Millicode in an IBM zSeries processor

L. C. Heller
M. S. Farrell

Because of the complex architecture of the zSeries® processors, an internal code, called millicode, is used to implement many of the functions provided by these systems. While the hardware can execute many of the logically less complex and high-performance instructions, millicode is required to implement the more complex instructions, as well as to provide additional support functions related primarily to the central processor. This paper is a review of millicode on previous zSeries CMOS systems and also describes enhancements made to the z990 system for processing of the millicode. It specifically discusses the flexibility millicode provides to the z990 system.

Introduction

As the instruction set for high-end processors has evolved over the years, more and more complex instructions and features have been added to the architecture. Conceptually more straightforward instructions, such as loads, stores, moves, branches, and logical and arithmetic instructions, can be implemented directly by the hardware. The more complex instructions and features, such as I/O instructions, Start Interpretive Execution (SIE), cross-memory instructions, interruption handlers, resets, and certain RAS (reliability, availability, and serviceability) features, must be implemented with some type of internal code. Starting with the S/390* G4 [1] processor in 1997 and continuing through the G5 [2, 3], G6 [3], z900 [4], and now the z990 [5] processors, the code internal to the central processor (CP) is called millicode.

On many processors prior to G4, which were implemented using multiple chips in bipolar technology, the internal code was placed on separate chips of the processor; this is known as horizontal microcode. With the G4 system, the entire processor was implemented on one chip. Because of the area constraints of the chip, a redesign of the internal code of the processor was required, since it would no longer fit on the chip, and, with new requirements being architected and designed for high-end systems, even more processor internal code would be needed. This led to the design of vertical millicode as the internal code of the processor. The millicode is written in assembler language, primarily with z/Architecture* [6] instructions, as well as with specialized millicode-only instructions.

One of the main objectives in designing the hardware to support millicode was to provide additional flexibility to the machine. Various features in the hardware were defined early in the design process with the intent of providing more control to millicode for use later in the test phase of the design process. Facilities are implemented to allow millicode, in many instances, to provide solutions for hardware problems which are found during the debug cycle, without requiring new changes to the hardware. The capabilities given to millicode also allow for the introduction of new functions late in the design cycle, after hardware changes are no longer possible.

Millicode structure

Millicode resides in a protected area of storage called the hardware system area, which is not accessible to the normal operating system or application program. However, in many ways, millicode is handled by the processor hardware similarly to the way operating system code is handled. The millicode is brought into the processor from system area storage and is buffered in the instruction cache. The instruction unit (I-unit) hardware fetches the millicode instructions from the cache, decodes them, calculates the operand addresses, fetches the operands, and sends them to the execution unit (E-unit) for the actual execution and the put-away of the results. Millicode

0018-8646/04/$5.00 © 2004 IBM

Table 1 z990 R-unit registers.

Address	*Mnemonic*	*Description*
00–0F	GRs	Program general registers
10–1F	MGRs	Millicode general registers
20–27	ARs	Program access registers
28–2F	MARs	Millicode access registers
30–3F	FPRs	Floating-point registers
40–4F	MCRs	Millicode control registers
50–5F	G2CRs	Guest-2 control registers
60–6F	G1CRs	Guest-1 control registers
70–7F	HCRs	Host control registers
80–9F	MCRs	Millicode control registers
A0–AF	IA	Instruction address, PSW, exception
B0–BF	—	Instrumentation counters
C0–CF	SYSR	System registers
D0–D3	TF	Guest-2 timing facilities
D4–DF	SYSR	System registers
E0–E3	TF	Guest-1 timing facilities
E8–EB	COP	Coprocessor communications
F0–F2	TF	Host timing facilities
F8–FF	SYSR	System registers

execution uses the same basic data flow as is used to execute system instructions. The similarity in the execution of millicode and system code resulted in a significant decrease in the design complexity of the hardware needed for the processor internal code, as well as a significant savings in the chip area. Previous systems required a large amount of unique hardware and a large hardware design effort for a separate execution engine to support the processor internal code.

The current implementations of millicode are written in a form of z/Architecture assembler language. That is, the millicode routines are written using the architected mnemonics for the z/Architecture instruction set. Within these routines, however, only those architected instructions that are implemented directly by the hardware are available for use by the millicode. There are several hundred of these instructions, which contain most of the basic operations (such as moves, stores, arithmetic and logical operations, and branches) that are typically needed for implementing a more complex function. The z/Architecture instructions which are themselves implemented by millicode cannot be used within millicode routines, since this would lead to recursion, which cannot be handled. In addition to the z/Architected instructions executed by the hardware, there are approximately seventy additional instructions, called milli-ops, which have been implemented for use only by millicode. These milli-ops deal primarily with setting and retrieving data that is unique to the internal hardware implementation, where no z/Architecture instruction is applicable. In other instances, milli-ops have been added to improve the performance of complex z/Architecture instructions that are implemented in millicode. Some of these unique instructions are discussed later.

Registers

While an operating system or application program is running, one of the key resources used for managing data and addresses is the architected program general registers (GRs). Similarly, a key resource used by millicode routines is the millicode GRs (MGRs). These are 16 64-bit registers which are used to handle intermediate results within a millicode routine, as well as to hold addresses needed to access customer storage, or to access data within the machine hardware system area. These GR register files are separate and distinct: The program GRs are accessed while the system program is running and the millicode GRs are accessed while the millicode is running. For example, when the system program issues a Load instruction, a program GR is updated; when a millicode routine issues a Load instruction, a millicode GR is updated. The instruction address in the program-status word (PSW) tracks the address of the instruction that is being executed for the program. This address is updated at the completion of every z/Architecture instruction or superscalar group of instructions, including being written with a new address for a successful branch instruction. For branch instructions, the instruction address is also kept in the branch target buffer (BTB) to assist with branch prediction for future branches that are decoded at the same instruction address. Similarly, when executing millicode routines, the hardware tracks the instruction address that is being executed within the millicode routine in a separate millicode instruction address register. For branch instructions executed within the millicode routine, the millicode instruction address is also kept in the BTB to assist with branch prediction for subsequent branches decoded at the same millicode instruction address.

R-unit registers

One of the more important features of current CMOS processors, at least with regard to the millicode implementation, is the concept of a recovery unit (R-unit). This unit contains the entire architected state of the processor [6] as well as the state of the internal controls of the processor. The R-unit includes the program general registers and access registers, millicode general registers and access registers, floating-point registers, architected control registers for multiple levels of SIE guests, architected timing facilities for multiple levels of SIE guests, information concerning the processor state, and information on the system configuration. In addition, there are registers which control the hardware execution, and data buses for passing information from the processor to the other chips within the processing complex. A description of these registers is included in **Table 1**.

The R-unit registers provide the primary interface between the millicode and the processor hardware, and are used by millicode to control and monitor hardware operations. These special registers in the R-unit are accessible to the millicode, and there are several unique milli-ops to access them, such as Read Special Register, Write Special Register, AND Special Register, OR Special Register, and logical immediate ANDs, ORs, and Inserts to various 2-byte fields of some of the R-unit registers. Through these instructions millicode can, whenever current execution requires it, read or alter much of the state information of the processor. This can take place either during the execution of an instruction which has to read or write specific state information, or during some other type of function, such as during the resetting of the processor or handling a recovery situation.

Millicode branching

The z/Architecture provides, as a result of many instructions, a condition code that indicates which of a number of results actually occurred during this execution of the instruction. The instructions that modify the condition code include compares, tests, adds and subtracts, and logical operations, as well as many others. The z/Architecture also provides branch instructions which specify a mask that forces a branch of the instruction stream to a new instruction address if a “1” in the corresponding mask bit selects the current condition code. Similarly, when a millicode routine issues an instruction that changes the condition code, a unique millicode condition code is modified instead of the program condition code. When the millicode routine issues a branch instruction that interrogates the condition code, it is actually the millicode condition code that is being examined. This leaves the system program condition code intact, so that subsequent branches by the program are examining the results of system instructions, not millicode instructions. There are, however, milli-ops that allow millicode to change the program condition code, either to an explicit value or to the current value of the millicode condition code, for use when the implemented function architecturally requires a condition code update.

In addition to the basic branch instructions provided by the z/Architecture, such as Branch on Condition, Branch on Count, and Branch and Save Register, which are available for millicode use, there are additional branch instructions provided for use only by millicode. One of these instructions is called Branch Relative Special (BRS). This instruction interrogates a particular bit or condition in the processor, as specified within the instruction, and then branches to a relative address within the millicode routine based on whether the bit or condition is true or false. Some of the more common conditions that can be interrogated are bits in the PSW, bits in control registers, SIE emulation modes, state of pending interruption conditions, or state of the asynchronous coprocessor. Using this BRS instruction, the millicode routine can easily and quickly check the state of the specified condition.

Another branch instruction available to millicode is Branch on Flags (BRFLG). This instruction allows the millicode routine to check one or more flag bits, and then branch based on the state of the flag or flags. These flags are merely local indicator bits which are used independently by different millicode routines. That is, a particular flag bit might be used to indicate a certain condition in one routine, but could indicate a totally different condition in another millicode routine. The flags may be set by hardware during millicode setup or by special instructions available to millicode. Once a flag is set up early in a routine to indicate a particular condition or path through the routine, the BRFLG instruction can be used later in the routine to interrogate the state of this flag and thus have a quick way of discerning the earlier condition.

There are 16 flags, divided into four groups of four flags each. With a single BRFLG instruction, one to four of the flags within a particular group can be interrogated. A branch can then be taken on the basis of whether

- All of the selected flags are 1s.
- All of the selected flags are 0s.
- Any of the selected flags are 1s.
- Any of the selected flags are 0s.

Operand access control registers (OACRs) and millicode addressing

While executing within a millicode routine, storage requests can be made to program storage on the basis of either the current addressing mode of the executing system program or a specific architected addressing mode, or to the hardware system area. Each storage request made by millicode must be tagged with the appropriate addressing controls so that hardware can address storage correctly. For this reason, hardware determines the addressing mode on the basis of which millicode base register is used to make the storage request. When the request specifies a millicode base register of 1 through 7, the storage access is made using the same addressing mode that is currently indicated by the system program. The current addressing mode is determined by using bits in the current PSW, specifically the dynamic address translation (DAT) bit, the address-space control bits, and the addressing mode bits. When a millicode base register of 12 through 15 is used for a storage request, the corresponding address is treated as a hardware system area address.

When millicode specifies a base register of 8 through 11, special hardware designates the address mode used for the storage access. Four OACRs exist in the R-unit, each corresponding one-to-one to millicode base registers 8 through 11. These registers include a storage access key, an address-space control (primary, secondary, home, or access register), an addressing mode (24-bit, 31-bit, or 64-bit addressing), and an addressing type (real, virtual, host real, absolute, or hardware system area), in addition to special controls which can block program event recording (PER) storage alteration detection or protection exceptions, and can pretest for store-type access exceptions. Millicode can set each of these four registers independently and, for subsequent storage requests, hardware will use the appropriate OACR to determine how to interpret the address presented by the millicode routine. By this definition, millicode routines can use any of the millicode GRs as base registers when accessing storage. This allows data to be moved between different address types simply by using different base registers for the operands.

GRs and ARs

For many of the functions that are implemented by millicode, the initial operands specified by the system program are located in the program GRs. However, for the millicode routine to work easily with this operand data, the data must be transferred to the millicode GRs. To assist with transferring the data, four 4-bit register indirect tags are defined, with each pointing to the GR specified as an operand by the system program instruction. The indirect tag can correspond to an R1, R2, R1+1, R2+1, R3, B1, B2, etc. field—whatever is required for a specific instruction. The register indirect tags are initialized by hardware to correspond to the same operands for any given instruction, but they could be set to indicate different operands for a different instruction. The millicode routine for an instruction can then use a specific register indirect tag as a pointer to a specific operand.

Two milli-ops make use of these register indirect tags to access program GRs: Extract Program GR Indirect (EXGRI) and Set Program GR Indirect (SPGRI), which are used respectively to read and write a program GR. These instructions take as their operands a register indirect tag (which points to a program GR) and a millicode GR. Millicode uses EXGRI to copy the data from the program GR specified by the register indirect tag operand to the millicode GR, where it can operate on the data. When the millicode routine returns the resultant data to a program GR, it uses SPGRI. The routine does not identify the specific system program GRs that are affected, only which operands of the instruction (R1, R2, etc.) are being used.

For some z/Architecture instructions, one or more of the GRs used as operands are not specified by the instruction, but are implied as a specific GR (such as GR 0 or 1). For these cases, the register indirect tags are not needed, and two other milli-ops can be used instead. The Extract Program GR instruction takes as its operands a program GR and a millicode GR. The execution of this instruction copies the data from the specified program GR to the millicode GR. Similarly, the Set Program GR instruction has as its operands a program GR number and a millicode GR number, and the data is copied from the millicode GR to the program GR.

When the data in the program GR is an address and the system program is operating in the access register mode, the millicode routine must obtain the program access register contents prior to making any storage accesses. For this purpose, there are instructions to move the data between a program AR and a millicode AR: the Extract Program AR Indirect and Set Program AR Indirect instructions, which specify the program AR using a Register Indirect Tag, and the Extract Program AR and Set Program AR instructions, which explicitly reference the specific program AR.

Perform Translator Operation (PXLO) and internal translator functions

For most storage references the translator hardware in the processor performs the dynamic address translation (DAT), with the resultant DAT mapping stored in the translation-lookaside buffer (TLB). Once the requested data is returned from storage, it is buffered in either the instruction cache or the data cache. This process is the same for references made by the system program and for references made by the millicode. This is a process normally performed directly by the hardware and is transparent to the millicode.

There are some occasions, however, when the millicode must become involved in some aspect of a translation process or involved in the manipulation of data in the TLB. Two examples of this are the z/Architecture instructions Load Real Address (LRA), for which the translation process must stop earlier and return the real address, which corresponds to the initial virtual address, and Purge TLB (PTLB), which causes the TLB to be purged of relevant entries. To assist in these functions, the PXLO milli-op was developed for use by millicode. The PXLO instruction also provides millicode with the ability to block exceptions and have them reported via condition codes or additional bits in specific R-unit registers. This allows millicode to manually process exception conditions, which is often useful in circumventing hardware problems.

There are a variety of subfunctions provided by the PXLO instruction that allow millicode to perform many different types of translator and TLB functions. When

Table 2 Sample PXLO commands for the z990 system.

PXLO command	*Description*
Load Address Space Control Element	Determines the ASCE used for a translation
Load Absolute Address	Obtains an Absolute Address of a translation
Load Real Address	Obtains the Real Address of a translation
Load Host Real Address	Used while in emulation mode to obtain the Host Real Address, when translating a Host Virtual Address
Load Page Table Entry	Obtains the Page Table Entry address for a translation
Load Host Page Table Entry	Used while in emulation mode to obtain the address of the Host Page Table Entry, when translating a Host Virtual Address
Purge TLB	Purges previous translations from the TLB
Invalidate Page Table Entry (IPTE)	Invalidates selected entries from the TLB
Read TLB	Reads an entry from the TLB
Write TLB	Writes an entry into the TLB
Purge Data Cache	Purges all entries from the Data Cache
Purge Instruction Cache	Purges all entries from the Instruction Cache

issuing the PXLO, millicode specifies a function code in the appropriate R-unit register, and if an address is required for the PXLO instruction, it is supplied as an operand of PXLO. Examples of PXLO commands for the z990 system are shown in **Table 2**.

The majority of these subcommands are provided to give millicode routines the capability to handle the instructions and internal functions (such as LRA and PTLB) required by the z/Architecture, while others are provided to assist with debugging and potential fixes for hardware problems.

System communication

There are many instances in which one processor has to communicate with the other chips in the system for signaling purposes, to set controls in the storage controller chip for memory scrubbing, to initiate an I/O operation at the I/O chips, or to communicate with the service processor through the clock chip. (These are only a few examples. There are many other reasons why millicode might have to send commands to any of the other chips in the system.) The method used by millicode to send these commands to a remote chip is called a system operation, or SYSOP.

There are multiple registers in the R-unit to handle the SYSOP execution: a control register, an output data register and an output address register, and a pair of input registers for data returned from the system. Each of the SYSOP commands may use a different subset of these registers. As appropriate for the command, the millicode sets up the control information and the output data and address register and then launches the SYSOP. The command is sent out of the processor and into the system, where all of the chips in the system receive the command. Only the target chip (or chips) take any action on the command; all other chips simply discard the command without making any response.

The processing of the SYSOP command at the remote chip occurs asynchronously with respect to any processing that is done on the initiating processor. The remote chip returns a response when it has received the command and/or completed its processing of the command. The millicode routine which initiated the SYSOP must monitor the SYSOP control register to determine when the SYSOP has completed, as well as whether the SYSOP completed successfully or an error was encountered. Then, depending on the particular SYSOP that was issued and the returned result, the input data registers can be read for any data returned from the SYSOP operation. If any errors are encountered during the processing of the SYSOP operation, the millicode routine must retry the operation, force an error condition to be indicated to the system, or take some other action. This action depends on the command being executed and the type of error that has been returned.

Millicode entry and exit

As the system program continues to issue instructions that do not require execution by millicode, the hardware can continuously fetch instructions from storage, decode them, and execute them. However, when an instruction is encountered that must be executed by millicode, the normal processing of the system program instruction stream stops, and the instruction addresses of both the current system program instruction and the next sequential instruction are saved.

Using the opcode of the instruction (in a modified format) as an index into the millicode section of the hardware system area, the appropriate millicode routine is fetched into the instruction cache. Each routine is given 128 bytes of contiguous storage before the next routine begins. If

Table 3 Sample hardware setup for millicoded instructions on the z990 system.

OpCode	*Mnemonic (format)*	*Hardware setup*
B207	STCKC (S)	1. Machine Check if Millicode Mode = 1 2. Privileged Op Exception if PSW.15 = 1 3. Specification Exception if Operand 2 is not doubleword aligned
		1. IAREGA7.0:31 = Instruction Text 2. MGR9 = Effective Addr. Operand 2 3. MAR9 = B2 ALET 4. RI2 = B2 GR (for AR Number)
DA	MVCP (SS)	1. Machine Check if Millicode Mode = 1 2. Special Op Exception if PSW.5 = 0 | PSW.17 = 1 | CR0.37 {CR0.5} = 0
		1. IAREGA7.0:47 = Instruction Text 2. MGR9 = Effective Addr. Operand 1 3. MGR11 = Effective Addr. Operand 2 4. MAR9 = B1 ALET 5. MAR11 = B2 ALET 6. RI0 = R1 GR 7. RI1 = R3 GR 8. RI2 = B1 GR (for AR Number) 9. RI3 = B2 GR (for AR Number) 10. Flag D = 1 if Operand 1 crosses a 2K/4K boundary 11. Flag E = 1 if Operand 2 crosses a 2K/4K boundary
		OACRs 8 and 10 initialized to Logical, Block-Per, Test Modifier
B257	CUSE (RRE)	1. Machine Check if Millicode Mode = 1 2. Specification Exception if R1 | R2 are Odd GRs
		1. IAREGA7.0:31 = Instruction Text 2. RI0 = R1 GR 3. RI1 = R2 GR 4. RI2 = R1+1 GR 5. RI3 = R2+1 GR

additional storage is required to complete the routine, the millicode will later branch to a unique location in system area storage that is defined for general use for millicode routines and has no size constraints.

Prior to execution of the first instruction of the millicode routine, setup is performed by the hardware to prepare for millicode execution. The actual instruction text is saved in a register for use by the millicode, if needed. If an address calculation is required for the operand of the system program instruction, the calculated address is placed in a millicode GR, and the associated program AR is copied into the corresponding millicode AR. Some of the OACRs are initialized with the access key and addressing mode of the current program PSW, and some are set to the real addressing mode with an access key of zero. The register numbers of the relevant program GRs, based on the format of the system program instruction, are placed in the register indirect tags. For some instructions, flags are set to indicate particular facts about the instruction operands, such as page crossings, equal operand values, or operand values of zero. For a limited number of instructions, the actual operand contents are set directly into millicode GRs during this millicode entry process. In the z990 processor, this typically takes only two or three cycles and is done in an effort to simplify the work required by millicode and therefore improve the overall performance of the instruction. A few examples of the hardware setup provided for millicode entry are given in **Table 3**.

Once all of the appropriate hardware facilities have been set up, the millicode routine has enough information about the specific details of the instruction and its operands to start execution of the instruction. For many instructions, the hardware also checks some of the program interruption conditions that may be possible for the instruction (privileged operation exception, specification exception, etc.). The millicode routine is responsible for checking for any possible program interruption conditions that are not checked by the hardware, in the appropriate architectural order.

If no interruption conditions are detected, the millicode routine continues its processing, working on the data that was set up during millicode entry, fetching program GRs into its own GRs, reading data from the R-unit, and requesting data from storage. The millicode routines can use almost all of the hardware-implemented z/Architecture instructions, as well as the special instructions available only for millicode execution. An instruction address register (other than the one that holds the saved operating system instruction address) is used to maintain the instruction address as the millicode routine executes. The routines can branch to other places within the same routine, branch to a different routine, or call a different routine as a subroutine, with the millicode instruction address register keeping track of which address to fetch and decode next.

As the millicode routine executes, architected facilities are updated with the calculated results. These facilities could be the program GRs, storage locations, or registers in the R-unit that control future execution. When all of the operations for the instruction of the system program have been performed, and any condition code has been set, the millicode routine can stop processing. A milli-op, Millicode End (MCEND), is issued which alerts the hardware that this is the last instruction in this millicode routine. When this MCEND is decoded, the hardware stops fetching instructions from the millicode instruction address register and resumes fetching instructions from the "next sequential instruction address" register of the system program, which was saved on entry into the millicode routine. The hardware then begins decoding an instruction from the system program instruction stream, and either has the instruction directly executed by hardware, or returns to another millicode routine for its execution.

Interruptions

During the normal course of processing by the system program, pending interruption conditions which have to be presented to the system program are detected. These could be program interruptions, I/O interruptions, external interruptions, etc., all of which must be routed to millicode for proper execution. When the interruption condition is detected, the hardware stops execution of the system program instruction stream and prepares to enter millicode. On the basis of the type of interruption condition to be presented, an offset into the millicode area in the system area is formed, and the appropriate millicode interruption routine is fetched.

There is no setup of millicode facilities (OACRs, register indirect tags, millicode GRs, flags, etc.) for an interruption, as is done for an instruction. Therefore, the millicode cannot make any assumptions about the contents of these registers when entering the routine. There is a milli-op, Extract Interrupt (EXINT), which passes to the millicode the exact interruption class within the interruption type. (For example, if the type is an external interruption, the EXINT returns a code to distinguish a Clock Comparator Interruption from a CPU Timer Interruption, or any other external interruption condition.) Once the specific class of interruption has been determined, the millicode routine can process the interruption according to the architecture. This would normally involve placing interruption codes and parameters in storage, as well as swapping the current PSW with the appropriate new PSW to point to a system program routine which can handle the interruption condition. In addition, with most interruption conditions, there are also hardware facilities within the R-unit that are updated. Prior to leaving the millicode routine, the Reset Interruption (RIRPT) milli-op must be issued. This instruction clears the specified interruption condition from the hardware and enables it to detect any new or lower-priority interruption condition that might exist. Once all of the architected and internal requirements of the interruption have been completed, the MCEND instruction is issued. The hardware then begins fetching system program instructions from the current instruction address in the "next sequential address" register, which could have been (and probably was) modified by the millicode during the interruption routine. This allows the processor to fetch the appropriate system program instructions and handle the interruption condition that was just presented.

Millicode flexibility

Various features were added to the hardware during the design phase with the goal of providing flexibility to help millicode resolve problems, including hardware problems, which might arise later. The majority of these hardware workarounds are needed during the early stages of testing on a prototype machine. Implementing the fixes in millicode helps to reduce the number of hardware releases required and allows testing to continue while waiting for the hardware fixes. At times, the millicode solution can be used as the permanent solution if the performance degradation is negligible.

Millicode often uses control bits in the R-unit registers to circumvent problems. For example, these control bits can be set during initialization to completely disable a function that is working incorrectly. Since millicode can also write these controls during execution, a routine can also, following the same example, disable a function when a specific condition exists and re-enable it when the condition no longer exists. This still allows the function to be generally available and can significantly reduce the performance impact of a millicode change which circumvents a hardware problem. This is especially useful when the function only has to be disabled in a situation in which performance is not an issue.

A number of R-unit registers were defined to provide this flexibility. There is an R-unit register which controls the disablement of certain hardware functions, particularly new or complex functions. This significantly reduces the risk of introducing these functions into the machine. Hardware also provides programmable controls that can be set to cause a limited number of specified hardware instructions to be executed by millicode. Millicode also has the capability to override the hardware configuration (central processors, system assist processors, etc.), which in general is determined using a defined algorithm, on the basis of data contained within an R-unit register. This allows for testing of unique configurations and, in some instances, stresses certain aspects of the system design.

Through the PXLO commands, millicode has specialized access to the translator and TLB, including the ability to block exceptions and have them reported either via the condition code or via additional bits in specific R-unit registers. Although the majority of these functions were provided to implement z/Architecture instructions, the capabilities were enhanced to allow millicode to manually process the exception conditions and handle them differently if required. The Read TLB and Write TLB commands, on the other hand, were provided solely for the purpose of debugging and circumventions. Through the SYSOP commands, millicode has access to status and controls representing operations performed outside the local processor. This allows millicode, for instance, to query the status of an MBA operation or quiesce request, particularly when a failure was indicated for the SYSOP command and recovery is involved.

There are a number of instructions, including Purge TLB, Set Storage Key Extended, Invalidate Page Table Entry, and Start Interpretive Execution (SIE), which may invalidate all or part of the TLB. These invalidations are made using either a PXLO, for local requests, or SYSOP, for broadcast requests. Ideally, for performance reasons, these commands should invalidate the minimum number of entries required by the architecture. An R-unit register was defined that allows millicode to specify a subset of entries to be purged from the TLB1, giving additional flexibility which could be used to circumvent a potential problem or provide additional function. The register is divided into four-bit fields, each of which is used to control a specific command; the definition of the four bits is determined by the type of command. Each field in this register is initialized to the value required to handle the typical case so that the register has to be written only if a special situation exists. For the TLB2, the flexibility is provided by the programmable picocode [1] engine in the translator. In addition, an extra PXLO command for the TLB2 was added to allow the possibility of defining a new command later and implementing it in picocode.

Millicode load and concurrent patch

The millicode is assembled into a single file for use by the machine. The translator picocode is also assembled and inserted into the millicode file. This file resides at a fixed location in the hardware system area and is loaded into storage during initialization; a pointer in the R-unit is then loaded with the starting address of the millicode. It is used by hardware as the base address when calculating the starting location of a particular millicode routine in storage. Another R-unit pointer to the picocode is initialized, and a picocode load, which loads the code from storage into the picoengine, is performed.

Whenever a problem is fixed in millicode, a patch can be made that includes the code correction. When the millicode patch is produced, the entire millicode file, which includes the latest picocode, is provided. In most cases, this code can be applied concurrently, while the machine is still running. To do this, millicode loads the new millicode patch data into an alternate location in system area storage, adjusts both the pointer to millicode and the pointer to picocode, requests a picocode load, and then begins running millicode and executing from the new location.

Concluding remarks

The concept of having an internal code in the system, rather than an implementation consisting completely of hardware, provides a means to implement complex functions, and the flexibility to make changes to the system without changing hardware. With the development of millicode as the internal code of the processors, the hardware design was simplified, since millicode has an instruction execution similar to that of the system program, and thus has many of the same dataflow requirements as the system program code. Also, with the entire processor state and control capabilities mapped into R-unit registers, and a few milli-ops to read and modify these registers, millicode has access to most facilities and states in the processor. This provides a greater ability to change processor states or controls when a problem dictates it or when a new functional requirement is defined.

As both millicode and hardware have evolved since the G4 processor, enhancements continue to be added to increase performance and flexibility and to provide new function. Experience gained from previous systems is used to more accurately predict the areas that would benefit from functional enhancements and hardware disables. For instance, hardware interlocks on writes and reads to the R-unit were first introduced to prevent sequencing problems that had been seen on the G4 processor. Similarly, the SYSOP strategy has been changed to allow the millicode more flexibility for launching SYSOPs and recovering from errors encountered throughout the system.

For the z990 processor, improvements over previous processor designs were made to conform both to new system architecture and to a new internal processor design. The zone-based quiesce and TLB purging design [1], which allows fewer entries to be purged from the TLB on any given purge operation, relied on the versatility provided by R-unit control bits and PXLO and SYSOP commands, as well as the flexibility provided by picocode. All of these pieces were required to design, debug, and deliver this function. Other changes were made to the z990 processor to improve recoverability, enhance instrumentation, and provide hardware disables for new features such as the superscalar processor design. These are just a few examples of how hardware, relying on the flexibility that millicode provides, is able to implement performance enhancements and new functions.

Acknowledgments

During the evolution of systems that have used millicode as their internal processor code, there have been constant improvements to the hardware to increase the performance of the routines written in millicode, as well as improvements in the capabilities that are available to the millicode designers. We would especially like to acknowledge Timothy Slegel and Charles Webb for their design efforts in the creation and continued development of hardware to support the execution of millicode, Jennifer Navarro for the design of the programmable TLB, and the entire logic design team of the processor. We would like to acknowledge the entire global millicode team, including Steve Fellenz, Christopher Conklin, Janet Easton, Judy Johnson, Randall Philley, Dennis Weston, Brenton Belmar, Ambrose Verdibello, Pamela Dewey, Scott Davies, Andrew Piechowski, Harald Boehm, Eberhard Engler, Bernd Nerz, Hans-Georg Zipperer, Christopher Oelsner, and Klaus Meissner, for all of their contributions.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. C. F. Webb and J. S. Liptay, "A High-Frequency Custom CMOS S/390 Microprocessor," *IBM J. Res. & Dev.* **41,** No. 4/5, 463–473 (July/September 1997).
2. T. J. Slegel, R. Averill, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 Microprocessor Design," *IEEE Micro* **19,** No. 2, 12–23 (March/April 1999).
3. M. A. Check and T. J. Slegel, "Custom S/390 G5 and G6 Microprocessors," *IBM J. Res. & Dev.* **43,** No. 5/6, 671–680 (September/November 1999).
4. E. M. Schwarz, M. A. Check, C.-L. K. Shum, T. Koehler, S. B. Swaney, J. D. MacDougall, and C. A. Krygowski, "The Microarchitecture of the IBM eServer z900 Processor," *IBM J. Res. & Dev.* **46,** No. 4/5, 381–395 (July/September 2002).
5. T. J. Slegel, E. Pfeffer, and J. A. Magee, "The IBM eServer z990 Microprocessor," *IBM J. Res. & Dev.* **48,** No. 3/4, 295–309 (May/July 2004, this issue).
6. IBM Corporation, *z/Architecture Principles of Operation* (SA22-7832); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.

Received September 24, 2003; accepted for publication January 20, 2004; Internet publication April 6, 2004

Lisa Cranton Heller *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (cranton@us.ibm.com).* Ms. Heller received a B.S.E.E. degree from the Massachusetts Institute of Technology in 1984, joining IBM that same year. She is a Senior Technical Staff Member and was the millicode team leader for the z990 processor. During her career, she has worked in the areas of processor virtualization (SIE), quiesce design, and TLB. Ms. Heller is currently working on future IBM zSeries processors.

Mark S. Farrell *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (msf@us.ibm.com).* Mr. Farrell received a B.S.E.E. degree from the University of Pittsburgh in 1977. He joined IBM that same year to work in the processor microcode area. He worked on the architecture and design of many microcode projects for S/390* systems, and was the leader in the switch to millicode implementations for the CMOS processors. He is a Senior Technical Staff Member, and has been the millicode team leader for many of the recent CMOS processors. Mr. Farrell is an author of 27 U.S. patents; he has received two IBM Outstanding Technical Achievement Awards, three IBM Outstanding Innovation Awards, and two IBM Corporate Awards. He currently works on the development of future IBM server processors and systems.