LD | R0,NewlineChar |
| 0C | OUT |  |
| 0D | LD | R0,OpDisplay\_ASCIIBUFF |
| 0E | PUTS |  |
| 0F | ADD | R6,R6,#-1 ; Push displayed number back on stack. |
| 10 | OpDisplay\_DONE LD | R0,OpDisplay\_Save0 |
| 11 | LD | R5,OpDisplay\_Save5 |
| 12 | LD | R7,OpDisplay\_Save7 |
| 13 | RET |  |
| 14 | NewlineChar .FILL | x000A |
| 15 | OpDisplay\_ASCIIBUFF .FILL | ASCIIBUFF |
| 16 | OpDisplay\_Save0 .BLKW | #1 |
| 17 | OpDisplay\_Save5 .BLKW | #1 |
| 18 | OpDisplay\_Save7 .BLKW | #1 |

**Figure !0.!) The calculator’s display routine.**

01 ;

1. ; This routine clears the stack by resetting the stack pointer (R6). 03 ;

04 OpClear LD R6,OpClear\_StackBase ; Initialize the Stack Pointer. 05 ADD R6,R6,#1 ; R6 = StackBase + 1 --> empty stack

1. RET
2. OpClear\_StackBase .FILL StackBase

**Figure !0.#0 The OpClear routine.**

**Exercises**

**!0.!** Describe, in your own words, how the Multiply step of the OpMult algorithm in Figure 10.14 works. How many instructions are executed to perform the Multiply step? Express your answer in terms of *n*, the value of the multiplier. (*Note:* If an instruction executes five times,

it contributes five to the total count.) Write a program fragment that

performs the Multiply step in fewer instructions if the value of the multiplier is less than 25. How many?

**!0.#** Correct Figure 10.1 so that it will add two single-digit positive integers and produce a single-digit positive sum. Assume that the two digits being added do in fact produce a single-digit sum.

**!0.$** Modify Figure 10.1, assuming that the input numbers are one-digit positive hex numbers. Assume that the two hex digits being added together do in fact produce a single hex-digit sum.

Exercises **403**

**!0.4** Figure 10.4 provides an algorithm for converting ASCII strings to binary values. Suppose the decimal number is arbitrarily long. Rather than store a table of 10 values for the thousands-place digit, another table for the 10 ten-thousands-place digit, and so on, design an algorithm to do the conversion without resorting to any tables whatsoever.

**!0.5** The code in Figure 10.4 converts a decimal number represented as ASCII digits into binary. Extend this code to also convert a hexadecimal number represented in ASCII into binary. If the number is preceded by an x, then the subsequent ASCII digits (three at most) represent a hex number; otherwise, it is decimal.

**!0.&** The algorithm of Figure 10.5 always produces a string of four characters independent of the sign and magnitude of the integer being converted. Devise an algorithm that eliminates unnecessary characters in common representations, that is, an algorithm that does not store leading 0s nor a leading + sign.

**!0.’** What does the following LC-3 program do?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | .ORIG LEA LEA TRAP | x3000  R6, STACKBASE R0, PROMPT  x22 | ; | PUTS |
| AND | R1, R1, #0 |  |  |
| LOOP | TRAP | x20 | ; | IN |
|  | TRAP | x21 |  |  |
| INPUTDONE LOOP2  DONE | ADD  BRz JSR ADD  BRnzp ADD  BRz JSR TRAP ADD  BRp TRAP | R3, R0, #-10 INPUTDONE PUSH  R1, R1, #1 LOOP  R1, R1, #0 DONE  POP  x21  R1, R1, #-1 LOOP2  x25 | ;  ; | Check for newline  HALT |
| PUSH | ADD | R6, R6, #-2 |  |  |
|  | STR RET | R0, R6, #0 |  |  |
| POP | LDR | R0, R6, #0 |  |  |
|  | ADD RET | R6, R6, #2 |  |  |

PROMPT .STRINGZ ‘‘Please enter a sentence: ’’ STACKSPAC .BLKW #50

STACKBASE .FILL #0

.END

\***!0.(** The calculator program assumes that if the user did not type one of the characters X,C,+,−,\*,D, then it must be pushing a value and so executes BRnzp PushValue. Modify the program so it is more robust;

that is, if the user typed something other than a digit, the main program would load R0 with the ASCII code for X, and branch to Test. If the user typed a digit, the main program would branch to PushValue.

\***!0.)** For the calculator program in Exercise 10.8, improve the robustness by modifying PushValue to make sure all the characters typed are digits.