Design Philosophy Behind Motorola's MC68000

Part 1: A 16-bit processor with multiple 32-bit registers.

Thomas W. Starnes
Motorola Inc., Microprocessor Division
3501 Ed Bluestein Blvd.
Austin, TX 78721

In the mid 1970s at Motorola, a new idea was taking shape. As more and more demands were being made on the MC6800 family of microprocessors, the push was on toward developing greater programmability of a 16-bit microprocessor. A project to develop the MC68000, known as Motorola's Advanced Computer System on Silicon (MACSS), was started.

The project team began with the freedom to design this entirely new product to best fit the needs of the microprocessor marketplace. Developers at Motorola explored many possibilities and made many difficult decisions. The result can be seen in the MC68000, viewed by most industry experts as the most powerful, yet easy to program, microprocessor available. In this first of four articles, I will discuss many of the philosophies behind the design choices that were made on the MC68000.

Many criteria can qualify a processor as an 8-, 16-, or 32-bit device. A manufacturer might base its label on the width of the data bus, address bus, data sizes, internal data paths, arithmetic and logic unit (ALU), and/or fundamental operation code (op code). Generally, the data-bus size has determined the processor size, though perhaps the best choice would be based on the size of the op code. I'll talk a bit about these features and then show how the MC68000 is both a 16- and 32-bit microprocessor.

About the Author

Thomas Starnes is an electrical engineer who has spent the last five years helping to plan the direction of the MC68000 family of processor products for Motorola.

Shaping a Design

Designers must make hundreds of decisions to shape the architecture of a new microprocessor. The needs of the users of the new product must be considered as the most important factors. After all, the users are the ones who really need a functional product, and if they are not happy with the features or performance, they will keep looking for a better alternative.

Unfortunately, it may be impossible to meet all of the needs of the users due to certain design limitations. The design must be inexpensive enough to produce in mass quantity. Also, current technology will permit only certain types and numbers of circuits to be manufactured on a silicon chip. These are the foremost factors that dictate the upper limits of the capabilities of a microprocessor.

In planning the new 16-bit MACSS, designers had to make a decision concerning the general architecture first. What should it look like? A great deal of software written for the MC6800 family already existed. A processor that provides enhancements over an older processor, yet can run all of the programs for the older processor, has a real asset: it can capitalize on the existing software base. This may attract users by ensuring that they won't have to rewrite at least some of their programs.

Unfortunately, architectures, such as the early 8-bit microprocessors, were rather crude. Because they were designed to replace logic circuits, not enough thought was put into the software aspect of the parts. Their instruction set was oriented toward hardware. The designers did not consider carefully the future of these products, their expandability and compatibility. To try to design a microprocessor to be compatible with the older 8-bit parts was limiting.

Designers at Motorola decided that the new MACSS should be the fastest, most flexible processor available. They would design it for programmers, to make their job easier, by providing functions in a way that most programmers could best use them.

Early on, it appeared that to have a really powerful new generation of microprocessors, a totally new

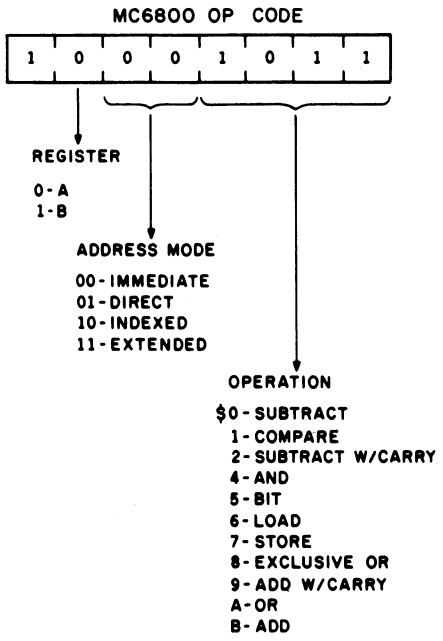


Figure 1: *Op code organization for the MC6800. This processor is limited in its abilities because of its 8-bit size.*

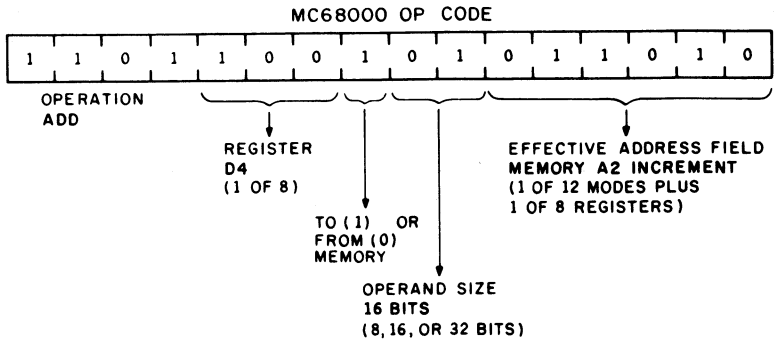


Figure 2: *The MC68000 ADD instruction op code shows the power available with 16-bit operations. Multiple registers with variable operand sizes and a large address field give a programmer tremendous flexibility in programming.*

architecture should be used and that earlier designs should be considered as examples rather than as models. This gave the MC68000 designers the freedom to introduce completely new concepts into microprocessors and to optimize the functionality of the new chip.

The planners decided there was one area in which ties to the 8-bit product family would be advantageous without exception. That area was in peripherals. Motorola decided that this new 16-bit microprocessor would directly interface to the 8-bit collection of MC6800 peripherals. Because so many input/output (I/O) operations are 8-bit oriented, it seemed logical to retain this compatibility even though the 8-bit microprocessor interface would naturally be about half as fast as a comparable 16-bit. Compatability with 8-bit MC6800 peripherals had the added benefit of immediately ensuring support of the new microprocessor with a complete family of peripheral chips, rather than requiring a wait of perhaps years for 16-bit versions to become available.

Expanded Capabilities

A properly designed 16-bit microprocessor has many advantages over the most sophisticated 8-bit microprocessor, especially to the programmer (see figures 1 and 2). The 8 bits of op code for the smaller processor provide only 256 different instruction variations. This may seem to be a lot at first glance, but consider the following.

If the microprocessor has two registers from which to move and manipulate data, those two registers require 1 bit for encoding the op code. If four different addressing modes are offered for accessing memory data, these require 2 more bits for encoding. This leaves the microprocessor with only 5 bits with which to encode the operation to be performed. Only 32 different operations can be performed.

Now admittedly this is plenty of operations for most applications, but realize that only two data registers and four memory-addressing modes are not very many to someone doing serious programming. Registers are there for fast data manipulation, and constantly swapping the contents of too few registers is not very fast. A more powerful microprocessor would have many registers, and they would all have to be accessible by the different operations.

Additionally, the more addressing modes you have for accessing memory data, the more efficiently you can get values in memory. Obviously, 8 bits of op code cannot give the microprocessor both the variety and the number of operations that a good 16-bit microprocessor can. With 64,000 different instructions possible in a 16-bit op code, you can perform far more complex operations.

This, then, is the real advantage of 16-bit over 8-bit microprocessors to the programmer. A 16-bit microprocessor will have twice the data-bus width of the 8-bit version. This wider bus allows twice as much information to go in and out of the processor in the same amount of time. This can, with proper internal design, almost double the rate at which operations take place over the rate of a similar 8-bit machine. Sixteen-bit micropro-

MC68010

Motorola has recently developed an improved version of the MC68000: the MC68010. It is completely compatible with object codes of earlier versions of the 68000 and has added virtual memory support and improved loop instruction execution.

By using virtual memory techniques, the 68010 can appear to access up to 16 megabytes of memory when considerably less physical memory is available to a user. The physical memory can be accessed by the microprocessor while a much larger "virtual" memory is maintained as an image on a secondary storage device such as a floppy disk. When the microprocessor is instructed to access a location in the virtual memory that is not within the physical memory (referred to as a page fault*), the access is suspended while the location and data are retrieved from the floppy disk and placed into physical memory. Then the suspended access is completed. The 68010 provides hardware support for virtual memory with the ability to suspend an instruction when a page fault is detected and then to complete the instruction after physical memory has been updated.*

The MC68010 uses instruction continuation rather than instruction restart to support virtual memory. When a page fault occurs, the microprocessor stores its internal state on the supervisor stack. When the page fault has been repaired, the previous internal state is reloaded into the microprocessor, and it continues with the suspended instruction. Instruction continuation has the additional advantage of handling hardware support for virtual I/O devices.

As mentioned in the body of this article, the 68000 uses a prefetch queue to improve the speed of instruction execution. The 68010 goes one step further by making the prefetch queue more intelligent. Detection of a three-word looping instruction will put the microprocessor into a special mode. In this loop mode, the microprocessor will need only to make data transfers on the bus, because it latches up the queue and executes the instruction repeatedly out of the queue. Once the termination condition for the loop is reached, normal operation of the prefetch queue is resumed. This operation is invisible to the programmer and provides efficient execution of program loops.

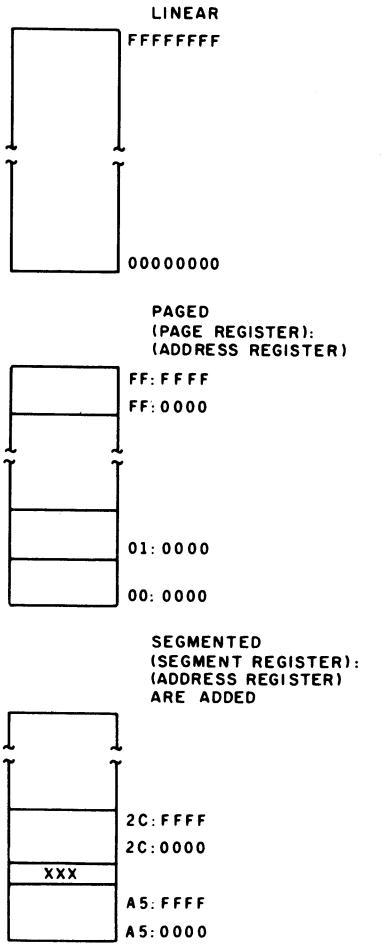


Figure 3: *Three methods of addressing memory. The Linear method arranges a contiguous memory area. The Paged method organizes memory into blocks or pages of a prescribed length. The Segmented method gives each user or program a specific area in memory. Both the Paged and the Segmented method give the programmer access to only a small portion of memory.*

cessors should give the programmer far greater flexibility in coding and perform similar operations in less than half the time of an 8-bit microprocessor.

Memory Accessing

Users of the 8-bit microprocessors originally had difficulty imagining what kind of programs could fill up 64K bytes of memory. Many systems had no more than 8K bytes of ROM (read-only memory) and RAM (random-access read/write memory). But as time went on and the general software base grew, systems with up to 64K bytes of memory became more prevalent. Either code had to become more efficient or ways of fitting more than 64K bytes of memory in a system had to be developed. Sixteen-bit microprocessors could make code more efficient.

In planning MACSS, designers foresaw that the 16-bit, 64K-byte addressing range of popular 8-bit microprocessors would be quickly outgrown by the newly proposed microprocessor. Each additional bit of address could double the addressing range of the processor.

Look at the techniques of expanding beyond a 16-bit addressing range and analyze the design trade-offs (see figure 3). You could extend the addressing range of early computers and minicomputers simply by appending some additional bits to the most significant of the 16 address bits. These additional bits were usually stored in an additional register, the page register. This method is called *paging*, because you work out of one page at a time. The page is set manually, and the lower 16-bits of address are included in the instruction stream or registers.

Paging has the advantage of being quite simple to implement in the processor. No real circuit change is needed over the straightforward 16-bit addressing, because all the expansion is done simply by appending bits to the core. It also has the advantage of having fairly dense code, because only 16 bits of address are carried around in the instructions.

However, there are many disadvantages to paging. The programmer is limited to accessing only the particular page of memory that happens to be set in the page register. To be assured that the right page is being used requires a check to see what is currently in the page register, possibly saving that page number, and loading the register with the desired page number. This takes time and requires both additional thought by the programmer and additional

code in the software. This additional code typically takes up the room saved by carrying around only 16 bits of address.

One way to get around the single-page limitation of paging is to provide many page registers. Other characteristics that determine which register will be active on a particular bus cycle include instruction fetch, data read/write, and stack access. While these additional registers give the programmer access to more than one page at a time, there is still only one page available for each type of access.

Some extensions to paging came out to compensate for some of the losses experienced in paging. *Segmentation*, for example, follows the same general principles of pagination. The key difference in segmentation is that the page number becomes a segment number and the segment number is essentially added to the core 16-bit address. This allows some relocation of the core address but still forces the programmer to check that the desired segment is loaded, and limits the range of any segment to only 64K bytes of memory.

To a programmer, the simplest address technique is a direct addressing of any memory location. This would be without regard for whether the wanted data is near recently accessed data or whether it is miles away. The programmer wants a linear view of data, that is, the ability to specify a very simple, albeit long, address that will access any data.

Now, beyond the processor's memory-addressing method, memory management is sometimes used. With it more sophisticated systems dynamically relocate or control the various blocks of memory. This is done for protection purposes in larger systems. The advantage is that you can protect one user's work space from the devastating effects of another user's poor programs running amuck. To this end, a separate memory management unit (MMU), in conjunction with the operating system, performs some addition to or translation of an address. This technique may sound similar to paging and segmenting memory, but this is done to serve a completely different purpose, and in a different way. The application program writer never sees this memory management and writes code as though the entire memory were available.

To expand the memory space on the MACSS, the best option, though not the easiest to implement on the chip, is a linear address space. This space is not broken up by paging, segmentation, or banking schemes. It is a very simple addressing technique, requiring the least effort by the programmer, while still allowing more advanced operations such as memory management.

A *linear address* is simply a straightforward 32-bit, for example, address. The address space is not broken up into blocks; and it is contiguous. Accessing such an address merely requires the expression of the 32-bits in the instruction or using a single address register. For convenience, if the upper 16 bits of the address are either all 0s or all 1s, then a shorter, 16-bit form of the address can be sign-extended to automatically provide the correct address. This is the way the MC68000 accesses memory and I/O.

How big an address space should a 16-bit microprocessor address? The natural address sizes greater than 16 bits are 24 and 32 bits, which are 3 and 4 bytes long, respectively. For a 16-bit microprocessor, the odd number of bytes becomes slightly unwieldy. Looking a little further into the future, it seemed that even the 16 megabytes of a 24-bit address might not meet the needs of large systems.

While 32 bits of address, reaching 4 gigabytes of memory, seems tremendous, once the need for more than 16 bits is established, 32 bits is the next most convenient size. It takes exactly two 16-bit bus transfers to move an address into the processor, and once the second transfer is needed, as it would be even for an 18-bit address, it is just as well to use the whole 16 bits brought in. Thus, engineers selected a virtual-memory address space of 32 bits for the MC68000.

Now, from a practical packaging standpoint, 32 address signal lines are quite a few. The placement of integrated circuits (ICs) in dual inline packages (DIPs) with greater than 40 leads was rare before 1980. With only a few systems in the early '80s requiring more than 16 megabytes of memory, it seemed a reasonable trade-off to bring only the 24 least significant address bits to the outside world. That way fewer pins would be required, and MACSS could fit within a 64-pin DIP. Still, all 32 bits of address are maintained within the processor, and there are simple means of determining the upper 8 bits' values.

Multiple Registers

With the size of the memory address space determined, it was easier to settle on the register scheme of the new processor. The size and the number of registers had to be decided.

Designers originally envisioned onboard registers for a processor because operating on memory data requires a time-consuming transfer across the external bus. It just happens that in programming most data is operated on a number of times in succession before a result is obtained. Often many combinations with many different data pieces are used. The merging of these two observations leads to onboard or on-chip registers for fast manipulation of frequently used data.

It seems that from the day registers were brought into the processor, programmers have wanted more registers for their use. The goal, then, when designing processors, is to provide as many registers as possible for the programmer. In the MC6800, only two registers (A and B) were available for data manipulation, and one index register (X) to point to non-stack data. These few registers are being loaded and saved almost as often as the data within them is manipulated.

The loading and saving of registers is usually wasted time. The amount of time spent bringing data into on-chip registers for fast manipulation

depends upon the exact use of that data. However, the more registers available, the more likely it is that a register will not have to be saved just so that some other data can be operated on in that register.

The design of the internal execution of instructions through a microprocessor will determine many things about the suitability of the chip for programming. Instructions may operate either on what are called *dedicated registers* or on a *general register set*. Each of these methods has advantages and disadvantages.

In a microprocessor that uses dedicated registers, an instruction includes the address of the data to be worked on in specific registers. These registers are inherent in the instruction. The ADD instruction, for example, will add only from a memory location to, say, register A—not to register B, and not from register A to memory. If the value to be added to is not already in register A, it must first be placed there. Before it can be placed there, a number in A may have to be saved. All of this can be quite troublesome. This is not very different from the situation in which there simply are not enough registers.

Contrast this with the example of a processor that uses true general-purpose registers. In a general-register machine, the ADD instruction may add data from memory to any of the internal registers. The instruction must contain information on which register it will operate on. This is determined when the instruction is assembled. If there were four registers in the processor, the ADD operation could be performed in register A, B, C, or D, as selected by the programmer.

Now if the value to be added to is in register C, the programmer simply designates C as the operand register. There is no need to shuffle registers and no need to save any register contents. The general-register machine, then, is easier to program and typically requires less time to execute an operation.

As it always happens, this ease of programming does not come free. You will see later that allowing a selection of registers requires bits in the op code for encoding and, therefore, more bits of the op code. Also, it is typically more difficult for the microprocessor designer to implement the circuitry that incorporates various registers because it takes time to determine which register is to be used and to activate that register. Streamlining internal operations so that this time is not detectable requires quite a bit of planning.

So while fewer registers or dedicated registers may be easier for the microprocessor designer to implement, they make programming the new chip more cumbersome and less flexible. But the extra time, effort, and expense of implementing general-register principles pays off by easing the programming of these devices.

Therefore, the MC68000 was designed with general-purpose registers. Any instruction may select any register for use as a source or destination operand or as a pointer in any allowable addressing mode. This tremendous flexibility gives programmers the ultimate in data and pointer placement.

A close observation of the use of registers indicates they usually have one of two purposes: they may retain data for manipulation, or they may contain an address that points to a memory location. The use of a register for each of these purposes is quite different.

When data is moved into or out of a register or is manipulated within the register, all types of conditional information from the operation are important. Thus, you typically would like all condition codes to be properly set after a data operation. This way these condition codes may be used to branch or with other data operations.

On the other hand, an address might be placed in or taken from a register, or modified by incrementing or decrementing. Rarely is it important whether a carry comes out of the ALU or whether the result is negative (i.e., has a 1 in the most significant bit). In fact, a programmer would prefer manipulation of an address to have no effect on the condition codes. Often in the middle of a complex data operation, you must bring in a new address or increment an address. To have this operation modify the condition codes most of the time will foul up the data operation in progress, and so is undesirable.

Therefore, two generic register types emerge: a data register (D0 through D7) and an address register (A0 through A7). The MC68000 has both types. In a data register, any operation will affect the condition codes of the microprocessor as is appropriate for the operation and the data used. However, in an address-register operation, condition codes will not be changed, but the codes from previous data operations will be retained. This way you can have address and index pointer changes made, without affecting the accuracy of the results, in the middle of a complex data operation that requires many instructions and transfers from memory.

What size and how many of each type of register should be included in the microprocessor? The more registers there are, the better it is for the programmer. Unfortunately, the more register and control circuits in the chip, the more expensive it is. A good balance must be attained.

Two registers are too few, four are nice, but it is difficult to imagine even a complex routine requiring more than eight different memory pointers. The encoding of eight registers requires an even three bits. Because it seemed that eight was a good upper bound, the MC68000 has eight address registers and also eight data registers.

With 16 registers available, divided half and half for data and address, almost any sizable routine will never require the temporary storing of a value in a register just so that the register can be used for something else. And, within the routine, manipulations of memory pointers in address registers will not interfere with an ongoing data calculation, because of the distinction of how the condition codes work for the different register types. It is easy to see how the MC68000 is easier to program.

Earlier I explained that MACSS would handle all of its addresses as 32-bit quantities. Anyone who has ever programmed 8-bit microprocessors, which have 8-bit accumulators and 16-bit index registers, has seen the difficulty with the two different sizes. Once programmers figure out how to put the 16-bit value in both 8-bit accumulators, things get tougher when they try to get arithmetic carries from the lower half to the upper half of the value.

A little of this experience led the MC68000 designers to decide that using data that is the same size as the address register could make some software design significantly easier. In order to handle a linear 32-bit virtual-address space, the MC68000 needed to have 32-bit address registers. How would 32-bit data registers fit into a 16-bit microprocessor?

You would expect a 16-bit microprocessor to process 8- and 16-bit data, but does it make sense for it to also process 32-bit data? Obviously, the addresses will have to be handled in that size. Designers recognized that in 8-bit microprocessors the ability to handle 16-bit data came in quite handy for more advanced applications. The 8-bit processors soon had to be upgraded to handle 16-bit operands, and users of 16-bit minicomputers needed 32-bit operations.

Once a few 32-bit operations become necessary in a microprocessor, you need a whole array of operations. If a multiplication operation generates a 32-bit result, in order to do anything with that result, other 32-bit operations are needed. For consistency, again, Motorola decided that the data registers would be 32 bits wide and operations on all 32 bits could take place with a single instruction.

Three Arithmetic Units

The exact manner of processing data and addresses through the MC68000 came about later, with careful analysis of the internal architecture and the need for address and data in the sequence of instructions. The chip ended up with three separate arithmetic units, which could work in parallel. I'll describe their purpose to give some insight into how the machine works.

The MC68000 has a 16-bit-wide ALU that essentially performs all data calculations and provides single-pass evaluation of the 16-bit data, for which the MC68000 is primarily designed. There are also two other internal arithmetic units. Both are 16 bits wide and are generally used in conjunction with each other to perform the various address calculations associated with operand effective addresses. This makes sense because all addresses are 32 bits wide. An effective address (EA) is the calculated result based on a selected addressing mode of the processor. In the MC6800, for instance, if an "index-register-plus-offset" address mode were used, the EA would be the result of adding the contents of X with the given offset. Because EA evaluation takes time and can be a significant portion of the instruction, it is important to perform this quickly.

At one time, then, one 32-bit address and one 16-bit data calculation can take place within the MC68000. This speeds instruction execution time considerably by processing addresses and data in parallel. The MC68000 also operates on 32-bit data. This is usually done by taking two passes of 16-bit data, one for the lower word and one for the upper word. This is reflected in the execution time of many 16- and 32-bit instructions.

Prefetch Queue

Another way designers made the MACSS faster was to include what is called a *prefetch queue*. This prefetch queue is more intelligent than other microprocessor queues; its control varies according to the instruction stream contents.

The prefetch queue is a very effective means of increasing microprocessor performance; it attempts to have as much instruction information as possible available before a particular instruction begins execution. The microprocessor uses an otherwise idle data bus to prefetch from the instruction stream. This keeps the bus active more of the time, increasing performance because processing of instructions is often limited by the time it takes to get all the relevant information into the processor.

The part of memory from which instructions are fetched, the program space, contains op codes and addressing information. The prefetch queue can contain enough information to execute one instruction, decode the next instruction, and fetch the following instruction from memory—all at the same time.

Exactly what is in the queue is very dependent upon the exact instruction sequences. The queue is intelligent enough to stay fairly full without being too wasteful.

For instance, when a conditional branching instruction is detected, the prefetch is ready to either branch or not by the time a decision is made. The queue tries to fetch both the op code following the branch instruction and the op code at the calculated branch location. Then, when the condition codes are compared and a decision is made whether to branch, the processor can begin immediate decoding of either instruction. The other unnecessary op code is ignored.

You can use the prefetch queue in many other special ways as well. One example is in speeding up the repetitious Move Multiple Registers instruction, where it is used to accelerate successive data transfers. The prefetch queue allows many frequently used instructions to execute in exactly the time it takes to fetch the op code (actually, the time to prefetch the next op code).

Microcoding

One other significant implementation feature from the MACSS project emerged from the choice between a *random logic design* versus a *microcoded design*. Both techniques have advantages and disadvantages. Earlier microprocessors were largely of random logic design. Advanced techniques of very large scale integration (VLSI) and the increasing complexity of the chips have made microcoding more attractive.

Random logic design of a microprocessor or other logic device is the building of the device from discrete components—gates, buffers, and transistors. This limits the components to those that are essential. There are no unused gates, duplicated circuits, or clever uses of otherwise unused components. The design is usually packed as tightly as possible and is quite fast.

The difficulty is that, as the design becomes more and more complex, as VLSI has, the planning and layout of the components and signal traces become exponentially more difficult and often impossibly so. This means that it takes exorbitant amounts of time to design the circuits.

Another problem with the use of random logic in very complex circuits occurs in modeling and testing. Before such circuits are finally placed in silicon, they must be modeled and simulated on computers because of the great difficulty in running down bugs once the chip is in silicon compared to debugging a wire-wrap board. The entire circuit must be modeled all at once to ensure that one combination of signals affects only the expected section of the device.

Similarly, once the circuit is in silicon, the pass/fail testing of the components in a random logic chip is quite difficult. You typically have only a few lines to send sequences of patterns through for testing. Because a particular section of the circuit may be exercised only by a very few given inputs, a normal test may not detect a stuck gate or other error caused by some strange combination of inputs.

On the other hand, in much the same way that microprocessors made designing systems with medium-scale integration/large-scale integration (MSI/LSI) easier, microprogramming has come to ease the complications in the design of microprocessors. Microprogramming is to a microprocessor what a microprocessor is to a logic design of a system. A microprocessor has central components that can be considered black boxes with inputs and outputs. For each given operation (instruction, interrupt condition, etc.), the microprocessor can route certain information to these black boxes as inputs, and the outputs can be routed to other components. The control of this routing is performed by a microcontroller or microsequencer.

Similar to a microprocessor, the microsequencer directs the flow of data through the various components (ALU, registers, condition flags, shifters, buses, etc.) according to microprogrammed instructions. Each instruction has its own microroutine, or sequence of microwords, which routes the associated data to the proper component in the proper order. Conditions and branches may redirect the microroutines.

Microcoding a complex circuit simplifies design mostly because it makes the circuit modular. It takes a controller, a block of microprogram, and the components through which data is to flow. Each of these elements may be modeled, built, and tested with individual inputs and outputs. Microcoding is a big step toward simplifying the design process because it breaks up the design into manageable blocks, thereby easing the testing of the finished product.

Another advantage of microcoding is that it allows tremendous flexibility in the exact operation of the circuit. Its microwords allow more combinations of the inputs through the components than most random logic would allow. Microcoding's programmability makes it especially attractive to silicon designers because random logic in silicon is not easily changed.

Last-Minute Changes Possible

You can change the microROM of the microcoded device right up to the minute before the masks for the device are processed. To change a small facet of an operation may mean altering a few bits in the microROM, but this changes only whether or not there is a gate on the bit's transistor—a simple alteration. Similarly, after the silicon is cast, should a change be necessary, it will likely be just a microcode change, which would be much easier than random logic modification in silicon.

The disadvantage of a microcoded circuit lies primarily in its generality. Because it is made up of modules and is programmed, the microcoded circuit is more wasteful of transistors and therefore makes a larger circuit. This may add up to 20 percent more board space or chip area than a tight random logic design. But microcoding has advantages that make up for this disadvantage, making it the design choice for modern VLSI circuits.

There are two types of microprogramming, horizontal and vertical (see figure 4). *Horizontal microcoding* is the more direct form. It is unencoded, so that, for instance, 1 bit in each microword would enable each register. For 16 registers, then, 16 bits of microcode must be dedicated. Horizontal microwords tend to be quite long, and because the size of the microcode directly affects chip size, they can quickly increase chip cost.

A denser but slower form of microcoding is *vertical microcoding*. Here, control functions are encoded, so that only 4 bits of microcode are required to select one of 16 registers. While it needs a much shorter microword, vertical microprogramming is

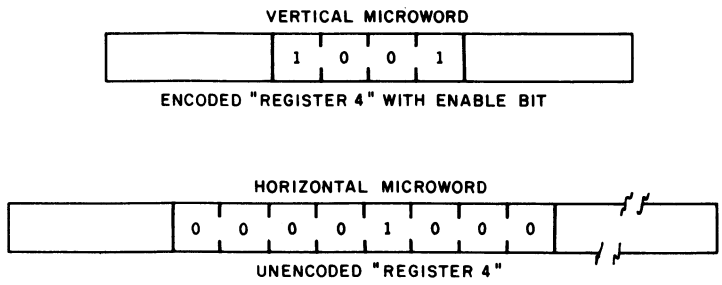


Figure 4: *Comparison of horizontal and vertical microcode patterns.*

potentially slower than horizontal microprogramming. Vertical microprogramming will take at least one level of logic gates to decode the encoded signals. This level of gates may just throw the total gate propagation delay over the threshold of the clock pickets, forcing an additional clock cycle into the instruction.

In the MACSS project, the MC68000 was selected to be microcoded. In retrospect this was a very wise decision. The first silicon prototype worked well enough so that the major circuits in the device could be tested, and subsequent "fixes" were often just microcode corrections. The instruction set was not firm until just before the masks went to wafer fabrication, allowing some late decisions to be made to improve the performance of the chip.

A combination of horizontal and vertical microcoding was used on the MC68000 to gain the optimum advantages of both. Essentially, a microcode and a nanocode were developed. The microcode is a series of pointers into assorted microsubroutines in the nanocode. The nanocode performs the actual routing and selecting of registers and functions, and directs results. This combination is quite efficient because a great deal of code can share many common routines and yet retain the individuality required of different instructions.

Decoding of an instruction's op code generates starting addresses in the microcode for the type of operation and the addressing mode. Completion of an instruction enables interrupts to be accepted or allows access to the prefetch queue for the next op code. The prefetch queue actually keeps bus use at 85 to 95 percent, i.e., the bus is idle only 5 to 15 percent of the time!

Conclusion

Let's look back now at the MC68000 and see what parts of it might qualify it as a 16-bit device. The internal data ALU is 16 bits. It processes 32-bit addresses, though only 24 bits are brought off chip. The op code that tells the processor what operation to perform is 16 bits wide. The data bus is 16 bits wide. The microprocessor will operate on either 8, 16, or 32 bits of data automatically. There are 16 general-purpose 32-bit-wide registers in the chip.

The MC68000 is generally considered a 16-bit microprocessor, though it uses 32-bit addresses and contains 32-bit registers. It also can operate on 32 bits of data as easily as 8 and 16. Many users of the MC68000 consider it a 32-bit just as much as a 16-bit processor. Whatever you consider it there is no doubt that the MC68000 is indeed a powerful microprocessor. In coming articles, I will discuss in more detail exactly what operations are available in the MC68000 and will illustrate examples of MC68000 code.■

Design Philosophy Behind Motorola's MC68000

Part 2: Data-movement, arithmetic, and logic instructions

Thomas W. Starnes
Motorola Inc., Microprocessor Division
3501 Ed Bluestein Blvd.
Austin, TX 78721

Last month, in part 1, I discussed the design philosophy behind the Motorola MC68000, a powerful 16-bit processor with multiple 32-bit registers. This month I'll describe the data-movement, arithmetic, and logic instructions of the MC68000. A thorough reading of the MC68000's user's guide (available from many computer bookstores and Motorola distributors) will give you all the details of each instruction's operation, but a look at the general categories of instructions, a discussion of why certain design decisions were made, and mention of some special capabilities of the instructions will give you insight into the power of this instruction set.

Instruction Format and Addressing Modes

Before I get into the instruction groups, let's first look at how assembly-language instructions are written. Table 1 illustrates a common instruction format and the choices that can be made within it. First, of course, you can pick one of several microprocessor instructions—for example, an addition (ADD), comparison (CMP), arithmetic shift left (ASL), or data move (MOV). If the instruction is one that handles data, you can, with the MC68000, select one of three data sizes: 8, 16, or 32 bits. This selection is made by following the mnemonic with a period and either a "B", "W", or "L", for byte, word, or long word; if no size is specified, the assembler will assume a 16-bit operation.

On a data operation, you need to make one or two more decisions, i.e., which addressing mode to use for the one or two operands the instruction requires. (See the text box on data organization on page 354 for more details.) Typically, you can select one of 14 modes; within most of these modes, one of eight address registers is selected. On many operations, you need to select a second addressing mode; this usually involves selection of one of eight data registers, but for the data-movement instruction, any addressing mode can be selected.

All MC68000 instructions are fully defined with 16 bits of *op code*. (Op code is short for operation code; it is the pattern of bits that a microprocessor interprets as a specific machine-language instruction executable by it.) Depending on the instruction or the addressing mode(s) selected, additional 16-bit extension words may follow the op code. These extension words provide additional addressing information and may make the total instruction length as long as 10 bytes. Because the instruction is always lengthened by multiples of 16 bits, you can ensure that instructions always begin on even-byte boundaries; because of the way the MC68000 fetches 16-bit quantities from memory, this placement of instructions increases the speed of program execution.

By far, the most common operation in any processor application is the movement of data. Other microprocessors move data with LOAD, STORE, PUSH, PULL, POP, and input/output (I/O) instructions. When you boil it all down, each instruction simply moves data from one location to another. So why not call them all MOVE? Simplicity of expression is a fundamental theme throughout the MC68000's instruction set: all similar operations should perform similarly in a number of respects. For example, if you can use an ADD operation with two 32-bit quantities, you should be able to use an add-with-carry operation with two 32-bit quantities. If you can select from 14 addressing modes to use an ADD operation, you should be able to select from 14 addressing

About the Author

Thomas Starnes is an electrical engineer who has spent the last five years helping to plan the direction of the MC68000 family of processor products for Motorola.

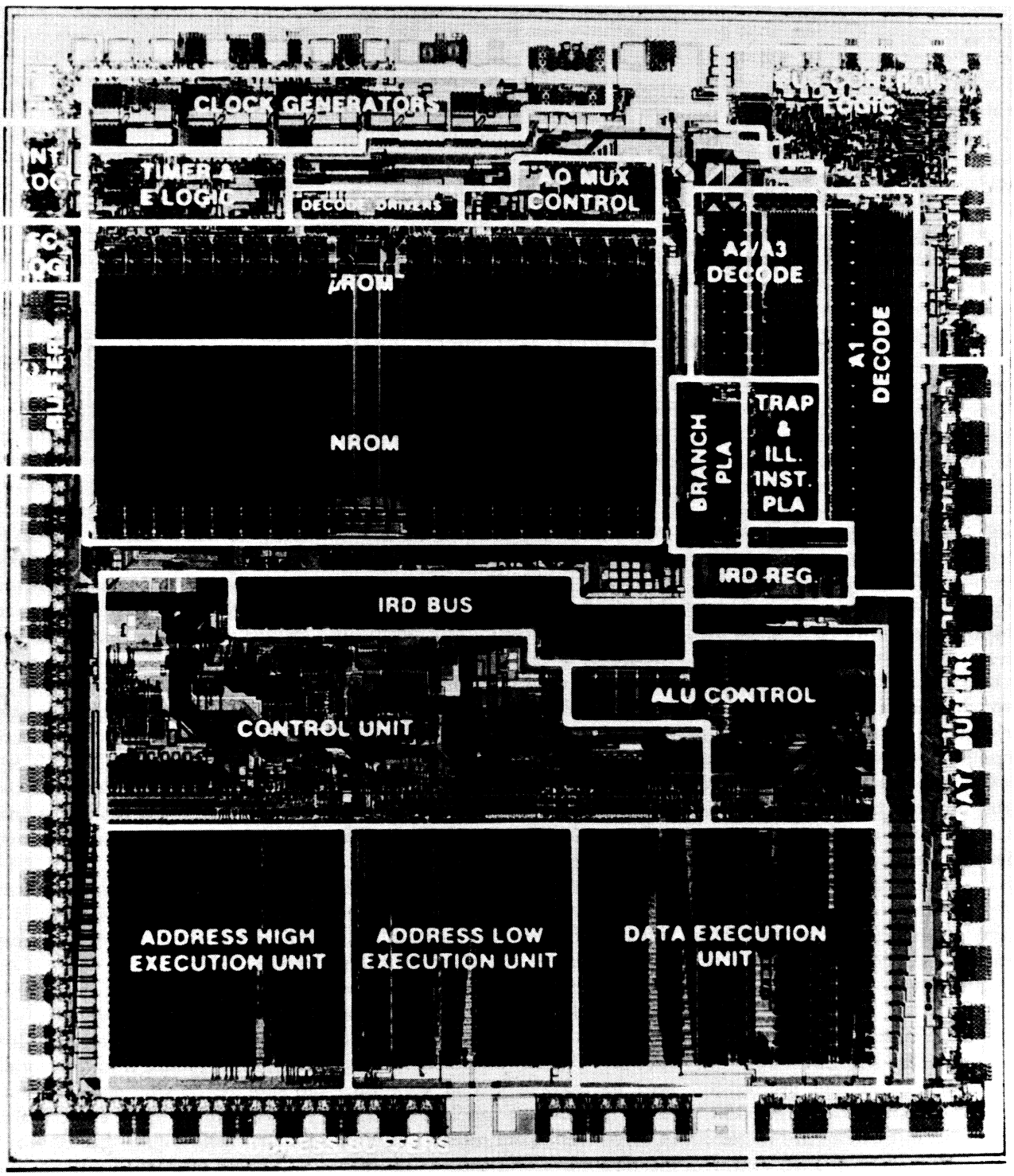


Photo 1: *The MC68000 microprocessor chip, which contains more than 68,000 transistors, is 246 by 281 mils (6.24 by 7.14 mm) in size. This photo shows the location of the major functions of the chip. "Int. Log." stands for "Interrupt Logic"; "A0 Mux Control", for "Microcode A0 Multiplexer Control"; and "FC Log.", for "Function Code Logic." The labels "μROM" and "NROM" indicate two areas of microcode. "Trap and Ill. Inst. PLA" stands for "Trap and Illegal Instruction Programmable Logic Array"; "IRD Reg.", for "Instruction Register Decode Register"; and "ALU Control", for "Arithmetic and Logic Unit Control." The Data Execution Unit houses the main functions of the arithmetic and logic unit, while the two Address Execution Units perform the arithmetic associated with the calculation of an address.*

Instruction format is:

mnemonic.size source,destination

Examples:

```
ADD.L    D1,D2
MOVE.B   #15,-1(A0)
ADD      D1,D2          (size assumed to be ".W")
BGE      LOOP1          (only one argument)
RTS                     (no arguments)
```

Explanations:

mnemonic = instruction abbreviation (ADD, CMP, MULS, etc.)
size (optional) = operand size:
.B means byte data (8 bits)
.W means word data (16 bits; default size)
.L means long word data (32 bits)
source (optional) = source operand addressing mode
destination (optional) = destination operand addressing mode

Table 1: *General format for MC68000 instructions.*

modes to use SUB (subtract). If certain status register codes are modified to reflect the results of an ADD, the same codes should also be modified when a SUB or NEG (negate) instruction is performed.

Varieties of MOVEs

With this philosophy in mind, all of the old LOAD, STORE, PUSH, PULL, POP, and I/O instructions from other microprocessors were rolled into one very powerful and flexible MOVE instruction in the MC68000. Let's look at just what this one instruction can do.

The MOVE instruction can move 8-, 16-, or 32-bit data from practically any location to practically any other. And a wide selection of addressing modes and registers for both the source and the destination should cover about any way you want to find the operands. Table 2 lists the different combinations of addressing modes available on both the MC68000 and the Intel 8086 families. Let's look at what some of the MC68000 addressing-mode combinations allow you to do.

Certainly, you can copy data between registers, but you can also copy data to or from a register to memory using any of the memory-addressing modes. Most microprocessors allow the programmer to transfer data on the top of a stack to or from a register only. What if the data is really needed elsewhere in memory? You must run a second instruction to make the second move and use a register for a temporary holding space. The MC68000 allows you to move top-of-stack data to or from any register, another stack, a queue, any memory location, or any I/O location, all in one smooth motion. And why shouldn't you be able to? An added advantage of the MC68000 comes from its ability to use any one of the eight address registers as a stack pointer; this allows you to build as many as eight different stacks without having to swap out registers.

You can also do direct memory-to-memory moves. There are 10 different

	destination	Dn	An	(An)	(An)+	−(An)	d16(An)	d8(An,Xn)	Abs.W	Abs.L
	#options	8	8	8	8	8	8	128	N/A	N/A
source	#options									
Dn	8	MI	MI	MI	M	M	MI	MI	MI	M
An	8	MI	MI	MI	M	M	MI	MI	MI	M
(An)	8	MI	MI	MI	M	M	M	M	M	M
(An)+	8	M	M	M	M	M	M	M	M	M
−(An)	8	M	M	M	M	M	M	M	M	M
d16(An)	8	MI	MI	M	M	M	M	M	M	M
d8(An,Xn	128	MI	MI	M	M	M	M	M	M	M
Abs.W	N/A	MI	MI	M	M	M	M	M	M	M
Abs.L	N/A	M	M	M	M	M	M	M	M	M
d16 (PC)	1	M	M	M	M	M	M	M	M	M
d8 (PC,Xn)	16	M	M	M	M	M	M	M	M	M
Immediate	1	M	M	M	M	M	M	M	M	M

Notes:
1. "M" means this combination available on the Motorola MC68000.
"I" means a comparable combination available on the Intel 8086 family.
2. "#options" refers to the number of different ways an addressing mode can be used in the MC68000 due to the availability of multiple registers that can be used; for example, the "d8(An,Xn)" option can use 8 An registers and 16 Xn registers for a total of 128 combinations. "N/A" means "not applicable."
3. Most of the source and destination addressing modes are explained in the text box "Data Organization and Addressing Modes," page 354. "An" and "Dn" are register addressing modes. "Abs.W" and "Abs.L" are word-address and long-word-address forms of absolute addressing.

Table 2: *Addressing modes available to the Motorola MC68000 and the Intel 8086. The information at the intersection of a row and a column indicates the availability of that source/destination addressing-mode combination for each microprocessor.*

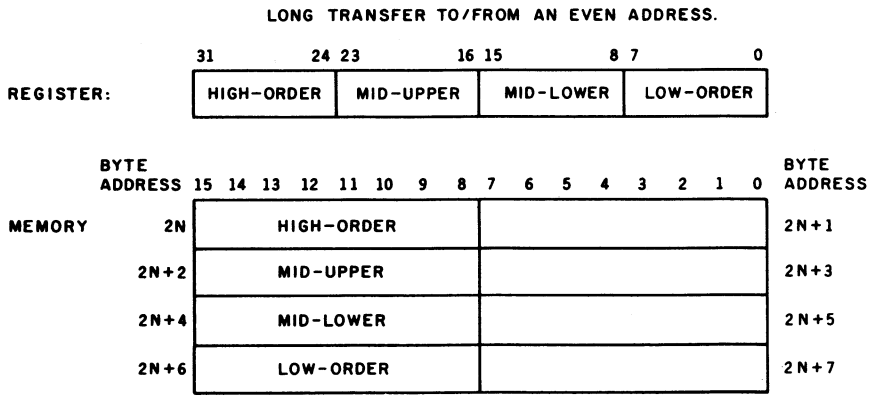


Figure 1: *Moving data from a 32-bit register to memory using the MOVEP instruction. Bytes from the register are stored in every other memory byte. The instruction takes 24 clock cycles to execute.*

memory-addressing modes to select from for the source operand and seven for the destination. Also, keep in mind that each addressing mode can use any of the eight address registers, further increasing the versatility of these move instructions. (Actually, in one mode, you have 16 registers to choose from—eight address and eight data registers.)

Just how many different ways are there to move general data in the MC68000? When you couple all of the combinations allowed with the selection of registers available, there are 34,888 different ways, and each one can be used for 8-, 16-, or 32-bit data. That ought to solve most programmers' data-shuffling problems!

The remaining data-movement instructions include SWAP, for instance, which exchanges the contents of any two data and/or address registers. You can read or modify the status register codes with a MOVE SR (Status Register) instruction.

The MC68000 was designed to interface directly to the MC6800 line of 8-bit peripherals so that all the existing peripheral circuits could easily be used on the MC68000. (Many of the 8-bit peripherals provide very useful functions that would need to be included in a 16-bit system.) To bring the best of the 8-bit peripheral world into the universe of 16-bit software, designers included a special MOVEP (Move Peripheral) instruction in the MC68000. Here's how and why it works.

Frequently, you must set up registers to ready a peripheral for operation. You need to connect an 8-bit peripheral to either the upper or lower half of the 16-bit-wide data bus. This means that the registers connected to a peripheral appear within the MC68000 memory address space as successive-even or successive-odd addresses. The MOVEP instruction will move either 16 or 32 bits of a given data register out to memory in 8-bit chunks, starting at a given location (see figure 1); addresses for each successive byte are incremented by two, not by the one that the normal MOVE uses. This allows the 2 or 4 bytes being transferred to be loaded into the proper peripheral port addresses. Thus, you can load as many as four 8-bit registers in one simple instruction. The MOVEP instruction is bidirectional, so that the registers can be either loaded or read.

Two special types of the MOVE instruction are the MOVEQ (Move Quick) and the MOVEM (Move Multiple Register). Often a register is used as a counter or a constant, with values that are typically rather small. The MOVEQ instruction makes it fast and easy to initialize a register to such values. MOVEQ will take any signed 8-bit immediate value between −128 and 127, extend its sign bit so that it will be correctly interpreted as a 32-bit number, and load it into one of the data registers. The op code for MOVEQ includes the 8-bit immediate value; this means the microprocessor can perform the operation very quickly. Because the small immediate value is part of the MOVEQ op code itself, the instruction is classified as a separate addressing mode of the MC68000 called the "quick immediate" addressing mode.

It is common in machine-language programming to have to save the contents of various on-chip registers, use the registers for some other purpose, and then restore their former contents. This happens when you are beginning or ending a subroutine, executing an interrupt handler, changing tasks, or calling the operating system. The MC68000 has a very handy instruction that makes this a fast, efficient operation. The MOVEM instruction will take any combination (or all) of the 16 data and address registers and move them either to or from memory in an organized manner. These registers can be transferred to or from any stack or to a specific location in memory. They are put in memory and taken from memory in reverse order to ensure that each register receives its proper contents. An option of the MOVEM instruction is that either the lower 16 bits of the registers or the entire 32-bit registers can be transferred. An example of this instruction is:

```
MOVEM.L D0/D4-D7/A4/A5,40(A6)
```

which would save the registers as shown in figure 2. (The instruction will save registers D0, D4 through D7, A4, and A5 into memory starting at the location pointed to by the value in register A6 plus the value 28 hexadecimal.) The list of registers to be transferred is compactly encoded in a 16-bit value that follows the MOVEM op-code word—an "on" bit indicates the associated register is to be transferred. Not only is the MOVEM instruction both compact and useful, it is also as fast as possible for the number of bytes of information that must be transferred.

Orthogonality

Arithmetic operations are key instructions in a microprocessor because they tend to be the ones that

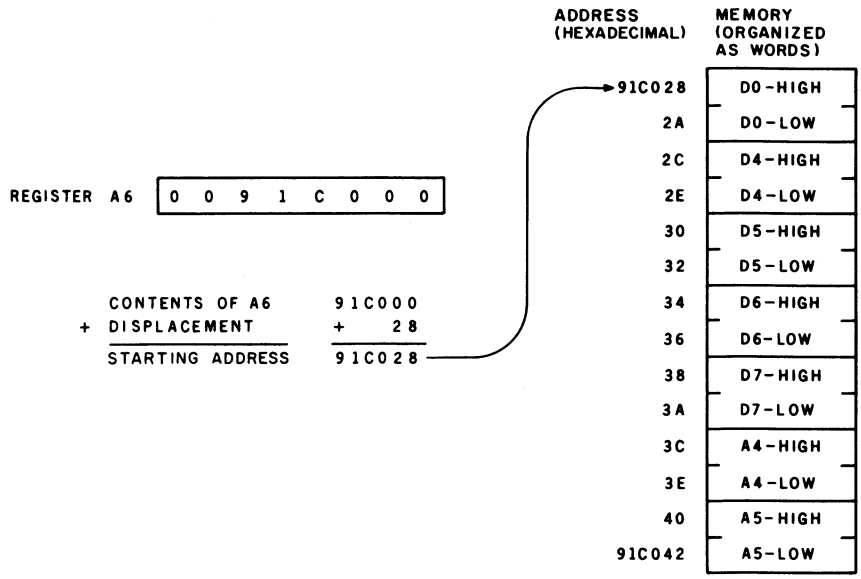


Figure 2: *Pushing multiple registers to memory with the MOVEM instruction. This instruction offers a very fast way to store selected registers in memory and, later, restore them properly. This figure shows the storage of registers D0, D4 through D7, A4, and A5 to location 91C028 hexadecimal. The instruction itself is MOVEM.L D0/D4-D7/A4/A5,40(A6). (Remember that 40 decimal equals 28 hexadecimal.) The instruction takes 58 clock cycles to execute.*

do the bulk of the work. The arithmetic and logic instructions allow programmers to write code exactly as they desire without having to rearrange data, gather more data, or do things in an unnatural order. As with so many other characteristics of the MC68000, the design of the arithmetic and logic instructions is very orthogonal—more so than for any previous microprocessor. *Orthogonality* can be defined as the ability of any allowed operation to use any resource in any way that any other operation may.

The arithmetic and logic instructions are very similar in the way they function, the way the condition codes are affected as a result, and the selection of addressing modes, registers, and operands available to them. The advantage of this is that, when coding, the programmer has only one uniform set of rules to remember. Older microprocessor designs forced programmers to have different sets of rules for even similar instructions, which decreased programmers' productivity by forcing them to recall and use correctly large amounts of essentially arbitrary information.

All of the dual-operand arithmetic instructions are true "one-and-a-half" address operations (i.e., one operand can be specified as a memory address, but the other must be an internal register—the result overwrites one operand). Thus, you can add any register to any register, a constant to any register, the top of a stack to any register, a value buried in a stack to any register, a table entry to any register, an input from an I/O device to any register, or any memory location to any register. Or, because the order may be reversed, you can add any register to any of the same (except that you can't add to a constant, and you can't use the program-counter relative addressing mode to specify a destination). Also, remember that any of these instructions can occur with 8-, 16-, or 32-bit data.

Arithmetic Instructions

Let's look at the types of arithmetic instructions that are available. Add (ADD), subtract (SUB), and compare (CMP) instructions are general two-operand instructions. ADDX and SUBX are used to work on numbers longer than 32 bits (the X condition bit in the MC68000 performs a function similar to the one the carry bit performs in most microprocessors). Two multiply and divide instructions are available: signed (MULS and DIVS) for single-precision instructions and unsigned (MULU and DIVU) for multiple-precision instructions.

The negate (NEG) and clear (CLR) instructions require only a single operand, and you can use a NEGX to negate multiple-precision values. To blend mixed sizes of data, the MC68000 provides a sign-extend instruction (EXT), while a TST (test) instruction is used to check for positive, negative, or zero conditions. A special instruction, the indivisible test-and-set (TAS), provides software synchronization in multimicroprocessor operations.

One variation on the ADD instruction enables the MC68000 to overcome a common limitation of other microprocessors. The normal "one-and-a-half" address design of most processors makes it difficult to use constant (immediate) values with anything but registers. The MC68000 overcomes this with the ADDI instruction, which allows a byte, word, or long-word immediate value to also be added to an operand in memory using any legal destination-memory addressing mode.

The MC68000 has no increment or decrement instructions. Why? Remember, the idea is to treat all similar instructions the same. An increment instruction adds 1 to a quantity and is often used to step to the next element in a table of byte-wide values. But the MC68000 programmer will often be manipulating 16- and 32-bit data, which require increments of 2 and 4, respectively, to the table address. The design team wanted to generalize the increment and decrement instructions to make them useful with all data sizes yet still retain the speed associated with an instruction that does not have to fetch an immediate argument. To solve these problems, and then some, the designers gave the MC68000 "add quick" (ADDQ) and "subtract quick" (SUBQ) instructions, which allow any number from 1 to 8 to be added to or subtracted from any register or any memory location. The instructions accomplish this in the shortest possible time by using 3 bits within the 16-bit op code to hold the increment or decrement

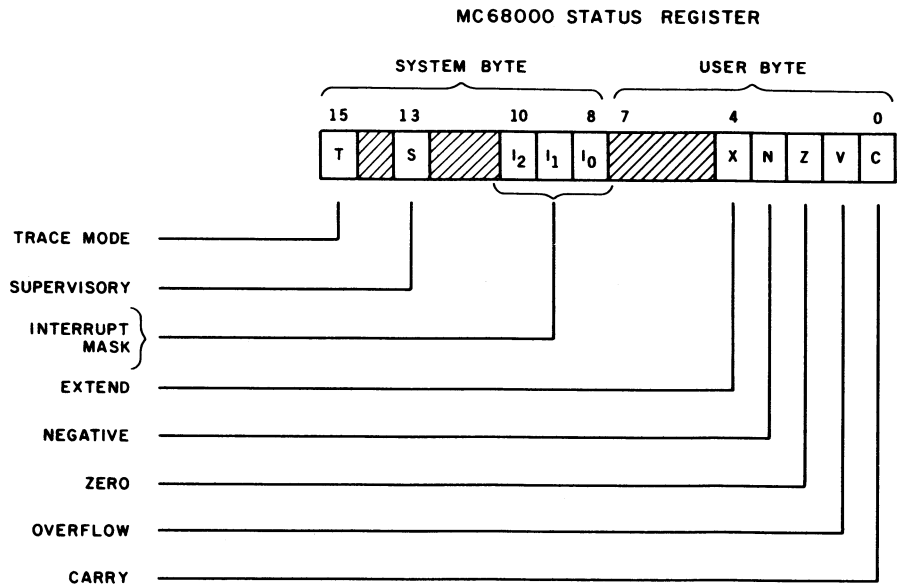


Figure 3: *The bits of the MC68000 status register (SR).*

amount (in this scheme, the bits 000 indicate an operand value of 8, not 0). Then, not only can you quickly and easily change an address pointer by 1, 2, or 4 for 8-, 16-, or 32-bit data, but you can also change counters by 3, 5, 6, 7, or 8. And the effect is identical to that using the standard ADDI instruction—even the status register codes are all the same.

Some odds and ends of arithmetic instructions include sign-extend (EXT), clear (CLR), and test (TST) instructions. Since three different sizes of data can be used in the MC68000, there should be a convenient way of changing size. If you want to move only part of a datum (for example, the bottom 16 bits of a 32-bit register), you need only use a MOVE instruction of the proper data size. If, however, you want to convert a datum to a larger-sized two's-complement value (for example, making a 16-bit value into a 32-bit expression), you need a special instruction. The EXT instruction will take an 8-bit or a 16-bit datum and duplicate its uppermost bit position through the higher portions of any data register in order to convert the datum to 16 or 32 bits wide, respectively. CLR simply loads a set of 0s into the destination. TST sets the negative and zero condition bits (discussed in the next section) according to the nature of the given operand.

Status Register Codes and Multiple-Precision Arithmetic

What if you are dealing with binary integers that require more than 32 bits for expression? Say you want to add two 128-bit (16-byte) numbers. If both of these numbers were in the MC68000 registers, all eight data registers would be in use. More likely, the two values would be in 16 consecutive bytes of memory, starting with the most significant byte of data. The normal procedure to add two such numbers is to add the two least significant bytes, remember the carry, go to the previous bytes and add them, remember the carry, and so on. This sequence of operations is handled neatly in the MC68000 by the predecrement address-register deferred mode, which uses the notation "−(An)". Use two address registers to point to the byte just past each operand. Each execution of an ADDX −(Am), −(An) instruction will decrement the values in the Am and An registers (m and n stand for numbers between 0 and 7), then add the two numbers pointed to by those registers. By putting this single instruction in a loop, you can quickly create the code needed to operate on multiple-precision numbers.

Let's detour for a second to discuss the status register of the M68000 (see figure 3). It contains the standard carry (C), overflow (V), zero (Z), and negative (N) bits found in other microprocessors. It also has a status-register bit not found on other microprocessors, the X (or extend) bit. This bit was created to eliminate confusion caused by traditional overuse of the carry bit.

To explain the extend bit, I should describe the carry bit. In most microprocessors, the carry bit is overused. It is changed by (among other things) an addition instruction, but it is used in two different ways. Sometimes it is used in a later addition, such as in multiple-precision additions; sometimes a program tests the bit and branches according to the carry bit's state. So programmers use the carry bit for two different purposes: for extended-precision arithmetic and for program control.

The MC68000 has a bit for each purpose. Both the carry and the extend bits are changed according to the results of an addition instruction. However, the carry bit is used by the microprocessor during testing for program control purposes, while the extend bit is used as an input for multiple-precision arithmetic operations. For ADD, SUB, NEG, and specified shift and rotate instructions, both the carry and extend bits are updated. Other instructions—MOVE, AND, OR, TST, CLR, MUL, and DIV—change only the carry bit. This design helps prevent inadvertent changes to either bit.

Because of the extend bit, the familiar "add with carry" operation in the MC68000 becomes ADDX or "add with extend bit." Look at why this is important. Once you start a multiple-precision arithmetic operation and get a partial result, the integrity of the extend bit will be maintained even if you have to suspend the addition to do some data movement with the MOVE instruction. Programming becomes easier because you don't have to save the status register codes when interrupting a multiple-precision operation.

I should mention one other thing about multiple-precision arithmetic. When you have finished the multiple-precision operation, what does the negative bit mean? It correctly indicates that the result was positive or

Data Organization and Addressing Modes

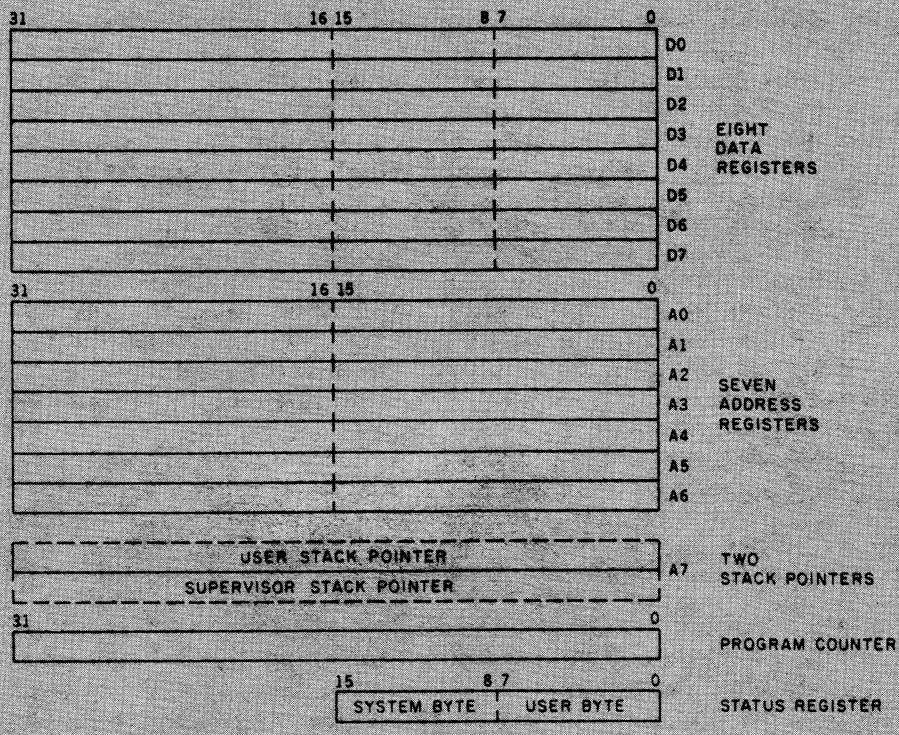
Motorola designed the MC68000 to offer a versatile set of data sizes and addressing modes for the assembly-language programmer. To understand fully the power of this machine, you must first understand how the MC68000 organizes and moves data.

Although the MC68000 "sees" its memory space as a collection of 8-bit bytes, it works on several different data sizes. It can address memory in the following sizes: byte (8 bits), word (16 bits), and long word (32 bits). In addition, it can manipulate individual bits—a bit is specified by the byte it is in and its bit number (between 0 and 7) within the byte.

The MC68000 has a definite preference for addressing 16- and 32-bit quantities that start at even addresses. In particular, it was designed to access quickly 16-bit quantities that start on an even address. Even though this speed comes at the expense of words and long words that begin at odd addresses (which would take two, not one, fetch operations to be accessed), this is not a serious disadvantage. MC68000 op codes are always 16 bits wide, and any arguments the MC68000 requires are always stored as one or more 16-bit words (even if the argument is only a byte quantity). Because of this, code that starts on an even-byte boundary will stay on an even-byte boundary and thus will be accessed at the fastest rate possible.

The MC68000 register set (see the figure) indicates the microprocessor's commitment to long-word data: even though this is a 16-bit machine, all the internal registers are 32 bits wide. The dotted lines in the data registers denote those registers' ability to handle 8-, 16-, or 32-bit-wide data. The address registers can handle either 16- or 32-bit-wide data.

The MC68000 supports six major addressing modes; although I will not go into the variations available in each mode, a short description of each will show you the basic ways the microprocessor can get its operands.



- *Inherent addressing: the instruction itself tells the microprocessor where to get its operand. An example of this is RTS, return from subroutine, which gets the return address from the stack.*
- *Register addressing: the operand is a data or address register. An example is MOVE D1,D2, which moves the contents of data register D1 to data register D2.*
- *Immediate addressing: the operand is specified as part of the instruction. An example is MOVE #3,D2, which moves the value 3 into data register D2.*
- *Absolute addressing: the operand is specified by a 16- or 32-bit address that is appended to the instruction. An example is MOVE $3F01,D2, which*

negative. In most microprocessors, though, the zero bit indicates only that the most significant portion of the result is 0, not that the entire result is 0. The multiple-precision arithmetic instructions in the MC68000 are designed so that the zero bit will accurately depict the status of the entire result. This is done by allowing multiple-precision instructions to reset the zero bit (denoting a nonzero result) but not to set it (denoting a zero result). With this scheme, the programmer's only responsibility is to set the zero bit before beginning the multiple-precision operation.

One final issue can come up in the middle of arithmetic operations, and the MC68000's handling of the problem illustrates another fundamental difference between it and so many other microprocessors. How many times have you interrupted a series of arithmetic operations to modify some memory pointers and later discovered that your completed arithmetic operation gave a wrong result because you inadvertently modified certain status register code bits? When you get right down to it, when you add 12 to a memory address, who cares if a carry was generated or if the result was negative? In fact, the negative bit has no meaning in relation to addresses. We as programmers are hurt

moves the contents of the word at address 3F01 hexadecimal into data register D2. This is also called direct addressing *because the operand address is being directly supplied.*

- *Register-deferred addressing: this mode has a lot of variations. Since these modes get rather complicated, I'll speak of them in terms of what the* effective address *is—that is, what memory location will be used to supply the source or destination operand. The simple* address-register deferred mode *makes not the register but the contents of the register the effective address. Because the register contents is itself the address of the given operand, this addressing mode is often called* indirect addressing*—the register indirectly supplies the effective address. Suppose that register A1 contains the value 100 and that the word at address 100 contains 12D hexadecimal. Then the instruction MOVE (A1),D2 will put the value 12D hexadecimal into data register D2. The notation "(An)" denotes address-register-deferred (or indirect) addressing using data register An.*

Two related variations on address register deferred are called address-register deferred with predecrement *and* address-register deferred with postincrement*; they have notations of "−(An)" and "(An)+", respectively. In the predecrement variation, the address register An is decremented before its contents is used as the effective address; in the postincrement variation, the contents of the address register is used as the effective address, then the register is incremented. These modes are often used in loops for repetitive operations on sequential areas of memory; they usually replace an explicit increment or decrement instruction, thus providing more compact, faster code. The incremental or decremental amount will be 1, 2, or 4 depending on whether the operand is a byte, word, or long word.*

Another variation is called address-register deferred with displacement*; in this mode (denoted "d16(An)"), the effective address is the contents of address register An plus the signed 16-bit value d16. Suppose address register A1 contains the value C100 hexadecimal and the word at address C25E hexadecimal contains 0. Then the instruction MOVE $15E(A1),D2 moves the value 0 into data register D2. (The effective address, C25E hexadecimal, is the sum of the displacement 15E hexadecimal and the contents of register A1, C100 hexadecimal.)*

The variation called address-register deferred with index and displacement *(denoted "d8(An,Xn)") limits the displacement to an 8-bit signed number but makes the effective address the sum of three numbers: the displacement, the contents of the address register An, and the contents of an index register Xn, where the index register can be any of the address or data registers. The extra register is added into the effective address calculation to allow the same instruction to point to the base of a table (using d8 and An) and, during execution, to refer to different nearby memory locations by putting different values into the index register Xn. The word "index" in the addressing mode name refers to the commonplace use of the index register to index into an array of numbers stored sequentially; in such a case, the value in the index register equals the subscript of the array element desired. Given the example directly above (register A1 contains C100 hexadecimal, word C25E hexadecimal contains 0) and the information that data register D6 contains is the value 100 hexadecimal, the instruction MOVE $5E(A1,D6),D2 moves the same value (0) to data register D2 just as the instruction MOVE $15E(A1),D2 would. In both cases, the effective address is C25E hexadecimal.*

- *Program-counter relative addressing: this mode has two variations, "d16 (PC)" and "d8 (PC, Xn)"; these are similar to the two forms of address-register deferred addressing described above. It is different in two ways: first, it uses the program counter (PC) instead of an address register; second, you cannot use this mode to specify a destination operand. The main advantage of this mode is that it allows you to write position-independent code. Because the program counter contains the address of the next instruction after the one currently executing, its use allows the current instruction to refer to data (or program locations for branches) relative to the instruction itself. The MC68000 designers included the restriction against using this mode for a destination operand to protect a program with errors from inadvertently destroying itself. In addition, this restriction prevents programmers from writing self-modifying code, a dangerous practice that programmers occasionally try to increase program performance.*

by the senseless changing of status register code bits when address-related operations are run. Why don't we leave these bits alone when changing memory addresses?

As you can imagine, the designers of the MC68000 have addressed this problem. One of the primary distinctions between the data and address registers in the MC68000 is that instructions with an address register as the destination do *not* modify the status register code bits. They are not changed by moving a new pointer value into an address register, incrementing or decrementing an address register, or by adding any value to an address register. This means that you should never run into a problem with memory-pointer modifications affecting your ongoing data arithmetic operations.

Another interesting note is that all operations to any address register affect the entire address register. Because all MC68000 addresses are 32 bits wide, any operations with an address register as destination must perform in a way that keeps the result valid as a 32-bit address. One solution, to require all inputs to address-register operations to be full 32-bit quantities, would be wasteful of memory space. So either word (16-bit) or long-word (32-bit) operations may take place in any of the ad-

Instruction	Operation
ADD.B D6,D2	adds the lower 8 bits of D6 to D2 (takes 4 clock cycles)
ADD.L 52(A1,D7.W),D6	the effective address is the sum of the constant 52, the contents of register A1, and the lower 16 bits of register D7; the long word at the effective address is added to the contents of register D6 (20 clock cycles)
ADD.W D3,(A7)	adds the lower 16 bits of D3 to the element on top of stack pointed to by A7 (12 clock cycles)
ADDI.L #$400,D1	adds 400 hexadecimal to the 32-bit contents of D1 (16 clock cycles)
ADDI.B #$A9,$30B(A6)	the effective destination address is the sum of the 30B hexadecimal and the contents of register A6; A9 hexadecimal is added to the byte at the effective address (20 clock cycles)
ADDA.W −(A5),A2	decrement register A5 by 2, then add the word pointed to by register A5 to register A2 (14 clock cycles)
ADDA.W #100,A5	add the value 100 to the contents of register A5 (12 clock cycles)
ADDQ.W #1,(A4)+	add 1 to the word pointed to by register A4, then increment register A4 by 2 (12 clock cycles)
ADDQ.B #3,D7	add 3 to the contents of register D7 (4 clock cycles)
ADDX.L −(A2),−(A5)	after decrementing both registers A2 and A5 by 4, add together the X bit and the two long words pointed to by A2 and A5 (30 clock cycles)

Table 3: *Examples of MC68000 addition instructions. The clock times given are worst-case times for the instruction.*

dress registers A0 through A7. If a word operation is performed, the 16-bit quantity is first sign-extended to 32 bits before it is used.

Processor Speed

How fast does the MC68000 execute instructions? Because of the consistency of the microprocessor, the answer for addition instructions will serve as a guide for all arithmetic and logic instructions. A prefetching mechanism in the MC68000 keeps decoded instructions waiting to be executed. So while the timing information given refers only to the time it takes to pass through the adder, recall that the prefetcher will have fetched the next op code while the current op code is being executed.

The minimum time it takes the MC68000 microprocessor to access memory (to read or write) is 4 clock cycles. With a clock frequency of 8 MHz (the frequency used in the standard MC68000 microprocessor), this bus cycle will take 500 ns (nanoseconds). (All subsequent timings will be given in clock cycles, which is a meaningful measurement for all the MC68000-family microprocessors, regardless of the speed of their system clocks—8, 10, or 12.5 MHz.) Every instruction will take at least 4 clock cycles to complete because this is the time it takes to fetch the next op code.

The MC68000 has only one 16-bit arithmetic and logic unit (ALU) for data operations. Therefore, 8- or 16-bit operations can be performed in a single pass through this unit; this takes 4 clock cycles. A 32-bit operation will require a second pass. Memory-addressing modes increase the time needed for an operation because the microprocessor requires more time to calculate the addresses, and a bus cycle is required for each 16 bits of addressing information or actual data that needs to be transferred. An indexed addressing mode, or anything with a displacement, for instance, will require 1 additional bus cycle for the address extension word and another to get the data (2 if the data is a long word); add about 8 more clock cycles (12 if the data is a long word) to the execution time of a given instruction that uses this mode. Some sample worst-case clock timings for various addition instructions are given in table 3.

Like the ADD instruction, other MC68000 arithmetic instructions come in several forms. The subtract instructions have forms analogous to the add instructions—SUB, SUBA, SUBI, SUBQ, and SUBX. Instructions for compare operations that are all similar (CMP, CMPA, CMPI) perform the subtractions without storing a result (the net effect is to set the appropriate status register bits). A memory-compare instruction (CMPM) allows two strings of binary integers in memory to be compared by sequencing through them to higher memory. Two versions of the single-operand negate instruction, NEG and NEGX, ignore and include, respectively, the state of the X bit.

Multiplication and Division

Two versions of multiply and divide instructions make fast work of more complex arithmetic. The two versions are unsigned (MULU and DIVU) and signed (MULS and DIVS) instructions; these versions interpret their operands as one's-complement and two's-complement numbers, respectively. All of these instructions can include immediate values as the multiplier or divisor so that variables can be operated on by constants.

The multiply instructions take two 16-bit operands (one from any memory location by any addressing mode or any data register, and the other from the lower 16 bits of any data register), multiply them, and place the resulting product into the full 32 bits of the same data register. The divide instructions take the dividend from any 32-bit data register and divide it by a 16-bit divisor, which may come from memory using any addressing mode or any data register. The quotient is placed in the lower 16 bits of the same 32-bit data register, while the 16-bit remainder is placed in

Listing 1: *A short MC68000 assembly-language routine to multiply two 32-bit numbers.*

```
Input:    register D0 contains 32-bit multiplicand
          register D1 contains 32-bit multiplier

Output:   registers D0 and D1 contain the 64-bit result, with
           the most significant byte in D0

          SUBQ      #4,A7          initialize product area
          CLR.L     -(A7)
          MOVE.L    D0,-(A7)       save copy of multiplicand

          MULU      D1,D0          multiply low-order parts
          MOVE.L    D0,8(A7)

          MOVE.W    (A7),D0        high-order multiplicand
          MULU      D1,D0           times low-order multiplier
          ADD.L     D0,6(A7)

          SWAP      D1             now use high-order multiplier
          MOVE.W    2(A7),D0       low-order multiplicand
          MULU      D1,D0           times high-order multiplier
          ADD.L     D0,6(A7)
          BCC       MUL32A         carry into high-order
          ADDQ.W    #1,4(A7)       word of product

MUL32A    MOVE.L    (A7)+,D0       high-order muliplicand
          SWAP      D0
          MULU      D1,D0           times high-order multiplier
          ADD.L     (A7)+,D0
          ADDQ      #4,A7
          MOVE.L    (A7)+,D1       load low-order product
```

the upper 16 bits of the same register.

The divide instruction has two characteristics that may be undesirable and so are specially handled. All of us remember from high school and college that there just isn't any good way to divide by zero. The result is infinite if it's defined at all. Well, Motorola's designers didn't think they knew any better than the mathematicians, so if a zero divisor is detected, the divide instructions do not execute, and a special "trap" procedure is entered. Since the trap operation will be covered in part 3 of this article, let's just say that a "zero-divide trap" specially calls the operating system to decide what to do.

The other thing that could happen is that the divisor could be just too small for the dividend and the quotient could require more than 16 bits in which to be expressed. When this overflow condition is detected, the division is halted, the overflow (V) status register code bit is set, and the instruction is concluded without overwriting either of the original operands. Thus, following any divide instruction, you should check the overflow bit and act accordingly.

For a number of reasons, there are no instructions to multiply two 32-bit numbers or to divide a 64-bit number by a 32-bit number. First, the need for such instructions is very infrequent in most applications. Second, there are no other facilities in the machine to handle 64-bit quantities. Finally, because such instructions would take a lot of time to execute, the MC68000 would occasionally take much longer to respond to an interrupt—a situation the designers did not want to create.

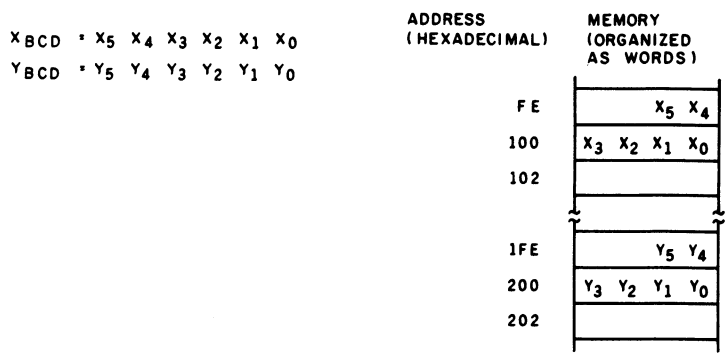
The multiply instructions take fewer than 70 clock cycles to execute using register operands, and the divide instructions require fewer than 140 clock cycles for an unsigned operation (158 cycles for a signed operation); however, different combinations of 1s and 0s in the operands can make these operations take less than these times to execute. A short MC68000 routine that performs a 32-bit by 32-bit multiplication is shown in listing 1. It executes in about 60 microseconds, which is less time than that taken by the dedicated instruction that does the same thing in the Z8000.

Binary-Coded Decimal Arithmetic

The final type of arithmetic instructions handles decimal digits. The most common form of human-interface data comes as *binary-coded decimal* or BCD data. This method of encoding numeric information as a string of bits stores each decimal digit of the number as a 4-bit binary number. Numbers are easily encoded into BCD form; once inside the computer, they are easily printable in human-readable form (much more so than numbers encoded in signed floating-point binary form). Because the BCD format is so useful, most microprocessors include instructions that operate on BCD numbers. To allow these BCD types of data to be manipulated, the MC68000 has three instructions that add (ABCD), subtract (SBCD), and negate (NBCD) packed digits. Each of these instructions works on two BCD digits packed into a byte.

Because BCD numbers may be many digits wide, the BCD instructions work as multiple-precision operations, which means they have the characteristics of the other multiple-precision instructions. The operands can be in data registers or in memory (in which case, they are operated on using the predecrement addressing mode). The value of the X status register code bit is included in the BCD operations, and the Z status register bit is handled so that it properly reflects the state of the entire result, not just the final portion.

Once again, the best thing about these instructions is the simplicity with which they operate, especially when compared with the often mysterious code a programmer had to write to do BCD arithmetic on most older microprocessors. A glance at MC68000 code performing BCD functions (see figure 4) shows how simple such code is. Here, two 6-digit numbers need to be added. While a short loop might make the routine more generally useful, inline code is fastest and illustrates the point best. First, we must load the two address registers to be used as memory pointers with the correct values. The next instruction (SUB D1,D1) is a quick way of both setting the Z bit



CODE TO ADD X_{BCD} TO Y_{BCD}

```
MOVE.L  #$102,A1    LOAD ADDRESS JUST PAST X IN A1
MOVE.L  #$202,A2    LOAD ADDRESS JUST PAST Y IN A2
SUB     D1,D1       CLEAR X STATUS BIT AND SET Z STATUS BIT
ABCD    -(A1),-(A2) BCD ADDITION OF BOTTOM TWO DIGITS
ABCD    -(A1),-(A2) BCD ADDITION OF MIDDLE TWO DIGITS
ABCD    -(A1),-(A2) BCD ADDITION OF TOP TWO DIGITS
```

Figure 4: *An example of multiple-precision binany-coded decimal (BCD) arithmetic. Because the predecrement addressing mode used ("ABCD−(A1),−(A2)") decrements the register pointers before performing the BCD addition, registers A1 and A2 must be loaded with a value that points to the byte immediately after the least significant byte of the number to be worked on.*

and clearing the X bit, though a MOVE #$04,CCR would do virtually the same thing.

The three ABCD (add binary-coded-decimal) instructions begin at the least significant two digits and move toward the most significant; this must be done to get accurate results from use of the extend bit. The result replaces the BCD number pointed to by A2; when the routine has finished, A2 points to the first byte of the BCD result (which is stored in order of most to least significant digit). Similar subtraction and negation operations can be built in the same way.

Logic Instructions

The MC68000 logic instructions are simple but powerful. The AND, OR, exclusive-or (EOR), and NOT instructions, like the arithmetic instructions, allow 8-, 16-, and 32-bit quantities in data registers or in memory to be operated on with any data register or an immediate constant, or to be inverted. These instructions are just as fast as the arithmetic instructions. Additionally, ANDI, ORI, and EORI instructions are used to clear, set, and toggle individual status register code bits.

A serial shifter in the MC68000 can be moved any number of bits to allow for shifting of 8-, 16-, and 32-bit data. The arithmetic-shift-right instruction (ASR) shifts the least significant bit to the X and C status bits while duplicating the most significant bit before moving it to the right. In the arithmetic shift left (ASL), the logical shift right (LSR), and the logical shift left (LSL), the bit shifted out of the data area goes into the X and C bits, while the bit into which no bit is being shifted is filled with a 0.

Status Register Instruction	Register					Status Register Codes		
						C	X	V
ASR.B #3,D3	(D3 before)	10111010	01011111	01100101	10101100	x	x	x
	(D3 after)	10111010	01011111	01100101	11110101 (12 clock cycles)	1	1	0
ASL.L #5,D1	(D1 before)	11101100	10100010	11011101	00101111	x	x	x
	(D1 after)	10010100	01011011	10100101	11100000 (18 clock cycles)	1	1	1
LSL.W D5,D7	(D5 before)	00101000	10001100	11101001	00101001	x	x	x
	(D7 before)	10111010	01011111	01100101	00010101	x	x	x
	(D7 after)	10111010	01011111	00101010	00000000 (24 clock cycles)	0	0	0
ROL.L D2,D1	(D2 before)	01100101	00101010	10111110	01110100	x	x	x
	(D1 before)	10010101	00101000	01000101	10010100	x	x	x
	(D1 after)	01011001	01001001	01010010	10000100 (48 clock cycles)	0	x	0
ROXR.W #4,D6	(D6 before)	10111010	01011111	01100101	00010101	x	P	x
	(D6 after)	10111010	01011111	101P0110	01010001 (14 clock cycles)	0	0	0
ROR $A0000	(word A0000 before)		10011100	10101001		x	x	x
	(word A0000 after)		11001110	01010100		1	x	0

Notes:

1. An "x" status-register bit may represent either a 0 or 1 value.
2. Notice that in the LSL.W and ROL.L examples the bottom six bits of the source operand (D5 and D2, respectively) are used as the number of bits to be shifted or rotated.
3. The "P" status-register bit in the "ROXR.W #4,D6" example is specially marked to show that it is shifted into the body of the D6 register as a result of the ROXR.W instruction. Note that .W causes only the bottom 16 bits of the register to be rotated.

Table 4: *Examples of shift and rotate instructions and their effect on registers and memory.*

The rotate instructions shift bits around in a circular manner so that bits shifted out of one end of an operand are shifted in the other end, with the bit being shifted out of the data area also being copied into the C status register code bit and, optionally, the X bit. The rotate instructions are rotate right and rotate left (ROR and ROL); the ROXR and ROXL instructions are used when you want to update both the X and C bits.

One single shift or rotate instruction can move register data as many as 31 bit positions in the selected direction. You can specify this count value either statically (as a value between 1 and 8 encoded into the instruction op code) when the instruction is written or dynamically (as a value between 0 and 63 stored in a specified data register) when the instruction is executed. For simplicity, memory operands to be shifted or rotated are limited to displacements of 1 bit and operations on word-sized data only. Table 4 illustrates some shift and rotate instructions, their timing, and their effects.

An important aspect of programming that until the MC68000 was quite limited is that of individual bit manipulation, the ability to single out bits of memory, test them, set them, and clear them. Such operations are useful; in I/O, for instance, you frequently need to sense the state of a single input line, drive a particular output line high, or turn a servomechanism off. These operations involve only a single bit associated with a latch, peripheral, or memory location.

In the past, most of us have done the best we could by executing AND, OR, and EOR instructions to the desired location. But the difficulty with these operations is their crudeness. Sure, they allow us to change more than 1 bit at a time, but it turns out that much more secure code can be written when single events or conditions affect single outputs. Also, because it is impossible to sense the state of more than one input at a time, nothing is gained by the ability of such instructions to work on multiple bits.

Four powerful MC68000 instructions make all bit-manipulation functions far simpler. They are the bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG) instructions. How will you specify the target bit? The MC68000 uses two methods, similar to those used for shift and rotate instructions. Either a data register or a series of bits in the bit-instruction op code names the bit to be affected; in this case, however, the bit number can be from 0 to 31 if a register is affected, or from 0 to 7 if the area affected is a memory location. (In the MC68000, bits in memory are identified by the bit number of the *byte* in which they reside.)

With true bit-manipulation instructions, not crude logic instructions, bit-manipulation operations—sensing the state of inputs, driving outputs, setting register bits, setting attribute bits, transposing bit matrices, or just building special data types—are straightforward tasks, not the chores they usually are with other microprocessors. The MC68000 makes it very easy to specify precisely the bit to be changed.

Conclusions

The computation and data-movement instructions that perform the major work in any MC68000 program are numerous, comprehensive, and, perhaps most important, straightforward and easy to use. The versatile MOVE instruction on the MC68000 replaces a confusing variety of data-movement instructions on other microprocessors. Flexible add, subtract, compare, negate, multiply, and divide instructions operate on any register, with constants, on stacks, and in memory using any addressing mode. For digital data rather than binary data, pairs of BCD numbers can be added, subtracted, and negated. The common logic operations of AND, OR, exclusive-or, and NOT can similarly operate on data registers and constants, and in memory.

When data needs to be shifted about, it can be arithmetic-shifted,, logic-shifted, or rotated left or right. It can also be shifted or rotated multiple bit positions, with the count of the movement either predetermined and constant, or variable and dependent upon other data.

Individual bits in data or I/O can be separately tested to determine their state; they can also be set, reset, or toggled. The bit to be worked on can be chosen either when the instruction is written or, based on other data, when it is run.

All the above instructions can operate on 8-, 16-, or 32-bit data, with a uniform yet flexible set of addressing modes. This combination of good instruction set design, computational power, and ease of use make the MC68000 microprocessor an excellent one for assembly-language programming. Next month, I'll discuss program-control instructions and several advanced instruction groups.■

Design Philosophy Behind Motorola's MC68000

Part 3: Advanced instructions

Thomas W. Starnes
Motorola Inc., Microprocessor Division
3501 Ed Bluestein Blvd.
Austin, TX 78721

Last month (May BYTE, page 342), I discussed the data-movement, arithmetic, and logic instructions of Motorola's MC68000 family of microprocessors (sometimes referred to as MACSS—Motorola's Advanced Computer System on Silicon). I examined a useful set of instructions based on a philosophy of *orthogonality*, which eliminates duplication of effort by similar instructions (thus making the microprocessor easier to understand and use). In this final part of the series, I will discuss branching, jumping, error-trapping, supervisor-mode, and other advanced instructions of the MC68000.

Branching and Jumping

Data-movement, arithmetic, and logic instructions do most of the computational work in programs, but computers would be little more than adding machines without *program-control instructions*. These instructions give computers the capability to make decisions by executing nonsequential areas of code based on conditions tested at the time of execution. Branch instructions enable the microprocessor to transfer control to portions of code relative to the instruction being executed—that is, to transfer control to the effective address, which is the sum of the current contents of the program counter and a given offset. You use branch instructions extensively when writing position-independent code. Jump instructions differ from branch instructions in that the jump instructions transfer control to absolute locations in memory, are unconditional, and can use any of the MC68000 addressing modes to specify the destination.

The MC68000 has a flexible conditional branching instruction, referred to as a *Bcc instruction*, in which the letters cc denote a variety of conditions that can be specified. There are 14 different conditions, including such things as greater than (BGT), less than or equal to (BLE), equal (BEQ), overflow (BVS), and low or same (BLS); a complete list is given in table 1. The BRA instruction is not conditional but always forces the branch to occur.

You cause branching by the addition of some value to the program counter (PC). Branch instructions include an 8- or 16-bit signed displacement value that you add to the program counter. Because the displacement is a signed number, it can cause either a forward or backward branch. If the condition being tested evaluates to "true," the MC68000 will take the branch; if it is not, it will execute the next instruction in sequence.

Even though all MC68000 instructions are multiples of 16 bits and must be aligned on word boundaries (i.e., start on even addresses), the MC68000 interprets the displacement in all branching operations to be a byte count, not a word count. This is done to give the machine maximum flexibility, while still providing the most opportunity for future growth. Limiting the machine to word offsets would have prevented any future MC68000-family processor from having instructions that could be misaligned (i.e., did not start on a word boundary) or be multiples of 8 bits. Since the flexibility to have misaligned instructions still exists, it makes sense to follow the natural byte-oriented addressing of the MC68000. A 16-bit offset gives an addressing range of −32,768 bytes to +32,767 bytes, not the puny 8-bit computer range of −128 to +127 bytes.

Versions of the jump and branch instructions also exist for subroutine calls. You can branch to a subroutine (BSR) with a displacement value, or you can jump to the subroutine (JSR) by specifying the absolute address. Subroutine calls save the return address (the current value of the program counter) on the system stack before transferring control to the subroutine; the return address is

Mnemonic	Condition Description	Flags Tested
T	true	1
F	false	0
HI	high	$\overline{C} \wedge \overline{Z}$
LS	low or same	$C + Z$
CC	carry clear	$\overline{C}$
CS	carry set	C
NE	not equal	$\overline{Z}$
EQ	equal	Z
VC	overflow clear	$\overline{V}$
VS	overflow set	V
PL	plus	$\overline{N}$
MI	minus	N
GE	greater or equal	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
LT	less than	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
GT	greater than	$(N \wedge V \wedge \overline{Z}) \vee (\overline{N} \wedge \overline{V} \wedge \overline{Z})$
LE	less or equal	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$

Table 1: *Conditional tests for the Bcc and DBcc groups of instructions. By substituting the letters in the first column for the letters cc, you can construct as many as 16 Bcc (branch on condition) and DBcc (test condition, decrement, and branch) instructions; for example, BHI branches if both the carry and zero bits in the status register are cleared. The third column indicates that the branch will take place if the expression evaluates to "true"; "∧" indicates a logical AND operation, while "∨" indicates a logical OR operation. The same conditions are available to another instruction group, Scc, which sets or clears all the bits of a given byte based on the condition being evaluated.*

removed from the stack and restored to the program counter when the MC68000 executes the RTS (return-from-subroutine) instruction.

Sometimes you will want to save and restore the condition codes that existed just before the subroutine was called. This is easy enough to do with the MOVE SR,−(A7) instruction, which pushes the status register onto the system stack (pointed to by register A7). You can also save selected registers with a single MOVEM instruction (discussed last month). At the close of the subroutine, you can use a MOVEM instruction to restore the saved registers and then use an RTR instruction (return and restore condition codes) to return and restore the saved condition codes in one operation.

Looping and String Constructs

Many times, a backward branch is used to create a *programming loop*, which is a very important part of programming because it allows operations to be repeated until a desired state or condition is reached. Although the looping constructs that most people are familiar with provide for a loop that ends with a given condition or one that ends after a certain number of iterations, a loop that can end by *either* means is often very useful. The double condition allows a loop to be performed until a given condition is met while ensuring that the loop does not process invalid data (in the case of, say, a string operation that reaches the end of the data without meeting the condition) or run forever (in the case of a numerical analysis algorithm that never converges to a given minimum tolerance).

The MC68000 has just the instruction for this kind of loop. The *decrement-counter-and-branch-conditionally* instruction (DBcc) uses any data register as a counter and branches based on both the evaluated condition and the data-register value. A DBcc instruction causes the following sequence of events. First, the MC68000 checks to see if the stated condition is met; if so, execution continues with the *next* instruction, thus ending the loop. If the condition is not met, the specified register is decremented by 1. If the resulting value is −1, the loop is again ended by having the execution continue with the next intruction; otherwise, the branch to the top of the loop occurs.

Note that the DBcc instruction tests the register for a value of −1. At first, this might seem odd, but there is a very good reason for it. Most looping constructs require extra steps to ensure that the loop can execute zero times when needed *and* that the loop tests for the desired condition *before* executing a given iteration. By having the loop entered just before the DBcc instruction (at the end of the loop) and by designing the DBcc instructions so that they end the loop on a value of −1 instead of 0, you create a loop that meets both of the above conditions without being burdened by an explicit second test. As an added bonus, a simple conditional branch instruction (using the same condition as the DBcc instruction) enables you to determine whether the program exited the loop because of the iteration counter or the condition.

The DBcc instruction provides a huge set of string operations, especially in conjunction with the predecrement and postincrement addressing modes. By using the appropriate MOVE instruction, for example:

```
    MOVE Dn,(An)+;
    MOVE (An)+,(An)+;
    MOVE −(An),−(An);
or  MOVE (An)+,−(An)
```

followed by a DBRA instruction, you can have the MC68000 fill a block of memory, copy strings, and reverse strings. CMPM −(An),−(An) with DBNE compares two strings, while CMP Dn,−(An) with DBEQ searches a string for a pattern match. (See part 2, May BYTE, page 342, for an explanation of this address mode notation.) Multi-instruction loops can make very powerful string operations quite simple.

You can see the real beauty of the

Listing 1: *A string translation program that uses the DBEQ instruction to end a loop based on either of two conditions: end of string (as determined by the string length, given in register D3) or discovery of a termination character (stored in register D4). This program translates a string character by character according to the character values stored in TABLE. For a given character, its value (stored in register D0) is used as an index into TABLE (pointed to by register A2); the actual translation takes place at LOOP.*

```
        MOVEQ     #$13,D4          load termination character into register
        MOVE      #COUNT,D3        load string length
        MOVE.L    #STRING,A1       load string beginning
        LEA       *+TABLE,A2       offset to conversion table
        CLR       D0               prepare index for word
        BRA       POOL             start translation
LOOP    MOVE.B    0(A2,D0),(A1)+   translate and store result
POOL    MOVE.B    (A1),D0          load next character
        CMP.B     D4,D0            termination character found?
        DBEQ      D3,LOOP          if not and not end of string, branch
```

Execution time where **n** bytes are translated:
72 + (40 * **n**) clocks = 649 μs for 128 bytes at 8 MHz

DBcc instruction in the assembly-language program of listing 1, which translates a string of characters until a terminating character shows up or the end of the string is reached. Register D3 has the string length in it, while register D4 contains the terminating character you're looking for. Register A1 points to the string, while the translation table (which is found some distance from this code) has its location placed in register A2. The routine in listing 1 runs very quickly and demonstrates the power that results from a combination of versatile instructions and various addressing modes.

High-Level Language Aids

Many high-level languages, such as Pascal, use sophisticated programming concepts that can be enhanced by the use of reentrant and recursive programming and subroutines with local variable areas. The MC68000 has the facilities to support these techniques.

You can enter reentrant code at any time by several execution processes, and it will return correct results to all of them. This is very important for interrupt routines that may interrupt themselves before completion. Only reentrant code will correctly execute the interrupt routine the second time, then return to its interrupted version and correctly execute it. The MC68000 instruction set makes reentrant programming easy.

Recursive programs are those that can call themselves. An example of such a program might draw a straight line between two points by repeatedly plotting the midpoint of the line, then calling itself to operate on the two line segments created by the new point. Recursive programs are created to solve complex algorithms with relatively small amounts of code. Their disadvantages include slow execution and heavy use of the stack (or some other area) for storing each level's set of temporary variables. Of course, the MC68000 designers included special instructions to make this task easier, LINK and UNLK (unlink).

LINK and UNLK allow subroutines to allocate part of the stack for the storage of local variables quickly and easily. Often, a programmer needing to refer to variables associated with a given subroutine call will decrement the stack pointer to reserve an area of memory for such use ("decrement" because stacks usually "grow" downward in memory) and save the address of the top edge of this area as a reference point; this address is called a *frame pointer* (FP) and is a value that, on the MC68000, is stored in one of the seven address registers A0 through A6. The stack pointer, of course, will move up and down during the execution of a subroutine as stack operations are performed. The stable frame pointer always gives a good reference point to the variables, while the stack pointer would give a wildly varying reference to those same variables. Now let's look at a good method for going into a new routine.

To show how the LINK and UNLK instructions help give the programmer access to local variable areas, let's look at the example of figure 1. (Remember that the frame pointer is actually an address register that is designated by the programmer for this use.) Assume that you are in subroutine A, which has its own local variable area, pointed to by the frame pointer. Before subroutine A calls subroutine B, it first places parameters on top of the stack; see figure 1a. After the subroutine call to B, the return address to A is pushed onto the stack (figure 1b). The LINK instruction contains the name of an address register that is to be taken as the frame register and a displacement that indicates the amount of memory to be saved for local variables. When it is executed, three things happen (figures 1c-1e): the contents of the frame pointer (pointing to a stack location containing the previous frame pointer) are pushed onto the stack, the frame pointer itself is made identical to the stack pointer, and the stack pointer is changed by the displacement given in the instruction. (The displacement is a signed value and must be negative to save local variable space—if it is positive, you will lose information from the stack.) As shown in figure 1e, the stack pointer points to the top of the stack, and the frame pointer points to one word below the subroutine B local variable area. When the UNLK instruction is executed, the process is reversed (figure 1f), leaving subroutine B ready to execute an RTS instruction and return control to subroutine A.

Address Calculation in Hardware

Most microprocessor operations deal either with data or program control. Most also use memory addresses and, in the case of the MC68000, have some rather sophisticated means of generating those addresses. But the addresses are used by the instruction only to get to the data or program location; the address itself is never available to the programmer and is

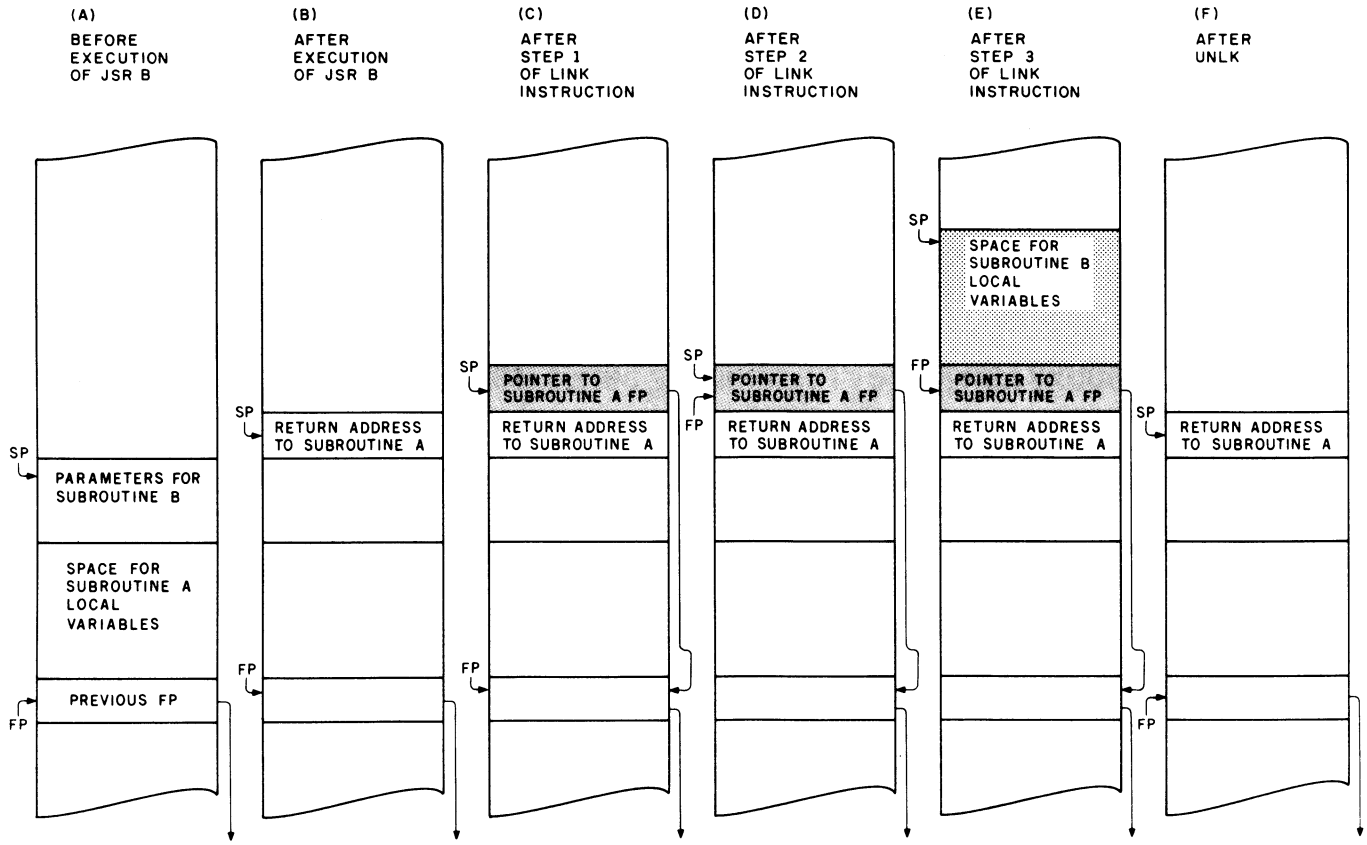


Figure 1: *Use of the LINK and UNLK (unlink) instructions, both of which help the assembly-language programmer manage memory areas to be used for local variables in subroutines. See text for details.*

often lost by the end of the instruction. However, it is sometimes the address itself that you need in your program. The MC68000 has two instructions that help you to get just the address, without using it to fetch any data. By having the MC68000 calculate the address itself (instead of writing a sequence of assembly-language instructions to do the same), you can do the calculation much faster without tying up either memory or registers.

The load-effective-address (LEA) and push-effective-address (PEA) instructions calculate a given effective address and place it either in any address register (LEA) or on the stack (PEA). You can calculate the effective address by using any available addressing mode with any of the appropriate registers. The LEA and PEA instructions can be useful when you are running position-independent code. Sometimes to take advantage of an addressing mode that runs more quickly than, say, the program-counter-relative addressing mode, you may want to calculate the address using LEA and access that area of memory by addressing indirectly through the address register in which the LEA instruction left the calculated address. PEA and LEA are also useful for passing pointers of data to other routines or placing them in memory. Sometimes, it's helpful to verify that an effective address is correct or at least in range. Without these two instructions, it would be extremely difficult to use processor-generated addresses.

Instructions for Shared Resources

Systems with more than one microprocessor running at a time are often designed to share some resources, e.g., memory, buffers, I/O, tasks, and so on. For this to happen, the programs running on the microprocessors must have a secure method of determining which processor has rights to a certain part of memory, a buffer, I/O, or a task. The MC68000 has an instruction, TAS (test and set), that makes such allocation of resources between multiprocessors simple and secure.

The key to this instruction is that it is *indivisible*, i.e., it can lock out all accesses to the designated addressing location until work on the location is complete. The test-and-set instruction tests a given byte, sets the negative (N) and zero (Z) status register bits accordingly, and then sets the most significant bit of the byte to 1.

In most cases, the microprocessor uses the TAS instruction as follows: It chooses a given byte to represent the status of a shared resource (this byte is often called a *semaphore*). If the TAS instruction shows the byte to be negative (if its most significant bit is 1), the querying microprocessor knows that the resource is in use. The processor can then either retest the semaphore byte until it shows the resource is available, or it can go about some other task. If the TAS instruction shows the byte to be positive (most significant bit is 0), the microprocessor knows the resource is free. Because the TAS instruction im-

mediately sets the most significant bit to 1 (and because the instruction cannot be interrupted before completion), all the microprocessors with access to the semaphore byte have correct information about the shared resource. The microprocessor that has access to the shared resource has the responsibility of clearing the most significant bit when it is finished.

The only reason this process can work effectively is that the indivisible read-modify-write bus cycle (a special bus cycle) that accompanies the TAS instruction prevents, with hardware signals, any other device from accessing the semaphore byte between the time the TAS reads it and the time it is through setting the bit in it. This means that no two processors can read a semaphore byte and both be told that the resource is available. Thus, a secure way exists for software to determine the availability of shared resources in a multiprocessor MC68000 system.

Supervisor and User Modes

The MC68000 executes instructions at one of two operating or privilege levels. The upper level, called the *supervisor level*, provides a protected environment for the operating system to run in, isolating it and its resources from the less trustworthy user code. After a reset operation, the MC68000 begins running in the supervisor mode, in which the operating system and all interrupt routines are also running. The lower level, called the *user* level, is where most application programs execute and, therefore, where the processor usually spends most of its time.

The only controlled way to get from the supervisor level to the user level is by changing the S/U (supervisor/user) status bit (bit 13) in the processor status register. This is how the operating system switches to begin a user-level program. Should an interrupt be handled in the middle of a user-level routine, the interrupt routine will run at the supervisor level, but upon return to the interrupted routine, the MC68000 will return to the user level.

User-level programs are guaranteed to go to the operating system only through one of the 16 TRAP number *n* instructions. You can view these instructions as supervisor calls; they immediately transfer control to a specific routine. Upon completion of the TRAP routine, the processor will usually return to the original user-level routine to continue. Thus, there are 16 different supervisor trap instructions, which, along with other types of trap instructions, are listed in table 2.

Many other means of getting to the supervisor level of execution exist, but they are either conditional (like error traps) or asynchronous (like interrupts). Regardless, all traps are handled similarly by the supervisor. Any trap causes the processor to save the old program counter and status register on the supervisor stack. Then it will go to its external vector table and get a value, appropriate for the cause of the trap, to load into the program counter. This allows each type of trap to have a separate handling routine to correct the problem causing the trap and return to the original program.

Some of these other trap forms are intentionally conditional. Depending upon whether the overflow (V) condition bit is set, one instruction, TRAPV, either does nothing or causes a trap to occur, which forces the MC68000 into the supervisor state. This enables the program to handle all overflow conditions uniformly in a single operating-system-level routine. Another such instruction is the check (CHK) instruction, which verifies that the contents of any data register is greater than 0 but less than a specified bound. If it is within the limits, then nothing happens and the next instruction is run. If it is outside the bounds, then program control jumps indirectly through the vector table to a certain trap routine for handling. This gives the programmer an easy way to check whether an array index is within the proper bounds for that array. In addition, attempts to divide by 0 and access misaligned data (words or long words in memory on odd-byte addresses) will cause trap routines to be executed.

Handling Illegal and Unimplemented Op Codes

To allow room for future expansion of the MC68000, designers did not use all of the possible bit patterns of the 16-bit op codes. Other microprocessors try to execute undefined op codes, often with disastrous results that cause you to lose control of the computer or even lose valuable work. To assure a completely foolproof system in the face of undefined op codes, the MC68000 refuses to execute any illegal instruction and, instead, executes a specified trap routine for corrective action.

To enable programmers to add whole blocks of new instructions to MC68000 processors, designers left two subgroups of possible op codes unimplemented. Any 16-bit op code beginning with binary 1010 or 1111

Type of Trap	Cause
address error	word or long-word access to an odd address
illegal instruction	no valid instruction exists for this op code (op codes starting with "1010 . . ." and "1111 . . ." generate other traps; see below)
zero divide	attempt to divide by zero
CHK instruction	CHK instruction failed (operand out of bounds)
TRAPV instruction	overflow has occurred (V bit set)
privilege violation	attempt to execute a privileged instruction while in the user mode
trace	an instruction has just ended and the T status register bit is set
line 1010 emulator	attempt to execute an op code that starts with "1010"
line 1111 emulator	attempt to execute an op code that starts with "1111"
TRAP *n* instruction	TRAP *n* instruction executed ($n = 0,1, \ldots ,15$)

Table 2: *Supervisor trap types and their causes.*

STOP
RESET
RTE
MOVE (when moving a word to the status register)
MOVE USP
AND, EOR, or OR (when combining an immediate value with the status register)

Table 3: *Privileged instructions in the MC68000.*

was left without definition in the MC68000. Attempts to execute either of these categories of op codes, even though they could be considered illegal instructions, are trapped separately. They cause either a line-1010-emulator or a line-1111-emulator trap routine to execute, enabling the programmer to emulate in software functions that are not implemented in the processor chip of the system. Currently, the 1111 op codes are defined mostly as floating-point instructions and so could be emulated on the MC68000. The 1010 op codes are still reserved for use in processors beyond the MC68020.

Privileged Instructions

Privileged instructions have a special characteristic—they can be executed only while the processor is running at the supervisor level. Attempts to run them at the user level force privilege-violation traps to occur, allowing the supervisor to take whatever action it thinks suitable.

The privileged instructions are listed in table 3 and are mostly self-explanatory. These instructions are restricted because they modify or control resources or services that must be under the control of the operating system. Many of these instructions modify the upper portion of the status register (SR), which contains the S/U supervisor bit, the interrupt mask, and a trace-mode switch. Such resources are not meant to be in the hands of the users, but maintained by the supervisor; this is why they are restricted.

Another supervisor-privileged resource is the *supervisor stack pointer* (SSP). This pointer is visible (as address register A7) only when the MC68000 is running at the supervisor level, just as the *user stack pointer* (USP) is visible (as register A7) only when the MC68000 is running at the user mode. However, when the operating system is ready to pull in a new user-level task, it needs to be able to access the hidden USP to initialize it. This is done with a special, privileged MOVE USP instruction.

The STOP instruction halts processor execution of further instructions, while waiting for an interrupt, a trace exception, or a reset to initiate new activity. The instruction also loads the status register with an immediate 16-bit value, allowing the programmer to enable certain interrupts before stopping the microprocessor. Only the supervisor can initiate this type of operation because, in a user's hands, the operation might throw off all sorts of operating-system timing integrity (such as time-slice clock signals) and generally brings the system to an irrecoverable halt. Also, the instruction must be restricted because it affects the entire status register.

The RESET instruction is a unique and powerful operation. Its execution pulses the reset line on the MC68000 without resetting the processor itself. You use this instruction typically after a catastrophic failure, from which the operating system is trying to recover on its own. It enables the operating system to initialize its external environment (i.e., reset the entire system except for the microprocessor) without forcing itself into a complete restart. Obviously, this instruction's power makes it inappropriate for the user level.

One final privileged instruction is the RTE (return-from-exception) instruction. An exception is anything that causes the microprocessor to perform an operation other than the next normal instruction. Interrupts and traps, then, are exceptions. The RTE is similar to the return-from-interrupt instruction of most microprocessors. It basically reloads both the program counter and the status register with the values from the top of the stack. Because all exceptions force the processor to execution in the supervisor mode, the RTE instruction will be executed only in that mode; this makes it a privileged instruction.

Note, once again, that privileged instructions can be executed from only the supervisor level of operation, where the operating system usually resides. The two different levels of privilege and the restricted use of privileged instructions allow you to build systems that prevent user-level application programs from, inadvertantly or otherwise, running rampant through operating-system code and data.

Conclusion

As you have seen in previous installments of this article, the MC68000 architecture is really designed with the programmer in mind. The MC68000 branch and jump instructions give you complete control over program flow and simplify often-used looping and string-movement constructs. The link and unlink instructions make it easier for you to create modular programs that use local variables. Other instructions carry out complex address calculations quickly, help mediate the use of shared resources, provide for the data integrity of the operating system, and allow recovery from errors under program control. In addition, planners designed the architecture and instruction set with far greater things in mind and made the set easy to expand to more powerful and more comprehensive functions. And all this has been done with a processor for which performance was a primary criterion.

Once you learn a few general concepts of programming the MC68000, coding an application comes easily. Pick up an MC68000 user's manual and a similar guide for any other 16-bit microprocessor. Then spend an hour or two learning each. Code a short program or two, and compare just how easy the MC68000 is to work with. And if you choose to write code for a larger program, you will find your task to be simple

regardless of program size.

The MC68000 was designed to be a programmer's instrument through which programmers and system designers could use their creativity to engineer a system to fit the application instead of having to figure out some trick to get the microprocessor to perform the needed task. We at Motorola believe that using the right tool gets a job done faster with fewer mistakes, and the MC68000 is such a tool. ■

About the Author

Thomas Starnes is an electrical engineer who has spent the last five years helping to plan the direction of the MC68000 family of processor products for Motorola.

A16640-2 PRINTED IN USA 4-85 IMPERIAL LITHO C30281 5,000