John Gilmore, Independent Consultant
431 Ashbury St.
San Francisco, CA 94117
Voice: (415) 621-9355 ABBS (300 baud): (415) 863-4703

December 12, 1980

Suggested Enhancements to the Motorola MC68000

The MC68000 is an outstanding processor -- one that the adjective "micro" does no justice to. It embodies a large-scale architecture, a very major advance over all other single-chip processors in existence. It does have rough edges, though, and it lacks facilities that could be provided at reasonable cost and with minimal changes. The body of this paper contains my suggestions for this "rounding out" process.

First, a personal comment. I wish the same functions had been provided without such extreme corner-cutting in encoding. The reusing of addressing modes in some instructions to encode other instructions, especially in light of the much-touted across-the-board consistency in addressing, leaves me wishing a few more opcodes had been used and the addressing left alone. I think a simpler instruction encoding -- simpler to explain, simpler to compile for, and simpler to decode -- would have been worth it.

1. Motorola documentation does not make any distinctions between the architecture of the 68000 family and the implementation of the MC68000. For example, the User's Manual states that all instructions must be aligned, but Motorola has said they might allow non-aligned instructions in future. It would allow users to plan much more effectively for upward compatability if they had a better idea what parts of the MC68000 implementation were considered relatively stable and which might be open to change.

2. Support virtual memory. The current chip does not provide sufficient information to recover from a page fault; in particular, (A) the instruction that was executing cannot in general be identified, (B) there is no way to determine how far execution of the instruction had progressed before the page fault, and (C) any executing instruction is aborted when a page fault occurs because of instruction pre-fetching.

While many instructions can be successfully restarted with heuristics, there are some that this is not possible for (eg MOVEM (A3),A0-A7 with a fault after A6 has been loaded. If we knew that A6 was the last one loaded, it would be possible -- but we don't).

The solution is for the chip to leave enough information around to enable us to restart, albeit with trouble. For example, (A) provide the address of the faulting instruction, (B) either rescind changes made by the instruction (hard), or provide sufficient information for software to recover or resume (easier); (C) Queue bus error indications from pre-fetch so that the trap only occurs when (and if) the instruction is decoded for execution. (This requires that the bus interface unit be able to get a bus error indication and then go thru several other bus cycles before taking the trap, but is vital.)

Clearly, the bugaboo is (B) -- rescinding changes or saying what has been changed. A complex, but workable, software algorithm for completing faulted instructions can probably be made with only two more pieces of information: the number of operand read bus cycles and the number of operand write bus cycles since the instruction began. (If immediate operands are accessed in mid-instruction, the count must include immediate operand reads, for cases like MOVE.L #X,D0 where the top half of D0 has been clobbered before faulting on reading the second word of the literal. If immediate operands are all read before "true execution" begins, this is not a problem.) It has not been proven that this information is always sufficient; clearly this needs to be done before considering solidifying such a scheme in silicon.

3. Cause an Address Trap whenever an address is used which does not fit on the physical address bus used. The high-order bit of the physical bus should represent all high-order bits of the EA; that is, if bit 23 (the uppermost) of the 24-bit address bus is 0, bits 23 thru 31 of the EA must have been zero; if the uppermost bit is 1, bits 23 thru 31 of the EA must have been one; otherwise (if bits 23 thru 31 of the EA were not all the same), an address trap occurs and no memory access is made. With a 24-bit address bus, addresses from 00000000-007FFFFF and FF800000-FFFFFFFF are OK; the rest (00800000-FF7FFFFF) produce Address Traps.

Why: It enforces upward compatability. 24-bit-bus users' programs will run correctly on a 32-bit-bus system, because they aren't depending on the truncation of addresses (which people are likely to rely on, as they do in System/360/370, and which will screw them when larger machines are released).

4. The Bit Manipulation instructions should be superceded by a new set of instructions, as there are too many things wrong with the current ones. The old instructions would continue to be supported as dinosaurs. The replacements (and the new BSX) will:

(A) number bits from high-order to low-order.
Numbering the bits in the same order as the bytes makes the machine's addressing totally consistent, and allows (B). (See "Internetworking Experiment Note #137: On Holy Wars and a Plea for Peace" by Danny Cohen of USC/ISI, for some historical background on bit orders.)

(B) correctly process bit numbers other than 0 thru 31.
Bit number n in a string should simply be addressed that way, no matter what the magnitude of n. The processor would shift off the low 3 bits of the longword bit number as bit-within-byte, and add the rest to the supplied byte address. Negative bit numbers are valid.

(C) use the Read-Modify-Write memory cycle.
Interleaved accesses by two processors (to two different bits in the same byte) can lose one of the modifications unless this is done. Explicit interlocking via TAS is clumsy and not well supported by high-level languages. Read-Modify-Write exacts a small penalty in DMA performance and in bus error recovery -- and it's worth it.

(D) set the X flag the same as the addressed bit.
X can be set by BSET, BCLR, BCHG, or BTST then shifted into a register or memory by ROXL/R. A bit shifted to X by ROXL/R can be stored by BSX, which sets the addressed bit from the X flag. A bit can be "picked up" by BTST then "put down" by BSX. These are useful in applications like the Data Encryption Standard or bit-array generation and processing.

(E) use short instructions for flags in stack frames or low memory. It currently takes 3 words to address a bit when using indexed, displacement, or short absolute addressing. Immediate bit numbers of 0-7 should appear in the first word of the instruction, making 2 words total for the above cases. Larger immediate values, or bit numbers in registers, can use larger instruction formats due to their lower frequency.

5. Provide instructions to aid in converting binary to and from BCD and ASCII. The four proposed instructions (TASC, TBCD, FASC, and FBCD) each use a long unsigned data register and a byte EA (using any addressing mode). TASC takes the value in the data register, divides it by 10, and returns the quotient to the data register, setting the condition codes from the quotient. It stores the remainder in the low nibble of the EA, with a 3 in the high nibble. Thus to convert an entire number, one might do:

```
MUMBLE:  TASC  D0,-(A3)   convert one digit
         BNE   MUMBLE     loop until done
```

FASC would fetch the byte at the EA, do nothing except set C if the byte was not between hex 30 and 39, else multiply Dn by 10 and add the low nibble, clearing C and setting V, N, and Z based on the result. To convert a whole number:

```
         CLR   D0          start off with zero
GETDIG:  FASC  D0,(A3)+    add in a digit
         BCC   GETDIG      loop
* fall thru when we've seen a non-digit.
```

Similarly, TBCD would divide by 100 and make two BCD digits out of the remainder, FBCD would multiply by 100, convert the byte of BCD to binary, and add it in. If either nibble is not in the range 0-9, FBCD sets C and no-ops. Why: Binary <-> decimal data conversion has always been a problem; this would simplify things for the majority of conversions being done in the 68000. ASCII <-> BCD conversions are already in the works as PACK and UNPK.

6. Reassign the opcode for ORI. Why: A word of zeros is an ORI.B to data register zero. It should be an illegal instruction. (Opcode 00001110 is free and in the right format.) At least provide a second opcode for ORI now, so people will use the new one and let the old one be phased out sometime in the future. Software could easily allow older programs to run, by intercepting the invalid instruction trap and changing the opcode or simulating the ORI (unless the instruction was ORI.B #0,D0 -- two words of zeros). Most users would gladly reassemble their programs, having old ORIs fixed on the fly until they did, so that wild branches into blocks of zeros could be trapped.

7. Provide an on-chip identifier which is accessible to user and supervisor software. The identifier should consist of 32 bits, where the high-order bits identify the processor revision, and the low-order bits identify the physical chip. The identifier need not be unique to the chip (although that would be nice), but at least 5 or 6 bits of uniqueness should be provided. The identifier must not be changeable once the chip has left the factory. Possible technologies: a small PROM programmed before packaging, by electronic means or by laser; or rather than replicating a mask exactly n times onto a wafer of silicon that would eventually produce n chips, have each copy on the wafer be slightly different in having a different circuit for the on-chip identifier ROM; this shouldn't affect too many of the masks, and if there are 64 on a wafer, this is enough uniqueness. It doesn't matter if some of the chips drop out later as long as 64 different kinds of chips come out in reasonably-close numbers. (With computer-controlled electron beam lithography, a unique number could be assigned to each chip as it is exposed.)

Why: to provide some protection against software piracy. Purchased software would contain the user's processor identifier in some encoded form before being shipped. The software would check this identifier during operation to ensure it matched the identifier of the current processor. If the software is pirated to another processor, it can refuse to operate (or refuse to operate correctly). (The checking of the processor identifier must be done with some stealth by the software designer, as the user could run the application in trace mode to find out where the check is made and patch it out. This will provide great opportunities for ingenuity on the part of software designers and security-breakers both. Nevertheless, having the identifier accessible is much, much better than not having it: most end-users will not have the inclination or knowledge to defuse the check if it is known to exist. Currently, of course, it is known not to exist, encouraging piracy even among normally "moral" people.)

8. Provide a 32-bit by 32-bit multiply instruction that returns a 32-bit result, and sets V if an overflow occurs. Signed and unsigned would be nice, as well as word multiplies that produce a word result (and V for overflow). Why: Except for the very unusual case of multi-precision work, n-by-n-giving-2n multiplies are painful for programmers to use. Most of us take the easy way out and simply ignore the top n bits of the product. There were several system security breaches in IBM's APL\360 system that were a direct result of this. (The system was multiplying the dimensions of an array together to determine how much space it would occupy; clever users created arrays with 65536 rows and 65536 columns, which multiplied into "0" bytes of space, but allowed them to access all of main storage with valid indices into the array.)

9. Provide a 32-bit by 32-bit divide instruction giving a 32-bit quotient and remainder. Signed and unsigned would be nice again. The instruction should explicitly specify the register which will contain the remainder, rather than having it be a register "next to" the quotient register, for two reasons: flexibility in register assignments, especially for compilers; and to allow a divide-without-remainder by specifying the same register as quotient and remainder. Many applications never use the remainder anyway. The Carry bit should be set if the remainder is nonzero in any case; this allows a program to check if the divide was "even", whether or not it requested a remainder. Word-by-word-giving-word divides with the same characteristics should also be supplied. Why: Because most users will want to divide longwords by longwords or words by words, not 64-bits by longwords or longwords by words.

10. Use one of the free bits in an EA extension word (currently containing the index register and offset) to indicate that the index register is in elements rather than bytes: if the instruction size is Word, shift the index register left one bit before adding it to the base and displacement; if the instruction size is Long, shift the index register left two bits; if the instruction size is Byte, do no shifting; if the instructin is Unsized, take an Invalid Instruction trap. Why: subscripting.

11. Provide all 32 bits on the address bus. This is important for applications that do serious virtual memory work (as other manufacturers have found out) -- such as making all files on a hard disk (or videodisk!) addressable by normal instructions without having to OPEN, READ, POSITION, etc.

12. Provide a small instruction cache for use in DBcc loops. The cache would store a small number of words, on the order of 3 to 20. A 32-bit cache base address register and a small cache length register would be used in conjunction with a comparision network that would indicate whether a given memory address (to be placed on the address bus for reading or writing) was within the cache. Any write into the same address range clears the cache (by setting its length to zero). Operand reads from the range are unaffected (as they might be coming from data space instead of instruction space) and do not

affect the cache. The cache is flushed by a successful branch instruction that branches to outside the range of the cache; this prevents it from getting fragmented and requiring more registers to keep track of it.

Successful branch instructions that branch into the range of the cache cause "cache fetch mode" to be entered, in which instruction fetches come from the cache. Cache fetch mode ends when the cache is flushed, or when the end of the cache is reached; normal instruction fetching then resumes.

Normal instruction fetching always adds the fetched instruction to the cache and increments its length (possibly dropping an old instruction, if it is full). The cache is always either empty, ends just before the current instruction, or is being fetched from.

Why: The flexibility of the autoincrement, autodecrement, DBcc approach to repetitive processes is wonderful, and adding specialized instructions (eg move a block of bytes) just for speed would be distasteful. But at least 1/3 of the time taken in such loops is consumed by instruction fetch. A typical MOVE loop, such as

```
FOO:  MOVE.L (A0)+,(A1)+
      DBRA   D0,FOO
```

consumes 11 cycles in doing work (4 to read a longword, 4 to write one, and 3 to update and test the counter) and 6 cycles in reading instructions, for each longword moved. More complicated loops spend even a higher percentage of their time in instruction reads. A 3-word cache would handle move, clear, compare, and multi-precision loops; extending this to 8 or 10 words would handle most hand-assembled loops and some compiler-generated loops.

13. As an alternative to providing an instruction cache, do the following: If the displacement of a successful DBcc instruction is -4, fetch the instruction at that address and save it in an internal register. Then go into a special execution mode in which that instruction is presented to the rest of the microcode for execution, then the condition testing and decrementing specified by the DBcc are done, and the process is repeated until the condition or decrement fails, or a trap occurs. The looping action is performed without further reference to main storage; the (1-word) instruction, the condition, and the data register to be decremented are saved in processor-internal storage. (If a trace or other trap occurs, of course, the PC must point to the right place.) Why: This will speed up MOVE, CMPM, ADDX, SUBX, NEGX, ABCD, SBCD, NBCD, and CLR loops at a much lower implementation cost than suggestion #12. It doesn't help larger loops such as addition of two vectors.

14. Provide a mode, controlled by a CCR bit, which would cause all instructions that read and subsequently replace an operand in storage to use the read-modify-write cycle, rather than a read followed by a write. This includes ADD, ADDX, NEG, NEGX, SUB, SUBX, AND, OR, EOR, NOT, ANDI, ORI, EORI, ADDI, SUBI, ADDQ, SUBQ, shifts and rotates, ABCD, SBCD, and NBCD. (This assumes that the bit operations are always interlocked; see #4.) Having the option of having the next Read-Modify-Write cycle turn off the mode bit would be very nice. This would allow a program to interlock all following instructions with ORI #LOCK,CCR; or to just interlock the next instruction with ORI #LOCK+TEMP,CCR.

Why: While it can't synchronize longword operations, it provides much better (and easier-to-use) protection in a multiprocessor system than explicit use of TAS whenever shared data is being modified.

15. Allow the TST, CLR, NEG, NEGX, NOT, ADDI, SUBI, and CMPI instructions to act on address registers. Why: some are useful (haven't you ever tested a pointer against zero?); no reason to restrict the rest. The immediate instructions can be simulated with ADDA, SUBA, and CMPA with immediate operands, but this makes it harder to compile code and blows "consistency". One can ADDQ but not ADDI to an address register!

16. On a RESET exception, the MPU fetches the new SSP while it still has the old PC; it should push it onto the newly-established supervisor stack, then fetch the new PC. If the system hangs, and RESET is the only way out, it sure is nice to be able to tell what piece of code hung it up. Having the old SSP would be nice, too, if there's a register in the chip where it could be saved until it's pushed.

17. Currently the overflow flag must be tested after each instruction that might set it, if indeed one wants to detect overflows. A possible solution might be to have the V flag never turned off except by explicit programmer action (ANDI to CCR, for example); this would allow the use of a single test at the end of a series of instructions. This, unfortunately, messes up the GE, LT, GT, and LE condition code tests, which depend on V's being set at the same time as Z, N, and C.

Another possibility is to have a mode which would cause a trap immediately after (or during) an instruction which sets V on. This has the side effect of taking traps on innocent instructions like CMP, which must set V the same way as SUB, or the multi-precision instructions, which might set V in the middle of an operation.

Probably the best way to fix it is to disregard the current mechanism and add what would do the job best. A new bit in the CCR should cause the following instructions to trap rather than setting V: ADD, ADDI, ADDQ, SUB, SUBI, SUBQ, DIVS, DIVU, NEG, ASL, and the proposed multiply instructions. Other instructions that set V are unchanged. The trap takes place before the results of the instruction are stored; no registers or memory are affected by the instruction.

18. The 68000 is weak on mixed-data-width operations, which is a shame since it supports so many different types of data. That could be remedied by putting a mode bit in the CCR which would cause two-operand instructions with data register results to sign-extend the other operand and use the whole register. In particular, ADD, ADDI, SUB, and SUBI would use the whole register as an operand (independent of the specified size of the EA or immediate operand) and return the result to the whole register. CMP, CMPI, and CHK would compare the entire data register. MOVE.B, MOVE.W, EXT.W, MOVE SR/CCR, MOVEP, and Scc would set the entire data register. It doesn't make sense to extend the logical operations, as the upper bits would either be unaffected or cleared; and the one-operand instructions need not deal with two sizes of operands.

Note that MOVEM already has this effect -- it should be possible in all operations. Also note that operations on address registers provide some of the above, but they don't set condition codes, don't support byte operands, and don't include all operations.

19. An IEEE standard floating point co-processor would be nice.

20. Allow the TST instruction on all addressing modes, not just alterable ones. Why: No reason to restrict it. There are several ways non-predictable data could be in I-space; consider a control table linked in with a code module, or some operating-system-provided, read-only parameters. (Note that BTST is correct in this regard.)

21. In the TST and TAS instructions, set the Carry bit to the low-order bit of the operand. This provides an "odd/even" test. Why: This won't mess up any condition tests, and allows the testing of several flags at once.

22. Provide byte, word, and long variants of subtraction which perform EA-Dn->Dn and Dn-EA->EA operations. Why: Since subtraction is non-associative, a compiler (or assembler language programmer) must watch out for which operand is where the result will go. Having an instruction where the minuend and result correspond, instead of the subtrahend and result, removes this difficulty. This also applies to division, if/when division has the flexibility and attention that subtraction has (Word/Long, result to Dn or Ea, etc).

23. Provide a conditional Trap instruction with all 16 possible conditions and the ability to pass a 6-to-8-bit literal to the trap handler. (The instruction should probably not vector; but just make the literal available to the handler). The instruction should execute in minimum time (two cycles) if the condition is not satisfied. Why: quick validity tests (ASSERT statements) and run-time error checks. This instruction can also obviate TRAPV and CHK. (It can currently be simulated with a Bcc to an odd address, but that might not work forever.)

24. Provide a return instruction which specifies a literal value, to be subtracted from the stack pointer after the PC is popped. Why: Subroutine calls with arguments on the stack require the caller to decrement the stack pointer past the arguments, after the routine returns. This instruction would allow the callee to do it, saving space and complexity.

25. Provide indirect Jump and JSR instructions. The EA could address a word self-relative, longword self-relative, or longword absolute pointer, at the user's option. Why: branch tables. Indirection can generally be simulated by loading the pointer, but one may want to JUMP or JSR without having a free register.

26. Provide a CARRY instruction which would add the C (or X) flag to bit 8 or 16 of a data register. Ideally it would have three variants: byte-to-word, word-to-long, and byte-to-long. Why: This would enhance the 68000's ability to add and subtract smaller-sized quantities to larger quantities. Following an ADD.B by a CARRY.B Dn.L would add the byte to the long register. This is especially useful in summing a series of bytes or words into a data register. Also see #18.

27. The SR should be extended to 32 bits. Why: I've already proposed enough new bits to fill the current 16-bit one!

28. ABCD and SBCD should trap if presented with non-BCD data.

29. No instructions should leave the condition codes undefined. Currently ABCD, CHK, NBCD, and SBCD do.