Chapter 8: Programming in Assembly Language

Objectives

When you are finished with this lesson you will be able to:

- deal with negative and real numbers as they are represented in a computer
- write programs with loop constructs
- use the most common of the addressing modes of the 68000 ISA to write assembly language programs
- Express the standard program control constructs of the C++ language in terms of their assembly language equivalents
- use the program stack pointer register to write programs containing subroutine calls

Introduction

In chapter 7 we were introduced to the basic structure of assembly language programming. In this lesson we'll continue our examination of the addressing modes of the 68K and then examine some more advanced assembly language programming constructs, such as loops and subroutines with the objective of trying to use as much of these new methods as possible in order to improve our ability to program in assembly language.

Since you already understand how to write programs in high-level languages, such as C, C++ and Java, we'll try to understand the assembly language analogs of these programming constructs in terms of the corresponding structures that you're already familiar with, such as DO, WHILE, and FOR loops. Remember, any code that you write in C or C++ will ultimately be compiled down to machine language, so there must be an assembly language analog for each high-level language construct.

One of the best ways to understand assembly language programming is to closely examine an assembly language program. It actually doesn't help very much to work through the various instructions themselves, because once you're familiar with the Programmer's Model of the 68K and the addressing modes, you should be able to find the instruction that you need in the reference manual and develop your program from there. Therefore, we'll focus our efforts on trying to understand how programs are designed and developed, and then look at several example programs to try to understand how they work.

Numeric representations

Before we move deeper into the mysteries of assembly language we need to pick-up a few loose ends. In particular, we need to examine how different types of numbers are stored within a computer. Up to now, we've only looked at a positive number from the perspective of the range of the number in terms of the number of binary bits we have available to represent it.

The largest positive number that you can represent is given by:

$2^N - 1$

where N is the number of binary bits.

Thus, we can represent a range of positive integers as follows:

number of binary bits	Maximum value	name	C equivalent
4	15	nibble	----
8	255	byte	char
16	65,535	word	short
32	4,294,967,295	long	int
64	18,446,744,073,709,551,616	long long	----

Obviously, there are two missing numeric classes: negative numbers and real numbers. Let's look at negative numbers first; then we'll move onto real numbers.

It may seem strange to you but there are no circuits inside of the 68K that are used to subtract one number from another. Subtraction, however, is very important, not only because it is one of the four commonly used arithmetic operations (add, subtract, multiply, and divide), but it is the basic method to use to determine if one value is equal to another value, greater than the other value, or less than the other value. These are all comparison operations. The purpose of a comparison instruction, such as CMP or CMPA, is to set the appropriate flags in the Condition Control Register (CCR), but not change the value of either of the quantities used in the comparison. Thus, if we executed the instruction:

```
test   CMP.W  D0,D1          * Does <DO> = <D1> ?
```

the zero bit would be set to 1 if <D0> = <D1> because this is equivalent to subtracting the two values. If the result of the subtraction is equal to zero, then the two quantities must be equal. If this doesn't click with you right now, that's OK. We'll discuss the CCR in much more detail later on in this lesson.

In order to subtract two numbers in the 68K architecture it is necessary to convert the number being subtracted to a negative number and then the two numbers may be added together. Thus:

$A – B = A + (–B)$

The 68K processor always assumes that the numbers being added together may be positive or negative. The algorithm that you write must utilize the carry bit, C, the negative bit, N, the overflow bit, V, and the extended bit, X, to decide upon the context of the operation. The processor also uses negative numbers to branch backwards. It does this by adding a negative number to the contents of the program counter, PC, which effectively causes a backwards branch in the program flow. Why this is so deserves a bit of a diversion. Recall that once an op-code word is decoded the computer knows how many words are contained in the instruction currently being executed. One of the first things it does is to advance the value in the PC by that amount so that the value contained in the PC is now the address of the next instruction to be executed.

Branch instructions cause the processor to execute an instruction out-of-sequence by simply adding the operand of the branch instruction to the value in the PC. If the operand, called a ***displacement***, is positive, the branch is in the forward direction. If the displacement is a negative number, the branch is backwards.

In order to convert a number from positive to negative we convert it to its ***two's complement*** representation. This is a two-step process.

Step 1: Complement every bit of the byte, word or long word. To complement the bits, you change every 1 to a 0 and every 0 to a 1. Thus, the complement of $55 is $AA because Complement (01010101) = 10101010. The complement of 00 is $FF, and so on.

Step 2: Add 1 to the complement. Thus, -$55 = $AB.

Two's complement is a version of a method of subtraction called ***radix complement***. This method of subtraction will still work properly if you are working in base, 2, 8, 10, 16 or any other base system. It has the advantage of retaining a single method of arithmetic (addition) and dealing with subtraction by converting the subtrahend to a negative number.

We define the radix complement of a number, N, represented in a particular base (radix) containing d digits as follows:

radix complement = $r^d – N$ for non-zero values of N and = 0 if N = 0. In other words, -0 = 0

Let's see how this works for the decimal number 4,934. Thus, $r^d – N = 10^4 – 4,934$

= 5,066

So 5,066 equals -4,934. Let's subtract 4,934 from 8013 and see if it actually works. First, we'll just subtract the numbers in the old fashioned way:

8,013 – 4,934 = 3,079

Now, we'll add together 8,013 and the radix complement of 4,934:

8,013 + 5,066 = 13,079 = >1 3079

Clearly, the least significant 4 digits are the same, but we're left with a 1 in the most significant digit position. This is an artifact of the radix complement method of subtraction and we would discard it. Why we are left with this number can be seen if we just multiply out what we've done.

If $x = a - b$, then $x = a - (r^d - b) = r^d + (a - b)$. Thus, we are left with a numeric result in two parts: the radix to the power of the number of digits and the actual result of the subtraction. By discarding the radix to the power portion of the number, we are left with answer we want. Thus, two's complement arithmetic is just radix complement arithmetic for the base two number system.

In order to convert a negative number back to its positive representation, perform the two's complement transformation in exactly the same way. The most significant bit of the number is often called the ***sign bit*** because a negative number will have a 1 in the most significant bit position, but it isn't strictly a sign bit, because the negative of +5 isn't –5.

In any case, this is a minor point because the N flag is always set if result of the operation places a 1 in the most significant bit position for that byte, word or long word. Examine the following snippet of code.

Code Example

```
       ORG        $400

start  MOVE.B     #00,D0

       MOVE.B     #$FF,D0

       MOVE.W     #$00FF,D0

       MOVE.W     #$FFFF,D0

       MOVE.L     #$0000FFFF,D0

       MOVE.L     #$FFFFFFFF,D0

       END        $400
```

Tracing the code in a simulator shows that the N bit is set each time the most significant bit is a 1 for the operation being performed. Thus, for a byte operation, DB7 = 1. For a word operation, DB15 = 1. For a long word operation, DB31 = 1.

Below is the simulator trace for this program. The N bit and the contents of register D0 are highlighted in red. Also notice that the simulator represents numbers as positive or negative as well. Thus, $FF is shown as –1 in instruction 2, but is shown in the register as $FF.

```
PC=000400 SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   Program start
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.B #0,D0

PC=000404 SR=2004 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=1   After
instruction
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0   MOVE.B #0,D0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.B #-1,D0

PC=000408 SR=2008 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   After
instruction
D0=000000FF D1=00000000 D2=00000000 D3=00000000 V=0   MOVE.B #-1,D0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.W #255,D0

PC=00040C SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   After
instruction
D0=000000FF D1=00000000 D2=00000000 D3=00000000 V=0   MOVE.W #255,D0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.W #-1,D0

PC=000410 SR=2008 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   After
instruction
D0=0000FFFF D1=00000000 D2=00000000 D3=00000000 V=0   MOVE.W #-1,D0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L #65535,D0

PC=000416 SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   After
instruction
D0=0000FFFF D1=00000000 D2=00000000 D3=00000000 V=0   MOVE.L
#65535,D0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L #-1,D0

PC=00041C SR=2008 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   After
instruction
D0=FFFFFFFF D1=00000000 D2=00000000 D3=00000000 V=0   MOVE.L #-1,D0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
```

By creating two’s complement negative numbers, all arithmetic operations are converted to addition. However, when we use negative numbers our range is cut in half. Thus,

- Range of 8-bit number is -128 to +127 (zero is positive)
- Range of 16-bit numbers is –32,768 to +32,767
- Range of 32-bit numbers is –2,147,483,648 to +2,147,483,647

What happens when your arithmetical operation exceeds the range of the numbers that you are working with? As software developers I’m sure you’ve seen this bug before. Here’s a simple example which illustrates the problem. In the following C++ program ***bigNumber*** is a 32-bit integer, initialized to just below the maximum positive value for an integer, +2,147,483,647. As we loop and increment the number ten times, we will eventually exceed the maximum allowable value for the number. We can see the result of the overflow in the output box.

Source Code	**Program output**
#include <iostream.h>	The big number equals 2147483641
int main(void)	The big number equals 2147483642
{	The big number equals 2147483643
int bigNumber = 2147483640 ;	The big number equals 2147483644
for (int i = 1; i <= 10 ; i++)	The big number equals 2147483645
cout << "The big number equals "	The big number equals 2147483646
<< bigNumber + i << endl ;	The big number equals 2147483647
return 0 ;	The big number equals -2147483648
}	The big number equals -2147483647
	The big number equals -2147483646

Unfortunately, unless we write an error handler to detect the overflow, the error will go undetected. We will have the same problem with two’s complement arithmetic.

Two’s complement notation makes the hardware easier but developing the correct mathematical algorithms can be a minefield for the unwary programmer! Algorithms for arithmetic operations must carefully set and use the condition flags in order to insure that the operation is correct.

For example, what happens when the sum of an addition of two negative numbers (bytes) is less than –128? You need to consider the state of the overflow bit. In this case, V=1 if a two’s complement addition overflows, but V=0 if no overflow occurs. For an n bit signed number, V=1 indicated that the true result is ***greater than*** $2^{n-1} – 1$, or ***less than*** $– 2^{n-1}$. Without the overflow flag, positive results will be interpreted as negative numbers, and *vice versa.*

The expression ***N XOR V*** always gives the correct sign of a two's complement result. This might not be so obvious. Suppose that you added $6A and $7C as bytes. These are two positive numbers. If we add these two numbers together we get $E6. However, $E6 is a negative number. The computer interprets this as -$1A, which is clearly incorrect. The overflow flag would have been set to 1 and the negative flag would have been set to 1, so we at least know that the sign of the result is a positive number and that an overflow has occurred. How would you remedy the situation? The same way that you do in C or C++: use a larger variable type. Now, if we add $006A and $007C we get $00E6, but the number is properly interpreted as a positive number. If you were only adding positive numbers together then you would also need to watch the Carry Bit to tell you if you overflowed on the addition.

We can see how using two's complement to represent signed numbers maps the range of the unsigned number to the range of the signed numbers. If we map the corresponding signed and unsigned numbers, we can see how the negative numbers are represented.

Unsigned 32-bit number	0 ……..2,147,483,647	2,147,483,648…….. 4,294,967,295
Signed 32-bit number	0……...2,147,483,647	- 2,147,483,648…………………..-1

Thus, the maximum value that the unsigned number can reach is equal to -1 as a two's complement and the number that is one digit greater than the maximum value of the signed number is equal to the maximum negative number in the range of the signed numbers.

Real numbers, or numbers containing a whole part and a fractional part, can only be approximated within most computers. You are all familiar with the ***float*** and ***double*** data types. These can only approximate a real number. In our computer, the question becomes one of, "How accurately do you want to approximate the real number? I'm sure that you, as software professionals, would never write this kind of an ***if*** statement because you know that the test could erroneously pass or fail do to progressive round-off and conversion errors.

```
float a;
float b;
if ( b <  a)
    [ do this ];
else
    [ do that ];
```

Let's see how real numbers are represented in a modern computer. Just as we represent the fractional part of a decimal number as the base 10 to progressively larger negative powers of 10 as we move to the right, away from the decimal point, we would use the same method for fractional numbers in any base. Thus, the binary number 10100101 ($A5) is equivalent to:

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}		
case 1	1	0	1	0	0	1	0	1	•					X	2^0
case 2		1	0	1	0	0	1	0	•	1				X	2^1

case 3			1	0	1	0	0	1	•	0	1			x	2^2
case 4				1	0	1	0	0	•	1	0	1		x	2^3
case 5					1	0	1	0	•	0	1	0	1	x	2^4

Thus, the binary integer 10100101 can be represented as the real number:

1010.0101 x 2^4

While the representation might look a bit strange at first, it is just the mantissa and exponent notation that you already are familiar with for decimal numbers. You can see from the above example that we limited ourselves to 4 decimal places. If we continued the process for another step without adding another decimal digit, our number would have become:

101.010 x 2^5

Thus, when we convert back to an integer value, our original number, $A5 becomes $A4. If you're an Accountant that's probably OK, but not so for Computer Scientists and Engineers!

Today, we represent floating point numbers in an industry standard form, called IEEE-754. The 1985 standard is maintained and published by the Institute of Electrical and Electronic Engineers (IEEE). The standard describes the format for representing single and double precision floating point numbers in computer systems. The standard is almost universally accepted because many of the higher performance microprocessors contain on-chip floating-point units, or ***FPUs***. Without a standard for representing floating point numbers, the representation of floating point numbers would be left to the microprocessor companies and much of the software libraries would not be portable across platforms.

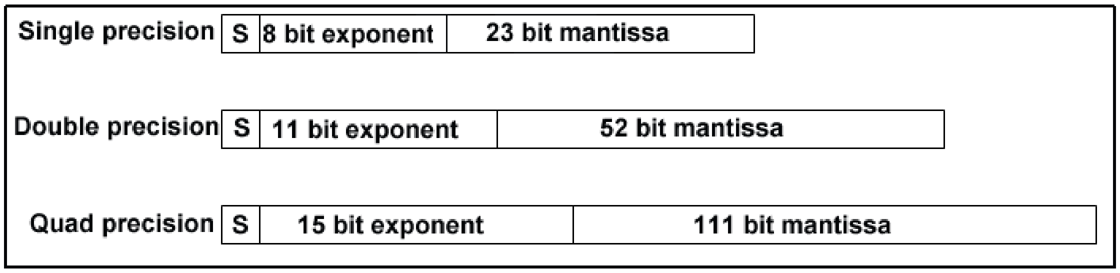


Figure 8.1: IEEE-754-1985 floating point number representations

The IEEE-754 standard defines single, double and quadruple formats for floating point numbers with each standard requiring 32, 64 and 128 bits, respectively. Figure 8.1 illustrates the partitioning of the bit fields for each of the formats.

The actual representation of the numbers is not obvious and there are several areas of the format that need further explanation. First, it is interesting to note that the floating point representation of a number uses the sign and magnitude representation for a number, rather than the integer representation of two’s complement. The sign bit, ***S***, is 0 for a positive number and 1 for a negative number.

The mantissa is always adjusted so that it is a number between 1 and 2. Thus, the form of the mantissa is similar to the way that we try to represent a decimal number in scientific notation. In general, a number in scientific notation has a mantissa that is greater than 1 and less than 10, so that we have one digit to the left of the decimal point and then a fractional part.

In the IEEE notation, a base 2 number is adjusted so that the mantissa always has a 1 to the left of the fractional part, but since the 1 is always there, it is omitted from the numeric fields in order to gain an extra bit of precision in the mantissa field. Thus, the single precision number actually requires 33 bits to represent it, but only 32 bits are stored because there is always a 1 to the left of the decimal point.

The exponent also requires a bit of investigation. There is no sign bit for the exponent, so it might first appear that we are using two’s complement for the exponent. This is a pretty good guess and the method used is similar in principle to using a two’s complement representation. Rather than two’s complement, the range of the bit field is offset by a bias value. In the case of an 8-bit exponent, the bias value is 127 (01111111). So, if the 8-bit value of the exponent is 10000001, the value of the exponent would be calculated as:

129 – 127 = 2.

For a double precision number the bias value is 1023 and for a quad precision number the bias value is 32,767. We can thus combine these parts and define a floating point number, N, as follows:[1]

$$N = -1^{S} \times 1.F \times 2^{E-B}$$

Here, S is the sign bit, F is the fractional part of the mantissa, E is the exponent and B is the bias. Thus, for a single precision number, we can represent the exponent in the range of 2^{-127} to 2^{128}.

Branching and Program Control

Most programs only execute five to seven instructions before taking a branch (non-sequential program fetch). Most branches are usually paired with the test instructions (CMP instructions) within the program so that a condition is tested and then the branch immediately follows the test. This pairing of the test and branch is a general rule because you don't want to lose the state of the flag, which is set by the test instruction, by executing another instruction. However, while pairing the test instruction with the branch instruction is the most common pairing, it isn't always necessary. Consider the follow two code snippets:

Snippet #1

```
        MOVE.W   #05,D0
loop    SUBI.W   #01,D0
        CMPI.W   #00,D0
        BNE      loop
```

Snippet #2

```
        MOVE.W   #05,D0
loop    SUBI.W   #01,D0
        BNE      loop
```

Both code snippets work equally well, but snippet #2 saves one instruction because the zero flag would be automatically set by the SUBI.W instruction. It is redundant to test the state of the flag with the CMPI.W instruction since it is already set by the subtraction instruction that precedes it. Ok, you might think that this is a bit extreme, but we worry about these things in assembly language. One of the primary reasons to program in assembly language is to gain absolute cycle by cycle control over the system. Shaving one redundant instruction could translate to speeding-up a time-critical routine by a few microseconds. Is that important? It's hard to say right now, but I can predict with some certainty that it might very well be important at some time in your future as a software developer.

The state of flags inside the condition code register, or CCR, (Z,N,V,X,C) determines if a particular branch will be taken. Sometimes, branches are taken as a result of exception processing, such as interrupts, bus errors, trap instructions or other errors. ***TRAP*** instructions are important because they are commonly used to interface the operating system, O/S, to the user's application code. We'll be using the **TRAP** instruction later on when we interface assembly language programs to the keyboard and display for user I/O. The TRAP instructions provide a well-controlled method of accessing the operating system services of the simulator.

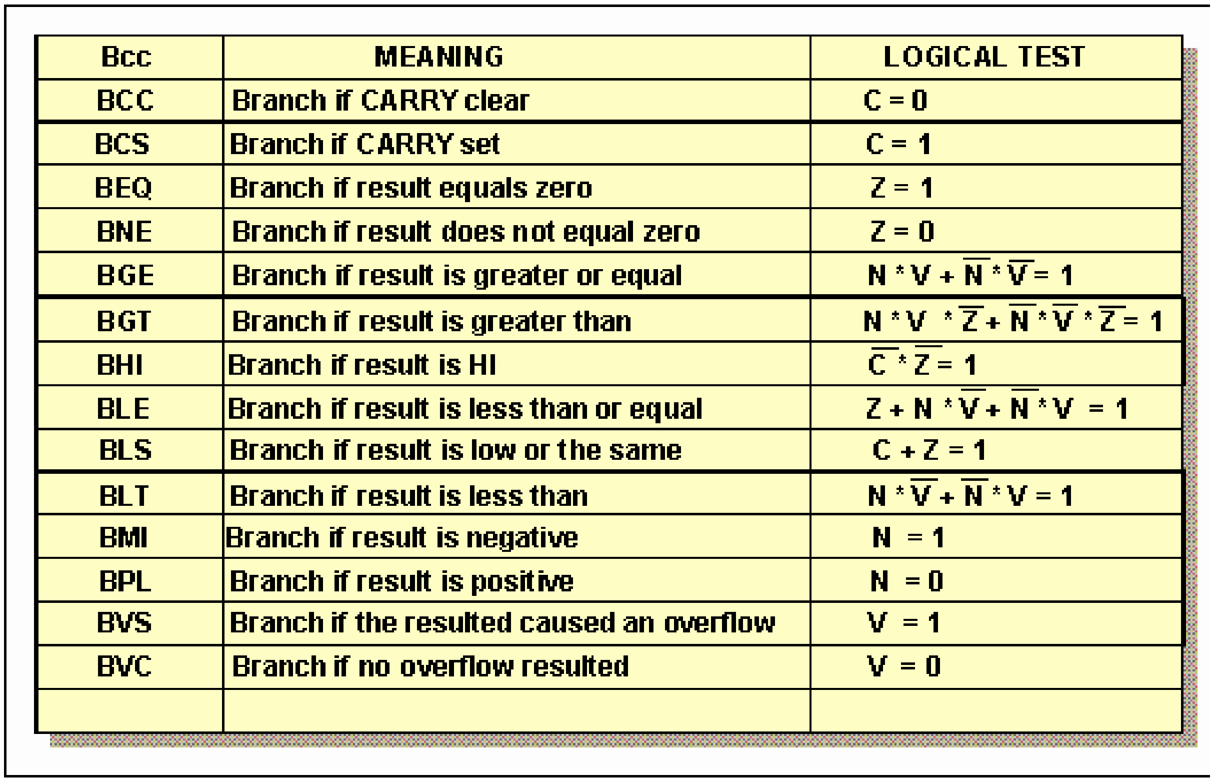
Status bits (***flags***) in the CCR may or may not change state with each instruction execution. Thus, as previously stated, it is customary to place the test instruction for the branch immediately in front of the branch instruction so that no other instruction will possibly alter the flag value. The following table summarizes the meaning of the CCR register flags.

Bit	**Definition**	**Meaning**
Z	ZERO	Set to 1 if result = 0, set to 0 if result is not zero
N	NEGATIVE	Bit equals the most significant bit (MSB) of the result
C	CARRY	Set to 1 if carry is generated out of the MSB of the operands for an addition. Also set to 1 if a borrow is generated. Set to 0 otherwise.
V	OVERFLOW	Set to 1 if there was an arithmetic overflow. This implies that the result cannot be represented by the operands. Otherwise set to 0**.**
X	EXTENDED	Transparent to data movement instructions. When affected by arithmetic instructions, it is set the same as the carry.

The branch instruction tests the state of the appropriate CCR flags, either individually or through a logical combination of the flags. If the logical condition evaluates to TRUE the branch is taken. Recall that the program counter (PC) is always pointing to the address of the next instruction to be executed. Adding or subtracting an ***offset value*** to or from the contents of the PC before the next instruction is fetched from memory, causes the processor to fetch the next instruction from a different location. The operand of a branch instruction is an 8-bit or 16-bit offset, called a ***displacement.*** Therefore, if the branch test condition evaluates to true, the displacement is added to the current value in the PC and this becomes the address of the next instruction.

In other words: <PC> + displacement →PC

The form of instruction is **`Bcc <displacement>.`** Here "**cc**" is shorthand notation for the condition code of the actual branch. You must replace **cc** with the proper test condition. Figure 8.2 is a summary of all of the conditional branch instructions and the way that they are evaluated.



Bcc	MEANING	LOGICAL TEST
BCC	Branch if CARRY clear	$C = 0$
BCS	Branch if CARRY set	$C = 1$
BEQ	Branch if result equals zero	$Z = 1$
BNE	Branch if result does not equal zero	$Z = 0$
BGE	Branch if result is greater or equal	$N * V + \overline{N} * \overline{V} = 1$
BGT	Branch if result is greater than	$N * V * \overline{Z} + \overline{N} * \overline{V} * \overline{Z} = 1$
BHI	Branch if result is HI	$\overline{C} * \overline{Z} = 1$
BLE	Branch if result is less than or equal	$Z + N * \overline{V} + \overline{N} * V = 1$
BLS	Branch if result is low or the same	$C + Z = 1$
BLT	Branch if result is less than	$N * \overline{V} + \overline{N} * V = 1$
BMI	Branch if result is negative	$N = 1$
BPL	Branch if result is positive	$N = 0$
BVS	Branch if the resulted caused an overflow	$V = 1$
BVC	Branch if no overflow resulted	$V = 0$

Figure 8.2: Summary of the conditional branch instructions and their meanings

Figure 8.3 is a segment of code containing three examples of conditional branches. The first branch test is a result of the instruction:

```
CMP.L   (A0),D0     * Read it back
```

Here, register A0 is a pointer to a location in memory. The value in that memory location is compared with the contents of D0 and if they're equal, the branch is taken (BEQ) to the instruction indicated by the label **`addr_ok`**.

The second branch test is a result of the comparison instruction

```
CMPI.B    #max_cnt,(A3) * Have we hit 4 bad locations yet?
```

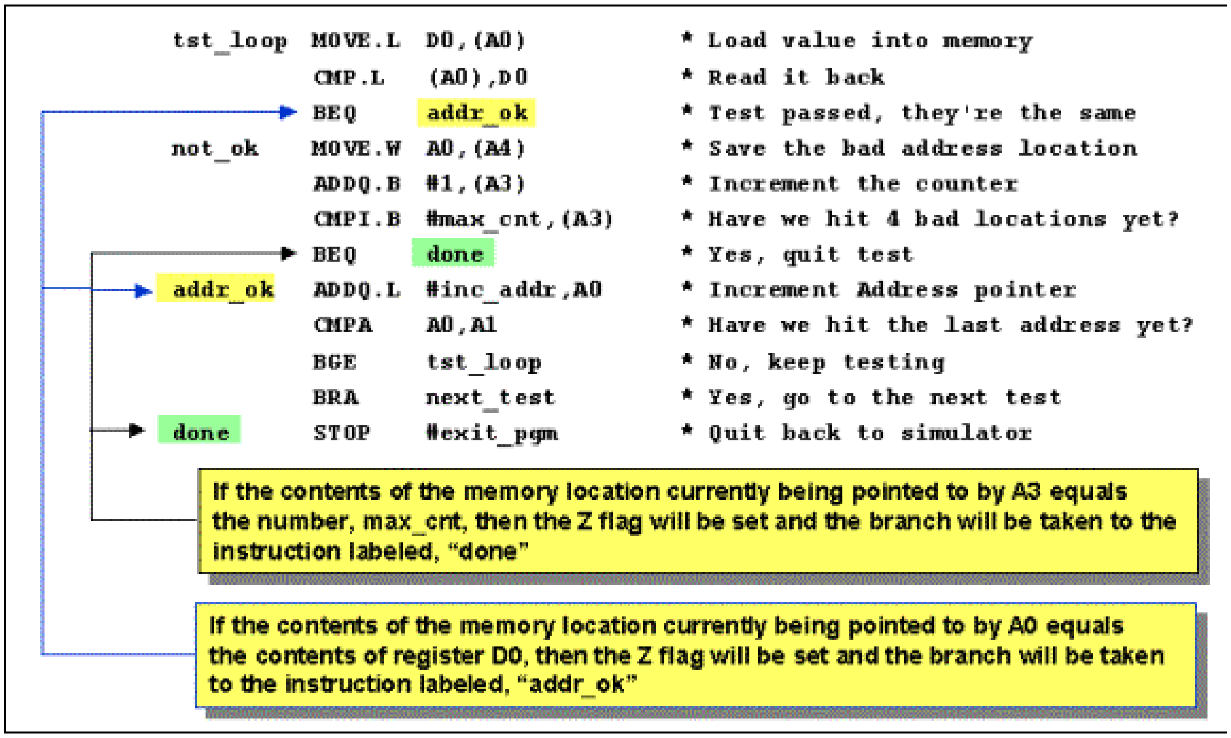
Here, the immediate value, **`max_cnt,`** is compared with the contents of the memory location pointed to by A3. If they are equal to each other the branch is taken to the instruction defined by the label **done**.

The third branch test is a result of the comparison instruction

```
CMPA    A0,A1           * Have we hit the last address yet?
```

Unlike the CMP instruction, the CMPA instruction is used when we need to compare the value of an address register. Here we compare the contents of two address registers, A0 and A1, and then branch if the value in register A1 is greater than or equal to the value in register A0.

Also note that we'll occasionally place a register in angle brackets. For example, <A0> = 0. This is just a shorthand way of saying "The contents of A0 = 0". You'll find that there are a number of instructions that are only used for address registers, or only used for data registers.



```
tst_loop  MOVE.L  D0,(A0)         * Load value into memory
          CMP.L   (A0),D0         * Read it back
          BEQ     addr_ok         * Test passed, they're the same
not_ok    MOVE.W  A0,(A4)         * Save the bad address location
          ADDQ.B  #1,(A3)         * Increment the counter
          CMPI.B  #max_cnt,(A3)   * Have we hit 4 bad locations yet?
          BEQ     done            * Yes, quit test
addr_ok   ADDQ.L  #inc_addr,A0    * Increment Address pointer
          CMPA    A0,A1           * Have we hit the last address yet?
          BGE     tst_loop        * No, keep testing
          BRA     next_test       * Yes, go to the next test
done      STOP    #exit_pgm       * Quit back to simulator
```

If the contents of the memory location currently being pointed to by A3 equals the number, max_cnt, then the Z flag will be set and the branch will be taken to the instruction labeled, "done"

If the contents of the memory location currently being pointed to by A0 equals the contents of register D0, then the Z flag will be set and the branch will be taken to the instruction labeled, "addr_ok"

Figure 8.3: Code segment showing conditional and unconditional branches

Finally, the **`branch always`** instruction **`(BRA)`** is an unconditional branch back to another instruction (not shown in this segment).

Addressing Modes

Up to now, we've focused on becoming comfortable with some of the fundamentals of assembly language programming. In order to do that, we've neglected a systematic study of the addressing modes of the 68K instruction set architecture. Before we go any further into other aspects of the architecture, such as stacks and subroutines, we'll need to at least have a single once-over of the primary addressing modes.

You're probably familiar with most of these right now from some of our examples. The following list of addressing modes is based upon the effective address field of the opcode word. Thus, we'll have a "mode/register" combination where that is appropriate; or a "mode/subclass" combination if we need to further subdivide the mode. This will become clearer as we discuss the modes.

Mode 0, Data Register Direct

Source or destination is a data register (D0 . . . D7)

Mode 1, Address Register Direct

Source or destination is an address register (A0 . . . A6)

Register direct addressing is the simplest of the addressing modes. The source or destination of an operand is a data register or an address register and the contents of the specified source register provide the source operand. Similarly, if a register is a destination operand, it is loaded with the value specified by the instruction. The following examples all use register direct addressing for source and destination operands.

- **`MOVE.B  D0,D3:`** Copies the source operand in register D0 to register D3
- **`SUB.L   A0,D3:`** Subtract the source operand in register A0 from register D3
- **`CMP.W   D2,D0:`** Compare the source operand in register D2 with register D0
- **`ADD.W   D3,D4:`** Add the source operand in register D3 to register D4

Registers are the most precious resource you have in a computer. They are an integral part of the computer's architecture and they will generally have the fastest access time in most assembly language operations.

Most arithmetic and logical operations must use one of the data registers or one of the address registers as one of the operands in the calculation. For example, the **`ADD`** instruction must use a data register as either the source or destination operand. The other operand may be one of several effective address modes.

Register direct addressing is also efficient because it uses short instructions. It takes only three bits to specify one of eight data registers. Register direct addressing is fast because the external memory does not have to be accessed. In general, programmers use register direct addressing to hold variables that are frequently accessed (i.e., scratchpad storage).

Good compilers make use of register direct addressing to increase performance. In fact, the declaration in C or C++ uses the keyword "register":

```
register int foo = 0;
```

tells the compiler that, if possible, you would like to keep "foo" available as a register variable rather than as a memory location. The compiler might not be able to grant your wish, but you've requested it.

It's pretty hard for assembly language programmers to keep track of registers and their contents, and to make the most effective use of them. This will become more apparent when we look at the architecture of the RISC processor. RISC processors have large register sets (the AMD 29000 has 256 registers) to give compilers plenty of fast, local storage.

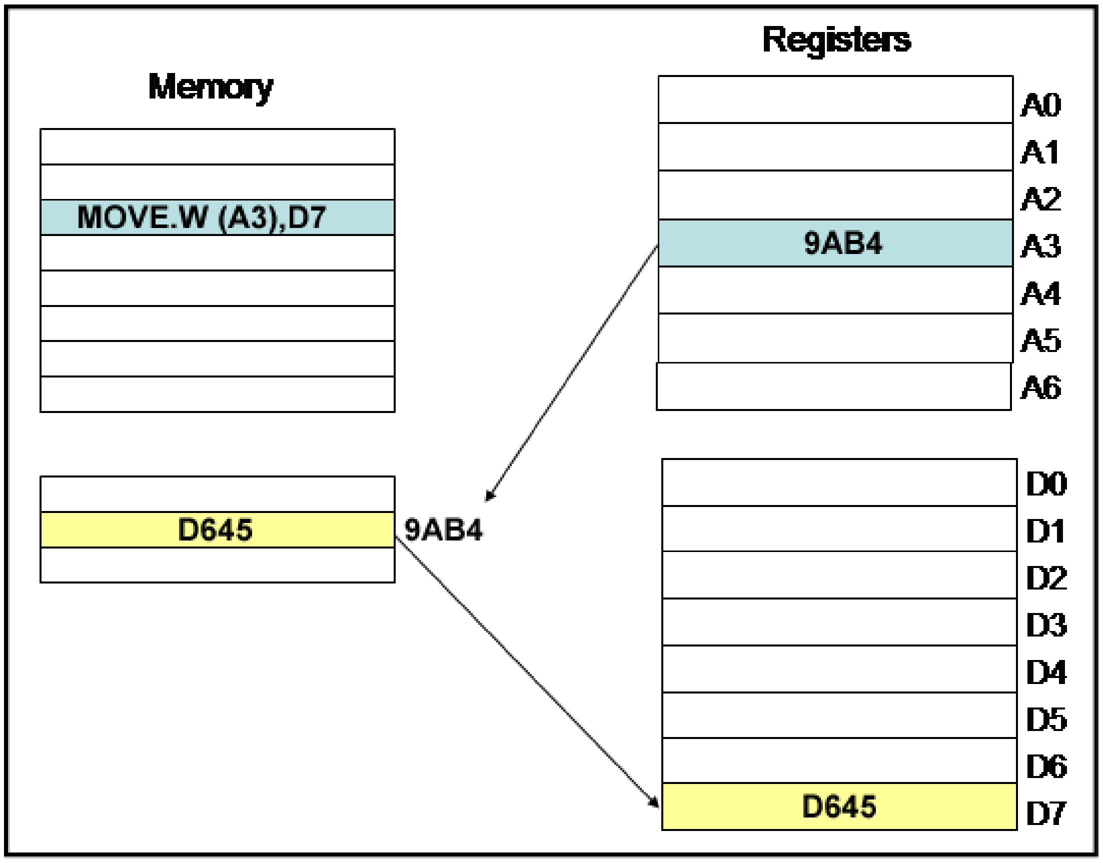
Mode 2: Address Register Indirect

The address register, A0 through A6, contains the memory address of the source or destination of the effective address. This is a pointer in C. We also call this indirect addressing because the contents of the address register is not the data we're interested in, it is a pointer to the data, so it allows us to indirectly access memory.

Therefore, with indirect addressing the contents of the address register is the address of the data we are trying to access.

Until now, we have always directly specified the source or destination memory location. Indirect addressing allows us to manipulate addresses by using the contents of an address register to specify the effective address of the operand. If you examine the assembly language code produced by a C compiler, you'll see that pointers directly translate to address register indirect addressing modes. Some processor architectures, such as those in the Intel 80X86 family, use indirect addressing almost exclusively.

In ***address register indirect addressing***, the instruction specifies that one of the 68000's address registers, A0 through A6, holds the memory address of the data we need. In writing the assembly language instruction, parentheses mean ***the contents of***. For example, the instruction, **MOVE.W (A3),D7** instructs the processor to load data register D7 with the contents of the memory location pointed to by address register A3. The source address register contains the address of the operand. The processor then accesses the operand pointed to by the address register and finally, the contents of the address register pointed to by A3 are copied to the data register. We can see this in the figure 8.4.



We can also interpret address register indirect addressing in terms of the behavior of the state machine. Consider these two snippets of code:

Snippet #1	Snippet #2
`MOVE.W   $0600,D0`	`MOVEA    #$600,A0`
`MOVE.W   D0,$4000`	`MOVEA    #$4000,A1`
	`MOVE.W   (A0),(A1)`

In Snippet #1 the opcode word is decoded to show that the source EA is an absolute word. This means that the processor must do a few things:

Go out to memory and read the next word of the instruction. It reads in $0600. It then places $0600 on the address bus and loads the contents of the memory location at address $0600 into register D0. The contents of memory location $0600 are unchanged.

In the next instruction, it sees that the destination EA is an absolute address. It then goes out to memory and reads the next word of the instruction, which is the value $4000. It then places $4000 on the address bus and writes the data stored in D0 out to memory address $4000.

The key is that the state machine must use the effective address mode in order to determine the value of the address location of the source or destination. A large part of how it sequences through the state machine is basically the determination of these real memory addresses from the address mode calculations. Once it has determined the real address value, the sequence to read or write to memory is always the same.

In snippet #2, the task seems slightly more complex, but it actually is much more efficient. The two MOVEA instructions are used to load the address registers, A0 and A1, with their initial values, then the process is much more efficient.

1. Place the contents of address register A0 onto the address bus and read the contents of the memory location into a temporary holding register inside of the processor.
2. Place the contents of address register A1 onto the address bus and write the contents of the temporary holding register out to that memory location.

Notice that in the above example once we loaded starting values into the address register we no longer concerned ourselves with what those values were. We could do all sorts of incrementing and decrementing of address values and our memory loads and stores would be handled appropriately.

I hope that you can see that address register indirect is a very powerful addressing mode. In fact, you'll probably use it more often than any other mode. The reason it is so important is that indirect addressing enables us to compute a memory location during program execution instead of being fixed when the program is assembled. Thus, if we are able to compute an address, then we can easily move up and down through tables and structures.

Mode 7, subclass 4: Immediate Addressing

The source value, preceded by the # sign, is the data. It is not a memory address. An immediate operand is also called a literal operand. We use the immediate addressing mode most often to initialize variables, or to provide commonly used numbers in the program. Keep these two important points in mind when trying to use immediate addressing.

1. The immediate addressing mode can only be used to specify a ***source operand*** because you cannot store a number (data) in a number.

2. You must place the pound sign, #, in front of the numeric value as an indicator to the assembler that this is an immediate value. Otherwise, it may be interpreted as a memory location. This may be a hard bug to find because you won't get an assembler error for specifying a memory location instead of a number.

For example, the instruction

```
MOVE.B #4,D0
```

uses a literal source operand and a register direct destination operand. The literal source operand, 4, is part of the instruction. The destination register, D0, is addressed using register direct addressing. Each operand of the instruction can use a different addressing mode. The effect of the instruction is to copy the literal value 4 to the data register, D0.

Mode 7, subclass 000: Absolute Addressing (word)

The memory location is explicitly specified as a 16-bit word. However, since the full address of the 68K architecture is 32 bits long, this addressing mode uses a 16-bit ***sign extended address.*** The most significant bit of the address operand is used to fill all the upper address bits from A15 . . . A31. If MSB = 0, then the address will be in the lower 32k of memory (A0 . . . A14). If MSB = 1, then the address will be the upper 32K of memory.

For example, if the 16-bit address is $B53C, or binary 1011 0101 0011 1100, then the "sign extended" 32-bit address is $FFFFB53C. If the 16-bit address is $7ABC, or binary 0111 1010 1011 1100, then the sign extended address is $00007ABC.

You might be wondering, "Why bother?" Well, for one thing, the absolute short, or word, form of the address saves memory space and is faster because it means that there is one less extension word fetch to make from memory. Also, most ROM code is placed in low memory and RAM code would be located in high memory. For example, the 68K Exception Vector Table is located in first 256 long words of memory.

Mode 7, subclass 001: Absolute addressing (long)

The memory location is explicitly specified as a 32-bit word. The actual address of the operand is contained in the two words following the instruction word. After reading the two address words from memory, the data to be operated on is then read from the external 32-bit memory address formed by concatenating the high-order word and low-order word to form the full memory address.

Since the 68K has only 24 external address bits, there exists a potential problem called ***aliasing.*** Aliasing is a result of inadvertently duplicating an address. Remember that a full 32-bit address may contain 256, 24-bit pages. The best solution is to try to keep all of the upper address bits, A24 through A31, equal to zero.

In direct or absolute addressing, the instruction provides the address of the operand in memory. Direct addressing requires two memory accesses to fetch the complete instruction. The first is to access the instruction and the second is to access the actual operand. For example, the instruction `CLR.B 1234` clears (sets to zero) the contents of memory location 1234.

Even though it seems pretty straightforward, absolute addressing is not used very often. In all but the most simple computer systems we need the flexibility to move code around in memory without regard to the absolute location that the code resides at. Absolute addressing prevents the code from being relocated and it slows the processor because multiple memory accesses are required in order to determine the memory address of the operand.

Mode 3, Address Register Indirect with Postincrement

The address register, A0 . . . A6, contains the address of the source or destination of the effective address. After the instruction is executed the contents of the address register is incremented by one. This is the 68K method of implementing the stack POP operation.

Mode 4, Address Register Indirect with Predecrement

The address register, A0 . . . A6, contains the address of the source or destination of the effective address. Before the instruction is executed the contents of the address register is decremented by one. This is the 68K method of implementing the stack PUSH operation. We'll discuss the stack PUSH and POP operation in a later section.

Mode 3 and mode 4 are examples of a general addressing method called ***auto-incrementing***. If the addressing mode is specified as (A0)+, the ***contents of the address register are incremented after*** they have been used. For example: Suppose that the address register, A3, contains the value $00009AB4. When used as an indirect pointer, it points to memory location $00009AB4. Assume that we're going to execute the instruction:

```
ADD.L    (A3)+,D4
```

In words, this instruction will add the contents of the source effective address to the contents of the destination effective address and replace the contents of the destination effective address with the new value.

The contents of register A3, <A3> = $00009AB4. Thus, the long word content of memory location $00009AB4 is added to the content of register D4 and the result is stored back into D4. The instruction is completed by incrementing the contents of address register A3. How much will it be incremented by? If you say 1, keep it to yourself. Since the operation is on a long word quantity, <A3> →<A3 + 4> , or $00009AB8. Thus, the size of the incrementing operation matches the size of the operand being manipulated.

If the instruction after the **`ADD.L (A3)+,D4`** instruction resulted in a branch back of the program, then the result would keep adding the contents of successive memory locations to D0. Auto-incrementing instructions are valuable because so much of memory is data in ordered lists, and as you'll see in a moment, they are also used to implement stack-based addressing operations.

Let's examine the use of the address register indirect with post-incrementing (whew!) instruction mode with a piece of example code. The following fragment of code uses address register indirect addressing with post-incrementing to add together five numbers stored in consecutive memory locations. Note the use of the instruction, LEA (load effective address).

This instruction is designed expressly for the purpose of placing an address into an address register. The program adds together five byte number values stored in memory.

Example

```
        ORG       $400
        MOVE.B    #5,D0      *Five numbers to add
        LEA       Table,A0   *A0 points to the numbers
        CLR.B     D1         *Clear the sum
Loop    ADD.B     (A0)+,D1   *Repeat loop, add number to total
        SUBQ.B    #1,D0      *Decrement loop counter
        BNE       Loop       *UNTIL all numbers added
        STOP      #$2700

Table   DC.B      1,4,2,6,5  *Some dummy data
        END       $400
```

The mode 3 and mode 4 addressing modes automatically increment the value in the address register after the operand is fetched from memory. The value in the register increments by one, two, or four bytes if the corresponding operation is on bytes, words or long words, respectively.

Let’s summarize the primary addressing modes.

- **Register direct addressing** is used for variables that can be held in registers.
- **Literal (immediate) addressing** is used for constants that do not change. Its primary use is to initialize variables.
- **Direct (absolute) addressing** is used to directly specify the address of variables that reside in memory. Although conceptually simple, it prevents programs from being relocatable.
- **Address register indirect addressing** is used when an address needs to be computed, or sequentially addressed.
- **Address register indirect with postincrement or predecrement** is used for sequential data manipulation or PUSH and POP operations to the stack.

The only difference between register direct addressing and direct addressing is that the former uses registers to store operands and the latter uses memory.

Let's look at a simple assembly language program which summarizes most of the concepts that we've just discussed. Assume that we have a sequence of bytes residing in successive memory locations beginning at a memory location labeled "data". Figure 8.5 is a flow chart of the algorithm.

In the program we assume that the string of bytes is terminated by a byte containing 00. We'll call this the NULL byte. It’s the same termination method used for character strings in C.

The program assumes that there can be a zero length string of numbers in memory, but the NULL byte must be present.

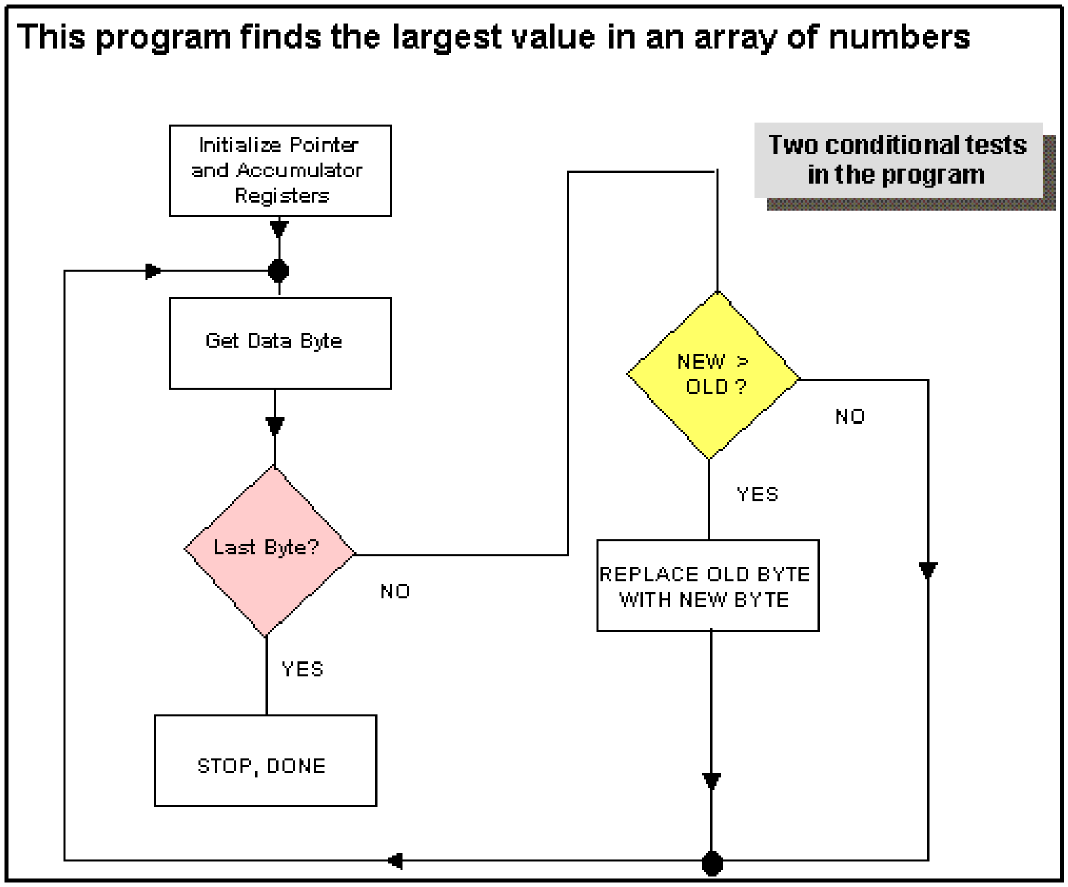

This program finds the largest value in an array of numbers
Initialize Pointer and Accumulator Registers
Two conditional tests in the program
Get Data Byte
NEW > OLD ?
NO
YES
Last Byte?
NO
REPLACE OLD BYTE WITH NEW BYTE
YES
STOP, DONE

Figure 8.5: Flow chart for an algorithm to find the largest number in a sequence of numbers

Here's the program:

```
*************************************************************************************
* Program to find the largest value in an array of bytes
*
* The array starts at memory location $1000 and is terminated by a null byte, $00
```

```
*
*********************************************************************************
```

```
 1 start     ORG      $400                   * Program origin
 2           LEA      data,A0                * Use A0 as a pointer
 3           CLR.B    D0                     * D0 will record the largest byte, zero it
 4 next      MOVE.B   (A0)+,D1               * Repeat loop, read a byte
 5           BEQ      exit                   * Is this the null byte?
 6           CMP.B    D0,D1                  * D1 - D0, If New > Old
 7           BLE      end_test               *
 8           MOVE.B   D1,D0                  * Then OLD = New
 9 end_test  BRA      next                   * Go back and get the next byte
10 exit      STOP     #$2700                 * Return to simulator
11
12           ORG      $1000                  * Data region
13 data      DC.B     12,13,5,6,4,8,4,10,0 * Sample data
14           END      $400                   * Program terminator and entry point
```

Let's analyze the program. Note that the numbers in the leftmost column shouldn't be placed in an actual program. They were placed there to aid in identifying each line of instructions. Also notice the header block enclosed in asterisks before the actual program code. This is used to describe the algorithm, much the way a comment block is placed at the beginning of a function in C++.

1. The ORG pseudo-opcode identifies the beginning of the program.
2. LEA (Load Effective Address) initializes address register A0, with the memory address of the data. In effect, we are assigning a pointer to the array variable, ***data***. We'll keep the pointer in the A0 register. In an earlier code snippet we used MOVEA instead of LEA to load a value into an address register. If you are setting up an address register for later use, LEA is the preferred instruction.
3. CLR.B is used to zero the contents of register D0 so that we can properly test for a larger value. Also, in case the array is zero length, we won't report a wrong value.
4. MOVE.B copies the byte pointed to by A0 into register D1. Since we are using the address register indirect with post incrementing mode, the contents of A0 are automatically increments to point to the next byte in the string ***after the data is retrieved.***
5. BEQ tests to see if the byte we retrieved is the NULL byte. If it is, the branch is taken and we go to the exit point of the program. If this test fails we fall through to the next instruction.
6. CMP.B checks to see if the new byte is larger than the currently largest byte.

7. BLE executes the result of the previous CMP.B instruction. If the new byte is not larger than the current byte in D0, it jumps over the next instruction and returns to the beginning of the loop. If it is larger, the next instruction is executed. Notice how lines 6 and 7 pair an instruction that compares two values with an instruction that branches based upon the result of the comparison.
8. MOVE.B replaces the value in D0 with the new largest value.
9. BRA always returns to the start of the loop.
10. STOP brings us back to the simulator.
11. ORG relocated the instruction counter to begin assembling code to be loaded in memory at address $1000 and above.
12. DC.B is the pseudo-opcode to load the sample data into memory at the memory location defined with the label "data".
13. END tells the assembler to stop assembling and to load the program to run at $400.

Assembly language and C++

Up to now, we've been considering assembly language in isolation, as if we're learning to program a new and cryptic programming language. That's sort of true. However, let's not lose sight of the fact that many of the compilers of high-level languages output assembly language. The assembly language output of these compilers is then assembled to object code. This means that the coding constructs that we've learned in our other programming classes must also translate to a comparable assembly language construct.

C and C++ already have built-in constructs for changing the program flow. **`IF/ELSE, WHILE, DO/WHILE, SWITCH, FOR,`** and function calls all can change the flow of the program. In assembly language we must build our own constructs using the assembly language instructions that are available to us: Branch, Jump, Jump to Subroutine (a function call in C).

Let's look at how some very familiar C++ constructs survive when we look under the hood and see what a compiler might do with them. First, recall that the assembly language comparison instructions (**`CMP, CMPI`** and **`CMPA`**) subtract one operand from the other in order to set the appropriate condition code flags, but do not save the result. Thus, if <D0> = 10 and <D1> = 10, the instruction:

```
CMP.B D0,D1
```

will subtract the contents of data register D0 (source register) from the contents of data register D1 (destination register). Since both registers contain the number 10, the subtraction will equal zero, with a result that Z=1. However, unlike the actual subtraction instruction,

```
SUB.B D0,D1
```

the contents of D0 and D1 are unchanged by the operation. Let's compare two code segments. The first, shown below, is a simple C++ ***IF*** construct:

```
int a = 3, b = 5;
if( a == b)
    { Execute this code };
else
    { Execute this code }
```

A compiler might convert this to this assembly language code segment:

```
             MOVE.L   #3,D0
             MOVE.L   #5,D1
             CMP.L    D0,D1
             BEQ       equal
not_equal    {This code executes}
equal        {This code executes}
```

Loop constructs in assembly language are similar to their C++ brethren. It is instructive to look at the loop construct in some detail because loops are among the most error-prone elements of assembly language programming.

Let's look at a simple **FOR** loop construct:

```
for ( int counter = 1; counter < 10 ; counter++ )
      {Execute these statements}
```

```
         MOVE.L        #1,D0                         *D0 is the counter
         MOVE.L        #10,D1                        *D1 holds terminal
                                                     value
for_loop CMP.B         D0,D1                         *Do the test
         BEQ           next_code                     *Are we done yet?
         {Execute some other loop instructions}      *Increment the
                                                     counter
         ADDQ.B        #1,D0
         BRA           for_loop                      *Go back
next_code{Execute the instructions after the loop}
```

The **DO/WHILE** construct is also commonly used in C++. The form of the instruction is:

```
DO
    {Execute these statements}
WHILE {The test condition is true}
```

The assembly language analog is shown below:

```
          MOVEA.W   #start_addr,A2      *Initalize the loop conditions
test_loop JSR       subroutine          *This is a function call in C++
          CMPI.B    #test_value,(A2)+   *Compare the new value
          BNE       test_loop           *The test condition is still true
          {Next set of instructions}    *This code follows the loop
```

Notice that the function call, **JSR** subroutine, happens at least once because this is a **DO/WHILE** loop. The instruction,

```
CMPI.B #test_value,(A2)+
```

is the heart of the equality test. Here the value, **test_value**, is tested against the value in memory pointed to by address register A2. After the instruction is executed, the value contained in register A2 is automatically incremented to point to the next memory location. Thus, automatically incrementing our memory pointer, A2, gives us the next value to test when we re-enter the loop.

Let’s do one more exercise in the analysis of an assembly language algorithm. In this case, we'll take a very simple assembly language coding example and successively refine it to make it as efficient as possible. In this case, efficiency will be measured by code compactness and execution speed. Here's the problem statement:

Write the data value, $FF, into memory locations $1000 through $1005, inclusive

Example #1 Brute Force Method

```
**************************************************************
* This program puts FF in memory locations $1000 through $1005
*************************************************************
* System equates

store_val    EQU     $FF         *Byte to load
pgm_start    EQU     $400        *Program runs here
stack        EQU     $2000       *Put stack here

             ORG     pgm_start
             LEA     stack,SP          *Initialize the stack pointer
             MOVE.B  #store_val,$1000  *Load first value
             MOVE.B  #store_val,$1001  *Load second value
             MOVE.B  #store_val,$1002  *Load third value
             MOVE.B  #store_val,$1003  *Load fourth value
             MOVE.B  #store_val,$1003  *Load fourth value
             MOVE.B  #store_val,$1004  *Load fifth value
             MOVE.B  #store_val,$1005  *Load sixth value
             STOP    #$2700            *Return to the simulator
             END     pgm_start         *Stop assembling here
```

Before we analyze the program, there is an instruction that might look strange to you. The instruction:

```
LEA    stack,SP      *Initialize the stack pointer
```

places the address of a variable that we've defined as ***stack*** (memory location $2000) into something called ***SP***. SP is the way that our assembler designates register A7 or A7', the stack pointer. We'll discuss the stack pointer in more detail later. For now, we'll accept on faith the need to locate the stack pointer somewhere in high memory, and we should do it as one of the first things that we do in our program. Also note that we use the ***Load Effective Address,*** LEA, instruction to do this.

Now, let's analyze the program. The program in example #1 runs, but it is far from efficient. All of our destination operands for the MOVE.B instructions are absolute addresses. We're wasting memory space for the instructions because we are exactly specifying the memory location for each data transfer. Suppose that we were moving 600,000 bytes instead of 6 bytes. Clearly that scenario would not work for this algorithm.

We can improve on Example #1 by using a data register to hold the data value, $FF, that we are moving to memory and we can also use an address register to point to the memory where we want to store the data. This saves us time and space because the data, $FF, is now available to us in a register, rather than having to retrieve it from memory each time. The disadvantage is that we must initially load some registers, which requires additional instructions. However, the overhead of loading the registers will be offset by the speed gained by using shorter instructions that execute in less time.

Example #2 Using registers

```
**************************************************************
* This program puts FF in memory locations $1000 through $1005
*************************************************************
* System equates
store_val       EQU       $FF        *Byte to load
pgm_start       EQU       $400       *Program runs here
stack           EQU       $2000      *Put stack here
start_addr      EQU       $1000      *First address to load

*Program starts here
                ORG      pgm_start
                LEA      stack,SP       *Initialize the stack pointer
                LEA      start_addr,A0 *Set up A0 as the pointer
                MOVE.B   #store_val,D0 *Put it in a data register
                MOVE.B   D0,(A0)       *Load first value
                ADDA.W   #01,A0        *Point to the next address
                MOVE.B   D0,(A0)       *Load second value
                ADDA.W   #01,A0        *Point to the next address
                MOVE.B   D0,(A0)       *Load third value
                ADDA.W   #01,A0        *Point to the next address
                MOVE.B   D0,(A0)       *Load fourth value
                ADDA.W   #01,A0        *Point to the next address
```

```
                 MOVE.B   D0,(A0)       *Load last value
              STOP     #$2700       *Return to the simulator
              END      pgm_start    *Stop assembling
```

This is better because each instruction that moves data only depends upon the contents of registers. Can we do better? Yes, we can eliminate the instruction,

```
ADDA.W       #01,A0
```

by using the auto-incrementing address mode, which is officially known as ***address register indirect with post-incrementing.***.

Example #3 Auto incrementing

```
**************************************************************
* This program puts FF in memory locations $1000 through $1005
*************************************************************
* System equates

store_val       EQU         $FF         * Byte to load
pgm_start       EQU         $400        * Program runs here
stack           EQU         $2000       *Put stack here
start_addr      EQU         $1000       *First address to load

*Program starts here

                ORG         pgm_start
                LEA         stack,SP         *Initialize the stack pointer
                LEA         start_addr,A0    *Set up A0 as the pointer
                MOVE.B      #store_val,D0    *Put it in a data register
                MOVE.B      D0,(A0)+         *Load first value, increment
                MOVE.B      D0,(A0)+         *Load second value, increment
                MOVE.B      D0,(A0)+         *Load third value, increment
                MOVE.B      D0,(A0)+         *Load fourth value, increment
                MOVE.B      D0,(A0)+         *Load fifth value, increment
                MOVE.B      D0,(A0)          *Load last value
                STOP        #$2700           *Return to the simulator
                END         pgm_start        *Stop assembling here
```

Is example #3 as good as we can do? This is an interesting question that requires some in-depth analysis. Why? Because for a simple program with 6 data write statements, trying to make the code more compact by using a loop construct might actually decrease performance rather than improve it. However, if we were moving a lot of data, this in-line algorithm would clearly not work.

The question about an in-line algorithm versus a loop construct is a very interesting one from the point of view of computer performance. Computers run most efficiently when they can execute large blocks of in-line code without needing to branch or loop. Compilers will often convert ***for*** loops to in-line code (if the number of iterations is a manageable value) in order to improve performance. We'll see the reason for this when we discuss pipelining in a later lesson.

Anyway, let's put in a loop construct and see if it improves things.

Example #4 The DO/WHILE loop construct

```
**************************************************************
* This program puts FF in memory locations $1000 through $1005
*************************************************************
* System equates

store_val       EQU         $FF         *Byte to load
pgm_start       EQU         $400        *Program runs here
stack           EQU         $2000       *Put stack here
start_addr      EQU         $1000       *First address to load
end_addr        EQU         $1005       *Last address to load

                ORG         pgm_start

                LEA         stack,SP       *Initialize the stack pointer
                LEA         start_addr,A0  *Set up A0 as the pointer
                LEA         end_addr,A1    *A1 will keep track of the end
                MOVE.B      #store_val,D0  *Put it in a data register
loop            MOVE.B      D0,(A0)+       *Load value and increment
                CMPA.W      A1,A0          *Are we done yet?
                BLE         loop           *No, go back
                STOP        #$2700         *Return to the simulator
                END         pgm_start      *Stop assembling here
```

We've managed to decrease our algorithm from 10 to 8 actual instructions (not counting pseudo-ops). That's better, but for this simple algorithm the value of the loop is hidden by the overhead of creating it. However, if the loop was going to execute 1,000,000 times or so, then writing the algorithm as in-line code would be prohibitive. Let's look at this algorithm with a ***for loop*** construct.

Example #5 The FOR loop construct

```
**************************************************************
* This program puts FF in memory locations $1000 through $1005
*************************************************************
* System equates

store_val       EQU         $FF         *Byte to load
pgm_start       EQU         $400        *Program runs here
stack           EQU         $2000       *Put stack here
start_addr      EQU         $1000       *First address to load
loop_ctr        EQU         6           *Number of time through the loop

                ORG         pgm_start

                LEA         stack,SP        *Initialize the stack pointer
                LEA         start_addr,A0   *Set up A0 as the pointer
                MOVE.B      #loop_ctr,D1    *D1 will keep track
                MOVE.B      #store_val,D0   *Put it in a data register
loop            MOVE.B      D0,(A0)+        *Load value and increment
                SUBQ.B      #01,D1          *Decrement counter
                BNE         loop           *Are we done yet?
                STOP        #$2700         *Return to the simulator
                END         pgm_start      *Stop assembling here
```

Counting instructions tells us that the ***for*** loop and the ***do/while*** loop constructs give us identical results. However, to really know if one would give us an improvement over the other we would really need to look at each instruction and count the actual number of machine cycles used. Then, and only then, would we know if the ***for loop*** was more or less efficient than the ***do/while*** loop.

One thing we can say at this point is that we're pretty good where we are. Can we get better? Yes, but it isn't obvious how we can do this. The answer lies with a more complex instruction, the DBcc instruction. Refer to the Motorola Programmer's Reference Manual for a complete discussion of this instruction. It's a difficult instruction to master, but it combines in one instruction the two instructions:

```
SUBQ.B          #01,D1          *Decrement counter
BNE             loop            *Are we done yet?
```

By using the DBcc instruction we can make our program even more compact.

Example #6 Using the DBcc instruction

```
**************************************************************
* This program puts FF in memory locations $1000 through $1005
*************************************************************
* System equates

store_val       EQU         $FF         *Byte to load
pgm_start       EQU         $400        *Program runs here
stack           EQU         $2000       *Put stack here
start_addr      EQU         $1000       *First address to load
loop_ctr        EQU         6           *Number of time through the loop

                ORG         pgm_start

                LEA         stack,SP       *Initialize the stack pointer
                LEA         start_addr,A0  *Set up A0 as the pointer
                MOVE.B      #loop_ctr,D1   *D1 will keep track
                MOVE.B      #store_val,D0  *Put it in a data register
loop            MOVE.B      D0,(A0)+       *Load value and increment
                DBF         D1,loop        *Decrement loop counter and
*                                          *branch if D1 != -1
                STOP        #$2700         *Return to the simulator
                END         pgm_start      *Stop assembling here
```

Well this is probably as good as it gets. The DBF instruction is very complex and difficult to master, but once you've mastered it, you can add it to your programmer's toolbox and use it when you need to shave a cycle or two. The instruction works this way. First, it either tests a condition code flag (DBcc) or is forced to always be false (DBF). Each time through the loop, the data register (in this case D1) is decremented, if the condition is false. The loop is exited under two conditions:

1- If the condition is true, or
2- The value in the data register = -1

Thus, the **`DBF D1,loop`** instruction keeps branching back to "loop" and decrementing D1 until <D1> = -1, then it exits the loop. Since we're using the version of the instruction that always tests to false, we're always going to branch back.

Let's review what we've just seen. In this sequence we've gone from a very straight-forward solution (take this data and put it here) to a much more compact and elegant solution. In the process we went from simple instructions and addressing modes to more complex instructions, addressing modes and algorithmic structures. In order to make the program as compact as possible, we used a rather complex instruction, DBF, in order to combine a register decrement operation with a test and branch operation. The following table shows the number of clock cycles and execution times (assuming a 16 MHz clock frequency) for the various instructions that we've used.

Instruction	Clock Cycles	Instruction Time (microseconds)
`MOVE.B #$FF,$1000`	**28**	**1.75**
`MOVE.B D0,$1000`	**20**	**1.25**
`ADDA.W #01,A0`	**12**	**0.75**
`MOVE.B D0,(A0)`	**8**	**0.5**
`MOVE.B D0,(A0)+`	**8**	**0.5**

There is a factor of 3.5 in the number of clock cycles required to move the byte value $FF into each memory location depending upon whether the data is loaded from memory each time and written to the specific memory location versus using the data and address registers to hold and manipulate the values.

Stacks and Subroutines

The concept of a stack is fundamental to how the computer manages its data and addresses. The stack is an example of a LAST-IN/FIRST-OUT or LIFO data structure. There are two special registers in the 68K architecture, A7 and A7’, that are dedicated to stack operations. A7 is the ***user stack pointer,*** and is referenced as SP, not A7, in assembly language instructions. A7’ is the ***supervisor stack pointer***.

In real life, the supervisor stack pointer would be reserved for operating system use, along with a set of special instructions that are usable in ***supervisor mode.*** Normally, our programs would run at a lower priority than the operating system. We would run our programs in ***user mode*** and would automatically access the A7 register for our stack pointer rather than the supervisor stack pointer. However, since the programs that we’re running on the simulator are rather simple and don't require the use of an operating system (other than your PCs operating system), we're always in supervisor mode and we'll be using the supervisor stack pointer for our operations. Both the supervisor mode and user mode refer to the stack pointer as SP, which stack pointer you get is determined by what mode you're in. Figure 8.6 summarizes the operation of the stack.

The reasons for the pre-decrement and post-increment addressing modes are now apparent. The stack pointer is always pointing to the memory address that is the top of the stack. When

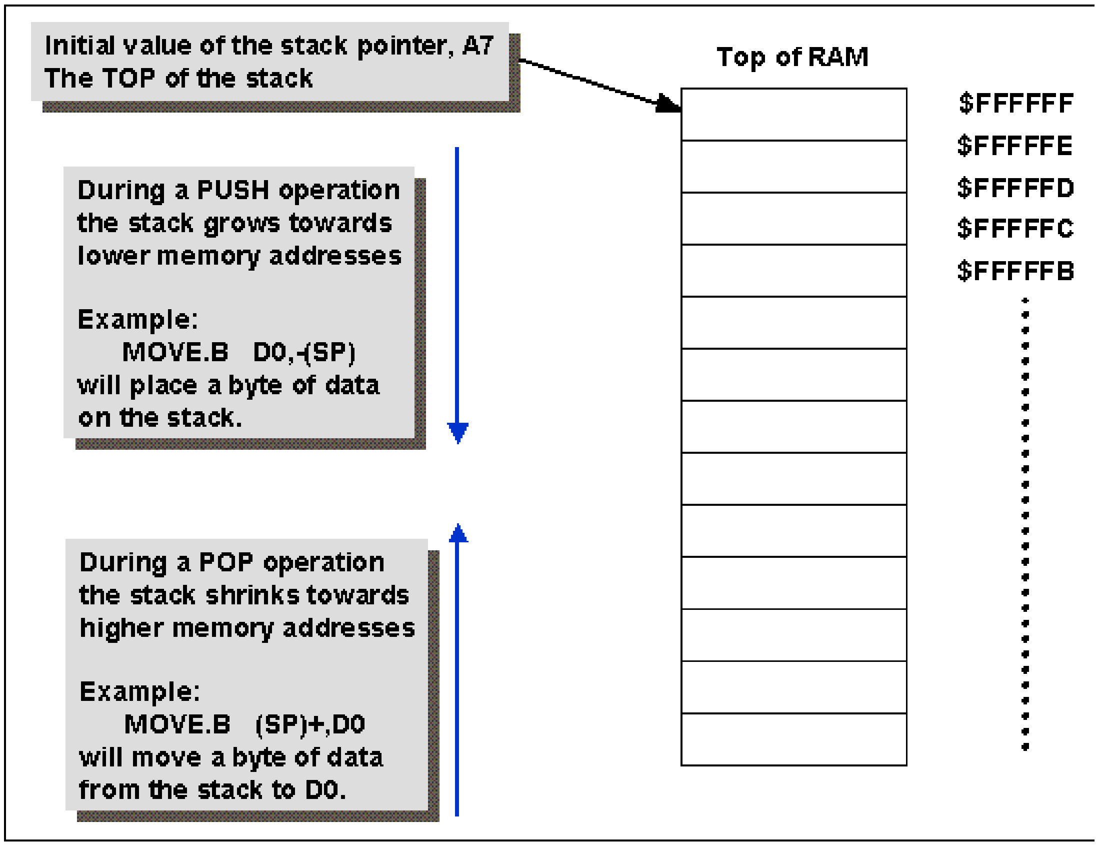

Initial value of the stack pointer, A7
The TOP of the stack
Top of RAM
$FFFFFF
$FFFFFE
$FFFFFD
$FFFFFC
$FFFFFB
During a PUSH operation the stack grows towards lower memory addresses
Example:
MOVE.B D0,-(SP)
will place a byte of data on the stack.
During a POP operation the stack shrinks towards higher memory addresses
Example:
MOVE.B (SP)+,D0
will move a byte of data from the stack to D0.

Figure 8.6: Operation of the 68000 stack pointer register, A7

additional data needs to be placed on the stack, the pointer must first decrement in order to point to the next available space, so that when the data storage occurs, the data is moved to free memory. The post-increment mode accesses the current memory location on the stack *and then* increments, so the SP is pointing to the previous item stored on the stack.

Now that we have shown how the stack implements the LIFO structure, we can move on to examine subroutines. Plain and simply, subroutines in assembly language are the equivalent of function calls in C. When the program encounters the ***jump to subroutine*** (JSR) instruction, the processor automatically *pushes* the long word address of the **next instruction** onto the stack and then jumps to the location specified in the operand. Since the program counter is always pointing to the next instruction to be fetched, the action of the JSR instruction is to first move the contents of the program counter to the current location of the stack pointer and then load the program counter with the new operand address. This effectively causes the program to “jump” to a new location in memory.

The ***return from*** subroutine (RTS) instruction is placed at the end of the subroutine. It will cause the stack to pop the return address back into the PC and the next instruction will be executed from the point in the program where the program jumped to the subroutine.

When you write a program in C or C++ the compiler manages all of the housekeeping that needs to go on when you do a procedure or a function call. In assembly language it is up to the programmer to:

- make sure that all subsequent stack pushes and pops line up,
- make sure that all resources used by the subroutine (registers and memory) are properly saved before the subroutine uses them and restored when the subroutine returns,
- decide on a mechanism for parameter passing between the subroutines and the main program.

In general, a subroutine will likely need to use register resources that are being used by the main program, or by other subroutines. The 68K processor has a mechanism to manage this. The **`MOVEM`** instruction is designed to quickly specify a list of registers that should be saved onto the stack upon the entry into the subroutine and restored upon exit from the subroutine.

By saving the registers that you intend to use in the subroutine upon entry, and then restoring them upon exit, the subroutine is able to freely use the registers without corrupting the data that is being used by other routines. Thus, it is generally not necessary to save all of the registers on entry into the subroutine, but it can’t hurt. You can always streamline the code once you decide which registers you aren’t using in your subroutine. Also, you will likely need to move parameters in and out of the subroutines, just as you do in C. However, it is up to you to decide how to do this, since there is no compiler there to force a convention upon you.

Thus, on entry to the subroutine, use the:

```
MOVEM   <register list>,-(SP)
```

to PUSH registers onto the stack, and on exit from the subroutine, use the

```
MOVEM (SP)+,<register list>
```

to POP registers from the stack.

One caveat (Darn those caveats!): It is very convenient to use registers to pass parameters into a subroutine and to use registers to return results from a subroutine. The C/C++ keyword ***return*** generally causes the compiler to put the result of the function call into a designated register and then return from the subroutine. Here's the "gotcha". If you intend to return a result in a register then you better not have that particular register as member of the group of registers that you save and restore upon subroutine entry and exit. Why? Because when you restore the register on exit you will write over the result that you wanted to return. Many students have burned the midnight oil trying to find that particular bug.

Creating a register list is made easy by the **REG** pseudo-op code assembler directive. REG means ***register range***. It allows a list of registers to be defined. The format is:

```
    <label>    REG    <register list>
```

Registers may be specified as a single register—A0 or D0—separated by slashes (i.e., A1/A5/A7/D1/D3), or register ranges may also be specified, such as A0–A3. For example, this will define a register list called “save_reg”

```
save_reg    REG    A0-A3/A5/D0-D7
```

Registers are automatically saved or restored in a fixed order, the ranges specified in the REG directive does not specify the order. Refer to your Programmer’s Reference Manual for a complete, although incomprehensible, explanation of the MOVEM instruction. Figure 8.7 summarizes the storing and restoring of resources when we enter and exit a subroutine.

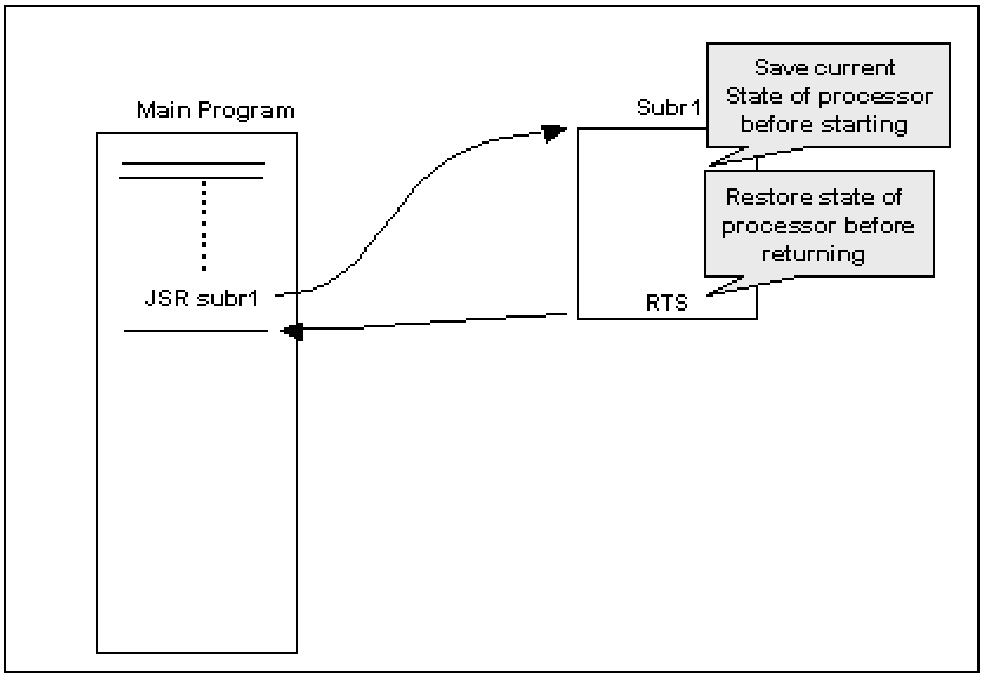

Main Program
JSR subr1
Subr1
RTS
Save current State of processor before starting
Restore state of processor before returning

Figure 8.7: Resource management with subroutines

Subroutines may be nested, just like function calls in C. It is important to keep in mind that subroutines require careful stack management because the return path depends upon having the correct sequences of return addresses stored on the stack. Subroutines should always return to the point where they were invoked; the next instruction after the JSR instruction.

However, the ***jump table*** is one of the few exceptions to this rule. We'll study the jump table in a later lesson. Figure 8.8 illustrates the nesting of subroutines.

Subroutines result in efficient coding because the same block of code can be reused many times. The alternative is to have all the code written in-line. Here are some guidelines for writing and using subroutines.

1. You must have the user stack established before attempting to execute a JSR instruction.
2. Locate the subroutines after the main part of your program, but before your data storage area.
3. Each subroutine should have a comment block header listing:
    a. Subroutine name
    b. What it does
    c. Registers used and saved
    d. Parameters input and returned
4. The first instruction line of the subroutine must have a label with the name of the subroutine. This is how the assembler is able to insert the effective address of the destination operand.
5. Save the registers that you'll use on entry to the subroutine.
6. Restore registers on exit from the subroutine.
7. Always return to point in the program from where the subroutine was called. In other words, don't push a new return address onto the stack and don't jump or branch from the subroutine to somewhere else, leaving the return address on the stack.
8. Nesting of subroutines is permitted, just like function calls in C.

A Sample Program: The Gory Details

We've spent a good bit of time analyzing the primary addressing modes and examining some programming concepts. It's time to get our hands dirty and examine a complete program that does something useful and puts together all of the concepts that we've learned so far. The program checks the integrity of the memory in the system and is widely used in many computer applications, especially at boot-up time when the system is going through its self-check. You computer does this every time you turn it on or press RESET. Here's the plan: let's walk through a sample memory test program to see how all of the elements that we've discussed so far are really used. Our goals will be to:

- Examine the structure of the assembly language program;
- See how comments are used;
- See how equates (EQU) are used to define constants;
- See how labels are used to define memory locations;
- Look at how the pseudo op directives (ORG, DS, DC, END) are used in practice;
- See how the Load Effective Address (LEA) command is used;
- Observe the various addressing modes used in the program;

- See how the compare instruction is combined with the branch instruction to test and modify program flow;
- Observe the header block on the subroutine; and,
- See how the subroutine is used and how parameters are passed.

As a way of introduction, let's discuss what the program is doing. It has been hard-coded to always test the memory region from $2000 to $6000, although the starting and ending addresses are somewhat arbitrary and the "equates" section allows us to easily change the region to be tested.

The program writes a byte value out to memory and then reads it back immediately. The value read back should be the same as the value written. If not, there could be a bad memory location, or perhaps a broken or shorted memory line. As a way to determine the cause of the problem, we use four different byte values: $00, $FF, $55 and $AA. These represent all data lines low, all data lines high, even data lines low and odd data lines high, and even data lines high, odd data lines low, respectively. We'll also keep track of the number of failures we detect by storing up to 10 address locations that failed the memory test.

The first section of the program includes the system equates. Consider this to be your header file. This is where you keep all of your #define statements. Notice that almost every constant or initial value is defined here and given a symbolic name. Also notice that for clarity, the maximum number of bad locations, ***maxcnt***, was defined as a decimal number. The assembler will convert it to hexadecimal for us.

```
***********************************************************
*
* Memory test program
*
* This is a program to test memory from byte address $2000
* to byte address $6000. It uses four test patterns, 00,
* $FF, $AA, $55
* and it can store up to 10 bad address locations.
*
***********************************************************
*
* System Equates
*
end_test     EQU      $11         * Test pattern terminator
test1        EQU      00          * First Test Pattern
test2        EQU      $FF         * Second test pattern
test3        EQU      $55         * Third Test Pattern
test4        EQU      $AA         * Fourth test pattern
st_addr      EQU      $2000       * Starting Address of Test
end_addr     EQU      $6000       * Ending address of test
stack        EQU      $7000       * Stack location
maxcnt       EQU      10          * Maximum number of bad addresses
```

The bold areas illustrate the use of the ***load effective address***, **`LEA,`** command to establish an address in the stack pointer or address register. The subroutine call instruction, ***`JSR`***, is italicized. The two symbolic variables, ***tests*** and ***bad_cnt*** are data storage locations at the end of the program.

```
          ORG      $400                   * Start of program
start     LEA      stack,SP               * Initialize the stack pointer
          CLR.B    D0                     * Initialize D0
          CLR.B    bad_cnt                * Initialize the bad address counter
          LEA      tests,A2               * A2 points to the test patterns
          LEA      bad_addr,A3            * Pointer to bad count storage
test_loop MOVE.B   (A2)+,D6               * Let D6 test the patterns for done
          CMPI.B   #end_test,D6            * Are we done?
          BEQ      done                    * Yes, quit
          LEA      st_addr,A0             * Set up the starting address in A0
          LEA      end_addr,A1            * Set up the ending address in A1
          JSR      do_test                * Go to the test
          MOVE.B   bad_cnt,D7             * Get the current count
          CMPI.B   #maxcnt,D7             * Have we max'ed out yet?
          BLT      test_loop              * Quit or keep going?
 done     STOP     #$2700                 * Return to the simulator
```

Notice that we saved the registers that we used in the subroutine. We did not need to save A0, A1 and A2 because they were used to pass in the address parameters that we were using.

```
****************************************************************
*
* Subroutine do_test
*
* This subroutine does the actual testing.
* A0 holds the starting address.
* A1 holds the ending address. A2 points to the test pattern to use in
* this test.
* Registers A5,D1 and D7 are used internally by the subroutine and are
* saved.
* This routine will test the memory locations from A0 to A1 and put the
* address of any failed memory locations in bad_addr and will also
* increment the count in bad_cnt. If the count exceeds 10
* the test will stop
*
**********************************************************************
do_test      MOVEM.W    A3/D1/D7,-(SP)   * Save the registers
check_loop   MOVE.B     (A2),(A0)        * Write the byte
             MOVE.B     (A0),D1          * D1 holds the value written
             CMP.B      (A2),D1          * Do the comparison
             BNE        error_byte       * Update counter
             BRA        next_test        * OK, test again
error_byte   MOVE.W     A0,(A3)+         * Store address and increment ptr
             ADDI.B     #01,bad_cnt      * Increment bad count location
             MOVE.B     bad_cnt,D7       * Have we maxed out?
             CMPI.B     #maxcnt,D7       * Check it
             BGE        exit             * Return, we're done.
next_test    ADDA.W     #01,A0           * Increment A0
```

```
             CMPA.W     A0,A1           * Test if we're done
             BGE        check_loop      * go back and test the next addr
exit         MOVEM.W    (SP)+,A3/D1/D7  * Restore the registers
             RTS                        * return to test program
```

The data storage region contains our test patterns and the reserved memory for holding the count and the bad addresses. Notice that the ***END*** directive comes at the end of all of the source code, not just the program code. The value defined by ***end_test*** is similar to the ***NULL*** character that we use to terminate a string in C. Each time through the test loop we check for this character to see if the program is done. Finally, note the label ***padding***. It was added to prevent a non-aligned access from occurring due to the fact that we have an odd number of bytes of data.

```
* Data storage region

tests     DC.B   test1,test2,test3,test4,end_test  * tests
padding   DC.B   00          * filler
bad_cnt   DS.W   1           * counter for bad     locations
bad_addr  DS.W   10          * save space for 10   locations
          END    $400        * end of program and  load address
```

Suggested Exercise

Carefully read the code and then build a flow chart to describe how it works. Next, create a source file and run the program in the simulator. In order to test the program, change the ending address for the test to something reasonably close to the beginning, perhaps 10 or 20 bytes away from the start. Next, assemble the program and, using the list file, set a breakpoint at the instruction in the subroutine where the data value is written to memory. Using the trace instruction write the data value to memory, but then change the value stored in memory before starting to trace the program again. In other words, force the test to fail. Watch the program flow and confirm that it is behaving the way you would expect it to. If you are unsure about why a particular instruction or addressing mode is used, review it in your notes or refer to your *Programmer's Reference Manual.* Finally, using this program as a skeleton, see if you can improve on it using other addressing modes or instructions. Please give this exercise a considerable amount of time. It is very fundamental to all of the programming concepts that we've covered so far.

Summary of Chapter 8

Chapter 8 covered:

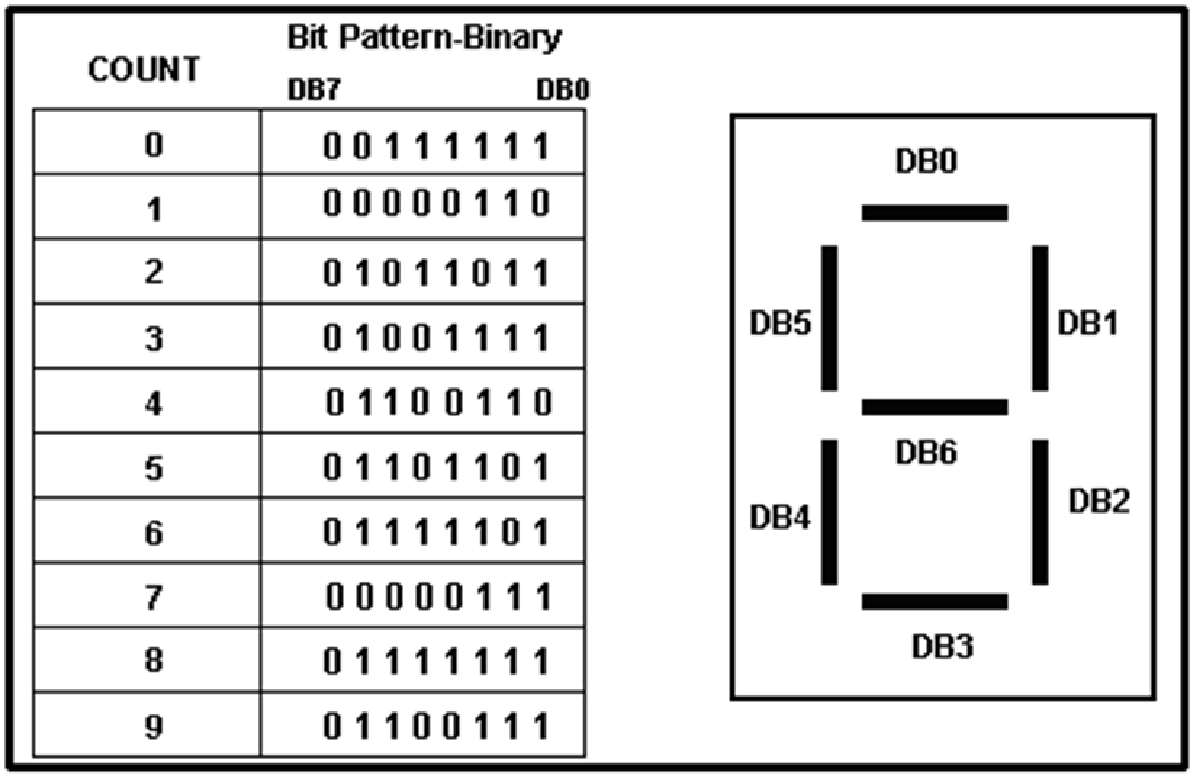
- How negative and real numbers are represented and manipulated within a computer.
- Branches and the general process of conditional code execution based upon the state of the flags in the CCR.
- The primary addressing modes of the 68K architecture.
- High level language loop constructs and their analog in assembly language.
- Using subroutines in assembly language programming.
- A detailed walk-through of an assembly language program to test memory.

- Bibliography and references

1- Alan Clements, ***68000 Family Assembly Language,*** ISBN 0-534-93275-4, PWS Publishing Company, Boston, 1994, pg. 29

Exercises for chapter 8

1- Shown below on the right is a schematic diagram of a 7-segment display. The table on the left represents the binary code that displays the corresponding digits on the display. Thus, to illuminate the number ‘4’ on the display, you would set DB1, DB2,DB5 and DB6 to logic level 1, and all the other data bits to logic level 0.



COUNT	Bit Pattern-Binary DB7 DB0
0	0 0 1 1 1 1 1 1
1	0 0 0 0 0 1 1 0
2	0 1 0 1 1 0 1 1
3	0 1 0 0 1 1 1 1
4	0 1 1 0 0 1 1 0
5	0 1 1 0 1 1 0 1
6	0 1 1 1 1 1 0 1
7	0 0 0 0 0 1 1 1
8	0 1 1 1 1 1 1 1
9	0 1 1 0 0 1 1 1

- The display is memory-mapped to byte address $1000.
- There is a hardware timer located at address $1002.
- The timer is started by writing a 1 to DB4. DB4 is a write-only bit and reading from it always gives DB4=0.
- When the timer is started DB0 (~BUSY) goes low and stays low for 500 milliseconds. After 500 milliseconds, the timer times-out and DB0 goes high again.
- DB0 is read-only and writing to it has no effect on the timer. All other bit positions may be ignored. The timer control register is shown schematically below:

Write a short 68K assembly language ***subroutine*** that will **count down to zero** from the number passed into it in register D0.B. The current state of the count down is shown on the seven-segment display. The count down rate is one digit every two seconds.

Notes:

- This is a subroutine. There is no need to ORG your program, set-up a stack pointer or use and END pseudo-op.
- You may assume that number passed-in is in the range of 1 to 9. You do not have to do any error checking. The subroutine is exited when the counter reaches 0.

2- Assume that some external device has transmitted a sequence of byte values to your computer. Along with the sequence of bytes the external device transmits a ***checksum value*** that you will use to determine if the byte sequence that you received is exactly the same as the byte sequence that was transmitted. To do this you will calculate the same checksum for the byte stream that you received and then compare it with the checksum that was transmitted to you. If they are equal, then it is extremely likely that there was no error in transmission.

Part A: Write an assembly language subroutine, **not a program**, which will calculate a checksum for a sequence of bytes located in successive memory locations. The checksum is simply a summation of the total value of the bytes, much like summing a column of numbers. The checksum value is a 16-bit value. Any overflow or carry beyond 16-bits is ignored.

1- The information that the subroutine needs is passed into the subroutine as follows:
    a. Register A0 = Pointer to the byte string in memory, represented as a **long word**.
    b. Register D0 = Checksum passed to the subroutine for comparison, represented as a **word** value.
    c. Register D1 = Length of byte sequence, represented as a **word** value.
2- Any overflow or carry generated by the checksum calculation past 16 bits is ignored. Only the **word** value obtained by the summation is relevant to the checksum.
3- If the calculated checksum agrees with the transmitted value, then address register A0 returns a pointer to the start of the string.
4- If the checksum comparison fails, the return value in A0 is set to 0.
5- With the exception of address register A0, all registers should return from the subroutine with their original values intact.

Part B: What is the probability that if there was an error in the byte sequence, it wouldn't be detected by this method?

3- What is the value in register D0 after the highlighted instruction has completed?

```
00000400 4FF84000      START      LEA       $4000,SP
00000404 3F3C1CAA                 MOVE.W    #$1CAA,-(SP)
00000408 3F3C8000                 MOVE.W    #$8000,-(SP)
0000040C 223C00000010             MOVE.L    #16,D1
00000412 203C216E0000             MOVE.L    #$216E0000,D0
00000418 E2A0                     ASR.L     D1,D0
0000041A 383C1000                 MOVE.W    #$1000,D4
0000041E 2C1F                     MOVE.L    (SP)+,D6
00000420 C086                     AND.L     D6,D0
00000422 60FE          STOP_HERE  BRA       STOP_HERE
```

4- Write a ***subroutine*** that conforms to the following specification. The subroutine takes as its input parameter list the following variables:

- A longword memory address in register A0, where A0 points to the first element of a sequence of 32-bit integers already present in memory.
- A 32-bit longword value in register D1, where the value in D1 is a search key.
- A positive number between 1 and 65,535 in register D0, where the value in D0 determines how many of the integer elements in the sequence pointed to by A0 will be searched.
- The subroutine returns in register D2 the value zero, if there is no match between the search key and the numbers in the sequence being searched; or the numeric value of the memory location where the first match is found

Note that once a match occurs there is no need to keep searching and you should assume that you have no knowledge of the state of the rest of the program that is calling your subroutine.

5- The memory map of a certain computer system consists of ROM at address 0000 through 0x7FFF and RAM at address 0x8000 through 0xFFFF. There is a bug in the following snippet of code. What is it?

Code Snippet:

```
MOVE.W      $1000,D0
MOVE.W      D0,$9000
LEA.W       $2000,A1
MOVEA.W     A1,A2
MOVE.W      D0,(A2)
```

6- Write a short 68K assembly language program that will add two 64-bit values together and then stores the result. The specifications are as follows:

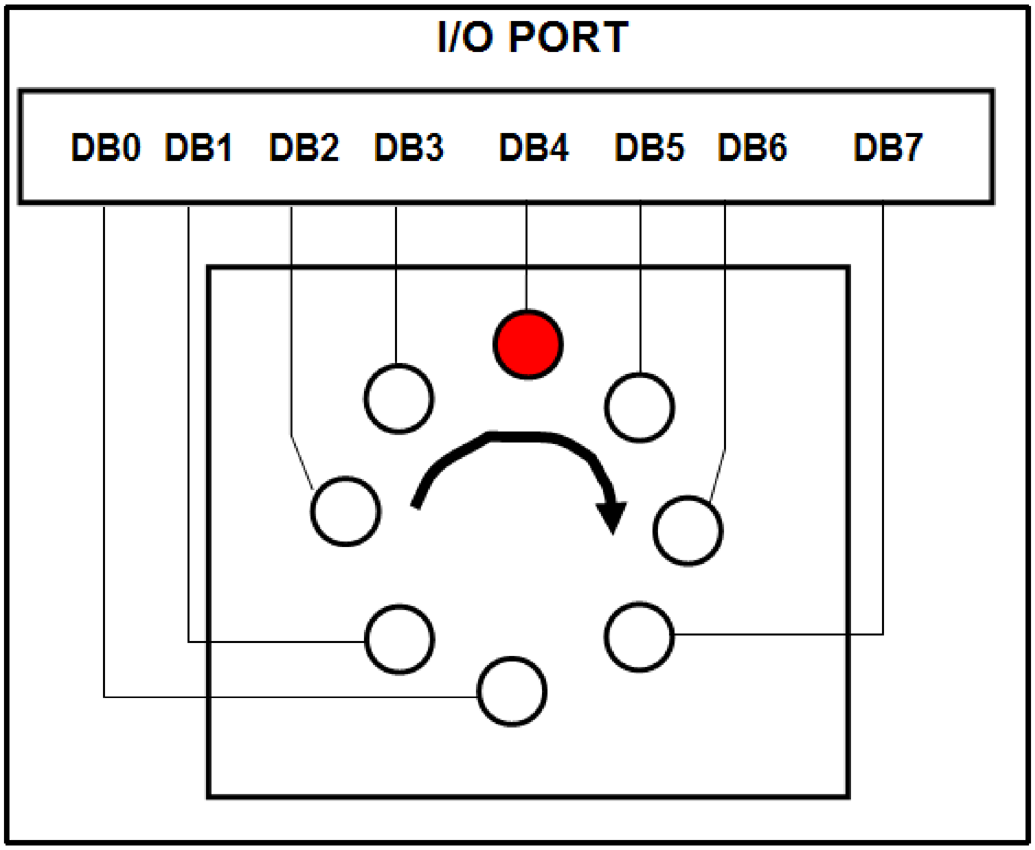
a) Operand 1: High order 32-bits stored in memory location $1000
b) Operand 1: Low order 32-bits stored in memory location $1004
c) Operand 2: High order 32-bits stored in memory location $1008
d) Operand 2: Low order 32-bits stored in memory location $100C

Store the result as follows:

a) High order 32-bits in memory location $1020
b) Low order 32-bits in memory location $1024

Any carry out generated by the high order addition in memory location $101F.

7- The diagram shown below represents a circular array of eight lights that are connected to an 8-bit, memory mapped, I/O port of a 68K-based computer system. Each light is controlled by a corresponding bit of the I/O port. Writing a 1 to a bit position will turn on the light, writing a 0 will turn it off.



Write a short 68K assembly language program that will turn on each lamp in succession, keep it on for two seconds, turn it off and then turn on the next lamp. The specifications are as follows:

- The 8-bit wide parallel I/O port is mapped at memory address $4000.
- There is a 16-bit wide time-delay port located at memory address $8000. DB0 through DB11 represent a count-down timer that can generate a time delay. Writing a value to DB0-DB11 will cause DB15 to go from low to high. The timer then counts down from the number stored in DB0-DB11 to zero. When the timer reaches zero, DB15 goes low again and the timer stops counting. Each timer tick represents 1 millisecond. Thus, writing $00A to the timer will cause the timer to count down for 10 milliseconds.
- DB15 is a read-only bit, writing a value to it will not change it or cause any problems.
- There is no interrupt from the timer, you will need to keep examining the memory location to see when the timer stops counting.

DB15				DB11											DB0
ST	X	X	X	T	T	T	T	T	T	T	T	T	T	T	T

ST = Timer status, 1=counting down, X = Not used
T = Timer countdown value

8- Write a short 68K assembly language subroutine that will send a string of ASCII characters to a serial port according to the following specification:

- The serial port is memory mapped as two successive byte locations at address $4000 and $4001. The actual port for sending and receiving characters is at address $4000 and the status port is located at address $4001.
- Data bit 0 (DB0) of the Status Port is assigned to indicate the state of the Transmit Buffer. When the serial device is ready to transmit the next character the Transmitter Buffer Empty signal, TBE, is true (TBE = 1). When the Transmit Buffer is empty the next character may be sent. Writing a character to address $4000 starts the serial data transmission and sets TBE = 0. The next character may be sent when TBE=1.
- On entry, the subroutine should save the values of any registers that it may use and restore these registers on exit.
- The location of the string to print is passed into the subroutine in register A0.
- The string is terminated with the null character, $FF.
- The stack pointer has already been initialized, there is no need to establish a stack pointer
- Assume that this is a polling loop, there is no interrupt occurring. You must continually test the state of TBE in order to know when to send the next character.

9- Write a program that fills all of memory between two specified addresses with the word pattern $5555 This is similar to what the block fill (BF) command does in the instruction set simulator, (ISS), but you will do it with 68K assembly language code. The memory region that you will fill is $2000 to $20FF, inclusive.

10- Write a memory test program that will be capable of testing memory ***as words*** (16-bits at a time) in the region from $00001000 to $0003FFFF. It should test up to, and including $0003FFFF, but not past it. A memory test program works as follows:

- Fill the every memory location in the region being tested with the word data value.
- Read back each memory location and compare it to the data value that you wrote. If they don't agree, then you have a bad memory location.
- Start testing the memory using two different test patterns: $FFFF and $AAAA.
- After you complete a memory test with one of the patterns, complement the bits (change 1's to 0's and 0's to 1's) and repeat the test with the new pattern. Thus, you'll be cycling through the test a total of 4 times using these patterns.
- Repeat the test one more time using the starting test pattern $0001. Use the ROL.L instruction to move the 1 bit one position to the left each time through the memory test until you've run the memory test 16 times, shifting the 1 to the left each time you repeat the test.
- Complement the test pattern that you just used and repeat the above bit shifting test.
- Here are the particulars for the assignment:

- The program should be ORG'ed to run at memory address $00000400.
- The program tests the memory from $00001000 to $0003FFFF, inclusive
- The stack pointer should be located at $000A0000.

- The starting memory test patterns are: $FFFF, $AAAA and $0001
- The test will fill all the memory region of interest with one of the test patterns. Next, it reads the pattern back and compares the value read to the value written. If you write the program in a way that writes a word to memory and then reads it back immediately then you are not "adhering to the specifications".
- The test is repeated for each of the two starting test patterns, their complement and the shifted bit pattern and finally, its complemented bit pattern.
- If an error is detected, the address of the memory location where the error occurred, the data written and the data read back is stored in memory variables.
- If more than one error occurs the program should store the total error count (number of bad locations found) and keep only the address and data information for the last error detected.
- You should allow for a count of up to 65,535 bad memory locations.

Discussion:

This program encompasses many of the fundamental aspects of assembly language programming. If you study the program, you'll see that it lends itself to using a subroutine to actually do the memory test. Imagine if you wrote it in C, what would it look like? How would you pass the testing parameters to the function?

- Don't forget to initialize the stack pointer!
- The program has several loops in it. What are they? How will you define the regions of memory to test? How do you know when you've done all of the tests? How do you know if you wrote all of the memory locations that you need to?
- Be sure that you understand how to use the pseudo-ops EQU, ORG, CRE, DC.L, DC.B, DC.W and DS.L, DS.W, DS.B, END
- Understand the instructions JRS, RTS, LEA and the addressing modes, (An) and (An)+ .
- This program can be done in less than 50 instructions, but you have to know what you are trying to do. The Easy68K simulator counts cycles. The program that runs in the least number of clock cycles, even if it has more instructions, is generally the more efficient one.
- Try coding it in stages. Once you've completed the flow chart for the program, write the assembly code for each block and test it. How do you test it? Well, you can assemble it. If it assembles properly, you've made some progress. Next, run it in the simulator and verify that it is doing what you want it to.
- When you test your program, one good programming trick is to use the EQUates to change the region of memory that your testing to only a few words, not the entire space. That way you can quickly walk through the code. So instead of testing from $00001000 to $0003FFFF, you test from $00001000 to $0000100A.
- Check to see what happens if a memory location has bad data. Change the data value in a memory location using the simulator after your program has filled it with the test pattern. Did your program catch it? Did it deal with it? The Easy68K simulator has a nice memory interface that you can access through the view window.

- This program is almost entirely comprised of three instructions, MOVE, Bcc, and CMP (CMPA). The addressing mode will probably be address register indirect because you'll constantly be writing to successive memory locations and reading from successive memory locations. The address register is an ideal place to hold the starting address, the ending address and the address of where you currently are in memory. To point to the next memory location, increment the address register's contents. You can do this explicitly by adding to it, or implicitly with the (An)+ addressing mode.
- Think about how you might terminate the test when you're using the ROL instruction. You could set up a counter, and then count 32 times. That would work. However, look closely at the ROL instruction. Where does the "1" bit go to after it shifts out of the MSB position? What instruction would test for that condition?

The general structure of your program should be as follows:
a- Comment header block: Tells what your program does.
b- System equates: Define your variables.
c- ORG statement: Your program starts here:
d- Main program code: Everything except subroutines is here.
e- STOP $#2700 instruction: This ends the program gracefully and kicks you back into the simulator.
f- Subroutine header block: All subroutines should have their own header block
g- Subroutine label: Each subroutine should have a label on the first instruction. Otherwise you can't get there from here.
h- Subroutine code: This does the work. Don't forget to keep track of what registers are doing the work and how parameters are passed.
i- RTS: Every subroutine has to return eventually.
j- Data area: All of your variables defined with the DC and DS pseudo ops are stored here.
k- END: The last line of the program should be END $400. This tells the assembler that the program ends here and that it should load at $400.

If all else fails, review the memory test programming example in the chapter. Finally, note that if you try to run your program under the various forms of Windows you may notice strange behavior. The program might seem to run very quickly if you run it in a small region of memory, but then seem to die if you run it in a larger region of memory. This is not your program's fault. It is a problem with Windows when it runs a console application.

Windows monitors I/O activity in a console window. When it doesn't see any input or output in the window, it severely limits the amount of CPU cycles given to the application tied to the window. Thus, as soon as your program begins to take some time to run, Windows throttles it even more.

There are several ways around it, depending upon your version of Windows. You could try hitting the ENTER key while your program is running. It won't do anything in the window, but it will fool the operating system into keeping your program alive.

Another trick is to open the PROPERTIES menu for the window and play with the sensitivity settings so that Windows doesn't shut you down.

Solutions to the exercises in chapter 8

1-

```
***********************************************************
*
* Subroutine timer:
*
* This subroutine counts down from a number between 1 and 9
* passed into it in register D0.B.
* The count-down rate is one digit every two seconds, using a
* 500 millisecond hardware timer located at address $00001002
* The seven-segment display is located at address $00001000
*
* No error checking is done in the routine
*
* All registers used are save upon exit.
***********************************************************
disp0       EQU   $3F         *Bit patterns for the display
disp1       EQU   $06
disp2       EQU   $5B
disp3       EQU   $4F
disp4       EQU   $56
disp5       EQU   $6D
disp6       EQU   $7D
disp7       EQU   $07
disp8       EQU   $7F
disp9       EQU   $67
trigger     EQU   $10         *This starts the timer
time_out    EQU   $01         *This test for done
display     EQU   $00001000   *Memory location of the display
delay       EQU   $00001002   *Memory location of the timer hardware
* Code begins here
timer       MOVEM.L     A0/A1/D1/D2/D3,-(SP)    *Save the registers on
entry
            LEA         patterns,A0       *A0 = ptr to display patterns
            MOVEA.L     #display,A1       *A1 = ptr to display
            MOVEA.L     #delay,A2         *A2 = ptr to time delay circuit
            CLR.L       D1                *D1 = index register
            MOVE.B      D0,D1             *Get index value
loop1       MOVE.B      00(A0,D1),(A1)    *Send pattern to the display
            CMPI.B      #00,D1            *Is D1 = 0 yet?
            BEQ         return            *Yes, go home
            MOVE.B      #4,D2             *Count down timer set-up

loop2       MOVE.B      #trigger,(A2)     *Start timer
loop3       MOVE.B      (A2),D3           *Get status
            ANDI.B      #time_out,D3      *Isolate DB0
            BEQ         loop3             *Keep waiting
            SUBQ.B      #1,D2             *Decrement D2
            BNE         loop2             *Go back
            SUBQ        #1,D1             *Point to next pattern
            BRA         loop1
return      MOVEM.L     (SP)+,D3/D2/D1/A1/A0    *Restore the registers
            RTS
```

```
patterns    DC.B        disp0,disp1,disp2,disp3,disp4
            DC.B        disp5,disp6,disp7,disp8,disp9
```

2-

```
***********************************************************
*
*  Subroutine: CHECKSUM
*  Description: This subroutine calculates a checksum for a
*               string of bytes stored in memory pointed to by address
*               register A0 and compares it with the checksum value
*               passed in register D0. The length of the string is
*               passed in register D1. If the calculated checksum
*               agrees with the transmitted value, then address
*               register A0 returns a pointer to the start of the
*               string. If the checksum comparison fails, the value
*               in the address register is set to 0. Any overflow
*               in the checksum beyond FFFF is ignored.
* Registers:    A0 = longword pointer to string in memory.
*               D0 = word value of checksum passed into subroutine
*               D1 = word value length of string.
*               Return values: All registers, with the exception of A0,
*               are returned with their original values intact.
*
*
***********************************************************
ZERO              EQU         0000
checksum    MOVEM.L A1-A6/D0-D7,-(SP) * Save the registers not changed
            MOVEA.L A0,A6             * Keep a local
            CLR.W   D2                * Use D2 as an accumulator
            CLR.W   D3                * Use D3 to hold byte operand
            CMPI.W  #ZERO,D1          * Test for 0 length string
            BEQ     exit              * We're done
loop        MOVE.B  (A0)+,D3          * Get value and advance pointer
            ADD.W   D3,D2             * Add and accumulate
            SUBQ.W  #1,D1             * Decrement counter
            BNE     loop              * Done?
            MOVE.L  A6,A0             * Restore A0
            CMP.W   D0,D2             * Are they equal?
            BEQ     exit              * OK
            LEA     ZERO,A0           * Not OK
exit        MOVEM.L (SP)+,D0-D7/A1-A6 * Restore the registers
            RTS                       * Go back
```

The probability of the checksum not detecting an error is 1 part in 2^{16}, or 1/65,536.

3- **<D0> = $0000002A**

4-

```
**************************************************************
*
* Subroutine : int_srch
```

```
*
* This subroutine searches a sequence of long-words in memory,
* starting at the address pointed to by A0. D2 returns the search
* results.
*
* Register usage:
*
* A0 = Pointer to search string
* D1 = Longword value to search for
* D0 = Number of elements to search for
* D2 = Returns zero or the address of the first search match
*
* Assumptions:
* D0 contains a positive number between 1 and 65,535.
* The user knows the length of the sequence in memory. No error
* checking will be done by the program.
*
*****************************************************************
int_srch    MOVEM.L     D3,-(SP)    *Save the registers I use
            CLR.L D3                *We'll use D3 as a 32-bit counter
            MOVE.W      D0,D3       *Load D3
loop        CMPI.L      #00,D0      *Is it zero?
            BEQ         exit        *D0 is = 0, so exit
            CMP.L       D1,(A0)+    *Check and advance A0
            BEQ         match       *They are equal
            SUBQ.L      #1,D3       *Decrement counter
            BRA         loop        *Go back for next test
match       MOVE.L      A0,D2       *Transfer address
            SUBQ.L      #4,D2       *Reset address
exit        MOVEM.L     (SP)+,D3    *Restore register
            RTS                     *Go home
```

5- The ROM is a read-only device. The last instruction, MOVE.W D0,(A2) is doing a write to ROM. This is incorrect.

6-
```
            org $400
start       CLR.B $101F
            MOVE.L      $1000,D0          *Get OP1, low order
            MOVE.L      $1008,D1          *Get OP2, low order
            MOVE.L      $1004,D2          *Get OP1, high order
            MOVE.L      $100C,D3          *
            ADD.L       D3,D2
            ADDX.L      D1,D0
            BCC         no_carry
            ADDQ.B      #1,$101F
no_carry    MOVE.L      D0,$1020
            MOVE.L      D2,$1024
            END $400
```

7-

```
delay       equ         2000        *2 second delay
```

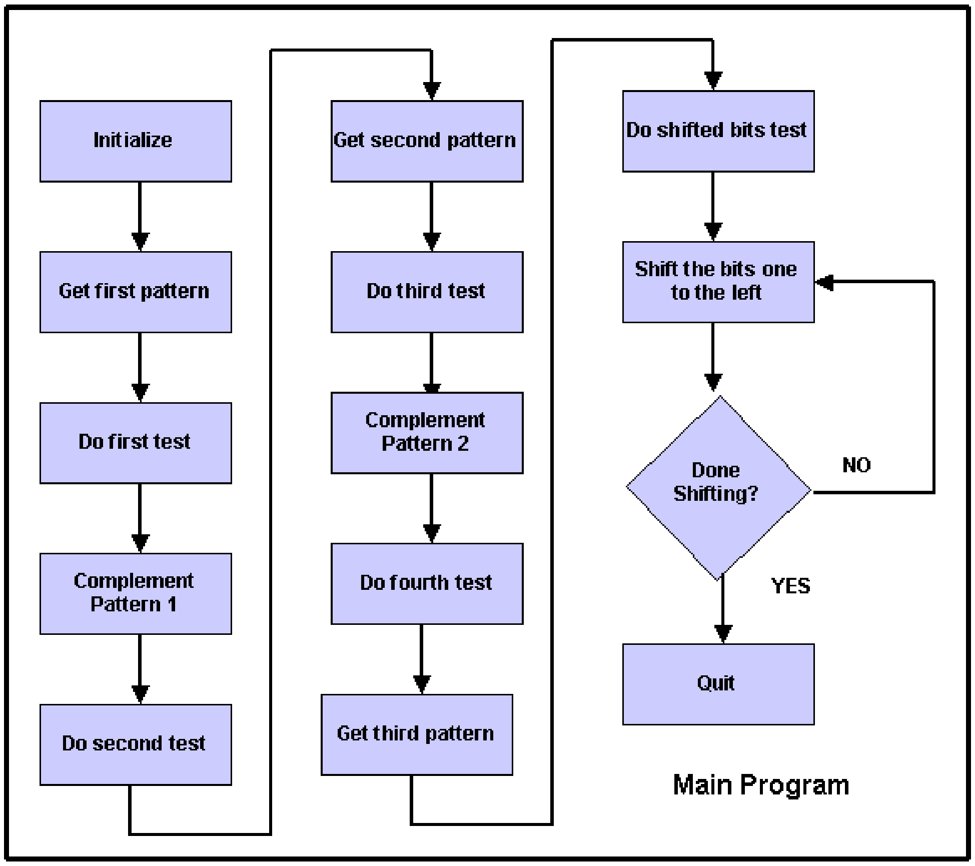
```
mask        equ         $8000       *timer status
bits        equ         01          *bit pattern
io_port     equ         $4000       *location of I/O port
timer       equ         $8000       *timer port

            org   $400

start       move.b      #bits,io_port     *load io port
loop        move.w      #delay,timer      *load timer
            move.b      io_port,d0        *get port
            rol.b       #bits,d0          *roll it left
            move.b      d0,io_port        *put it back
            move.w      #delay,timer      *set delay
wait        andi.w      #mask,timer       *check status
            beq         wait              *not zero, keep waiting
            bra   loop                    *do it again
            end $400
```

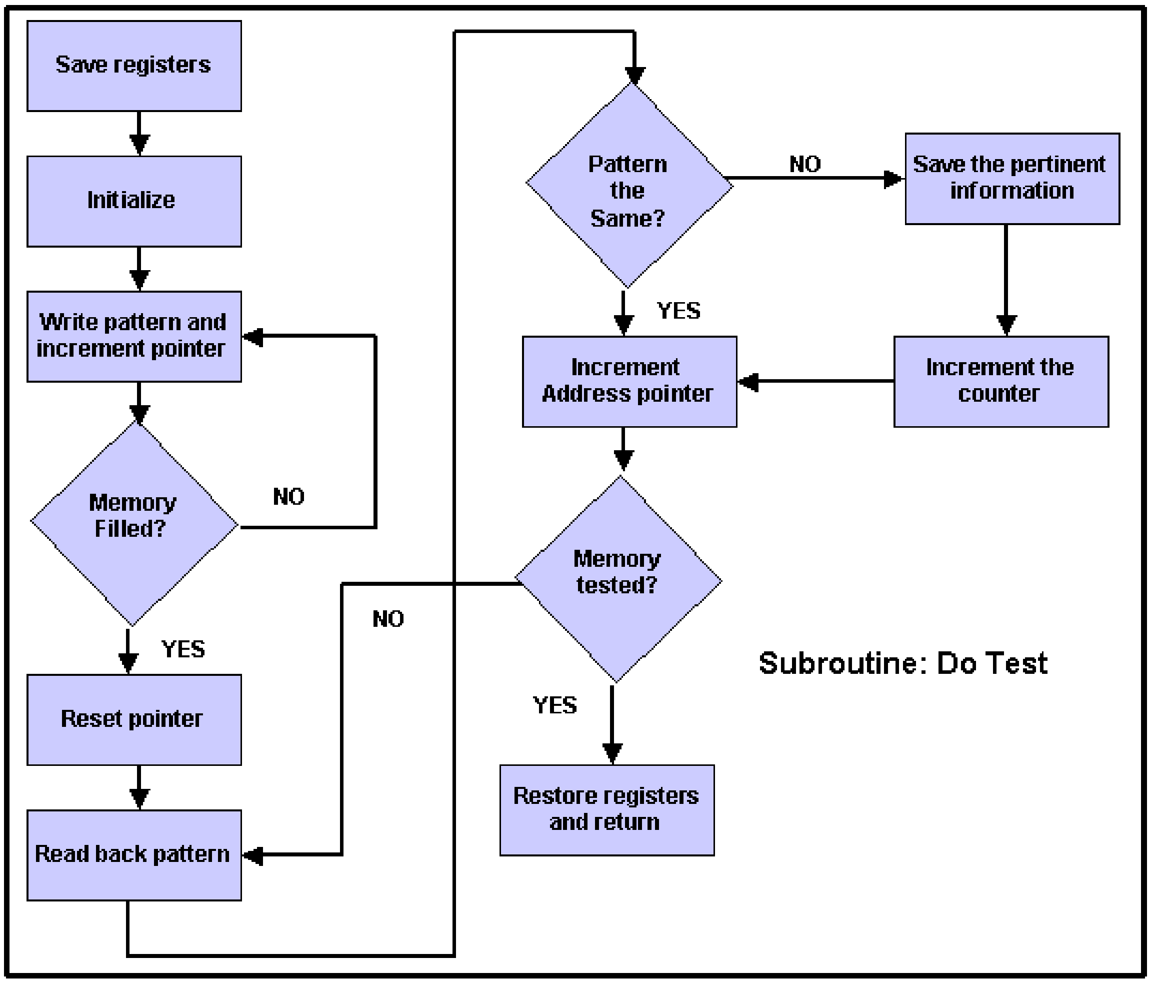
8-

```
***********************************************************************
* Subroutine Send_String
* Sends a string of byte characters to a serial port
* A pointer to the data is passed in address register A0
* The data string is terminated with the null character, $FF
* The data is sent from D0 and the status is checked in D1
* The subroutine saves all registers used
**********************************************************************
* Equates for subroutine
Xmit        EQU   $4000       *Data Port for Serial I/O
Status            EQU   $4001 * Status Port for serial I/O
EOS         EQU   $FF         * End of String Character
TBE         EQU   01          * Transmitter buffer empty mask
reg_list    REG   D0/D1       * Saved registers

**********************************************************************
Send_Str    MOVEM.W           reg_list,-(SP)    *Save registers
Byte_loop   MOVE.B            (A0)+,D0          *Get a byte
            CMPI.B            #EOS,D0           *Is it FF?
            BEQ               Exit              *If yes, exit
Xmit_loop   MOVE.B            Status,D1         *Get status byte
            ANDI.B            #TBE,D1           *Empty?
            BEQ               Xmit_loop         *Not yet. Keep waiting
            MOVE.B            D0,Xmit           *Ship it
            BRA               Byte_loop         *Do it again
Exit        MOVEM.W           (SP)+,reg_list    *Get ready to leave
            RTS
```

9-
```
************************************************************
*
* This is a program to fill memory with the word pattern $5555
*
**************************************************************
            OPT     CRE
fill_st     EQU     $00002000     * Start of block to fill
fill_end    EQU     $000020FF     * Last address to fill
pattern     EQU     $5555         * Fill pattern

start       EQU     $400          * Program begins here

            ORG     start
            LEA     fill_st,A0    * Load starting address
            LEA     fill_end,A1   * Load ending address
            MOVE.W  #pattern,D0   * Load pattern to write
loop        MOVE.W  D0,(A0)+      * Move it and advance pointer
            CMPA.L  A1,A0         * Are we done yet?
            BLE     loop          * No? Go back and repeat
            STOP    #$2700        * Pops us back to the simulator
            END     start
```

10- The flow charts for the main program and the subroutine are shown below:





```
*********************************************************************
*
* Memory test program
*
*********************************************************************

* System equates

pattern1     EQU     $AAAA         * First test pattern
pattern2     EQU     $FFFF         * Second test pattern
pattern3     EQU     $0001         * Third test pattern
st_addr      EQU     $00001000     * Starting address of test
end_addr     EQU     $0003FFFF     * Ending address of the test
stack        EQU     $000A0000     * Location of the stack pointer
word         EQU     2             * Length of a word, in bytes
byte         EQU     1             * One byte long, NO MAGIC NUMBERS!
bit          EQU     1             * Shifting by bits
exit_pgm     EQU     $2700         * Simulator exit code
data         EQU     $500          * Data storage region
start        EQU     $400          * Program starts here
```

```
* Main Program

         OPT      CRE            * Turn on cross references
         ORG      start          * Program begins here

         LEA      stack,SP       * Initialize the stack pointer
         LEA      test_patt,A3   * A3 points to the test pattern to use
         LEA      bad_cnt,A4     * A4 points to bad memory counter
         LEA      bad_addr,A5    * A5 points to the bad addr location
         LEA      data_read,A6   * A6 points to data storage
         CLR.B    bad_cnt        * Clear bad address count
         MOVE.W   (A3)+,D0       * Get current pattern, point to next one
         JSR      do_test        * Run first test
         NOT.W    D0             * Complement bits for next test
         JSR      do_test        * Run second test
         MOVE.W   (A3)+,D0       * Get next pattern
         JSR      do_test        * Run third test
         NOT.W    D0             * Complement bits for fourth test
         MOVE.W   (A3),D0        * Get last pattern
shift1   JSR      do_test        * Run shift test
         ROL.W    #bit,D0        * Shift bits
         BCC      shift1         * Done yet? No go back
         MOVE.W   -(A3),D0       * Get test pattern 3 again
         NOT.W    D0             * Complement test pattern 3
shift2   JSR      do_test        * Run the test
         ROL.W    #bit,D0        * Shift the bits
         BCS      shift2         * Done yet? If not go back
done     STOP     #exit_pgm      * Quit back to simulator

*************************************************************************
*
* Subroutine: do_test
*
* Performs the actual memory test. Fills
* the memory with the test pattern of interest.
* Registers used: D1,A0,A1,A2
* Return values: None
* Registers saved: None
* Input parameters:
* D0.W = test pattern
* A4.L = Points to memory location to save the count of bad addresses
* A5.L = Points to memory location to save the last bad address found
* A6.L = Points to memory location to save the data_read back and data
* written
*
* Assumptions: Saves all registers used internally

************************************************************************


do_test   MOVEM.L   A0-A2/D1,-(SP)  * Save registers
          LEA       st_addr,A0      * A0 points to start address
          LEA       end_addr,A1     * A1 points to last address
          MOVE.L    A0,A2           * Fill A2 will point to memory
fill_loop MOVE.W    D0,(A2)+        * Fill and increment pointer
```

```
          CMPA.L    A1,A2           * Are we done?
          BLE       fill_loop
          MOVE.L    A0,A2           * Reset pointer
test_loop MOVE.W    (A2),D1         * Read value back from memory
          CMP.W     D0,D1           * Are they the same?
          BEQ       addr_ok         * OK, check next location
not_ok    MOVE.L    A0,(A5)         * Save the address of the bad location
          ADDQ.W    #byte,(A4)      * Increment the counter
          MOVE.W    D1,(A6)+        * Save the data read back
          MOVE.W    D0,(A6)         * Save the data written
          SUBQ.L    #word,A6        * Restore A6 as a pointer
addr_ok   ADDQ.L    #word,A2        * A2 points to next memory location
          CMPA.L    A1,A2           * Have we hit the last address yet?
          BLE       test_loop       * No, keep testing
          MOVEM.L  (SP)+,D1/A0-A2   * Restore registers
          RTS                       * Go back
* Data Space

          ORG       data
test_patt DC.W      pattern1,pattern2,pattern3 * Memory test patterns
bad_cnt   DS.W 1                    * Keep track of # of bad addresses
bad_addr  DS.L 1                    * Store last bad address found here
data_read DS.W 1                    * What did I read back?
data_wrt  DS.W 1                    * What did I write?

          END        start
```

If you examine the code you'll see that there are no magic numbers. All numeric values used as operands are given symbolic names with the EQUates so that each instruction is as readable as possible.