

Notes on M68000 Programming

Physics 116B - D. Pellett

June 10, 1999

Introduction

You should have become acquainted with the M68000 assembly language and assembler directives by studying the MAS manual and working through some of its examples, as assigned previously. These notes further illustrate the following basic concepts needed to understand M68000 assembly language programming and processor operation in general:

1. assembly language structure and directives
2. relationship between program instructions or data and their representation in the computer memory
3. use of typical data transfer, integer arithmetic and logical instructions and their variations
4. use of external subroutines for keyboard, screen and file I/O
5. use of branch instructions to make repetitive calculations (loops)
6. accessing an array in successive memory locations using an address register (one form of indirect addressing)
7. characteristics of the Macintosh operating system for accessing data and code
8. how to make and use our own "local" subroutines
9. use of the run time stack for subroutine calls, returns and arguments

As a concrete example, we will develop a program to read 20 ASCII characters one by one from the keyboard, print them as a single string and add up the characters corresponding to decimal integers. We will also look at some of the details of the actual machine code (binary ones and zeros) produced by the assembler.

A Simple Program

We begin with a simple program to input one ASCII character and store it in a memory location. Note the use of assembly language directives, MAS subroutines and actual assembly language instructions. The `getchar` routine places the binary representation of the ASCII character entered from the keyboard in register D0. The `stop` routine terminates the program gracefully rather than causing the system to hang up at the end, requiring a reboot. Since these routines are not part of this program, it is necessary to specify their names in the `xref` directive. One byte of storage has been set aside at the memory location labeled `msg` by the `ds.b` directive.

If you input 'a', you will find that the byte stored is \$61, the representation of 'a' in the American Standard Code for Information Interchange (ASCII). A chart of ASCII characters and their hexadecimal representation is included at the end of this note.

```
; Program to read and store a single ASCII character
      xref      getchar, stop      ; indicates that these are external
start:  jsr      getchar           ; getchar puts the character in d0
      move.b   d0,msg            ; move the character to memory
      jsr      stop
      data
msg:    ds.b    1                 ; set aside one byte of storage
      even                               ; add byte to fill to word boundary
      end
```

A Simple Program Yields Complex Code

In this section, I will examine the actual machine code yielded by the three-line program, above. We will get a hint of why the M68000 is in a class of processors called complex instruction set computers (CISC). (*You may prefer to skip over this section for now and come back after getting an overview of the other programs.*)

I ran this program using the debugger. The labels, `start` and `msg`, refer to actual memory addresses, which are assigned when the program is loaded into memory. On entry into the debugger, the *program counter* (PC) contained \$00717252. The PC is a processor register which keeps track of where the currently executing instruction resides in memory. The location of `start` was \$00717252, so the PC was set to begin execution of the program.

The actual machine language instructions produced by the assembler were displayed starting at memory location \$00717252:

Memory address	Instruction
\$00717252:	\$4eba
\$00717254:	\$045c
\$00717256:	\$1b40
\$00717258:	\$0004
\$0071725a:	\$4eba
\$0071725c:	\$042a

Let's decode this and see what's going on.

First Instruction: jsr getchar

The first instruction should be JSR. If you look up the JSR instruction in the Motorola Programmer's Reference Manual (MPRM) excerpts, you find that it starts with a 16 bit word beginning with the byte, \$4e. The second byte of this word is composed of the bits %10mmrrr, where %mmm and %rrr refer to the mode and register of the "effective address" which can be decoded using the table given in the description of the instruction. So this is indeed JSR, and the second byte is \$ba = %10111010, so the addressing mode is %111 with register %010, meaning (d_{16}, PC). Whew!

But we're not done yet. Just what is meant by "addressing mode (d_{16}, PC)"? This means that the entry to the subroutine is equal to the contents of the PC at the time of execution plus the (sign-extended) 16 bit *displacement*, given in the subsequent 16 bit word, namely \$045c. According to the instruction description, the PC contents at the time is the address of the extension word, so the jump is to the address, \$717254+\$045c=\$7176b0, which should be the first instruction in the subroutine, `getchar`. The MAS program has set these things up for us through the `xref` directive when the assembled program and libraries were loaded in the computer memory at run time.

You have just seen an example of a technique called *indirect addressing*. The instruction does not contain the address (or in other cases, operand) itself in this mode. Instead, it tells how to get the address (or operand) from somewhere else: in this case, a location determined by a 16 bit displacement relative to the PC contents. In general, when we use indirect addressing, we indicate it with the use of parentheses: (d_{16}, PC), for example.

Two other possible types of addressing for the JSR instruction are absolute short and absolute long, in which the entry point address is given in the next one or two 16-bit words immediately following the first one. The 16 bit absolute short address is sign-extended to 32 bits before use to avoid ambiguity. This means if the left-most bit of the 16 bit word is 1 (negative, using two's complement notation), the 16 bit extension to 32 bits consists of a string of 1's, so the negative two's complement number is the same, only 32 bits long instead

of 16. If the leftmost bit is zero, then 0's are used instead. These and other addressing modes are described in Chapter 2 of MPRM. Table 2-4 lists 18 different addressing modes. We will not need to use *all* of them in this course.

Macintosh programming rules out use of absolute addresses for labels so code can be *relocatable*. A series of instructions can be placed anywhere in memory without changing the code. This is possible because all addressing is *indirect* relative to the PC for instructions and relative to register A5 for data (as we will see next). The operating system is then free to move code around to free up blocks of memory in a multiprogram environment.

Second Instruction: `move.b d0,msg`

The first word of this instruction is \$1b40 or %0001101101000000. From the MPRM, we see that the most significant byte is indeed %0001 for MOVE.B. The next 12 bits give the effective address (EA) of the destination (6 bits) and the source 6 bits), respectively (see pp. 4-116 - 4-118 of MPRM).

The destination EA is coded as %101101, meaning (d_{16}, A_n) where A_n is A5 (%101). Again, we have indirect addressing. In this case, it is relative to the address in register A5 with the displacement given in the next word of the instruction, namely \$0004. A5 contains \$0070ee80 so the instruction will store the data in location $\$0070ee80 + \$0004 = \$0070ee84$. This is indeed the address assigned to `msg`. Note how the address was specified relative to the contents of A5 rather than an absolute address. All program data will be stored in a single memory area starting at the address in A5 (the *base address* for the data area). For things to work properly in this system, **you must not alter the contents of A5**.

The source EA is coded as %0000000, meaning register D0, as expected.

Last Instruction: `jsr stop`

Here we see \$4eba again, as in the first instruction. This is a JSR with indirect addressing relative to PC with 16 bit displacement given in the next word, \$042a. The jump will be to $\$0071725c + \$042a = \$00717686$, which should be the entry to the `stop` subroutine.

A Program With a Loop and an Array

Now we modify this program to input 20 characters. We have to expand the storage and make a loop to perform the instructions to read and store 20 times. The storage area will begin at the location labeled `msg`. I chose not to give a different name to each character. Each can be found by its location relative to the first address: we have a one-dimensional array of characters. I also chose to set aside another location, `nchar`, for the number of characters, making the code easier to modify. Note that `nchar` must be less than or equal to the number of bytes set aside for `msg`. I have allowed 100 locations for `msg`, since space is not at a premium.

We use the following generally useful techniques to accomplish this in the program below.

1. We keep the memory address for the current character in A1.
2. We keep the number of characters remaining to be read in D1.
3. We use *indirect addressing* in the MOVE.b instruction to store the character in the proper address (the address in A1).
4. We use an indirect address mode (*postincrement*) which automatically increments the address in A1 appropriately after the character is stored.
5. We use the DBRA instruction to subtract one from (*decrement*) D1 at the end of the loop, test if we are done, and if not, repeat the loop.

```
; Program to read and store 20 ASCII characters
        xref      getchar, stop      ; indicates that these are external
start:   lea      msg, a1             ; put address of msg in a1
        move.w   nchar, d1           ; number of bytes to read
        jmp      enter              ; enter loop at end
loop:    jsr      getchar            ; getchar puts the character in d0
        move.b   d0, (a1)+           ; move the character to memory
enter:   dbra    d1, loop            ; subtract 1 from d1 and see if done
        jsr      stop               ; goes here when d1 equals -1
        data
msg:     ds.b    100                 ; set aside 100 bytes of storage
nchar:   dc.w    20                 ; number of characters to read
        end
```

Use of the GETCHAR Instruction at Run Time

The GETCHAR instruction uses an *input buffer*, which can be a bit confusing in operation. It does not send characters to the program from the keyboard until the return

key (entered as the ASCII character, CR = \$0D) is pressed. The buffer is filled up to the point where the CR is entered, then control is passed to the program, which processes each character in sequence from the buffer. For example, if you press ‘a’ followed by CR, the characters ‘a’ (\$61) and CR (\$0d) will be entered in the buffer and the program will put them in the `msg` array, counted as two characters. If you enter ‘aaaa’ CR, you get 4 a’s and a CR, counted as 5 characters. If you just want characters with no CR’s in your array, enter all 20 in line followed by CR. The debugger will not pause for input until it has run out of characters to send in its buffer, but data is being entered in the program loop anyway.

More Details About the Instructions

The LEA instruction, “load effective address,” computes the absolute address for `msg` by adding the 16 bit displacement for `msg` to the contents of A5 (the base address of the data area) and puts it in A0.

The value of `nchar` was preassigned to be 20 in the `dc.w` directive so the `MOVE.w` instruction puts this number in D1.

You may wonder why we jump to `enter` rather than just proceeding into the character reading loop. The `dbra d1,loop` instruction subtracts 1 from the number in D1 and branches to `loop` if the result is not -1. It goes to the next instruction if the result in D1 is -1. If D1 starts at 1, for example, the DBRA command would have to execute twice before terminating (reaching -1). In order to keep `nchar` equal to the number of times through the loop (*iterations*), we jump directly to the DBRA instruction to enter the loop (the `jmp enter` instruction). This introduces the possibly useful feature that if we ask for 0 characters to be read (by initializing `nchar` to 0), the loop terminates immediately without trying to read anything.

A further note: DBRA (“decrement and branch”) is the same as DBF, a special case of the general command, DBcc (“test condition, decrement and branch”). The DBcc command tests for a certain condition in the condition code register (the `cc` in DBcc) before decrementing and looking for -1. See pp. 4-90,91 of MPRM for details. These codes would be set as a result of a previous instruction, such as `CMP`. If the test is true, the loop terminates (*i.e.*, the next instruction is executed rather than branching). But here, we just want to count loops so we set the condition for termination in the initial test to “false” (DBF, a.k.a. DBRA). DBcc only operates on the lower 16 bits of the register used, so this scheme allows a maximum of \$FFFF = 65535 iterations.

The instruction, `move.b d0,(a1)+`, shows how to specify the address register indirect addressing with postincrement described earlier. Using parentheses, `(a1)`, indicates that the number in D0 is to be put *in the address given in A1* rather than in A1 itself. Then adding the `+` after the closing parenthesis, `(a1)+`, says to increment A1 afterwards so that the next time through the loop, the address will be that of the next character. Since this is `MOVE.b`, the incrementing adds 1. If it were `MOVE.w`, A1 would be incremented by 2, etc.

A Program with a Loop, an Array, Comparisons, Arithmetic and a Subroutine

Now we put together the entire program. The goal is to input 20 characters, sum up any integers we find, print the resulting character string and print the sum. Here is a structured list of the things we need to do (a flow chart would also be appropriate):

- We do the necessary initialization.
- Then for each of the 20 characters, we
 - read the character
 - put the character in the `msg` array.
 - check if it represents a number in the range 0-9
 - * if so, add it to a running total
- Now that we have all the input, we print the array on the screen as a character string
- print the sum of the integers
- and stop.

The testing for integers and summing will be done by a local subroutine. The idea is that the ASCII characters for the numbers 0-9 are in the range \$30 to \$39 (after eliminating the left-most parity bit, which could be either 0 or 1, by ANDing the character bit-by-bit with \$7F). The `CMP` and `Bcc` instructions are used to check that the number is in the proper range.

If the character is a number, all that remains to be done is to strip off the higher order bits by bitwise ANDing the character with \$000F and add it to the sum.

The act of using `AND` instructions, as above, to select ranges of bits in a word and setting others to zero is called *masking*.

The new instructions and constructs in the program are `CLR`, `AND`, `CMP`, `BLT`, `BGT`, `RTS` and immediate addressing (signified by `#`). In immediate addressing, a constant is imbedded in the instruction itself, as in `cmp.b #$30,d0`. The use of `CMP` in conjunction with `Bcc` needs a separate discussion, as does the actual mechanism of a subroutine call and return.

If you run this program, be sure to review the information in the previous section on how the characters get transferred to the program from the input buffer.

```

; Program to read and store 20 ASCII characters and sum up any integers
        xref    getchar, strout, decout, newline, stop
start:   lea     msg,a1           ; put address of msg in a1
        move.w  nchar,d1        ; number of bytes to read
        clr     d2              ; clear the register for the sum
        jmp     enter           ; enter loop at end
loop:    jsr     getchar         ; getchar puts the character in d0
        move.b  d0,(a1)+        ; move the character to memory
        jsr     addit           ; subroutine to test and add integers
enter:   dbra   d1,loop         ; subtract 1 from d1 and see if done
; now output the information
        lea     msg,a0          ; set up for outputting character string
        move.w  nchar,d0
        jsr     strout          ; output the string
        jsr     newline
        move.w  d2,d0          ; output the sum
        jsr     decout
        jsr     newline
        jsr     stop           ; end of program
; Subroutine addit tests ASCII characters to see if they represent numbers
; and if so, adds them to the sum in d2
addit:   and.b   #$7F,d0        ; mask off parity bit of character
        cmp.b   #$30,d0        ; see if it is less than $30
        blt     skip           ; if so, skip to return statement
        cmp.b   #$39,d0        ; see if it is greater than $39
        bgt     skip           ; if so, skip to return statement
        and.w   #$000F,d0      ; get the number
        add.w   d0,d2          ; add to sum in d2
skip:    rts                    ; return from subroutine

        data
msg:     ds.b   100             ; set aside 100 bytes of storage
nchar:   dc.w   20             ; number of characters to read
end

```

Branch Instructions and Signed Comparison Branches

Branch instructions (Bcc) check the condition codes, which take on meaning in the context of the instruction which set the codes. We will consider the case when the codes are set by execution of a CMP instruction, as in the program above. Some branch instructions and the condition codes they test are given in the next table. Note that the branch must be to a statement label, not an absolute address.

Instruction	Case for branch
BEQ <label>	Z=1
BNE <label>	Z=0
BMI <label>	N=1
BPL <label>	N=0
BVS <label>	V=1
BVC <label>	V=0
BCS <label>	C=1
BCC <label>	C=0

Some other variations of Bcc with more complex condition code tests are useful with CMP (or CMPI). The CMP instruction (as in `cmp.b <source>, <destination>`) compares two numbers by subtracting the destination from the source and setting the condition codes accordingly. For example, if the result is zero, the Z bit is set (see Sec. 1.1.4 of MPRM for details). The computer does this in such a way that the numbers themselves remain unchanged. Some other handy Bcc tests for use after a CMP instruction are:

Instruction	Case for branch
BGE <label>	<code>destination ≥ source</code> (<code>destination - source ≥ 0</code>)
BGT <label>	<code>destination > source</code> (<code>destination - source > 0</code>)
BLE <label>	<code>destination ≤ source</code> (<code>destination - source ≤ 0</code>)
BLT <label>	<code>destination < source</code> (<code>destination - source < 0</code>)

These must be performed before another instruction changes the condition codes. The instruction descriptions tell which codes each affects. For example, MOVE affects N, Z, V and C. Conversely, MOVEA does not affect the condition codes at all, allowing addresses to be moved to address registers without affecting the results of a previous calculation.

Note the awkward inversion of the notation: BGE branches when *second* argument \geq *first* argument, for example.

Use of Subroutines, the Run-Time Stack and the User Stack Pointer

In the previous example, we put some of our code in a subroutine, `addit`. The transfer to the subroutine from the main program (“subroutine call”) was accomplished with the `JSR` command. At the end of the subroutine, the `RTS` command returned us, as if by magic, to the instruction following the original `JSR`. This magic was accomplished through use of the *run-time stack*. The stack is an area in memory set aside for temporary storage of addresses and data. The name derives from its organization: you push items onto it or pop them off from one end only, like a stack of papers on a desk. But on the M68000, the stack grows downward from the bottom rather than upward from the top. The last item pushed in is the first item to pop out, so it may also be called a LIFO (last in, first out) list. Register A7 is used as the user stack pointer (SP) and *contains the last occupied address*.

Let’s look at the use of the stack in a subroutine call and return. The `JSR` command description states that `JSR <destination>` first makes room on the stack for the longword (4-byte) *return address* by subtracting 4 from the SP. It then places the current contents of the program counter, PC, in the address *now* contained in SP. At this point, PC contains the address immediately following the `JSR`, which is in fact the return address, and this is what is placed on the stack: the address of the next instruction to execute after the subroutine is finished.

At the end of the subroutine, we issue an `RTS`. This takes the 4-byte number from the stack at the location whose address is in SP, *i.e.*, the return address, and places it in the PC. `RTS` then adds 4 to the SP and returns control to the next instruction, which is found at the location in PC, namely the instruction just after the original `JSR`. Also, the SP now contains the address it had before the original `JSR`. The calling program takes up exactly where it left off, but the subroutine has now accomplished its task.

The beauty of this scheme is that as long as there is sufficient memory for the stack, we can call other subroutines inside our first subroutine, which themselves may call other subroutines, and still get back to the original calling point unharmed, as long as each subroutine behaves properly in its use of the stack and the `RTS` command.

We can also use the stack for our own purposes, as long as we carefully store and retrieve items in the appropriate LIFO order, so as to return SP to its original state at the end and not overwrite valid data remaining on the stack.

For example, we can use the stack to save the contents of registers before calling a subroutine which alters them. A single 16-bit word could be placed on the stack with `move.w d1, -(sp)`. The SP is first decremented by two bytes (since it is `move.w`), then the contents of the lower 16 bits of D1 is placed at that address on the stack. To restore the data to D1, `move.w (sp)+, d1` is executed. The `MOVEM` (move multiple) command can be used to put a whole list of registers on the stack: `movem.l d0-d3/a0, -(sp)` places D0, D1, D2, D3 and A0 on the stack as longwords and `movem.l (sp)+, d0-d3/a0` restores them. A description of `MOVEM` from MPRM is included with these notes. The status register can be saved and restored using the commands `MOVE` to `CCR`, `MOVE` from `SR` or `RTR` (return and restore condition codes). Whether the calling program or the user subroutine itself saves the registers used is a matter of choice.

To preserve word alignment, byte data is stored in a 16 bit word on the stack (*i.e.*, the SP is automatically decremented or incremented by two for byte-size operands moved to or from the stack). Stack use is more efficient if longword alignment is preserved, but this is neither automatic nor mandatory. It will be the case if only longwords are used, or other items are used in pairs.

The stack can also be used to pass arguments to and from subroutines. The stack pointer (A7) can then be used with the “address register indirect with index” addressing mode to access these arguments on the stack. One must be careful here not to interfere with the normal subroutine call and return.

The M68000 has a number of commands for maintaining additional stacks and for other stack operations. There are examples in Ch. 7 of the book by Ford and Topp.

Note that if a subroutine does not use local storage (nothing in the data area, for example), and keeps its arguments and saved registers on the stack, it could, in principle, call itself. Of course, such a recursive subroutine must provide a way to end this recursion. There must be some condition for which it simply returns rather than calling itself yet again. A recursive subroutine for computing factorials is given as an example in Ch. 10 of Ford and Topp.

Exercises

1. Improve the final program to give input request and output description messages as in the MAS examples.
2. Have the program input the number of characters to read from the keyboard. Check that the number will fit in the buffer. If not, output a warning and input only the number of characters the buffer will hold.
3. Have the program convert any upper case letters to lower case.
4. Follow in detail the use of the run time stack in the call to the subroutine.