New Technical Notes
Macintosh

Developer Support

Speedy the Math Coprocessor
Hardware **M.HW.MathCoProc**

Written by: Rich Collyer June 1989

This Technical Note presents an overview of the 68881 and 68882 math coprocessors, and it covers general information about the chips as well as how using the chips directly can help speed your math–intensive code.

Introduction

Generally we don't recommend that you assume the existence of specific hardware. However, if your program does proper feature checking using `_SysEnvirons` and there is a Floating–Point Unit (FPU) available, than you can use code which will run your math intensive code much faster. This Technical Note is basically a condensed version of the *Motorola MC68881/MC68882 Floating–Point Coprocessor User's Manual*. I will cover some of the basics of what the chips can do, their differences, and how to take advantage of what they have to offer.

If `_SysEnvirons` returns `hasFPU = FALSE`, then your code should use the routines provided by the Standard Apple Numeric Environment (SANE). The routines which SANE provide are covered in the *Apple Numerics Manual*.

So What Can These Chips Do?

The MC68881 and MC68882 are floating–point coprocessors which implement the IEEE standard for binary floating–point arithmetic. The two chips are fully interchangeable and are primarily for use as coprocessors to the MC68020 and MC68030 central processors. The two chips will work as peripheral processors to the MC68000, MC68008, and MC68010 central processors.

Both chips have eight 80–bit general purpose floating–point data registers (`FP0-FP7`), 67–bit arithmetic units with precision greater than the extended format, 67–bit barrel shifter, 46 instructions, trigonometric and transcendental functions, and 21 constants. The MC68882 also has the capability of concurrent execution of multiple floating–point instructions.

Internal Registers for a Higher Capacity to Think

There are eleven separate registers in these puppies: eight data registers, one control register, one status register, and one address register.

Data Registers

There are eight 80–bit floating–point data registers labeled FP0–FP7. The extended format, which is used by these registers, will be covered later. When using the FPU from an MPW C and Pascal application, you can us FP0–FP3 for temporary storage without saving and restoring their values. If you wish to use FP4–FP7 in your assembly routine, then you must save these registers at the start of your assembly code and restore them before you leave the assembly code.

Control Register (FPCR)

Below is a representation of the control register. For the most part, there is no need for you to do anything to the control register directly. It is used internally for determining precision, rounding, and error checking.

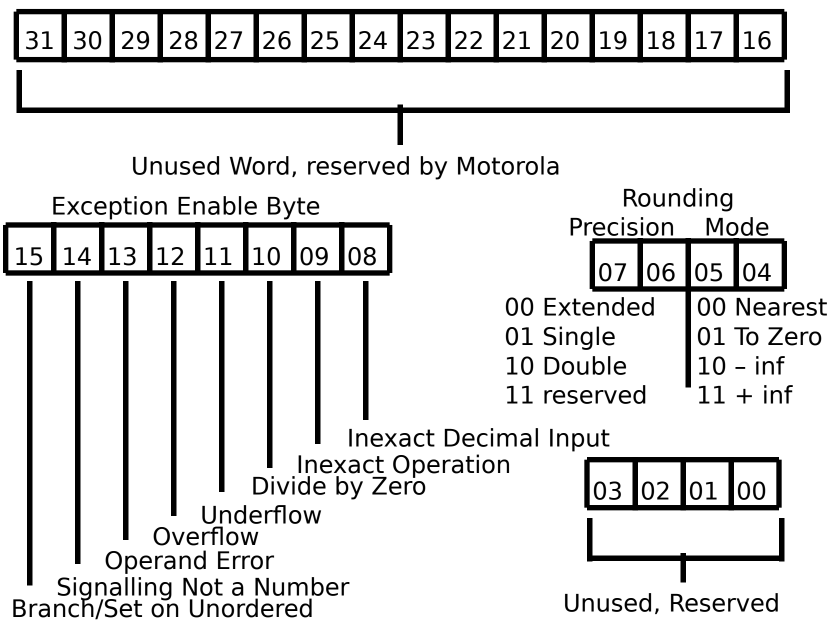

Figure 1–Control Register

Status Register (FPSR)

The status register is diagrammed in Figure 2. This register is also used mostly for internal chores. The condition–code byte is set at the end of each arithmetic instruction. The condition–code byte is translated into a data type; Table 1 shows the relationship between condition codes and data types. The condition code is also used to determine logic equates. If you wish to determine if two numbers are equal, than the Compare statement (FCMP) will check the condition code. Table 2 shows the relationship between the condition codes and logic equates.

The quotient byte is set at the completion of FMOD (Modulo Remainder) and FREM (IEEE

Remainder). This byte can be used before a transcendental function to determine the quadrant of a circle in which an operand resides. The FP–exception status byte is used in conjunction with the exception–enable byte of the control register. The FP–accrued exception byte is used to keep a history of the FP exceptions that have occurred since the last set or clear.

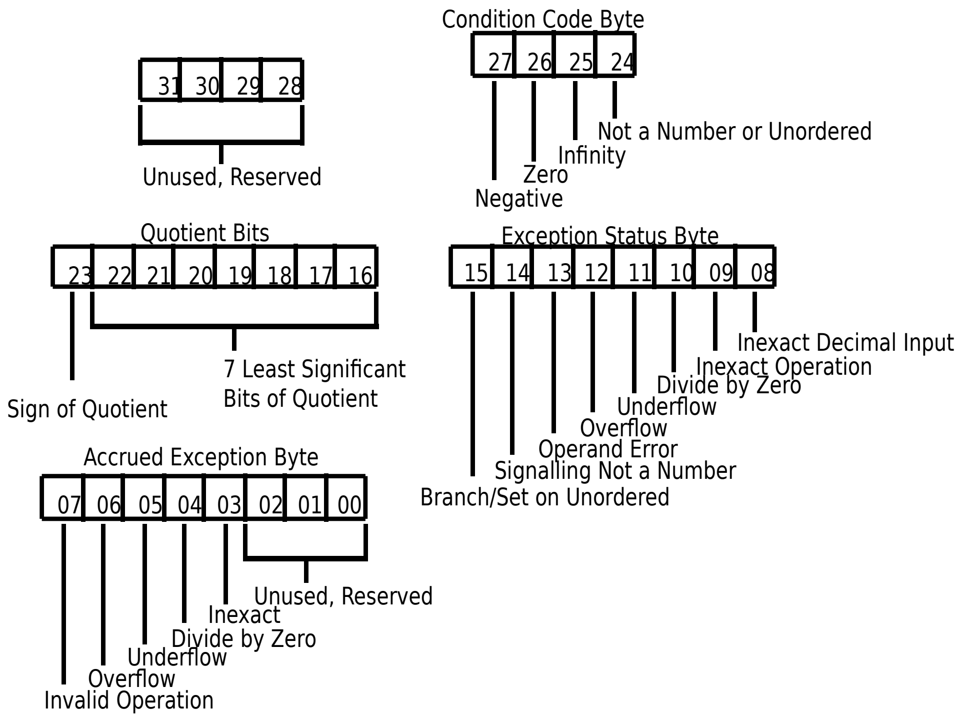

Figure 2–Status Register

Negative	Zero	Infinity	NAN	Result Data Type
0	0	0	0	+ Normalized or Denormalized
1	0	0	0	– Normalized or Denormalized
0	1	0	0	+ zero
1	1	0	0	– zero
0	0	1	0	+ infinity
1	0	1	0	– infinity
0	0	0	1	+ NAN
1	0	0	1	– NAN

Table 1–Condition Code versus Result Data Type

Logic Equate	Abbreviation	Condition Code
Equal to	EQ	Z
Not Equal	NE	not Z
Greater Than	GT or OGT	not(N or NAN or Z)
Not Greater Than	NGT or UGT	NAN or Z or N
Greater Than or Equal	GE or OGE	Z or (not(NAN or N))
Not (Greater Than or Equal)	NGE or UGE	NAN or (N and (not Z))
Less Than	LT or OLT	N and (not(NAN or Z))
Not Less Than	NLT or ULT	NAN or (Z and (not N))
Less Than or Equal	LE or OLE	Z or (N and (not NAN))
Not (Less Than or Equal	NLE or ULE	NAN or (not (N or Z))
Greater or Less Than	GL or OGL	not (NAN or Z)
Not (Greater or Less Than)	NGL or UEQ	NAN or Z
Greater, Less or Equal)	GLE or OR	not NAN
Not (Greater, Less or Equal)	NGLE or UN	NAN
	Oxx is ordered	Z –> Zero
	Uxx is unordered	N –> Negative

Table 2–Logic Equates

Address Register (`FPIAR`)

Since the coprocessor can do concurrent processing with the MC68020 and MC68030, as well as with itself, the program counter is not necessarily pointing to the logical address of the instruction upon which it is working. So the address register stores the logical address of each floating–point instruction before executing it.

Floating–Point Data Formats

There are four floating–point numeric formats: single–precision binary real format, double–precision binary real format, eXtended–precision binary real format, and Pack decimal real format (a.k.a., BCD). I have given examples of what the FPU will convert your numbers to. The number which I have used for the four examples is Planck's constant (4.136 x 10^{-15} eV–sec). Other than the size, the first three formats are very similar. The three formats all have the same conversion method and ordering of information.

Single (S) 32 bit

Single precision is represented by 32 bits of information. The high bit (bit 31) is the sign bit (s). The next byte of information (bits 30–23) is the exponent (e), and the last 23 bits (bits 22–0) are the fraction (f). The bits of information are converted into a floating–point number by the following equation:

$$(-1)^s * 2^{(e-127)} * (2^0 + f)$$

The fraction (f) is the strange value. Each bit in the fraction value represents a negative exponent of two. So if bit 22 and bit 16 are high, and all the rest of the bits are low, than the fraction would equal 0.5078125 or ($2^{-1} + 2^{-7}$). So when I give the FPU the number 4.136e–15, it converts the number into the hexadecimal number $04F1503DE, which, in the above equation, looks like:

$$(-1)^{0} * 2^{(79-127)} * 2^{0}+2^{-3}+2^{-5}+2^{-7}+2^{-14}+2^{-15}+2^{-16}+2^{-17}+2^{-19}+2^{-20}+2^{-21}+2^{-22}$$

This number is than converted back to a base ten number as 4.13600004803759899e–15. As you can see, the number is correct up to the seventh decimal place.

Double (D) 64 bit

Double precision is represented by 64 bits of information. The high bit (bit 63) is the sign bit (s), The next 11 bits of information (bits 62–52) are the exponent (e), and the last 52 bits (bits 51–0) are the fraction (f). The bits of information are converted into a floating–point number by the following equation:

$$(-1)^{s} * 2^{(e-1023)} * (2^{0} + f)$$

When I give the FPU the number 4.136e–15 as a double, it converts the number into the hexadecimal number $03CF2A07BBC5ED155. This number is than converted back to a base ten number as 4.13600000000000015e–15. As you can see, the number is correct up to the fifteenth decimal place.

EXtended (X) 96/80 bit

Extended precision is represented by 96 bits of information; SANE and FP data register use 80–bit extended numbers, but the FPU extended numbers are 96 bits with 16 unused bits, so the two are basically the same. The high bit (bit 95) is the sign bit (s), The next 15 bits of information (bits 94–81) are the exponent (e), there are 16 unused bits (bits 80–64), and the last 64 bits (bits 63–0) are the fraction (f). The bits of information are converted into a floating–point number by the following equation:

$$(-1)^{s} * 2^{(e-16383)} * (2^{0} + f)$$

When I give the FPU the number 4.136e–15 as a extended, it converts the number into the hexadecimal number $03FCF(0000)9503DDE2F68AA66F. This number is than converted back to a base ten number as 4.136e–15. This number is correct to about the nineteenth decimal place.

Pack Decimal Real (P) BCD Format 96 bits

Pack Decimal Real is represented by 96 bits of information. The bits of these numbers are represented as follows:

bit 95 Sign of Mantissa
bit 94 Sign of Exponent
bit 93–92 used for +–infinity and NANs,otherwise zero
bits 91–81 10–bit Exponent (3 digit exponent)
bits 80–68 unused, zero
bit 67–0, 68 bit Mantissa (17 digit mantissa)

When I give the FPU the number 4.136e–15 as a PDR, it converts the number into the hexadecimal number $4015000413600000000000000. This hexadecimal number is filled into the above bit as follows:

bit 95 Sign of Mantissa	0 (binary)
bit 94 Sign of Exponent	1 (binary)
bit 93–92 used for +–infinity and NANs,otherwise zero	00 (binary)
bits 91–80 11–bit Exponent (3 digit exponent)	000000010101 (binary)
bits 79–68 unused, zero	000000000000 (binary)
bit 67–0 68 bit Mantissa (17 digit mantissa)	41360000000000000 (hex)

This number is than converted back to a base ten number as 4.136e–15. This number is correct to the seventeenth decimal place.

So What Tools Do I Have to Play With?

There are four types of opcodes which the math coprocessors support: moves, monodic, dyadic, and miscellaneous conditions. When a coprocessor operation is executed, the first operation which the coprocessor performs is to convert the data to the internal extended precision format, and when the operation is completed, the data is converted to the destination data format.

Moves

The first type which I will describe are the move opcodes. Below is a list of the various formats in which the move commands come.

Move
```
FMOVE.<fmt>   <ea>, FPn
FMOVE.<fmt>   FPm, <ea>
FMOVE.X       FPm, FPn
```

Move Multiple
```
FMOVEM <ea>, FP0 - FP3/FP7
FMOVEM FP2/FP4/FP6, <ea>   ;the registers are always moved as 96 bit extended
                           ;data without conversion
```
Move Register
```
FMOVE.L       <ea>, FPCR   ;move to control register
FMOVE.L       FPCR, <ea>   ;move from control register
```

Move Constants from ROM to floating-point register
```
FMOVECR.X     #ccc, FPn    ;see Table 3 for #ccc
```

Save and Restore Machine State
```
FSAVE         <ea>         ;virtual machine state save
FRESTORE      <ea>         ;virtual machine state restore
```

`<ea>` is a main processing unit (MPU) effective address operand (any 68xxx addressing mode).
`<fmt>` is the data format size (Byte, Word, Long, Single, Double, eXtended, Packed decimal).
`FPm` and `FPn` are floating–point data registers.

#ccc	Mathematical Representation	Numeric Representation
$00	pi	3.14159265358979324
$0B	log(base 10)(2)	0.301029995663981195
$0C	e	2.71828182845904524
$0D	log(base 2)(e)	1.442695040888963410
$0E	log(base 10)(e)	0.434294481903251828
$0F	zero	0
$30	ln(2)	0.693147180559945309
$31	ln(10)	2.302585092994045684
$32	10^0	1
$33	10^1	10
$34	10^2	100
$35	10^4	10,000
$36	10^8	100,000,000
$37	10^16	10,000,000,000,000,000
$38	10^32	100...(28 more zeros)...00
$39	10^64	100...(60 more zeros)...00
$3A	10^128	100...(124 more zeros)...00
$3B	10^256	100...(252 more zeros)...00
$3C	10^512	100...(508 more zeros)...00
$3D	10^1024	100...(1020 more zeros)...00
$3E	10^2048	100...(2044 more zeros)...00
$3F	10^4096	100...(4092 more zeros)...00

Table 3–Constants

Monodic

A monodic operation has one operand. The operand may be a floating–point data register or an MPU effective address. The result is always stored in a floating–point data register. The syntax for monodic operations is listed below:

```
Fxxxx.<fmt>   <ea>, FPn
Fxxxx.X       FPm, FPn
Fxxxx.X       FPn
```

where: `<fmt>` is (B,W,L,S,D,X,P)
`xxxx` is one of the Trigonometric (`SIN`), Transcendental (`ATANH`), Exponential (`ETOXM1`), Misc. commands (`NEG`)

Dyadic

A dyadic operation has two operands. The first operand can be in a floating–point data register, or an MPU effective address. The second operand is the contents of a floating–point data register. The result of the operation is stored in the second operand. The syntax for dyadic operations is listed below:

```
Fxxxx.<fmt>   <ea>, FPn
Fxxxx.X       FPm, FPn
```

where `<fmt>` is (B,W,L,S,D,X,P)
`xxxx` is a arithmetic (`ADD`), compare (`CMP`)

Condition operations

There are four condition operations: branch (`FBcc`), decrement and branch (`FDBcc`), set according to condition (`FScc`), and trap on condition (`FTRAPcc`).

Why and How do I Program for a 68882?

Any code which runs on a 68881 will run on a 68882 and vice versa. You do not need to take special care to program for the 68882, but if the chip is available, than special care can noticeably improve the speed of your code. Figure 3 demonstrates the difference between code run on a 68881 and the same code run on a 68882. The 68882 is completely finished running before the 68881 has even started executing the `FMOVE` instruction. The extra work which you need to do to take advantage of the concurrent processing is fairly minimal.

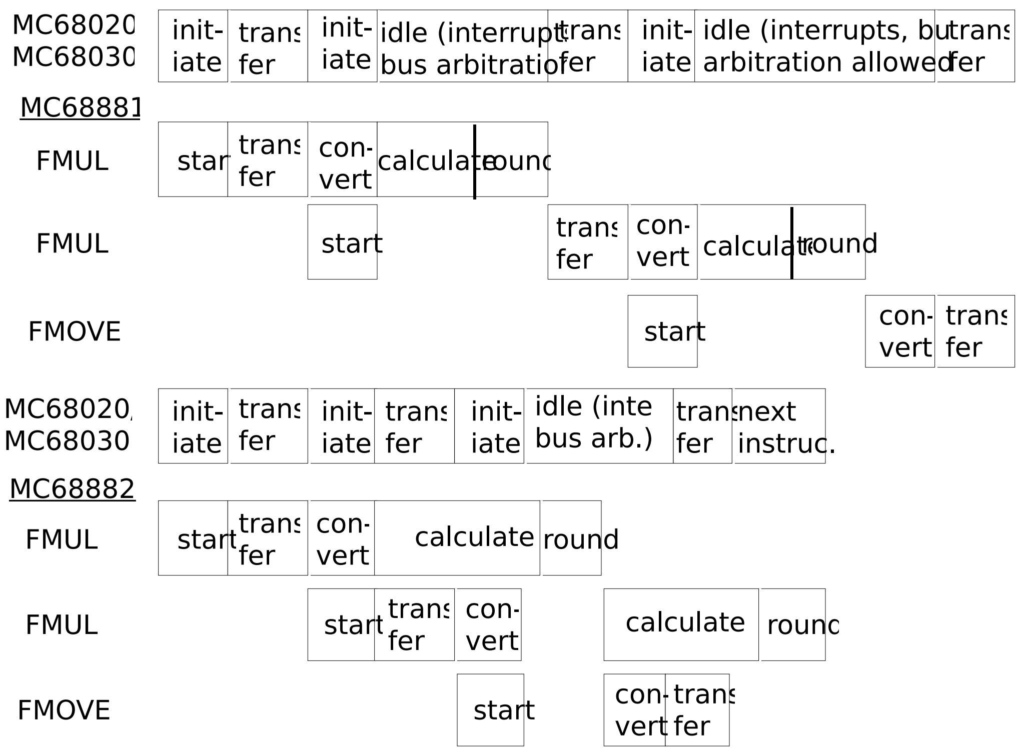

Figure 3–Concurrent Execution versus Non–Concurrent Execution

Before you jump right in and start writing code, you need to understand that there are three different levels of concurrency. The first level is the minimum concurrency operations. These are operations which cannot run concurrently with other operations. Most of these operations are non–floating–point format operations. The minimum concurrency operations are listed in Table 4.

Instruction	Operand Syntax	Operand Format
`FMOVE`	<ea>, FPn	B,W,L,P
	FPm, <ea>	B,W,L
	FPm, <ea>	P
	FPm, <ea>	P
	<ea>, FPcr	L
	FPcr, <ea>	L
`FMOVECR`	#ccc, FPn	X
`FMOVEM`	<ea>, <list>	L,X
	<ea>, Dn	X
	<list>, <ea>	L,X
	Dn, <ea>	X
`FTST`	FPm	B,W,L,P
`F<monodic>`	<ea>, FPn	B,W,L,P
`F<dyadic>`	<ea>, FPn	B,W,L,P
`FSINCOS`	<ea>, FPc:FPs	B,W,L,P

Table 4–Minimum Concurrency

The next level of operations are the operations which can share some of the FPU time with other operations, these are the partial concurrency operations and they are listed in Table 5. The partial concurrency operations include most of the floating–point format operations.

Instruction	Operand Syntax	Operand Format
`FTST`	<ea>	S,D,X
	FPm	X
`F<monodic>`	<ea>, FPn	S,D,X
	FPm, FPn	
`F<dyadic>`	<ea>, FPn	S,D,X
	FPm, FPn	
`FSINCOS`	<ea>, FPc:FPs	S,D,X
	FPm, FPc:FPs	X

Table 5–Partial Concurrency

The highest level of concurrency is the fully–concurrent operations which are listed in Table 6. The only operations which can run fully concurrently are the `FMOVE` operations. There are certain guidelines which you need to follow in order to achieve full concurrency, these guidelines are outlined in Table 6. The most important rule to follow is to avoid register conflict. There are basically two type of register conflict. The first is when the destination register of an operation is the source register of the following operation, and the following operation is a fully–concurrent operation:

```
FADD.<fmt>    <ea>, FP0
FMOVE.<fmt>   FP0, <ea>     ;FP0 conflicts
```

The second type of register conflict occurs when the destination register of an operation is the destination register of the following operation, and the following operation is a fully–concurrent operation:

```
FADD.<fmt>    <ea>, FP0
FMOVE.<fmt>   <ea>, FP0     ;FP0 conflicts
```

where `<fmt>` is S, D, or X

Instruction	Syntax	Format	No Concurrency	Partial Concurrency
FMOVE	FPm, FPn	X	a	b,c,f
FMOVE	<ea>, FPn	S,D,X		b,c,f
FMOVE	FPm, <ea>	S,D	a	b,d,e
FMOVE	FPm, <ea>	X	a	b

a: Register conflict of FPm with preceding instruction's destination FP data register
b: NAN, unnormalized or denormalized data type
c: Rounding Precision in FPCR set to Single or Double
d: INEX2 bit in FPCR EXC byte is enabled
e: An overflow or underflow occurs
f: Register conflict of FPn with preceding instruction's destination FP data register

Table 6–Fully Concurrent

The next most important optimization rule is to unroll loops. If you properly unroll your loops, than you will be able to eliminate more of the register conflicts. Most loops are designed to do one iteration of a set of instructions. This means that each iteration of the loop is accomplishing one iteration of the set of instructions. If you unroll the loop, then each iteration of the loop can accomplish two or more iterations of the set of instructions. Figures 4 and 5 demonstrate how to unroll your code. The second version (Figure 5) is 25–30 percent faster than the first.

```
                    MOVE.L     #count,D0

LOOPTOP             FMOVE.X    <ea_Xi>, FP3
                    FNEG.X     FP3
                    FETOX.X    FP3
                    FMOVE.X    FP3,FP4             ;conflict
                    FSUB.X     <ea_Xi>, FP3
                    FNEG.X     FP4
                    FSUB.X     #1, FP4
                    FDIV.X     FP4,FP3
                    FNEG.X     FP3
                    FADD.X     <ea_Xi>,FP3
                    FMOVE.X    FP3, <ea_Xi>        ;conflict
                    DBRA       D0, LOOPTOP
```

Figure 4–Newton–Raphson's Method

$$X_{i+1} = X_i + f(X_i)/f'(X_i) : f(X) = exp(-x) - x$$

```
                        MOVE.L      #count,D0
                        FMOVE.D     <ea_Xi>, FP0

LOOPTOP                 FNEG        FP0,FP3
                        FETOX       FP3
                        FMOVE       FP3,FP4            ;conflict
                        FSUB        FP0,FP3
                        FNEG        FP4
                        FSUB.X      #1,FP4
                        FDIV        FP4,FP3
                        FSUB        FP3,FP0
                        DBRA        D0, LOOPTOP

                        FMOVE.D     FP0, <ea_Xi>
```

Figure 5–Newton–Raphson's Method (resister–based, unrolled)

$X_{i+1} = X_i + f(X_i)/f'(X_i) : f(X) = exp(-x) - x$

Conclusion

The last comment which I have to make is for code which is to run during interrupt time. If you plan to use the math coprocessor during interrupt time, you must call `FSAVE` at the start of your routine and `FRESTORE` at the end of your routine. If you do not make these calls and you interrupt another program which is using the FPU, then the other program will not find the FPU in the same state that it was in before the interrupt, and this causes certain death. For more information, refer to Technical Note #235, Cooperating with the Coprocessor.

If you made it this far, and you are still awake, then you should be already to start writing assembly routines for your code which will speed up your math–intensive programs. Just remember that before you try to use the code, you need to check `hasFPU` with a call to `_SysEnvirons`, and if the machine does not have an FPU, then use an alternate SANE version of the math code.

Further Reference:

- *Apple Numerics Manual*, Second Edition
- *Motorola MC68881/MC68882 User's Manual*
- Technical Note M.OV.GestaltSysenvirons —
  Gestatlt and Sysenvirons : a Never Ending Story
- Technical Note M.HW.MathCoProc —
  Cooperating with the Coprocessor