# Programming the Amiga in Assembly Language:

*COMPLEX FUNCTIONS*

*by William P. Nee*

There are usually several way to improve the performance of your source code when writing an assembly language program. The most obvious is to consider a rewrite of the mechanics of the program. For example, there are several ways to write a Mandelbrot program. The simplest sets every point to some color while the more complicated ones outline the set and then automatically fill in the color. The code can be very long but will result in a much quicker program. This type of rewriting, however, is very individualized and will be unique for each program.

**RELATIVE CODE**

There are some general procedures you can consider, though, with any program. First, try to write your code with all labels or variables as "position independent". That is, your code does not refer to the variable location but rather it's distance from where you are in the program at that point—much like a branch would do. For example, instead of putting the contents of LABEL2 into register d0, put the information, say, 300 bytes ahead into d0. This is done by referencing the variable as LABEL(PC); the "pc" means position counter and will cause the assembler to compute a distance rather than location. But, what does this do to our code? Let's write a short program and look at the assembled code.

```
START    MOVE.L   ACROSS,D0          2039 0000 000C
         MOVE.L   DOWN(PC),D1    223A 0008
         RTS
ACROSS   DC.L     0
DOWN     DC.L     0
         END
```

Notice the difference in Line 1 and Line 2. With just a small change in the source code, we saved two bytes; while this isn't very much, if this code was within an iteration loop of 1000 times, that would be a savings of 2000 bytes and an increase in speed.

Unfortunately, only the source operand can be written in PC format, not the destination operand; you can't write MOVE.L  #1,DATA1(PC), at least not with the current crop of assemblers. But there is a technique that allows you to reference any variable as a distance from a fixed variable whose address is stored in a register, usually a4 or a5. This will add one more line of code when you put the initial variable address in the register. Let's store the start of all our variables, call it V, in register a5. Now any location after V can be referred to as LABEL-V(A5); again, this computes a distance not an address. Now look at a variation of our first program.

```
START    LEA      V(PC),A5          4BFA 0020
         MOVE.L   ACROSS(PC),D0  203A 001C
```

```
MOVE.L     DOWN-V(A5),D1  222D 0004
MOVE.L     #1,LABEL1      23FC 0000 0001 0000 002A
```

```
                MOVE.L    #2,LABEL2-V(A5)       2B7C 0000 0002 000C
                MOVEQ     #3,D2                 7403
RTS
V:
ACROSS          DC.L      0
DOWN            DC.L      0
LABEL1          DC.L      0
LABEL2          DC.L      0
LABEL3          DC.L      0
                END
```

The first line stores the address of our variable storage area in a5. Be sure not to change a5 during the loop in which you're using this technique. The code length for the next two lines is the same so using LABEL(PC) or LABEL-V(A5) creates identical length code for the source operand. The next two lines show how the code length can be decreased using the destination operand as a register offset. By the way, the address of V is the same as the address of ACROSS; I find it easier to add V at the beginning and any macro using this technique won't have to be rewritten.

Notice that the last line shows how MOVEQ will create shorter code that MOVE.L for immediate values between -128 and +127. In fact, MOVEQ #0 is even quicker than the code CLR. And it's quicker to double or halve a number with a LSL or LSR rather than use MUL or DIV.

The quickest way to use variables is to assign them to registers at the start of the program. Since this does limit the register's use, you probably want to do this with only two or three of your most used variables such as a counter or across/down. You can still use those registers outside a loop but leave them alone during any loop when they are being used as their assigned register. Now let's see how our code could look.

```
ACROSS          EQUR      D4
START           LEA       V(PC),A5                    4BFA 0016
                MOVE.L    DOWN(PC),D0                 203A 0012
                MOVE.L    ACROSS,D1                   204
                MOVEQ     #3,D0                       7003
                MOVE.L    D0,LABEL1-V(A5)                   2B40 0004
                MOVE.L    LABEL2-V(A5),LABEL3-V(A5)   2B6D 0008 000C
                RTS
V:
DOWN            DC.L  0
LABEL1          DC.L  0
LABEL2          DC.L  0
LABEL3          DC.L  0
                END
```

Since ACROSS was equated to register d4 at the start of the program, we don't reserve space for it at the end. You could, and probably should, also equate DOWN to a register; I didn't just to show you the difference in code length. You can also see that it's quicker to move an immediate number into a register and then move that register into a label instead of moving the

number directly into the label.

To demonstrate some of these techniques I wrote a program, using an Amiga 3000, that iterates the complex functions SIN(Z), COS(Z), SINH(Z), COSH(Z), and EXP(Z). Points are set when the real or imaginary parts of Z exceed specific values. More on this later.

First, I rewrote the DPMATHMACROS.I code or, more specifically, the MOVEDP portion of it. Now if a label is going to be moved to a register, the macro uses LABEL(PC). Since most DP locations use two consecutive registers and label locations, the complete portion of the macro would read

```
MOVE.L    LABEL1(PC),D0
MOVE.L    LABEL1+4(PC),D1
```

This will move both long word parts of LABEL1 into d0/d1. How would you write this if the destination operand is a label? Simple; use the V offset twice. For example, to move the contents of d0/d1 to LABEL2 use code such as

```
MOVE.L    D0,LABEL2-V(A5)
MOVE.L    D1,LABEL2+4-V(A5)
```

This is the equivalent of MOVEDP D0,LABEL2. If you make this a macro, you must always use the V offset from a5. I find it easier to incorporate these two lines into my source code instead. This leaves me free to decide which register to use or even not to use this technique.

**COMPLEX FUNCTIONS**

As I mentioned earlier, this program will iterate functions of Z, or more precisely, (C1+iC2)*FN(Z)+(F1+iF2). This will let you multiply the function by a complex number, add a complex number or both. Later on I'll give you some specific values to try with different functions and suggest a way to make an animation out of one of these functions.

Since a lot of my programs use complex numbers (numbers which include i as part of their value) I made an include file for them called COMPLEX.I. It's Listing 1 in the article and on the magazine disk. Most of the complex functions we'll use in this article involve the hyperbolic function (EXP(B)+EXP(-B))/2. For example, the sine of Z (where Z=a+ib) is:

SIN(a)*(EXP(b)+EXP(-b))/2+i*COS(a)*(EXP(b)-EXP(-b))/2

If you multiply this by (c1+ic2) and add (f1+if2) you'll get:

REAL=c1*SIN(a)*(EXP(b)+EXP(-b))/2-c2*COS(a)*(EXP(b)-EXP(-b))/2+f1

IMAG=c2*SIN(a)*(EXP(b)+EXP(-b))/2+c1*COS(a)*(EXP(b)-EXP(-b))/2+f2

To save a lot of repetition, I first compute an e1 and e2 where e1=SIN(a)*(EXP(b)+EXP(-b))/2 and e2=COS(a)*(EXP(b)-EXP(-b))/2. Now SIN(Z) reduces to:

REAL SIN(Z)=C1*E1-C2*E2+F1

IMAG SIN(Z)=C2*E1+C1*E2+F2

Substitute these new real and imaginary values for a and b respectively. Finally, put the value of b back in registers d0/d1; you'll se why in a minute.

Each function is iterated 31 times. If a specific part of the function we're using exceeds a given value, then set that point to a color corresponding to the iteration count. The part of the function you pick and the value you compare this to make up, to a great extent, the display. In these five examples all the values are compared to 50 but you can easily

change that in the source code. The parts of Z compared to 50 are
SIN(Z) - ABS(B)  COS(Z) - ABS(B)

SINH(Z) - ABS(A)  COSH(Z) - ABS(A)
EXP(Z) - (A)

COS(Z) is computed much as SIN(Z). It's original value is:
COS(a)*(EXP(b)+EXP(-b))/2-i*SIN(a)*(EXP(b)-EXP(-b))/2
The hyperbolic functions are the opposite of the previous two functions. Swap a and b and change the exponential signs. So, SINH(Z) is
COS(b)*(EXP(a)-EXP(-a))/2+i*SIN(b)*(EXP(a)+EXP(-a))/2
And COSH(Z) is
COS(b)*(EXP(a)+EXP(-a))/2+i*SIN(b)*(EXP(a)-EXP(-a))/2
Of course, multiply each function by (c1+ic2) and add (f1+if2). That leaves the exponent function EXP(Z). This can be written as EXP(a)*EXP(ib). Now EXP(ib) is also COS(b)+i*SIN(b) so the entire function can be written as EXP(a)*COS(b)+i*EXP(a)*SIN(b). Let e1=EXP(a)*COS(b) and e2=EXP(a)*SIN(b) and the parts of exponent Z are
REAL EXP(Z)=C1*E1-C2*E2+F1
IMAG EXP(Z)=C2*E1+C1*E2+F2
For this function compare the value of the real part of (a) with 50 to see if a point will be set. This function should be iterated more than 31 times—more like 150 times—but I wanted to keep the program simple. You could add another menu selection for specific iteration counts.

Notice that this include file uses (pc) code whenever possible and assumes that variable location V will be stored in register a5; all the variables (e1, e2, etc.) must be located within the parent program.

**THE LISTING**

Now let's look at the entire program, Listing 2. After the include files, some variables are equated to registers, followed by three macros; notice that there are two different message check macros since I didn't want any menu check while the actual computations are being carried out. The various libraries are opened (be sure to have both the MATHIEEEDOUBBAS and MATHIEEEDOUBTRANS libraries in your LIBS:). A short cover screen reminds you of what the program hot-key options are. You could incorporate all these keys into menu selections instead. Bypass this screen by removing the REM in front of ";  JMP  SETUP".

After opening the screen and window, eight gadgets will appear for C1, iC2, F1, iF2, XLEFT, XRIGHT, YBOT, and YTOP. The first two are the complex number to multiply the function by; next is the complex number to add; the last four are the range of values to be shown on the screen. Default values appear initially within these gadgets, but you can change them in the source code to any values you want. The first message check waits to see what menu item (function) you pick. The macro eval_menunumber puts the item number (0-4) in TYPE; then the menu is removed.

Next the value in each gadget is converted to a DP number and stored in it's correct location; each location is a double longword. A requester will ask if you want the small (128X128) or large (320X200) version and scale the coordinates accordingly. The routine SHOWIT starts the actual computations and drawing. I left some of the MOVEDP lines in with a REM to remind you how they're actually being written relative to V(a5). Notice

that you can't use

MOVEQ #0 for DOWN, ACROSS, and COUNT since they're equated to address registers.

Depending on your menu item selection stored in TYPE, the desired function macro is called and depending on the function, part of Z is compared to 50. If it's lower, the program continues, but if it's higher, that point is set according to the iteration count value. After each point is set the ACROSS distance is increased by XINC. The message check this time waits to see if you change palettes (#0-#9), want to quit (q), change picture size (x), return to the coordinate menu (c), or zoom by pressing the LMB. When you've finished with the zoom box, a requester will appear asking if it's O.K. to zoom. If so, a second requester will confirm the size. The new coordinates will be rescaled and the new picture drawn.

**OPTIONS**

These same routines are possible when the picture is completed. You can change palettes, go back to the coordinate menu, quit the program, or zoom. The end of the listing closes the menu, window, screen, and libraries. Notice that the variables after MYWINDOW start with V. In cases where a variable is used only once (zoom, palette, etc.), I do not use the V offset; that's reserved for loops where the savings in time is more noticeable.

There are 10 palettes that can be picked at any time. Some look better with different functions and at different zoom levels. The GADGETBUFFER contains the default value for each of the eight gadgets. Feel free to change these to any other values. Some suggested parameters for trying the functions are

| | C1 | iC2 | F1 | iF2 | XLEFT | XRIGHT | YBOT | YTOP |
|---|---|---|---|---|---|---|---|---|
| SIN(Z) | 1 | 0 | 0 | 0 | -4 | 4 | - 4.5 | -4.5 |
| | 1 | .109 | 0 | 0 | -3 | 3 | -3 | 3 |
| COS(Z) | 2.95 | 0 | 0 | 0 | -1 | 1 | -1 | 1 |
| COSH(Z) | 1 | 0 | -2.5 | 0 | -2.25 | 2.25 | -3 | 3 |
| EXP(Z) | .4 | 0 | 0 | 0 | -1 | 4 | -2.5 | 2.5 |
| | 4 | 0 | 0 | 0 | -.028 | 2.35 | -1.28 | 1.28 |

You can make a very interesting animation using SIN(Z). Keep C1 at 1 and let iC2 range from 0 to about .128 for 50 frames; then reverse the order for another 50 frames to get a ping-pong effect. I use Pictsaver to save each frame. At several iC2 values (.109, .113, etc.) you'll see that tendrils from various objects touch. This animation is about 1.7MB and takes three disks; it plays with the PD program *ShowAnim* and requires 3MB of memory. I also have a reduced picture size version taking up about 600,000 bytes. If you're interested, send me one or three formatted disks with a stamped return mailer and I'll make you a copy of the animation along with the animation player; no other programs are required to view the animations.

**NEXT TIME**

This program is on the magazine disk as SINZ.ASM and SINZ along with all the include files. If you make any changes, assemble the source code using A68K SINZ.ASM and link it using BLINK SINZ.O; both A68K and BLINK are also on the disk. In the next article I'll introduce you to a new assembler and we'll use the floating point registers to turn chaos into a

thing of beauty.