Introduction to the RAMM Computer

The RAMM computer is used as a teaching tool in the introductory computer science course at Angelo State University. Students are introduced to the concepts of the instruction cycle, program counter, instruction register, arithmetic registers, memory, stored programs, sequential processing, branching, loading, assembly, source code, object code, etc.

The RAMM computer provides the students with an integrated environment. They use an imbedded text editor to type in their machine language or their assembly language code. They can run their programs in a “trace” mode and observe changes in memory, the arithmetic registers, the program counter, and output. They can enter their data from a text file or interactively from the keyboard.

This manual describes a simulated computer named RAMM (Representative Auxiliary Machine Model) which was developed to demonstrate the various properties and techniques of programming a fixed-word-length computer. We call the RAMM computer a "simulated" computer because it does not exist as a box of electronic components with wires and connections. The RAMM computer is actually a computer program that when executed simulates the actions of a computer. The technique of simulating a computer with a computer program is very useful in studying the characteristics and properties of a computer without having to go to the expense of actually building one. As we learn about the RAMM computer, it will be as real to us as if it were a computer built of electronic "hardware."

Fred C. Homeyer originally designed the RAMM computer simulation in 1971 at Angelo State University in San Angelo, Texas. At various times in its life it has been implemented in FORTRAN, BASIC, COBOL, Pascal and most recently in C++. The current version runs under MS Windows 3.11/95/98/NT.

The first RAMM computer was a program written in Basic FORTRAN IV. This program was executed on an IBM 360 Model 25 computer system. The FORTRAN program that constituted the RAMM computer could be run or executed on any mainframe computer that could execute Basic FORTRAN IV programs. The original input medium for the RAMM computer was punched cards and the output medium was the line printer. The second mainframe version of RAMM could process input from either a video display terminal or punched cards while the output medium could be either video display terminal or line printer or both.

In 1984 a new version of the RAMM computer was developed to run on the TRS80 Model 4 microcomputer. This new simulation program was written in the BASIC programming language. The version of RAMM that executed on that microcomputer had input from the microcomputer keyboard and output on the microcomputer display screen and any printer that might be attached to the microcomputer. The microcomputer version of RAMM was available for both the TRS80 Model 4 microcomputer and the IBM PC microcomputer.

In 1995 another version of the RAMM computer was developed to run on an IBM PC compatible system with Microsoft Windows. Programming for the RAMM computer can be accomplished using RAMM machine language or RAMM assembly language and both languages are discussed here.

Structure and Characteristics of the RAMM Computer

The RAMM computer is a 100-word (4 decimal digits per word) machine with one-address logic. By a one-address logic machine we mean that only one operand is specified per computer instruction. We can think of the memory of the RAMM computer as a 100-element array where each array element contains a number consisting of exactly four decimal digits. Each word in the memory of the RAMM computer is designated by a unique numeric address. The first word of memory in the RAMM computer has an address of *00* while the last word has the address *99*. The range of addresses in the RAMM computer extends from *00* to *99* and includes all two-digit numbers between these numbers. We could represent the memory of the RAMM computer as in the diagram below:

Address	Memory
00	xxxx
01	xxxx
02	xxxx
03	xxxx
...	...
97	xxxx
98	xxxx
99	xxxx

A good definition of memory ***address*** might be a "unique number denoting a memory location to which data may be sent to be stored and from which it may be retrieved." A computer ***word*** is a set of characters (in this case decimal digits) treated as a unit by the computer. So we see that the RAMM computer has a word consisting of four decimal digits. Since all information in a computer is normally in a two-state form we can think of the information in the RAMM computer memory as being in binary-coded decimal notation. Note that binary-coded decimal notation is NOT the same as number representation in pure binary notation. Each word of memory in the RAMM computer has a unique address, that is, when we specify a particular memory word by giving its numeric address, that numeric address belongs **only** to that memory location. Another good way to think of a computer memory is to liken it to a message service in a hotel. There are normally a number of compartments, one for each room in the hotel, in the message room. We can leave information for a particular room by designating the room number or we can retrieve information for a particular room by giving the room number. No two rooms have the same number.

The basic language of any computer is called ***machine language***. Machine language is the only language to which the computer can respond directly. All programs written in language other than machine language must be translated into the machine language of the computer before execution of the program can occur. There are many different machine languages as each computer (or family of computers) has its own specified set of instructions that it can execute. The set of instructions that a computer can execute is determined when the computer is designed and built. The machine language of the computer is numerical and represented in the memory of the computer in two-state form. Each memory word can contain a computer instruction or a data value. One of the most powerful properties of programming in the machine language of a computer is the ability to treat the contents of a memory location at one time as a computer instruction and at another time as a data value. In this way the program being executed on the computer can dynamically change itself during the execution of the program.

Even though the RAMM computer is a "software" computer program that simulates the actions of a computer, it exhibits properties of the hardware of a real computer. For our purposes we may think of the RAMM computer as if it really exists with switches, lights, etc. RAMM is a decimal, sequential computer using BCD (binary-coded decimal) representation. By ***sequential computer*** we mean that instructions are executed from sequential or consecutive memory locations unless a transfer of control instruction is specified.

All arithmetic is performed in RAMM through the use of two arithmetic registers called the A-register and the Q-register. Each of the two registers can contain a number consisting of four decimal digits. The Q-register is used in conjunction with the A-register to perform multiplication and division. The descriptions of the various instructions available in the RAMM computer are presented in a subsequent chapter. The range of values that can be stored in a memory location of the RAMM computer is *-9999* to *+9999*. The range of values that can be read from a data file or from the keyboard is -999 to +9999. This discrepancy in the range of values which can be *computed* versus the range of values which can be *input* stems from the historical limits of the punched card. Data values may be placed in the memory of the computer through a program storing operation or through the execution of a "read data" instruction in a RAMM computer program.

RAMM Instruction Descriptions and Examples

The instruction format for a RAMM machine language instruction is a two-digit operation code and a two-digit operand address. The instruction format may be represented as *XXyy* where *XX* is the operation code (op code) and *yy* is the operand address. In the instruction definitions given below, both the mnemonic operation code (used in RAMM Assembler language programming) and the numeric operation code (used in RAMM machine language) are given. Each machine language instruction will be listed as *XXyy* where *XX* is the numeric operation code and *yy* represents the operand.

Notation and Terminology used in Instruction Definitions

Enclosing the name of the location or register within parentheses indicates the ***contents*** of a location or register. For example, if we wish to designate that the contents of memory location 14 is 0345 we would write this as follows: (14) = 0345. If we wish to denote that the contents of the A-register is 7777 we would write (A-register) = 7777.

The ***current instruction*** is the instruction that is being executed at a particular instant under consideration.

The ***current location*** is the memory location in which the current instruction is stored.

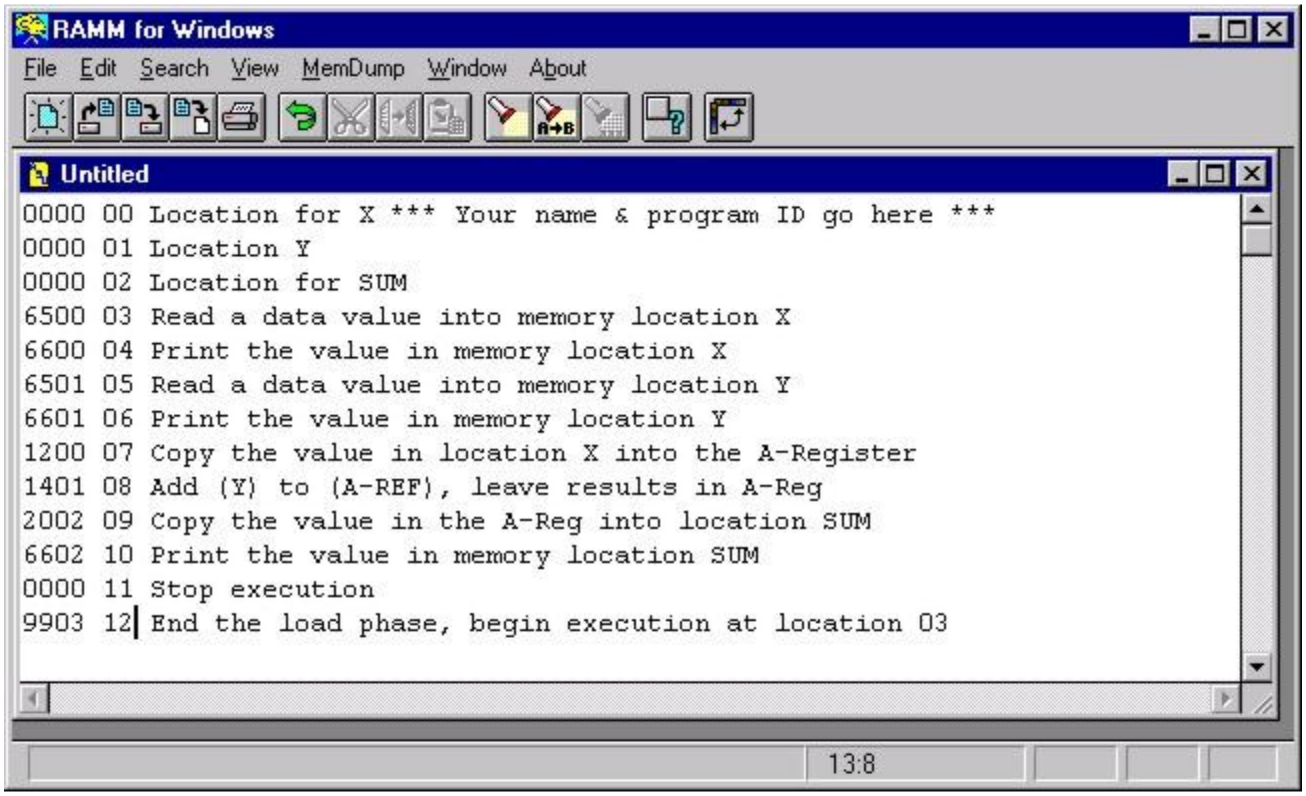
The ***current sequence of control*** is that order of execution in which, when the current instruction has finished executing, the next instruction to be executed is the one stored in the next consecutive memory location.

Data Value Format

Data to be processed during execution of a RAMM program can be input one value at a time with the ENTER key pressed after inputting each data value. Negative numbers must be prefixed by a minus sign but positive numbers do not have to be prefixed by a + sign. All nonnegative values input are in the form of four decimal digits with no sign. A negative value is entered with a minus sign followed by exactly three decimal digits. If a data file is used, the input values are typed one value per line of a simple text file (ASCII file) in character positions 1 through 4 of the line. The "largest" (greatest magnitude) negative number that can be input as a data value is -999 because the RAMM computer only looks at columns 1 through 4 of the data record.

Loading and Executing a RAMM Machine Language Program

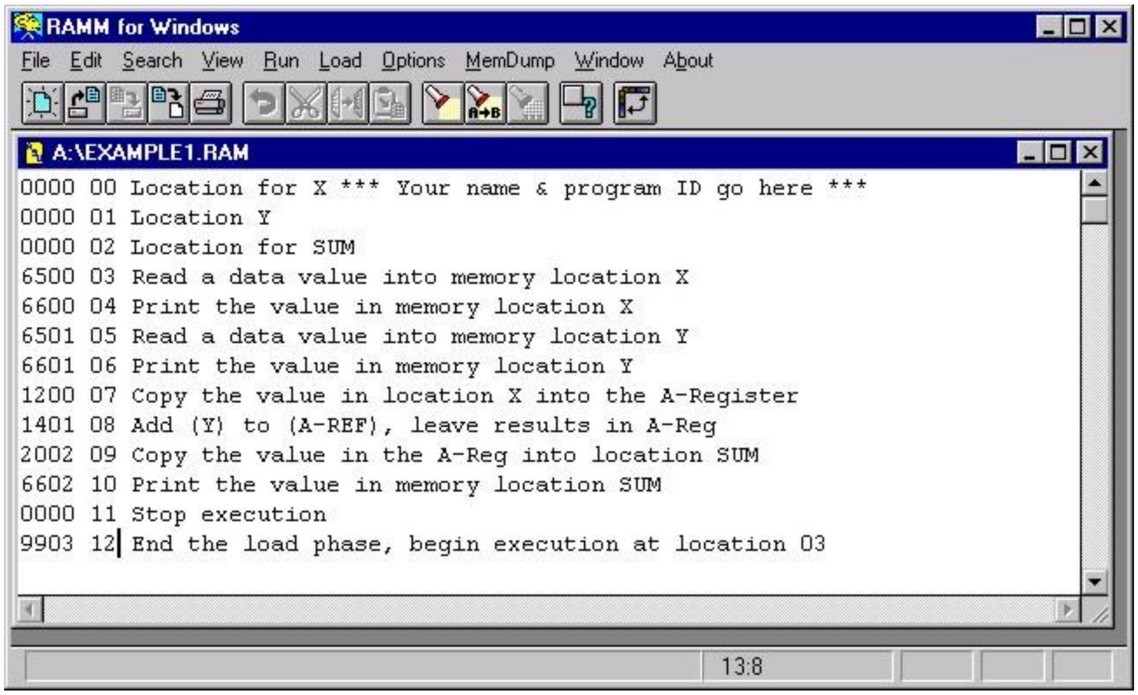
A RAMM program begins with the first instruction loaded into memory location *00* and ends with a *99* (END) instruction. The *99* or END instruction must be the last instruction in the program. To load a RAMM machine language program you should activate the RAMM Computer from the *Start/Programs* menu on the campus network. *Choose File/New* to pull up an *Untitled* text window. Type in the RAMM machine language program, beginning each instruction in the first column of a new line (Figure 1). Save the file to your disk with *File/Save/**labname**.RAM* or *File/Save As…/labname.RAM*.



```
0000 00 Location for X *** Your name & program ID go here ***
0000 01 Location Y
0000 02 Location for SUM
6500 03 Read a data value into memory location X
6600 04 Print the value in memory location X
6501 05 Read a data value into memory location Y
6601 06 Print the value in memory location Y
1200 07 Copy the value in location X into the A-Register
1401 08 Add (Y) to (A-REF), leave results in A-Reg
2002 09 Copy the value in the A-Reg into location SUM
6602 10 Print the value in memory location SUM
0000 11 Stop execution
9903 12 End the load phase, begin execution at location 03
```

Figure 1

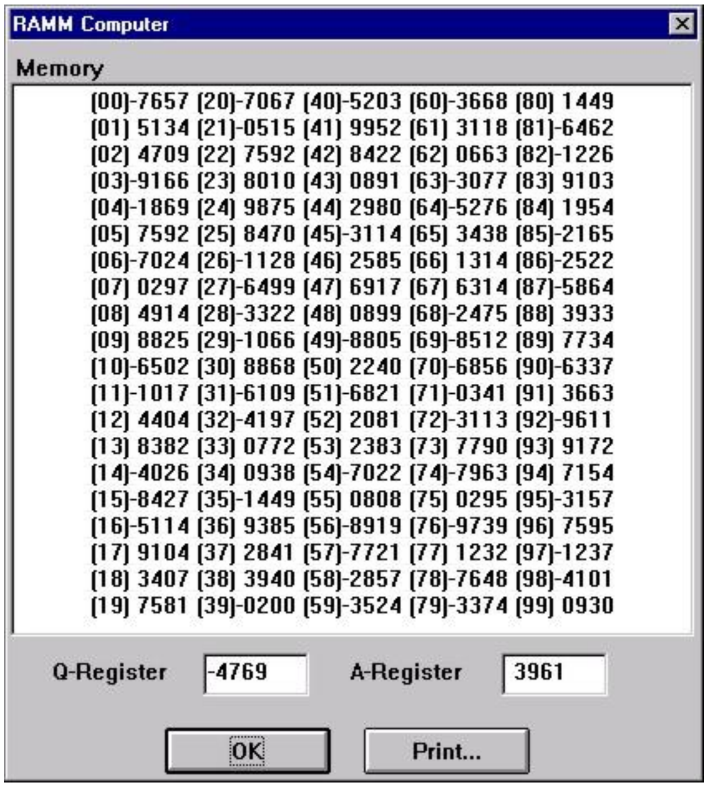
Once the *Untitled* bar has changed to a file name with the extension *RAM* the *Menu* options at the top now offer the choices *Run*, *Load* and *Options* in addition to others offered before the file in the active window was identified as a ".*RAM*" file (Figure 2).



```
0000 00 Location for X *** Your name & program ID go here ***
0000 01 Location Y
0000 02 Location for SUM
6500 03 Read a data value into memory location X
6600 04 Print the value in memory location X
6501 05 Read a data value into memory location Y
6601 06 Print the value in memory location Y
1200 07 Copy the value in location X into the A-Register
1401 08 Add (Y) to (A-REF), leave results in A-Reg
2002 09 Copy the value in the A-Reg into location SUM
6602 10 Print the value in memory location SUM
0000 11 Stop execution
9903 12 End the load phase, begin execution at location 03
```

Figure 2

Select *Memory Dump* at the top of the window to see that the RAMM Computer memory contains random values, not the instructions given in your program, *Example1.RAM* (Figure 3).

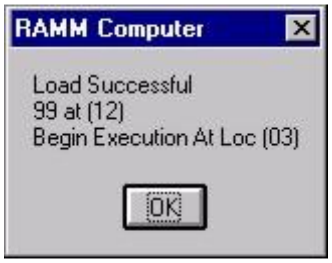


```
[00]-7657 [20]-7067 [40]-5203 [60]-3668 [80] 1449
[01] 5134 [21]-0515 [41] 9952 [61] 3118 [81]-6462
[02] 4709 [22] 7592 [42] 8422 [62] 0663 [82]-1226
[03]-9166 [23] 8010 [43] 0891 [63]-3077 [83] 9103
[04]-1869 [24] 9875 [44] 2980 [64]-5276 [84] 1954
[05] 7592 [25] 8470 [45]-3114 [65] 3438 [85]-2165
[06]-7024 [26]-1128 [46] 2585 [66] 1314 [86]-2522
[07] 0297 [27]-6499 [47] 6917 [67] 6314 [87]-5864
[08] 4914 [28]-3322 [48] 0899 [68]-2475 [88] 3933
[09] 8825 [29]-1066 [49]-8805 [69]-8512 [89] 7734
[10]-6502 [30] 8868 [50] 2240 [70]-6856 [90]-6337
[11]-1017 [31]-6109 [51]-6821 [71]-0341 [91] 3663
[12] 4404 [32]-4197 [52] 2081 [72]-3113 [92]-9611
[13] 8382 [33] 0772 [53] 2383 [73] 7790 [93] 9172
[14]-4026 [34] 0938 [54]-7022 [74]-7963 [94] 7154
[15]-8427 [35]-1449 [55] 0808 [75] 0295 [95]-3157
[16]-5114 [36] 9385 [56]-8919 [76]-9739 [96] 7595
[17] 9104 [37] 2841 [57]-7721 [77] 1232 [97]-1237
[18] 3407 [38] 3940 [58]-2857 [78]-7648 [98]-4101
[19] 7581 [39]-0200 [59]-3524 [79]-3374 [99] 0930

Q-Register -4769        A-Register 3961
```

Figure 3

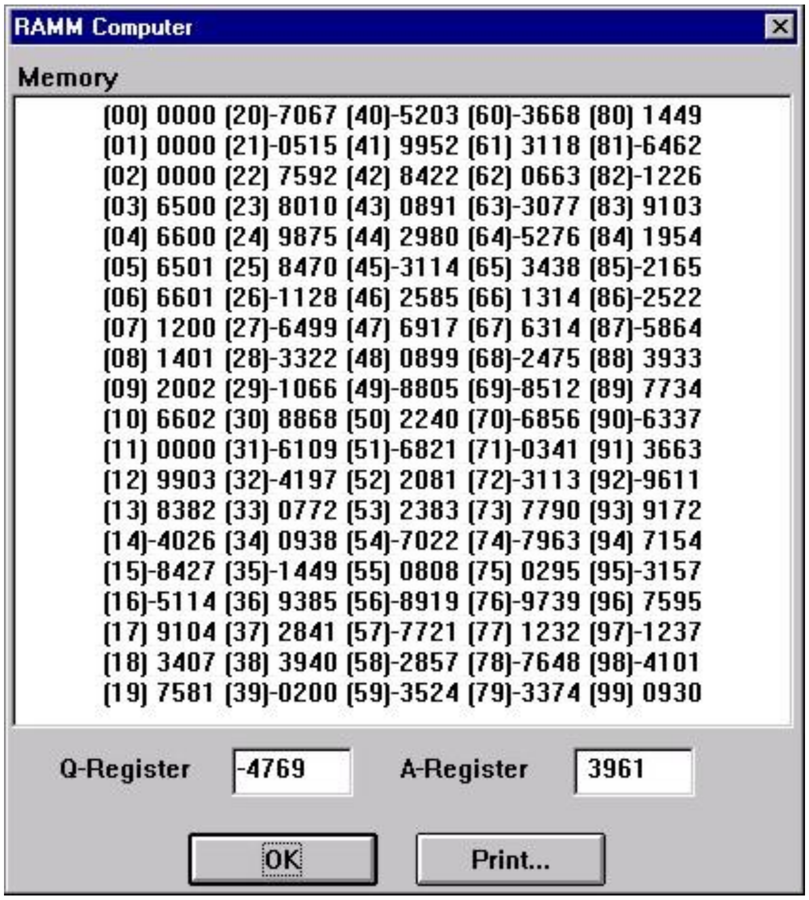
Now select *Load* at the top of the window to cause your instructions to be stored in the RAMM Computer memory. Note that the "*Load Successful*" message (Figure 4) gives other important information. It indicates that at location *12* of the program a *99* operation code was found and loaded. The operand of that instruction indicated that execution of the program should begin at memory location *03*.



```
Load Successful
99 at (12)
Begin Execution At Loc (03)
```

Figure 4

If you select *Memory Dump* after loading your program *A:\Example1.RAM* you will see that your program has been placed in the RAMM Computer Memory, beginning at location *00* and ending at location *12* (Figure 5).



```
(00) 0000 (20)-7067 (40)-5203 (60)-3668 (80) 1449
(01) 0000 (21)-0515 (41) 9952 (61) 3118 (81)-6462
(02) 0000 (22) 7592 (42) 8422 (62) 0663 (82)-1226
(03) 6500 (23) 8010 (43) 0891 (63)-3077 (83) 9103
(04) 6600 (24) 9875 (44) 2980 (64)-5276 (84) 1954
(05) 6501 (25) 8470 (45)-3114 (65) 3438 (85)-2165
(06) 6601 (26)-1128 (46) 2585 (66) 1314 (86)-2522
(07) 1200 (27)-6499 (47) 6917 (67) 6314 (87)-5864
(08) 1401 (28)-3322 (48) 0899 (68)-2475 (88) 3933
(09) 2002 (29)-1066 (49)-8805 (69)-8512 (89) 7734
(10) 6602 (30) 8868 (50) 2240 (70)-6856 (90)-6337
(11) 0000 (31)-6109 (51)-6821 (71)-0341 (91) 3663
(12) 9903 (32)-4197 (52) 2081 (72)-3113 (92)-9611
(13) 8382 (33) 0772 (53) 2383 (73) 7790 (93) 9172
(14)-4026 (34) 0938 (54)-7022 (74)-7963 (94) 7154
(15)-8427 (35)-1449 (55) 0808 (75) 0295 (95)-3157
(16)-5114 (36) 9385 (56)-8919 (76)-9739 (96) 7595
(17) 9104 (37) 2841 (57)-7721 (77) 1232 (97)-1237
(18) 3407 (38) 3940 (58)-2857 (78)-7648 (98)-4101
(19) 7581 (39)-0200 (59)-3524 (79)-3374 (99) 0930

Q-Register -4769     A-Register 3961
```

Figure 5

Choosing *Run* will cause your program to be executed. If you have one or more *READ* (*65yy*) instructions in your program, a dialogue box should appear asking you to enter a data value to be used for each *READ* instruction encountered (Figure 6). I have typed in the data value *0527* in this case.

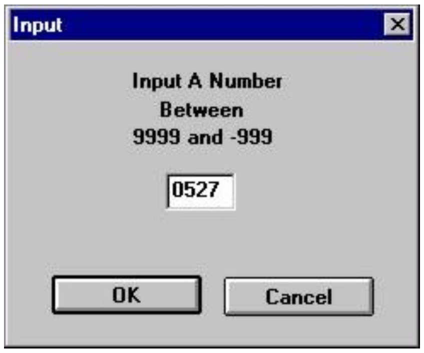


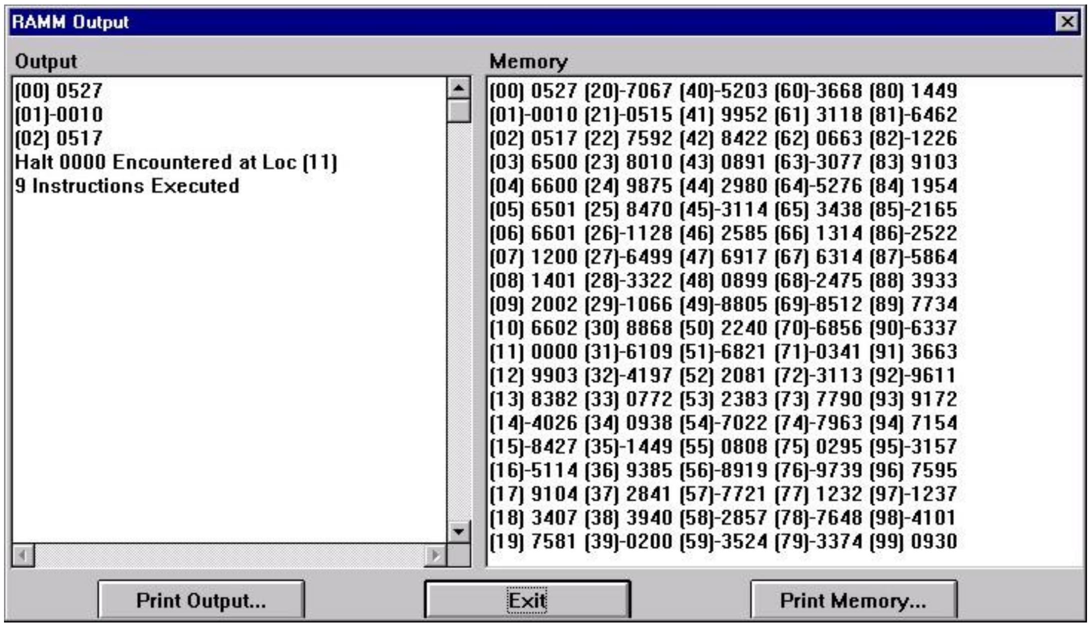
Figure 6

When asked for the second number, I typed in the data value *-010*. The expected result should be *0517* because my *Example1.RAM* program was to add the two values read in and then print their sum.

Note that the *Output* (Figure 7) window shows the values printed out with their corresponding memory locations given in parentheses. It also indicates the instruction that caused execution to stop. A *00* operation code is a legal halt. Any other two-digit value that is not a legitimate RAMM op code would have been treated as an illegal halt. In either case, the memory location containing the instruction causing the halt is given. The number of instructions executed is also given. This value can be greater than the total number of instructions in the program and even greater than the total number of locations in memory because loops can reuse instructions.

Note that your program has changed the first three locations in memory. They were set up to hold variable values that did indeed vary. Your instructions did not, however, change. Only the first *13* locations (*00-12*) were of interest to us in this program. The other memory locations continue to have "garbage" in them.

Printing the *Output Window* or the *Memory Contents* after running your program can be accomplished by clicking on the *Print* button for either choice.



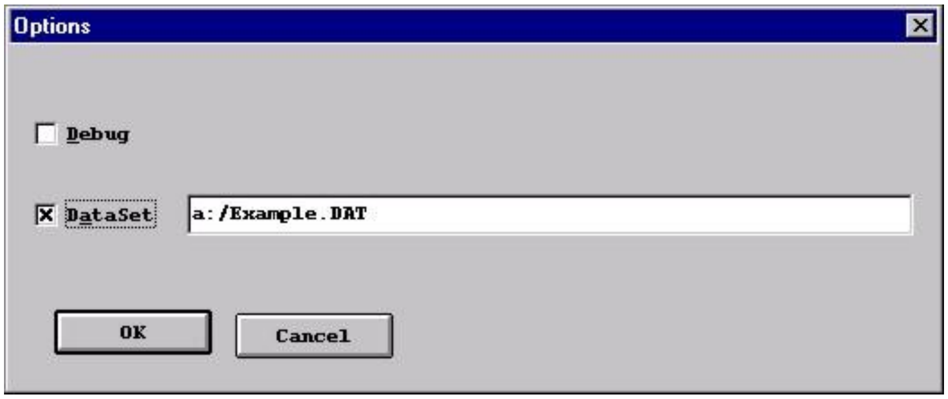
```
(00) 0527
(01)-0010
(02) 0517
Halt 0000 Encountered at Loc (11)
9 Instructions Executed
```

```
(00) 0527 (20)-7067 (40)-5203 (60)-3668 (80) 1449
(01)-0010 (21)-0515 (41) 9952 (61) 3118 (81)-6462
(02) 0517 (22) 7592 (42) 8422 (62) 0663 (82)-1226
(03) 6500 (23) 8010 (43) 0891 (63)-3077 (83) 9103
(04) 6600 (24) 9875 (44) 2980 (64)-5276 (84) 1954
(05) 6501 (25) 8470 (45)-3114 (65) 3438 (85)-2165
(06) 6601 (26)-1128 (46) 2585 (66) 1314 (86)-2522
(07) 1200 (27)-6499 (47) 6917 (67) 6314 (87)-5864
(08) 1401 (28)-3322 (48) 0899 (68)-2475 (88) 3933
(09) 2002 (29)-1066 (49)-8805 (69)-8512 (89) 7734
(10) 6602 (30) 8868 (50) 2240 (70)-6856 (90)-6337
(11) 0000 (31)-6109 (51)-6821 (71)-0341 (91) 3663
(12) 9903 (32)-4197 (52) 2081 (72)-3113 (92)-9611
(13) 8382 (33) 0772 (53) 2383 (73) 7790 (93) 9172
(14)-4026 (34) 0938 (54)-7022 (74)-7963 (94) 7154
(15)-8427 (35)-1449 (55) 0808 (75) 0295 (95)-3157
(16)-5114 (36) 9385 (56)-8919 (76)-9739 (96) 7595
(17) 9104 (37) 2841 (57)-7721 (77) 1232 (97)-1237
(18) 3407 (38) 3940 (58)-2857 (78)-7648 (98)-4101
(19) 7581 (39)-0200 (59)-3524 (79)-3374 (99) 0930
```

Figure 7

Creating a Data File

If there are many data values to be read by a particular program it may be desirable to create a data file containing those values and to indicate that data file in the *Options Window* (Figure 8). Choose *Options* from the list at the top of the RAMM Computer Window. Click on the box beside *DataSet*. Then type in the name of the data file you wish to use for the program that is to be executed next.

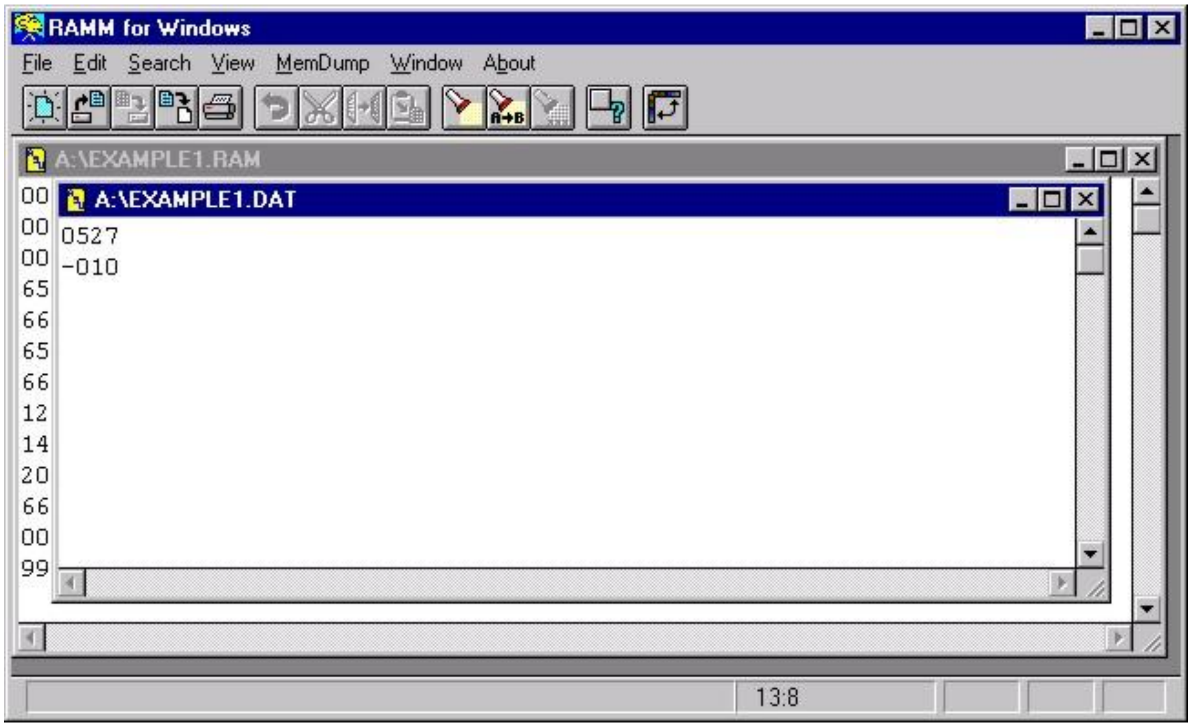


☐ Debug

☒ DataSet a:/Example.DAT

Figure 8

To create a data file, choose *File/New*. Type in the data values, one per line, beginning each line in column 1 and using only the first four columns. No blanks are allowed. Use leading zeroes as needed to enter values between -999 and 9999, inclusive. (Figure 9) Save the file, explicitly typing in an extension of *DAT*. Now you are ready to click on the window containing your RAMM program to activate it or to open the *.RAM* file containing your program. When you run your program, each read instruction encountered will cause one value to be brought in from the designated data file.



```
0527
-010
```

Figure 9

Index of RAMM Instructions

00 *yy*	`HLT`	Halt	
12 *yy*	`LDA`	Load A register	(A reg) = (*yy*)
14 *yy*	`IAD`	Integer Add	(A reg) = (A reg) + (*yy*)
15 *yy*	`ISB`	Integer Subtract	(A reg) = (A reg) - (*yy*)
16 *yy*	`LDQ`	Load Q register	(Q reg) = (*yy*)
20 *yy*	`STA`	Store A register	(*yy*) = (A reg)
21 *yy*	`STQ`	Store Q register	(*yy*) = (Q reg)
24 *yy*	`IMU`	Integer Multiply	(A reg) = least significant 4 digits of ((A reg) * (*yy*)) (Q reg) = most significant 4 digits of ((A reg) * (*yy*))
25 *yy*	`IDV`	Integer Divide	(A reg) = int quotient of ((A reg) / (*yy*)) (Q reg) = int remainder of ((A reg) / (*yy*))
36 *yy*	`AZJ`	Jump to location *yy* if (A reg) is zero	
37 *yy*	`AMJ`	Jump to location *yy* if (A reg) is negative	
50 *yy*	`NOP`	No Operation	
65 *yy*	`RDI`	Read Decimal Integer	(*yy*) = value input from keyboard or input file
66 *yy*	`PRI`	Print Decimal Integer	Print (*yy*) as an integer on output line
75 *yy*	`UNJ`	Unconditional Jump to location *yy*	
99 *yy*	`END`	End of program, begin execution at *yy*	
	`BSS k`	Reserve *k* storage locations (k≥0)	
	`DEC k`	Create decimal constant of *k* (-999≤k≤9899)	

Description of RAMM Instructions

00*yy* HLT Halt

This instruction causes execution of the program to terminate. The operand *yy* is ignored in execution.

Example: Suppose that we have the following instruction sequence and execution begins at memory location 15.

(15) =1204
(16) =1407
(17) =2003
(18) =0000

After loading the A-register with the contents of memory location 04 and adding to that the value of the contents of memory location 07 and storing the sum that is left in the A-register into memory location 03, program execution terminates. When a **HLT** instruction is encountered in program execution the computer prints a message similar to the one listed below:

```
Halt 0000 Encountered at Location (18)
      4 Instructions Executed
```

12*yy* LDA Load A-register

This instruction replaces the contents of the A-register with a copy of the contents of memory location *yy*. The initial contents of the A-register are destroyed and the contents of *yy* are unchanged.

Initial values are:

(A-register) =	2356
(05) =	0123

Example: Suppose that the following instruction is given:

1205

Execution of this instruction causes the following registers and memory locations to be affected:

(A-register) =	0123
(05) =	0123

14*yy* IAD Integer Add

This instruction adds a copy of the contents of memory location *yy* to the contents of the A-register and leaves the sum in the A-register. The initial contents of the A-register are destroyed and the contents of memory location *yy* are unchanged. If overflow occurs, an error message will be output and execution of the program will continue. Note that prior to issuing the **IAD** instruction, one of the addends should already be in the A-register. This can be accomplished by a load instruction or possibly as the result of another arithmetic operation.

Initial values are:

```
(A-register) =2356
(03) =        0005
(04) =        0012
```

Example: Suppose that the following instruction sequence is given:

```
1203
1404
```

After execution of this instruction sequence, the final contents of the A-register and memory locations involved would be as follows:

```
(A-register) =0017
(03) =        0005
(04) =        0012
```

Note that the **LDA** instruction (1203) loaded the first addend into the A-register. After execution of the **LDA** instruction, the A-register contained 0005. The **IAD** instruction then caused the 0005 contained in the A-register to be added to the 0012 that was contained in memory location 04 and the sum of the two values, namely 0017, was placed in the A-register.

It is possible for a situation called ***OVERFLOW*** to occur after execution of an addition or subtraction instruction. Overflow occurs when a sum or difference of two 4-digit numbers cannot be contained in the four-digit A-register. For example, if the numbers 9999 and 0001 were added together, the sum would be 10000. Obviously, you can't put a 5-digit number into a 4-digit register. The computer will issue a message in the output window on the screen that overflow has occurred at the memory location that contained the **IAD** instruction. The contents of the A-register would be 0000 (sum *mod* 10^4). Execution of the program will continue with the next instruction in the program. Note that the overflow condition can occur in execution of both **IAD** and **ISB** instructions.

15*yy* ISB Integer Subtract

This instruction subtracts a copy of the contents of memory location *yy* from the contents of the A-register and leaves the difference in the A-register. The initial contents of the A-register are destroyed and the contents of *yy* are unchanged. If overflow occurs, an error message will be output on the screen and in the output window and execution will continue. Normally one of the numbers to be used in the **ISB** instruction (the minuend) will be placed in the A-register prior to issuing the subtract instruction.

Initial contents are:

```
    (A-register) =0017
    (04) =        0012
    (05) =        0123
```

Example: Suppose that the following instruction sequence is given:

```
        1205
        1504
```

After execution of this instruction sequence, the register and memory locations affected would be as follows:

```
    (A-register) =0111
    (04) =        0012
    (05) =        0123
```

In the execution of this instruction sequence, the **LDA** instruction would have caused the A-register to contain a copy of the contents of memory location 05, namely 0123. The **ISB** instruction caused the contents of location 04, namely 0012, to be subtracted from the value in the A-register and the difference of the two values is placed in the A-register. In other words, this instruction sequence performed the subtraction of 0012 from 0123 and generated the difference, 0111. An overflow condition can be caused by subtracting two negative numbers that would generate a negative value of magnitude greater than -9999. For example, - 9999 - 1 = - 10000. The computer would give an overflow message in the output window and on the screen and execution would continue with the next instruction in the program. Also, the contents of the A-register would be 0000 (-10000 *mod* 10^4).

16*yy* LDQ Load Q-register

This instruction replaces the contents of the Q-register with a copy of the contents of memory location *yy*. The initial contents of the Q-register are destroyed and the contents of location *yy* are unchanged.

Initial contents are:

```
(Q-register) =1789
(03) =        0005
```

Example: Suppose that the following instruction is given:

```
1603
```

After execution of this instruction, the contents of the affected register and memory location would be as follows:

```
(Q-register) =0005
(03) =        0005
```

20*yy* STA Store A-register

This instruction replaces the contents of memory location *yy* with a copy of the contents of the A-register. The initial contents of *yy* are destroyed and the contents of the A-register are unchanged.

Initial contents are:

```
(A-register) =2356
(07) =        0003
```

Example: Suppose that the following instruction is given:

```
2007
```

After execution of the **STA** instruction, the contents of the affected register and memory location would be as follows:

```
(A-register) =2356
(07) =        2356
```

The **STA** instruction merely takes a copy of the contents of the A-register and places it in the memory location whose address is 07.

21*yy* STQ Store Q-register

This instruction replaces the contents of memory location *yy* with a copy of the contents of the Q-register. The initial contents of *yy* are destroyed and the contents of the Q-register are unchanged.

Initial contents are:

```
(Q-register) =1789
(05) =        0123
```

Example: Suppose that the following instruction is given:

```
2105
```

After execution of this instruction the final contents of the register and memory location affected would be as follows:

```
(Q-register) =1789
(05) =        1789
```

24*yy* IMU Integer Multiply

This instruction multiplies two 4-digit integers, forming an eight-digit product. The multiplicand must be in the A-register before execution of the **IMU** instruction. The multiplier is a copy of the contents at memory location *yy*. The product is contained in the combined QA-registers as an 8-digit integer. The least significant 4 digits of the product (low order digits) are contained in the A-register and the most significant 4 digits are contained in the Q-register. The initial contents of both the A- and Q-registers are destroyed. The contents of *yy* are unchanged.

Initial contents are:

```
(A-register) =2356
(Q-register) =1789
(04) =        0012
(05) =        0123
```

Example: Suppose that the following instruction sequence is given:

```
1204
2405
```

After execution of the instruction sequence, the memory location and registers affected would be as follows:

```
(Q-register) =0000
(A-register) =1476
(04) =        0012
(05) =        0123
```

Note that in this example all of the significant digits of the product can be contained in the A-register and thus the Q-register contains all zeroes. In practice, we normally know the range of products the values will have and if the significant digits of the product are four digits or less we only save the contents of the A-register after a multiply operation. Overflow cannot occur in a multiplication operation in RAMM since the largest product that could be generated by two 4-digit numbers would be 9999 x 9999 = 99980001 and this product is represented by only 8 digits. If the **IMU** instruction were given both multiplier and multiplicand of 9999 the product would be placed in the registers as follows:

```
(Q-register) =9998
(A-register) =0001
```

25*yy* IDV Integer Divide

This instruction divides a 4-digit integer by a 4-digit integer producing a 4-digit integer quotient and a 4-digit integer remainder. The 4-digit integer to be divided (the dividend) must be loaded into the A-register before the execution of the divide instruction. The divisor is a copy of the contents of memory location *yy*. After execution of the **IDV** instruction the A-register contains the integer quotient and the Q-register contains the integer remainder.

Initial contents are:

```
(A-register) =2356
(Q-register) =1789
(03) =        0005
(05) =        0123
```

Example: Suppose that the following instruction sequence is given:

```
1205
2503
```

After execution of the instruction sequence the memory location and registers involved would be as follows:

```
(A-register) =0024
(Q-register) =0003
(03) =        0005
(05) =        0123
```

Note that the quotient in the A-register is the integer quotient. We normally save only the A-register after a divide operation. We call this quotient a *truncated* quotient since the fractional portion of the quotient is truncated or left off the answer. We could generate a rounded quotient if we were to examine the contents of the Q-register and if these contents were greater than or equal to one-half the divisor we add 1 to the integer quotient in the A-register. In both cases an error is introduced that may gradually get worse if we continue to use this quotient in subsequent arithmetic operations. The error that is introduced as a result of saving truncated quotients is a fraction expressed as (divisor - 1) / divisor. For example if the divisor were 5000 then the error that could be introduced could be as large as 4999 / 5000 or almost 1. Each time we use this truncated quotient in another **IDV** operation we get another truncated quotient that has an error of almost 1. If we performed divide operations on ten successive quotients and used the truncated quotient each time in the next division, the final result could have an error as large as 10.

Overflows are impossible with the integer divide operation. A 4-digit dividend and a 4-digit divisor can generate a quotient with a maximum of 4 digits and a remainder with a maximum of 4 digits.

36*yy* AZJ Jump if A-register is Zero

If the A-register contains zero, the current sequence of control is terminated and a new sequence is begun at location *yy*. The next instruction to be executed is contained in memory location *yy*. If the A-register contains a non-zero number, the current sequence of control continues. The contents of all memory locations and the A- and Q-registers are unchanged by this instruction. This is one of the two instructions in RAMM called conditional transfer instructions. A branch is made on the condition that the A-register contains zero at that point in the program. The mnemonic **AZJ** could represent "A-Register is Zero, Jump."

Example: Suppose that the contents of the A-register are zero and the following instruction sequence with memory addresses as listed appears in a program:

```
        .
(45) =3650
(46) =1204
        .
        .
        .
(50) =1207
(51) =1503
        .
        .
```

If we pick up program execution at memory location 45 in this example, since the contents of the A-register are zero, a transfer of control is made to memory location 50 for the next instruction. A new sequence of control is begun at location 50. If the contents of the A-register had not been zero when the **AZJ** instruction was encountered then the next instruction executed after location 45 would have been the instruction at location 46.

37*yy* AMJ Jump if A-register is negative (minus)

If the A-register contains a negative number, the current sequence of control is terminated and a new sequence is begun at location *yy*. The next instruction to be executed is the contents of memory location *yy*. If the A-register contains a positive number or zero, the current sequence of control continues. The contents of all memory locations and the A- and Q-registers are unchanged by this instruction. The mnemonic **AMJ** stands for "A-Register Minus, Jump." The **AMJ** instruction is one of two conditional branch instructions in RAMM.

Using the **AZJ** and **AMJ** instructions together allows you to implement three branches of a decision symbol in a flowchart. Compare value A to value B by computing A - B, leaving the result in the A-Register.

A - B = 0	implies A = B
A - B = negative value	implies A < B
A - B = positive value	implies A > B

A combination of **AZJ** and **AMJ** can check the case of a positive result since if the result is not zero or negative, then it must be positive.

Example: Suppose that the following instruction sequence is given with memory locations as indicated and suppose that the A-register currently contains a negative number.

```
        . . .
(60) =3772
(61) =1204
        . . .
(72) =1205
(73) =1403
        . . .
```

If we pick up the program execution at memory location 60, then the next instruction to be executed is at memory location 72. A new sequence of control is begun at 72. If the A-register had not contained a negative value at this point in the program then the next instruction executed after the **AMJ** instruction would have been at memory location 61.

Example: Compare two values and make a branch based upon this comparison. Suppose that we have the following memory contents and instructions listed below:

```
        .
(10) =0004
(11) =0007
        . . .
(20) =1211
(21) =1510
(22) =3730
(23) =3640
(24) =1205
        . . .
```

Suppose that we pick up program execution at location 20. Here we are comparing the contents of location 10 to the contents of location 11. The result in the A-register after execution of the **ISB** instruction at location 21 would be 0003. Since this is not a negative value the branch specified by the **AMJ** instruction at location 22 would not be taken and since the A-register is not zero the branch specified by the **AZJ** instruction at location 23 is also not taken. Thus the next instruction executed after this sequence of steps is the instruction at memory location 24. This is an example of using **AMJ** and **AZJ** instructions together to get the net effect of "A-Register Positive, Jump" function. We don't need an **APJ** because we use **AMJ** and **AZJ** together to achieve the same result.

50*yy* NOP Pass - No Operation

This instruction causes no operation to be performed. The current sequence of control continues. The operand *yy* is ignored during the execution of the instruction. The purpose of the **NOP** is to allow the programmer to place one or more "do nothing" instructions in a sequence of program code. In subsequent revisions of the program, the programmer may wish to replace these **NOP** instructions with other code. The "do nothing" instructions may also be used as "place holders" for instructions to be computed or modified during execution. The ability of the programmer to cause the program to dynamically change itself during execution is one of the most powerful techniques available in machine language programming.**Example**: Suppose the following code segment is given at the memory locations indicated and that execution begins at memory location 33.

```
        . . .
(33) =1204
(34) =1405
(35) =2007
(36) =5000
(37) =1205
        . . .
```

This code caused the contents of memory location 04 to be added to the contents of memory location 05 and the sum stored in memory location 07. Execution of the **NOP** instruction at location 36 causes no change in the registers or memory locations and the sequence of control would pass on to memory location 37.

65*yy* RDI Read Decimal Integer

This instruction reads a decimal integer from the input stream (either a data file or value input through the keyboard) and stores the value in the memory location whose address is *yy*. The initial contents of *yy* are destroyed and **the contents of the A- and Q-registers are unchanged**. The **RDI** instruction allows us to input values in an executing program. The value read in replaces the value currently stored in memory location *yy*.

Example: Suppose that the following memory contents are given at the memory locations indicated:

```
        .
(10) =0019
        .
        .
        .
(35) =6510
        .
        .
```

If we pick up program execution at location 35, the **RDI** instruction will request that you input a value by displaying a dialog box on the screen with the message: "Input a value." You should input a data value in the range -999 to 9999. If you input the value 0345 through the keyboard, then after execution of the **RDI** instruction, the contents of memory location 10 would be 0345.

66*yy* PRI Print Decimal Integer

This instruction prints on the next output line the integer stored in memory location *yy*. The contents of *yy* and the A- and Q-registers are unchanged. The **PRI** instruction produces output by listing the address of the memory location to be printed out and then the contents of that memory location, for example "(96) = 0012" would be generated by the instruction 6696 if memory location 96 contained the value 0012. The **PRI** instruction is the only means to output values from a RAMM program. Note that the only values that can be printed are 4-digit decimal integers.

Example: Suppose that the following code sequence is given with memory contents as indicated:

```
        .
(23) =1234
        .
        .
        .
(27) =9988
        .
        .
(35) =6623
(36) =6627
```

If we pick up program execution at memory location 35, the **PRI** instructions would cause the following output to be listed on the screen:

```
(23) = 1234
(27) = 9988
```

75*yy* UNJ Unconditional Jump

This instruction terminates the current sequence of control and starts a new sequence at memory location *yy*. The next instruction to be executed is the contents of *yy*. The contents of all memory locations and the A- and Q-registers are unchanged by this instruction. We also call this instruction an unconditional branch instruction. The current sequence of control in a program is that order of execution in which, when the current instruction has finished executing, the next instruction to be executed is stored in the next consecutive memory location. The **UNJ** instruction is one of three instructions that can cause a transfer of control in a program.

Example: Suppose we are given the following instruction sequences located at the memory locations indicated:

```
      .
(23) =7532
(24) =1204
      . . .
(32) =1205
(33) =1403
      . . .
```

If we pick up program execution at memory location 23 and execute the instruction contained there, then the next instruction executed after the unconditional jump instruction at location 23 will be at memory location 32 and program control will continue to location 33 and subsequent consecutive memory locations until a new transfer of control instruction is given.

99*yy* END End program loading

This instruction indicates the end of the program loading operation and also specifies that execution of the program is to begin at memory location *yy*. The **END** instruction is the last instruction of any RAMM program. Since **END** is processed during the program-loading phase, that is, the phase where the program is stored into the RAMM computer memory, it cannot be considered an executable instruction. The **HLT** instruction is used to terminate execution of the program while the **END** instruction is used to terminate loading of the program. If the **END** instruction is encountered during execution of a program, the 99 operation code causes termination of processing with an error message that an illegal operation code was encountered.

Example: Suppose that the following code sequence is given:

```
(65) =1204
(66) =1403
(67) =2005
(68) =6605
(69) =0000
(70) =9902
```

As the RAMM program is loaded into memory of the RAMM computer the instruction that is loaded into memory location 70 causes the loading operation to be terminated. Notice that the **HLT** instruction in location 69 was not recognized as a halt at the time that the program was loaded into the RAMM computer memory since the only instruction that is actually processed during the program loading phase is the 99 (**END**) instruction. After the **END** instruction is processed the program will begin execution at memory location 02 when the programmer specifies that he wishes to "run" or execute the program.

END in Assembly Language

The **END** pseudo-instruction signifies the end of a program and must be the last statement in the program. The instruction terminates the assembly (translation) process and also the loading process. If no fatal assembly errors have occurred execution will begin at memory location *yy* when the programmer specifies that he wishes to "run" or execute the program. If *yy* is omitted, execution begins at location 00. A symbol in the location field is illegal.

BSS k Reserve k storage locations

The **BSS** pseudo-instruction reserves *k* consecutive memory locations starting at the current location. The contents of the locations reserved are not altered by this instruction. The constant *k* must be an unsigned RAMM constant. If *k*=0, the symbol occurring in the location field is assigned to the current location but no memory location is reserved.

DEC k Create constant

The **DEC** pseudo-instruction stores the (decimal) RAMM constant *k* in the current location. If a symbol occurs in the location field, it is assigned to the current location.

RAMM Assembly Language

The fundamental language of any computer is its absolute ***machine language***. Machine language is the only language to which the computer can respond directly and therefore all other computer languages must be translated into the computer's machine language before execution of the program can be accomplished. There are many different machine languages as each computer processor has a specified set of instructions that it can execute. The machine language of a computer is normally in binary representation.

There are several difficulties inherent in programming in numeric machine code. Numeric operation codes (even when represented in decimal code or hexadecimal code corresponding to the binary representation) are hard to associate with any specific operation such as "add", "jump", and so on. The assignment of addresses to both instructions and data and the insertion of appropriate addresses as operands in instructions are tedious and conducive to error. Changing a few words of code in a program may make it necessary to alter many addresses scattered throughout the program.

A programmer very seldom writes a program in machine language. Instead, languages categorized as ***assembly languages*** are sometimes used. Assembly language is characterized by the fact that for each assembly language instruction there is usually one machine language instruction that will accomplish the desired operation. We say that there is a ***one-to-one*** correspondence between assembly language instructions and machine language instructions. The assembly language of a computer is termed ***machine dependent*** since the allowable instructions in an assembly language must usually have counterparts in the machine language of the computer involved. We can think of the assembly language for a computer as being merely a symbolic form of machine language coding with some assistance from a translator program (an assembler) in checking that the code is grammatically (or syntactically) correct.

Assembly language provides the programmer with a greater freedom and ease in program coding than is available in machine language. Once the programmer has written the program in assembly language, the program is input to the computer and translated into machine language by a program called an ***assembler***. The assembler translates the symbols and symbolic codes of assembly language, called the ***source code***, into numeric codes and addresses in the machine language of the computer. The machine language program that results from this translation process is called the ***object code***. The object code is then input to the computer for execution of the computer instructions to be performed. The diagram below (Figure 10) illustrates the computer processing involved starting with input of a source program coded in assembly language to output produced by the object program when it is finally executed to solve the problem.

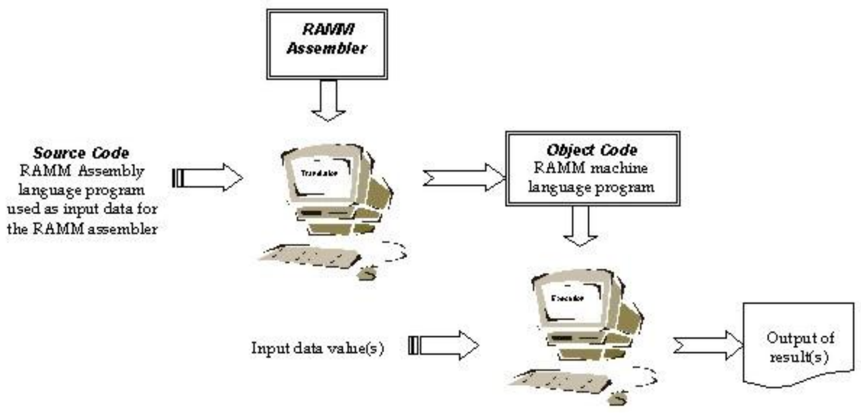

RAMM Assembler
Source Code
RAMM Assembly language program used as input data for the RAMM assembler
Object Code
RAMM machine language program
Input data value(s)
Output of result(s)

Figure 10

The components of an assembly language are similar to those of a written natural language. For example, there is an ***alphabet***, there are rules of grammar (***syntax***), and there are ***statements***. Some statements produce numerical information to be stored in memory and others simply give directions to the assembler. In the pages that follow, the assembly language for the RAMM computer will be presented.

There are differences, however, between computer languages and natural languages that may cause major difficulties to the programmer. First, unlike natural languages, the grammar of an assembly language is usually well defined, very precise, and somewhat restrictive. Second, a computer language is used for communication between a human being and a machine, while a natural language is used for communication between two human beings. Third, a computer (or computer program) will do exactly what it has been told to do. A person who receives an ungrammatical letter is likely to allow for the poorly formed sentence or the misused word and understand, at least to some extent, what was intended rather than what was said. The computer makes no such allowances.

For even greater freedom writing programs, ***high-level languages*** (HLLs) may be used. Many of these languages are also termed ***procedure-oriented*** because the symbols and notations used in the language are related to the type of problem being solved. For example, much of the code in a FORTRAN program looks like the mathematical formulas that the program was written to solve. Some common high-level languages are FORTRAN, COBOL, Pascal, C, C++ and BASIC. An instruction written in a high-level language usually results in a number of machine language instructions. For this reason, we say that there is a ***one-to-many correspondence*** between instructions written in a high-level language and instructions in machine language. High-level languages are ***machine independent*** because a program written in a high-level language can be translated into the machine languages of many different computers. As assemblers translate assembly language programs, compilers or interpreters translate high-level language programs into machine language before execution of the computer instructions can take place.

The remainder of this paper will be concerned with a presentation and description of the RAMM assembly language. As with any written language, there must be an alphabet or character set so that information may be transcribed.

Characters

A ***character*** in RAMM assembly language is a printed alphabetic capital letter ***A*** through ***Z***, a numeral ***0*** through ***9***, or the special characters **+**, **-**, and ***blank***. The term ***alphanumeric character*** includes both alphabetic and numeric characters and no others. The lowercase ***b*** will be used to indicate a blank when it is desirable to call attention to a blank in this presentation of the RAMM assembly language. The RAMM Assembler does not, however, recognize the lowercase ***b***. Blanks will also be indicated by blanks where it is not likely to cause confusion. The term ***first character*** refers to the leftmost character of a string of characters.

Symbols

A RAMM ***symbol*** is composed of up to four characters left justified in a field. The first character must be alphabetic; the remaining, alphanumeric. A blank terminates a symbol. Left justified means that the leftmost character in the field appears in the leftmost column of the field. Any blank characters in the field will be at the right side of the field. The following are examples of RAMM symbols:

Valid
C
A2Z
DATA
345
END See note*

Invalid	*Reason*
XY-Z	illegal character, -
T 4	illegal character, blank
CA2B3	too many characters

*Note: It is recommended that the ***OP*** code mnemonics not be used as symbols.

Since a blank terminates a symbol, ***Tb1*** and ***Tb2*** would both be taken as the symbol ***T***.

A symbol is a name for a location in memory. When a programmer wishes to refer to a location in memory, he names the location by a symbol and uses the symbol whenever he wishes to refer to that memory location. It is desirable to use symbols that have some association to the problem being solved. For example, a location containing the integer constant ***2*** might well be named with the symbol ***K2***, or a location containing the result of an addition operation might be named ***SUM***.

Constants

A ***constant*** in RAMM is one of the following:

1. A nonnegative decimal integer containing exactly four decimal digits. All leading zeros must appear. The largest nonnegative value is 9899. A nonnegative constant must be unsigned.

2. A negative decimal integer that begins with a minus sign followed by exactly three decimal digits. The smallest negative value is -999.

A blank terminates the constant. Examples of valid RAMM constants are:

```
0001
0123
-003
-123
8999
```

Note that, in addition to the range of constant values that can be entered using the DEC operation code, values up to a maximum of 9999 can be computed, stored and printed during execution of a program. For example, the constant 8999 could be added to the constant 1000 leaving a result of 9999 in the A-Register. This value could then be stored into a main memory location for further use during execution.

Delimiters

A blank is used in RAMM as a ***delimiter*** to terminate a symbol or a constant and to separate fields.

Instructions

An ***instruction*** in RAMM assembly language has the form:

OP ***R***

where the ***OP*** is an operation code and ***R*** is a RAMM symbol or RAMM constant.

In an instruction, the ***operation code***, ***OP***, tells what action is to take place. Operation codes are defined by the designers of the language and are usually mnemonic (memory aided) abbreviations of the operations to be performed. ***R*** is the ***operand*** of the instruction and tells what information is to be used in performing the operation. In many instructions it is the symbol that represents the address of a word to be used. A list of RAMM operation codes (op codes) is given elsewhere.

There are two types of instructions in an assembly language: ***computer instructions*** and ***pseudo-instructions*** (often called pseudo-operations). In a ***computer instruction***, the operation code corresponds to a machine language operation code of the computer. For a computer instruction, the assembler translates a mnemonic ***OP*** into a machine language code. Similarly, a symbolic ***R*** is translated into the numeric equivalent (an address between ***00*** and ***99***, inclusive) that the assembler has assigned to it. The assembler then forms from these parts the numeric machine instruction that corresponds to the symbolic computer instruction written by the programmer.

The second type of assembler language instruction, the ***pseudo-instruction***, furnishes information to the assembler and does not correspond to a numeric machine instruction. From the pseudo-instruction, the assembler determines some kind of action that the assembler is to take with respect to the program that it is translating. A pseudo-instruction may, for example, instruct the assembler that a certain number of consecutive memory locations are allocated with the first one named by a symbol, or that the assembler has reached the end of the program and hence is to stop translating.

Statements

Statements are the "sentences" of the assembly language. The most common form of statement contains an instruction consisting of an operation code and its operand (usually a symbol for an address). It also contains, at least implicitly, the address of the memory location in which the instruction is to be stored, and remarks by the programmer. All of this must be presented to the computer in a standard form, which will now be discussed.

Fields

The programmer enters the consecutive statements for his program, one per line, in a text file. In many assembly languages (including RAMM), certain parts of a statement must be placed in specific columns in the line. The consecutive columns reserved for some specific use are collectively called a ***field***. A given field is usually identified according to the type of information to be placed in that field, for example, "the ***OP*** code field." If information must be located within a field so that the leftmost character occurs in the leftmost column of the field, it is said that the contents of the field must be left justified. Similarly, if the rightmost character must occur in the rightmost column, it is said that the contents of the field must be right justified. All ***symbols***, ***op codes*** and ***constants*** in RAMM assembly language are left justified in the fields in which they appear.

The RAMM Assembly language statement is composed of six fields: ***location***, ***operation code***, ***address***, ***sign***, ***number***, and ***remarks***. Depending on the operation code, one or more of these fields may be blank.

Col.	1-4	5-6	7-9	10	11-14	15	16	17-19	20-40
Field	**location**		**op code**		**address**	**sign**	**number**		**remarks**

- The ***location field*** (columns ***1-4***) may contain blanks or a valid RAMM symbol: up to four alphanumeric characters, left-justified, with no imbedded blanks.
- The ***operation code field*** (columns ***7-9***) may contain a valid three- character operation code.
- The ***address field*** (columns ***11-14***) may contain:

  blanks, or
  a valid RAMM symbol: up to four alphanumeric characters, left-justified, or
  a valid RAMM constant less than ***9900***.

- The ***sign field*** (column ***15***) may contain a **+**, **-**, or blank. This field is used to allow addressing relative to a given symbol that appears in a location field.
- The ***number field*** (column ***16***) can be blank or contain an integer between ***0*** and ***9***, inclusive. This field is used to address a memory location relative to the one identified by the symbol in the location field.
- The ***remarks field*** (columns ***20*** through ***40***) is completely ignored by the assembler (except in producing a listing). This field may contain any alphanumeric or special character.

Field separators

Columns ***5-6***, ***10***, and ***17-19*** must contain blanks. These columns serve as field separators.

Instruction statement

In an instruction statement, the ***OP*** code field and the address field must contain a legal RAMM assembly instruction. The location field may be blank, or may contain a RAMM symbol. The remarks field may contain anything desired. The sign and number fields may both be blank or contain a sign and number, respectively.

Pseudo-instruction statement

In a pseudo-instruction statement the ***OP*** code field must contain a legal RAMM pseudo-instruction (***BSS***, ***DEC***, or ***END***). The address field must contain a legal RAMM constant or symbol. The location field may contain blanks or a legal RAMM symbol. The remarks field may contain anything desired.

End statement

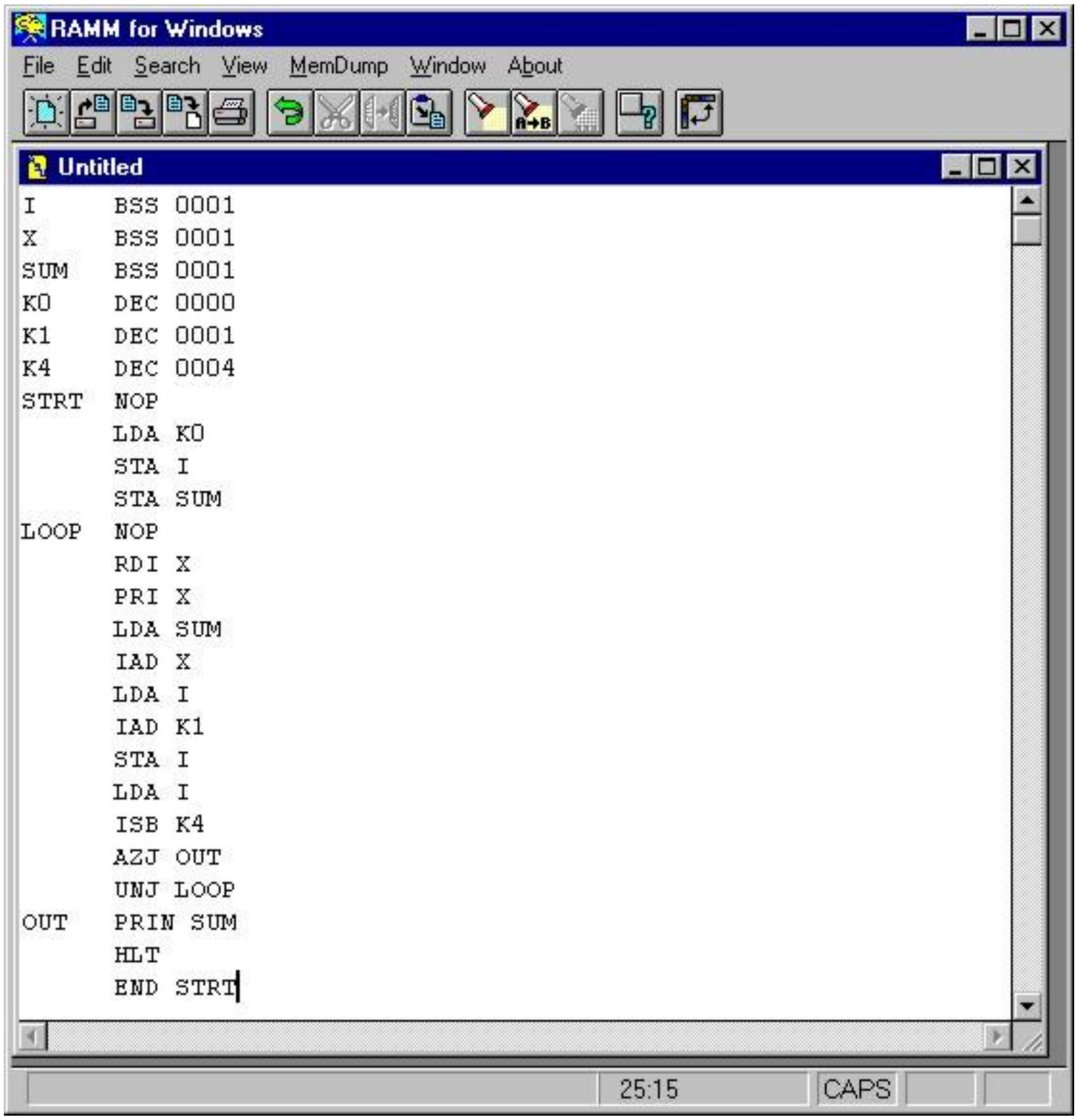
The ***END*** statement is formed by using a blank (recommended) location field, the ***END*** pseudo-instruction as the ***OP*** code, and a remark, if desired, in the remarks field. This statement must be the final statement of the program, and no statement may occur after it. This statement stops the translation process. Note the ***END*** instruction corresponds to the ***99*** code that is recognized by the RAMM computer as the marker for the end of the machine language code. If execution of the machine language program is not supposed to begin at ***00***, the address field of the ***END*** instruction should have a RAMM symbol that indicates the first instruction to be executed in the program.

Use of the Location field

In RAMM assembly language the location field is synonymous in all statements with an address in memory. The operation code and address fields of the statement determine what information the assembler puts into that location in memory. The programmer writes a symbol in the location field of a statement only if he must refer to that location elsewhere in his program. Since a symbol is synonymous with an address in memory, it is obvious that the same symbol cannot refer to more than one address. Therefore, a given symbol may be used only once per program in the location field of a statement. If the programmer inadvertently uses the symbol more than once in the location field, the assembler will use the address of the first occurrence. A symbol in the address field of a statement (the ***R***-term of an instruction) refers to the location that bears that symbol name. If the programmer must refer in some instruction to a location, that location must be named by a symbol. Hence every RAMM symbol occurring in the address field within a program must occur in the location field of some statement in the program. If the programmer inadvertently refers in the address field of a statement to a symbol that does not occur in the location field of any statement in the program, the assembler informs the programmer that he has used an undefined symbol.

Assembling, Loading and Executing a RAMM Assembly Language Program

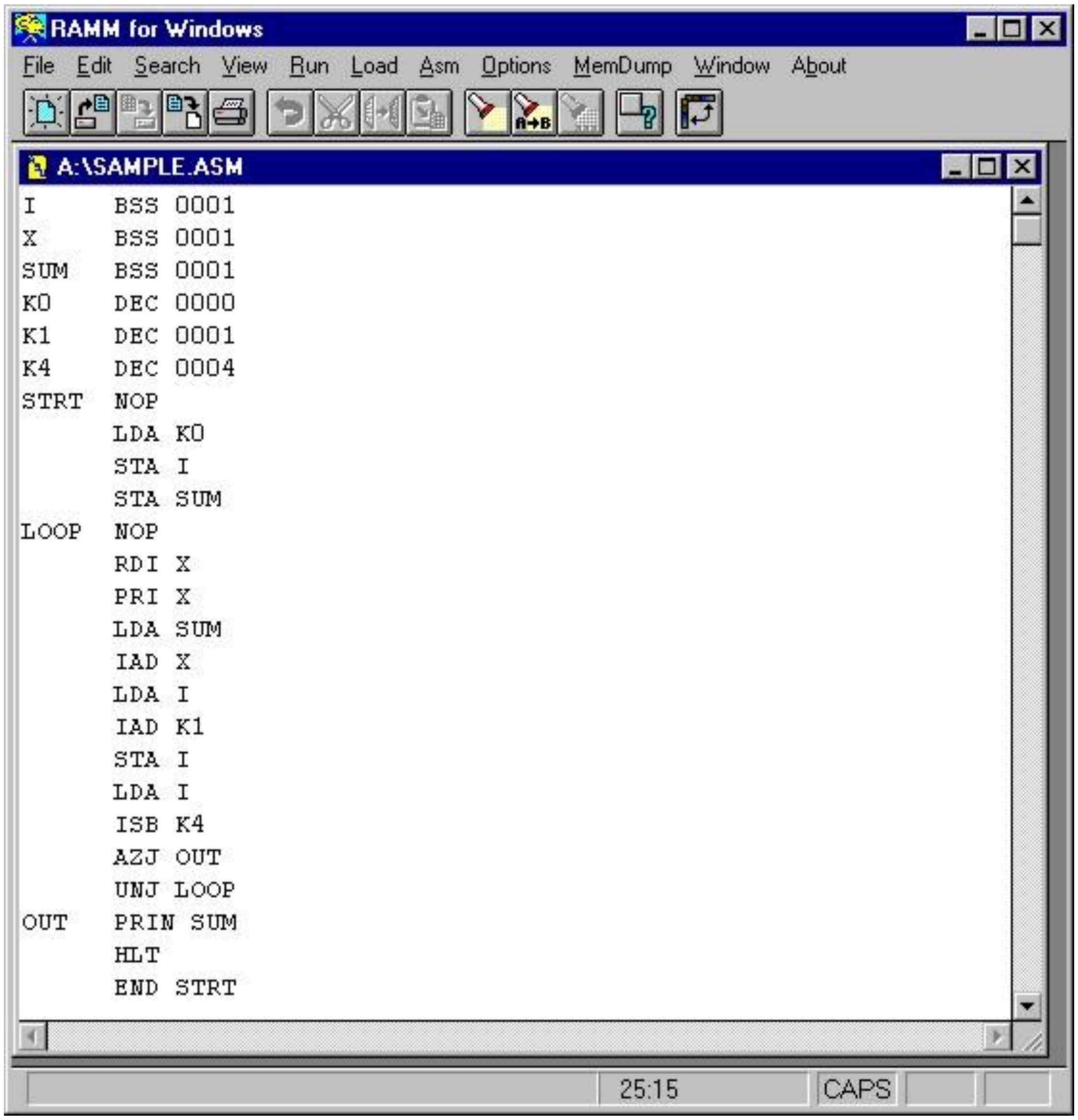
To create a RAMM assembly language program you should activate the RAMM Computer from the *Start/Programs* menu on the campus network. Choose *File/New* to pull up an *Untitled* text window. Type in the RAMM assembly language program (Figure 11), beginning each instruction in the first column of a new line and using blanks to pad the instruction so that each field falls in its required columns. Save the file to your disk with *File/Save/**labname.ASM*** or *File/Save As…/labname.ASM*.



```
I      BSS 0001
X      BSS 0001
SUM    BSS 0001
K0     DEC 0000
K1     DEC 0001
K4     DEC 0004
STRT   NOP
       LDA K0
       STA I
       STA SUM
LOOP   NOP
       RDI X
       PRI X
       LDA SUM
       IAD X
       LDA I
       IAD K1
       STA I
       LDA I
       ISB K4
       AZJ OUT
       UNJ LOOP
OUT    PRIN SUM
       HLT
       END STRT
```

Figure 11

Note in Figure 12 that the choices at the top of the screen have increased to include Run, Load, Asm (because the file has the extension .ASM), and Options. Each assembly language program must be processed through three phases: assemble, load, and execute. First, the assembly language source code must be ***assembled*** (translated) into machine language object code. Second, the machine language object code must be ***loaded*** into the main memory of the RAMM Computer. Third, the machine language code that is in main memory can be ***executed*** (***run***) so that the required processing can take place. It is during this third phase that any input values are read in from the keyboard or from a dataset file (previously created, stored and named under *Options*) and that output is produced. The third phase may be repeated during a session without explicitly stepping through another assembly or load phase as long as no other program has been loaded into the RAMM main memory.


RAMM for Windows
File Edit Search View Run Load Asm Options MemDump Window About
A:\SAMPLE.ASM
25:15
CAPS

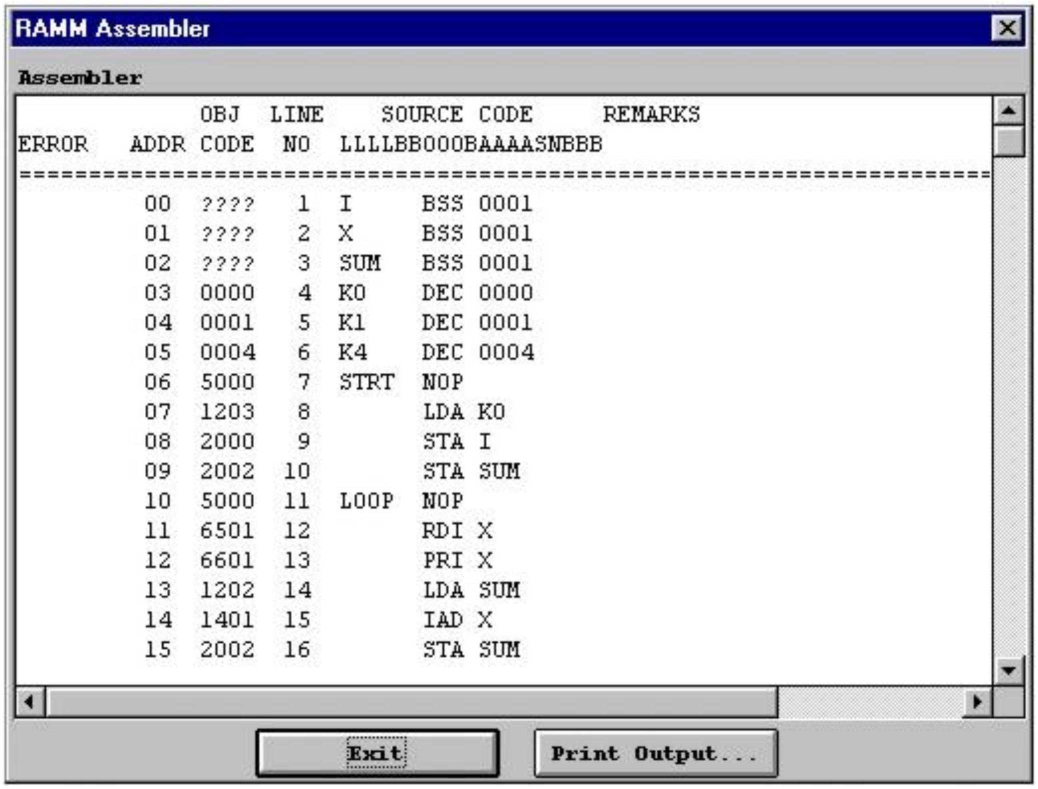
```
I       BSS 0001
X       BSS 0001
SUM     BSS 0001
K0      DEC 0000
K1      DEC 0001
K4      DEC 0004
STRT    NOP
        LDA K0
        STA I
        STA SUM
LOOP    NOP
        RDI X
        PRI X
        LDA SUM
        IAD X
        LDA I
        IAD K1
        STA I
        LDA I
        ISB K4
        AZJ OUT
        UNJ LOOP
OUT     PRIN SUM
        HLT
        END STRT
```

Figure 12

After the first phase has been accomplished by selecting *Asm* at the top of the screen, the source code and object code will be displayed in a window like Figure 13. Note the different columns of information provided in this screen. The ERROR column should be empty. Be sure to scroll down in the window to view the rest of the code. Errors are identified by character codes that can sometimes be associated with the instruction field containing the error. Some errors may be caused by a problem in a prior statement.

Note the column identification shown for the source code by ***LLLLBBOOOBAAAASNBBB***. That gridline can often help a programmer spot an alignment problem (for example, OP codes in the wrong columns).

It is instructional to notice that the contents of memory locations set up with the ***BSS*** pseudo-operation code contain unknown values illustrated by *????*. Note also in this program that ***Line Number 12*** shows us that the assembly language code ***RDI X*** has been translated into the machine language code ***6501*** and will be loaded into main memory location ***11*** during the next phase when we choose *Load*.

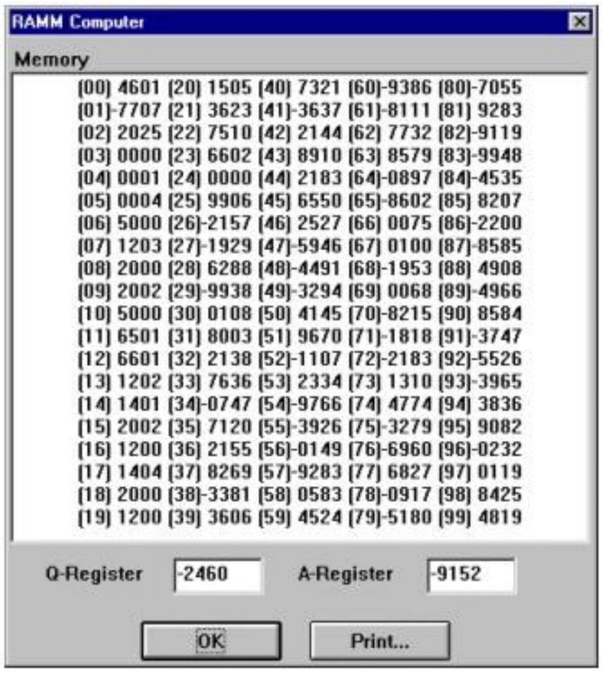


```
                OBJ  LINE    SOURCE CODE     REMARKS
ERROR    ADDR CODE  NO   LLLLBBOOOBAAAASNBBB
==========================================================================
          00  ????   1   I     BSS 0001
          01  ????   2   X     BSS 0001
          02  ????   3   SUM   BSS 0001
          03  0000   4   K0    DEC 0000
          04  0001   5   K1    DEC 0001
          05  0004   6   K4    DEC 0004
          06  5000   7   STRT  NOP
          07  1203   8         LDA K0
          08  2000   9         STA I
          09  2002  10         STA SUM
          10  5000  11   LOOP  NOP
          11  6501  12         RDI X
          12  6601  13         PRI X
          13  1202  14         LDA SUM
          14  1401  15         IAD X
          15  2002  16         STA SUM
```

Figure 13

Printing the output from the assembly phase can be helpful for debugging the source code (correcting the errors). Return to the text file to correct the assembly language code. (Save the changed file.) Repeat the assembly phase. Once the assembly phase shows that the ERROR column is empty a final printing of the successful assembly provides a copy of both the assembly language and the corresponding machine language versions of the program.

The next phase is the Load phase. Return to the text file after finishing a successful assembly. Choose Load. A request for Memory Dump will produce the window in Figure 14. The values in memory locations 00, 01, and 02 for your program will probably not match the values shown in Figure 14 for those locations, however, because the BSS operation code does not place a value in a memory location. The BSS operation code just reserves the requested number of memory locations.

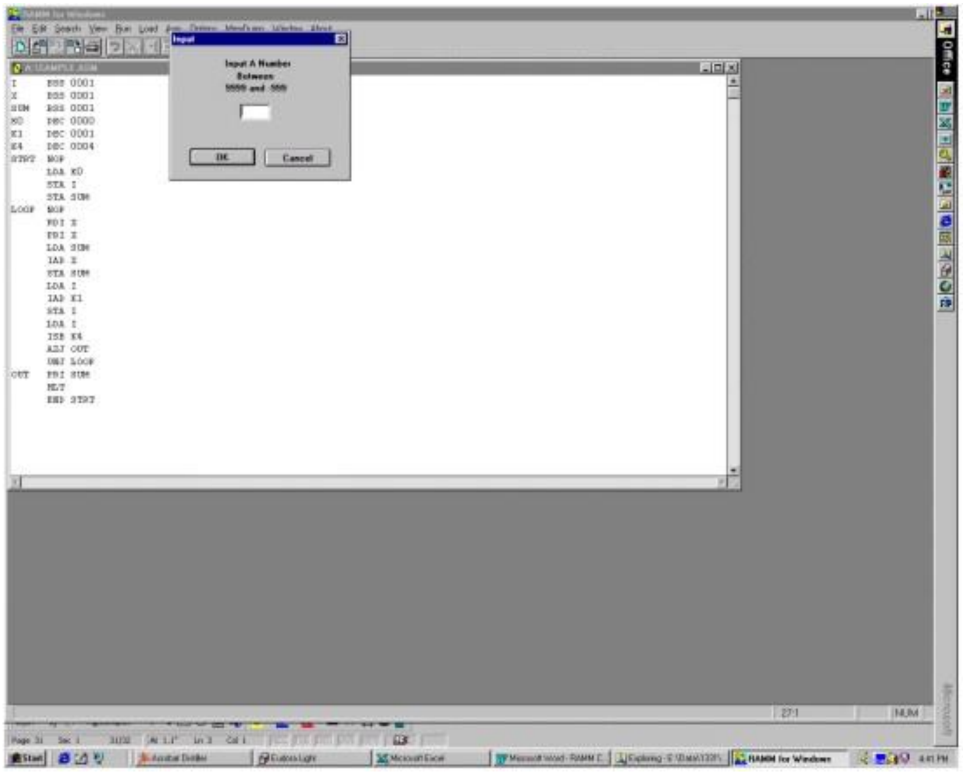


```
[00] 4601 [20] 1505 [40] 7321 [60]-9386 [80]-7055
[01]-7707 [21] 3623 [41]-3637 [61]-8111 [81] 9283
[02] 2025 [22] 7510 [42] 2144 [62] 7732 [82]-9119
[03] 0000 [23] 6602 [43] 8910 [63] 8579 [83]-9948
[04] 0001 [24] 0000 [44] 2183 [64]-0897 [84]-4535
[05] 0004 [25] 9906 [45] 6550 [65]-8602 [85] 8207
[06] 5000 [26]-2157 [46] 2527 [66] 0075 [86]-2200
[07] 1203 [27]-1929 [47]-5946 [67] 0100 [87]-8585
[08] 2000 [28] 6288 [48]-4491 [68]-1953 [88] 4908
[09] 2002 [29]-9938 [49]-3294 [69] 0068 [89]-4966
[10] 5000 [30] 0108 [50] 4145 [70]-8215 [90] 8584
[11] 6501 [31] 8003 [51] 9670 [71]-1818 [91]-3747
[12] 6601 [32] 2138 [52]-1107 [72]-2183 [92]-5526
[13] 1202 [33] 7636 [53] 2334 [73] 1310 [93]-3965
[14] 1401 [34]-0747 [54]-9766 [74] 4774 [94] 3836
[15] 2002 [35] 7120 [55]-3926 [75]-3279 [95] 9082
[16] 1200 [36] 2155 [56]-0149 [76]-6960 [96]-0232
[17] 1404 [37] 8269 [57]-9283 [77] 6827 [97] 0119
[18] 2000 [38]-3381 [58] 0583 [78]-0917 [98] 8425
[19] 1200 [39] 3606 [59] 4524 [79]-5180 [99] 4819
```

Q-Register -2460   A-Register -9152

Figure 14

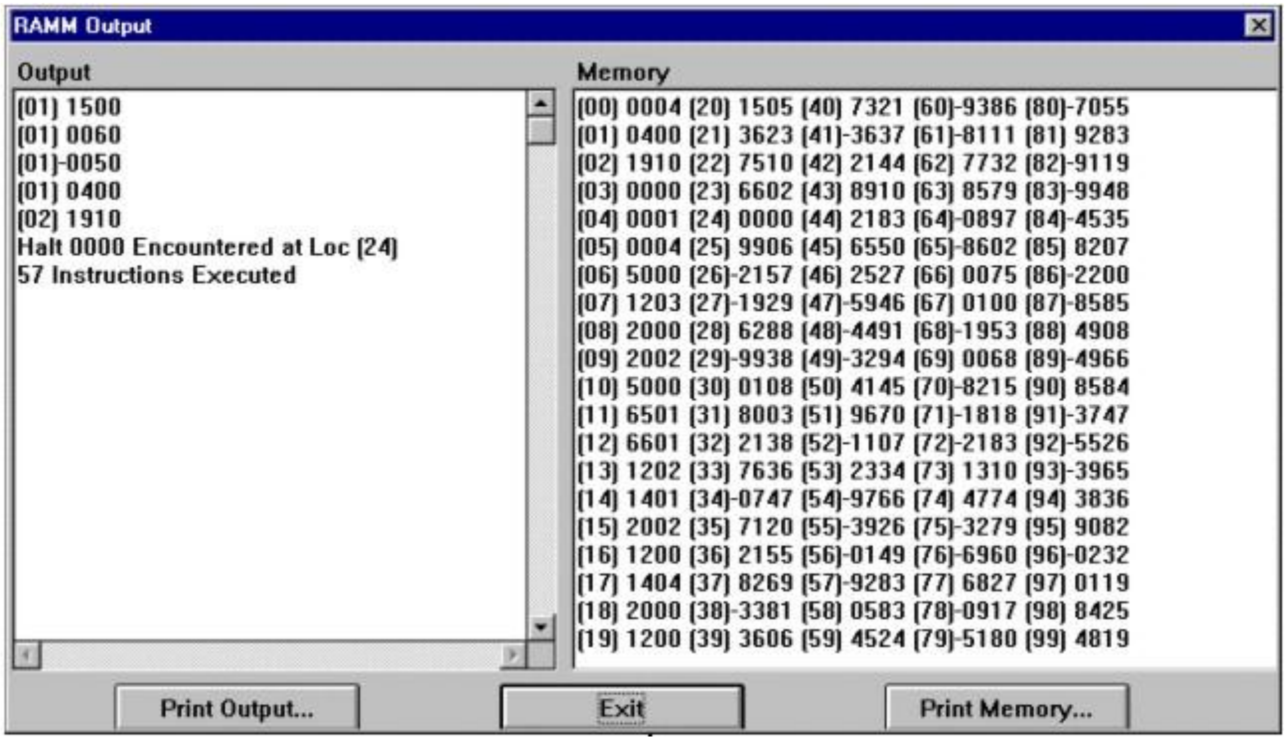
Return to the text file and choose Run to activate the third phase (execution). At this point, a dialog box will appear (Figure 15) requesting the first data value. A total of four data values should be requested for this current example that was to read four data values, print them and print their sum.



```
I     BSS 0001
X     BSS 0001
SUM   BSS 0001
K0    DEC 0000
K1    DEC 0001
K4    DEC 0004
STRT  NOP
      LDA K0
      STA I
      STA SUM
LOOP  NOP
      RDI X
      PRI X
      LDA SUM
      IAD X
      STA SUM
      LDA I
      IAD K1
      STA I
      LDA I
      ISB K4
      AZJ OUT
      UNJ LOOP
OUT   PRI SUM
      HLT
      END STRT
```

Figure 15

Final results of the sample program are shown in Figure 16.



```
(01) 1500
(01) 0060
(01)-0050
(01) 0400
(02) 1910
Halt 0000 Encountered at Loc (24)
57 Instructions Executed
```

```
(00) 0004 (20) 1505 (40) 7321 (60)-9386 (80)-7055
(01) 0400 (21) 3623 (41)-3637 (61)-8111 (81) 9283
(02) 1910 (22) 7510 (42) 2144 (62) 7732 (82)-9119
(03) 0000 (23) 6602 (43) 8910 (63) 8579 (83)-9948
(04) 0001 (24) 0000 (44) 2183 (64)-0897 (84)-4535
(05) 0004 (25) 9906 (45) 6550 (65)-8602 (85) 8207
(06) 5000 (26)-2157 (46) 2527 (66) 0075 (86)-2200
(07) 1203 (27)-1929 (47)-5946 (67) 0100 (87)-8585
(08) 2000 (28) 6288 (48)-4491 (68)-1953 (88) 4908
(09) 2002 (29)-9938 (49)-3294 (69) 0068 (89)-4966
(10) 5000 (30) 0108 (50) 4145 (70)-8215 (90) 8584
(11) 6501 (31) 8003 (51) 9670 (71)-1818 (91)-3747
(12) 6601 (32) 2138 (52)-1107 (72)-2183 (92)-5526
(13) 1202 (33) 7636 (53) 2334 (73) 1310 (93)-3965
(14) 1401 (34)-0747 (54)-9766 (74) 4774 (94) 3836
(15) 2002 (35) 7120 (55)-3926 (75)-3279 (95) 9082
(16) 1200 (36) 2155 (56)-0149 (76)-6960 (96)-0232
(17) 1404 (37) 8269 (57)-9283 (77) 6827 (97) 0119
(18) 2000 (38)-3381 (58) 0583 (78)-0917 (98) 8425
(19) 1200 (39) 3606 (59) 4524 (79)-5180 (99) 4819
```

Figure 16