THE MICRO-ARCHITECTURE OF THE ECLIPSE® MV/8000: CONCEPTION AND IMPLEMENTATION

Jonathan S. Blau Charles J. Holland David L. Keating

Data General Corp.
Westboro, Ma. 01580

Abstract

The microcode of the ECLIPSE MV/8000 controls the hardware to emulate an instruction set. In the MV/8000 the micro-architecture is defined and limited by the following constraints: 1) the desire to implement microcode in a limited number of locations; 2) the use of LSI technology; 3) a virtual memory architecture.

This paper will attempt to show how each of these factors contributed to the micro-architecture, to describe that architecture, and to relate some of the more interesting microcoding issues.

The ECLIPSE MV/8000 Instruction Set Architecture

[1] The major objectives of the MV/8000 instruction set architecture are the definition of a 31-bit address space, the manipulation of 32-bit integers, and the achievement of binary compatibilty with existing 16-bit ECLIPSE computers without resorting to a mode bit. A superset of the ECLIPSE C/350 instruction set [2] was created which allows existing ECLIPSE and NOVA® programs to run unmodified on the MV/8000.

In the MV/8000 instruction set, most ECLIPSE C/350 instructions have a counterpart that manipulates 32-bit data and the extended memory address space (31 bits). The four C/350 accumulators are extended from 16 bits to 32 bits. For example, the C/350 instruction ADD, add two accumulators, is supplemented with WADD for 32-bit accumulator addition. The generation of 31-bit logical addresses is accomplished by instructions which have 31-bit displacements and use the same logical address indexing modes as the C/350.

The 4 gigabyte logical address space is divided into eight segments (see figure 1) to facilitate memory protection and sharing. Eight Segment Base Registers (SBRs) hold information for the current process. Each SBR points to an area in main memory that contains logical to physical translations and protection information for that segment.

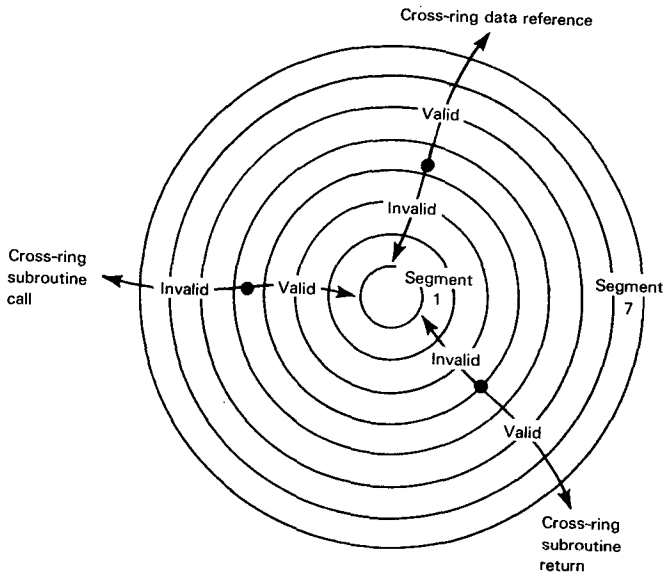

Figure 1 Ring Structure

The 8 segments also establish a hierarcy of protection, called rings of protection. Data references are limited to the current or higher numbered (lower priority) rings. A reference to a lowered number ring causes a protection fault and control is transferred to an operating system error routine. Program flow is limited to the current ring for branches unless a special cross ring instruction is used. Control may be transferred to an inward ring by the use of the XCALL or LCALL instructions, and to an outward ring by the use of the WRTN instruction. An inward transfer, however, is limited to well-defined entry points specified by a gate-way mechanism set up in the inner ring. This gate mechanism protects programs and data in more privileged rings, containing, for example, the operating system code, from illegal entry or modification by a program residing in a less privileged ring. This allows the operating system functions and runtime libraries to be shared but also reside in each program's address space. Additionally, the modification of mapping registers is not necessary for inner ring calls.

0194-1895/80/0000-0090$00.75© 1980 IEEE

Demand paging mechanisms are an integral part of the architecture of the MV/8000. Demand paging is supported by both instruction and hardware state. Hardware provides a means to suspend the currently executing instruction on a logical to physical translation fault. Internal instruction state is saved and then control is passed to a demand paging algorithmm in the most privileged ring, RING0. At the end of page fault resolution, the WDPOP instruction is executed which restores the machine state, and control proceeds as if the translation fault had never occured. Instructions which retrieve information about the reference and modification of pages of memory allow the operating system designer to select his own paging algorithm.

Other features of the instruction set architecture include a fixed and floating point trap facility for preventing loss of precision in arithmetic operations, queue manipulation instructions, DO loops, memory-to-memory instructions, and memory-to-accumulator instructions.

System Overview

It is important to understand that the major design criterion for the MV/8000 hardware was to balance maximum performance with minimum cost. The MV/8000 CPU consists of 7 boards (see figure 2):

1. Arithmetic and Logical Unit (ALU) - Address generation and arithmetic functions including floating point are executed in 220 nanosecond microcycles using 32-bit data paths. There is a barrel shifter to aid in multiple bit shifts and additional hardware to assist multiply, divide, and floating point operations. An 8-bit floating point exponent unit and a 1-bit floating point sign unit reside on the ALU. To achive this on one 15x15 inch board, the use of 2901 4-bit slices for the register file and ALU functions, and PALs® (Programmable Array Logic) for compression of control logic was necessary. [3] [4]

2. Address Translation Unit (ATU) - The 4-gigabyte program address space requires a large logical to physical translation map to exist in main memory. The ATU encaches the most recently used 256 map entries, thus statistically minimizing the times that the translation map is accessed. References to new pages, however, require the hardware to trap; that is, the hardware aborts the current microinstruction and inserts a microroutine that finds the translation in memory before resuming with that aborted microinstruction. The translation tables in memory also contain a resident bit so the software can indicate that a page is out on disk. If this bit is set, a page fault will occur and microcode will save the entire machine state and return control to the operating system. To help the operating system maintain a least recently used (LRU) algorithm for page swapping, the ATU keeps Referenced and Modified bits for each page of physical memory. The ATU also provides execute, read, and write protection checking on memory references, and may initiate a process fatal trap for illegal references.

   Also on the ATU is a general purpose 256 x 32-bit scratchpad memory. The microcode uses this to maintain constants, temporary storage, stack registers, and the Segment Base Registers.

3. The Instruction Processor (IP) keeps its own cache of the machine language instruction stream so that program loops of less than 1K instruction words (1 word = 16 bits) will be totally contained in this cache. Therefore no further memory accesses for instructions are made until the loop is finished. The IP also prefetches instructions up to one cache block (8 words) at a time. It maintains a pipeline for decoding instructions so that instructions are fully decoded to the starting microcode address before the previous instruction has completed execution. Thus, microcode can begin execution of each instruction without the necessity of fetching it from memory first. The IP also provides information on each instruction that allows similiar instructions to share the same microcode.

4. The Microsequencer provides the basic control for the system in the form of a 75-bit microinstuction every 220 ns. [5]

5. The Input/Output Channel (IOC) maintains two maps to support direct I/O to memory operations. This board controls the I/O busses.

6. The Console Controller (CC) monitors the processor while it is running, reporting to the soft console (a microNOVA Board Computer). At power-up it loads the MV/8000 instruction set microcode and may also load special diagnostic microcode if diagnostics are needed. [6]

7. The System Cache keeps the most recently used memory data in fast memory and only accesses main memory in blocks of 8 words (a cache block). It keeps memory bandwidth high by reducing the number of references to main memory. It maintains separate CPU and I/O ports to memory so that there is no contention for memory between the CPU and I/O devices. Each port has a bandwidth of 18.2 megabytes/sec, or 32 bits every 220 ns microcycle.

Minimizing the Amount of Microcode

(Note: we will refer to machine language instructions as macroinstructions to distinguish them from microinstructions in cases where it may not be clear from the context to which instructions we are referring).

The microsequencer has only 4096 lines of microcode to implement over 400 instructions, and microcode space was recognized as a limited resource. As a result, several features were designed into the machine to help share microcode routines.

The microsequencer has powerful sequencing capabilities, including dispatching and microsubroutines via a sixteen level stack. Six microcode flags (1-bit registers) may be set and tested by microcode. Additionally, an entire macroinstruction's microroutine may be used as a microsubroutine since the method of parsing the next macroinstruction is simply a pop of the microstack. If the stack is empty, the starting microaddress is supplied by the IP. If the top of the stack contains a value, it is a microsubroutine return. With this method, the microcode for one macroinstruction might call another macroinstruction as

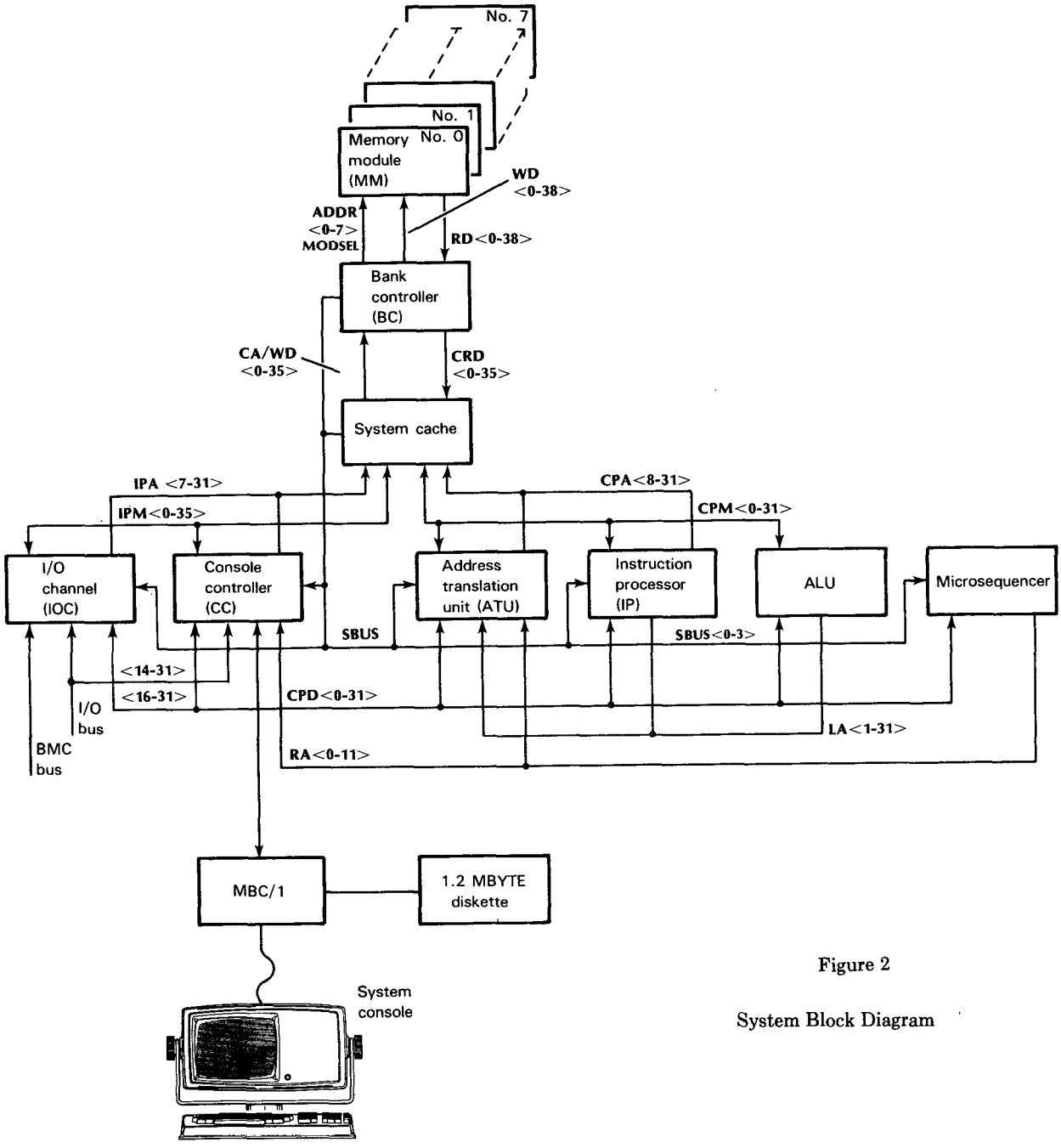

Figure 2

System Block Diagram

a microsubroutine, returning back to the calling microcode when it is finished execution. See [5] for more information about the MV/8000 microsequencer.

For macroinstruction decoding, the IP maintains data in read-only memory, referred to as decode ROMs, which provide a starting microaddress, initialization for two flags that aid execution of the microroutine, and information to cause an effective address calculation (EFA). The initial effective address calculation, if indicated, uses special microcode to perform the indexing and indirection chain resolution that is common to most memory reference instructions. After the EFA calculation, control goes to the specified starting microaddress.

The two flags, the Width and the Address flags, preset control state in the CPU and are also helpful for sharing microcode. The width flag controls the ALU during tests and shifts. If 0, the ALU operates on 16-bit data. If 1, the ALU operates on 32-bit data. The address flag differentiates between 31-bit MV/8000-specific addresses and 15-bit addresses. When 0, it forces the current segment number and zeros into bits 1-16 of the logical address (LA) buss. This is equivalent to the ECLIPSE C/350 address space. By manipulating these flags, microroutines that are logically the same, but use different data, may be shared by C/350-specific and MV/8000-specific instructions.

Short Debug Time

Since design time was a critical factor for success with the MV/8000, effort was made to minimize debug time. The hardware, microcode, and diagnostics were developed simultaneously. A software simulator of the MV/8000 micro-architecture [7] provided the means for debugging the microcode and diagnostics together before execution on a hardware prototype. As a result, in many cases the MV/8000 diagnostics executed without an error on the first try on the prototype. Meanwhile, the printed circuit boards were debugged by means of special diagnostic microcode that performed simple microoperations one step at a time. The use of writeable control store on the MV/8000 allows both diagnostic microcode and the MV/8000 instruction set to be loaded from diskette as well as facilitating quick modification to the microcode for debugging purposes.

The built-in diagnostic features of the MV/8000 are also very useful in manufacturing and field support of the hardware. The CC board has the ability to halt, continue, and single step the CPU. It can load microcode and diagnostics from a diskette and it can examine and modify that data both in the processor or on the diskette. Through special hardware and with the assist of special microcode, it can examine and modify the state of virtually any register in the machine. Through the soft console, the user can program the CC to monitor machine processing or take control for diagnostics. See [6] for more information about MV/8000 diagnostics.

The Microinstruction Format

Figure 3 is the microinstruction format for the MV/8000. A brief description of the functions of the fields follows; for a more thorough description see UINST.LS (MV/8000 Microcode Specification) [8]. The word can be viewed as consisting of four major parts: the next address control, the 2901 slice control, memory conrtol, and microcode mode control. Also refer to figure 4, a simplified ALU Block diagram.

NAC, the next address control field, controls what microinstruction the microsequencer fetches next. Test selection, branching, microsubroutine calling, and loading of the microsequencer state is specified in this field. [5]

AREGS and BREGS are the register selection fields for the 2901 slices. This field also allows for the modification of the register selections based on bits in the macroinstruction. BREGS specifies the destination register for the 2901 register file. ALUS, ALUOP, and ALUD fields are the 2901 control lines as described by UINST.LS [8]. These fields are modified in accordance with the specification of the CSM field. CRYINS is the field which specifies the carry input to the 2901.

The CSM (control store mode) field specifies if the ALU is being used in full-cycle mode or split-cycle mode. Full-cycle mode allows one operation to occur in each 220 ns microcycle on the ALU. The split-cycle mode double-cycles the ALU to accelerate multiply and divide, and to allow two ALU operations in one cycle. This means two operations occur in each 220 ns microcycle on the ALU. The first operation occurs in the first 110 ns (first half), and the second operation occurs in the second 110 ns (second half). For example, address generation must occur during the first half cycle. The Q mode selections of the CSM field use QREG for computing a memory address and updating QREG in the first half. Another ALU operation occurs during the second half, and this second result is loaded into a destination register. Other split-cycle modes allow microcode to specify the ALU control during one half of the cycle while forcing specific actions during the other. The microcoder must be aware of timing details of the hardware to know when to use the full-cycle or the split-cycle modes.

D1ST and D2ND control the selection of the D input to the 2901 in both the first and second halves of the microcycle. The SHFT field specifies the single bit shift input for the 2901, unless D2ND specifies the use of the barrel shifter, in which case the SHFT field controls the function performed by the barrel shifter. This shifter is generally used in conjuntion with a split-cycle mode to allow an ALU result computed during the first half to be shifted by several nibbles (4 bits) at a time during the second half.

The RAND field specifies low usage and distinct microorders which generally cannot be used in conjunction with one another. The interpretation of the bit field assignments of this field are controlled by the CSM field. Examples of functions of this field are: ATU function commands used for protection and enforcment of the ring structure of the architecture [1], control of the floating point exponent and sign units on the ALU, and instruction processor commands to perform a branch, forcing the data in its decode pipeline to be flushed.

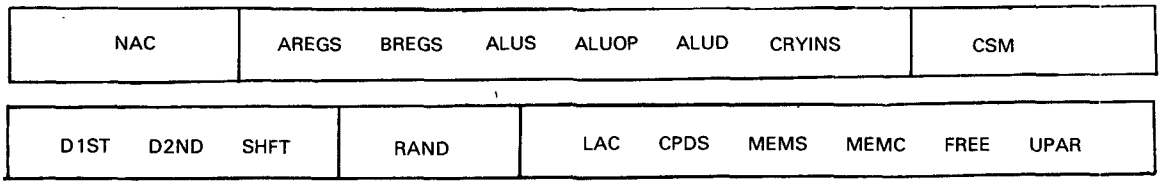

Figure 3

Microinstruction Format

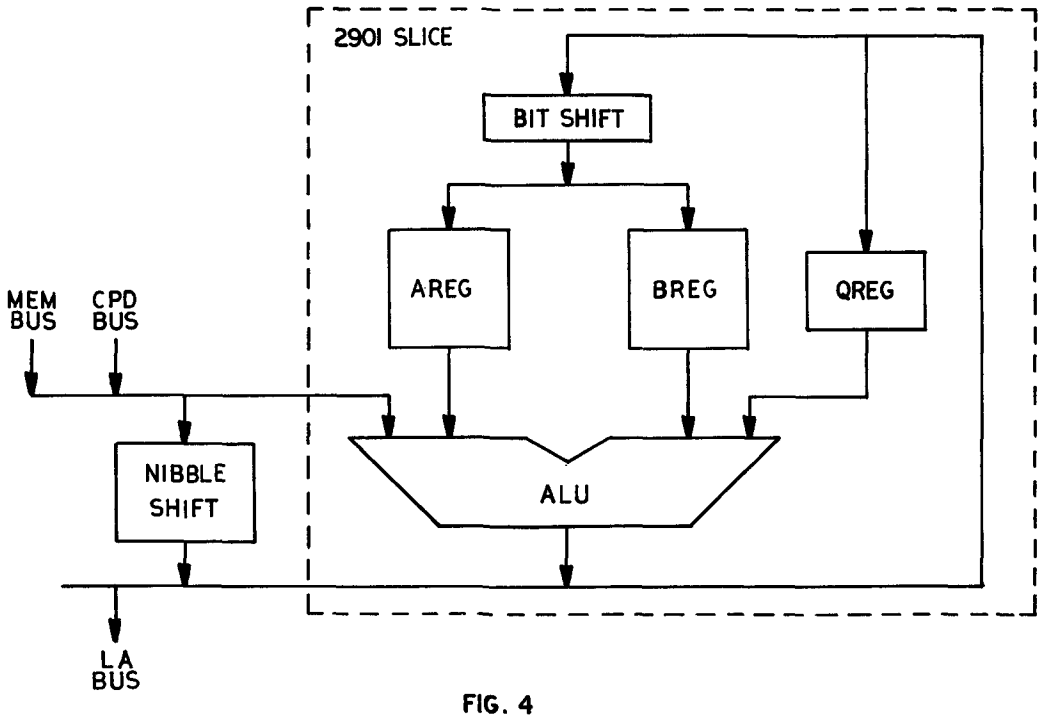

FIG. 4

Simplified ALU Block Diagram

The LAC field selects which source drives the logical address (LA) buss for memory addressing. The CPDS field selects which source drives the CPD, central processor data buss, for interboard communications. The MEMS and MEMC fields control the type of memory access to be done. The FREE field is reserved for future extensions and the UPAR is a parity bit to allow hardware to detect a single bit error in each microinstruction.

Specifics of Microcoding

1. Microcoding the Decode ROMs.

Significant performance is obtained in the MV/8000 from the preprocessing done by the instruction processor. The IP fetches the next macroinstruction and decodes it to determine what the starting state of the machine will be when the instruction is executed.

For an instruction which adds two accumulators together, ADD, the information passed to the rest of the CPU at the start of the microroutine is:

1) a width flag which tells the ALU whether a macroinstruction operates on 16- or 32-bit quantities,

2) a starting microaddress for the microroutine, and

3) a descriptor which specifies if the macroinstruction may skip the next 16-bit word in the instruction stream.

The width flag allows the sharing of microcode which operates on two 16-bit accumulators with the microcode which operates on two 32-bit accumulators. The descriptor specifying if the macroinstruction may skip the next word is useful because it allows the instruction processor not to move data beyond the stage of its pipeline which would force the flushing of the entire pipeline if the skip is performed. This information is provided by a set of decode ROMs which are addressed by combinational logic based upon the macroinstruction opcode. The contents of these ROMs are specified by the microcoder allowing him to control the initial machine state for each macroinstruction.

The instruction processor also provides additional information for instructions which have immediates as one of their arguements, e.g.: ADDI, and for memory reference instructions which have displacements, e.g.: LDA. It extracts the immediate or displacement and places this value in a register on the ALU board during macroinstruction decoding.

2. Memory Reference Instructions

The memory reference instructions in the MV/8000 form a large class of instructions which share the same basic functionality of resolving an address. With this large number of instructions, it became desirable to share the microcode for resolving addresses across the group. This instruction group has an index specifier, a displacement, and an indirection bit for address resolution. Seven classes of effective address (EFA) resolutions exist: memory reference of 16- and 32-bit data, branch, calculate address, cross ring, and 3 types of memory reference of 8-bit data. The information needed to identify an effective address group, index specifier placement, and displacement type are programmed in the decode ROMs for each instruction. After the address calculation, these EFA microinstructions transfer control to the start of the macroinstruction's microroutine.

3. Microcode Traps

The existence of demand paging, fixed and floating point overflow, and protection faults required the microsequencer to provide a clean way of handling hardware traps. A trap is an event that interrupts the microcode in order to handle an "unexpected" event, such as the ones listed above. Upon a trap, the microsequencer places the current microprogram counter on the microstack and saves the next address and the previous test bit. Control is then transfered to one of the trap routines and an abort signal tells other boards to avoid clocking any registers, thereby nullifying the current microinstruction. The individual trap routines may push the sequencer state on the microstack before continuing, or they may carefully avoid further traps. When trap processing is complete the interrupted microinstruction is restarted by special sequencer commands. [5]

An ATU miss occurs when the ATU translation cache does not contain a valid translation for the logical address desired. The CPU traps to the long address translation (LAT) microcode routine which queries the page table in memory, restarts the memory access, and loads tha ATU with the missing translation before returning to the interrupted microinstruction. The LAT may be as short as 5 cycles, but it is also the initiator of page faults if the particular page is not in memory (requiring that the operating system bring in the page from disk). In this event, the microcode stores in main memory the internal state necessary to successfully complete the macroinstruction and then branches to the operating system. This state is called the context block. The instructions which compute an EFA do not need extensive state storage to complete, so a short context block is stored for these instructions. This accelerates many of the instructions that take the LAT trap.

Another type of memory reference trap is caused by the protection structure of the architecture. In this case the instruction is aborted and fault information is passed back to the operating system to inform the user of the error.

4. I/O Interrupts

I/O interrupts can occur any time during execution of the instruction stream. This poses a problem of detection and servicing in a system which has real time constraints. It is desired that interrupts be handled at known times in the microcode algorithms. The time when interrupts are detected and serviced in the MV/8000 is generally between macroinstructions. The instruction processor examines the interrupt pending flag at each instruction decode. If an interrupt is pending, the IP forces a branch to a microroutine dedicated to the servicing of interrupts, which saves the program counter and branches to a software interrupt service routine.

However, some instructions are sufficently long that a real time response is not possible if the interrupt check is done only between instructions. The microcode is therefore required to check the state of the interrupt pending flag for instructions exceeding a fixed amount of time. If an interrupt is pending, some instructions will abort their processing to be restarted later, while others will save sufficient state so that they may resume where they left off. The microcode designer makes this decision for each instruction that must check the interrupt pending flag. A bit in the Processor Status Register (PSR) exists for the microcode to flag when an instruction is resumable. After the interrupt is handled, the interrupted instruction is restarted. Resumable instructions always check the PSR initially to determine if it had been interrupted. If so, it restores its state and continues where it was interrupted.

A memory location is reserved for the Program Counter (PC) when an interrupt occurs. C/350-specific programs expect a 15-bit PC to be in location 0 (the first 16-bit word of each segment) and may return from interrupts by a JMP @ 0 (jump indirect through 0). MV/8000-specific programs need a 31-bit PC. When a user writes a program that uses the 31-bit address space, the interrupt routine must begin with an MV/8000-specific instruction. The microcoded interrupt service routine puts bits 17-31 of the PC in location 0, as before for C/350-specific programs, but then examines the first instruction of the interrupt handler. If it is an MV/8000-specific instruction, the 32-bit PC is placed in locations 2,3.

5. Performance Considerations

The MV/8000 design goal of high performance required study of the microcode of instructions that appear frequently in the instruction stream. These instructions include those used heavily in scientific applications: fixed and floating point arithmetic. The handling of exceptional conditions of these instructions, namely fixed and floating point faults, should not impair the performance of the instructions in the absence of exceptional conditions.

This design philosophy placed an interesting constraint on the design process, which is best demonstrated by example. One class of MV/8000-specific instructions are called Memory-to-Accumulator Instructions [1]. These instructions reference data in memory, perform a binary operation between this data and the contents of a specified accumulator, and place the result in the accumulator. If a fixed point overflow exception occurs, the CPU initiates a fixed point fault: a "return block" is pushed onto the (software) stack and the processor branches to the fixed point fault handler, a routine supplied by software. Upon arrival at this routine, AC0 (accumulator 0) contains the PC of the instruction that initiated this fault.

This class of instructions is used frequently in software. Additionally, this class is important because it reduces the length of the instruction stream. One memory-to-accumulator instruction replaces two instructions that might be used instead: an instruction to read memory followed by an instruction to perform the operation. Therefore, the memory-to-accumulator instructions help keep the decode pipeline of the IP full by reducing the number of macroinstruction fetches that the IP must perform.

For this example we shall discuss LWADD, add a 32-bit doubleword from memory to a 32-bit accumulator. Upon initial analysis, the microcoder assumes that this instruction will take 3 microcycles, or 3 x 220 ns = 660 ns, as follows:

1) Compute the effective address calculation and "start" (or address) memory with the result;

2) Read the 32-bit doubleword from memory;

3) Add this data to the specified accumulator and place the result into the accumulator. Initiate a fixed point fault if the result overflows 32 bits. Parse the next macroinstruction.

When writing the detailed microcode, the microcoder realizes that three cycles are not enough since the instruction length must be saved in a microcode General Register (GR). If there is a possibility of a fixed point fault, a requirement of the architecture is that the PC of the faulting instruction be placed into AC0. This is performed by the fixed point fault microcode service routine. Since cycle 3) causes a parse of the next macroinstruction, the PC in this routine is already changed to point to the macroinstruction following the LWADD. The routine subtracts the length in GR from the PC to get the PC of the faulting instruction, which it places into AC0. It cannot assume a particular length (such as the length of LWADD) since instructions of different lengths can cause a fixed point fault.

Therefore, the microcoder concludes that he must take 4 cycles, or 4 x 220 ns = 880 ns, as follows:

1) same as 1) above;

2) same as 2) above;

3) Store the instruction length into GR;

4) Add the memory data to the specified accumulator and place the result into the accumulator. Initiate a fixed point fault if the result overflows 32 bits. Parse the next macroinstruction.

Since LWADD is a high frequency instruction, it seems unacceptable to penalize its every occurrence by 220 ns simply to support the infrequent occurrences of a fixed point fault. After the design engineers conferred on this problem, a simple hardware change was discovered which allows the control of an unused data path in parallel with the computation of the result during the last cycle. This path is used to load the instruction length into a special register, the LAR. Additionally, this change occurs within a PAL (Programmable Array Logic) [4], so the hardware prototype is fixed by burning a virgin PAL with the new programming and inserting this part into the system. No wire changes are necessary.

The final 3 cycle LWADD microroutine appears as follows:

1) Compute the effective address calculation and "start" (or address) memory with the result;

2) Read the 32-bit doubleword from memory;

3) Add this data to the specified accumulator and place the result into the accumulator. Store the instruction length into LAR. Initiate a fixed point fault if the result overflows 32 bits. Parse the next macroinstruction.

This discussion illustrates a constraint on the design process: the expectations of the (microcode) designers as to machine performance forced the design team to alter the design during the debug process. The design process thus becomes dynamic itself. Flexibility in the design, such as the use of PALs, PROMs (Programmable Read-Only Memory), and writeable control store make this methodology feasible.

Notes:

[1]ECLIPSE MV/8000 Principles of Operation, DGC 014-000648
[2]ECLIPSE C/350 Principles of Operation, DGC 014-000610
[3]IDM2900 Family Microprocessor Databook, National Semiconductor Corporation
[4]PAL Handbook, Programmable Array Logic, Monolithic Memories Inc.
[5]David I. Epstein, "The ECLIPSE MV/8000 Microsequencer", submitted to MICRO-13 Workshop
[6]Paul Reilly, Elizabeth Shanahan, Steven Staudaher, "An Implementation of Microdiagnostics on the ECLIPSE MV/8000", submitted to MICRO-13 Workshop
[7]Neal R. Firth, "The Role of Software Tools in the Development of the ECLIPSE MV/8000 Microcode", submitted to MICRO-13 Workshop
[8]UINST.LS (MV/8000 Microcode Specification), DGC 080-005005