Fast Fractals:

Programming The 386 Under MS-DOS

This is one of our trick articles. The fractals are really just a sneaky way to get your attention. After all, how many of you would read an article about speeding up 386 software by a factor of 100? (On second thought...)

As excitement over the 386 dies down, we're left with an uncomfortable feeling. A 32-bit operating system — one that will provide memory management over huge address spaces — is a long way off, and when it comes, it will cost a lot. But don't despair; the 386 has many powerful features that can be used under a 16-bit operating system, features that can give your programs a real kick.

This article shows how to use 32-bit instructions on the 386 under MS-DOS. As an example, we'll turbocharge the familiar Mandelbrot "zoom." Our 386 version will use fixed point math to obtain the equivalent of one MFLOPS (million floating point operations per second), without a math coprocessor. Consequently, the program will take minutes, not the usual hours or days, to produce fractal diagrams like Figure 1.

To show how simple the process is, the program will be written for a 16-bit C compiler/assembler — that is, a compiler/assembler that doesn't understand 386 mnemonics. I begin with an overview of 386 "modes," then move into a brief discussion of the Mandelbrot calculation and fixed point math.

To minimize repetition, I'll assume you've read Larry Fogg's article on the Mandelbrot set (*Micro C* issue #39 Jan./Feb. 1988) and Earl Hinrich's article on fixed point (*Micro C* issue #41 May/June 1988). Another useful reference is *The 80386/387 Architecture* by S. Morse, E. Isaacson and D. Albert.

Modes

1. The 386 has three modes — real, protected, and virtual 8086 (V8086).
2. Real mode is the default, or how the processor wakes up after you turn on the computer. Segments in real mode are 16 bits long, and there are no memory protection schemes.
3. When the 386 gets nudged into protected mode, the segments can be 32 bits long, allowing huge address spaces, and all sorts of multitasking and memory protection features become available.

V8086 mode is very handy for running old 8086 applications within a multitasking operating system; programs under V8086 behave very much like real mode programs.

32-Bit Instructions

In all modes, it's possible to access 32-bit registers and use 32-bit instructions; the 32-bit extensions of the familiar 8086 registers are called eax, ebx, ecx, edx, edi, esi, ebp and esp. The lower 16 bits of these registers can still be accessed as ax, bx, etc. MS-DOS is designed for real mode, so that's the mode we'll use.

By default, instructions in real mode operate on 8 or 16-bit quantities. To use 32-bit operands, an instruction must be prefixed with an override byte — 66H for register operations, and 67H for addressing.

If you have a 386 assembler, you won't have to worry about the override bytes. You can freely mix 16-bit and 32-bit instructions in the same assembly language program, and the assembler will automatically insert the overrides in the object code.

If you don't have a 386 assembler, you can still use the 32-bit instructions by inserting "DB 66H" or "DB 66H, 67H" before a 16-bit instruction (normally, DB — define byte — is used in the data segment, but most assemblers also allow DBs in code). If there is no corresponding 16-bit instruction, you can put the entire

To use 32-bit operands, an instruction must be prefixed with an override byte — 66H for register operations, and 67H for addressing.

byte encoding for the operation after a DB. For example, to code the 386 instructions —

```
add  eax,ebx
shld edx,eax,16
```

With a non-386 assembler, you could write:

```
DB 66H
add ax,bx       ;add eax,ebx

DB 66H, 0FH, 0A4H, 0C2H, 10H
                ;shld edx,eax,16
```

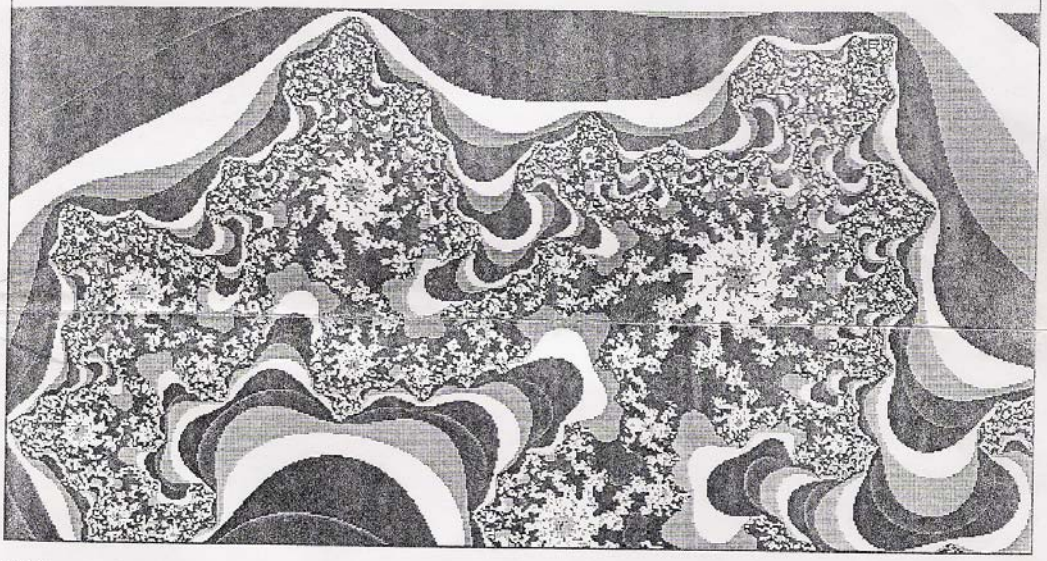
The first instruction is easy, since the older 80x86 processors have a 16-bit "add" that corresponds exactly to the 32-bit add; however, we have to replace the "shld" with a byte encoding, since the earlier processors had no double shift instructions. Byte encodings can be determined from any good 386 programming guide.

Defined Bytes

There are good reasons for using the

H. W. Stockman
5308 Noreen Drive N.E.
Albuquerque, NM 87111

Figure 1— p = -0.717 to -0.705 q = -0.3135 to -0.306



"DB" approach, even if you have a 386 assembler. For example, it is often desirable to call an assembly language function from a high level language.

Many compilers require that the function be assembled with a particular version of Microsoft's MASM. Unfortunately, MASM puts a peculiar header on functions assembled specifically for 386 real mode, even if the function contains nothing but 8086 instructions. The header is enough to confuse the heck out of non-Microsoft linkers, like the linker for Turbo C.

The DB approach circumvents the header problem, since you never have to tell the assembler that the module contains 386 code. Furthermore, many C compilers come with their own assemblers; typically, these assemblers can't understand 386 mnemonics, so there is no alternative to the DB method. I've taken the DB approach in this article because it is the most general, and because my favorite compiler is DeSmet C/asm88, which doesn't understand 386 mnemonics.

Mandelbrot Calculations

The core of the Mandelbrot calculation is the series —

$$z_{n+1} = z_n^2 + c,$$

where Z and C are complex numbers, and $Z_0 = 0$. In terms of real and imaginary components, we define $Z = X + iY$ and $C = P + iQ$. It can be shown that if $Z_n^2 \equiv X_n^2 + Y_n^2$ exceeds 4, the series will diverge.

Mandelbrot maps, such as Figure 1, show the divergence behavior of the series, as a function of P (horizontal axis) and Q (vertical axis). At each (P,Q) point in the map, we calculate the series until n hits an upper limit nmax (typically 100 to 1000), or until $Z_n^2 > 4$. The color of each point is keyed to the value of n when the calculation stopped.

Figure 1 contains 640 by 400 pixels, or 256,000 different P,Q combinations. Each pixel required an average of about 100 iterations before divergence, and each iteration required the equivalent of 13 or 14 floating point operations. Thus, Figure 1 took the equivalent of 300 to 400 million floating point operations; that's why Mandelbrot diagrams are said to be "calculation-intensive"!

Fixed Point Math

One way to speed up Mandelbrot calculations — especially on computers that don't have floating point hardware (e.g., an 80387 or 80287 math coprocessor) — is to substitute fixed point for floating

point math. This idea isn't new; see, for example, the article by H. Katz in *Dr. Dobb's Journal*, Nov. 1986. However, the 386 has instructions that make fixed point math very easy and very fast.

Fixed point numbers have an integer and a fractional portion, separated by a conceptual binary point. In our program, we'll store fixed point numbers as 32-bit integers, with the integer portion in the upper 8 bits, and the fractional portion in the lower 24 bits; thus the binary point is between bits 23 and 24.

With this system, we can represent positive or negative numbers with absolute values between about $6*10^{-8}$ and 127.9999999. That range is fine for Mandelbrot diagrams; X and Y will never get too big, because the calculation will stop if either X_n^2 or Y_n^2 exceeds 4. And it turns out that regions of the diagram with both P and Q < 0.25 are pretty uninteresting.

Fixed point numbers are added, subtracted, and compared just like ordinary 32-bit integers, but multiplication is slightly more complicated. To calculate the fixed point product of X and Y, we might try the C statement —

```
PROD = (X*Y)/16777216;
```

that is, the integer product X*Y must be divided by 2^{24} = 16777216 to get the correct fixed point product PROD. Of course, there is a problem with this C code; the product X*Y could be 64 bits long, which would cause overflow in most *compiled* code.

The problem disappears in assembly language, since most 32-bit processors have no trouble multiplying 32-bit numbers to form a 64-bit product. For example, suppose we have two fixed point numbers stored in the 386 registers eax and ebx; to multiply these numbers, and place the fixed point product in edx, requires only two instructions:

```
imul ebx ;eax is implicit destination
shld edx,eax,8 ;result now in edx
```

The imul places the 64-bit product in the register pair edx:eax (the high bits in edx), and the double shift instruction adjusts the binary point (we shift left by 8 bits, instead of right by 24 bits, because we want the product to end up in edx).

Some programmers prefer fixed point with 16-bit integer and 16-bit fractional portions. However, I've seen Mandelbrot calculations done with the 16/16 format, and the inaccuracies were pretty obvious, even at modest magnifications.

Figure 2 — Typical Floating Point Mandelbrot

```
/* Floating point version of mandel() for DeSmet C.                          */
/* ncol and nrow are number of columns and rows that can be displayed by */
/* chosen graphics adapter/mode: e.g., (ncol,nrow) = (720,348) for herc, */
/* (640,350) for EGA; endcolor is one less than max # colors the graphic */
/* adapter/mode can display: e.g., endcolor is 1 for hercules, 15 for    */
/* most EGAs; ptptr is pointer to pixel plotting function. nmax is the   */
/* maximum number of iterations we will allow for Mandelbrot series.     */
/* DeSmet function csts() reads keyboard buffer, doesn't wait if no key  */
/* was pressed. similar to Turbo C combination of kbhit() and getch()    */

mandel(Pmin, Pmax, Qmin, Qmax, nmax, ncol, nrow, ptptr, endcolor, switchpt)
double     Pmin, Pmax, Qmin, Qmax;
int        switchpt,nmax,ncol,nrow,endcolor;
void       (*ptptr)();
{
     int        color,row,col,n,transform();
     double     P,Q,dP,dQ,x,y,ytemp,modulus_sqrd;

     dP = Pmax-Pmin;
     dQ = Qmax-Qmin;

     for(col=0; col<ncol; ++col){
          for(row=0; row<nrow; ++row){
               P = Pmin + dP*col/(ncol-1);
               Q = Qmin + dQ*row/(nrow-1);
               n = 0;
               x = y = modulus_sqrd = 0.0;
               while(modulus_sqrd <= 4.0 && n < nmax){
                    ytemp = x*y;
                    x = x*x - y*y + P;
                    y = ytemp + ytemp + Q;
                    modulus_sqrd = x*x + y*y;
                    n++;
                    }
               /* transform the stopping iteration n to a color */
               color = transform(n,switchpt) & endcolor;
               (*ptptr)(col,row,color);
               }
          /* if a 'q' is pressed at keyboard  abort the funtion */
          if(csts()=='q') return('q');
          }
}

END OF LISTING
```

Figure 3 — Fixed Point Version of Mandelbrot

```
/* Fixed point version of mandel() */

mandel(Pmin, Pmax, Qmin, Qmax, nmax, ncol, nrow,
       ptptr, endcolor, switchpt)
double     Pmin, Pmax, Qmin, Qmax;
int        switchpt,nmax,ncol,nrow,endcolor;
void       (*ptptr)();/* ptptr is pointer to pixel plotting function*/
{
     int        color,row,col,n,mandwhile(),transform();
     long       P,Q,P0,Q0,dP,dQ;
     long       muldiv();

     dP = (Pmax-Pmin)*16777216 + 0.5;  /* 16777216 = 2 to the 24th */
     dQ = (Qmax-Qmin)*16777216 + 0.5;
     P0 = Pmin*16777216 + 0.5;
     Q0 = Qmin*16777216 + 0.5;

     for(col=0; col<ncol; ++col){
          for(row=0; row<nrow; ++row){
               P = P0 + muldiv(dP,col,ncol-1);
               Q = Q0 + muldiv(dQ,row,nrow-1);
               n = mandwhile(P,Q,nmax) + 1;
               /* +1 above depends on how 1st iteration is defined */
               color = transform(n,switchpt) & endcolor;
               (*ptptr)(col,row,color);
               }
          if(csts()=='q') return('q');
          }
}

END OF LISTING
```

Figure 4 — Assembly Language Fixed Point Calculations

```
; muldiv(long A, int b, int c)
; Multiplies 32-bit "A" by 16-bit "b", then divides by
; 16-bit "c" such that abs(c) >= abs(b)...NO CHECK FOR 0 DIVISOR...
;
; NOTE how DeSmet figures stack upon entry:
; 16-bit return address at sp, "A" at sp+2, b at sp+6, c at sp+8 ...
; NOTE DeSmet does not use "word ptr" formalism ...
;
; REGISTERS:              eax     ...initially holds "A"
;                         ebx     ...holds b
;                         ecx     ...holds c
;                      edx:eax    ...product A*b (before idiv)
;
; RETURN result in dx:ax.

cseg
public    muldiv_
muldiv_:
     db 67h,66h,8bh,44h,24h,02h          ;mov eax,dword [esp+2]
     db 67h,66h,0fh,0bfh,5ch,24h,06h     ;movsx ebx,word [esp+6]
     db 67h,66h,0fh,0bfh,4ch,24h,08h     ;movsx ecx,word [esp+8]
     db 66h,0f7h,0ebh                    ;imul ebx
     db 66h,0f7h,0f9h                    ;idiv ecx
     db 66h,0fh,0a4h,0c2h,10h            ;shld edx,eax,16
     ret                                 ;short return...

END OF LISTING
```

With our 8/24 format, and our limited P,Q range, we'll have accuracy similar to IEEE single precision. At some point, even this accuracy isn't enough; if we try to examine regions with very small P and Q ranges (e.g., $P_{max} - P_{min} < 0.00001$), part of our Mandelbrot map could be noise.

Also, note that we can get a fixed point number in C by multiplying the corresponding "float" by 2^{24}, then truncating the result to a long integer. In general, though, we will try to do all our fixed point math with integer operations.

The Program

In this section we'll discuss the workhorse functions that do the Mandelbrot calculations over a fixed P,Q range. I'll leave it up to you to supply the calling program. Alternatively, you can download the complete program and executable code from the Micro C RBBS (503) 382-7643, or send $6 to Micro Cornucopia, P.O. Box 223, Bend, OR 97701, for the source listing (ask for Issue Disk #43). Once again, I strongly recommend reading Larry Fogg's *Micro C* article on the Mandelbrot set.

Figure 2 shows how the Mandelbrot calculation is typically coded in floating point math; this is a modification of Larry Fogg's mandel(). Figure 3 shows how the function is altered to use fast assembly language routines. The assembly functions are described below.

To save space in the listings, I've converted the 386 instructions completely to byte encodings, but all the mnemonics are commented in, so it should be easy to adapt the programs to any compiler/assembler. If you wish to use another compiler/assembler, be sure you understand the order in which values are pushed onto the stack. Not all compilers are like DeSmet C, so you may have to change the values of "m" in instructions containing "[esp+m]."

Also note that some compilers will require you to save the si and di registers at the beginning of each function. Finally, we'll assume that the upper 16 bits of esp are zeroed and considered meaningless by our operating system; since we're launching the program from MS-DOS, that's a valid assumption.

The Assembly Language Functions

The first assembly function, muldiv(), performs the operations A*b/c, where A is a 32-bit integer, and b and c are 16-bit integers such that $|b| \le |c|$. The calculation is done in a manner that avoids overflow and truncation. This type of function is extremely useful in DSP (digital signal processing) applications; since muldiv() requires very few 386 instructions (see Figure 4), it serves as a good beginning example.

Note that with the 386 we can use [esp] directly to address the stack; that is, we don't have to go through the "push bp, mov bp, sp ..." formalism so familiar to 8086 programmers. However, we must make sure esp actually has a 16-bit value when we address the stack, else we'll generate a real mode stack exception error.

Figure 5 — Looping Routine

```
; mandwhile(long P, long Q, int nmax)
; Performs all calculations for "while" loop in L.Fogg's mandel()
; function, using fixed pt math with binary pt between bits 23 and 24...
;
; FOR: DeSmet C/asm88 small case...
;
; In REAL MODE: uses 32-bit overrides on register length and addressing to
; gain performance of 32-bit regs and instructions...
; Returns # iterations before divergence (16-bit int returned in ax).
; Note 1/3 rd of instructions are for rounding, to get just 1/2 extra bit of
; accuracy: if you aren't that picky, dump these instructions for 10% or more
; extra speed...
;
; REGISTERS:            ebp     ...ytemp at top of loop, modulus_sqrd
;                                   at bottom of loop
;                   edx:eax     ...multiplication and temporary storage
;                       edi     ...P
;                       esi     ...Q
;                       ebx     ...x
;                       ecx     ...y
;
; RETURNS stopping number of iterations in ax.

dseg
public    counter
counter   dw   0
cseg
public    mandwhile_
mandwhile_:
          db 66h, 55h                       ;push ebp
         ;---get P ...
          db 67h, 66h, 8Bh, 7Ch, 24h, 06h   ;mov edi,dword [esp+6]
         ;---get Q ...
          db 67h, 66h, 8Bh, 74h, 24h, 0Ah   ;mov esi,dword [esp+10]
         ;---get nmax...
          db 67h, 8Bh, 44h, 24h, 0Eh        ;mov ax,word [esp+14]
         ;---initialize counter ...
          mov word  counter,ax
         ;---zero out x and y storage ...
          db 66h, 33h, 0DBh                 ;xor ebx,ebx
          db 66h, 33h, 0C9h                 ;xor ecx,ecx

looper: ;---ytemp=x*y---
          db 66h, 8Bh, 0C3h                 ;mov eax,ebx
          db 66h, 0F7h, 0E9h                ;imul ecx
          db 66h, 05h, 00h, 00h, 80h, 00h   ;add eax,800000H    ;for rounding...
          db 66h, 83h, 0D2h, 00h            ;adc edx,0          ;for rounding...
          db 66h, 0Fh, 0A4h, 0C2h, 08h      ;shld edx,eax,8     ;div by 2**24...
          db 66h, 8Bh, 0EAh                 ;mov ebp,edx
         ;---x=x*x - y*y + P------
          db 66h, 8Bh, 0C3h                 ;mov eax,ebx
          db 66h, 0F7h, 0EBh                ;imul ebx
          db 66h, 05h, 00h, 00h, 80h, 00h   ;add eax,800000H
          db 66h, 83h, 0D2h, 00h            ;adc edx,0
          db 66h, 0Fh, 0A4h, 0C2h, 08h      ;shld edx,eax,8
          db 66h, 8Bh, 0DAh                 ;mov ebx,edx
         ;---
          db 66h, 8Bh, 0C1h                 ;mov eax,ecx
          db 66h, 0F7h, 0E9h                ;imul ecx
          db 66h, 05h, 00h, 00h, 80h, 00h   ;add eax,800000H
          db 66h, 83h, 0D2h, 00h            ;adc edx,0
          db 66h, 0Fh, 0A4h, 0C2h, 08h      ;shld edx,eax,8
          db 66h, 2Bh, 0DAh                 ;sub ebx,edx
          db 66h, 03h, 0DFh                 ;add ebx,edi      ;add P...
         ;---y=(ytemp<<1) + Q----
          db 66h, 0D1h, 0E5h                ;shl ebp,1
          db 66h, 03h, 0EEh                 ;add ebp,esi      ;add Q...
          db 66h, 8Bh, 0CDh                 ;mov ecx,ebp
         ;---modsqrd = x*x + y*y---
          db 66h, 8Bh, 0C3h                 ;mov eax,ebx
          db 66h, 0F7h, 0EBh                ;imul ebx
          db 66h, 05h, 00h, 00h, 80h, 00h   ;add eax,800000H
          db 66h, 83h, 0D2h, 00h            ;adc edx,0
          db 66h, 0Fh, 0A4h, 0C2h, 08h      ;shld edx,eax,8
          db 66h, 8Bh, 0EAh                 ;mov ebp,edx
         ;---
          db 66h, 8Bh, 0C1h                 ;mov eax,ecx
          db 66h, 0F7h, 0E9h                ;imul ecx
          db 66h, 05h, 00h, 00h, 80h, 00h   ;add eax,800000H
          db 66h, 83h, 0D2h, 00h            ;adc edx,0
          db 66h, 0Fh, 0A4h, 0C2h, 08h      ;shld edx,eax,8
          db 66h, 03h, 0EAh                 ;add ebp,edx
         ;---test if modsqrd > "4"---(67108864 = 4*(1 << 24))---
          db 66h, 81h, 0FDh, 00h,00h,00h,04h  ;cmp ebp,67108864
```

```
        jae home
        dec word  counter
        jnz looper
home:   mov ax,word  counter
        neg ax
        db 67h, 03h, 44h, 24h, 0Eh      ;add ax, word [esp+14]
        db 66h, 5Dh                     ;pop ebp
        ret

END OF FIGURE 5
```

Figure 6 — Smoothing Color Transitions

```
; 2-20-88 transform(int n, int switchpt)
;
; FOR: DeSmet small case C/asm88
;
; ...returns in ax the "color" which is then anded with (maxcolors-1)...
; converts the iterations returned by mandwhile() to a log scaling, starting at
; the switchpt. Equiv. to C coding:
; if(n < switchpt) color = n;
; else for(color=i=switchpt,inc=1; i<n; i=switchpt+(inc<<=1),color++);
; ...However, 386 is so fast we need to translate to asm to get full benefit!
;
; NOTE DeSmet short return via jmp !!

cseg
public    transform_
transform_:         pop di          ;return address...
                    pop cx          ;n...
                    pop bx          ;switchpt...
                    sub sp,4        ;required by DeSmet for short return...
                    cmp cx,bx
                    ja do_loop
                    mov ax,cx
                    jmp di          ;short return...
                    ;------
          do_loop:  mov ax,bx       ;init color (ax) with switchpt...
                    mov si,1        ;si = inc...
                    mov dx,bx       ;init i (dx) with switchpt...
                    ;------
          loopit:   cmp dx,cx       ;compare i (dx) to n (cx)...
                    jge done
                    shl si,1        ;inc <<= 1...
                    mov dx,bx
                    add dx,si       ;i = switchpt + inc...
                    inc ax
                    jmp loopit
                    ;------
          done:     jmp di          ;short return, color in ax...

END OF FIGURE 6
```

Figure 7 — Detects Presence of 386

```
; is_386() ... no arguments...
; Checks for presence of 386...
;
; FOR: DeSmet C small case, asm88
; RETURNS: 0 in ax if processor not a 386; else returns non-zero ...
; NOTE short return via jmp.
;
; REF: Juan E. Jimenez, Turbo Technix Jan/Feb 88, p. 55.

cseg
public    is_386_
is_386_:            pop di          ;return address ...
                    pushf
                    xor ax,ax
                    push ax
                    popf
                    pushf
                    pop ax
                    and ax,8000h
                    sub ax,8000h
                    jz home2
                    ;------
                    mov ax,7000h    ;if here, either 286 or 386 ...
                    push ax
                    popf
                    pushf
                    pop ax
                    and ax,7000h
          home2:    jmp di

END OF FIGURE
```

The "movsx" instruction (move with sign extend) is new to the 80x86 family, and is handy in converting 16-bit quantities for use in 32-bit instructions.

Perhaps the most notable aspect of muldiv() is the use of shld edx, eax,16 at the end of the function. This instruction is needed because DeSmet C, like most 16-bit 80x86 compilers, returns 32-bit integers in dx:ax. After the idiv instruction, the desired result is in eax; the shld instruction leaves eax unchanged, but copies the upper 16 bits of eax into dx, so ultimately the correct value is returned in dx:ax.

Looping Calculations

The second and more important function, mandwhile() (see Figure 5), replaces the inner "while" loop in Figure 2; it returns only the stopping value of n.

After the 32*32 bit multiplications, the "add eax,800000H" and "adc edx,0H" instructions assure proper rounding (even for negative numbers) when we use the shld instruction. This step adds only a little accuracy, so you can eliminate the rounding for about 10% more speed and substantially shorter coding. Again, note that we actually accomplish the 24-bit "division" — required to adjust the binary point — with an 8-bit left shift.

mandwhile() is well suited for the 386 architecture, and should run fast even on computers with slow memory, because the algorithm uses the 386 pre-fetch queue very efficiently.

The imul instruction is relatively "slow," averaging about 30 clocks over the P,Q region of interest. While each imul executes, there is plenty of time to fetch and decode the faster add and mov instructions. The queue gets emptied only at the end of each loop iteration, and is quickly refilled.

The shld instruction is ideal for adjustment of the binary point after multiplication. Typically, the equivalent operation takes 3 instructions on other 32-bit microprocessors — e.g., if a 68020 had the 64-bit product in D2:D1, you would probably code:

```
lsl.l #8,D2
lsr.l #24,D1
or.l   D1,D2
```

to replace the shld. Furthermore, the 68020 32*32 multiplication is somewhat slower than the 386 multiplication, so the 386 really has an advantage.

Final Functions

The last two functions called by mandel() are transform() (see Figure 6) and (*ptptr)(). The purpose of transform() is to "slow down" the alternation of colors in rapidly changing portions of the map, making it easier to see broad patterns. Values of n up to the "switch point" are returned unchanged by the function, but values above switchpt are converted to log2(n-switchpt) + switchpt. This function is easily written in C, but for speed I've also included an assembly language version.

The value returned by transform() is "anded" with endcolor to obtain the pixel color (the "anding" process assumes your graphics adapter can display 2^m colors, with endcolor = 2^m - 1). The transform() function is optional; if you don't want to use it, simply code color = n & endcolor instead. The color and coordinates of the point are then passed to (*ptptr)(), the pixel plotting function.

Since ptptr is an argument of mandel(), we can allow main() to detect the graphics adapter in the computer and choose the appropriate plotting function. If you know for sure that you will be using only one graphics adapter, you can remove ptptr from the argument list of mandel() and replace (*ptptr)() with an explicit plotting function.

The plotting functions can be coded in C (e.g., *Micro* C issue #39 Jan./Dec. 1988, pp. 24 and 84), but the rest of mandel() is so fast, it is preferable to use assembly language.

I've included another useful function in Figure 7; is_386() returns a non-zero value if the computer has a 386 processor, else it returns a zero. Thus you can keep non-386 computers from trying to run 386 code (and consequently going off into never-never land).

How Fast?

How much speed did we gain by using 32-bit instructions? In the past, I've written fixed point Mandelbrot programs in 16-bit 8086 code; on a 16 MHz 80386, 16-bit code takes eight times longer to run than the 32-bit version.

In fact, our 32-bit version is at least three times faster than a 16 MHz 80387 with hand-coded assembly language; 12-15 times faster than an 8 MHz 80287; 22 times faster than fixed point routines on a Macintosh Plus; 100-200 times faster than software floating point on an 8 MHz 80286; or as fast as an 8600 VAX superminicomputer.

There are probably more efficient algorithms than Figure 2. If you find them, perhaps you'll get the calculation time down to seconds.

Final Words

We barely skirted the 386 features you can use under MS-DOS. We didn't cover advanced addressing modes, nor all the powerful new instructions. The 386 is full of goodies, such as BSF and BSR (bit scan forward and bit scan reverse), which make software floating point fast and very easy.

It's important to explore. The chip has so much potential, it would be criminal to use the 386 as nothing more than a fast 286. Instead of chewing your fingernails while Microsoft inflates OS/2 a few more megabytes, start lacing your MS-DOS programs with high-speed 32-bit instructions.

Yes, you're still stuck with 16-bit segments; but how many of your programs really need data structures larger than 64K? Probably very few, and OS/2 can't help you with that problem anyway. When the protected mode operating system finally rolls around, you will be that much ahead of the pack.

♦ ♦ ♦