MC68881 Floating-Point Coprocessor as a Peripheral in an M68000 System

INTRODUCTION

The MC68881 floating-point coprocessor is a complete implementation of the proposed IEEE Standard for Binary Floating-Point Arithmetic (Task P754, Draft 10.0). All data formats, data types, operations, modes, conversions, and exception handling required in a conforming implementation of the proposed standard are supported entirely in hardware. Additionally, a full library of fast elementary transcendental functions is implemented in the hardware.

The MC68881 is a high performance floating-point unit designed to interface with the 32-bit MC68020 as a coprocessor; it provides a logical extension of the MC68020 instruction set and register set in a manner which is completely transparent to the programmer. The MC68020 microprocessor implements the M68000 coprocessor interface entirely in hardware. All interprocessor transfers are initiated by the MC68020. During the processing of an MC68881 instruction, the MC68020 transfers instruction and data to the coprocessor via standard M68000 write bus cycles and receives data, requests for service, and status information from the coprocessor via standard M68000 read bus cycles. The MC68881 contains a number of coprocessor interface registers which are addressed like memory by the MC68020 microprocessor. These registers, which are not part of the MC68881 programmer-visible register set, are mapped into CPU address space in an MC68020 system.

The MC68881 distinguishes CPU address space accesses from program and data accesses by a unique function code encoding. The MC68881 registers that are used by the floating-point instructions and hence visible to the programmer are the floating-point registers, FPO-FP7, and the control, status, and instruction address registers.

The MC68881 can also be used as a peripheral processor in systems where the main processor does not implement the M68000 coprocessor interface on-chip (e.g., MC68000, MC68008, and MC68010 systems). Since the coprocessor interface is based solely on standard M68000 bus cycles, it is easily emulated by software in these systems. The MC68881 is considered to be a peripheral processor in these systems because its coprocessor interface registers are memory-mapped in data address space. Two methods of software emulation of the coprocessor interface are possible: 1) M68000 F-line instruction trap (traps are exceptions caused by instructions), or 2) in-line code implemented as either subroutine calls or macros.

When assembled for execution in an MC68020 system, the first word of an MC68881 instruction always has a hexadecimal F (binary 1111) in the most significant nibble (Figure 1). When MC68000, MC68008, or MC68010 processors encounter an F-line instruction, the current processor status is saved, the F-line emulation trap vector is fetched, and instruction execution resumes in the trap handler. When this trap handler is a software emulation of the coprocessor interface, object code containing MC68881 instructions is upward compatible to an MC68020 system without recompiling, reassembling, or relinking. MC68881 instruction performance will significantly increase when such code is ported to a MC68020 system where the coprocessor interface is implemented by on-chip hardware. However, a performance penalty is paid for the upward compatibility provided by the F-line emulation trap method.

The current processor status (either three or four words depending upon the M68000 processor being used) must be pushed onto the stack when the exception is taken, and popped off of the stack when the RTE (return from exception) instruction is executed. More significantly, the trap handler must decode the MC68881 instruction to determine the proper protocol for a given instruction. The performance penalty can become intolerable if the coprocessor emulation trap handler must support all M68000 instruction set and register set in a manner which is completely transparent to the programmer. The MC68020 microprocessor implements the M68000 coprocessor interface entirely in hardware. However, a performance penalty is paid for the upward compatibility provided by the F-line emulation trap method.

The current processor status (either three or four words depending upon the M68000 processor being used) must be pushed onto the stack when the exception is taken, and popped off of the stack when the RTE (return from exception) instruction is executed. More significantly, the trap handler must decode the MC68881 instruction to determine the proper protocol for a given instruction. The performance penalty can become intolerable if the coprocessor emulation trap handler must support all M68000 effective addressing modes. This can be relieved by utilizing only specific addressing modes.

Upward compatibility of object code is not relevant to every M68000 system, especially when it incurs a significant performance penalty. Such systems can emulate the coprocessor interface using in-line code, macros, or subroutine calls. Macros are particularly attractive since they provide the performance of in-line code while allowing the source code to be upward compatible to an MC68020 system via re-compilation or reassembly. This application note provides both a discussion of the coprocessor interface protocol followed by detailed information on both a macro technique and an example of an F-line instruction trap technique for software emulation of the coprocessor interface.

MC68881 AS AN MC68020 SYSTEM COPROCESSOR

COPROCESSOR CONCEPT

The M68000 coprocessor interface is the first general purpose coprocessor interface. The main processor instruction set and internal details are unknown to the coprocessor, and the coprocessor instruction set and internal details are unknown to the main processor. The burden
of decoding the coprocessor instruction is not placed on the main processor, and the coprocessor does not monitor the main processor instruction stream.

The MC68881 instruction set and register set are logical extensions of the MC68020 sets. The MC68020/MC68881 execution model appears to the programmer as if implemented on one chip. All data transfers are performed by the main processor at the request of the MC68881; thus, memory management, bus errors, address errors, and bus arbitration all function as if MC68881 instructions were executed by the main processor. The main processor also performs all effective address calculations and processes all coprocessor-detected exceptions at the request of the MC68881.

The interface is designed to provide the programmer with a sequential instruction execution model even though main processor and coprocessor instruction execution overlap occurs. For some instructions this overlap to enhances throughput.

**MC68020 INTERPROCESSOR BUS TRANSFERS**

The coprocessor interface is based upon standard M68000 asynchronous read and write bus cycles. No new bus signals are required by the MC68020 to initiate a floating-point instruction, and the MC68020 and MC68881 may run at different clock rates. The MC68881 is connected like a peripheral to the main processor and requires one extra pin, chip select (CS), in order to be accessed. Chip select is generated from the MC68020 function codes and address lines, similar to the way peripheral chip selects are generated. All other MC68881 pins connect to signals present on the M68000 Family processors.

The MC68881 has a set of coprocessor interface registers by which the main processor and coprocessor communicate. When performing a coprocessor access, the MC68020 outputs a 111 on its function code lines thus specifying CPU address space. Therefore, the MC68881 coprocessor interface registers are memory-mapped into CPU address space and do not infringe upon data or program address space. A portion of this CPU address space is allocated for coprocessor communication. The MC68020 outputs 0010 on bits 19-16 of the address bus for coprocessor accesses as shown in Figure 2. The MC68020 also outputs the Cp-ID field (bits 11-9 from the first word of the coprocessor instruction of Figure 1) on bits 15-13 of the address bus. The main processor selects the appropriate coprocessor interface register within the selected coprocessor via bits 4-0 of the address bus as shown in Figure 3. The MC68881 chip select is therefore based on these elements: the MC68020 three function code outputs, the Cp-ID field (address bits 15-13 of the address bus), and the CPU space type field (bits 19-16 of the address bus).

Notice, the MC68020 handles only four CPU address space cycles:

<table>
<thead>
<tr>
<th>CPU Space Type</th>
<th>CPU Space Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field (A19-A16)</td>
<td>Breakpoint Acknowledge</td>
</tr>
<tr>
<td>0000</td>
<td>Access Level Control</td>
</tr>
<tr>
<td>0010</td>
<td>Coprocessor Communications</td>
</tr>
<tr>
<td>1111</td>
<td>Interrupt Acknowledge</td>
</tr>
</tbody>
</table>

Therefore, when decoding the chip select for the MC68881, only two bits are needed (A18 and A17) to distinguish a coprocessor operation from the other CPU address space operations. A suggested method for connecting the MC68881 to the MC68020 is illustrated in Figure 4.

Figure 5 illustrates the connection of an MC68881 to an MC68000 or MC68010 as a peripheral processor over a 16-bit data bus. The MC68881 is configured to operate over a 16-bit data bus when the SIZE pin is connected to VCC, and the A0 pin is connected to ground (GND). The sixteen least significant data pins (D0-D15) must be connected to the sixteen most-significant data pins (D16-D31) when the MC68881 is configured to operate over a 16-bit bus (i.e., connect D0 to D16, D1 to D17,...and D15 to D31). The DSACK1 pin of the MC68881 is connected to the DTACK pin of the main processor, and the DSACK0 pin is not used.
Figure 2. Coprocessor CPU Space Decoding (Function Code = 111)

<table>
<thead>
<tr>
<th>ADDRESS BUS</th>
<th>31</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>13</th>
<th>12</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 1 0</td>
<td>x  x</td>
<td>x 0 0 0 0 0 0 0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **COPROCESSOR COMMUNICATIONS**
- **COPROCESSOR IDENTIFICATION**
- **COPROCESSOR INTERFACE REGISTER SELECT**

*Denotes required registers for minimum support of all coprocessor instruction types. The MC68881 implements these registers.
**Denotes additional registers implemented by the MC68881.

Figure 3. Coprocessor Interface Registers

**Figure 4. Suggested MC68020 to MC68881 Connections**
When connected as a peripheral processor, the MC68881 chip select (CS) decode is system dependent. If the MC68000 is used as the main processor, the MC68881 CS must be decoded in the supervisor or user data spaces. However, if the MC68010 is used for the main processor, the MOVES instruction may be used to emulate any CPU space access that the MC68020 generates for coprocessor communications. Thus, the CS decode logic for such systems may be the same as in an MC68020 system, such that the MC68881 will not use any part of the data address spaces.

Figure 5. 16-Bit Data Bus Peripheral Processor Connection

Figure 6 illustrates the connection of an MC68881 to an MC68008 as a peripheral processor over an 8-bit data bus. The MC68881 is configured to operate over an 8-bit data bus when the SIZE pin is connected to ground (GND). The eight least-significant data pins (D0-D7) must be connected to the twenty-four most-significant data pins (D8-D31) when the MC68881 is configured to operate over an 8-bit data bus (i.e., connect D0 to D8, D16, and D24; D1 to D9, D17, and D25;...and D7 to D15, D23, and D31). The DSACK0 pin of the MC68881 is connected to the DTACK pin of the MC68008, and the DSACK1 pin is not used.

Figure 6. 8-Bit Data Bus Peripheral Processor Condition
When connected as a peripheral processor, the MC68881 chip select (CS) decode is system dependent, and the CS must be decoded in the supervisor or user data spaces.

**MC68881 INSTRUCTION DEFINITION**

The MC68881 instructions can be subdivided according to the type of coprocessor operation performed: general, branch, save, restore, or conditional. Each instruction, when assembled, consists of from one to eight words (Figure 7). The first word (operation word) always has a hexadecimal F (11 11) in the high-order nibble as seen in Figure 1. The type field (bits 8-6) of the operation word indicates the coprocessor instruction type. For instruction types which require an effective address (general, save, restore, and conditional), the type-dependent field of the operation word specifies the effective addressing mode. For the conditional instruction type, this field specifies the condition to be evaluated by the coprocessor: the conditional predicate (CPRED).

<table>
<thead>
<tr>
<th>15 13 12 10 9 6 5 0</th>
<th>OPERATION WORD</th>
<th>COPROCESSOR COMMAND WORD (IF ANY)</th>
<th>EXTENSION WORDS (IF ANY, 1-6 WORDS)</th>
</tr>
</thead>
</table>

Figure 7. General Format of Coprocessor Instruction

The format of the second word of the coprocessor instruction varies with different instruction types. For the general instructions, the second word is the coprocessor command word specifying the function to be performed by the coprocessor. The MC68881 has been designed such that all general type operations are specified by a single command word. In order to process a conditional instruction type, the main processor must deliver the conditional predicate (CPRED) to the MC68881 for evaluation. Since the type-dependent field of the operation word may specify the effective addressing mode, the CPRED is found in the low-order six bits of the second word. The additional word(s) following the second word in Figure 8 is the extension to the effective addressing mode or the immediate operands present in the instruction. In the branch, save, and restore instructions, all information needed to initiate processing in the coprocessor is found in the operation word. Thus, the extension word(s) (if any) directly follows the operation word (no coprocessor command word).

<table>
<thead>
<tr>
<th>15 13 12 10 9 6 5 0</th>
<th>OP-CLASS</th>
<th>Rx</th>
<th>Ry</th>
<th>EXTENSION FIELD</th>
</tr>
</thead>
</table>

Figure 8. MC68881 General Instruction Command Word

All of the MC68881 arithmetic, move, move multiple, move constant, and transcendental instructions are of the general type. All general type instructions are initiated by the main processor writing the coprocessor command word (the second word of the instruction) to the MC68881 coprocessor interface command register. The format of the MC68881 command word is shown in Figure 8. General type instructions are broken down into groups, called op-classes, based on the function of the instruction and argument location (external or internal to the coprocessor). The values Rx, Ry, and the extension field depend on the specific op-class. For instance, the values required for a floating-point register to floating-point register operation are as follows: Rx is the source floating-point register, Ry is the destination floating-point register, and the extension field is the operation to be performed (add, move, sin, etc.). Table 1 lists the op-classes, their definitions, and their respective Rx, Ry, and extension fields.

For the branch and the conditional type instructions, the main processor initiates processing by writing the conditional predicate (CPRED) from the six low-order bits of either the first or second word of the instruction, respectively, to the coprocessor for evaluation. These conditional predicates are found on lines 82-97 of the EQUATE table listed in APPENDIX A MACROS at the back of this document.

In the case of an operating system context switch, the coprocessor internal state can be saved by the FSAVE instruction. This MC68881 instruction only saves the invisible state of the machine (that which is not normally available to the user). Thus all control, status, instruction address, and floating-point data registers (the user-visible registers) must be saved by the user. Only the registers available to the programmer need to be stored. To initiate the FSAVE instruction sequence, the main processor reads a word (the save format word) from the save interface register location of the coprocessor. The save format word provides the status of the coprocessor state (the null state, the idle state, or the busy state) and also the size (0 bytes, 24 bytes, or 190 bytes respectively) of the internal state of the machine to be saved. The save format word is written to the effective address by the main processor at the end of the instruction execution no matter which state the coprocessor is in.

The restore type instruction which restores a previously saved state is initiated by the main processor writing a save format word to the coprocessor interface restore register. This informs the coprocessor which coprocessor internal state is to be restored. If visible registers are saved after the execution of the FSAVE command, they must be restored before the execution of the FRESTORE instruction.

When executing any MC68881 instruction, the MC68020 follows a basic protocol. First, the coprocessor information (command word, conditional predicate, or format word) is written to the appropriate coprocessor interface register by the main processor (the FSAVE instruction is initiated by a read). The main processor then reads the appropriate coprocessor interface register to acquire the coprocessor status and any service requests. The coprocessor may indicate that it is busy processing a previous instruction and ask the main processor to query again. (This is the mechanism by which the sequential instruction execution is maintained because the coprocessor...
### Table 1. Command Word Fields of General-Type Instructions

<table>
<thead>
<tr>
<th>Op-Class Field</th>
<th>Rx Field</th>
<th>Ry Field</th>
<th>Extension Field</th>
<th>Instruction Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Source FP Data Register</td>
<td>Destination FP Data Register</td>
<td>Operation to Perform</td>
<td>FP Data Register to FP Data Register Instructions</td>
</tr>
<tr>
<td>001</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Unused, Reserved</td>
</tr>
<tr>
<td>010</td>
<td>Source Data Format (see Note 2)</td>
<td>Destination FP Data Register</td>
<td>Operation to Perform</td>
<td>External Operand to FP Register</td>
</tr>
<tr>
<td>010</td>
<td>111</td>
<td>Destination FP Data Register</td>
<td>Constant ROM Offset</td>
<td>Move Constant to FP Data Register</td>
</tr>
<tr>
<td>011</td>
<td>Destination Data Format (see Note 2)</td>
<td>Source FP Data Register</td>
<td>00000000 (Unless Packed BCD, see Note 2)</td>
<td>Move out FP to External Destination</td>
</tr>
<tr>
<td>100</td>
<td>FPcr Select (see Note 1)</td>
<td>000</td>
<td>00000000</td>
<td>Move/Move Multiple to Control Register</td>
</tr>
<tr>
<td>101</td>
<td>FPcr Select (see Note 1)</td>
<td>000</td>
<td>00000000</td>
<td>Move/Move Multiple from Control Register</td>
</tr>
<tr>
<td>110</td>
<td>A/D S/D 0 (see Note 3)</td>
<td>00m (see Note 4)</td>
<td>Contains Register List</td>
<td>Move Multiple FP Data Register to MC68881</td>
</tr>
<tr>
<td>111</td>
<td>A/D S/D 0 (see Note 3)</td>
<td>00m (see Note 4)</td>
<td>Contains Register List</td>
<td>Move Multiple FP Data Register from MC68881</td>
</tr>
</tbody>
</table>

**NOTES:**

1. **FPcr**
   - Floating-Point Control Register
   - 000: Reserved
   - 001: FPIAR (Instruction Address Register)
   - 010: FPSR (Status Register)
   - 100: FPCR (Control Register)
   - 101: FPCR and FPIAR
   - 110: FPCR and FPSR
   - 111: FPCR, FPSR, and FPAIR

2. **Value**
   - Data Format
   - 000: Long Word Integer
   - 001: Single Precision Real
   - 010: Extended Precision Real
   - 011: Packed Decimal Real
   - 100: Word Integer
   - 101: Double Precision Real
   - 110: Byte Integer
   - 111: For Op-Class = 01, Packed Decimal Real with Dynamic k Factor specified as CPU Data Register by Extension Field Encoding, rrr0000

3. **A/D = 0**
   - Most Significant Bit of Register List Selects FP7
   - **A/D = 1**
   - Most Significant Bit of Register List Selects FP0
   - **S/D = 0**
   - Bit Mask in Extension Field
   - **S/D = 1**
   - Bit Mask in CPU Data Register Selected by Extension Field, 0rrr0000

4. When Bit Mask is Transferred to Command Word: "m" is the Most-Significant Bit of the Register List

The coprocessor may indicate an exceptional condition and request the main processor to begin exception processing by providing the proper exception vector. The coprocessor may request additional service of the main processor, for example, evaluating effective address and transferring data through the coprocessor interface operand register. Finally, the coprocessor may indicate to the main processor that no further servicing is required.

**INTERFACE REGISTERS**

The MC68881 contains a number of interface registers which are memory-mapped within the MC68020 CPU address space. These are the registers identified by "**" and "***" in Figure 8. The coprocessor registers are memory-mapped into the main processor's system like any other peripheral, although they are accessed in a different address space.

The main processor initiates communication with the MC68881 by writing to (or reading from) a specific 16-bit
register, which is memory-mapped into the system. The specific register chosen depends on the instruction type. A general instruction coprocessor command word is written to the command register, and a branch or conditional instruction CPEm is written to the condition register. The main processor must read the save interface register to initiate the sequence of saving the internal state. To restore this state, the main processor writes to the restore interface register. The save and restore functions support virtual memory, demand paging, and task time-slicing.

After the initial write to the register in the general, conditional, and branch instructions, the response register is read by the processor to determine its next action (e.g., come-again or evaluate effective address and transfer data to/from the coprocessor).

An example of the communication sequence may be demonstrated with the memory to floating-point register add instruction. The main processor first writes the coprocessor command to the command interface register and queries the response register until requested to pass data. At which time, the host reads the data from memory and writes it to the operand register, four bytes at a time. The response register is re-read until the coprocessor signals the main processor to stop. The MC68020 is then free to process the next main processor instruction, while the MC68881 performs the floating-point add on the data. In this case, as in all normal processor/coprocessor communications, the host processor processes only the requests specified in the coprocessor primitives until released by the MC68881.

The MC68881 utilizes only three registers other than those previously mentioned. These are: the register select register used in the move multiple instructions, the instruction address register used only when exceptions are enabled, and the control register used by the main processor either to handshake the processing of the coprocessor exception or to abort invalid coprocessor service requests.

**RESPONSE PRIMITIVES**

Response primitives are service requests from the coprocessor to the main processor. These primitives have capabilities which allow for the synchronization of the main processor/coprocessor general, conditional, and branch instruction executions. Also the coprocessor may request services such as external memory accesses, transfer of data, and exception processing by the main processor. Figure 9 is a list of all possible coprocessor response primitives recognized by the MC68020. If the come-again (CA) bit is set, the main processor processes the primitive and then reads the response register again to seek further service requests. If the CA bit is not set, the main processor is released from further services (except when the MC68020 is in trace mode). If the PC bit is set, the main processor writes the program counter position of the first word of the coprocessor instruction to the instruction address interface register prior to performing the requested service. In the event that the MC68881 generated a trap exception, this PC value is required by the exception trap handler to determine which instruction caused the exception.

The MC68881 utilizes only six of the possible responses. These are the primitives noted by an “*” in Figure 9. The transfer multiple coprocessor registers primitive allows the transfer of multiple coprocessor registers to or from memory. The dr bit specifies in which direction the transfer is to be made: from the coprocessor to memory if the bit is set, or from the memory into the coprocessor if the bit is cleared. The transfer single main processor register requests the main processor to transfer the contents of one of its registers to or from the coprocessor. If D/A equals one, the transferred register is an address register. If D/A equals zero, a data register is transferred. The register number is located in the register field.

The evaluate effective address and transfer data primitive requests the main processor to evaluate the effective address specified by the floating-point instruction and to transfer data to or from that address (from or to the coprocessor). The valid EA type field indicates which addressing modes are valid for the transfer, while the length field gives the number of bytes to be transferred.

The Null primitive alerts the main processor to the coprocessor status after all other service requests (excluding exception requests) have been granted by the main processor. If the CA bit is set, the main processor queries the response register until the bit is cleared, at which time the main processor is released by the coprocessor. Even though the main processor may be signaled for release (when CA equals zero), it can still pass the program counter (PC equals one) and/or accept pending interrupts (IA equals one and CA equals one). The processing finished (PF) bit is a status bit which indicates whether or not the coprocessor has finished its instruction. The MC68020 tests the bit only while in trace mode to ensure that the instruction processing is complete. In the case of a conditional instruction the null primitive also contains a T/F bit (bit 0). This bit is tested by the main processor to determine whether or not the conditional predicate is true (one) or false (zero).

For all the MC68881 primitive responses, the CA bit is always set (CA equals one) with the exception of the null and exception request primitives. Both the take pre-instruction exception and the take mid-instruction exception primitives contain the exception vector number. Pre-instruction exceptions occur under two conditions: 1) after no further information is needed from the main processor in a previous floating-point instruction, and 2) before the coprocessor begins processing the present instruction. These pre-existing exceptions represent either an illegal command word for the present instruction, or the termination of a previous instruction with an exception. This delayed reporting allows for synchronization between the host and the coprocessor in the event of any pre-existing exceptions. A floating-point register to memory move (op-class 001) operation is the only instruction capable of generating a mid-exception primitive. It is detected in the last read of the response register during instruction execution because the MC68881 performs the floating-point calculation and releases the main processor only after the data transfer to memory. With the memory to floating-point register (op-class 010) or the floating-point register to floating-point register (op-class 000), the main
**BUSY**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*TRANSFER MULTIPLE COPROCESSOR REGISTERS*

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>dr</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TRANSFER STATUS AND SCANPNC**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**SUPERVISOR CHECK**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**TAKE ADDRESS AND TRANSFER DATA**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

*TRANSFER MULTIPLE MAIN PROCESSOR REGISTERS*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**TRANSFER OPERATION WORD**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

*NULL*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**EVALUATE EFFECTIVE ADDRESS AND TRANSFER ADDRESS**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | 0  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |

*TRANSFER SINGLE MAIN PROCESSOR REGISTER*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**TRANSFER MAIN PROCESSOR CONTROL REGISTER**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

*Figure 9. Coprocessor Response Primitives (Sheet 1 of 2)*
**TRANSFER TOP-OF-STACK**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>dr</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**TRANSFER FROM INSTRUCTION STREAM**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | 0  | 0  | 1  | 1  | 1 | 1 |   |   |   |   |   |   |   |   |   |

**EVALUATE EFFECTIVE ADDRESS AND TRANSFER DATA**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | dr | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |

**TAKE PRE-INSTRUCTION EXCEPTION**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0  | PC | 0  | 1  | 1  | 1 | 0 | 0 |   |   |   |   |   |   |   |   |   |

**TAKE MID-INSTRUCTION EXCEPTION**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0  | PC | 0  | 1  | 1  | 1 | 0 | 1 |   |   |   |   |   |   |   |   |   |

**TAKE POST-INSTRUCTION EXCEPTION**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0  | PC | 0  | 1  | 1  | 1 | 1 | 0 |   |   |   |   |   |   |   |   |   |

**INVALID (RESERVED)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | 0  | 0  | 1  | 1 | 1 | 1 | 1 | x | x | x | x | x | x | x | x |

**WRITE TO PREVIOUSLY EVALUATED EFFECTIVE ADDRESS**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | 1  | 0  | 0  | 0 | 0 | 0 |   |   |   |   |   |   |   |   |   |

**INVALID (RESERVED)**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| CA | PC | 1  | 1  | 1 | 0 | 0 | 0 | x | x | x | x | x | x | x | x | x |

**NOTES:**
- dr: 0 = Into Coprocessor
  1 = Out of Coprocessor
- D/A: 0 = Data
  1 = Address
- * Supported by MC68881
- Valid EA Types:
  - 000 = Control Alterable
  - 001 = Data Alterable
  - 010 = Memory Alterable
  - 011 = Alterable
  - 100 = Control
  - 101 = Data
  - 110 = Memory
  - 111 = Any Mode Allowed (No Restrictions)

Figure 9. Coprocessor Response Primitives (Sheet 2 of 2)
processor is freed to execute the next instruction as the coprocessor performs the requested operation.

The save, restore, and move multiple instructions do not generate exceptions.

SOFTWARE TO EMULATE THE C0PROCESSOR INTERFACE

DEFINITIONS AND ASSUMPTIONS

In order to utilize the floating-point coprocessor in a MC68000/MC68008/MC68010 system, a software emulation of the coprocessor interface must be developed. There are two possible methods of software emulation: 1) in-line code such as macros or subroutine calls, or 2) the M68000 F-line emulation trap.

The coprocessor must reside in a different address space than in an MC68020 system. When the MC68020 accesses the coprocessor, it does so in CPU space by outputting a 111 on the function code lines. The equivalent function codes generated on either MC68000, MC68008, or MC68010 signify an interrupt acknowledge bus cycle, i.e. these processors do not implement CPU space. Thus the MC68881 must be accessed as a peripheral with the coprocessor interface registers memory-mapped in data address space in these systems, not in CPU space.

To accommodate the practical use of this application, the demonstration software will perform a floating-point instruction in the fastest way possible while not violating the safety provided in the IEEE standard and the MC68881. To do this, a number of factors are first considered. The interprocessor protocol used with all instruction type classes may be minimized to include only those checking mechanisms necessary to perform the basic function. The most significant simplification is to restrict the use of the ENABLE byte of the floating-point control register. Each bit of this byte represents a type of coprocessor detected exception. Figure 10 identifies all coprocessor detected exceptions, their corresponding vector number (passed to the MC68020 in an exception primitive), and their position in the ENABLE byte. Two exceptions are not represented in this byte: the protocol violation and the illegal coprocessor command. This demonstration software ignores all take exception responses from the MC68881 (generated by the conditions specified in the ENABLE byte) which reduces the overhead required for their recognition in the software. An exception primitive, due to a floating-point operation, will be generated only if the MC68881 records an exception and the corresponding bit in the ENABLE byte is also set. Thus an exception primitive cannot be generated if the default condition ($00) in the ENABLE byte is never altered by the user. This eliminates the need for monitoring the response register for pre-instruction or mid-instruction exceptions. If a coprocessor-detected exception occurs, it will never be detected by the main processor.

Note that two coprocessor-detected exceptions, the protocol violation and the illegal coprocessor command, are not represented by a bit in the ENABLE byte. A protocol violation occurs anytime communication between the main processor and coprocessor is improper and is reported by the MC68881 as a mid-instruction exception. An illegal coprocessor command is a coprocessor command not implemented by the MC68881 and is reported as a pre-instruction exception. Therefore, by not checking for exceptions, a protocol violation or illegal coprocessor command may occur without being detected.

Another consequence of the previously mentioned simplification is that the program counter will never be requested of the main processor if exceptions are disabled. The PC bit of the responses will never be set if the bits in the ENABLE byte are not set. Therefore, the overhead for testing of this bit can be saved.
IN-LINE CODE, MACROS, OR SUBROUTINES

The first approach discussed is the one to directly drive the MC68881 as a peripheral from the user program in user data space. This approach is used when speed of the MC68881 instruction execution is more important than upward compatibility of the object code. Two methods are available to drive the peripheral in user data space: in-line code (macros) or run-time libraries (subroutines). The trade-off between the two approaches concerns time versus space. Each time a macro is used the length of the source code increases by the size of the macro. When a subroutine is called, the overhead of the subroutine call and the execution of the RTS instruction must be incurred. No F-line trap is taken in either method, which saves the time to perform the corresponding stacking and instruction decoding.

Macros allow the coding of a repeated pattern of instructions which may contain variable entries at each iteration of the pattern. By incorporating the use of macros with conditional assembly instructions, some of the necessary floating-point instruction decode can be done during assembly time, reducing the run-time overhead. Assuming that the assembler used has the ability to pass parameters and manipulate these within the macro during assembly, the programmer need not generate the code to parse the floating-point instruction to detect the addressing mode used — it is passed directly by the macro into the assembled code. Subroutines can also act as an extension to the in-line routine to perform functions common to each instruction type (exception handling routines, error checking routines, etc.). The major advantages of using these methods over the F-line trap approach are: 1) the time saved by not taking the F-line trap, and 2) the time saved by assembly time instruction decode.

This application note includes software for the macro approach to drive the code in user data space. The same code would be pertinent if implementing a run-time library approach.

There are different ways to define the macros to drive the MC68881 depending on the particular application. The ideal method would be to write a macro supporting each floating-point instruction employed to achieve source code compatibility with the MC68020. In order to support the complete MC68881 instruction set, the library of macro definitions would be quite large. For conciseness, multiple coprocessor instructions are consolidated into single macro instructions, collected by the method of operand transfer required of the instruction (located in APPENDIX A MACROS). As an example, all MOVE-INS, MOVE-OUTS, etc. have their own macros corresponding to their respective directions and precisions. Most MC68881 instructions are supported, but source code compatibility with the standard MC68881 instruction set is lost due to consolidation of specific instructions into single macros. The conditional trap and the move multiple coprocessor system register instructions are not included in this example set.

Every macro requires at least the same amount of information supplied by the programmer as the represented MC68881 floating-point instruction and, in some cases, more information. This data is passed to the macro by parameters. An example of a general instruction class macro call is:

MACRONAME FUNCTION, SOURCE, DESTINATION

MACRONAME specifies the method of operand transfer, the FUNCTION is the general operation to be performed, the SOURCE is the location of the source operand, and the DESTINATION is the location of the destination operand. Different macros request different information of the user. For instance, the FSAVE instruction only requires one parameter to be passed to the macro. Generally, each macro follows a similar format and is described in Functional Description. The methodology of these macro definitions can be employed by the programmer who wishes to develop a separate macro for each instruction.

Functional Description

The following paragraphs provide the programmer with information on how to use the macros. All of the macros for the general instruction which transfer a source operand to the coprocessor (move-in) are listed below:

MEMREGB Function, SourceEA, FPn
MEMREGW Function, SourceEA, FPn
MEMREGL Function, SourceEA, FPn
MEMREGS Function, SourceEA, FPn
MEMREGD Function, (An), FPn
MEMREGX Function, (An), FPn
MEMREGP Function, (An), FPn

They only differ by the precision of the data transferred. Each macro of this class transfers a source operand (specified by SourceEA or (An)) of a specific precision to the MC68881 and performs the operation specified by function. A list of the functions describing the supported MC68881 instructions and their functions can be found on lines 25-61 of the EQUATE table of the demonstration software found in APPENDIX A MACROS.

The macros which transfer data from a single floating-point register to an effective address (move out) are:

REGMEMB FMOVE, FPm, DestinationEA
REGMEMW FMOVE, FPm, DestinationEA
REGMEML FMOVE, FPm, DestinationEA
REGMEMS FMOVE, FPm, DestinationEA
REGMEMD FMOVE, FPm, (An)
REGMEMX FMOVE, FPm, (An)
REGMEMP FMOVE, FPm, (An), [k-factor]

The MC68881 FMOVE command is the only MC68881 instruction supporting this direction of transfer. These macros request the passing of the instruction as a parameter to be consistent with the other macros. The k-factor, requested by REGMEMP, may be passed in data register D0 or as immediate data. Thus:

[k-factor] = D0
or
[k-factor] = #xxxx

A k-factor in the range +1 to +17 indicates the desired number of significant digits in the decimal mantissa after conversion to packed decimal format. A k-factor in the range -64 to 0 indicates the desired number of significant digits to the right of the decimal point in a fixed-point format.
The general instruction macro which performs "floating-point register to floating-point register" operations is:

REGREG Function,FPn,FPMn,[FPq]

Since the MC68881 performs all data manipulations in extended precision (no user-specified precisions), only one macro is needed to support these general instructions. The new parameter introduced, [FPq], supports the one special case general instruction "FSINCOS" which generates the sine placing it in FPM and generates the cosine and placing it in FPq.

The conditions supported on-chip by the MC68881 are available to the programmer from the coprocessor ROM with the macro:

FMOVEROM #CC,Fpn

CC is the hex number representing the constant to be accessed. A list of the constants and their corresponding identification numbers is found on lines 660-681 of the EQUATE table in the demonstration software found in APPENDIX A MACROS.

The two macros which move data into or out of the coprocessor control, status, and instruction address registers are:

MOVINCSI SourceEA,Register
MOVOUCSI Register, DestinationEA

MOVINCSI moves data into the control, status, or instruction address registers, and MOVOUCSI moves data out. The register field is specified by CONTROL, STATUS, or IADDRESS corresponding to the register to be transferred.

As the M68000 microprocessors use the MOVEM instruction to move multiple registers into and out of memory, the MC68881 also supports moving multiple floating-point registers. The macros which support the movement of multiple floating-point data registers are:

FMOVEMMR SourceEA,fp0,fp1,fp2,fp3,fp4,fp5,
fp6,fp7,Postincrement
FMOVEMRM fp0,fp1,fp2,fp3,fp4,fp5,
fp6,fp7,DestinationEA, Predecrement

Each parameter of the fp0, fp1, ..., fp7 string represents the selection bit for that floating-point register. If a register is to be moved, then the corresponding parameter in the macro call must be set to a one (otherwise, the bit must be set to a zero). If the programmer decides to use the indirect addressing with postincrement mode in the FMOVEMMR macro, then the postincrement field must be set to a Y, otherwise, this parameter must be set to a N. The same system applies to the FMOVEMRM macro with respect to the predecrement field (Y if addressing mode is used, N if not).

The no operation or synchronizing instruction is supported by the macro:

FNOPP

No parameters are necessary to perform this function.

Only one macro is needed to support the branch instruction class, the conditional branch:

FBCC.[Size] Condition,Label

The size specification allows the macro to distinguish between a long branch and a short branch. If the size is not specified, the default is long. The user must specify the condition to be tested in the condition field. A list of the conditions and their corresponding label is found on lines 67-98 of the EQUATE table found in APPENDIX A MACROS. If after execution, the condition is satisfied, the macro will cause a branch to label.

The two macros which support the MC68881 decrement and branch and the conditional set instructions are:

FDGCC Condition,Dn,Label
FDGCC Condition,Label

In both macros, the condition and label parameters serve the same purpose as those of the FBCC macro. The conditions are listed on lines 67-98 of the EQUATE table found in APPENDIX A MACROS. Note that Dn of FDGCC cannot be equal to D0 because it is used as a work register by the macro. Any other data register may be used.

The save or restore of the internal or invisible state of the coprocessor is executed by the macros:

FSAVEST (An)
FRESTRST (An)

To ensure proper restoration of the MC68881 state, the FRESTRST command should be used only on data stacked by the FSAVEST command. One exception to this rule is for a software reset. A software reset of the MC68881 occurs if a null save format word is stored on the stack and then restored into the MC68881. This can be used by the operating system to initialize the MC68881 when starting a new task.

Theory of Operation

The following paragraphs provide information on how to develop macros for the user who will either create his own, or need to modify the demonstration software to suit a particular application. The coprocessor recognizes each coprocessor instruction by the specific bit pattern written to the various coprocessor registers. The save instruction is an exception because it is initiated with a read from the save interface register. For each coprocessor instruction type class, a unique format for the bit pattern exists which is the basis for instruction grouping into macros. A detailed description of the macro development as well as a general discussion of each type instruction class follows.

All macros are developed following a simple protocol: 1) know the bit pattern of the information to be written to the coprocessor (the instruction to be performed) and write it to the appropriate interface register, 2) test for the known possible responses from the coprocessor, 3) perform requested operation (if any), and 4) test for the release of the main processor. If it is necessary to add exception detection, the programmer must add software to compare the response to the appropriate pre-exception or post-exception bit pattern and call a user-specified exception processing macro or subroutine if the result is positive.

Each MC68881 instruction follows a specific protocol starting from the write of the operation until the receipt of the null primitive. MEMREGn (n=B, W, L, S, D, X, P) and MOVINCSI follow the sequence shown in Figure 11. REGMEMn (n=B, W, L, S, D, X, P) and MOVOUCSI follow the sequence shown in Figure 12. Instructions represented by these REGMEMn macros generate at least one null come-again response after the initial write to the
command interface register to allow the coprocessor to perform the specified operation before the transfer of data to memory. The FMOVEMMR and FMOVEMRM macros follow the protocols seen in Figures 13 and 14, respectively. The demonstration software only supports the static forms of the instruction, i.e. the bit mask is transferred in the instruction rather than in a main processor data register (as in the dynamic form). Thus, Figures 13 and 14 represent only the static forms of the move multiple floating-point register instructions. The REGREG macro follows the sequence of Figure 15. The FDSCC, FBCC, and FSCC macros basically execute the same code except for the function to be performed after evaluating of the result of the conditional test as seen in Figure 16. Figures 17 and 18 represent the protocol followed by the FSAVE and FRESTORE instruction, respectively.

The macro detail will be explained by discussing an example of a general coprocessor instruction, the MEMREGW (lines 435-470) macro. This macro performs a floating-point operation on a word datum at an effective address pointed to by source EA address, and leaves the result in a specified floating-point register. The macro call takes the form:

```
MEMREGW Function, SourceEA, FPn
```

The function, SourceEA, and FPn are all parameters passed to the macro. Function is the operation of the MC68001 instruction, and FPn (n equals 0-7) is the specific floating-point register used. Both of these parameters represent a binary bit pattern. Thus, the need for an EQUATE table arises. An “EQU” statement (refer to the M68000 Assembler Manual) defines a symbol as a binary value when referenced anywhere in the source code. In the EQUATE table (lines 25-61) found in APPENDIX A, all general floating-point instructions are assigned the appropriate bit pattern (e.g. FMOVE EQU $00) to represent the extension field of the command word in Figure 8. Also, as seen on lines 119-126, each floating-point register (FPn) is equated to its corresponding numerical value (e.g. FP6 EQU $06). The other parameter passed to the MEMREGW macro is the effective address (SourceEA).
Since parameters are separated by commas in the format of these macro calls, the indexed register indirect with offset addressing mode, (di|An,Dn)), cannot be passed as a single parameter. The comma between An and Dn causes the assembler to see this effective address as two parameters; Therefore, the macro will be passed one additional parameter for this case. Anticipating this case, when four parameters are passed in, the macro simply recombines the appropriate two parameters and reconstructs the effective address as a single parameter. The first line of this macro (line 452) is a conditional assembly command where the assembler tests for the occurrence of a fourth parameter, signifying the use of the indexed address mode. This test is done by comparing this parameter ('4') against a null character string (""), if a fourth parameter is present, a separate routine (lines 461-469) will be used to combine parameters '2 and '3 to reconstruct the effective address parameter.

Once the assembler has chosen which routine to assemble, the next task entails developing the command word shown on line 453 of the listing:

```assembly
MOVEM.W
#$5000+(3<<7)+01,MC68881+COMMAND
```

This task demonstrates how the command word is formed for all the effective addressing modes except the
indexed modes. The assembler, instructed by the arithmetic operator +, adds the three fields to generate the proper command word. The immediate data is the command word developed by the addition of the isolated fields seen in Figure 8. The command word base, 5000, represents the op-class 2 and data format for a word operand (RX). Each macro for a general instruction will have a unique command word base (shown in Table 1) specifying the op-class and data precision. A good understanding of the command word structure in Table 1 is helpful in developing general instruction macros.

The second field, Ry in Figure 8, is added to the command word by the assembler and represents the number of the floating-point register used in the transfer. This parameter is passed to the macro by the programmer as the third parameter, FPn. The symbol, <<, causes the assembler to shift the value of the third parameter to the left seven bits, placing it in its proper position in the command word.

The third field of the summation is the extension field which specifies the binary representation of the instruction to be performed. These representations are shown in the demonstration software in lines 25-61.

Note, when the addressing mode is indexed, the extension field remains the first parameter passed to the macro, but the Ry field becomes the fourth parameter.
and is accommodated by line 462. When using the indexed addressing mode, the assembler needs to recreate a comma (in the indexed format) which was not passed with the parameters. In line 466, a comma is placed between the second and third parameters passed to the macro recreating the proper EA.

The command word is always written to the absolute address of the coprocessor command interface register. The demonstration software uses MC68881 to represent the base address of the coprocessor interface registers in data space. Specific interface registers are referenced by adding the displacement of that particular register to the base address. Consequently, each register has a symbol equal to the appropriate displacement in the EQUATE table (e.g., COMMAND EQU $0A) on lines 104-114.

Note that if the MC68881 is mapped into the highest page or the lowest page in the address map, the macros can use short absolute addressing mode instead of long absolute addressing mode. This will allow the macros to assemble into smaller object codes and may execute faster since the processor spends less time fetching extension words.

Once the command word is written to the command register, only two responses can be read from the response register: 1) null come-again and evaluate effective address and 2) transfer data. Only two responses will occur because exceptions are not allowed. By only testing the response register (line 454) for the null come-again ($8900), the main processor will pass the data when it reads any response other than the null come-again response.

This macro, as well as all other macros except FSA-VEST and Frestrst, must test the response register for the coprocessor release of the main processor. This service protects against spurious protocol violations. Protocol violations are unexpected accesses to the MC68881 interface registers. For example, the coprocessor may be expecting data to be written to the operand register but instead receives a write to the command register. A spurious violation occurs when an expected response register access occurs sooner than expected in systems where the processor and coprocessors are running at different clock speeds. Since exceptions are assumed to be disabled by the macros, the CA bit is monitored to determine the coprocessor state. When CA is set to zero, the main processor is released. The following instructions perform this function throughout the macro definitions:

```
; NULREL TST:.. MC68881 + RESPONSE
BMI.S ; NULREL
```

In summary, this example defines the sequence to be executed in all macros of the general instructions op-class 010. Each macro causes the main processor to write the move-in operation to the command register and to read the response register until asked to pass the data. After evaluating the effective address of the data and writing it to the operand register, the main processor rereads the response register until released by the coprocessor.

The packed BCD, double precisions, and extended precisions operations would require the use of several other conditional assembly instructions to support all the addressing modes that the byte, word, long word, and single precision macros allow. These instructions are necessary due to the fact that multiple accesses from memory are required to transfer data through the 32-bit wide operand register. To simplify this application, these three precisions, MEMREGD, MEMREGX, and MEMREGP, are only supported by the address register indirect addressing mode. The other addressing modes can be implemented by following the demonstration software as an example.

The move-out macros (REGMEMn, n=R, W, L, S, D, X, P) of op-class 011 of the general instruction class are structured in the same manner as MEMREGn (n=R, W, L, S, D, X, P). Coprocessor distinction between the move-in and the move-out operations result from the different op-class specifications within the command word.

The one difference between the two op-classes is in the packed BCD macros. This difference is due to the nature of the MC68881 FMOVE out packed BCD from the coprocessor instruction which requires the user to submit additional information to the coprocessor: the k-factor. The k-factor is passed to the operand register from either a data register or as immediate data in the command word. To be able to handle all data registers, the packed BCD macro would be extensive using elaborate conditional statements. Therefore, the programmer is only allowed to use data register D0, which fixes that part of
Figure 19. F-Line Emulation Sequence
the extension field representing the data register as a constant. Thus, only one pair of conditional instructions is needed.

The REGREG macro supports op-class 000 which performs a coprocessor register-to-register operation. No services are needed of the main processor other than to submit the coprocessor instruction. Therefore, after writing the command word to the command register, the response register is queried until the null primitive is granted. The several other conditional assembly statements in REGREG support the unique general arithmetic instruction, FSINCS. Since this instruction requires two destination floating-point registers for the results of the operation (FPm and FPq), another parameter must be passed to the coprocessor. The conditional assembly statement tests for the existence of a fourth parameter. To be able to support this instruction in the REGMEMn and MEMREGn macros, similar procedures should be followed. An example is implemented in the MEMREGB macro (line 327-434).

Op-class 010 with Rx equal to 111 represents the operation registering the access of the coprocessor constants (FMOVEOM). A command word specifying the constant to be retrieved is written to the command register. Since no further services of the main processor are needed, the remaining function is to test the response register for the release signal (CA equals zero).

Both macros, MOVINCIS (op-class 100) and MOVOUTCSI (op-class 101) move the coprocessor system registers. Each perform the same instruction sequence as MEMREGn and REGMEMn, respectively, with the only difference being the value of the command word. The move multiple coprocessor system register instructions are not supported by the macros.

The final op-classes of the general instructions to be discussed are those corresponding to the movement of multiple floating-point data registers (FMOVEPRM and FMOVEPRMR). Due to the nature of the M88000 Family memory organization, the macros are constructed differently. The user specifies which registers are moved by selecting the corresponding parameter in the macro call, and the coprocessor detects which registers are affected by the bit mask specified in the command word. The binary bit mask is formed by the parameter list. The list is treated as a string of concatenated bits which is required by the MC68881 to represent the register select mask.

Since the floating-point data registers are 96 bits wide, three consecutive accesses of 32 bits each must be made to acquire the data. FMOVEPRM (lines 761-834) organizes this data so that the high-order bit is situated in low-order memory. The coprocessor delivers FP0 first (if selected) and FP7 last, except when the indirect addressing with prederecrement mode is being used. In which case, the coprocessor sends FP7 first and FP0 last so that FP0 is always placed in low memory. Therefore, the conditional test for the prederecrement mode is required to reverse the order of the bit mask sent to the coprocessor. The FMOVEPRMR (lines 835-881) macro moves data into the registers by moving FP0 first as the coprocessor always expects FP0 first. The FMOVEPRMR macro does not allow the prederecrement addressing mode.

The second type class to be discussed is the branch instruction class which is supported by the FBCC macro. The main processor writes the conditional predicate (CPRED) to the condition register and reads the response register until signaled to be released. Then the T/F bit of the response primitive is examined, and if status indicates, the branch is taken.

The FDBCC and FSBCC macros support the conditional type instructions. Both macros follow the same protocol as FBCC (i.e., write CPRED to the condition register, read the response register, and after being released, perform requested function if condition satisfied). All of the branch and conditional macros must modify a data register which serves as a temporary variable. When the coprocessor grants the null release primitive, the T/F bit is also passed in the response. As the MC68881 does not expect another response register access, the response is saved in D0 so the CA bit can be tested. When CA equals zero, the T/F bit is already available in D0.

The no-operation macro exists for the no-operation or synchronizing FNOR instruction. It is a branch never instruction. The main processor writes the second word of the coprocessor instruction to the condition register and queries the response register until released by the coprocessor.

One MC68881 conditional instruction not implemented is the conditional trap (FTRAPcc) instruction because the MC68020 has these coprocessor traps. The trap instruction is not available on the MC68881, the MC68000, or the MC68010. To cause a trap from the users space in a MC68000/MC68008/MC68010 system, the overflow bit in the control register can be set, and the TRAPV instruction executed. However, the trap handler can not distinguish between the simulated coprocessor condition and the overflow condition that would normally use this trap vector.

The final two coprocessor instruction types to be discussed are the save and restore performed by macros FSAVEST and FRESTRST, respectively. Only one addressing mode is supported in the macros. Several other conditional assembly instructions, similar to those in the FMOVEM and FMOVEMR macro, can be implemented to utilize more addressing modes. To initiate the save sequence, the main processor reads the format word from the save register. This 16-bit register is reread until the high-order byte no longer contains a 01 (coprocessor busy). At this point, the length of the coprocessor data to be transferred resides in the low-order byte of the format word. The main processor isolates this length and begins to transfer the data from the operand register (making long word accesses to memory via the indirect addressing with prederecrement addressing mode. After saving the invisible portion of the coprocessor state, the main processor stores the format word at the top of the stack, in low-order memory. This assures proper restoration of the MC68881 state when the FRESTRST macro is executed. In FRESTRST, the main processor writes the previously saved format word from memory to the restore register, reads the restore register, and begins writing the stored data to the operand register until the proper number of bytes has been transferred. Indirect addressing with postincrement addressing mode is used.
In summary, the performance of the MC68881, driven as a peripheral in a MC68000/MC68008/MC68010 system, is enhanced by using the macro approach. This is primarily due to the fact that most of the instruction decode is done at assembly. This line code is upwardly source code compatible to a MC68020 system via re-compilation or reassembly. For instance, the following code provides an example of how to alter a macro (for reassembly) in order to acquire floating-point source code compatibility when porting the user software to an MC68020 system (equate table must be deleted):

MEMREGB MACRO
"1.B" "2.3"
ENDM

The macro call will remain the same. For example, this macro call:

MEMREGB FADD,DO,FP0

expands to create the following MC68881 floating-point add source code when used in conjunction with the previous macro definition:

FADD DO,FP0

A few consequences of this technique exist: 1) the object code is not MC68881 replaceable because if the code were moved up to a MC68020/MC68010 system, the MC68881 would still be a peripheral processor in user data space (to benefit from the MC68020 coprocessor interface, the macros would have to be changed and the user program reassembled); 2) a macro library and/or other routines are required to contain the macro software; 3) the full environment is not presented to the user as not all addressing modes nor the FTRAPc instruction are supported, all checking done by the MC68020 is not implemented (e.g., illegal format errors), and exceptions are not enabled; 4) the MC68881 is not an independent operating hardware device because peripheral I/O access is used, and 5) the demonstration software does not support the M68000 immediate addressing mode.

F-LINE EMULATION SOFTWARE

As an alternative to using macros or in-line code, an F-line trap emulation could be implemented in an MC68000/MC68008/MC68010 system when the user requires the user program object code containing MC68881 instructions to be upward compatible with the MC68020 without recompiling, reassembling, or relinking. By using this approach, the coprocessor will be driven as a peripheral from supervisor space by supervisor software. Complete source and object code compatibility with the MC68881 instruction set can be maintained.

Because some M68000 systems separate user and supervisor space, different types of emulations must be developed. This application note includes two examples of an F-line emulation of the general instruction operation performed on data moved into the coprocessor: the protected and unprotected versions. The software for other types such as move outs can be inferred from the examples given. The protected version (APPENDIX B PROTECTED F-LINE EMULATION SOFTWARE) is used on systems which segregate user and system address spaces. The unprotected version (APPENDIX C UNPROTECTED F-LINE EMULATION SOFTWARE) can be used on any M68000 system which allows direct access to user spaces from the supervisor state.

Functional Description

If the coprocessor instruction were decoded by the trap routine to determine the addressing modes used to access the instruction operands, then a significant overhead would be incurred with a commensurate loss of performance. Hence, the demonstration software assumes a single addressing mode will always be used. This is register indirect, (A0). If the programmer desires to use other addressing modes, this can be accomplished by simply performing:

LEA EA,A0

before executing the floating-point instruction. Note, the LEA instruction will not work when using PC relative addressing mode in a system that splits program and data spaces (although this is rarely encountered). Also, as implemented in the macro approach, no error or exception checking is performed by the F-line emulator approach.

In the examples, all memory to floating-point register operations are supported including FMOVECR and all FPn operations.

Theory of Operation

The following paragraphs provide information for users creating their own F-line trap emulations. The read-write protocol of the move-in macros (Figure 11) is implemented in both the protected and unprotected forms of the emulation. The F-line trap emulation differs from the protocol of the macro approach. In the emulation, after the main processor has transferred the data to the coprocessor, the final read of the response register is no longer needed. Sufficient time will expire between any two consecutive floating-point instructions due to the overhead of the F-line trap which ensures that no spurious protocol violations will occur.

A flowchart of the unprotected emulation version is seen in Figure 19. In the protected version, the same sequence of events occurs with the exception that the floating-point instruction and source operands are accessed in user memory from supervisor space utilizing the MC68010 MOVES and MOVEC instructions. Only the unprotected version (APPENDIX C UNPROTECTED F-LINE EMULATION SOFTWARE) is referenced.

When a coprocessor instruction is encountered, the F-line trap is taken. The location of the coprocessor instruction (program counter) and other information (depending on the main processor executing the instruction) is placed on the stack. The data at the program counter location (the operation word of the coprocessor instruction) is examined to determine whether the instruction is a general type (line 25). If so, the second word (the command word) is written to the command register (line 30). Then, the main processor queries the response register until the coprocessor no longer processes the previous instruction (null come-again).

AN947/D MOTOROLA
Next, the op-class specified in the command word is examined to determine the main processor’s next action. First, the main processor tests for the move multiple coprocessor registers (data or system) into or out of the MC68881 (op-classes with high-order bits set, e.g. 1xx) in lines 33-34. If found, the main processor would jump to a routine to handle this special function. This function is not implemented in this application but is a straightforward routine.

Subsequently, testing for a floating-point register-to-register operation occurs in line 35. In this case, no further services are needed of the main processor, and a jump to the RTE instruction is taken.

Finally, a distinction between the move-in and move-out operations is made (line 37-38). An additional routine can be developed to support the move-out sequence. When a move-in operation has been identified, the main processor then extracts the precision of the external operand from the command word. If the instruction is found to be a FMOVE (precision 111), the main processor immediately branches to the RTE instruction. Otherwise, the main processor branches to the small routines for handling the respective data transfers. Since long words, packed BCD, single, double, and extended precision data transfers all require at least one 32-bit data transfer, one routine handles all five data types (lines 54-56). Two other routines (lines 59-60 and lines 63-64) support the byte and word transfers. After the data has been delivered to the coprocessor, the main processor returns from exception via the RTE instruction (the instruction which completes the F-line trap and re-enters the user program).

In both versions of the F-line trap emulation, the work registers were stored at the beginning of the routine and then restored prior to the exit.

In summary, the F-line emulation trap is initiated when the main processor identifies a coprocessor instruction by a hexadecimal F in the most-significant nibble of the first word of the instruction (and takes the F-line trap).

The object code containing the MC68881 instruction is upward compatible to an MC680020 system without re-compiling, reassembling, or relinking.

The major consequence in implementing an F-line emulation instead of in-line code is the time factor incurred by both the overhead of the F-line trap and the instruction decode in the trap routine. Listed in Table 2 are the clock cycles required to perform the various operations using the two F-line emulations and the macro approaches. Even with the overhead associated with the F-line emulation this approach offers a speed advantage over floating-point software packages and at the same time maintains MC68020/MC68881 upward compatibility. Timings are based on a no-wait state system (four clock cycle bus cycle). The MC68881 overhead is not taken into account because the main processor is released to perform the next instruction of the user program while the coprocessor executes its instructions. FMOVE is an example instruction that could be replaced by any of the general instructions (excluding the move multiples).

**CONCLUSION**

The MC68881 floating-point coprocessor can be utilized as a peripheral in a MC68000/MC68003/MC68010 system by either directly driving the device as a peripheral or by simulating the complete coprocessor instruction set. Either method, depending on the application, is sufficient to utilize the high performance of the MC68881 and offers superior speed and versatility over floating-point software packages. The macros or in-line code of this application provide a faster way to access the device for the users interested in achieving the highest performance of the MC68881. Alternately, for applications that can trade-off performance to achieve object code upward compatibility with MC68020 systems, an example of an F-line emulation trap has also been included.

<table>
<thead>
<tr>
<th>Table 2. Operation Execution Time (Clock Cycle) for MC68010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation (MC68881)</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>F(op),B EA,FPhn</td>
</tr>
<tr>
<td>F(op),W EA,FPhn</td>
</tr>
<tr>
<td>F(op),L EA,FPhn</td>
</tr>
<tr>
<td>F(op),S EA,FPhn</td>
</tr>
<tr>
<td>F(op),D EA,FPhn</td>
</tr>
<tr>
<td>F(op),X EA,FPhn</td>
</tr>
<tr>
<td>F(op),P EA,FPhn</td>
</tr>
<tr>
<td>F(op),X FPhn,FPhn</td>
</tr>
<tr>
<td>FMOVECR #ccc,FPhn</td>
</tr>
</tbody>
</table>

(op) = MOVE, ADD, SUB
APPENDIX A
MACROS

***************************************************************
** SOURCE CODE TO DRIVE THE MC68881 AS A PERIPHERAL **
** TO NOT SHOW MACRO EXPANSION IN THE LIST FILE DELETE 'OPT MEX' BEFORE **
** ASSEMBLY. **
** TO SHOW THE CONDITIONAL ASSEMBLY INSTRUCTIONS IN THE MACRO EXPANSION **
** DELETE THE 'OPT NOCL' (LOCATED AFTER THIS BOX) BEFORE ASSEMBLY. **
***************************************************************

OPT NOCL
OPT MEX

****

THIS IS THE EQUATE FILE TO SUPPORT THE MACROS USED **
TO DRIVE THE MC68881 AS A PERIPHERAL **
WITH THE M68000 FAMILY 

COPY

 THESE ARE THE INSTRUCTION BIT PATTERN EQUATES

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

FMOV EQU $00 MOVE
FINT EQU $01 INTERGER PART
FSINH EQU $02 SINH
FSQRT EQU $04 SQUARE ROOT
FLOGNP1 EQU $06 LOGN (1/X)
FETOX EQU $08 \[(E**X)-1\]
FTANH EQU $09 TANH
FATAN EQU $0A ARCTAN
FASIN EQU $0C ARCSIN
FATANH EQU $0D ARCTANH
FSIN EQU $0E SINE
FTAN EQU $0F TANGENT
FETOX EQU $10 E**X
FTMOTOX EQU $11 2**X
FTENTOX EQU $12 10**X
FLOG EQU $14 LOGN
FLOG10 EQU $15 LOG10
FLOG2 EQU $16 LOG2
FABS EQU $18 ABSOLUTE VALUE
FCOSH EQU $19 COSH
FSIN EQU $1A TRIGERGATE
FACOS EQU $1B ARCCOS
FACOS EQU $1D COSINE
FGETEXP EQU $1E GET EXPONENT
FGETMAN EQU $1F GET MANTISSA
FDIV EQU $20 DIVIDE
FMCD EQU $21 MODULOMAINDER
FADD EQU $22 ADD
FMUL EQU $23 MULTIPLY
FSGLDIV EQU $24 SINGLE DIVIDE
FSREM EQU $25 IEEE REMAINDER
FSOCALE EQU $26 SCALE EXPONENT
FSGLMUL EQU $27 SINGLE MULTIPLY
FSUB EQU $28 SUBTRACT
FCMP EQU $38 COMPARE
FTST EQU $3A TEST
FSINCOS EQU $30 SIMULTANEOUS FP SINE AND COSINE
**These are the mnemonics used as the condition codes for the branch instructions.**

67 EQ EQU $01 EQUAL
68 NEQ EQU $0E NOT EQUAL
69 GT EQU $12 GREATER THAN
70 NLT EQU $1D NOT GREATER THAN
71 GE EQU $13 GREATER THAN OR EQUAL
72 NGE EQU $1C NOT (GREATER THAN OR EQUAL)
73 LT EQU $14 LESS THAN
74 NGE EQU $1B NOT LESS THAN
75 LE EQU $15 LESS THAN OR EQUAL
76 NLE EQU $1A NOT (LESS THAN OR EQUAL)
77 LTI EQU $16 GREATER OR LESS THAN
78 NGL EQU $19 NOT (GREATER OR LESS THAN)
79 GLE EQU $17 GREATER OR LESS THAN
80 NLE EQU $18 NOT (GREATER OR LESS THAN)
81 OGT EQU $02 ORDERED GREATER THAN
82 ULE EQU $0D UNORDERED OR LESS THAN
83 OGE EQU $03 ORDERED GREATER THAN
84 OLT EQU $0C UNORDERED OR LESS THAN
85 UGE EQU $04 ORDERED LESS THAN
86 OGE EQU $0B UNORDERED OR GREATER THAN
87 ULE EQU $05 ORDERED LESS THAN OR EQUAL
88 UGT EQU $0A UNORDERED OR GREATER
89 GOL EQU $06 ORDERED GREATER OR LESS THAN
90 UEQ EQU $09 UNORDERED OR EQUAL
91 OR EQU $07 ORDERED
92 UN EQU $08 UNORDERED
93 F EQU $00 FALSE (NEVER)
94 T EQU $0F TRUE (ALWAYS)
95 SF EQU $10 SIGNALING FALSE (NEVER)
96 SF EQU $1F SIGNALING TRUE (ALWAYS)
97 SEQ EQU $11 SIGNALING EQUAL
98 SNEQ EQU $1E SIGNALING NOT EQUAL

**These equates represent the offsets for the base address of the MC6881 interface registers.**

104 COMMAND EQU $0A COMMAND REGISTER
105 RESPONSE EQU $00 RESPONSE REGISTER
106 OPER EQU $10 OPERAND REGISTER
107 COND EQU $0E CONDITION REGISTER
108 SAVE EQU $04 SAVE REGISTER
109 RESTORE EQU $06 RESTORE REGISTER
110 REGSEL EQU $14 REGISTER SELECT
111 CONTROL EQU $9000 MC6881 CONTROL REGISTER
112 STATUS EQU $8800 MC6881 STATUS REGISTER
113 IADDRESS EQU $8400 MC6881 INSTRUCTION ADDRESS REGISTER
114 TBIT EQU $0 TRUE/FALSE BIT OF THE RESPONSE REGISTER

**These equates represent the floating point registers.**

119 FP0 EQU $00 FLOATING POINT REGISTER #0
120 FP1 EQU $01 " " " 
121 FP2 EQU $02 " " " 
122 FP3 EQU $03 " " " 
123 FP4 EQU $04 FLOATING POINT REGISTER #4
124 FP5 EQU $05 " " " 
125 FP6 EQU $06 " " " 
126 FP7 EQU $07 " " " 

MOTOROLA
22

AN947/D
MC68881 SINGLE PRECISION FP-REG. VALUE TO MEMORY OPERATION

WHERE: INSTRUCTION = FP INSTRUCTION NUEMONIC (I.E. FMOVE)
FPM = SOURCE FP REGISTER
<EA> = DESTINATION ADDRESSING MODE

NO REGISTERS MODIFIED OR DESTROYED!

VALID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX)

MISSION, FPM, <EA>

WHERE:
INSTRUCTION = FP INSTRUCTION NOEMONIC (I.E. FMOVE)
FPM = SOURCE FP REGISTER
<EA> = DESTINATION ADDRESSING MODE

NO REGISTERS MODIFIED OR DESTROYED!

VALID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX)

MISSION, FPM, <EA>

WHERE:
INSTRUCTION = FP INSTRUCTION NOEMONIC (I.E. FMOVE)
FPM = SOURCE FP REGISTER
<EA> = DESTINATION ADDRESSING MODE

NO REGISTERS MODIFIED OR DESTROYED!

VALID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX)

MISSION, FPM, <EA>

WHERE:
INSTRUCTION = FP INSTRUCTION NOEMONIC (I.E. FMOVE)
FPM = SOURCE FP REGISTER
<EA> = DESTINATION ADDRESSING MODE

NO REGISTERS MODIFIED OR DESTROYED!
193 * MOVE.L MC68881+OPER,\3,\4 DATA
194 \$NULREL TST.B MC68881+RESPONSE LONG WORD TRANSFER
195 \$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
196 \$NULREL TST.B MC68881+RESPONSE BRANCH UNTIL NULL RELEASE
197 ENDC
198 ENDM
199 **********************************************************************
200 * MC68881 WORD LENGTH FP-REG. VALUE TO MEMORY OPERATION *
201 * RECMEM INSTRUCTION,FPM,<EA> *
202 * WHERE: INSTRUCTION= FP INSTRUCTION NUMERIC (I.E. FMOVE) *
203 * FPM= SOURCE FP REGISTER *
204 * <EA>= DESTINATION ADDRESSING MODE *
205 * NO REGISTERS MODIFIED OR DESTROYED! *
206 * VALID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX) *
207 * XXX.W, XXX.L, (D,PC), D(PC,IX) *
208 * **********************************************************************
209 * RECMEM MACRO *
210 IFC \"4\",\" IS <EA>=INDIRECT WITH INDEXING *
211 MOVE.W \#7000+(\2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION *
212 \$NULCA CMPF \#8990C,MC68881+RESPONSE READ RESPONSE REGISTER *
213 BEQ.S \$NULCA REPEAT UNTIL EVALUATE EA AND TRANSFER *
214 MOVE.W MC68881+OPER,\3 WORD DATA TRANSFER *
215 \$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? *
216 BMI.S \$NULREL BRANCH UNTIL NULL RELEASE *
217 ENDC *
218 IFC \"4\",\" IS <EA> NOT = INDIRECT WITH INDEXING *
219 MOVE.W \#7000+(\2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION *
220 \$NULCA CMPF \#8990C,MC68881+RESPONSE READ RESPONSE REGISTER *
221 BEQ.S \$NULCA REREAD UNTIL EVALUATE EA AND TRANSFER *
222 MOVE.W MC68881+OPER,\3,\4 WORD DATA TRANSFER *
223 \$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? *
224 BMI.S \$NULREL BRANCH UNTIL NULL RELEASE *
225 ENDC *
226 IFC \"4\",\" IS <EA>=INDIRECT WITH INDEXING *
227 MOVE.W \#7000+(\2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION *
228 \$NULCA CMPF \#8990C,MC68881+RESPONSE READ RESPONSE REGISTER *
229 BEQ.S \$NULCA REREAD UNTIL EVALUATE EA AND TRANSFER *
230 MOVE.W MC68881+OPER,\3,\4 WORD DATA TRANSFER *
231 \$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? *
232 BMI.S \$NULREL BRANCH UNTIL NULL RELEASE *
233 ENDC *
234 *
235 *
236 *
237 ENDM *
238 *
239 *
240 MC68881 BYTE LENGTH FP-REG. VALUE TO MEMORY OPERATION *
241 *
242 RECMEM INSTRUCTION,FPM,<EA> *
243 *
244 WHERE: INSTRUCTION= FP INSTRUCTION NUMERIC (I.E. FMOVE) *
245 FPM= SOURCE FP REGISTER *
246 <EA>= DESTINATION ADDRESSING MODE *
247 *
248 *
249 *
250 Valid ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX) *
251 * XXX.W, XXX.L, (D,PC), D(PC,IX) *
252 *
253 RECMEM MACRO *
254 IFC \"4\",\" IS <EA>=INDIRECT WITH INDEXING *
255 MOVE.W \#7000+(\2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION *
256 \$NULCA CMPF \#8990C,MC68881+RESPONSE READ RESPONSE REGISTER *
257 BEQ.S \$NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
259 * MOVE.B MC68881+OPER, \3 ELT DATA
260 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
261 B.M.S \NULREL BRANCH UNTIL NULL RELEASE
262 ENDC
263 IFNC \4, \*
264 \NULCA CMP \#57800+\(2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION
265 \NULCA CMP \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
266 BEQ. S \NULCA READ DATA
267 \NULREL TST.B MC68881+OPER, \3 \4 BYTE DATA TRANSFER
268 MOVE.B MC68881+OPER, \3 \4 BYTE DATA TRANSFER
269 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
270 B.M.S \NULREL BRANCH UNTIL NULL RELEASE
271 ENDC
272 END
273
274******************************************************************************
275 * MC68881 DOUBLE PRECISION FP-REG. VALUE TO MEMORY OPERATION
276 * REGEMX MACRO
277 MOVE.W \#57800+\(2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION
278 \NULCA CMP \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
279 BEQ. S \NULCA READ DATA
280 \NULREL TST.B MC68881+OPER, \3 HIGH ORDER LONG WORD
281 MOVE.L MC68881+OPER, \3 LOW ORDER LONG WORD
282 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
283 B.M.S \NULREL BRANCH UNTIL NULL RELEASE
284 ENDM
285
286******************************************************************************
287 * MC68881 EXTENDED PRECISION FP-REG. VALUE TO MEMORY OPERATION
288 * REGEMX MACRO
289 MOVE.W \#56800+\(2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION
290 \NULCA CMP \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
291 BEQ. S \NULCA READ DATA
292 \NULREL TST.B MC68881+OPER, \3 HIGH ORDER LONG WORD
293 MOVE.L MC68881+OPER, \3 MID-ORDER
325  MOVE.L MC68881+OPER,8,\3  LOW ORDER WORD
326 \$NULCEL TST.B MC68881+RESPONSE  IS RESPONSE NULL RELEASE?
327  BMI.S \$NULCEL  BRANCH UNTIL NULL RELEASE
328  ENDM
329
330  ******************************************************
331  MC68881 PACKED BCD FP-REG. VALUE TO MEMORY OPERATION
332  ******************************************************
333  REGEMP  INSTRUCTION,FPM,<EA>[K-FACTOR]
334  WHERE:  INSTRUCTION= FP INSTRUCTION NUemonic (I.E. FADD)
335  FPM= SOURCE FP REGISTER
336  <EA>= DESTINATION ADDRESSING MODE
337  [K-FACTOR]= OPTIONAL IMMEDIATE K-FACTOR
339  VALID ADDRESSING MODES:  (AN)
340
341  ******************************************************
342  REGEMP MACRO
343  IFC \4,\4  IS K-FACTOR IN REGISTER?
344  MOV.EW #57C00+\2<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION
345 \$NULCEL CMP.I #6900,MC68881+RESPONSE  READ RESPONSE REGISTER
346  BEQ.S \$NULCEL  REREAD UNTIL TRANSFER MAIN PROCESSOR REG
347 \$AGAIN CMP.I #6900,MC68881+RESPONSE  READ RESPONSE REGISTER
348  BEQ.S \$AGAIN  REREAD UNTIL EVALUATE EFFECTIVE ADDRESS
349  AND TRANSFER DATA
350  MOV.E L MC68881+OPER,\3  LOW ORDER LONG WORD
351  MOV.E L MC68881+OPER,4,\3  MID-ORDER LONG WORD
352  MOV.E L MC68881+OPER,8,\3  HIGH ORDER LONG WORD
353 \$NULCEL TST.B MC68881+RESPONSE  IS RESPONSE NULL RELEASE?
354  BMI.S \$NULCEL  BRANCH UNTIL NULL RELEASE
355  ENDC
356  IFC \4,\4  IS K-FACTOR IN INSTRUCTION?
357  MOV.E W #66C00+\2<<7)+\4,MC68881+COMMAND MEM. TO REG. OPERATION
358 \$NULCEL CMP.I #6900,MC68881+RESPONSE  READ RESPONSE REGISTER
359  BEQ.S \$NULCEL  REREAD UNTIL EVALUATE EFFECTIVE ADDRESS
360  AND TRANSFER DATA
361  MOV.E L MC68881+OPER,\3  LOW ORDER WORD
362  MOV.E L MC68881+OPER,4,\3  MID-ORDER WORD
363  MOV.E L MC68881+OPER,8,\3  HIGH ORDER WORD
364 \$NULCEL TST.B MC68881+RESPONSE  IS RESPONSE NULL RELEASE?
365  BMI.S \$NULCEL  BRANCH UNTIL NULL RELEASE
366  ENDC
367  ENDM
368
369  ******************************************************
370  MC68881 BYTE IN MEMORY OR IN Dn TO FP-REG. OPERATION
371  ******************************************************
372  MEMREGB INSTRUCTION,<EA>,FPM
373  WHERE:  INSTRUCTION= FP INSTRUCTION NUemonic (I.E. FADD)
374  <EA>= SOURCE ADDRESSING MODE
375  FPM= DESTINATION REGISTER
376  NO REGISTERS MODIFIED OR DESTROYED!
377  VALID ADDRESSING MODES:  Dn, (AN)+, - (AN), D(AN), D(AN,IX)
378  XXX.W, XXX.L, (D,FC), D(FC,IX)
379  THE COMMENTED OUT CODE SHOWS HOW A USER MAY IMPLEMENT FSINCSOS
380  IN A MEM. TO REG. TRANSFER USING THE FOLLOWING INSTRUCTION FORMAT:
381  MEMREGB INSTRUCTION,<EA>,FPM,FHQ  (FPH= 2ND DESTINATION REG.)
391 * 392 ******************************************************************************** 393 * 394 MEMREG MACRO 395 * 396 IFC "!1", 'FSINCOS' IS INSTRUCTION FSINCOS 397 IFC "!4", 'FSINCOS' IS INSTRUCTION NOT FSINCOS 398 MOVE.W #$5800+(\3<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION 399 \NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER 400 BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER 401 MOVE.B \2,MC68881+OPER BYTE DATA TRANSFER 402 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? 403 B.MI.S \NULREL BRANCH UNTIL NULL RELEASE 404 ENDC 405 IFNC "!3", 'FSINCOS' IS INSTRUCTION PART OF THE ADDR.MODE 406 MOVE.W #$5800+(\4<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION 407 \NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER 408 BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER 409 MOVE.B \2,\3,MC68881+OPER BYTE DATA TRANSFER 410 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? 411 B.MI.S \NULREL BRANCH UNTIL NULL RELEASE 412 ENDC 413 ENDC 414 IFNC "!1", 'FSINCOS' IS INSTRUCTION NOT FSINCOS 415 IFC "!4", 'FSINCOS' IS <EA> NOT INDIRECT WITH INDEXING 416 MOVE.W #$5800+(\3<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION 417 \NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER 418 BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER 419 \NULCA CMPI #$8900,MC6881+RESPONSE READ RESPONSE REGISTER 420 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? 421 B.MI.S \NULREL BRANCH UNTIL NULL RELEASE 422 ENDC 423 ENDC 424 IFNC "!3", 'FSINCOS' IS INSTRUCTION PART OF THE ADDR.MODE 425 MOVE.W #$5800+(\4<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION 426 \NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER 427 BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER 428 \NULCA CMPI #$8900,MC6881+RESPONSE READ RESPONSE REGISTER 429 \NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE? 430 B.MI.S \NULREL BRANCH UNTIL NULL RELEASE 431 ENDC 432 ENDC 433 ENDC 434 ENDC 435 **** 436 **** 437 * MC68881: WORD IN MEMORY OR IN Dn TO FP-REG. OPERATION 438 * 439 * MEMREG INSTRUCTION, <EA>, FP N 440 * 441 * WHERE: INSTRUCTION= FP INSTRUCTION NUENONIC (I.E. FADD) 442 * <EA>= SOURCE ADDRESSING MODE 443 * FPN= DESTINATION REGISTER 444 * 445 * NO REGISTERS MODIFIED OR DESTROYED! 446 * 447 * VALID ADDRESSING MODES: Dn, (AN)+, -(AN), D(AN), D(AN,IX) 448 * XXX.W, XXX.L, (D,PC), D(PC,IX) 449 * 450 ******************************************************************************** 451 * 452 MEMREG MACRO 453 * 454 IFC "!3", 'FSINCOS' IS <EA> NOT INDIRECT WITH INDEXING 455 MOVE.W #$5800+(\3<<7)+\1,MC68881+COMMAND MEM. TO REG. OPERATION 456 \NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER 457 BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER 458 \NULCA CMPI #$8900,MC6881+RESPONSE READ RESPONSE REGISTER 459 BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER 460 **
MC68000 

**Instruction: FADD**

- **Instruction mnemonic:** FADD
- **Addressing modes:** DN, (AN)t, -(AN), D(AN), D(AN,IX)

**Notes:**
- No registers modified or destroyed!
- Valid addressing modes: DN, (AN)+, -(AN), D(AN), D(AN,IX), XXX.W, XXX.L, (D,PC), D(FC,IX)
MEMREGS MACRO

IFC ', ' IS INDICATIVE WITH INDICATING 

MOVE.W \#4400+\(\(\x9c<<7\))+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA

* MOVE.L \2,MC68881+OPER SINGLE PRECISION DATA TO FP REG.
\NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \NULREL BRANCH UNTIL NULL RELEASE
ENDC

IFNC ', ' IS NOT INDICATIVE WITH INDICATING 

MOVE.W \#4400+\(\(\x9c<<7\))+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA.

* MOVE.L \2,MC68881+OPER SINGLE PRECISION DATA TO FP REG.
\NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \NULREL BRANCH UNTIL NULL RELEASE
ENDC

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

MEMREGD MACRO

MOVE.W \#4400+(\(\x9c<<7\))+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA

* MOVE.L \2,MC68881+OPER HIGH ORDER LONG WORD
MOVE.L \2,MC68881+OPER LOW ORDER LONG WORD
\NULCA CMPI \#8900,MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \NULCA BRANCH UNTIL NULL RELEASE
ENDM

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

MEMREGX MACRO

MOVE.W \#64900+(\(\x9c<<7\))+\1,MC68881+COMMAND MEM. TO REG. OPERATION

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

AN947/D

MOTOROLA 29
MC68881 PACKED BCD VALUE MEMORY TO FP-REG. OPERATION

MEMREG MACRO

MOVE.L $<400++/\3<<7)+/1,MC68881+COMMAND MEM. TO REG. OPERATION
\NUlCA CMP #8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \\NUlCA REREAD UNTIL EVALUATE EA AND TRANSFER

MOVE.L \2,MC68881+OPER HIGH ORDER LONG WORD
MOVE.L 4\2,MC68881+OPER MID-ORDER LONG WORD
MOVE.L 8\2,MC68881+OPER LOW ORDER LONG WORD
\NUlCA TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \\NUlCA BRANCH UNTIL NULL RELEASE

ENDM

MEMREG PF INSTRUCTION (I.E. FADD)

MOVE.W #<400++/\3<<7)+/1,MC68881+COMMAND MEM. TO REG. OPERATION
\NUlCA CMP #8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \\NUlCA REREAD UNTIL EVALUATE EA AND TRANSFER

MOVE.L \2,MC68881+OPER HIGH ORDER LONG WORD
MOVE.L 4\2,MC68881+OPER MID-ORDER LONG WORD
MOVE.L 8\2,MC68881+OPER LOW ORDER LONG WORD
\NUlCA TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \\NUlCA BRANCH UNTIL NULL RELEASE

ENDM

IFNC 'FSINCOS' ROUTINE FOR OTHER ARITHMETIC INSTR.

MOVE.W #<2<<10)+/\3<<7)+/1,MC68881+COMMAND REG. TO REG. FSINCOS
\NUlCA TST.B MC68881+RESPONSE READ RESPONSE REGISTER
BMI.S \\NUlCA REREAD UNTIL NULL RELEASE (CA=0)

ENDC

IFNC 'FSINCOS' ROUTINE FOR ALL OTHER ARITHMETIC INSTRS.

MOVE.W #<2<<10)+/\3<<7)+/1,MC68881+COMMAND REG. TO REG. OPERATION
\NUlCA TST.B MC68881+RESPONSE READ RESPONSE REGISTER
BMI.S \\NUlCA REREAD UNTIL NULL RELEASE (CA=0)

ENDC

ENDM
MC68881 CONSTANT IN ROM TO FP-REG. OPERATION

WHERE:

$00  PI
$0B  LOG10(2)
$0C  E
$0D  LOG2(E)
$0E  LOG10(E)
$0F  0.0
$30  LOG2(2)
$31  LOG10(10)
$32  10^0
$33  10^1
$34  10^2
$35  10^4
$36  10^8
$37  10^16
$38  10^32
$39  10^64
$3A  10^128
$3B  10^256
$3C  10^512
$3D  10^1024
$3E  10^2048
$3F  10^4096

FPN= FP DESTINATION REGISTER

NO REGISTERS MODIFIED OR DESTROYED!

---

MC68861 CONDITIONAL BRANCH

WHERE: <SIZE> ALLOWABLE BRANCH SIZES

CONDITION= CC, THE Floating POINT CONDITION (I.E. GT)

ADDRESS= BRANCH ADDRESS

REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7

DX
A

BEGIN COPROCESSOR COMMUNICATION
IS CA-BIT SET
REREAD UNTIL NULL RELEASE (CA=0)
IS CONDITION TRUE
BRANCH IF CONDITION TRUE!
715 *******************************************************************************
716 * *
717 * K6881 TEST FP CONDITION, DECREMENT, AND BRANCH *
718 * *
719 * FDBCC CONDITION, DN, ADDRESS *
720 * *
721 * WHERE: CONDITION= CC, FLOATING POINT CONDITION *
722 * DN= MAIN PROCESSOR DATA REGISTER TO BE DECREMENTED *
723 * ADDRESS= BRANCH ADDRESS *
724 * *
725 * REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7 *
726 * D X *
727 * A *
728 *
729 * FDBCC MACRO *
730 * MOVE.W #1,MC6881+COND BEGIN COPROCESSOR COMMUNICATION *
731 \NOPASS MOVE.W MC6881+RESPONSE,DO IS CA-BIT SET *
732 \NOPASS BML.S \NOPASS REREAD UNTIL NULL RELEASE (CA=0) *
733 BTST #TFBIT,DO IS CONDITION TRUE *
734 DBNE \2,\3 SUBTRACT 1 FROM COUNTER UNTIL COUNTER *
735 * EQUALS -1 *
736 * ENDM *
737 *
738 * MC6881 CONDITIONAL SET *
739 * *
740 * FSCC CONDITION, ADDRESS *
741 * *
742 * WHERE: CONDITION= CC, FLOATING POINT CONDITION *
743 * ADDRESS= BRANCH ADDRESS *
744 * *
745 * REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7 *
746 * D X *
747 * A *
748 *
749 * FSCC MACRO *
750 * MOVE.W #1,MC6881+COND BEGIN COPROCESSOR COMMUNICATION *
751 \NOPASS MOVE.W MC6881+RESPONSE,DO IS CA-BIT SET *
752 \NOPASS BML.S \NOPASS REREAD UNTIL NULL RELEASE (CA=0) *
753 BTST #TFBIT,DO IS CONDITION TRUE *
754 SNE \2 SET BYTE AT POINTER(\2) TO 1'S IF *
755 * CONDITION TRUE, IF CONDITION FALSE *
756 SET BYTE TO 0'S *
757 * ENDM *
758 *
759 * MC6881 FP MOVE MULTIPLE COPROCESSOR REGISTERS TO MEMORY *
760 * *
761 * FMVENSFM FPR0,FPR1,FPR2,FPR3,FPR4,FPR5,FPR6,FPR7,<EA>,PREDECREMENT *
762 * *
763 * WHERE: FPR0=(FP REG .#0) 1 IF SELECTED, 0 IF NOT *
764 * FPR1=( " #1) " " *
765 * FPR2=( " #2) " " *
766 * FPR3=( " #3) " " *
767 * FPR4=( " #4) " " *
768 * FPR5=( " #5) " " *
769 * FPR6=( " #6) " " *
770 * FPR7=( " #7) " " *
771 * <EA>= DESTINATION ADDRESSING MODE *
772 * PREDECREMENT= Y (IF PREDECREMENT MODE IS BEING USED), OR *
773 * N (IF OTHER MODE IS BEING USED). *
774 * *
775 * REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7 *
776 * A X *
IFIED MACRO

IFC "A", "Y"

IS THE ADDRESSING MODE PREDECEND

THIS CODE IS FOR PREDECEND ADDRESSING MODE

MOVE.W #E000+4, \(1\)+\(2\)+\(3\)+\(4\)+\(5\)+\(6\)+\(7\)+\(8\)-1, D3

READ REGISTER RESPONSE REGISTER

LOAD HIGH ORDER WORDS

LOAD LOW ORDER WORDS

STACK HIGH ORDER WORD IN LOW ORDER MEMORY AND LOW ORDER WORD IN HIGH ORDER MEMORY

HAVE ALL REGISTERS BEEN TRANSFERRED

READ REGISTER RESPONSE REGISTER

SET UP A MEMORY POINTER

LOAD DATA ON TO THE STACK

IS RESPONSE NULL RELEASE?

BRANCH UNTIL NULL RELEASE

IS ADDRESSING MODE NOT PREDECEND

THIS CODE IS FOR ALL VALID ADDRESSING MODES OTHER THAN PREDECEND
**MC68881 FP MOVE TO MULTIPLE COPROCESSOR REGISTERS FROM MEMORY**

**FMVEMMR <EA>, FPR0, FPR1, FPR2, FPR3, FPR4, FPR5, FPR6, FPR7, POSTINCREMENT**

WHERE: 

<table>
<thead>
<tr>
<th>FPR0= (FP REG. #0)</th>
<th>1 IF SELECTED, 0 IF NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR1= (&quot;#1&quot;)</td>
<td></td>
</tr>
<tr>
<td>FPR2= (&quot;#2&quot;)</td>
<td></td>
</tr>
<tr>
<td>FPR3= (&quot;#3&quot;)</td>
<td></td>
</tr>
<tr>
<td>FPR4= (&quot;#4&quot;)</td>
<td></td>
</tr>
<tr>
<td>FPR5= (&quot;#5&quot;)</td>
<td></td>
</tr>
<tr>
<td>FPR6= (&quot;#6&quot;)</td>
<td></td>
</tr>
<tr>
<td>FPR7= (&quot;#7&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

POSTINCREMENT= Y (IF POST-INCREMENT MODE IS BEING USED)

POSTINCREMENT= N (IF OTHER VALID MODE IS BEING USED).

REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7

A X

D X

VALID ADDRESSING MODES: AN, (AN)+, D(AN), D(AN,IX)

XXX.W, XXX.L, (D,PC), D(FC,IX)

**FMVEMMR MACRO**

MOVE.W #D000+4\(2\), (\1\2\3\4\5\6\7\8\9\), MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

**MOVINSI MACRO**

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

**MC68881 FP MOVE TO CONTROL, STATUS, OR INSTRUCTION ADDRESS REGISTER**

**MOVINSI <EA>, REGISTER**

**WHERE: <EA>= VALID SOURCE ADDRESSING MODE**

**REGISTER= CONTROL, STATUS, OR ADDRESS**

**NO REGISTERS MODIFIED OR DESTROYED!**

**VALID ADDRESSING MODES: DN, AN, (AN)+, (AN), D(AN), D(AN,IX)**

**XXX.W, XXX.L, (D,PC), D(FC,IX)**

**MOVINSI MACRO**

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

\(\1\) \(\2\) \(\3\) \(\4\) \(\5\) \(\6\) \(\7\) \(\8\) \(\9\) MC68881+COMMAND CF REGISTER BIT

**AN947/D**
BEQ.S \@NULCA
MOVE.L \@,MC68881+OPER
\@NULREL TST.B MC68881+RESPONSE
BMI.S \@NULREL
ENDC
IFNC '\3',''
MOVE.W \1,MC68881+COMMAND
\@NULCA CMP.W #$8900,MC68881+RESPONSE
BEQ.S \@NULCA
MOVE.L \1,\2,MC68881+OPER
\@NULREL TST.B MC68881+RESPONSE
BMI.S \@NULREL
ENDC

MC68881 FP MOVE FROM CONTROL/STATUS/INSTRUCTION ADDRESS REGISTER
MOVOUCSI REGISTER,\<EA>
WHERE: REGISTER= CONTROL, STATUS, OR ADDRESS
\<EA>= VALID SOURCE ADDRESSING MODE
NO REGISTERS MODIFIED OR DESTROYED!
VALID ADDRESSING MODES: DN, AN, (AN)+, -(AN), D(AN), D(AN,IX)

MOVOUCSI MACRO
IPC '\3',''
MOVE.W \1+2000,MC68881+COMMAND
\@NULCA CMP.W #$8900,MC68881+RESPONSE
BEQ.S \@NULCA
MOVE.L MC68881+OPER,\12
\@NULREL TST.B MC68881+RESPONSE
BMI.S \@NULREL
ENDC
IFNC '\3',''
MOVE.W \1+2000,MC68881+COMMAND
\@NULCA CMP.W #$8900,MC68881+RESPONSE
BEQ.S \@NULCA
MOVE.L MC68881+OPER,\12,\3
\@NULREL TST.B MC68881+RESPONSE
BMI.S \@NULREL
ENDC
ENDM

MC68881 FSsave the internal of the machine
THIS IS A PRIVILEGED INSTRUCTION WHICH IS GENERALLY ONLY USED
IN THE OPERATING SYSTEM FOR CONTEXT SWITCHING!
FSaveST \<EA>
WHERE: \<EA>= PREDECREMENT MODE -(AN)
REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7
AX
DXX
VALID ADDRESSING MODES: -(AN)
FSAVEST MACRO
\\START MOVE.W MC68881+SAVE,DO
READ THE SAVE REGISTER
970 MOVE.W DO,D1
MAKE A COPY OF THE FORMAT WORD
971 ANDI.W #$FF00,D1
ISOLATE THE FORMAT WORD
972 BEQ.S \NULL
IF NULL IDLE, NO STATE SAVE
973 CMP1.W #$50100,D1
IS THE COPROCESSOR BUSY
974 BEQ.S \START
KEEP CHECKING UNTIL CP IS FINISHED
975 * PROCESSING
976 LEA MC68881+OPER,A0
LOAD OPERAND REGISTER TO A0
977 * MOVE.B DO,D1
THE LENGTH OF THE DATA TO BE TRANSFERRED
978 LSR.B #2,D1
DEVIDE BY 2 TO ADJUST FOR WORD TRANSFER
980 EXT.W D1
ESTABLISH COUNT AS A WORD FOR DBRA
981 \LOAD MOVE.L (A0),\l
STORE THE INVISIBLE STATE
982 DBRA D1,\LOAD
REPEAT UNTIL ALL DATA IS TRANSFERRED
983 \NULL SWAP DO
PLACE FORMAT WORD IN UPPER 16 BITS OF DO
985 MOVE.L DO,\l
STORE FORMAT WORD ON THE STACK
986 ENDM
987 ****************************************************bh*h***h****k***h******k**
988 * * MC68881 FRESTORE OF THE INTERNAL OF THE MACHINE. *
989 * * THIS IS A PRIVILEGED INSTRUCTION WHICH IS GENERALLY ONLY USED *
990 * * IN THE OPERATING SYSTEM FOR CONTEXT SWITCHING! *
993 * * FRESTRT <EA>
995 * * WHERE: <EA>= POSTINCREMENT MODE (AN)+
997 * * REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7
999 * * A X
1000 * * D X
1001 * * VALID ADDRESSING MODES: (AN)+
1003 * *
1004 ***************************************************************
1005 FRESTRT MACRO
1006 MOVE.L \l,D0
MOVE FORMAT WORD AND RESERVED WORD TO DO
1007 SWAP DO
PLACE FORMAT WORD AS THE LOW ORDER
1008 MOVE.W DO,MC68881+RESTORE
STORE FORMAT WORD IN RESTORE REG.
1009 MOVE.W DO,MC68881+RESTORE,D0
READ THE RESTORE REGISTER
1010 MOVE.W DO,D1
MAKE A COPY OF THE RESPONSE FORMAT WORD
1011 ANDI.W #$FF00,D1
ISOLATE THE FORMAT WORD
1012 BEQ.S \NULREL
IF NULL IDLE RESPONSE, NO STATE RESTORED
1013 LEA MC68881+OPER,A0
LOAD OPERAND REGISTER TO A0
1014 * *
1015 MOVE.B DO,D1
THE LENGTH OF THE DATA TO BE TRANSFERRED
1016 LSR.B #2,D1
DEVIDE BY 2 TO ADJUST FOR WORD TRANSFER
1017 EXT.W D1
ESTABLISH COUNT AS A WORD FOR DBRA
1018 \LOAD MOVE.L \l,(A0)
STORE THE INVISIBLE STATE
1020 DBRA D1,\LOAD
REPEAT UNTIL ALL DATA IS TRANSFERRED
1021 \NULREL EQU *
1022 ENDM
1023 ***************************************************************
1024 * * MC68881 FNOPP COMMAND *
1026 * * FNOP *
1027 * * NO REGISTERS MODIFIED OR DESTROYED! *
1030 * *
1031 ***************************************************************
1032 FNOPP MACRO
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Comment 1</th>
<th>Comment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1033</td>
<td>MOVE.W #$0000,MC68881+COND</td>
<td></td>
<td>FNOP COMMAND TO FP REG.</td>
<td>TEST RESPONSE</td>
</tr>
<tr>
<td>1034</td>
<td>\NOPAS</td>
<td>TST.B MC68881+RESPONSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1035</td>
<td>BMI.S \NOPAS</td>
<td>ENDM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1036</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>