MOTOROLA
SEMICONDUCTOR
ENGINEERING BULLETIN

Order this document
by EB162/D

EB162

Programming Tips

This document includes the following topics:

	Page Numbers
•Single-Step Instruction Execution on the MC88110	1–6
•Efficient Implementation of Semaphores on the MC88110	6–9
•Efficient Implementation of Breakpoints on the MC88110	9–12
•Initialization Code for the MC88110	12–16

SINGLE-STEP INSTRUCTION EXECUTION ON THE MC88110

Single-step instruction execution implies that processor control is returned to a user-defined process after each instruction is executed from a target instruction stream. Single-stepping the MC88110 can be a useful tool for software debugging purposes and system state diagnostics. For example, the processor state can be examined after execution of each instruction from a targeted instruction stream.

The MC88110 provides hardware support for single-step instruction execution through bit 23, the trace bit (**TRC**), of the processor status register (**PSR**). If the **TRC** and serialize (**SER**) bits are set by software, the MC88110 takes the instruction trace exception (vector offset 0x78) after execution of a single instruction. This article assumes that the reader has some knowledge of the exception model as implemented on the MC88110.

There are three distinct components that make up the entire algorithm of single-stepping the MC88110. The control function, the target instruction stream, and the instruction trace exception handler each contribute to the overall process, as illustrated in Figure 1.

To single-step an instruction, place the address of the target instruction in the exception-time executing instruction pointer register (**EXIP**) and set the **TRC** and **SER** bits in the exception-time processor status register (**EPSR**). At this point, executing an **rte** instruction transfers program control to the first instruction in a target instruction stream. After the first instruction of the target instruction stream has been executed to completion, program control is transferred to the instruction trace exception handler (vector offset 0x78).

This document contains information on a new product. Specifications and information herein are subject to change without notice.

© MOTOROLA INC., 1992

9/92

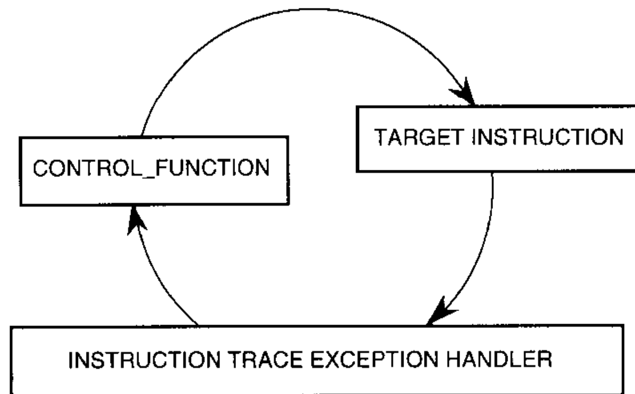


Figure 1. Components of the Single-Stepping Algorithm

The following paragraphs further explain the control function, the target instruction stream, and the instruction trace exception handler, all of which make up a complete single-stepping software envelope.

THE CONTROL FUNCTION

The control function is configured to execute a single instruction from the target instruction stream, thus, it is called once for every instruction to be single-stepped. The control function initiailizes the MC88110 to execute a single instruction at a specified address then begins execution at the instruction trace exception handler. The control function requires two predefined data structures held in main memory, each of which is capable of holding the entire programming model of the MC88110. The first data structure (referred to as host_model) contains the programming model in which the control function executes. The second data structure (referred to as target_model) contains the programming model in which the target instruction stream executes.

The control function should first save the current programming model into the host_model data structure. A trap-not-taken should be executed just before the programming model save occurs in order to ensure an empty history buffer. Note that saving the programming model may corrupt it. For example, if the control function places the address for the host_model data structure into one of the general purpose registers, that register's original contents are overwritten. Control registers **cr16–cr20** can be used as a temporary scratch pad while archiving the programming model. The control function can save the original data from one of the general purpose registers into the scratch pad, before using it as a pointer to the host_model data structure.

Once the current working environment has been saved, the data in target_model can be loaded. Once again, control registers **cr16–cr20** can be used to ensure that the programming model corresponds exactly to the information in target_model.

The control function assumes that the calling routine has already placed the address of the instruction which is to be single-stepped in the **EXIP** entry in the target_model data structure. The first time the control function is called, the calling routine must load the **EXIP** entry in target_model with the address of the target instruction stream. However, as program control passes through the instruction trace exception handler, the

EXIP will be automatically updated and saved back into the target_model for the next time the control function is called to continue single-stepping this target instruction stream.

At this point, the **TRC** and **SER** bits in the **EPSR** are set and an **rte** operation is executed to begin a single-step of the target instruction stream. The **SER** bit must be set along with the **TRC** bit in order for the single-step operation to function properly.

When the **rte** operation is executed, program flow is transferred from the control function, to the target instruction stream, to the instruction trace exception handler, and back to the control function again. Once program flow returns to the control function, the control function will terminate and return program flow to its calling routine.

Figure 2 shows the pseudo-code as well as the logical program flow into and out of the control function.

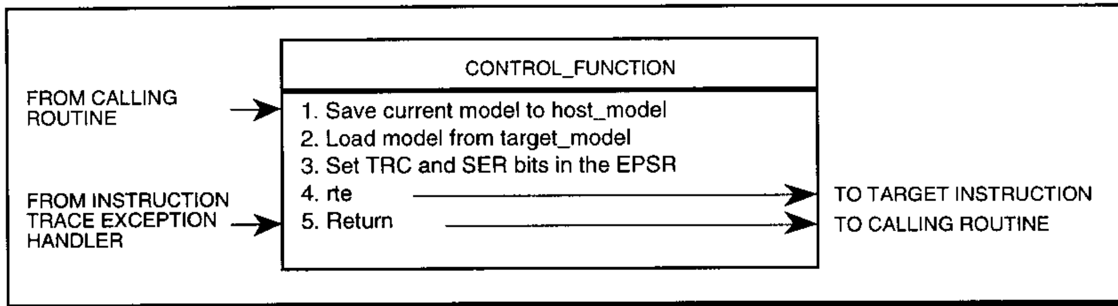


Figure 2. Control Function Pseudo-Code and Program Flow

Notice that program flow first enters the control function from a calling routine and finally exits the control function to that same calling routine. This ensures that the calling routine will see a consistent system stack before and after a call to the control function.

TARGET INSTRUCTION STREAM

Program flow enters the target instruction stream by the execution of an **rte** operation from within the control function. Since the **SER** bit was set before the **rte** in the control function, the instruction pointed to by the **EXIP** will be issued and executed to completion before another instruction in the target instruction stream can be issued. When the target instruction completes execution, the instruction trace exception is recognized. The instruction trace exception is a result of the **TRC** bit being set in the **PSR**.

The MC88110 automatically updates the **EXIP** and the exception-time next instruction pointer register (**ENIP**) during the instruction trace exception recognition. There are three possible cases as to how the **EXIP** and **ENIP** will be updated by the MC88110, as shown in Figure 3. The first possibility is if the instruction which was just executed was not a taken flow-control operation. In this case, the **EXIP** is updated with the next sequential address and the **ENIP** is not updated. The second possibility is if the instruction which was just executed was a taken non-delayed flow-control operation. In this case, the **EXIP** is updated with the target instruction address generated by the flow-control operation and the **ENIP** is not updated. The third possibility is if the instruction which was just executed was a taken delayed flow-control operation. In this case, the **EXIP** is updated with the next sequential address and the **ENIP** is loaded with the target

instruction address generated by the flow-control operation. In addition, bit 0 of the **EXIP** is set to ensure that the flow-control transfers to the address contained in the **ENIP** after the instruction pointed to by the **EXIP** is executed.

When the instruction trace exception is recognized, the MC88110 automatically updates the **EXIP** and **ENIP** according to the following flowchart.

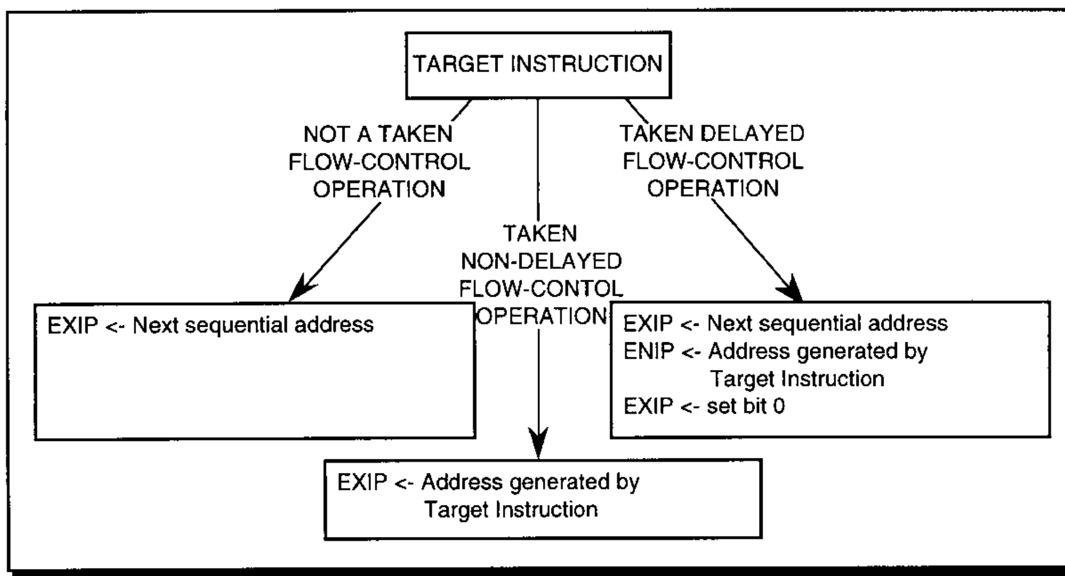


Figure 3. EXIP and ENIP Update Possibilities

The instruction trace exception handler must properly save these values into the target_model data structure. Note that no software is required for the updates made to the **EXIP** and **ENIP** as a result of the instruction trace exception. These updates are performed by hardware and are automatic.

INSTRUCTION TRACE EXCEPTION HANDLER

Program flow is transferred to the instruction trace exception handler after a single instruction has been executed from the target instruction stream. The instruction trace exception handler should first clear the **TRC** and **SER** bits in the **EPSR**. Next, the entire programming model should be saved to the target_model as soon as possible. This preserves the programming model in which the target instruction stream is executing. In addition, this saves the **EXIP** and **ENIP** values in the target_model data structure for the next time the control function is called by an external routine.

Once the programming model has been saved into the target_model data structure, the data in host_model can be loaded. This restores the processor to the state that the control function is expecting. As in the control function, control registers **cr16–cr20** can be used as a temporary scratch pad when saving and restoring the programming models.

After the programming models have been properly set, the instruction trace exception handler must load a predefined address into the **EXIP**. This predefined address should be a label located inside the control function. This label should be located just before the exit from the control function back to its calling routine. After the correct address is loaded into the **EXIP**, an **rte** instruction should be executed, transferring program flow back to the control function.

Figure 4 depicts the pseudo-code as well as the logical program flow into and out of the instruction trace exception handler.

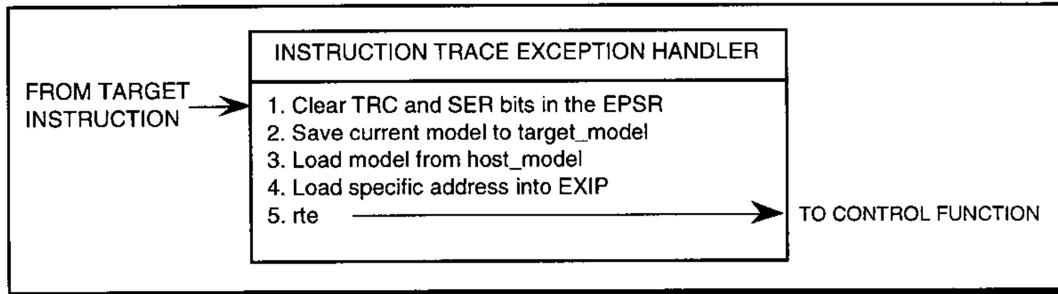


Figure 4. Instruction Trace Exception Handler Pseudo-Code and Program Flow

SINGLE-STEP MODEL OVERVIEW

Figure 5 shows the entire single-stepping model and how program control flows through all three components.

Notice that the **rte** at the end of the instruction trace exception handler simply exits to an exit in the control function. This step could be removed and the **rte** from the instruction trace exception handler could exit to the calling routine. However, certain stack allocations are made at the beginning of the control function and those allocations must be de-allocated before a return to a calling routine. Thus, for consistency, the instruction trace exception handler should exit to an exit in the control function.

It is also important to note that the MC88110 automatically updates the **EXIP** and **ENIP** when the instruction trace exception occurs. This allows an external routine to repeatedly call the control function, thus single-stepping through a targeted code sequence with no overhead. Another benefit of the MC88110 automatically loading the **EXIP** and **ENIP** with the proper values is that the programmer is no longer burdened with handling the special cases of taken flow-control operations.

Hardware support for single-stepping the MC88110 allows for a simple, yet robust debugging mechanism. When implemented correctly, a single-stepping model executes any valid MC88110 operation and returns program control to a user-defined routine.

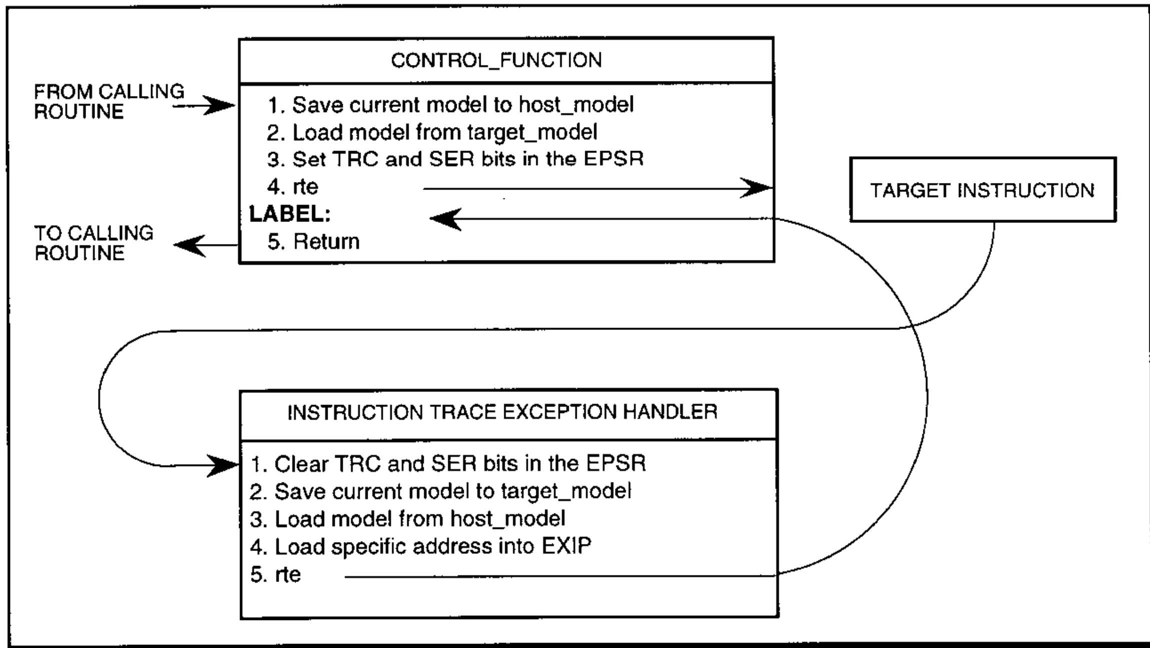


Figure 5. Single-Stepping Model Pseudo-Code and Program Flow

EFFICIENT IMPLEMENTATION OF SEMAPHORES ON THE MC88110

A semaphore can be defined as an apparatus for signaling. In a software environment, a semaphore is often used as a signal to indicate whether a specific resource is being used. In practical terms, a semaphore might simply be a variable in memory that is set when a resource is in use, and clear when that resource is free.

In a system environment with shared resources, a potential owner may need to check a semaphore's value before assuming ownership of a resource. For example, assume that process **A** would like to use an I/O device. Since it is possible that another process is currently using that device, process **A** must check the semaphore that indicates if the device is in use. If the semaphore indicates that the device is free, process **A** sets the semaphore to alert other processes that this resource is now in use. Process **A** is now free to use the resource. Generally, process **A** clears the semaphore when the resource is no longer needed. This enables other processes to take control of the I/O device.

Several subtleties arise after further study of the previous example. For example, is there an efficient way for process **A** to poll the semaphore until the desired resource is available? Once process **A** sees that the resource is available, what assurance is there that, by the time process **A** arbitrates for the system bus and sets the semaphore, another process has not already set the semaphore and assumed ownership? How can process **A** be sure that the setting of the semaphore actually updates main memory rather than only updating the on-chip data cache?

Fortunately, the MC88110 provides an instruction that greatly simplifies the management of semaphores. The **xmem** instruction exchanges the contents of the destination register with a specified memory location.

Functionally, **xmem** is similar to a load followed by a store to the same address, however, **xmem** has specific characteristics that make it ideal for use with semaphores.

Xmem serializes the MC88110 and allows all previous operations to complete (effectively clearing the register scoreboard and all internal pipelines) before the **xmem** executes. This ensures the state of the machine when access to the semaphore is performed. If modification of the semaphore occurs before all internal pipelines are clear, an exception may occur after the semaphore is modified, due to an instruction that was issued prior to the **xmem**.

Xmem can be an atomic operation. In other words, **xmem** is very much like a load followed by a store where control of the system bus is never relinquished during the load-store combination. This ensures that no other process can take control of the system bus and modify the semaphore during the time between the load and store to the semaphore.

The **xmem** instruction is a cache-inhibited memory update operation. When the **xmem** operation is used, the value of the semaphore in main memory will be updated while the value that might be held in the on-chip data cache will remain the same. However, if a cache-inhibited access hits in the primary cache, the line is marked invalid and copied back to memory, if modified.

Since the **xmem** instruction exchanges the contents of the register and memory, the value of the semaphore just before the memory modification, can be checked. This ensures that the semaphore was indeed clear and no other process could have possibly assumed ownership of the resource.

Now that the **xmem** operation has been described, the previous example can also be described further. Assume that process **A** requires the specified I/O device. Before process **A** can set the semaphore and take ownership of the device, it must check the semaphore to see if another process is currently using the resource. Polling of the semaphore most likely involves repeated accessing of the memory location until the semaphore has been cleared. The **xmem** operation should not be used for polling the semaphore. Executed repeatedly, the **xmem** operation can use a substantial amount of system bus bandwidth, which can lower overall system performance.

An alternative means of testing the semaphore is to use **ld** operations to continuously poll the semaphore. Since the semaphore resides in shared memory, the first **ld** operation accesses main memory and brings the value of the semaphore into the on-chip data cache. Repeated **ld** operations to the semaphore only accesses the on-chip data cache, leaving the system bus free to service other processes (including the process currently using the required resource). Since the semaphore is in shared memory, the hardware supported, cache coherency protocol automatically invalidates the copy of the semaphore in the on-chip data cache when the value of the semaphore in memory changes. At that point, the **ld** operation again accesses main memory to load the current (cleared) value of the semaphore. Process **A** now has notification that the semaphore is clear and the resource is free.

The **xmem** operation can now be used to atomically access and set the semaphore. Once the **xmem** operation has set the semaphore, process **A** can ensure that the semaphore was indeed clear at the time it was set by checking the destination register of the **xmem** instruction. If the destination register of the **xmem** indicates that the semaphore was set at the time process **A** set the semaphore, another process has set the semaphore and assumed control of the resource between process **A**'s **ld** and **xmem** operations. Process **A** must now return to its semaphore poll loop and re-arbitrate for control of the required resource.

There is a hidden advantage of the cache-inhibited characteristic of the **xmem** operation which is revealed in this algorithm. If the **xmem** operation was not cache-inhibited by default, the system could achieve a cache-inhibited access through page descriptors in the MMU. However, the system needs the memory which the **xmem** operation is accessing to be cacheable. Allowing this memory to be cacheable enables the sequence of **ld** operations to access the on-chip data cache rather than using valuable system bus bandwidth.

Following is an example of a test and set sequence for the MC88110:

```
semaphore_test:
    or       r2,r0,0x1                 ; r2 <- flag we will store in semaphore
                                       ; r3 <- address of semaphore
    or       r3,r0,lo16(semaphore_1)   ; initialize lower half of semaphore address
    or.u     r3,r3,hi16(semaphore_1)   ; initialize upper half of semaphore address

test_loop:
    ld       r4,r3,r0                  ; r4 <- value of semaphore
    bcnd     ne0,r4,test_loop          ; if the semaphore is not equal to 0, (the
                                       ; semaphore is still set and the resource is still
                                       ; being used by another process), reload and test
                                       ; the semaphore.

                                       ; If we get this far, the semaphore has tested
                                       ; clear, and we are free to set the semaphore and
                                       ; take ownership of the resource.
    xmem     r2,r3,r0                  ; exchange contents of r2 (our flag) with the
                                       ; contents of the semaphore.  Since the xmem
                                       ; operation will update main memory (as opposed
                                       ; to only updating the value held in cache),
                                       ; we are assured that all other processes
                                       ; will be notified that the resource is
                                       ; being used.
    bcnd     ne0,r2,semaphore_test     ; This is a reassurance test to make sure that
                                       ; the value of the semaphore was clear when
                                       ; we set it.  If the value we loaded from the
                                       ; semaphore is not clear, then another process has
                                       ; already set the semaphore and taken control
                                       ; of the resource in the time between the "ld" and
                                       ; "xmem" operations.  If another process has
                                       ; indeed taken control of the resource, we need
                                       ; to return to the test and re-arbitrate for the
                                       ; required resource.
```

In the above sequence, the semaphore can be tested an arbitrary number of times and set, while only using three memory accesses. The first memory access occurs the first time the **ld** operation is executed. The first **ld** operation brings the value of the semaphore into the on-chip data cache. The following **ld** operations

only access the semaphore value in the on-chip data cache, leaving the system bus free for other processes. The second memory access occurs the last time the **ld** operation is executed. This memory access is the result of a change to the semaphore value in main memory and the snooping hardware invalidating the value of the semaphore in the on-chip data cache. Since the value of the semaphore in the on-chip data cache is invalid, the **ld** operation has to access main memory. The update of the semaphore value in main memory should indicate that the semaphore has been cleared, signaling that the desired resource has been released by another process. At this point the third memory access is the **xmem** operation, which is used to set the value of the semaphore in memory.

EFFICIENT IMPLEMENTATION OF BREAKPOINTS ON THE MC88110

A breakpoint is defined as a point in a program where execution is suspended so that examination of the programming model is possible. There are two distinct types of breakpoints which can be implemented on the MC88110: instruction breakpoints and data breakpoints.

An instruction breakpoint occurs when the address of a specific instruction is used to specify when execution should be suspended. When an instruction at that address is encountered, execution is stopped. Instruction breakpoints are fairly straightforward to implement in software, thus there is little reason for the MC88110 to provide extensive hardware support for this algorithm. The instruction breakpoint must be implemented almost entirely in software on the MC88110. The algorithm for this implementation is discussed in this section.

A data breakpoint occurs when the address of a specific piece of data is used to specify when execution should be suspended. When data at that address is encountered, execution is stopped. Data breakpoints are significantly more complicated to implement in software, thus the MC88110 provides significant hardware support for the implementation of data breakpoints.

This document describes only instruction breakpoints. For additional information, refer to **8.6 Data Breakpoints** in the *MC88110 Second Generation RISC Microprocessor User's Manual.*

BREAKPOINTS OVERVIEW

The goal of the instruction breakpoint is to suspend program execution at a given point in the instruction stream. Therefore, we must assume that there is some sort of control monitor present to provide a user interface and memory modification capabilities. The algorithm also assumes that the user will supply an address for our algorithm to trigger on.

The basic algorithm is to replace the opcode found at the user-supplied address with a trap operation. This trap operation transfers program control to a specified function that restores the proper opcode to the user-supplied address and returns program flow to the control monitor. The following paragraphs describe the data structure and functions needed to successfully implement this algorithm.

DATA STRUCTURES

Only one data structure is needed to implement instruction breakpoints on the MC88110. This data structure, called breakpoint_database, is used to store the necessary information about each breakpoint. Each entry in breakpoint_database must be large enough to hold both an address and an opcode.

Figure 6 illustrates the structure of the breakpoint_database. The address portion of each entry is simply the address of the breakpoint (which is supplied by the user). In addition, the opcode portion of each entry is the opcode found in memory at the address in the address portion. In other words, the opcode is read from user-supplied address.

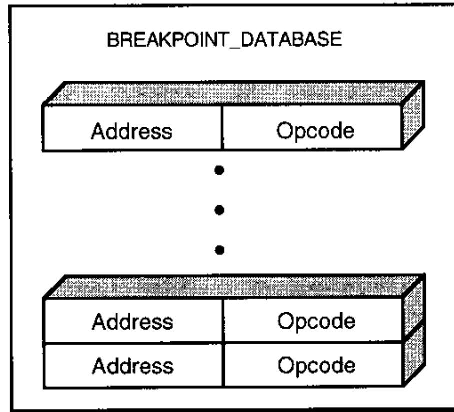


Figure 6. Breakpoint_Database Structure

SUPPORTING FUNCTIONS

There are two functions needed to implement the instruction breakpoint algorithm on the MC88110. The first function needed is called beakpoint_insert. This function is given an address as input from the user. An opcode is then read from memory (from the address supplied as input). At this point, the address and opcode are inserted into breakpoint_database. The pseudo-code for breakpoint_insert follows:

```
breakpoint_insert( input_address )
{
     i = next available entry in          /* Set i to be the next available entry    */
     breakpoint_database                  /* in breakpoint_database.    */

     breakpoint_database[i].address       /* Insert the address of the new           */
     = input_address                      /* breakpoint into breakpoint_database.    */

     breakpoint_database[i].opcode        /* Insert the opcode read from the address */
     = opcode read from input_address     /* supplied by the user into    */
                                          /* breakpoint_database.                    */
}
```

Breakpoint_insert should be called from the monitor program whenever the user has requested the insertion of another breakpoint.

The second function needed to implement the instruction breakpoint algorithm will be called the breakpoint_exception_handler. This function reads an address as input. Breakpoint_exception_handler then searches breakpoint_database for an entry with a matching address portion. When a match is found, the opcode portion of that entry is written to memory. The pseudo-code for breakpoint_exception_handler follows:

```
breakpoint_exception_handler()
{
    while( breakpoint_database[i].address != input_address )
        i++

    memory at input_address =                  /* Once we have exited from the above     */
    breakpoint_database[i].opcode              /* 'while' loop, we know that i points to */
                                               /* the entry in breakpoint_database whose */
                                               /* address portion matches input_address. */
                                               /* It is now time to restore the proper   */
                                               /* opcode to the input_address.           */
}
```

Program flow is transferred to breakpoint_exception_handler as soon as the user-specified breakpoint is encountered. The input_address is not sent to breakpoint_exception_handler as a parameter. Rather, upon taking an exception, the MC88110 automatically places the address of the excepting instruction in the **EXIP** register (control register 4). Breakpoint_exception_handler can obtain input_address by simply reading the **EXIP**. Once breakpoint_exception_handler has completed, program flow should be transferred to the control monitor program.

Note that none of the support functions described above actually traverses the breakpoint_database and writes trap operations to the specified addresses. This task must be performed by the control monitor just before acting on a request by the user to execute a specified instruction stream. In other words, when the user requests that a specified instruction stream be executed, the monitor should write the trap operations to memory before beginning execution of the specified instruction stream.

EXAMPLE OF THE INSTRUCTION BREAKPOINT ALGORITHM

The following list gives a simplified example of the instruction breakpoint algorithm and shows how that algorithm's functionality is integrated into a control monitor.

1. The user has supplied an address to the control monitor and requests that an instruction breakpoint be set.
2. The control monitor calls breakpoint_insert and sends the user supplied address as a parameter.
3. The breakpoint_insert function makes the appropriate entry into the breakpoint_database.
4. The user now requests that the control monitor begin execution of a target instruction stream.
5. The control monitor writes a new opcode (a trap instruction) to each address found in breakpoint_database and transfers program flow to the target instruction stream.
6. If an instruction breakpoint is encountered, program flow is transferred to breakpoint_exception_handler by the execution of a trap instruction.
7. The breakpoint_exception_handler retrieves the address of the trap instruction which was just executed. Upon entering the breakpoint_exception_handler, this address will be found in the **EXIP** register on the MC88110. Breakpoint_database is now traversed and all of the original opcodes are written back to the appropriate memory addresses.
8. Program flow is transferred back to the control monitor.

Once the algorithm for instruction breakpoints has been implemented on a platform, the programming model can be examined at almost any point in an instruction stream.

INITIALIZATION CODE FOR THE MC88110

When the MC88110 is brought out of reset, it begins fetching instructions from address 0x0. It is up to the programmer to place correct initialization code at this location in memory. The terms "initialization code" refer to the code that initializes the programming model of the MC88110 to a known and stable state. This document provides an overview of the initialization code necessary for the MC88110.

The initialization code contained in this document does not put the MC88110 into a "highest performance" state. This initialization code is meant to place the MC88110 into a conservative, "debug" state in which the on-chip caches are disabled and all instructions are issued and executed serially.

INITIALIZATION OF THE PROCESS STATUS REGISTER

The **PSR** is the most important control register on the MC88110. For this reason, the programmer should begin the initialization procedure with this register. Refer to **2.2.4.1.2 Processor Status Register** in the *MC88110 Second Generation RISC Microprocessor User's Manual* for further details on the **PSR**. The following code segment can be used to initialize the **PSR**.

```
Init_PSR:
                                        ; This routine will initialize the PSR to a
                                        ; known and stable state.

    or.u    r2,r0,0xA200                ; 0xA200 goes into the upper 16 bits of r2

    or      r2,r2,0x03E2                ; 0x03E2 goes into the lower 16 bits of r2

    stcr    r2,cr1                      ; PSR <- 0xA20003E2
                                        ;       || |||
                                        ;       || |||_exceptions enabled
                                        ;       || ||  disable external
                                          interrupts,
                                        ;       || ||  enable misaligned access
                                        ;       || ||  exceptions,
                                        ;       || || enable SFU1
                                        ;       ||  ||_enable SFU2,
                                        ;       ||  |   disable SFU3,
                                        ;       ||  |   disable SFU4,
                                        ;       ||  |   disable SFU5
                                        ;       ||  |   disable SFU6,
                                        ;       ||      disable SFU7
                                        ;       ||      disable trace mode
                                        ;       ||_serialize memory instr's,
```

```
;       I unsigned immediate offsets
;        I & constants
;       I
;       I_clear carry bit,
;          serialize all instr's,
;              Big Endian data format,
;              Supervisor Mode
```

INITIALIZATION OF THE INSTRUCTION MMU/CACHE CONTROL REGISTER

The instruction MMU/cache control register (**ICTL**) should follow the **PSR** in the initialization procedure. Refer to **8.9.1.2 Instruction MMU/Cache Control Register** in the *MC88110 Second Generation RISC Microprocessor User's Manual* for further details on the **ICTL**. The following code segment can be used to initialize the **ICTL**.

```
Init_ICTL:
                                        ; This code segment will initialize the
                                        ; ICTL to a known and stable state.

     or      r2,r0,r0                   ; "zero-out" r2.  We will be using r2 as our
                                        ; mechanism to write into the ICTL so we need
                                        ; to begin with zero value in r2.

                                        ; We will be leaving bits 19–25 zero which
                                        ; tells the MC88110 that block size for the
                                        ; instruction BATC will be 512-Kbytes.

     or      r2,r2,0x8000               ; Set the 15th bit in r2.  This will disable
                                        ; "double instruction issue" capabilities on
                                        ; the MC88110

                                        ; Do not modify the 14th bit.  Leaving this
                                        ; bit zero keeps branch prediction disabled.

                                        ; Do not modify the 8th bit.  Leaving this
                                        ; bit zero keeps "instruction cache freeze
                                        ; bank 0" disabled.

                                        ; Do not modify the 7th bit.  Leaving this
                                        ; bit zero keeps "instruction cache freeze
                                        ; bank 1" disabled.

     or      r2,r2,0x40                 ; Set the 6th bit in r2.  Setting this bit
                                        ; will enable hardware table search
                                        ; operations.  We will disable the
                                        ; instruction MMU in the next step, so a
                                        ; hardware table search should not occur
                                        ; anyway.
```

```
                                        ; Do not modify the 5th bit. Leaving this
                                        ; bit zero keeps the instruction MMU
                                        ; disabled.

                                        ; Do not modify the 2nd bit. Leaving this
                                        ; bit zero keeps the target instruction cache
                                        ; disabled. For further information, refer to
                                        ; 9.3.4.2 Target Instruction Cache
                                    ; in the MC88110 Second Generation RISC
                                    ; Microprocessor User's Manual.

                                        ; Do not modify the 0th bit. Leaving this
                                        ; bit zero keeps the instruction cache
                                        ; disabled.

stcr        r2,cr41                     ; Write the constructed r2 into the ICTL
```

INITIALIZATION OF THE DATA MMU/CACHE CONTROL REGISTER

The data MMU/cache control register (**DCTL**) should be initialized immediately after the **ICTL** has been initialized. Refer to **8.9.2.2 Data MMU/Cache Control Register** in the *MC88110 Second Generation RISC Microprocessor User's Manual* for further details on the **DCTL**. The following code segment can be used to initialize the **DCTL** to a known and stable state.

```
Init_DCTL:
                                        ; This code segment will initialize the
                                        ; DCTL to a known and stable state.

    or      r2,r0,r0                    ; "zero-out" r2. We will be using r2 as our
                                        ; mechanism to write into the DCTL so we need
                                        ; to begin with zero value in r2.

                                        ; We will be leaving bits 19–25 zero which
                                        ; tells the MC88110 that block size for the
                                        ; data BATC will be 512-Kbytes.

    or      r2,r2,0x2000                ; Set the 13th bit in r2. This will indicate
                                        ; to the MC88110 that xmem operations should
                                        ; a load followed by a store.

                                        ; Do not modify the 12th bit. Leaving this
                                        ; bit zero keeps decoupled cache accesses
                                    ; disabled.

                                        ; Do not modify the 11th bit. Leaving this
                                        ; bit zero tells the MC88110 to check page or
                                        ; block descriptors to determine whether or
                                        ; not write-through mode will be used. We
                                        ; will be disabling the data cache anyway, so
                                    ; this bit is presently a "don't care".
```

```
                                        ; Do not modify the 10th bit.  Leaving this
                                        ; bit zero keeps data breakpoint register 1
                                        ; disabled.

                                        ; Do not modify the 9th bit.  Leaving this
                                        ; bit zero keeps data breakpoint register 1
                                        ; disabled.

                                        ; Do not modify the 8th bit.  Leaving this
                                        ; bit zero keeps "Data cache freeze bank 0"
                                        ; disabled.

                                        ; Do not modify the 7th bit.  Leaving this
                                        ; bit zero keeps "Data cache freeze bank 1"
                                        ; disabled.

    or          r2,r2,0x40              ; Set the 6th bit in r2.  Setting this bit
                                        ; will enable hardware table search
                                        ; operations.  We be disabling the data MMU
                                ; in the next step, so a hardware table
                                ; search should not occur anyway.

                                        ; Do not modify the 5th bit.  Leaving this
                                        ; bit zero keeps the data MMU disabled.

                                        ; Do not modify the 1st bit.  Leaving this
                                        ; bit zero keeps data snooping disabled.
                                        ; Since the data cache is disabled, there is
                                        ; no reason to enable snooping.

                                        ; Do not modify the 0th bit.  Leaving this
                                        ; bit zero keeps the data cache disabled.

    stcr        r2,cr41                 ; Write the constructed r2 into the DCTL
```

INITIALIZATION OF THE INSTRUCTION AND DATA CACHES

Now that the cache control registers have been initialized, the on-chip caches should be initialized. The MC88110 provides hardware support for flash-invalidation of the entire instruction and data caches. This flash-invalidation is performed through the **ICMD** and **DCMD** registers. Refer to **8.9.1.1 Instruction MMU/Cache/TIC Command Register (ICMD)** and **8.9.2.1 Data MMU/Cache Command Register (DCMD)** in the *MC88110 Second Generation RISC Microprocessor User's Manual* for further details on the **ICMD** and **DCMD** registers. The following code segment can be used to initialize the instruction and data caches to a known state.

```
Init_caches:
                                        ; This code segment initializes the on-chip
                                        ; data and instruction caches to known state.
                                        ; Namely, both caches will be invalidated.

    or          r2,r0,0x1               ; We will use r2 as a temporary register to
                                        ; build a command to invalidate the entire
                                        ; caches.

    stcr        r2,cr25                 ; By writing a single "1" into cr25, we have
                                        ; told the MC88110 to invalidate the entire
                                        ; on-chip instruction cache.

    stcr        r2,cr40                 ; By writing a single "1" into cr40, we have
                                        ; told the MC88110 to invalidate the entire
                                        ; on-chip data cache.
```

INITIALIZATION OF A TEMPORARY SYSTEM STACK

The final step in the initialization sequence is to allocate a block of memory for a temporary system stack space. The programmer should set aside approximately 10 Kbytes of memory to be used as stack space. Initializing the actual memory used for the stack space is not necessary because the MC88110 should only read variables from the stack that it has previously written to. However, the stack must be initialized in the sense that the programmer must inform the MC88110 where the allocated block of memory is. This can be accomplished by placing the highest address in the allocated block of memory in general register **r31**. Programming convention dictates that **r31** be used as a stack pointer on the MC88110. Note that the highest address of the allocated block should be used. This is because **r31** is decremented before an entry is pushed onto the stack. Likewise, after that entry is popped from the stack, **r31** is incremented.

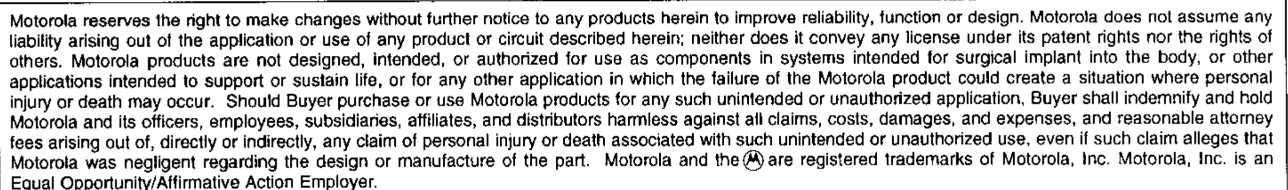
The following instruction sequence properly initializes **r31** to point to the end of an allocated block of memory.

```
or.u    r31,r0,hi16(stack_space)     ; Move the upper 16 bits of the address of
                                     ; the stack space into the upper 16 bits of
                                     ; r31

or      r31,r31,lo16(stack_space)    ; Move the lower 16 bits of the address of
                                     ; the stack space into the lower 16 bits of
                                     ; r31

addu    r31,r31,stack_size           ; Add the size of the stack to r31
                                     ; so that r31 points to the end of the
                                     ; allocated block of memory
```

Although the initialization code for the MC88110 is quite short, it is an important step in powering up the part. Once the MC88110 is placed in a known, stable state by this initialization code, the programmer is free to begin loading additional software into the platform.


Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and the Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

Literature Distribution Centers:

USA: Motorola Literature Distribution: P.O. Box 20912; Phoenix, Arizona 85036.
EUROPE: Motorola Ltd.; European Literature Center; 88 Tanners Drive Blakelands, Milton Keynes, MK14 5BP, England.
JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141 Japan.
ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Center, No. 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.

MOTOROLA