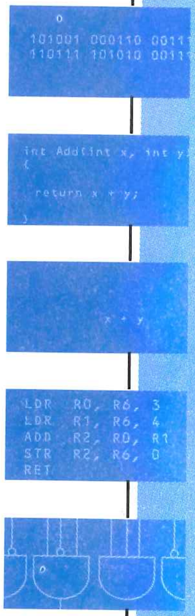
chapter

6



Programming

We are now ready to start developing programs to solve problems with the computer. In this chapter we attempt to do two things: first, we develop a methodology for constructing programs; and second, we develop a methodology for fixing those programs under the likely condition that we did not get it 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 *debugging*. The opportunities for introducing bugs into a complicated program are so great that it usually takes much more time to get the program to work (debugging) than it does to create it in the first place.

6.1 Problem Solving

6.1.1 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 some 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 transform 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 finiteness (it terminates), definiteness (each step is precisely stated), and effective computability (each step can be carried out by a computer).

In the late 1960s, the concept of *structured programming* emerged as a way to improve the ability of average programmers to take a complex description of a problem and systematically decompose it into sufficiently smaller, manageable units that they could ultimately write as a program that executed correctly. The mechanism 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.

6.1.2 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 down into smaller units of work such that the collection 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*.

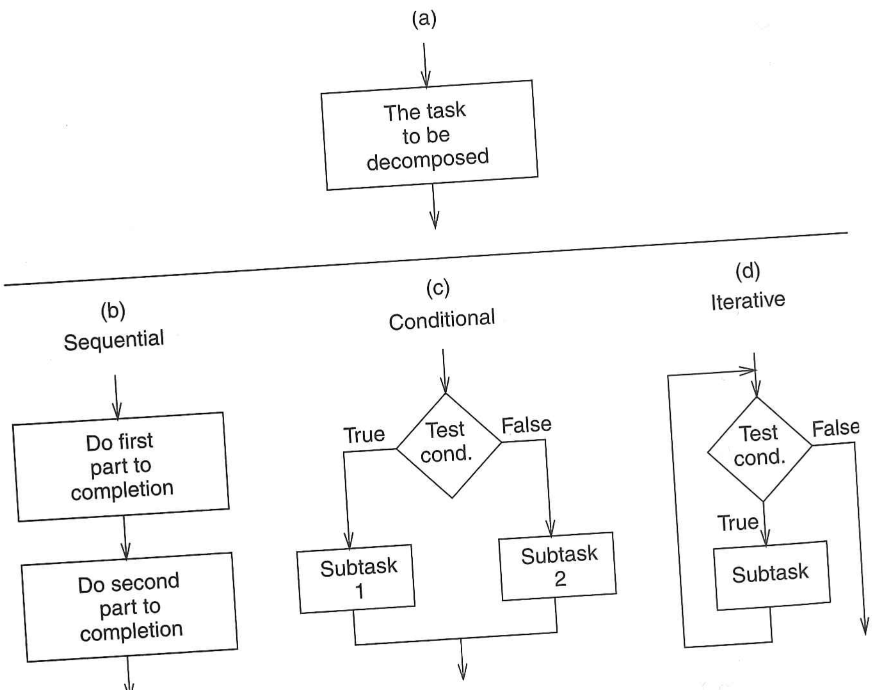


Figure 6.1 The basic constructs of structured programming

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 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. 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., Figures 6.1b, 1c, and 1d), there is exactly *one entrance into the construct* and *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 in the following example.

6.1.3 LC-3 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 decomposition constructs. That is, Figures 6.2b, 6.2c, and 6.2d correspond respectively to the three constructs shown in Figures 6.1b, 6.1c, and 6.1d.

We use the letters A, B, C, and D to represent addresses in memory containing LC-3 instructions. A, for example, is used in all three cases to represent the address of the first LC-3 instruction to be executed.

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 B_1 to Address B_1+1. The program continues to execute instructions through address D_1. 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 B_2. If the condition is true, the PC is set to Address C_2+1, and subtask 1 is executed. (Note: x corresponds to the number of instructions in subtask 2.) If the condition is false, the PC (which had been incremented during the FETCH

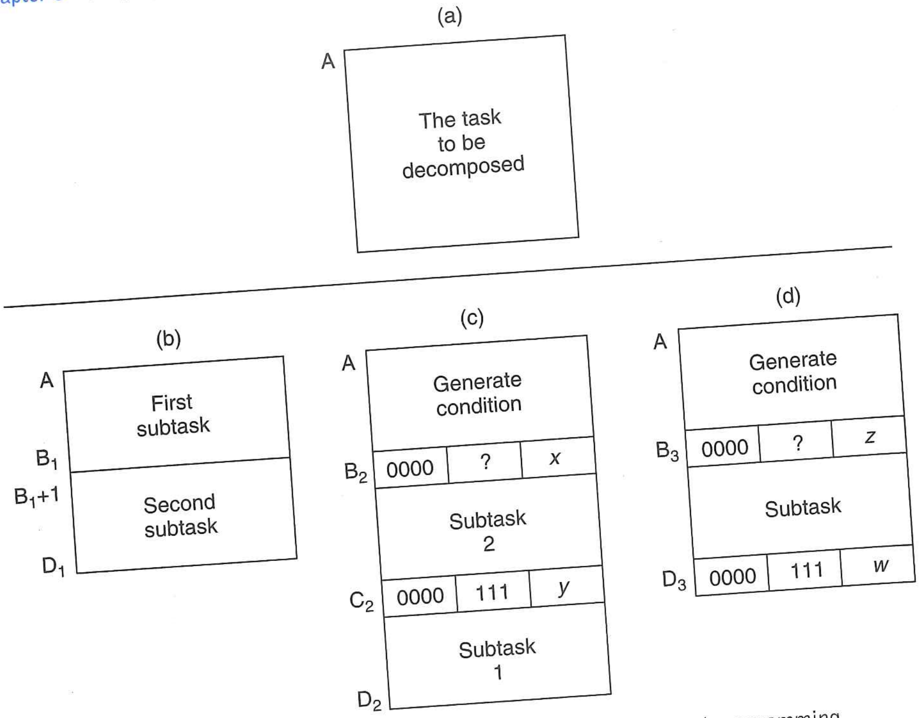


Figure 6.2 Use of LC-3 control instructions to implement structured programming

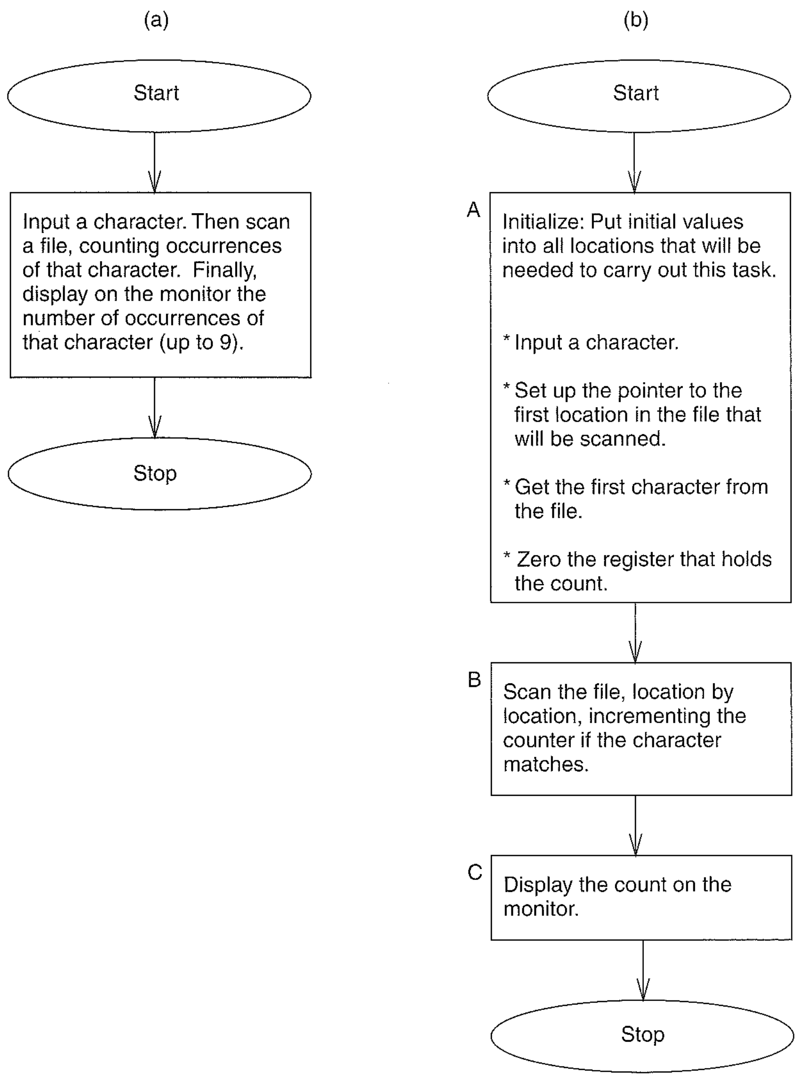
phase of the branch instruction) fetches the instruction at Address B_2+1, and subtask 2 is executed. Subtask 2 terminates in a branch instruction that at Address C_2 unconditionally branches to D_2+1. (Note: *y* corresponds to the number of instructions in subtask 1.)

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 condition code is set, and a conditional branch is executed. In this case, the condition bits of the instruction at address B_3 are set to cause a conditional branch if the condition generated is false. If the condition is false, the PC is set to address D_3+1. (Note: *z* corresponds to 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 B_3+1, and the subtask will be executed. The subtask terminates in an unconditional branch instruction at address D_3, which sets the PC to A to again generate and test the condition. (Note: *w* corresponds to the total number of instructions in the decomposition shown as Figure 6.2d.)

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

6.1.4 The Character Count Example from Chapter 5, Revisited

Recall the example of Section 5.5. The statement of the problem is as follows: "We wish to count the number of occurrences of a character in a file. The character



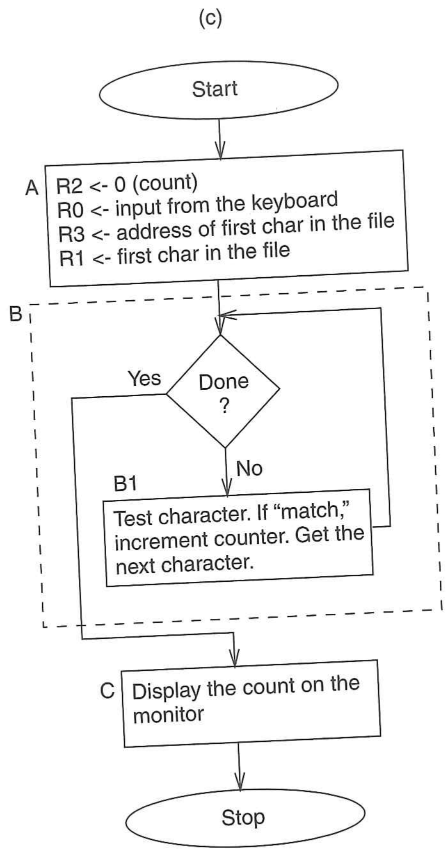
(a)
Start
Input a character. Then scan a file, counting occurrences of that character. Finally, display on the monitor the number of occurrences of that character (up to 9).
Stop
(b)
Start
A
Initialize: Put initial values into all locations that will be needed to carry out this task.
* Input a character.
* 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.
B
Scan the file, location by location, incrementing the counter if the character matches.
C
Display the count on the monitor.
Stop

Figure 6.3 Stepwise refinement of the character count program

in question is to be input from the keyboard; the result is to be displayed 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 interest 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.


(c)
Start
A
R2 <- 0 (count)
R0 <- input from the keyboard
R3 <- address of first char in the file
R1 <- first char in the file
B
Yes
Done ?
No
B1
Test character. If "match," increment counter. Get the next character.
C
Display the count on the monitor
Stop

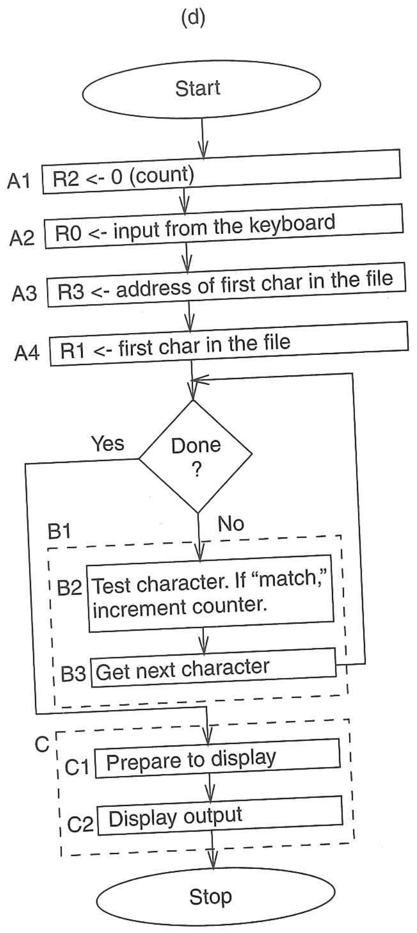

(d)
Start
A1 R2 <- 0 (count)
A2 R0 <- input from the keyboard
A3 R3 <- address of first char in the file
A4 R1 <- first char in the file
Yes
Done ?
No
B1
B2 Test character. If "match," increment counter.
B3 Get next character
C
C1 Prepare to display
C2 Display output
Stop

Figure 6.3 Stepwise refinement of the character count program (**continued**)

To do this, we will need a mechanism for scanning all the characters in a file, and we will need a counter so that when we find a match, we can increment that counter.

We will need places 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 some mechanism for knowing when the file terminates.

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 actual process of determining how many

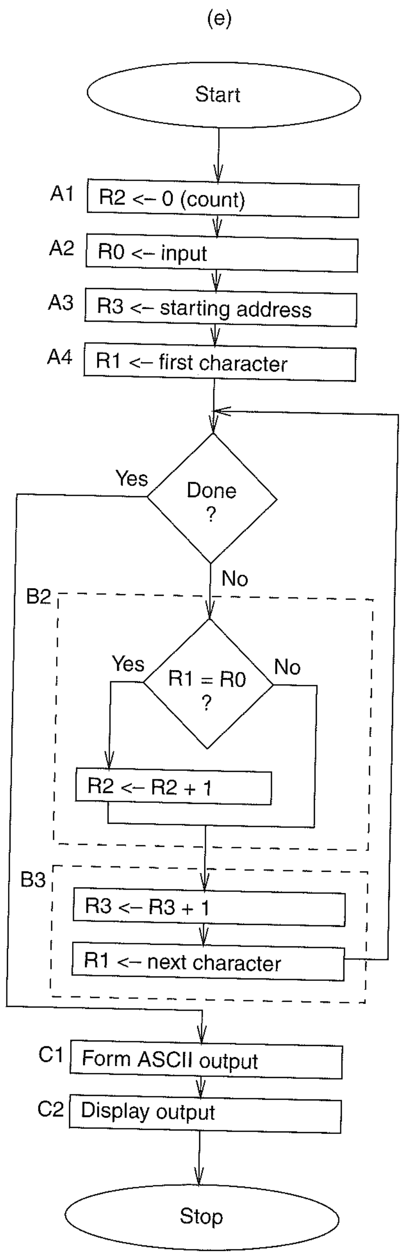


Figure 6.3 Stepwise refinement of the character count program (continued)

occurrences 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 getting the first character from the file. Collectively, these four steps comprise the initialization of the algorithm shown in Figure 6.3b as A.

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. This program uses the sentinel method by terminating the file we are examining with an EOT (end of text) character. The test to see if there are more legitimate characters in the file is a test for the ASCII code for EOT.

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 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 ASCII (C1), and then the actual character output is performed (C2).

Finally, Figure 6.3e completes the decomposition, replacing B2 with the elements 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 the easy part, actually) is to write the LC-3 code corresponding to each box in Figure 6.3e. Note that Figure 6.3e is essentially identical to Figure 5.7 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 possible 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 proceed, you are ready to return to square one (Figure 6.3a) and restart the process of systematically decomposing the problem.

6.2 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 "driving 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 provided, 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 you take a wrong turn. The simplest way to keep track of where you are as 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 control 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 program 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 execution 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.

6.2.1 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, we will examine some debugging techniques available through **dbx**, the source-level debugger for the programming language C. Right now, however, we wish to stay at the level of the machine architecture, and so we will see what we can accomplish with a few very elementary interactive debugging operations. When debugging interactively, the user sits in front of the keyboard and monitor 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. Deposit values in memory and in 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.

Set Values

It is useful to deposit values in memory and in registers 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. 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 module. 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 placing an ASCII code in R0.

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. 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 instructions 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 computer 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 every instruction executed.

The **Set Breakpoint** command causes the program to stop execution at a specific instruction in a program. Executing the debugging command Set Breakpoint 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 the address of the 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, thereby allowing the user to examine what has been computed up to that point. (When one no longer wishes to have the program stop execution at that point, one can remove the breakpoint by executing the Clear Breakpoint command.)

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.

6.2.2 Examples: Use of the Interactive Debugger

We conclude this chapter with four examples, showing how the use of the 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 executes 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.

Example 1: Multiplying Without a Multiply Instruction

Consider the program of Figure 6.4a. The goal of the program is to multiply the two positive numbers contained in R4 and R5. A program is necessary since the LC-3 does not have a multiply instruction.

If we go through the program instruction by instruction, we note that the program first clears R2 (that is, 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 sounds like the program should work! Upon execution, however, we find that if R4 is initially 10 and R5 is initially 3, the program produces the value 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 description of **exactly** what each instruction is doing. While this can be very tedious and not

(a)

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3200	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	`R2 <- 0`
x3201	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	`R2 <- R2 + R4`
x3202	0	0	0	1	1	0	1	1	0	1	1	1	1	1	1	1	`R5 <- R5 - 1`
x3203	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	`BRzp x3201`
x3204	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`

(b)

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

(c)

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

Figure 6.4 The use of interactive debugging to find the error in Example 1. (a) An LC-3 program to multiply (without a Multiply instruction). (b) A trace of the Multiply program. (c) Tracing with breakpoints.

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. We have included in Figure 6.4a, next to each instruction,
such an annotation.

Figure 6.4b 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. If we examine the contents of the registers, we see that the
branch condition codes were set wrong; that is, the conditional branch should be
taken as long as R5 is positive, not as long as R5 is nonnegative, as is the case in
x3203. That causes an extra iteration of the loop, resulting in 10 being added to
itself four times, rather than three.

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

We should also note that we could have saved some of the work of tracing the
program by using a breakpoint. That is, instead of examining the results of **each**
instruction, setting a breakpoint at x3203 allows us to examine the results of
each iteration of the loop. Figure 6.4c shows the results of tracing the program,
where each step is one iteration of the loop. We see that the loop executed four
times rather than three, as it should have.

One last comment before we leave this example. Before we started tracing
the program, we initialized R4 and R5 with values 10 and 3. When testing a
program, it is important to judiciously choose the initial values for the test. Here,
the program stated that the program had to work only for positive integers. So, 10
and 3 are probably OK. What if a (different) multiply program had been written
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 values, and a good test of such a program
is to initialize its variables to the unusual values, the ones the programmer may
have failed to consider. These values are often referred to colloquially as *corner cases*.

Example 2: Adding a Column of Numbers

The program of Figure 6.5 is supposed to add the numbers stored in the 10
locations starting with x3100, and leave the result in R1. The contents of the 20
memory locations starting at location x3100 are shown in Figure 6.6.

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
x3002, the loop control (R4), which counts the number of values added to R1, is

(a)

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	`R1 <- 0`
x3001	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	`R4 <- 0`
x3002	0	0	0	1	1	0	0	1	0	0	1	0	1	0	1	0	`R4 <- R4 + 10`
x3003	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	0	`R2 <- M[x3100]`
x3004	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	`R3 <- M[R2]`
x3005	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	`R2 <- R2 + 1`
x3006	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	`R1 <- R1 + R3`
x3007	0	0	0	1	1	0	0	1	0	0	1	1	1	1	1	1	`R4 <- R4 - 1`
x3008	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	1	`BRp x3004`
x3009	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`

(b)

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

Figure 6.5 The use of interactive debugging to find the error in Example 2. (a) An LC-3 program to add 10 integers. (b) A trace of the first four instructions of the Add program

Address	Contents
x3100	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 6.6 Contents of memory locations x3100 to x3113 for Example 2

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 loop again (x3008). After 10 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 program 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 6.5b 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 x3100 into R2, not the **address** x3100. Our mistake: We should have used the opcode 1110, which would have loaded the address of x3100 into R2. We correct the bug by replacing the opcode 0010 with 1110, and the program runs correctly.

Example 3: Determining Whether a Sequence of Memory Locations Contains a 5

The program of Figure 6.7 has been written to examine the contents of the 10 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 work as follows: The first six instructions (at x3000 to x3005) initialize R0 to 1, R1 to −5, and R3 to 10. In each case, the register is first cleared by ANDing it with 0, and then ADDing the corresponding immediate value. For example, in x3003, −5 is added to R1, and the result is stored in R1.

The instruction at x3006 initializes R4 to the starting address (x3100) of the values to be tested, and x3007 loads the contents of x3100 into R2.

x3008 and x3009 determine if R2 contains the value 5 by adding −5 to it 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.

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 location x3108, 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 se at x300D. The results are shown in Figure 6.7b.

(a)

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	0	0	0	1	0	0	0	0	0	`R0 <- 0`
x3001	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	`R0 <- R0 + 1`
x3002	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	`R1 <- 0`
x3003	0	0	0	1	0	0	1	0	0	1	1	1	1	0	1	1	`R1 <- R1 - 5`
x3004	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	`R3 <- 0`
x3005	0	0	0	1	0	1	1	0	1	1	1	0	1	0	1	0	`R3 <- R3 + 10`
x3006	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1	`R4 <- M[x3010]`
x3007	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	`R2 <- M[R4]`
x3008	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	`R2 <- R2 + R1`
x3009	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	`BRz x300F`
x300A	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	`R4 <- R4 + 1`
x300B	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	`R3 <- R3 - 1`
x300C	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	`R2 <- M[R4]`
x300D	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	`BRp x3008`
x300E	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	`R0 <- 0`
x300F	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`
x3010	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	`x3100`

(b)

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

Figure 6.7 The use of interactive debugging to find the error in Example 3. (a) An LC-3 program to detect the presence of a 5. (b) Tracing Example 3 with a breakpoint at x300D.

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.

However, the value 0 stored in x3103 causes the load instruction at x300C to clear the P condition code. This, in turn, causes the branch at x300D not to be taken, R0 is set to 0 (x300E), and the program terminates (x300F).

The error in the program was putting a load instruction at x300C between x300B, which kept track of how many values still needed to be tested, and x300D, the branch instruction that returned to x3008 to perform the next test. The load instruction sets condition codes. Therefore, the branch at x300D was based on the value loaded into R2, rather than on the count of how many values remained to be tested. If we remove the instruction at x300C and change the target of the branch in x300D to x3007, the program executes correctly.

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 presently. The program of Figure 6.8 has been written to examine the contents of a memory location, find the first bit (reading left to right) that is set, and store the bit position of that bit into R1. If no bit is set, the program is to store −1 in R1. For example, if the location examined contained 0010000000000000, the program would terminate with R1 = 13. If the location contained 0000000000000100, the program would terminate with R1 = 2.

(a)

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	`R1 <- 0`
x3001	0	0	0	1	0	0	1	0	0	1	1	0	1	1	1	1	`R1 <- R1 + 15`
x3002	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	`R2 <- M[M[x3009]]`
x3003	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	`BRn x3008`
x3004	0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	`R1 <- R1 - 1`
x3005	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	`R2 <- R2 + R2`
x3006	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	`BRn x3008`
x3007	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	`BRnzp x3004`
x3008	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`
x3009	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	`x3100`

(b)

PC	R1
x3007	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 6.8 The use of interactive debugging to find the error in Example 4. (a) An LC-3 program to find the first 1 in a word. (b) Tracing Example 4 with a breakpoint at x3007.

The program is supposed to work as follows (and it usually does): x3000 and x3001 initialize R1 in the same way as we have done in the previous examples. In this case, R1 is initialized to 15.

x3002 loads R2 with the contents of x3100, the value to be examined. It does this by the load indirect instruction, which finds the location of the value to be loaded in x3009.

x3003 tests the high bit of that value, and if it is a 1, it branches to x3008, where the program terminates with R1 = 15. If the high bit is a 0, the branch is not taken and R1 is decremented (x3004), indicating the next bit to be tested is bit [14].

In x3005, the value in R2 is added to itself, and the result is stored back in R2. That is, the value in R2 is multiplied by 2. This is the same as shifting the contents of R2 one bit to the left. This causes the value in bit [14] to move into the bit [15] position, where it can be tested by a branch on negative instruction. x3006 performs the test of bit [14] (now in the bit [15] position), and if the bit is 1, the branch is taken, and the program terminates with R1 = 14.

If the bit is 0, x3007 takes an unconditional branch to x3004, where the process repeats. That is, R1 is decremented (x3004), indicating the next lower bit number, R2, is shifted one bit to the left (x3005), and the new occupant of bit [15] is tested (x3006).

The process continues until the first 1 is found. The program works 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 illuminating. 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 x3100 was x0000; that is, there were no 1s present. The program worked fine as long as there was at least one 1 present. For the case where x3100 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. It is also often the most important one. 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. We will see more examples of this kind of error later in the book.

Exercises

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

6.2 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.

6.3 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.

6.4 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 $R1 < R2$.

6.5 Which of the two algorithms for multiplying two numbers is preferable and why? $88 \cdot 3 = 88 + 88 + 88$ OR $3 + 3 + 3 + 3 + \ldots + 3$?

6.6 Use your answers from Exercises 6.3 and 6.4 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.

6.7 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
```

6.8 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?

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

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

6.11 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.

6.12 *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.

b. 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.

6.13 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.

6.14 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 in R2 to be 3?

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

6.16 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									
x3001	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																

6.17 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