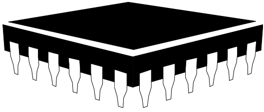
A model microprocessor



We present the assembler language language of a model processor and the processor-simulator software, *asm.exe*, that is available online at *www.framingham.edu/faculty/dkeil/asm_setup.exe*. You may run the programs you see here, and you may test your own assembler-language programs on the simulator.

A set of operations

Our imaginary processor can recognize and execute fourteen different operations. (Real processors often provide more than 200 operations.) Although a processor accepts instructions only in the form of binary numbers, we will write these instructions using mnemonics. A mnemonic is a name or abbreviation that suggests what an operation does. A language that has one mnemonic for each machine-language instruction of a processor is called an *assembly language*. Our model processor simulator assembles and executes programs that are written in the assembler language described here.

Before formally presenting the vocabulary of the language of our model processor, let's look at a short example program.

```
// ECHO.ASM: Accepts number, displays it.
     input  num
     print  num
     stop
num  data   0
```

The first line is a *comment*. The two slashes at the beginning of the line tell the system software to ignore it. A comment is included for the information of a human reader. A good guideline in writing software is to put a comment at the beginning of any program and anywhere else where your intention might be unclear to a person reading the program.

The second line, *input num*, instructs the computer to prompt the user for an integer and to wait until the user enters one at the keyboard. The value entered is stored at a RAM location named *num*. The last line of the program, *num data 0*, reserves a RAM location for *num*.

The third line of the program, *print num*, causes the value at location *num* to be displayed on the monitor.

The fourth line, *stop*, terminates execution.

A sample run of *echo.asm* yields the following results.

Sample I/O:

```
[Input:] 60
[Output:] 60
```

The numeric value, 60, in our sample depends only on the action of the user, so it cannot be considered part of the program itself; it's just an arbitrary value.

Here is the assembly language interpreted by our model processor:

Operation mnemonic	Integer code	Meaning
stop	0	Stop program execution.
load	1	Copy to the accumulator (ACC) data from a named RAM address. (The *load* instruction must be distinguished from the operation of loading a program into RAM.)
store	2	Copy current contents of ACC to a named RAM address.

add	3	Add the data value at a named RAM address to current ACC contents, leaving the result in the ACC.
sub	4	Subtract data value at the named RAM address from current ACC contents, leaving the result in the ACC.
input	5	Wait for the user to enter an integer at the keyboard and copy it to a named RAM address.
print	6	Display on screen the integer stored at the named RAM address.
jump–	7	If current ACC value is negative, set the program counter (PC) to the address named in operand (*jump–* thus interrupts execution of instructions in their stored order, making it possible either to bypass a sequence of instructions or to return to a previously executed instruction.)
jump0	8	If current ACC value is zero, reset the PC to a named address. (See *jump–*.)
jump	9	Set the PC to a named address, without regard to the value in the ACC.
sinput	10	Wait for the user to enter a character string at the keyboard and copy it to a named RAM address, 8 characters per 64-bit cell.
sprint	11	Display on the screen the character string stored at the named RAM address.
call	12	To call a subroutine, change the value in the program counter to the operand value and store the current instruction address.
return	13	To return from a subroutine, retrieve the instruction address stored by the most recent outstanding *call* instruction and put that in the program counter.

Load, *store* and *input* are destructive operations in that they replace existing data. A value *load*ed into the ACC replaces the existing contents. A valued *store*d at, or *input* into, a RAM cell replaces the previous contents of the cell.

Programs for our model processor consist of instruction statements and data statements. Instruction statements specify actions to be taken when the program runs. Data statements specify data values to be stored in RAM when the program is loaded.

Instruction statements

You may have noticed that the descriptions of all but two of the fourteen operations refer to a named RAM address. An instruction statement consists of at most three parts: a label, an operation mnemonic (*load*, *add*, etc.), and an *operand*.

A *label*, whose name is chosen by the programmer, is required if we wish to give a name to the address of a program statement. All data is referred to using lables for the addresses of data items.

An *operation mnemonic* is part of every instruction statement. It tells what action is to be taken.

An *operand* is part of every instruction statement except for the *stop* statement. The operand is the named RAM address mentioned in each operation definition except for the *stop* definition. If we wish to have a statement trigger a jump to another statement, then the jump statement must name the destination of the jump using the other statement's label.

Here is an example of an instruction statement:

With it, we are saying that an integer will be accepted from the keyboard and copied to a location named *n*. This instruction is labelled *start*, so that a different program statement may perform a jump to it with the instruction, *jump start*, for example.

The input/output instructions (*input* and *print*) are designed to copy data to and from RAM, not to and from the accumulator. To get a value from the user and put it in the accumulator, for example, would require two steps: an *input* and a *load*. To display the value that is in the accumulator would require a *store* instruction and a *print* instruction.

An instruction statement without a label must begin with at least one space; otherwise the assembler software must assume that the word "input," for example, is a label.

Data statements

In addition to instruction statements, a program for our model computer must contain one data statement for every RAM address used by the program to store data. A data statement consists of a *label*, the word *data* or *sdata*, and a *data value*. The *label* identifies the RAM address where the data value is stored.

The word *data* or *sdata* identifies the statement as a data statement, as opposed to an instruction statement. A statement with the word *data* in it allocates space for an integer. The *data value* may be any integer in the range -2^{63} to $2^{63}-1$. This range is determined by the number of bits (64) available in a storage location. The data value is stored at the RAM address identified by the label when the program is loaded into memory.

Here is an example of a data statement:

From it we know that a storage location named *n* is assigned an initial value of 275. Each instruction statement and each data statement occupies a separate memory cell when the program is loaded into RAM.

Let's look at a complete program for the model computer. Program *double.asm*, below, doubles an input value and prints the original value and the doubled value. Unlike *echo.asm*, it displays a prompt for the user and identifies its output, using the *sprint* instruction.

RAM
Address

```
        // DOUBLE.ASM: Accepts integer, displays its value doubled.
 0                sprint   prompt     Tell user what to do
 1                input    innum      Wait for user input to cell <innum>
 2                load     innum      Copy data at address <innum> into ACC
 3                add      innum      Add data at <innum> to value in ACC
 4                store    sum        Copy sum in ACC into address <sum>
 5                print    innum      Display value stored at <innum>
 6                sprint   outlabl
 7                print    sum        Display value stored at <sum>
 8                stop                End program execution
 9      innum     data     0          <innum> is a label for RAM cell 9
 10     sum       data     0          <sum> is a label for RAM cell 10
```

```
11    prompt    sdata    "Enter an integer: "
12    outlabl   sdata    "doubled is"
```

Sample I/O:

```
Enter an integer: 13
13 doubled is 26
```

The first line of this program begins with a double slash. In the language of our assembler, which translates from mnemonics and labels to binary coding, a double slash signals for the rest of the line to be ignored. Any text after the operand is treated as a comment as well. Extensive comments are included here to help a new programmer like yourself understand how the program does its job. Comments are not loaded into RAM cells and are ignored by the computer.

At the end of the program are four data statements. The first reads

```
innum  data  0
```

It causes one RAM cell to be allocated and associated with the label *innum*, and for the value 0 to be stored there until some other value replaces it. The address of this cell is 9, because the data statement is the tenth in the program, and each statement stores an instruction or data item in one cell, starting with 0.

The third data statement reads

```
prompt sdata "Enter an integer: "
```

It causes the string in quotes to be stored in three consecutive memory locations. Eight characters fit in each cell because each cell holds 64 bits and each character occupies eight bits. Any value that follows *sdata* should appear in double quotes.

With all program examples where it is appropriate, we will include sample input and output for a typical 'run' (execution) of the program. We will use the abbreviation 'I/O' for 'Input/Output'.

Let's note that on real computers, accepting and displaying data are complex operations. Each of the machine operations, *input*, *print*, *sinput* and *sprint*, for the model computer would consist of several operations in a real machine language, including calls to a part of the operating system called the BIOS (Basic Input/Output System). Consider these questions with reference to the *input* operation:

- How does the processor determine if and when a key has been pressed?
- How does the processor know whether or how to display the character associated with the pressed key?
- How is it known where on the screen the character is to be displayed?

In our program examples, the initial zeros specified in data statements are often arbitrary values. They will never be used, because the first statements that refer to the labels defined by these data statements are input statements. A data statement requires that some value be specified, whether meaningful to the program or not. Data statements can provide a means of storing needed data in RAM at program load time. The following program, *add3.asm*, uses data statements to store data values at cells 6, 7 and 8 when the program is loaded into RAM. During program execution, these three stored values are added and the sum is displayed.

RAM
Address

	// ADD3.ASM: Displays the sum of 129, -374, and 248			
0		load	x	Load ACC from cell 6 (X)
1		add	y	Add data at cell 7 (Y) to ACC contents
2		add	z	Add data at cell 8 (Z) to ACC contents
3		store	sum	Copy ACC data to cell 9 (SUM)
4		print	sum	Display data in cell 9
5		stop		End program execution
6	x	data	129	cell 6 starts with the value 129
7	y	data	-374	cell 7 starts with the value -374
8	z	data	248	cell 8 starts with the value 248
9	sum	data	0	

Sample I/O:

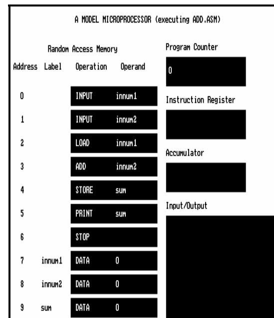
```
[Output:] 3
```

Program *add3* finds the sum of the three integers named in its three data statements, using the accumulator to store its subtotals. It then displays that sum. The output is 3 because 129 + –374 + 248 equals 3.

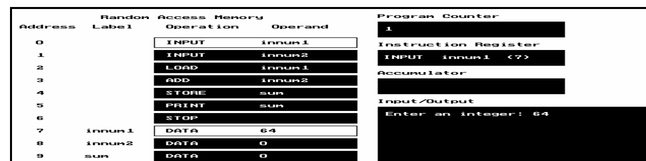
The fetch-execute cycle

To execute a program, the operating system loads the program file into RAM from the disk drive. The program is loaded into memory in sequence, one instruction statement or data statement per memory cell, with the first instruction always placed at cell 0. At program loading time, the program counter (PC) is set to an initial value of 0.

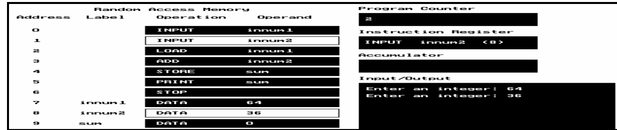
Below is the model computer with program *add.asm* (from subdirectory *01* of your work disk) loaded into memory. Notice that the first instruction statement (*input innum1*) is loaded into cell 0, and the Program Counter (PC) is set initially to 0. After the program is loaded into RAM, it runs to completion by repeating a *fetch-execute cycle* until a *stop* instruction is executed.



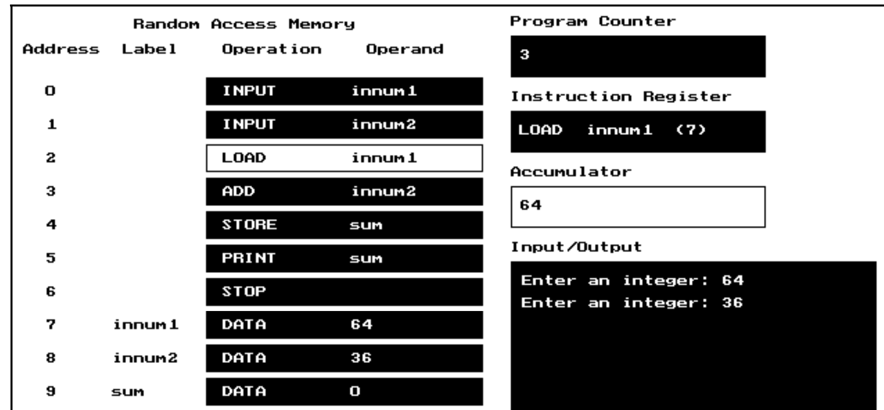
Let's step through program *add*, instruction by instruction. There are seven instructions in the program. Seven repetitions of the fetch-execute cycle will occur. The panels that follow show the fetch-execute cycle for each instruction.



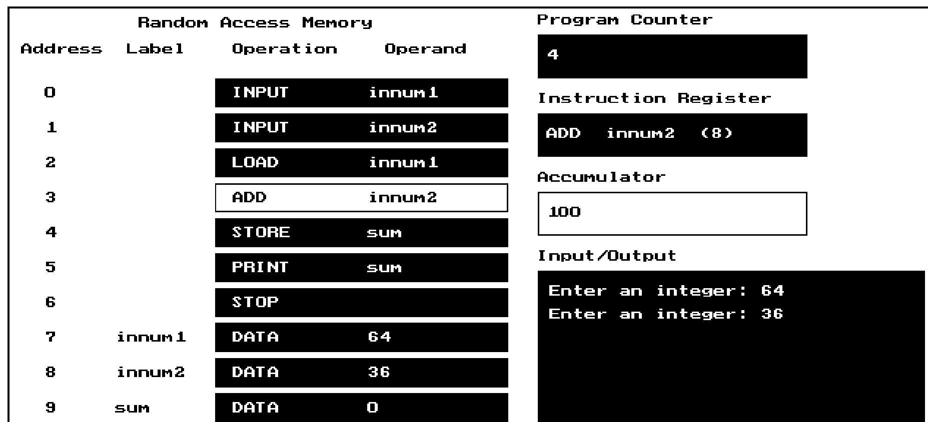
1. Instruction at cell 0 (*input innum1*) fetched into IR.
2. PC incremented from 0 to 1.
3. Wait for user to enter a number. Copy user input (64) to cell 7 (*innum1*).



1. Instruction at cell 1 (*input innum2*) fetched into IR.
2. PC incremented from 1 to 2.
3. Wait for user to enter a number. Copy user input (36) to cell 8 (*innum2*).



1. Instruction at cell 2 (*load innum1*) fetched into IR.
2. PC incremented from 2 to 3.
3. Load data value (64) at *innum1* (cell 7) into ACC.



1. Instruction at cell 3 (*add innum2*) fetched into IR.
2. PC incremented from 3 to 4.
3. Data value (36) at *innum2* (cell 8) added to contents of accumulator (64), leaving 100 in ACC.

Address	Label	Operation	Operand
0		INPUT	innum1
1		INPUT	innum2
2		LOAD	innum1
3		ADD	innum2
4		STORE	sum
5		PRINT	sum
6		STOP	
7	innum1	DATA	64
8	innum2	DATA	36
9	sum	DATA	100

Random Access Memory

Program Counter: 5

Instruction Register: STORE sum (9)

Accumulator: 100

Input/Output:

```
Enter an integer: 64
Enter an integer: 36
```

1. Instruction at cell 4 (*store sum*) fetched into IR.
2. PC incremented from 4 to 5.
3. Store contents of accumulator (100) at *sum* (cell 9).

Address	Label	Operation	Operand
0		INPUT	innum1
1		INPUT	innum2
2		LOAD	innum1
3		ADD	innum2
4		STORE	sum
5		PRINT	sum
6		STOP	
7	innum1	DATA	64
8	innum2	DATA	36
9	sum	DATA	100

Random Access Memory

Program Counter: 6

Instruction Register: PRINT sum (9)

Accumulator: 100

Input/Output:

```
Enter an integer: 64
Enter an integer: 36
100
```

1. Instruction at cell 5 (*print sum*) fetched into IR.
2. PC incremented from 5 to 6.
3. Output data value (100) at cell 9 (*sum*) to monitor.

Address	Label	Operation	Operand
0		INPUT	innum1
1		INPUT	innum2
2		LOAD	innum1
3		ADD	innum2
4		STORE	sum
5		PRINT	sum
6		STOP	
7	innum1	DATA	64
8	innum2	DATA	36
9	sum	DATA	100

Random Access Memory

Program Counter: 7

Instruction Register: STOP End

Accumulator: 100

Input/Output:

```
Enter an integer: 64
Enter an integer: 36
100
```

1. Statement at cell 6 (*stop*) fetched into IR.
2. PC incremented from 6 to 7.
3. Stop. Don't fetch another instruction.

How to use the branch control structure in assembler language

Consider the problem of telling whether the user is old enough to vote. The following pseudocode describes a solution:

If age is at least 18
    Say "OK"

This algorithm can be implemented in our model processor's assembler language as follows:

```
// IS18.ASM: Says "OK" on input 18 or over
          sprint   prompt
          input    age
          load     age
          sub      min_age
          jump-    exit   Do nothing if age < 18
          sprint   ok
exit      stop
age       data     0
prompt    sdata    "Enter your age: "
min_age   data     18
ok        sdata    "OK to vote"
```

Sample I/O:

Run 1:

```
Enter your age: 17
```

Run 2:

```
Enter your age: 18
OK to vote
```

The *jump–* operation is a *conditional* jump, causing a jump only when the contents of the accumulator are negative.

The *jump* operation is shown in a flowchart by an arrow: *jump–*, *jump0*, and *jump* reset the program counter (PC) to a named RAM address. By skipping or repeating instructions, the jump operations allow a program to execute statements in other than sequential order. Different instruction sequences within the program can be executed, depending upon the value in the accumulator.

Example: comparing input values

Consider the problem of displaying the greater of two input values. Here is a solution:

If $a > b$
    display a
otherwise
    display b

To code the algorithm in our assembler language, we must provide two output instructions: one to show the first number, and another to show the second. Furthermore, we must provide for each of these instructions to be skipped over in case the other is to be executed. Figure A- 1 shows a more detailed flowchart, written to correspond very closely to the assembler-language statements that are needed:

```
// GREATER.ASM: Displays the greater of two input values
          input     a
          input     b
          load      a         If (a - b) < 0, then b > a, so jump
          sub       b         past "print a" to "print b"
          jump-     bgrtr     If (a - b) > or = 0, then print <a>
          print     a         and jump past "print b" to "stop"
          jump      end
bgrtr     print     b
end       stop
a         data      0
b         data      0
```

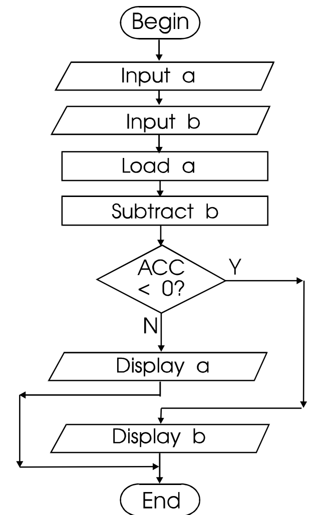


Figure A- 1: Flowchart for **greater.asm**

Sample I/O:

Run 1:

```
[Input:] 13
[Input:] 29
[Output:] 29
```

Run 2:

```
[Input:] 3409
[Input:] 29
[Output:] 3409
```

Run 3:

```
[Input:] 100
[Input:] 100
[Output:] 100
```

This program copies the first input value into the accumulator and subtracts the second, leaving the difference in the accumulator. If the difference is negative, the second must have been larger, so the program causes a jump to an instruction that will display its value. Otherwise, the first number is displayed and a jump occurs to the end of the program (the instruction labelled *end*), skipping the instruction to display the second number.

Program *greater.asm* illustrates a common situation that we must handle in writing assembly-language programs. We must sometimes execute one instruction (or group of instructions) or another, but not both. *A common mistake is to fall through from an instruction that should be executed to one that should be bypassed.* The key is to "play leapfrog." (When we begin our study of C++, we will not have to concern ourselves with this problem. The C++ *if...else* branch statement, discussed in Chapter 6, takes care of "jumping" for us.)

Let's focus on just the branch structure portion of program *greater*, beginning with making a decision based on the contents of the accumulator. Notice how the flowchart highlights the leapfrog concept.

The instructions *jump–* and *jump0* are conditional jumps. Whether or not a jump occurs depends on the contents of the accumulator. The instruction *jump*, on the other hand, is unconditional. It resets the program counter to the RAM address named in the operand, regardless of the contents of the accumulator.

How to write a loop in assembler language

Whereas in assembler and machine languages a branch is written as a conditional jump *downward* in the program, a loop uses a jump *upward or backward*. The jump is to

an already-executed statement, thus repeating that statement and those following it until the jump statement.

Example: adding four input values

The program below solves the problem of displaying the sum of four values obtained from the user. The following pseudocode sketches a solution:

1. Set sum to 0.
2. Do four times, using a counter (*count*) to keep track of how many values currently remain to be input:
   a. input number
   b. add number to sum.
3. Display sum.

The informal "do four times" translates to a *counted loop* that starts at the statement labelled *start*:

```
// ADD4.ASM: accepts 4 integers, displays sum
start    load    count   If the count has decreased to 0,
         jump0   prt     print the sum of the 4 input values
         sub     one     If COUNT > 0, then decrease
         store   count   COUNT by one,
         input   n       input another value,
         load    sum     and add it to SUM.
         add     n
         store   sum
         jump    start   Return to the top of the loop.
prt      print   sum
         stop
count    data    4
one      data    1
n        data    0
sum      data    0
```

Sample I/O:

```
[Input:] 20
[Input:] 84
[Input:] -62
[Input:] 7
[Output:] 49
```

This program assigns the value 4 to *count* at the time the program loads. The loop body extends from the statement labelled *start* to the *jump start* statement. Looping occurs at that statement, which causes program control to jump from the end of the loop to the beginning.

The statement that exits from the loop is *jump0 prt*, which triggers a jump out of the loop whenever a zero is found in the accumulator as the value loaded from memory location *count*.

The control aspect of the loop consists in subtracting one from *count*, jumping out of the loop as soon as *count* reaches zero. The body of the loop repeatedly gets a value, *n*, from the user and adds this value to *sum*. At the end of the program, *sum* equals the sum of all values entered by the user, and *sum* is displayed.

Example: using a sentinel value

A variation of the previous problem is to loop, getting input and accumulating a sum, until the user enters a special *sentinel value*. The flowchart, from Chapter 2, is in Figure A- 2.

An assembler-language version of the solution is:

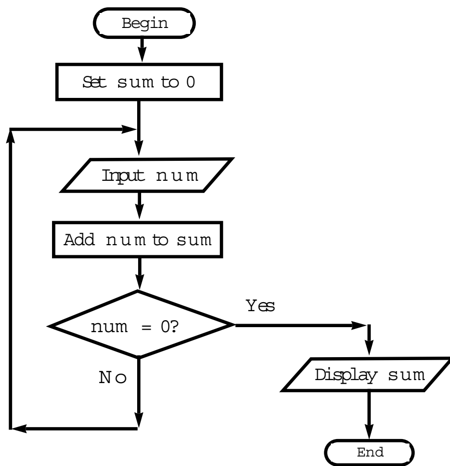


Figure A- 2: Sentinel-controlled loop

```
// ADD_N.ASM: Gets input until user enters zero; displays sum.
start  input   n
       load    sum
       add     n        Add input value to <sum>,
       store   sum      store it
       load    n
       jump0   exit
       jump    start
exit   stop
n      data    0
sum    data    0
```

Sample I/O:

```
[Input:] 3
[Input:] 5
[Input:] 0
[Output:] 8
```

Three instructions in this program, starting at line 3, are represented by a single box in the flowchart:

```
load    sum
add     n
store   sum
```

A sequence of instructions, beginning with a *load* and ending with a *store* to the same address from which the *load* occurred, amounts to assigning a new value to a *variable.* You studied variables in algebra; much of your work in C++ will involve assigning values to variables.

A simulator that runs assembler programs

An executable program on your work disk, *asm.bat*, can graphically demonstrate how assembler and machine code are used to control a microprocessor. The examples shown above are stored as text files in subdirectory 01 of your work disk, ready for *asm* to load and run.

The program *asm* does the following:

- It displays the names of all *.asm* files it finds in the current disk directory and prompts the user to enter the name of one of them to load and run.
- It asks the user whether to step through the program, in order to observe the execution of each instruction, or to run the program uninterruptedly.
- It displays the components of the processor described in this appendix. If the attempt to load the named *.asm* file is unsuccessful, an error message is displayed; otherwise, each instruction and data value of the *.asm* program is displayed in the RAM cell into which it has been loaded.
- When the program has executed, *asm* gives the user the option to load and run the same program again (for example, if you want to test the program with two or more sets of input data).

Tutorial

Let's use program example *add.asm* to illustrate the entire process of loading and running an assembler-language program. We'll do this as a hands-on activity, and the steps below assume that you're at a PC-compatible computer keyboard.

1. Run the processor simulator.
   You are shown a list of *.asm* files in directory 01 and prompted for the name of a program. Choose *add.* This loads the file *add.asm* into the simulator.
2. You are prompted to choose between step mode and fast mode. Step mode allows you to observe the status of every RAM cell and each of the three registers (ACC, IR, and PC) after each instruction executes. Press the "f" key for fast program execution.
3. You will see the model processor displayed on the computer screen, with the executable program instructions and data values loaded into RAM, beginning at cell 0. The screen looks like this:

A MODEL MICROPROCESSOR (executing ADD.ASM)

Random Access Memory

Address	Label	Operation	Operand
0		INPUT	innum1
1		INPUT	innum2
2		LOAD	innum1
3		ADD	innum2
4		STORE	sum
5		PRINT	sum
6		STOP	
7	innum1	DATA	0
8	innum2	DATA	0
9	sum	DATA	0

Program Counter: 0

Instruction Register

Accumulator

Input/Output

4. The program, *add.asm*, begins to execute on the screen. The input/output area prompts you:

   `Enter an integer:`

   with a blinking cursor. The processor is executing the first instruction:

   `input innum1`

   which was copied from the disk into RAM cell 0 when the program loaded, and which has just been copied from cell 0 into the instruction register (IR). Enter an integer.
5. Another prompt will appear; enter a second integer.
6. Since you chose fast (rather than step) execution of *add.asm*, and since there are no more *input* instructions, the program runs to completion. What values are displayed in the input/output area? Does the program do what it was designed to do? Look at the data values in RAM cells 7 (*innum1*) and 8 (*innum2*), and at the contents of the three registers (ACC, IR, and PC).
7. If you choose to run again, then *Add.asm* is loaded again, the PC is reset to 0, and you must again choose either Step or Fast execution. If you choose Step (S), then watch the *Input/Output* area for prompts, and try to predict -what changes will occur in the RAM cells and registers as each instruction executes.
8. When program execution is complete, the program prompts you to decide whether to run *add.asm* another time. Make your choice.
9. When, at some point, you elect not to load and run *add.asm* again, the list of program files is redisplayed. Press [Enter] to return to the DOS prompt.

Open *add.out* in Notepad. *Add.out* is a record of the input and output for each consecutive run of the file *add.asm*. When we tested this hands-on activity, we entered the integers 64 and 36. Our *add.out* file looked like this:

```
OUTPUT OFADD.ASM:
-------------------
[Input:] 64
[Input:] 36
[Output:] 100
```

Writing programs for the model processor

To code an assembly-language program for the model processor, you must create a text file with the file name extension *.asm*.

Although you can use any text editor to create the file, we suggest that you begin to familiarize yourself with the C++ programming environment you will be using beginning with Chapter 3. A good starting point is learning to use its text editor. If you do use a C++ compiler's editor to create *.asm* files, you must remember to include the extension *.asm* when naming the file you wish to create or edit. If you don't, the editor may assume you're creating a C++ program file, and automatically assign the file name extension *.cpp*.

Another way to create *.asm* program files is to use the DOS editor, *edit*, the Windows *Notepad* editor or a word processor. If you choose a word processor, it's necessary that the document you create be saved as a text file before you attempt to use it with the model processor. Most word processors have this capability. Use the *Save As* option and put double quotes around the file name you wish to use.

Right now is a good time to try writing your first model-processor program. Since we're only concerned, for the present, with the mechanics of creating an *.asm* program file that will successfully assemble and run, here's a simple program that accepts an integer and prints its negation. Use your editor to create and save the file *negat.asm*.

```
// NEGAT.ASM: Displays negation of input value
      input  num
      load   num
      sub    num
      sub    num
      store  negat
      print  negat
      stop
num   data   0
negat data   0
```

When you have created and saved this file, make certain that the file name has the extension *.asm*. If you find that it has a different extension, such as *.cpp*, you can rename it by clicking on its directory icon twice slowly, seeing the name highlighted, and typing the new name.

When you think you have a correct version of *negat.asm* saved on your disk, go back two pages in this appendix and follow the instructions for loading and running an *.asm* file. If the process is successful, proceed with the instructions for running it. Otherwise (if an error message is displayed), note the message and line number of the faulty statement and edit *negat.asm* to correct the error. Don't be discouraged if you make a few mistakes at first. You'll shortly become familiar with all phases of the process.

Review problems

1. What component of the model computer presented in this appendix stores: (a) the result of a just-executed arithmetic computation; (b) a data value that has just been input from the keyboard; (c) the instruction currently executing; (d) the address of the next instruction to execute?
2. What type of data can the model computer process?
3. What does it mean to load a program? To load a data item?
4. If a program for the model computer consists of 12 statements, into what RAM address is the last statement loaded?
5. Show the output that will be generated when the following program is executed. Do this as a pencil-and-paper exercise.

```
  load  X
  add   Y
  store X
  print X
  sub   Z
  store X
      print     X
  stop
X data  0
Y data  1
Z data  5
```

6. Which instructions among *stop*, *load*, *store*, *add*, *sub*, *input*, *print*, *jump*, *jump0*, and *jump–* change the value in the accumulator?
7. Consider the partial picture of the model computer during program execution in Figure A- 3. In the fetch-execute cycle, the instruction *load x* has been brought into the IR and executed, the PC has been incremented to 1, and the instruction *add x* is next to be brought into the Instruction Register and executed.
   (a) When *add x* has been executed, what will be the contents of the accumulator?
   (b) What instruction will then be brought into the IR?
   (c) What will be the new value in the PC?
8. The program example, *double.asm*, early in this appendix, loads *innum* and then performs the operation *add innum.* Suppose the statement *add innum* were changed to *sub innum.* What would the output of the program be, given an input of 13?
9. Here is a program for dividing a non-negative integer by a positive integer:

```
        input DVDND
        input DVSR
LOOP    load  DVDND
        sub   DVSR
        jump- PRT
        store DVDND
        load  QUOT
        add   ONE
        store QUOT
        jump  LOOP
PRT     print QUOT
        print DVDND
        stop
ONE     data  1
QUOT    data  0
DVDND   data  0
DVSR    data  0
```

   Suppose the input dividend (DVDND) is –100 and the input divisor (DVSR) is –25. What will the output be?
10. Which control structures require jump instructions in machine language? Which use backward jumps? Which use forward jumps?

Criticize the code described or listed below. What is nonsensical about each example?

11. (a) a *store* instruction at the start of a program
    (b) ...

```
     store     x
     load      x
     ...
```

    (c) A *load*, *store*, *add*, *sub*, or *input* just before a *stop*.

A MODEL MICROPROCESSOR (executing TEMP.ASM)

Random Access Memory

Address	Label	Operation	Operand
0		LOAD	x
1		ADD	x
2		STORE	y
3		PRINT	x
4		PRINT	y
5		STOP	
6	x	DATA	17
7	y	DATA	85

Program Counter: 1

Instruction Register: LOAD x (6)

Accumulator: 17

Input/Output

Figure A- 3: For use with Review Problem 7

Programming exercises

For each numbered problem-solving exercise at the end of Chapters 1 and 2, specified by your instructor, write a program in the language of the model processor. Test it with the simulator, *asm*. Be sure to begin each program with a comment and to insert comments wherever necessary to clarify your intentions.