The designers of this microprocessor used the continuation method to provide an elegant general solution to the problem of virtual memory support.

Virtual Memory and the MC68010

Douglas MacGregor and David S. Mothersole
Motorola, Inc.

Just a few years ago, the introduction of 16-bit devices signalled a new generation of microprocessors. With them came powerful capabilities previously available only on minicomputers and mainframes. However, most of these microprocessors did not have the facilities to easily manage this new power. With the introduction of the MC68010, Motorola is providing support for virtual memory and virtual machine operation. In order to assist the reader in understanding how this was done, we will first briefly define the concepts of virtual memory. We will then compare two different methods of implementation from an architectural perspective. After providing this background, we will present the details of the actual implementation of the MC68010 and review its facilities.

The MC68010 16-bit microprocessor is an extension of the MC68000. It provides virtual memory capability, virtual machine support, and increased performance, while maintaining code compatibility with the M68000 architecture. One of the most important requirements of any new member of a processor family is that it not require major revisions of the software written for previous members of the family. For this reason we considered it essential that the MC68010 be code-compatible with the MC68000. Since the extensions provided by the MC68010 required changes in the processor interface to the operating system (i.e., exception processing, privileged instructions, and dispatching), the MC68010 was designed so that the required software modifications would be confined to the operating system.

Virtual memory

The most significant feature of the MC68010 is its ability to support virtual memory in a system by providing all of the mechanisms needed for its implementation. When the MC68000 was introduced, with its 32-bit addresses and its addressing range of 16 megabytes, it became clear that there was a need for mechanisms by which this large address space could be hierarchically maintained and accessed. The fundamental notion of a virtual memory system involves maintaining a large address space on a hierarchy of memory devices with different storage capacity, cost, and speed ratios.[1] A simple example of a system with three levels of hierarchically organized memory is shown in Figure 1.

The concept of using a paged memory with a backing store is not at all new; it was first introduced in the late 1950's when it was used in the ATLAS machine developed at Manchester University.[2] Although the virtual memory concept has been expanded in the last two and a half decades, the basic theory underlying it has changed little. What has changed is the size of the processors on which this support is provided; functionality that was previously only available on mainframes and high-end minicomputers is now available on microprocessors.

The need to organize memory heirarchically becomes all the more acute as the clock frequencies at which processors execute continue to increase. In the case of the MC68000, a physical memory design that could hold the entire 16-megabyte address space and provide no wait-state access would involve substantial expense.

Virtual memory concepts. Originally, the need to more efficiently utilize expensive memory provided the motivation for the development of virtual memory concepts. A virtual memory system allows the user to execute programs on a very large store of virtual address space without regard for its physical existence. A memory management system, which may comprise hardware and/or software, maps the virtual (or logical) address of the user into the smaller physical memory. Virtual and physical address spaces are divided into fixed-size pages to facilitate mapping. The user need not have any knowledge about the organization of the physical memory into which his program is mapped. If an access is attempted to an address on a page which is not resident in the physical memory, a page fault occurs, interrupting the processor and initiating exception processing. The virtual memory system can correct this fault by fetching the page from an element lower in the hierarchy and substituting it for one of the pages in the physical store. While this replacement operation is occurring, the processor is free to service other users and thus support multiprogramming more efficiently. After correcting the fault, the processor is then permitted to resume execution of the faulted program.

This description is very general and in theory simple. Unfortunately, when it is necessary to implement virtual memory on a processor, some of the details of implementation create several difficult problems for either the group designing the processor or the end user.

MC68010 processor design goals. The goal adopted by the microprocessor design group at Motorola was to develop a processor capable of cleanly and elegantly supporting the fault detection/fault correction/program resumption process. To achieve this, the group needed to design a processor able to recognize a fault indication on any bus access attempted and, regardless of the instruction being executed at the time of the fault, able to carry out a simple recovery and resumption process.

Ironically, the reason that it is not possible to provide a complete recovery for 100 percent of fault conditions lies in one of the strongest aspects of the M68000 family architecture—its generality. The one situation from which the MC68010 cannot make a successful recovery is a fault on an access to the system stack pointer. The fundamental cause of this problem is the general nature of the M68000 stack pointer. Under almost all circumstances, it is desirable to treat the stack pointer as a general-purpose register. This generality, however, also implies that it is possible to load any address into the supervisor stack pointer without regard to the residency of that address in physical memory. If a fault occurs, the processor needs to save the internal state of the processor on the supervisor stack before it can proceed with handling the exception. Thus, if the supervisor stack is not resident in physical memory, the attempt by the processor to save the state of the faulted process results in yet another fault, a double bus fault, which overwhelms

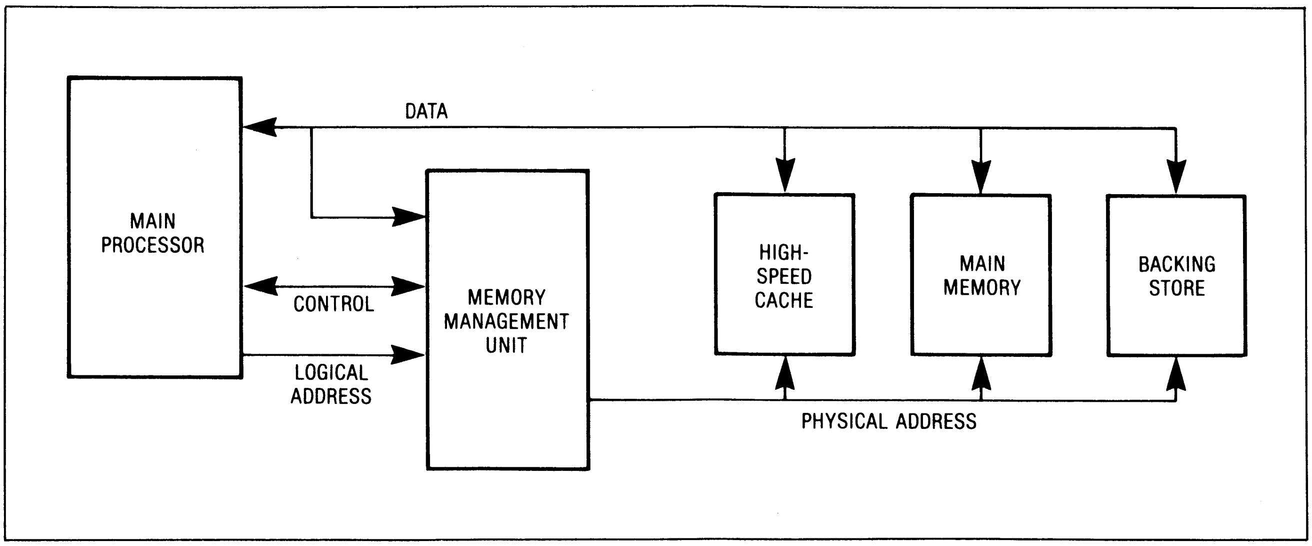


Figure 1. System with three levels of hierarchically organized memory—high-speed cache, main memory, and backing store.

the fault recovery hardware and forces the MC68010 into an unrecoverable situation. In order for the processor to provide *complete* protection, any address to be loaded into the supervisor stack pointer would first have to be checked to ensure its validity. However, such an activity would be inconsistent with the general stack-pointer register concept. Nevertheless, it is desirable to provide complete coverage. Another solution is available—keeping the supervisor stack resident in physical memory. This eliminates the need to check addresses to be loaded into the supervisor stack pointer. Thus, the general stack-pointer register concept is preserved and complete fault coverage is provided.

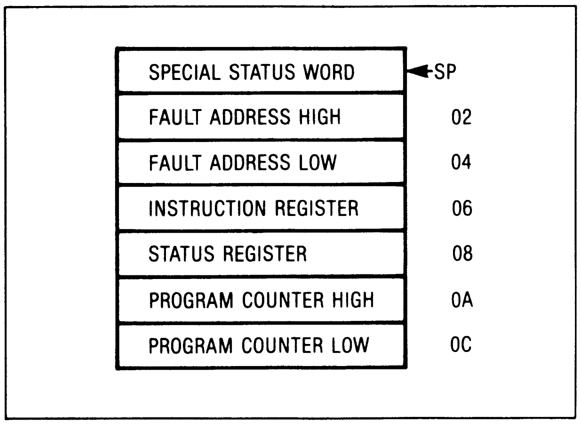


Figure 2. MC68000 address error/bus error stack.

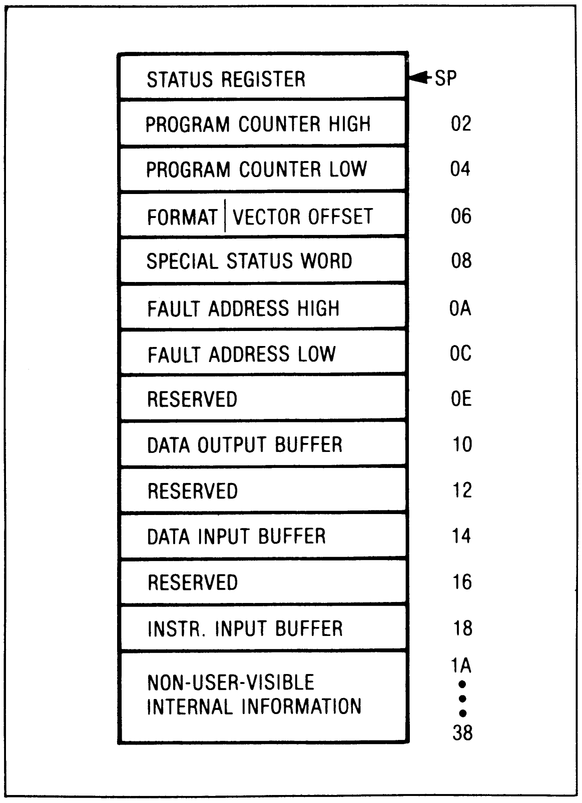


Figure 3. MC68010 address error/bus error stack.

Basic virtual memory processor requirements. In order to provide virtual memory support, a processor must be able to perform three basic functions: recognizing a fault, saving any information needed to recover from the fault and executing the exception handler, and restoring the saved state and resuming normal processing. The MC68000 provides some of these functions, since it can recognize the unsuccessful termination of a bus cycle, save some state information, and execute the exception handler.[3] By expanding these capabilities to include a complete save of the internal state of the processor, and by providing the ability to restore the state of the machine and resume execution, the MC68010 provides all of the mechanisms needed to support virtual memory.

An essential element in providing virtual memory support is the ability to recognize an access fault when it occurs. The MC68000 can recognize these faults both internally and externally. When an access is made to a misaligned instruction or operand, an address error is internally detected, initiating the address error-fault handling routine. Externally, the bus error (BERR) pin provides the user with a method to signal that some aspect of the access has generated an error. In the context of virtual memory, a Motorola MC68451 memory management unit, or any other address translation device which can detect a fault situation, can signal a fault to the main processor via the BERR pin. BERR is shown in Figure 1 as a control signal. Although this fault recognition capability does not need to be enhanced to support virtual memory, some of the timings associated with it were made more liberal to provide support for error detection and correction hardware.

Once the processor has recognized an access fault, the next step is to save any state information that will be needed to reconstruct the state of the machine after the fault has been corrected. The MC68000 saves only enough internal state information (Figure 2) to provide the user with an approximate indication of the state of the processor when the fault occurred. This information, though providing the fault address, function codes, and type of access, does not provide enough data to allow the internal state of the machine to be reconstructed. One of the side effects of the pipelined instruction stream on the MC68000 is that the program counter does not necessarily point at the instruction in which the fault occurred, but rather points to the vicinity of the instruction.[4] Furthermore, because of the pipelining, the instruction register is updated before the end of an instruction, an action which can result in the stacked value of the instruction register also being misleading. In order to provide the required data, the MC68010 has to expand the size of the state that is stored on a fault from seven words to 26 words. This stack frame, shown in Figure 3, consists of the data stacked by the MC68000 but also includes more detailed information about the access type, internal temporary registers, and various internal status bits. The stack is divided into two parts—a user-visible section in which everything that the user needs to know about the access and its correction are provided, and a non-user-visible section in which the internal status and temporaries are stored. After the internal state has been saved, the processor returns control to the operating system by providing a vector to the address error or bus error exception-handler routine. The operating system is then responsible for reconfiguring the system to either successfully complete the access or abort the process. While this activity is taking place, a different program can be dispatched to the potentially idle processor in a multiprogrammed environment.

The last step is the most complex for the processor. After the operating system has made any repairs that are necessary, the state of the program suspended by the fault must be reloaded and the execution of that program resumed. If operating in a multiprocessor system, the suspended program can be dispatched to any processor throughout the system, which may or may not be the processor that was originally faulted. The processor must use the saved state information to reconstruct the internal state of the machine and must allow execution of the faulted program to resume. Reconstructing the internal state is composed of two steps: reading the internal state from memory and loading it into the machine, and evaluating this state to determine what actions are needed to restore the state of the machine to its prefault condition. In the MC68000, an address error or a bus error is considered to be exception from which no recovery can be made. Hence, it has no facilities to allow the machine to return from these exceptions. The mechanism described here is an addition made on the MC68010.

Processor instruction flow. In order to provide some framework for the topics to be discussed later, it will be advantageous to first study an example of a simple instruction and define the terms used in describing instruction flow.

Any instruction can be implemented internally as a series of microinstructions. A microinstruction is an integral unit of activity within a processor. Let us examine a simple move instruction. In a word-sized move-memory-to-memory instruction, in which both source and destination are addressed using the predecrement addressing mode [MOVE. W −(An), −(Am)], the following activities must take place. The address register An is decremented by two and the processor then uses that address to read the word-sized data. The data are then written to the memory location addressed by address register Am decremented by two. In each case, the decremented value is stored back into address registers An and Am, respectively. This instruction can be partitioned into three microinstructions, as shown in the flowchart in Figure 4.

There must be some way to synchronize the external world and the processor. This is done by allowing the internal sequencing of the machine to be effected by the bus controller. When an access is initiated in a microinstruction, that microinstruction is not considered complete until that access has been terminated. In reality, because the execution time of a microinstruction is half the time required to execute a bus cycle, there can be two microinstructions associated with one bus access. In this case, the first microinstruction initiates the bus access and the second microinstruction waits until the access is completed before releasing the machine to continue execution. If the access is completed successfully, then normal processing is allowed to continue. Figure 5 provides two examples of this synchronization, one without any memory delays, and the second with a one-clock wait state. In the first case, microinstruction E initiates the bus cycle and microinstruction F completes it. In the second case, microinstruction F is extended by one clock cycle to accommodate the memory delay. In the example provided in Figure 4, the processor is not allowed to begin execution of microinstruction C until the access in microinstruction B has been completed. This ordering is necessary because microinstruction C will use that data to write to memory and to set the condition codes. The processor, however, has completed all of the other activities associated with microinstruction B and is simply waiting for the access to be completed.

Virtual memory implementation methods

There are two basic methods of implementing virtual memory on a processor: instruction "restart" and instruction "continuation." Both methods have their

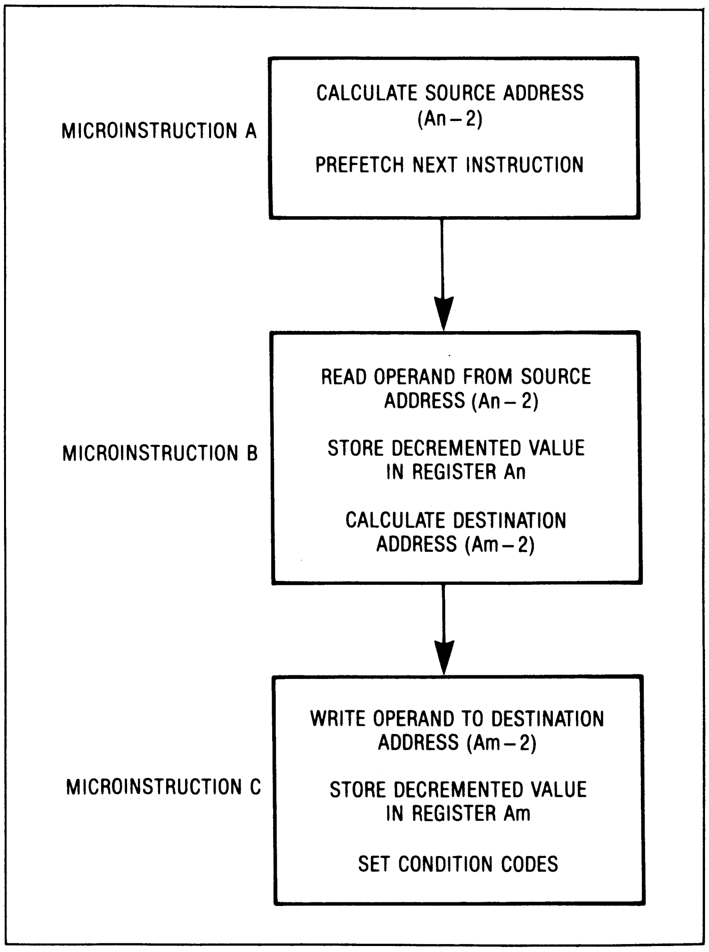

MICROINSTRUCTION A
CALCULATE SOURCE ADDRESS (An−2)
PREFETCH NEXT INSTRUCTION
MICROINSTRUCTION B
READ OPERAND FROM SOURCE ADDRESS (An−2)
STORE DECREMENTED VALUE IN REGISTER An
CALCULATE DESTINATION ADDRESS (Am−2)
MICROINSTRUCTION C
WRITE OPERAND TO DESTINATION ADDRESS (Am−2)
STORE DECREMENTED VALUE IN REGISTER Am
SET CONDITION CODES

Figure 4. MOVE.W −(An), −(Am) microflow.

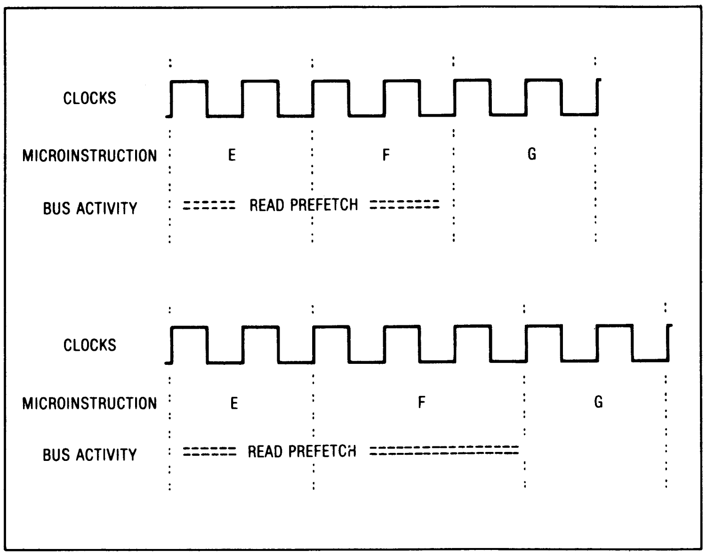


Figure 5. Bus controller/micromachine synchronization.

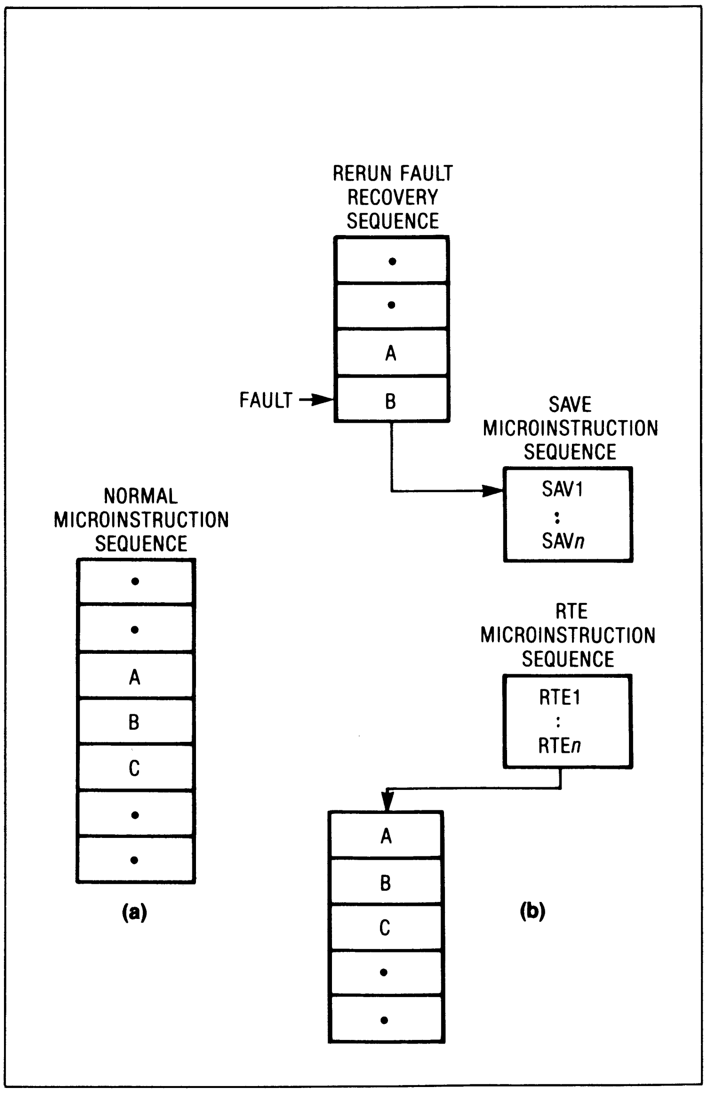

RERUN FAULT RECOVERY SEQUENCE
A
B
FAULT
SAVE MICROINSTRUCTION SEQUENCE
SAV1
:
SAVn
NORMAL MICROINSTRUCTION SEQUENCE
A
B
C
RTE MICROINSTRUCTION SEQUENCE
RTE1
:
RTEn
A
B
C
(a)
(b)

Figure 6. Restart method microflow—normal microinstruction sequence (a); rerun fault recovery sequence (b).

advantages and disadvantages. The MC68010 was implemented with the less commonly used of the two—the instruction continuation method—for reasons that we will explain later in this article.

Instruction restart method. The most commonly used method of virtual memory support is the restart method. In this method, the instruction in which the fault occurred is restarted from the beginning, after the exception handler has completed all activity associated with the correction of the fault. This is done regardless of the stage of the instruction the processor had reached when the fault was recognized. Figure 6 illustrates the flow of the microcode for a faulted routine with the restart method. Under normal conditions, microinstructions A, B, and C will execute consecutively (Figure 6a). If microinstruction B has a fault associated with it (Figure 6b), the processor will execute A and B but will then be interrupted by the save routine. It will then save the state and execute whatever handling routine is appropriate. This handling routine should conclude with a return from exception (RTE) instruction. This return will restore the state and then restart the faulted instruction over again at microinstruction A. Thus, in this scheme there will always be an A, B, C sequence of microinstruction flow.

The restart method implies that the processor is able to restore or reconstruct the state of the machine as it existed at the beginning of the instruction. When a user-visible resource is used as both a source and destination for data within one instruction, this method becomes quite complicated. One example of this problem occurs with extended precision arithmetic operations, another with autoincrement/autodecrement addressing modes. In an extended arithmetic operation, a bit of the status register indicates whether a previous carry or borrow should be considered in the current calculation. The instruction will in turn set that same bit to indicate whether there was a carry or a borrow result in the current calculation. If the processor is faulted after this bit of the status register is updated, the original value must be restored before the instruction can be restarted. This is evident in the sample instruction presented in Figure 4, where an autodecrement addressing mode is used. It is generally desirable to update the decremented address register while the operand fetch is taking place (as shown in the flowchart). Furthermore, if both source and destination addressing modes use the same address register, then the updated value of the register must be used in the address calculation of the destination address. If a fault is detected later in that instruction, the register must be restored to its original value before the instruction can be re-executed.

There are three methods commonly used to deal with this problem. First, the processor can prevent any user-visible resource from being altered until the instruction is completely executed, when it can be assured that no fault will occur. Second, the processor can maintain copies of resources as they are altered. These copies will contain the original value of the resource at the beginning of the instruction. If the instruction is faulted, the copies are used to restore the user state. Third, any updating of a user-visible resource can be tagged to indicate which resource was altered and how the original data can be restored.

The level of difficulty involved in implementing one of these methods depends on how complete the instruction set is and how orthogonal it and its addressing modes are. If the number of situations where resource conflicts can occur are limited, the complexity of providing a comprehensive solution is manageable. However, if a processor with a powerful instruction set is used, extensive resources could be required. Unfortunately, it is too often deemed acceptable in the microprocessor community to provide a limited solution to the problem either by identifying a limited set of instructions that can be provided to the user as virtual memory instructions, or by expecting the user to conduct all of the repairs needed to reconstruct the internal state of the machine. In the first case, one can define instructions in which there can be no resource conflicts and advise the user to employ them to ensure that the page to be accessed resides in physical memory. For instructions in which there *can* be resource conflicts, no reliable recovery can be made, and hence their use is somewhat limited. The second solution is also unattractive, since it requires the user to evaluate the saved state of the processor to determine if the faulted instruction presented a potential resource conflict and, if it did, to make the needed corrections. In either case the processor simply restores the state of the machine and executes the faulted instruction again, regardless of the validity of the restored state.

There is another class of problems associated with the restart method that, although very subtle and in some cases improbable, still presents situations from which the processor cannot be predictably recovered. The first situation involves an access made to an I/O device. It is felt that in most systems there are significant advantages to memory-mapped I/O, since any general instruction is capable of communicating with an I/O device. If, however, there is a fault of any kind after the access to the I/O device, and if there is successful restoration after the fault, the processor proceeds to refetch or rewrite data to or from the I/O device. This can be catastrophic, since the status of many I/O devices is altered as the result of the access. Thus, the second access to the device can result in incorrect data being transferred. Another problem can occur when an operand is transferred from memory to memory and the operands overlap. For example, if a long-word move from an address register indirect to an address register indirect [(MOVE.L (An), (Am)] is executed with address registers An and Am pointing as shown in Figure 7, the instruction moves the long word X:Y into location Y:Z. This is done by reading the words X and Y, then writing X to location Y and Y to location Z. However, when the write to location Z takes place, a fault can occur. In the restart method, when the processor executes the instruction the second time, it reads X from location X but also reads X from location Y, since location Y has been updated by the first partial execution of the instruction. Thus, the result of the instruction is that both locations Y and Z contain X.

Instruction continuation method. The second method of virtual memory implementation, the instruction continuation method, provides an attractive alternative to the restart method. In the continuation method, the entire non-user-visible state of the machine is saved when an access fault is detected. On completion of the fault handler routine, the processor is allowed to resume instruction processing at the same location within the instruction at which execution was suspended by the fault. This action occurs regardless of the location within the instruction at which the access fault occurs.

In the example given in Figure 8, activity within the processor is suspended in microinstruction B until the completion of the data access. If that access is terminated unsuccessfully, the processor saves the internal state of the machine and enters the exception handler routine. The operating system is free to make any repairs to the system that are necessary. After these repairs are complete, the operating system signals the processor to restore the state and resume normal execution. The state is reloaded into the machine and the access that caused the fault is repeated. When the access is successfully completed, the processor resumes execution with microinstruction C. Thus, the continuation method is analogous to an interrupt operation at the microinstruction level.

There are several problems associated with the continuation method of virtual memory support; they involve

- instructions that require execution without interruption,
- silicon resources that must be provided to support the saving and restoring of the internal state, and
- the time that is required for such saving and restoring.

Moreover, the greater complexity of the continuation method makes any virtual memory implementation that uses it more vulnerable to design errors.

In general, continuation provides a more natural method of virtual memory support. It is less disruptive

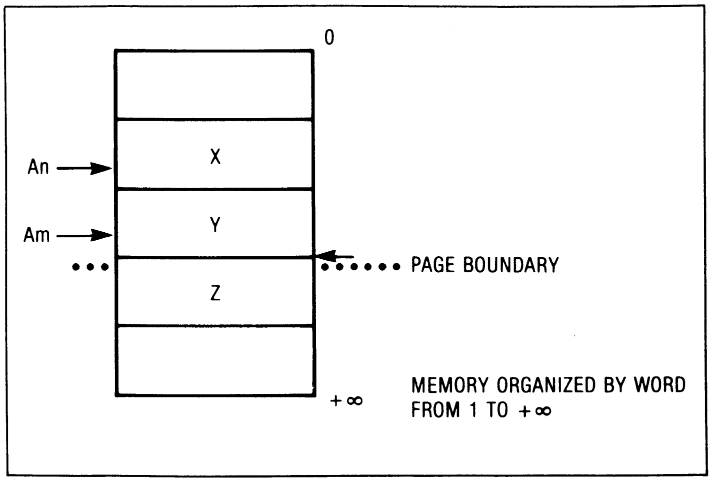


Figure 7. The operand overlap problem.

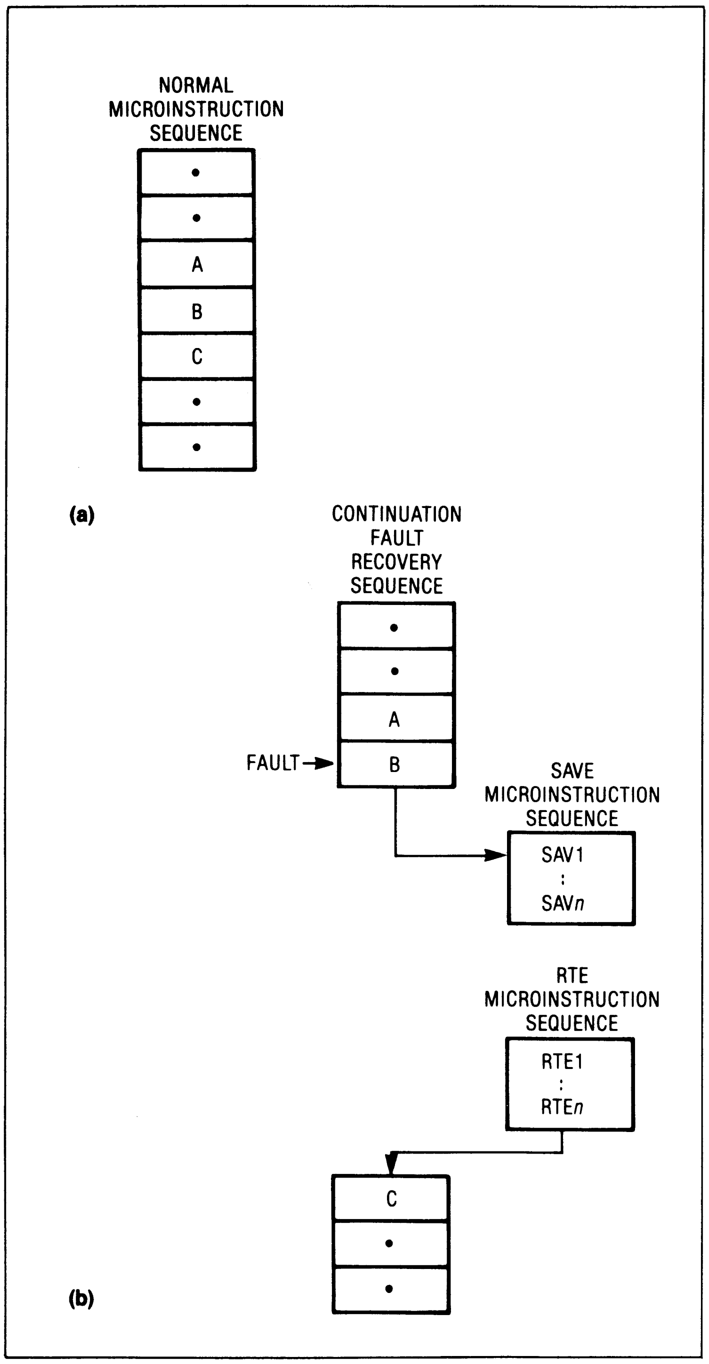

NORMAL MICROINSTRUCTION SEQUENCE
•
•
A
B
C
•
•
(a)
CONTINUATION FAULT RECOVERY SEQUENCE
•
•
A
FAULT
B
SAVE MICROINSTRUCTION SEQUENCE
SAV1
:
SAVn
RTE MICROINSTRUCTION SEQUENCE
RTE1
:
RTEn
C
•
•
(b)

Figure 8. Continuation method microflow—normal microinstruction sequence (a); continuation fault recovery sequence (b).

to simply suspend operations or to interrupt execution at the microinstruction level until the fault has been corrected. The suspended program can then resume execution from the point at which execution was originally interrupted. Sometimes the instruction needs to be executed without interruptions, however, and in this case the continuation method is not naturally suited to providing the required support. An example of this type of instruction is the test and set (TAS) of the M68000 instruction set. The TAS instruction provides the user with a primitive operation which can be used to implement resource protection mechanisms via semaphores or other means. For the TAS instruction, the processor must provide an uninterruptible read-modify-write sequence during which the bus cannot be arbitrated away. If a fault occurs on the write portion of the access, it is necessary to reinitiate the entire read-modify-write cycle rather than simply rerun the write portion of the cycle. In this case, there must be some mechanism to force the entire read-modify-write cycle to be reinitiated.

The continuation method as implemented on the MC68010 requires that the entire internal state of the processor be saved when an access fault occurs. Because of the complexity of the MC68010, this state comprises a large number of temporary execution-unit registers, sequencer state registers, and control latches. For the state to be saved, the information about it must be stored internally. In addition, there must be access paths from the state information, and there must be the control needed to access this information. Similarly, additional control logic is required to perform the reloading of the state. In terms of silicon area, the additional resources require a 22 percent increase over the MC68000 in total area. Figure 9 is a labelled die photograph of the MC68010; the crosshatched areas indicate the approximate increase in area due to virtual memory support. While 22 percent is a significant amount of growth, the most critical aspect of product design—development time—was greatly reduced, since most of the increase involved areas consisting of regular structures.[5,6]

Another expense associated with the larger stacked internal state is the additional time it takes to read and write the state during the save and restore operations. While the time required to execute the save and restore operations is shorter than the normal execution time for some instructions, any time required to perform these functions degrades overall performance. This is a particularly important concern because once the state has been saved in a multiprogrammed environment, a different program can be dispatched to the idle processor. This allows the time required for the page replacement and other system repairs to be available to the system for other processing tasks. Another problem associated with the save time is that it contributes directly to the interrupt latency. Because any instruction can cause a fault, the interrupt latency must be calculated by adding the time required to save the state to the time required to execute the longest instruction. In the MC68010, the interrupt latency could have increased by nearly 50 percent over the interrupt latency of the MC68000. This did not occur due to other optimizations made in the design. Specifically, the latency associated with normal instruction processing was reduced by 50 percent, from 284 clock cycles to 148. The resulting normal instruction latency is equivalent to the time required to save the state—132 clock cycles. Thus, we were able to avoid the latency problem inherent in saving the state. The MC68010 interrupt latency is essentially the same as that of the MC68000.

One of the most difficult problems associated with the continuation method is the increased vulnerability to microcode errors it causes. Since an access fault can occur at almost any point within the execution of an instruction, the processor must suspend and then resume execution from that point as if the fault had never occurred. This implies a much more detailed prediction of the state of the machine than that performed in the MC68000. In order to simplify the verification of correct execution, the microinstructions associated with an instruction can be categorized into general microinstruction routines. The categories, which are also present in the MC68000, are shown in Figure 10. All of the microinstructions are attributed to one of three categories—either they fetch an immediate operand, evaluate an effective address and fetch the operand, or do the operation fundamental to the instruction. These routines perform all or part of the activity associated with an instruction. A simple instruction may be composed of only one routine while a more complex instruction can be composed of several functional routines. Let us examine variations of the ADD instruction (Figure 10). Breaking instructions into these routines makes it possible to define a series of boundary conditions. These boundary conditions define the state of the machine at the beginning of the functional routine. With these simplified state definitions, the routines can be verified by confirming that the entrance boundary conditions are satisfied, the functionality of the routine is correct, and the exit boundary conditions are satisfied. Similarly, it is possible to verify the correctness of instructions by checking the functionality of the routines that compose the instruction and by verifying that the boundary conditions synchronize correctly. One of the difficulties implicit in the continuation method is that when the processor is suspended and then restored, its state must reflect the exact conditions that existed before the fault. In the continuation environment, the steps taken to verify the validity of operations must be more rigorous.

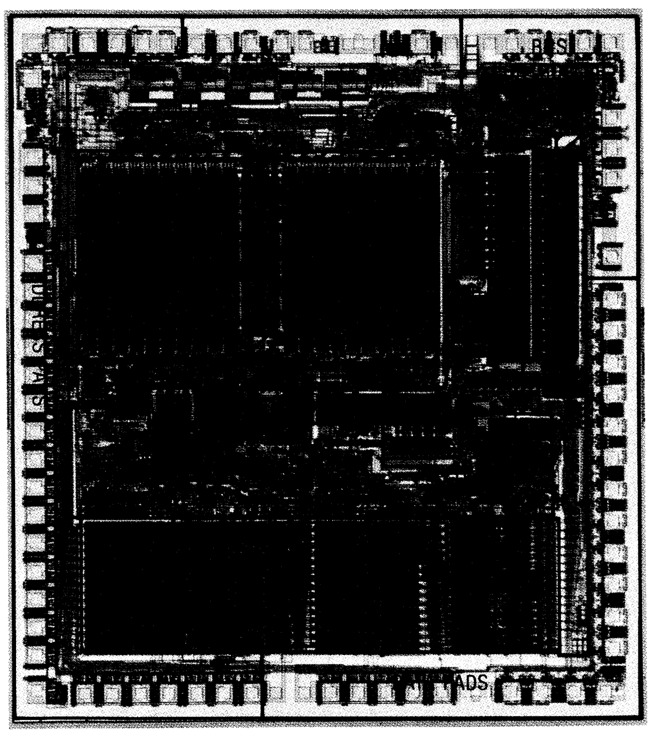


Figure 9. The MC68010—the crosshatched areas indicate circuitry added to support virtual memory.

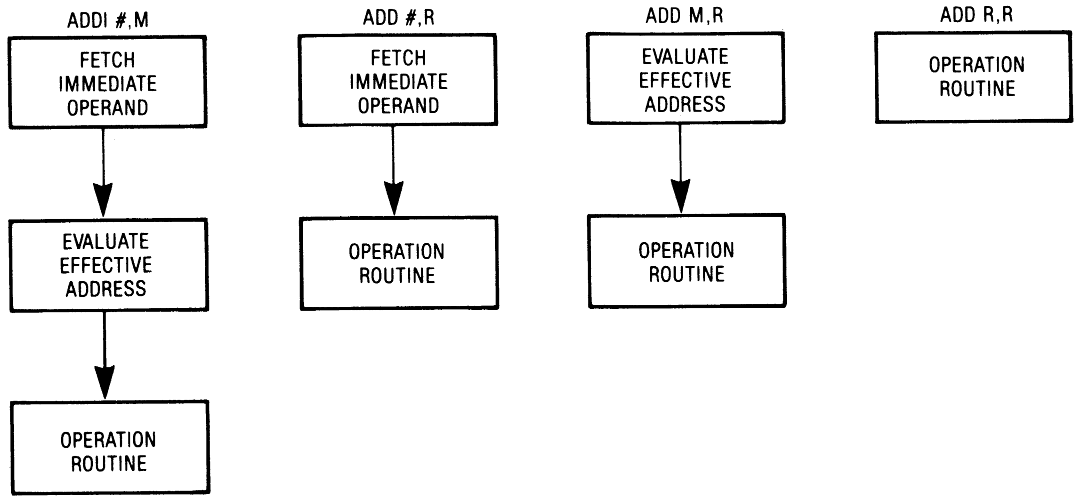


Figure 10. Decomposition of instructions into functional routines.

An example of this difficulty is an error that was found on one of the initial (preproduction) versions of the MC68010 processor. Figure 11 will be used to illustrate this problem. Figure 11a shows the normal processing of two instructions. These instructions are composed of microinstructions A, B, and C for one instruction and D, E, and F for the next. In Figure 11b, examples of the same two instructions are given, but this time with an interrupt being recognized during the first instruction. This interrupt is acted on at the boundary between the two instructions. At the conclusion of the first instruction (at microinstruction C), the interrupt microinstructions are executed if the interrupt has been recognized internally during microinstruction B. After all of the operations associated with the interrupt are completed, execution of the second instruction commences.

The problem was encountered when a fault occurred in microinstruction C after an interrupt had been recognized in microinstruction B. Figure 11c shows this situation as it should have been handled. Because of internal piping, the microinstruction sequencer must know the next microinstruction to be executed during the current microinstruction. This is state information which must be saved to allow the processor to return to the location from which activity was suspended. In this case, the next microinstruction that was scheduled when the fault occurred was the first microinstruction of the interrupt routine. Unfortunately, the interrupt that caused the sequencer to select the interrupt handler had been resolved immediately after the save routine, so that when the interrupt handler was executed again, a superfluous interrupt-acknowledge cycle occurred. To solve the problem, the MC68010 had to internally recognize this situation during the return operation and take compensating actions. It should be clear, however, that totally protecting the user from inconsistencies requires strenuous efforts. In this case, the boundary conditions did not correctly mesh with the conditions for the beginning of a new instruction; that is, the boundary conditions were not properly defined to prevent this condition from occurring.

When examining the difficulties associated with the continuation method, one finds few situations in which the operation of the machine is altered at all. The difficulties are instead related to the resources required to implement the method, the execution time associated with a larger state, and the difficulty of verifying the correctness of the machine. If resources are available to solve the problems described above, one can make many simple enhancements to the continuation method that will provide services of practical value to the user. These will be described in detail later. In general, the continuation method is much more natural and less disruptive to the flow of the machine.

What is the MC68010?

The Motorola MC68010 is a 16-bit microprocessor with 32-bit registers, an expanded instruction set, and flexible addressing modes. The MC68010 is object-code-compatible with the M68000 family of processors. It offers the following facilities to the user:

- seventeen 32-bit data and address registers,
- 16-megabyte direct addressing range,
- virtual memory/virtual machine support,
- 57 instruction types,
- high-performance looping instructions,
- operations on five main data types,
- memory-mapped I/O, and
- 14 addressing modes.

As shown in the programming models (Figures 1 and 2), the MC68010 offers seventeen 32-bit general-purpose registers, a 32-bit program counter, a 16-bit status register, a 32-bit vector-base register, and two three-bit alternate-function-code registers. Eight of the registers, D0-D7, are considered data registers and can operate on byte (8-bit), word (16-bit), and long-word (32-bit) data. The other nine general-purpose registers, A0-A7,A7', are considered address registers and can be used on word and long-word address operations. Any one of the 17 registers can be used as an index register.

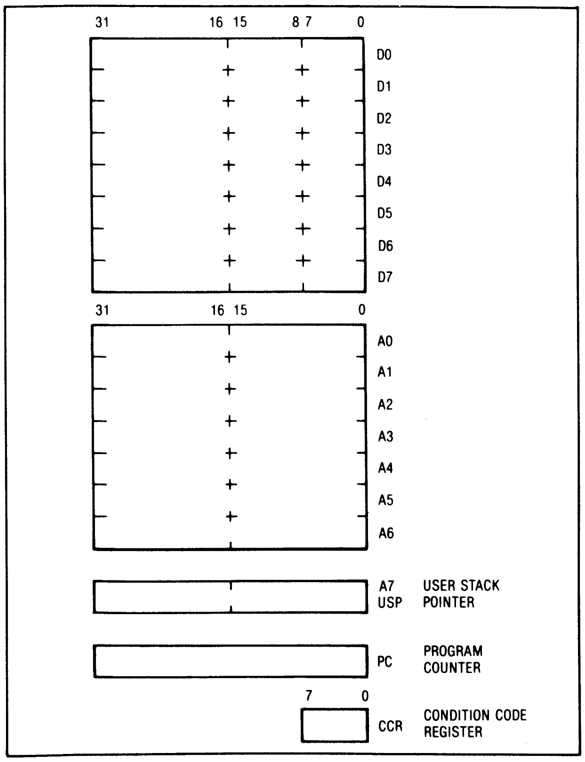


Figure 1. User programming model for the MC68010.

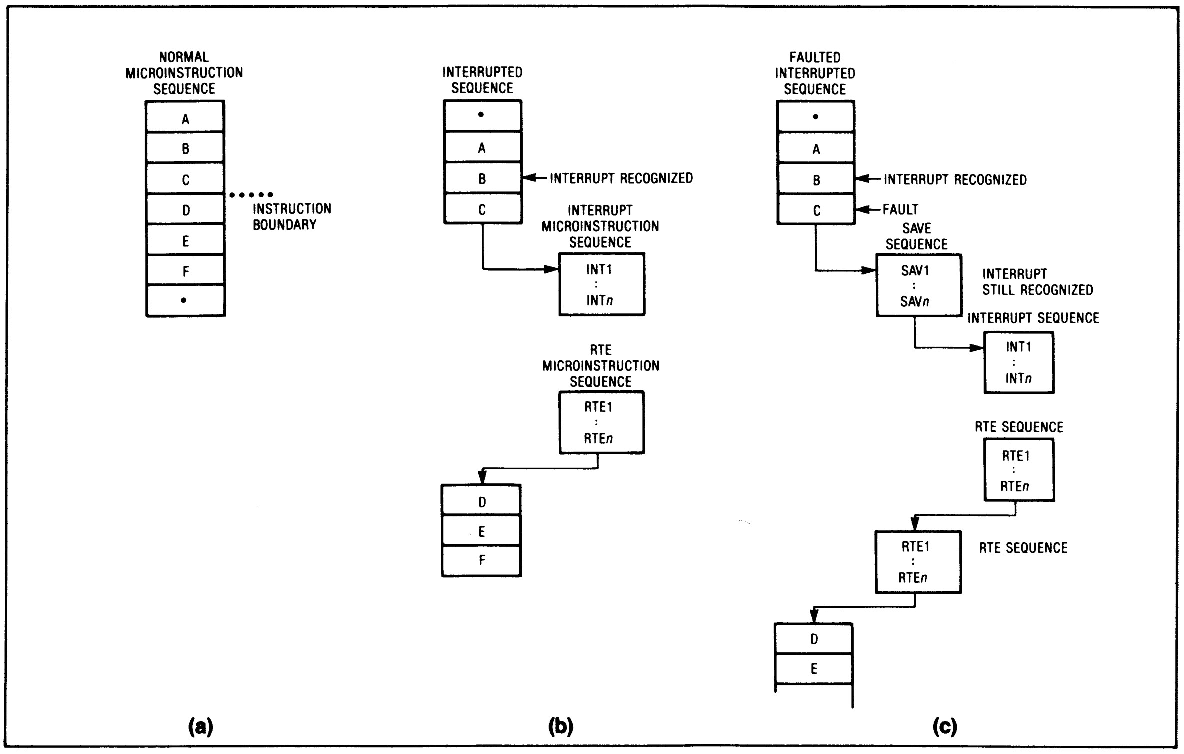


Figure 11. Interrupt processing on the MC68010—normal microinstruction sequence (a); interrupted sequence (b); faulted interrupted sequence (c).

The status register (Figure 3) contains the interrupt mask (eight levels available) as well as the condition codes: extend (X), negative (N), zero (Z), overflow (V), and carry (C). Additional status bits indicate whether the processor is in the trace (T) mode or the supervisor (S) state.

The vector-base register is used to determine the location of the exception vector table in memory. It supports multiple vector tables. The alternate-function-code registers allow the supervisor to access any of the eight address spaces.

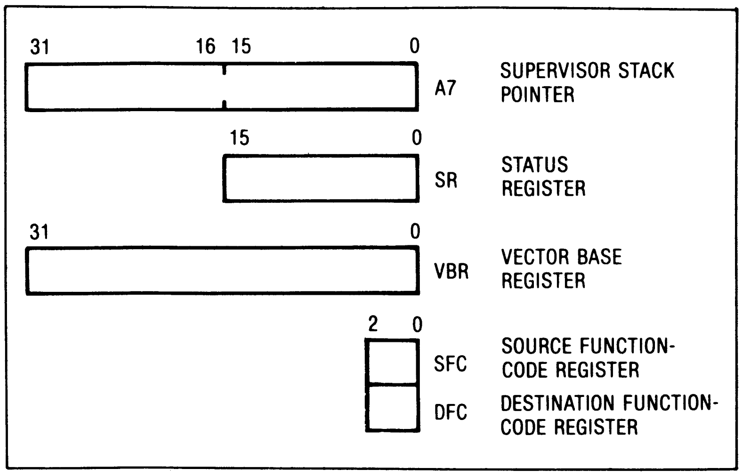


Figure 2. Supervisor programming model for the MC68010.

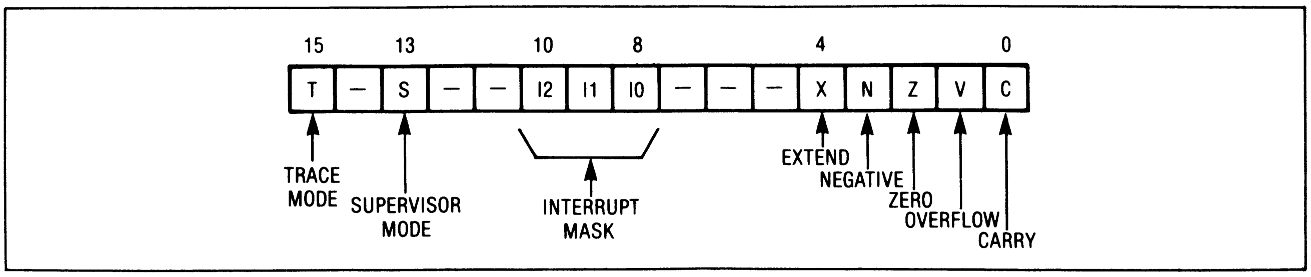


Figure 3. MC68010 status register.

MC68010 virtual memory implementation details

As we explained earlier, we selected the continuation method as the virtual memory implementation method in the MC68010 because of the complexity of the M68000 instruction set. In addition, the continuation method made possible a high degree of fault coverage, which is consistent with the M68000 family exception philosophy. The details of how the continuation method was implemented on the MC68010 can be described in terms of four areas—additional hardware, architectural methods, the save process, and the restore process.

Enhanced internal hardware. Additional hardware resources were added and devoted to the task of saving and restoring the internal state of the machine. These resources include not only the latches and registers used to hold data, but also the control logic used to latch and transfer the data during the save and restore operations.

The saved state consists of 26 words—15 contain execution-unit registers, three save the instruction pipe registers, four hold bus controller information, one consists of the status register, and three contain miscellaneous bits of state information from throughout the processor. To both save this data and preserve information relevant to the faulted access, additional registers are provided to store the address and data associated with the faulted access. The three words of miscellaneous state information are latched so that they can be saved and restored. Additional control logic is provided to interpret the miscellaneous state information, which may have been modified on the stack to ensure proper operation.

Architectural extensions. The MC68000's return from exception (RTE) instruction was expanded so that it can determine the type of exception associated with the stack frame and take the action appropriate for that type. This results in increasing the amount of information stacked during an exception by one word. The additional word contains the stack frame (i.e., the exception type) and the exception vector offset. The addition of the exception vector offset to the stack frame allows generic exception handlers to be used by the operating system software. Figure 12 illustrates the difference between the exception stack frame of the MC68000 and that of the MC68010. By using the general RTE instruction for the machine restore, we maintained compatibility with the MC68000 and yet enhanced the generality and expandability of the instruction.

The execution of the RTE on the MC68010 is very similar to that on the MC68000. The processor reads the status register, program counter, and stack format into the machine. The format word is then evaluated. If the short stack format is present, then the information needed for the return is resident in the machine and normal processing resumes at the address indicated by the

The MC68010 can operate on five basic types of data: bits, BCD digits, bytes, words, and long words. The 14 address modes include six basic types: register direct, register indirect, absolute, program counter relative, immediate, and implied. These are shown in Table 1.

The MC68010 instruction set is shown in Table 2. It readily supports structured high-level languages. Each instruction, with few exceptions, operates on byte, word, and long-word data, and most instructions can use any of the 14 addressing modes. The basic instructions can be combined with the available data types and addressing modes to provide over 1000 total instructions. Furthermore, 33 of the basic instructions can be used in the loop mode with certain addressing modes and the DBcc instruction to provide 230 string, block manipulation, and extended arithmetic operations.

Table 1.
Addressing modes.

Mode	Generation
Register Direct Addressing	
Data Register Direct	EA = Dn
Address Register Direct	EA = An
Absolute Data Addressing	
Absolute Short	EA = (Next Word)
Absolute Long	EA = (Next Two Words)
Program Counter Relative Addressing	
Relative with Offset	EA = (PC) + d_{16}
Relative with Index and Offset	EA = (PC) + (Xn) + d_8
Register Indirect Addressing	
Register Indirect	EA = (An)
Postincrement Register Indirect	EA = (An), An ← An + N
Predecrement Register Indirect	An ← An − N, EA = (An)
Register Indirect with Offset	EA = (An) + d_{16}
Indexed Register Indirect with Offset	EA = (An) + (Xn) + d_8
Immediate Data Addressing	
Immediate	DATA = Next Word(s)
Quick Immediate	Inherent Data
Implied Addressing	
Implied Register	EA = SR, USP, SSP, PC, VBR, SFC, DFC

NOTES:
EA = Effective Address
An = Address Register
Dn = Data Register
Xn = Address or Data Register used as Index Register
SR = Status Register
PC = Program Counter
( ) = Contents of
d_8 = 8-Bit Offset (Displacement)
d_{16} = 16-Bit Offset (Displacement)
N = 1 for byte, 2 for word, and 4 for long word. If An is the stack pointer and the operand size is byte, N = 2 to keep the stack pointer on a word boundary.
← = Replaces

References

1. *MC68010—16-bit Virtual Memory Microprocessor*, Motorola, Inc., Austin, TX, Dec. 1982.

2. E. Stritter and T. Gunter, "A Microprocessor Architecture for a Changing World: The Motorola 68000," *Computer*, Vol. 12, No. 2, Feb. 1979, pp. 43-52.

restored program counter. If the long stack format is present, then the 26 words of state information must be read from the stack and restored to their appropriate location before execution can continue at the point of the exception.

In order to allow for expansion and for verification of the state information, we installed certain protection mechanisms into the restore process. Currently there are only two valid stack formats, \$0 for the normal four-word format, and \$8 for the long 29-word format. Any other formats are identified as illegal by the MC68010 and cause a "format error" exception.

Machine fault and state save process. The state save process begins when a bus fault is detected via assertion of the BERR pin or via a program-generated address error. A flowchart of the save operation is provided in Figure 13. The processor latches and holds information relevant to the faulted cycle, which includes the function code (address space), data access type (read/write), and various internal status information. The processor next saves information resident in the save process hardware by storing it in registers dedicated to this task. Examples of this information include the contents of the address output buffer register and data output buffer registers. This clears a path for external accesses to memory; this cleared path allows the remainder of the internal state to be saved on the stack. After the completion of the state save, exception processing continues, with vector generation followed by execution from the vector location. Detection of another bus fault during the state save process constitutes a double bus fault exception, which causes the processor to halt all processing pending assertion of the external reset pin.

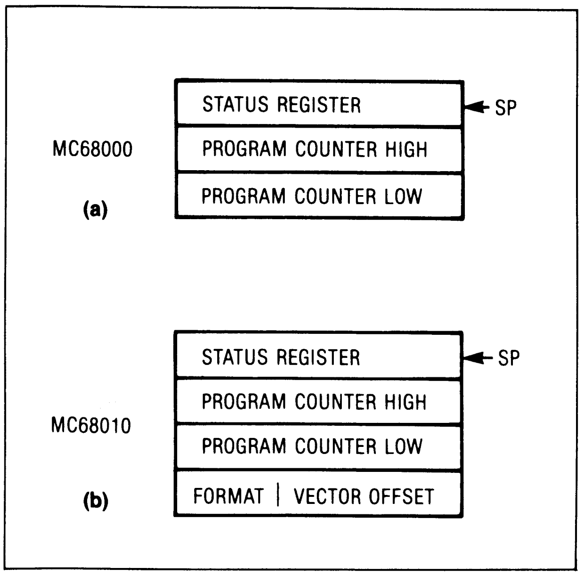


Figure 12. Exception stack frame of the MC68000 (a) and of the MC68010 (b).

Table 2.
Instruction set summary.

Mnemonic	Description
ABCD*	Add Decimal with Extend
ADD*	Add
AND*	Logical And
ASL*	Arithmetic Shift Left
ASR*	Arithmetic Shift Right
B_{CC}	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR*	Clear Operand
CMP*	Compare
DB_{CC}	Decrement and Branch Conditionally
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR*	Exclusive Or
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL*	Logical Shift Left
LSR*	Logical Shift Right
MOVE*	Move Source to Destination
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD*	Negate Decimal with Extend
NEG*	Negate
NOP	No Operation
NOT*	One's Complement
OR*	Logical Or
PEA	Push Effective Address
RESET	Reset External Devices
ROL*	Rotate Left without Extend
ROR*	Rotate Right without Extend
ROXL*	Rotate Left with Extend
ROXR*	Rotate Right with Extend
RTD	Return and Deallocate'
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine
SBCD*	Subtract Decimal with Extend
S_{CC}	Set Conditional
STOP	Stop
SUB*	Subtract
SWAP	Swap Data Register Halves
TAS	Test and Set Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST*	Test
UNLK	Unlink

*Loopable Instructions

Machine restore and return process. After the exception handler has completed any corrections it deems necessary, the processor can be directed to reload its stacked state and resume execution at the point at which the fault occurred. This is initiated by the execution of the enhanced RTE instruction described previously. A flowchart of the RTE process is provided in Figure 14. Before the actual internal restore operation begins, the processor performs checks on the integrity of the restore stack frame. Since the MC68010 is a microcoded design, part of the state information includes the address of the next microinstruction to be executed. This makes necessary a mechanism by which the processor can check the validity of the microinstruction address associated with the bus fault. This mechanism detects the situation in which there are multiple processors with different versions of microcode in the same system. If this situation exists, it is possible for a process to be faulted while on one processor and then redispatched to a different processor with a different set of microcode.

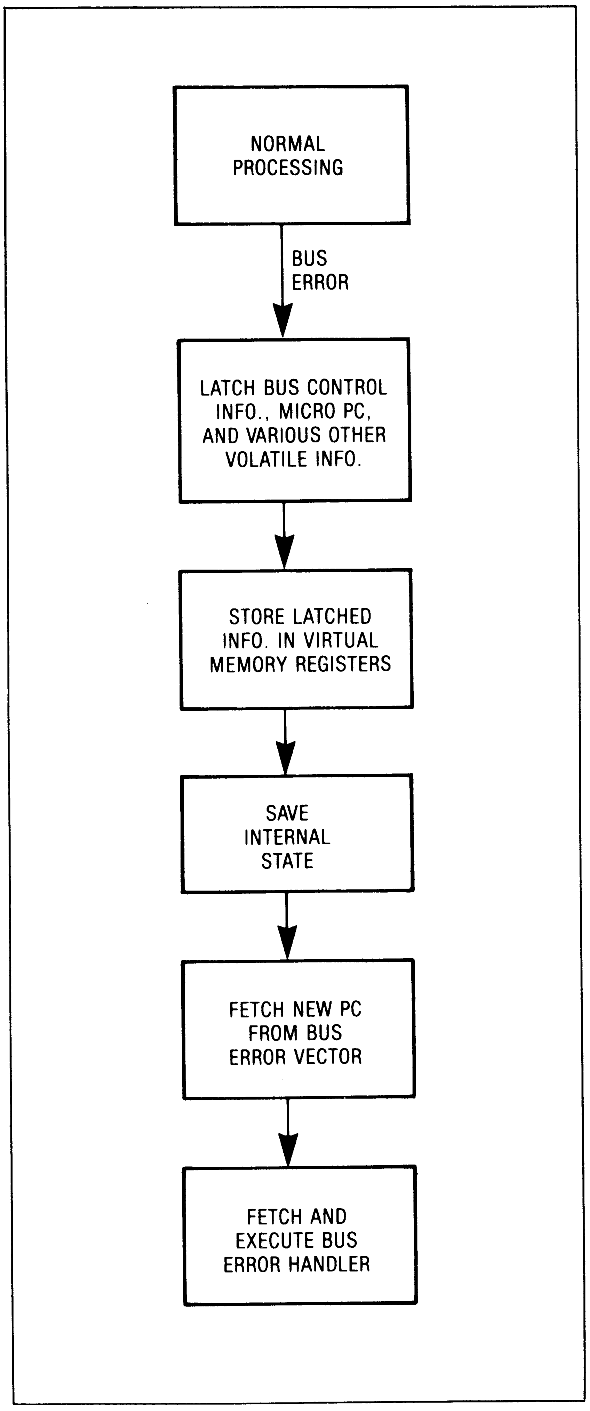

NORMAL PROCESSING
BUS ERROR
LATCH BUS CONTROL INFO., MICRO PC, AND VARIOUS OTHER VOLATILE INFO.
STORE LATCHED INFO. IN VIRTUAL MEMORY REGISTERS
SAVE INTERNAL STATE
FETCH NEW PC FROM BUS ERROR VECTOR
FETCH AND EXECUTE BUS ERROR HANDLER

Figure 13. Save process flow.

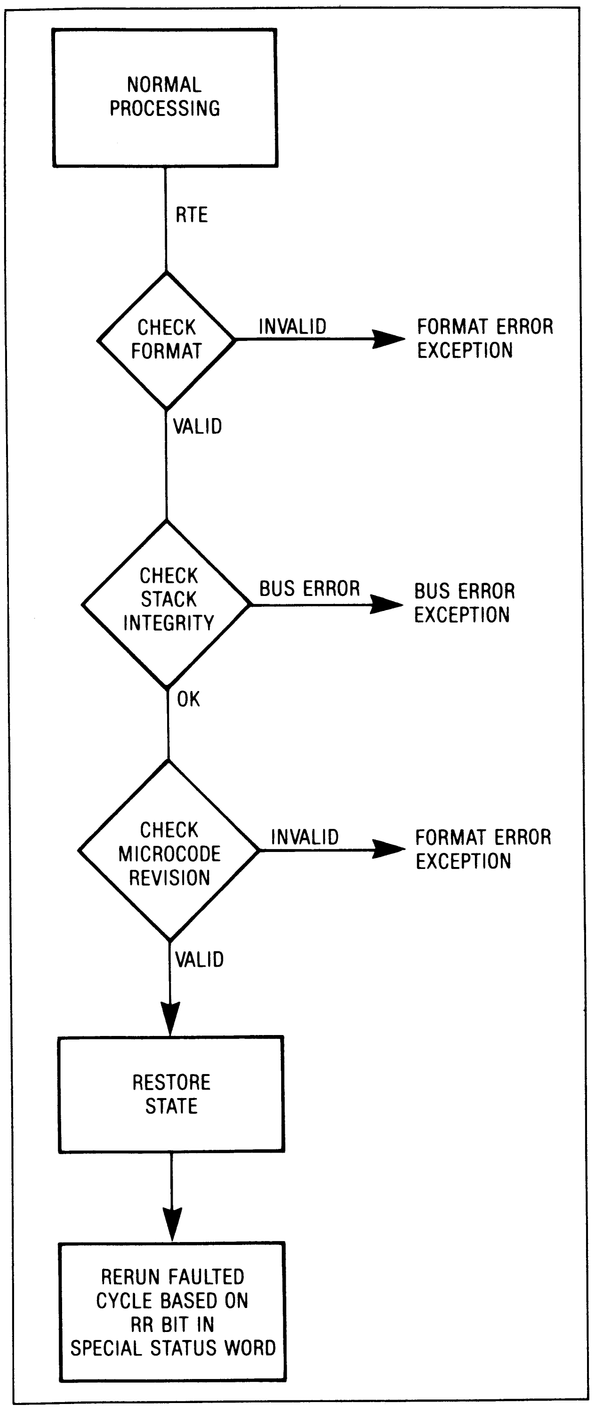
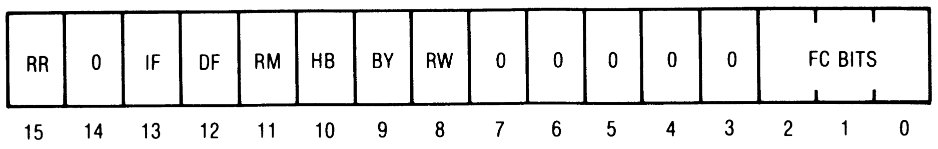


Figure 14. RTE process flow.

Since the microcode is different, the pointer to the next microinstruction will not be valid, and a format error exception must be taken or erroneous execution may occur. The processor then performs a restore check which pertains to the supervisor stack, although in general the integrity of the supervisor stack pointer is the responsibility of the software. Because of the size of the stored machine state, we found it desirable to have the processor ensure that the entire stack frame is resident in physical memory before it is read in, while it is still possible to accept another fault. For this reason the length of the save stack frame is traversed and its residency assured before significant amounts of state information are loaded during the restore process. Once the validity of the stack is determined, the entire 26 words of machine state information are read into the machine and restored to their original locations. A bus fault during the loading of the machine results in a double bus fault, since during the loading the registers that are dedicated to saving the registers associated with the bus activity are not predictably loaded. However, it is possible to have a fault before the stack frame has been traversed, or upon the rerunning of the access that originally caused the fault, without precipitating a double bus fault. Only the faulted access must be completed before the processor is allowed to begin execution of the next microinstruction.

So that the user can handle a number of different situations, he has been given the power to determine how the access that caused the fault will be handled. Fault information is available to the supervisor fault handler in the special status word (Figure 15) that resides on the supervisor stack. This information allows the fault handler to determine the cause of the fault and to take the appropriate corrective and compensatory action. It also includes the nature of the fault, the fault address, and the prospective destinations for the data within the microinstruction. The fault handler also has the ability to signal the processor whether it will correct the faulted access or whether the processor should reattempt it. This is done by means of the rerun bit of the special status word. Situations in which it may be desirable for the operating system to complete the access include operation with misaligned operands or data, operation with I/O faults, or virtual operations (i.e., when the accessed resource does not exist). All of these are readily supported by this mode. The meaning of a software rerun does not limit itself simply to transferring the appropriate data—when the exception handler signals the main processor that it has completed the access, the processor assumes that all aspects of the transfer have been accomplished. In the case of a TAS instruction with an uninterruptible read-modify-write cycle, a software rerun includes the setting of the condition code bits within the status register to reflect the data that were read. One of the limitations of the MC68000 is that it cannot support misaligned data or instructions (address error exception). However, if a misaligned program must be executed, then a software rerun must be performed. The only way in which an address error fault can be corrected on the MC68010 is through a software rerun or through a modification of the fault address on the stack. While it is certainly possible to alter the address that caused the fault, there are few situations in which this is appropriate. If a software rerun is not made, the processor restores the state and attempts to make the same access that previously caused the fault. The access will fault again with the same results.

Once the state of the machine is restored and the access is completed, either by the user or by the machine, the processor is permitted to continue execution at the microinstruction following the faulted microinstruction. Note that if the rerunning of the access is left to the processor, it is possible for that



RR: RERUN FLAG; 0=PROCESSOR RERUN (DEFAULT), 1=SOFTWARE RERUN
IF: INSTRUCTION FETCH TO THE INSTRUCTION INPUT BUFFER
DF: DATA FETCH TO THE DATA INPUT BUFFER
RM: READ-MODIFY-WRITE CYCLE
HB: TRANSFER INTO/OUT OF HIGH BYTE OF BUFFER
BY: BYTE TRANSFER FLAG
RW: READ/WRITE FLAG
FC: FUNCTION CODE OF THE FAULTED ACCESS
0: RESERVED BY MOTOROLA; READS ZERO

Figure 15. The special status word.

access to cause another bus fault. Thus, if the problem that first caused the fault is not corrected, and the fault handler signals to the processor that the machine is to complete the access, a continuous fault-loop effect occurs. During this loop, the stack frame occupies the same location; thus, such a loop does not cause the stack to grow.

MC68010 facilities

Virtual machine operation. The MC68010 provides the mechanisms needed to implement a virtual machine environment in which any degree of emulation is supported. This is achieved in large part by the virtual memory mechanisms described above. Virtual I/O, for instance, is readily achieved by defining a memory area as an I/O device which is not physically resident. When an access is made to that address, an access fault occurs. The fault address can then be evaluated by the operating system to determine the activity that should take place. After the appropriate action has been taken, a software rerun can be signalled and the RTE executed. Indicating to the processor that the access has been completed makes it possible to provide virtual I/O transfers. This technique can of course be generalized to any other type of virtual activity that the processor requests the operating system to execute.

Performance enhancements. Since some new internal resources had to be added to the processor to support virtual operations, we wanted to apply these resources, whenever possible, to other instructions to improve their performance. The result of these efforts is a small performance improvement, which we have estimated to be about 15 percent for a typical instruction mix. A common criticism of the MC68000 is that it is not optimized for fast block operations. Instructions dedicated to handling block operations, however, carry with them some rather unattractive architectural consequences, as they tend not to fit well into the instruction map and do not have the full range of available address modes. The MC68010 provides perhaps the best solution to the performance/regularity problem by recognizing code sequences in which the block operations are defined and by executing these loops very quickly, with no superfluous instruction accesses.

Several new microprocessors which support virtual memory have been introduced recently, with each providing different degrees of such support. The MC68010, utilizing the instruction continuation method, cleanly and elegantly supports the fault detection/fault correction/program resumption process. The options available as a consequence of the use of continuation method—hardware and software rerun—provide powerful support for various implementations of virtual memory. The continuation method also provides the ability to make any access virtual via the software rerun method of return.

One of the most challenging aspects of any design is trying to provide an elegant general solution to a problem while at the same time ensuring that any exceptions to the general case are also handled appropriately. In the MC68010, this challenge has been met. ■

References

1. Peter Denning, "Virtual Memory," *Computing Surveys,* Vol. 2, No. 3, Sept. 1970, pp. 153-189.
2. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-Level Storage System," *IRE Trans. Electronic Computers,* Vol. EC-11, No. 2, Apr. 1962, pp. 223-235.
3. J. Zolnowsky and N. Tredennick, "Design and Implementation of System Features for the MC68000," *Proc. Compcon Fall 79,* Sept. 1979, pp. 2-9.
4. *MC68000 16-bit Microprocessor User's Manual,* 3rd ed., Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 57-69.
5. Saburo Muroga, *VLSI System Design—When and How to Design Very-Large-Scale Integrated Circuits,* John Wiley and Sons, New York, 1982, pp. 417-421.
6. E. Stritter and N. Tredennick, "Microprogrammed Implementation of a Single-Chip Microprocessor," *Proc. 11th Ann. Workshop on Microprogramming* (Micro-11), Nov. 1978, pp. 8-16.

Douglas MacGregor defined the control structures and wrote the microcode for the MC68010 and the MC68020. He enjoys studying Japanese language and culture as well as reading Farley Mowat. He served six years in the Navy, obtaining some direction in life, while completing a BA in history and Asian studies at night. After evaluating the job market, he obtained an MS in computer science from the University of Illinois, from which he went to Motorola's Microprocessor Design Group in Austin, Texas.

David S. Mothersole is project manager of the MC68020 microprocessor systems design group. He has been involved with the definition of the M68000 architecture since coming to Motorola in November of 1978. His areas of research include computer architecture and microprocessor bus structures. A member of the IEEE, he holds a BS and MS in electrical engineering from the University of Texas.

The authors' address is Microprocessor Systems Design Group, Mail Drop M2, Motorola MOS Integrated Circuits Division, 3501 Ed Bluestein Blvd., Austin, TX 78721.

A16642-1 PRINTED IN USA 12-84 IMPERIAL LITHO C27123 10,000 AN209