# The IBM eServer z990 microprocessor

T. J. Slegel
E. Pfeffer
J. A. Magee

*The IBM eServer™ z990 microprocessor implements many features designed to give excellent performance on both newer and traditional mainframe applications. These features include a new superscalar instruction execution pipeline, high-bandwidth caches, a huge secondary translation-lookaside buffer (TLB), and an onboard cryptographic coprocessor. The microprocessor maintains zSeries® leadership in RAS (reliability, availability, serviceability) capabilities that include state-of-the-art error detection and recovery.*

## Introduction

The microprocessor in the IBM eServer* z990 was designed to optimize the performance of both newer and traditional mainframe applications while maintaining the outstanding reliability, availability, serviceability (RAS) features customers expect from IBM mainframes. Although some high-level concepts of the microprocessor design are based on those of the IBM eServer z900 and earlier generations, most of the internal structures and lower-level details are brand new for the IBM eServer z990.

Prior CMOS IBM mainframe processors [1–4] had a relatively simple microarchitecture and were optimized for traditional applications that ran on them. These applications tended to fully exploit the complex instructions in the z/Architecture* instruction set [5, 6]; a significant number were written in assembly language (or at least used it for performance-critical routines); and they stressed the storage subsystem. Therefore, the instruction pipeline in these earlier CMOS processors was not superscalar; that is, they could execute only one instruction per clock cycle. Instead, they achieved good performance by using additional hardware to execute the complex instructions quickly and had a very robust storage subsystem including large L1 caches and translation-lookaside buffers (TLBs). As a side note, the processor in the bipolar technology ES/9000* Type 9021 [7] was one of the very first commercial superscalar out-of-order-execution processors in the industry, but when IBM mainframes made the transition to CMOS, it moved to a simpler processor design point, since the performance improvement on applications did not justify the much more complex design point given clock frequency and time-to-market requirements.

As the mainframe has evolved, customers are now running a wide variety of applications and middleware on these systems. Newer applications are typically written in a high-level language such as C, C++, or Java**. This code tends to have characteristics very different from those of traditional code. Most significantly, it tends to use the simpler instructions in the instruction set and relatively infrequently uses the complex instructions for which prior processors were optimized. When this microprocessor was originally conceived, a design goal was that it must execute both traditional and newer types of code with high performance; therefore, the design team decided that a superscalar design point was again necessary.

In moving forward to the z990, a number of features in earlier processors have been maintained. These include millicode [8], which is the vertical microcode that executes on the processor, and the recovery unit (R-unit), which holds the complete microarchitected state of the processor and is checkpointed[1] after each instruction. If a hardware error is detected, the R-unit is then used to restore the checkpointed state and execute the error-recovery algorithm. Additionally, the z990 processor, like its predecessors, completely duplicates several major functional units for error-detection purposes and uses other error-detection techniques (parity, local duplication, illegal state checking, etc.) in the remainder of the processor to maintain state-of-the-art RAS characteristics.

[1] The term *checkpointed* here means that the instruction has completed, for the purpose of error recovery, and there is no need to retry it in the event of an error [2].



**0018-8646/04/$5.00 © 2004 IBM**

It also contains several mechanisms for completely transferring the microarchitected state to a spare processor in the system in the event of a catastrophic failure if it determines that it can no longer continue operating.

Another concern as the mainframe has evolved is floating-point performance. Since the mid-1960s when the S/360 architecture was originally defined, IBM mainframes have implemented a proprietary floating-point arithmetic based on a hexadecimal data format (HFP). In 1998, the G5 processor first implemented the IEEE 754 binary floating-point (BFP) standard [9]. It executed both data formats using a shared floating-point pipeline. However, when an application used BFP arithmetic instead of the traditional format, it received poorer performance; the BFP pipeline was effectively two cycles longer, and a new operation could be submitted only every other cycle. Since applications have recently begun to use BFP arithmetic as much as the traditional format, BFP arithmetic performance had to be substantially improved. Therefore, the z990 processor contains a new floating-point unit [10]. Although the two data formats again share a common pipeline, the new design provides equal performance for BFP and HFP operations.

The processor also contains several novel features never before seen on any general-purpose microprocessor. One of these is a huge secondary TLB (we call it the TLB2) that supplies virtual to real address translations in the event of a miss in the first-level TLBs. If there is a miss in the TLB2, addresses are translated by means of a programmable translation engine. Additionally, the processor contains several sophisticated mechanisms to minimize performance loss when broadcast purge TLB operations are required. Another new feature is the on-board cryptographic coprocessor. This very-high-performance and low-latency engine was designed to speed up Web transactions using the Secure Socket Layer (SSL) protocol. It implements the Data Encryption Standard (DES) and Secure Hash Algorithm (SHA) functions.

## Overview of superscalar microarchitecture

The microprocessor in the IBM eServer z990 is relatively uncommon compared with other modern processors because, although it is superscalar, it executes instructions in strict architectural order. However, it makes up for this by having a shorter pipeline and much larger caches and TLBs compared with other processors, along with other performance-enhancing features.

In the initial choice of a high-level microarchitecture for a processor, many factors have to be considered. The pipeline stages and overall length must be carefully chosen to achieve the optimal balance between frequency (with all else being equal, a higher-frequency target requires more pipeline stages) and the performance penalties associated with a longer pipeline (mispredicted branch and dependent data and address usage). The cache and TLB sizes have to be balanced to achieve good hit rates while not requiring excessive chip area and the associated difficulty of maintaining frequency targets that entails. Finally, and equally important, adding complexity to achieve performance must be balanced against product schedules. In the end, we believe that the microarchitecture of this processor is optimal for the CMOS technology in which it is implemented, and for the complex instruction set of the z/Architecture.

### *Processor pipeline*

The basic pipeline structure is presented in **Figure 1**, along with the functions performed by each of the major units. As shown here, instruction text is presumed to have been prefetched from the L1 instruction cache (I-cache) and latched in instruction buffers in the instruction unit (I-unit). There are two instruction decoders available that operate in the D1 cycle. Branch prediction is performed by the branch target buffer (BTB) in the cycles before decode. The BTB holds both target address and direction information based on prior branches it has encountered. There are also static direction-prediction algorithms for branches not found in the BTB at instruction decode time.

The next cycle is the AA1 cycle in the I-unit, which corresponds to the C0 cycle in the L1 data cache (D-cache). In this cycle, an operand address calculation (Agen) is performed. Note that the z/Architecture instruction set is not a load-store architecture. It allows a single instruction to fetch data from storage and simultaneously perform an arithmetic or logical operation on it [11], which would be equivalent to two RISC instructions. Therefore, operand addresses are calculated earlier in the pipeline compared with most RISC processors, but this can present performance bottlenecks in the event of address generation interlocks (AGI), where the address that is to be computed requires a general register (GR) that has not yet been updated. To help reduce AGI stalls, several bypasses are available. Load-type instructions can bypass data from the C2 cycle back to the Agen units. The Load Address instruction is effectively executed in the Agen units themselves, and results are saved on a stack to be used by future instructions which may require that updated register value before the register file is updated after the true execution cycle. Also, in this AA1 cycle, the I-unit performs superscalar grouping of instructions for execution. Once formed in the AA1 cycle, the group stays together until execution is complete. In parallel, the D-cache has its C0 cycle. Here, the access lookaside buffer (ALB), if the processor is operating in access register mode, and the absolute address history table (AAHT) are accessed. The

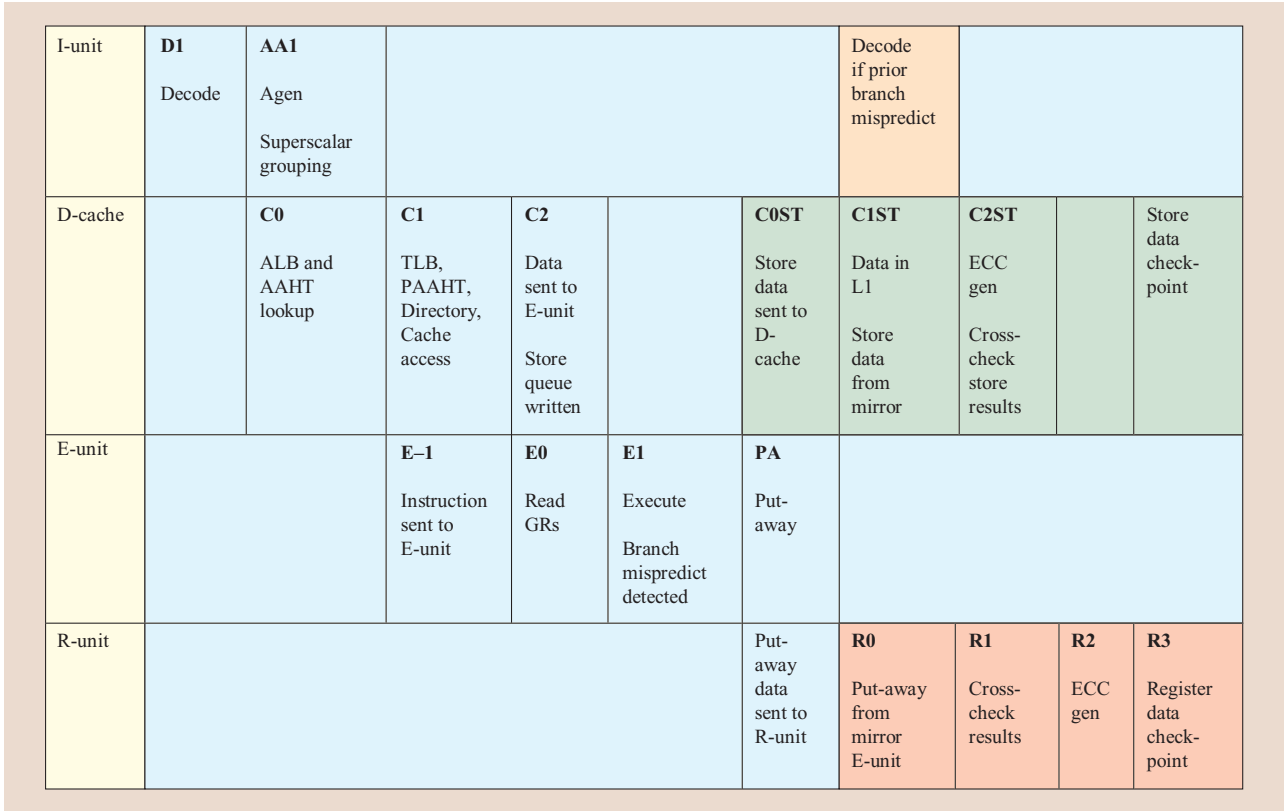


**Figure 1**

Processor instruction pipeline for most common instructions.

AAHT is used to predict the real (or, more precisely, absolute) address value of virtual address bits when the cache arrays are accessed in the next cycle [1]. If the instruction is a branch, this cycle also determines whether the target address has been predicted correctly by the BTB, or, if it has not been predicted by the BTB, the target address is computed.

The following cycle is the C1 cycle in the D-cache, where the main arrays are accessed. These include the D-cache TLB, the directory array (sometimes referred to as the tag array), the page absolute address history table (PAAHT), and the cache data array itself. The PAAHT can be thought of as a small and fast TLB that is accessed in parallel with the main D-cache TLB. There are two complete pipelines in the D-cache for processing requests. If all arrays indicate a hit for a given request, data is latched at the end of this cycle. For fetches, two independent pieces of cache data, corresponding to the two pipelines, can be returned to the execution unit in a single cycle. This cycle also corresponds to the E–1 cycle in the I-unit and execution units (E-unit, which comprises the fixed-point unit and floating-point unit), where control information is passed between units on how to execute this instruction group.

Following the C1 cycle is the C2 cycle in the D-cache, where cache data is returned to the E-unit. If there has been a miss in the TLB in the prior cycle, a request is sent to the translator unit (X-unit), where virtual-to-real address translation is performed. If it is a store-type instruction, the store queue is written with address and control information to track this store until it is completed. The C2 cycle also corresponds to the E0 cycle in the E-unit. Here registers are accessed and controls set up in preparation for execution.

The next cycle is the E1 cycle, in which arithmetic or logical results are computed in the fixed-point unit (FXU). Up to three instructions may be executed simultaneously in the FXU. If it is a floating-point operation, this is extended from E1 through E5 corresponding to the five stages in the floating-point unit (FPU) pipeline. If a branch instruction is part of the group, this is the cycle in which it is determined whether the branch direction has

been predicted correctly. If it is incorrect, the I-unit has typically prefetched the nonpredicted path and is ready to decode the correct path two cycles later.

The put-away cycle (PA) is where result(s) from the E-unit are written on two 64-bit result buses. There are copies of the general registers in both the E-unit and I-unit, and these are written on the PA cycle. Store data is sent to the D-cache on this cycle. Results are also sent to the R-unit in the PA cycle for comparison and checkpointing.

The remaining stages of the pipeline are for completing stores (shown in green in Figure 1) and for checkpointing results in the R-unit. None of these extra stages affect performance on mispredicted branches or in the event of data dependencies; their sole purpose is to allow for error checking and recovery. Here, results from the mirror E-unit are compared to ensure that there were no hardware errors in execution. Traditionally, IBM mainframes have duplicated these units not only to gain perfect error checking, but also to improve operating frequency by eliminating the need to include conventional error-checking techniques in critical paths. However, it is somewhat more expensive than traditional error checking in terms of area and power. In the early phases of the processor design, it was determined that duplicating the units for RAS purposes would have an adverse effect on the processor frequency because of the additional wiring to and from the duplicated units. Thus, a novel scheme was invented that runs the mirror units one cycle later than the main units. This allows the core to be floorplanned in such a way that the main units, along with the nonduplicated units, such as the caches, can be packed closely together, and the duplicated units can be placed further apart. Since they are running one cycle later, the frequency impact this would otherwise have is mitigated. The remainder of the R-unit pipeline generates ECC on the results, and this data is saved in the checkpoint array of the R-unit.

A high-level diagram of the major units is shown in **Figure 2**. Note that the duplicated I-unit and E-unit are not shown on the diagram.

In addition to the main instruction execution flow described above, several operations take place in other units. Data compression based on the Lempel–Ziv 2 algorithm [12] is supported in the coprocessor unit. The design point was significantly enhanced compared with the prior-generation z900 processor by nearly doubling performance for most operations. This unit also executes the z/Architecture instruction Translate (TR) and related character-translation instructions, at nearly twice the performance of the z900. Also, the new design point allows for caching of translation tables across instructions to improve the startup time for these instructions. This coprocessor unit includes the new cryptographic engines that are discussed later.

### *Dual cores and system interface*

A design goal of the IBM eServer z990 system was to pack as much processing power into a single box as was practical, while meeting cost targets for the final product. To this end, it was decided the optimal solution was to put two complete processor cores on a single microprocessor chip; this is called a dual-core design point. Each processor core runs totally independently of the other; however, they do share a common interface with the system control element (SCE), as shown in the figure. The dual-core processor chip significantly reduces the number of I/O signals to and from the SCE, improving packaging efficiency.

The interface with the SCE consists of two 16-byte data buses plus a number of smaller address and control buses. These data buses operate independently, so that one of the cores can be using one of the buses while the other core can be using the other in any combination of fetching and storing data. Similarly, a single core can be using both data buses for any combination of fetches and stores. The logic generally gives priority to fetches over stores, while ensuring that both processor cores receive equal bandwidth. Although the processor cores run at twice the clock frequency of the SCE, the interfaces run at the slower SCE frequency. The total bandwidth available from the L2 cache and main memory to both cores on a chip is in excess of 19 GB per second.

The dual-core design significantly increases the complexities associated with the RAS implementation. When one core detects a hardware error and initiates its recovery process, the other core on the chip is also forced through a similar recovery sequence, even though no error occurred on this other core. This is to simplify the error-recovery protocols on the shared interface to the L2 cache. Both cores complete the recovery process at about the same time, and they are then allowed to continue normal operation. If an error is so severe that processing cannot continue, both cores are placed in the checkstopped[2] state at the same time, even though only one core may have had the error. This scheme simplifies the hardware implementation in the event that the error was in the shared interface. However, this design point does require there to be at least two spare cores available in the system in all configurations to facilitate transparent processor sparing [2], which allows the complete state of a checkstopped core to be moved to a spare core without software intervention.

[2] In a checkstopped state, all processing has been halted.

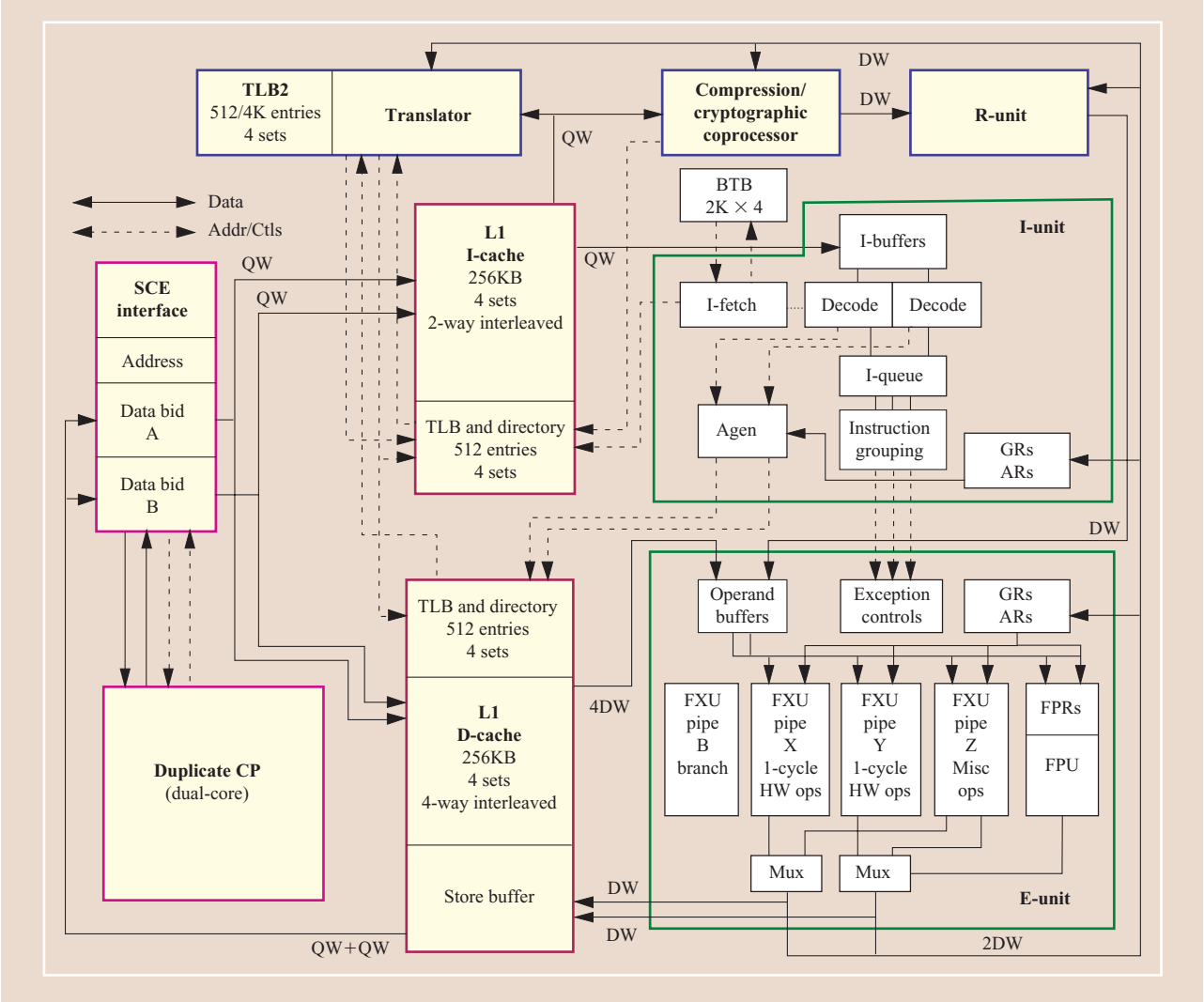


Figure 2

Diagram of z990 processor. (DW: doubleword; QW: quadword; ARs: access registers; FPRs: floating-point registers; HW ops: hardware executed instructions.)

### *Instruction set architecture enhancements*

In addition to the numerous new microarchitecture features in the z990, there are a number of enhancements to the instruction set architecture that can be used by compilers and OS software to enhance performance and add new function. The most significant of these for application programs is the Long-Displacement Facility, which adds 44 new instructions and enhances the behavior of 69 other instructions. Previously, most instruction formats that accessed storage contained a 12-bit unsigned displacement field. The Long-Displacement Facility changes some of these formats to a 20-bit signed displacement, providing an addressing range of ±512 KB from the base and/or index register value. This larger range enhances application performance by reducing the need for multiple base registers and hence allows these registers to be used for other purposes; there is less saving and restoring register contents to temporary memory locations. This extra byte of displacement is placed in a previously unused byte in the instruction text for those 48-bit instruction formats that had it available. Note that this new high-order byte is not contiguous with the prior 12-bit displacement field, but this hardly matters, since only compilers, assemblers, and the hardware itself are concerned with the actual machine language format of the instructions.

Another new instruction available for OS usage is Invalidate DAT (dynamic address translation) Table Entry

(IDTE). Previously, two instructions were available to take virtual storage away from an application. One invalidated a single page-table entry and its corresponding entry in the TLBs. The other could invalidate any type of DAT-table entry, but then it invalidated all entries in the TLBs. Unfortunately, this overpurged the TLBs in many cases and required the relatively lengthy process of retranslating virtual addresses to refill the TLBs. The new IDTE instruction purges any number of entries the OS desires, and then purges the TLBs of just those corresponding entries, which improves performance significantly.

## Instruction execution

The z990 processor executes the vast majority of the complete z/Architecture instruction set totally in hardware. The remainder of the instructions are implemented by millicode. In general, millicode is used for the most complex instructions and also for the infrequently used z/Architecture instructions when the cost of a hardware implementation is not justified. Millicode is also used to perform a variety of service functions in the system. From a hardware point of view, there is little difference in executing millicode instructions or z/Architecture instructions. There is no separate execution engine for millicode; instead, it uses the normal instruction pipeline. Throughout the rest of this paper, the descriptions apply equally to millicode and z/Architecture instructions.

### *Instruction fetching and branch prediction*

Instructions are fetched from a very large (256-KB) L1 I-cache. It is four-way set-associative and two-way interleaved. The interleaving is on 16-byte quadwords (QW), which allows fetch data to be sent to the I-unit concurrently with a line-fill operation from the L2 cache. The corresponding I-cache TLB is also large: 512 total entries and four-way set-associative.

The I-cache access time is three cycles, with the first and last cycles consisting primarily of receiving the request from the I-unit and returning the instruction text to the I-unit, respectively. Normally, these cycles are completely hidden and do not affect performance. Instruction fetching is usually far ahead of instruction decoding. When a branch is about to complete execution, the I-unit prefetches I-text down the path that was not predicted prior to the branch execution. This allows the processor to be ready to decode the nonpredicted path immediately in order to minimize the penalty from mispredicted branches.

Instruction text from the I-cache is temporarily saved in instruction buffers (I-buffers) prior to being decoded. In the event that instruction text necessary for decoding the next instruction has not been prefetched into the I-buffers, these buffers can be bypassed and instruction text transferred directly from the I-cache to the instruction decode registers. There are a total of 16 I-buffers available, each a quadword wide. These are composed of eight main buffers for sequential and predicted paths, and four sets of two buffers for prefetching nonpredicted paths. Since dynamic branch prediction occurs in the instruction-fetching portion of the pipeline, not at decode time, the fetching logic works closely with the branch-prediction logic to prefetch instruction text down the predicted instruction stream. The I-buffers, therefore, hold this predicted stream that includes accounting for branches that are predicted to be taken.

From a very high-level point of view, the branch-prediction logic, including the branch target buffer (BTB), is similar to that used in the prior z900 processor. The BTB contains a total of 8K entries and is four-way set-associative. A significant number of branches in the z/Architecture, especially when older code is being executed, are not relative and include a base register, an index register (optional), and a displacement. Therefore, to achieve acceptable performance it is necessary to predict the target address in addition to predicting whether or not a branch is taken. Thus, the target address is saved in the BTB in addition to the instruction address of the branch itself. Unlike most other processors, the BTB keeps only branches that are predicted to be taken and uses a three-state history: strongly taken, weakly taken, and invalid entry. Placing only taken branches (as opposed to including not-taken branches) in the BTB allows for high direction accuracy on dominantly resolved taken branches. A static prediction algorithm, based on the instruction type at decode time, is used to predict branch direction for those not found in the BTB. Not placing branches that are dominantly resolved not-taken in the BTB yields the same result as placing them in the BTB, since the static algorithm usually predicts them as not-taken. The performance tradeoff for entering nondominant resolution versus the additional predicted taken entries that can be placed in the BTB makes it advantageous not to write not-taken branches into the BTB.

There can be up to four unresolved branches active in the pipeline at once from the decode stage through put-away, and also up to four prior to decode in the instruction buffers.

### *Instruction decoding and superscalar grouping*

Two instruction decoders are available, and each of them can decode any type of instruction. Most instructions can decode in a single cycle. However, some complex instructions take more than one cycle; these include instructions that have two independent storage operands. Most other types of instructions can decode together as a pair; the only significant exception is a pair in which a branch-type instruction is the older instruction.

If the pipeline is flowing freely with no stalling, instructions proceed to the superscalar grouping station. Alternatively, if there are prior stalls in the pipeline such that older instructions are still waiting at the grouping station, the newer instructions just decoded move into an eight-entry instruction queue. This queue contributes to tighter packing of instructions at execution time and also allows address generation and cache accesses to be initiated earlier while the corresponding instructions wait in this queue.

Up to three instructions may be grouped together for execution. The largest combination of instructions is a branch instruction, which must be the oldest instruction in the group, followed by two other simple instructions. These simple instructions can consist of any combination of most single-cycle instructions including arithmetic, logical, loads, and stores. Two store-type instructions in a group are not permitted, although any other combinations of load-type and store-type instructions are allowed. The idea of requiring a branch instruction to be the oldest in a group is somewhat counterintuitive, since it would be more natural to execute a branch instruction after a prior instruction has set the condition code. However, the chosen scheme avoids the frequency impact of having to set the condition code and resolve the branch direction in the same cycle, given the very short pipeline of this processor. Our studies showed a very minimal instructions-per-cycle (IPC) impact with the branch required to be the oldest instead of a more general scheme in which the branch can be anywhere in a group, and the frequency benefit more than made up for the lost IPC.

A number of factors can limit the ability of the hardware to create efficient execution groups. Address generation must be completed before, or in the same cycle in which, the corresponding instruction is grouped. Therefore, if the newer instruction has an address-generation register dependency from an older instruction already in the group, the newer instruction is prevented from grouping. Also, read-after-write (RAW) register dependencies are generally not permitted, but this has no performance impact because the two instructions would not be able to execute in the same cycle anyway.

However, one type of RAW dependency, called *operand forwarding*, is allowed. This permits an older load-type instruction that is updating a given register to group with a newer instruction that is using that same register as a source for execution purposes. The load-type instruction must be a simple load from storage or a register-to-register load and not an arithmetic/logical operation. By simply changing the bus routing at the input to the execution pipelines, operand forwarding can be accomplished with minimal complexity and with a significant IPC improvement.

Another type of allowed RAW dependency we call *condition code forwarding*. Unfortunately (for modern software and hardware), when the original S/360 architecture was defined nearly forty years ago, it included a special condition code setting for certain arithmetic instructions if overflow is detected. This makes it difficult for a C/C++ compiler to implement a common construct such as the following:

```
int a, b;
. . .
if ((a+b)>0)
  . . .
```

The problem is that the integer add instruction for the `a+b` calculation might possibly overflow and set the condition code to indicate this. However, the C language is defined to ignore overflow and just check for a positive result. Therefore, C/C++ compilers typically insert a Load And Test instruction immediately following the add to force the condition code to indicate simply a negative, positive, or zero result (with no indication of overflow). The processor optimizes this compiler construct by allowing grouping of instructions that can set an overflow condition code followed by a Load And Test instruction, even though it is a RAW dependency, to simply set the condition code by suppressing the overflow case. If the mask in the program status word (PSW) is set to give an exception on overflow, this must be handled differently.

### *Execution pipelines*

After grouping, instructions move to the E-unit for execution. Several execution pipelines are available for instructions:

- B pipeline – Executes branches; consists of control logic that simply resolves whether or not a branch is taken.
- X pipeline – General-purpose execution pipeline for simple instructions. Also, executes access-register-type instructions and millicode special instructions.
- Y pipeline – General-purpose execution pipeline for simple instruction (does not handle access-register-type or millicode special instructions).
- Z pipeline – Consists of dedicated special hardware for complex instructions including fixed-point multiplies and decimal instructions. In other cases, a virtual Z pipeline is created by logically combining the X and Y pipelines into a 16-byte-wide dataflow for processing Move Character and other storage-to-storage instructions. Instructions that use the Z pipeline always execute alone.
- FPU pipeline – Dedicated unit for processing all floating-point instructions including both IEEE BFP arithmetic and traditional IBM hexadecimal arithmetic. Fixed-point divide instructions are also executed here.

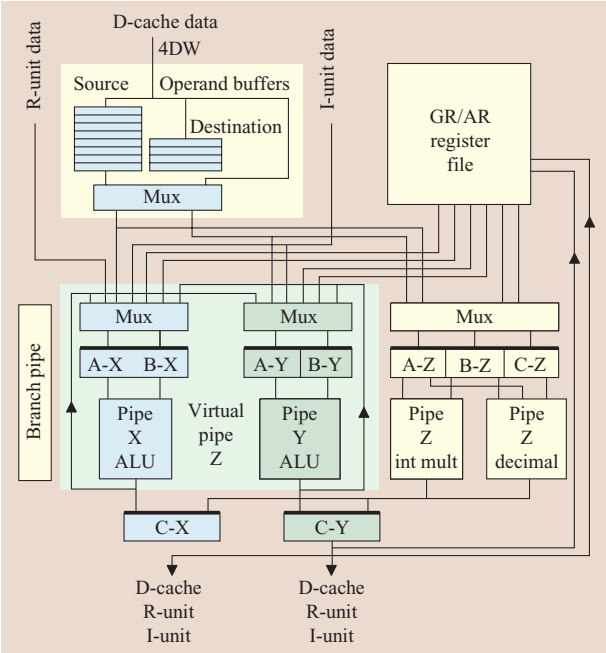


**Figure 3**

Diagram of FXU. (int mult: integer multiplier; decimal: decimal execution unit.)

**Figure 3** is a diagram of the FXU showing the execution pipes and some of the dataflow switching. The Z pipeline is shown both as a dedicated logic section and as a virtual grouping of the X and Y pipelines. The two result buses can have independent arithmetic/logical results or stores on them associated with the X and Y pipelines. The buses can also contain a single 16-byte result associated with the Z pipeline.

Unlike many processors, there is no load-use or result-use penalty (except that RAW-register-dependent instructions are not allowed to group together). In other words, when a register is loaded from storage, that data is immediately available in the next cycle to be used as an operand in a dependent instruction; or, in the case of operand forwarding, it is even available in the same cycle. For arithmetic/logical results, this data is also available for use in an instruction in the next cycle. These features provide a significant performance boost without the complexity that an out-of-order execution processor would otherwise have to have in order to attempt to hide these latencies.

## Data cache loads and stores

The z990 is the first z/Architecture processor, including its predecessor architectures S/360 through ESA/390, to allow storage operand accesses out of their architectural conceptual sequence. Even the bipolar ES/9000 Type 9021 processor, which performed fairly aggressive out-of-order execution, still accessed storage in-order. Prefetching storage operands early and out-of-order allows the processor to hide much of the cache-miss latencies that other processors hide via out-of-order execution, but with a considerably simpler design point.

The z/Architecture presents some significant challenges to a processor which attempts to perform out-of-order storage accesses. It has one of the strongest sets of storage ordering and consistency rules of any popular instruction set, particularly when dealing with multiprocessing (MP) environments. These strict rules provide some significant benefits to software, since they implicitly eliminate some potential bugs that can occur when programming in an MP environment. On the other hand, they make it difficult to implement the hardware for a processor that performs well, and particularly for one that attempts to do out-of-order storage accesses.

The design philosophy of the storage-management algorithms in the z990 processor was to internally violate the architecture rules to achieve optimal performance. Then, a significant amount of complex logic was added to determine whether software running on this processor or another processor in an MP system might be able to observe the hardware not following the storage ordering and consistency rules. If this logic detects that observation of an architectural violation might be possible, the hardware invokes mechanisms that allow the processor to return to an architecturally correct state. In the end, as far as software is concerned, the processor obeys all of the storage rules as defined for the z/Architecture.

There are several distinct parts to the operand storage accessing logic. Operand fetch and store requests are initially generated in the I-unit and are then sent to the D-cache for processing. Data is returned from the cache to the operand buffers, which are physically located in the FXU. Store operations involve a store queue, which holds address and control information, and a store buffer, which holds the actual data.

### *D-cache structure*

Operand fetch and store requests are generated by the Agen units. For most simple instructions, two independent storage requests can be made in a single cycle to the D-cache. The basic piece of operand data fetched or stored is an eight-byte doubleword (DW). For instructions that access more than one DW, or for an operand that spans a DW boundary address, multiple storage requests are made to the D-cache.

Like the I-cache, the data cache is also large at 256 KB, four-way set-associative, with a line size of 256 bytes. It has four independent interleaves based on address bits, each a DW wide, which may be accessed simultaneously.

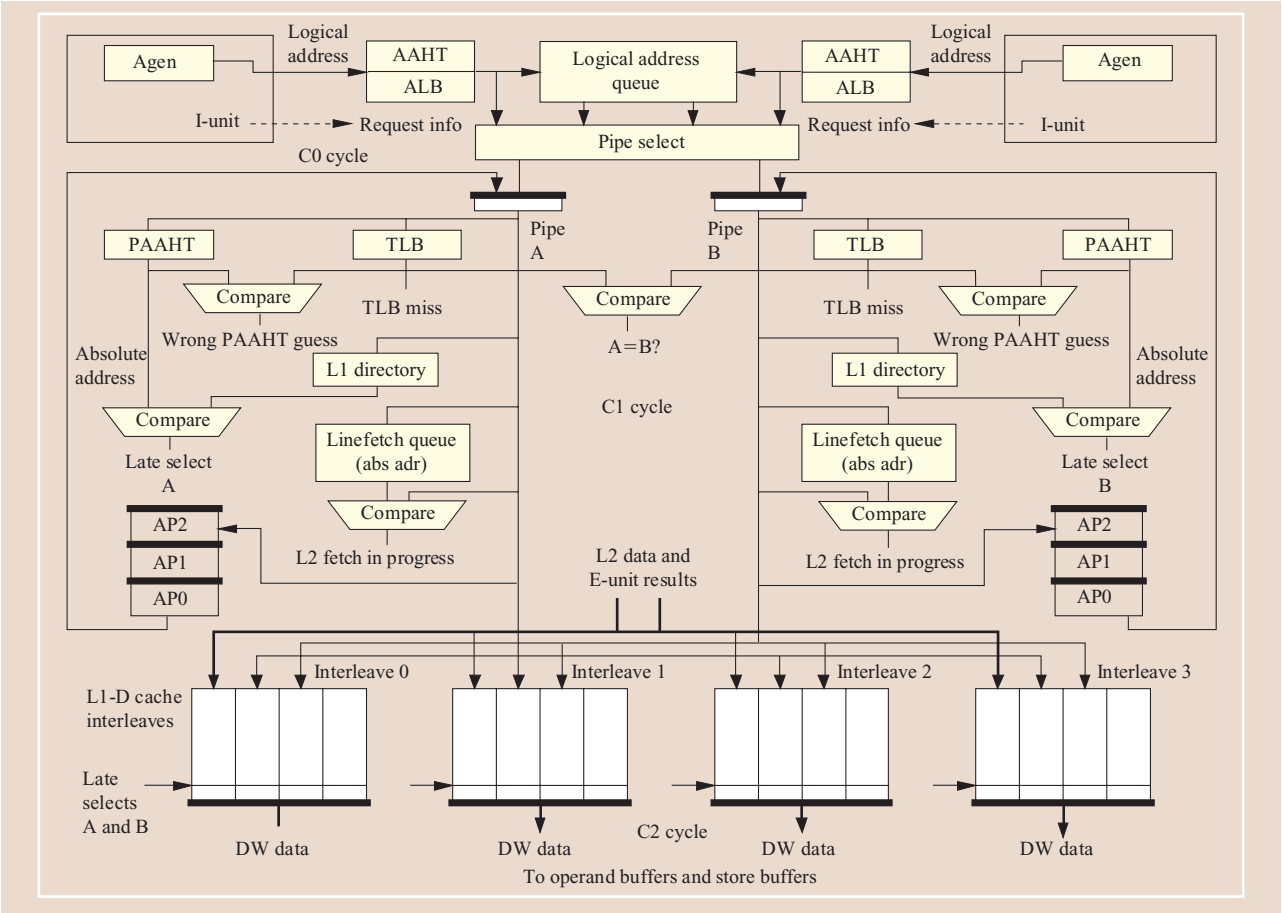


**Figure 4**

D-cache diagram.

Any combination of fetch, store, and line-fill operations from the L2 cache may be occurring simultaneously. The TLB associated with the D-cache is 512 total entries and is four-way set-associative.

The D-cache structure is shown in **Figure 4**. Two independent pipelines are available in the D-cache for processing requests; one corresponds to even cache lines and the other to odd cache lines. Requests are steered to one or the other based on bit 55 of the logical address. However, if two simultaneous requests have the same address bit 55, in some cases (for example, if the hardware determines that the two requests are for the same cache line) one of the requests is routed to the "wrong" pipeline, and in many cases it can also access the cache with no performance penalty. A given request is simultaneously searched in the PAAHT, TLB, and directory arrays to determine whether there is a hit. In the same cycle the request goes to the cache data arrays from which the fetched data will come, on the presumption that there will be a hit in the other arrays. If there is a miss in the PAAHT, there will be a two-cycle penalty to obtain the correct translation from the TLB. A miss in the TLB will cause a request to be sent to the secondary TLB and, if it hits, will result in a seven-cycle penalty. If there is a miss in the D-cache itself, the request is forwarded to the L2 cache for servicing. In the event of a miss (PAAHT, TLB, or cache directory), the request is recycled in the AP buffers until the request is satisfied.

### *Data fetches and operand buffers*

The complications of out-of-order storage accesses manifest themselves most clearly in the design of the operand buffers. These buffers hold operands that have been prefetched, but their corresponding instruction may not be ready for execution. There are eight DW buffers available for normal operand fetches. An additional four DW buffers are available for the second operand of storage-to-storage-type instructions.

Logic in the D-cache constantly monitors cross-interrogate (XI) requests received from the L2 cache to determine whether another processor might be storing into an operand that has already been prefetched into the operand buffers, hence allowing observation of an out-of-order fetch. If this is detected, the operand buffers implement a mechanism of refetching the data to ensure that the latest copy is being used. The entire pipeline is then flushed to clean up the state of any other prefetched instructions. Since the processor does not implement out-of-order execution, it has no natural mechanisms for nullifying a partially completed instruction that operates on more than a single DW of data. Hence, there is the added complexity of refetching operands to complete the instruction without violating any architectural rules. In this mode, a request is initiated for a single DW of data, and no further fetch requests are initiated until data is returned for this earlier fetch request. Fortunately, since this mode of operation seldom has to be invoked, there is no measurable IPC impact.

### *Data stores*

The D-cache implements a write-through policy with store-type data being written into the cache while the off-chip L2 cache implements a write-back policy. Store data is aligned with a DW boundary in the FXU and FPU and then sent to the D-cache, where it is written to the cache. In parallel the store data is copied into one of four 32-DW store buffers. These buffers are ECC-protected and hold the data of an entire instruction until it is checkpointed, when it can be released and sent to the L2 cache (the maximum amount of store data for a single hardware-implemented instruction is 256 bytes). From the time a store request is originally generated by the I-unit, information associated with the request is held in a store queue until all of the data has been sent to the L2 cache.

The z990 also implements the concept known as *silent stores*. If the DW of data being stored is identical to the value of that DW already in memory for that same address, there is no need to actually perform the new store operation. When a store queue entry is initially formed, the contents of the DW are read from the D-cache and saved in a special buffer. This buffer must also be monitored for XI requests from other processors so that this is truly the most recent copy of the data in the system. Later, when the store is actually performed from the FXU or FPU, the new store data is compared to the old data read previously. If they match, the store is not sent to the L2 cache, although it is still written into the L1 D-cache. With up to 16 processors storing in a shared L2 cache using a write-through L1 policy, silent stores significantly reduce the amount of traffic and increase system performance.

Unlike most RISC architectures, the z/Architecture requires, in general, that stores into the instruction stream (self-modifying code) behave as a program expects; that is, even if a program modifies the very next instruction, the processor must execute the new version of that instruction. This requirement presents tremendous challenges to designers and requires considerable complex hardware to implement. The z990 processor tracks instruction fetches on a 256-byte-cache-line basis. If any store takes place in a cache line that has been fetched from the I-cache and all instructions contained in it have not yet been completed, the entire pipeline is flushed as soon as the store completes. Obviously, this scheme can adversely affect performance for self-modifying code,[3] or for a program that keeps data in the same cache line as instructions. However, some scheme such as this is necessary to achieve correct architectural results.

## TLB2 and the programmable translator

We have found that many applications that run on zSeries systems can obtain significantly better performance by using much larger TLBs than are common in the industry. Part of this is due to the characteristics of the applications themselves, but another factor is that many customers run numerous z/OS* and Linux** images simultaneously in a logically partitioned (LPAR) environment. With Linux, each image has its own virtual machine environment created by z/VM* and actively uses numerous virtual pages. Obviously, since each virtual page requires a TLB entry, very large TLBs are required to avoid thrashing when context switches are performed among the operating systems and virtual machines.

### *TLB2*

The z990 microprocessor provides a TLB arrangement which advantageously uses two buffers: a relatively small first-level TLB1 and a larger second-level TLB2. The second-level TLB feeds address translation information to the first-level TLB when the desired virtual address is not contained in the first-level TLB. The TLB2 comprises two four-way set-associative subunits: one, called the Combined Region Segment Table Entry (CRSTE) TLB2, covers the higher-level address-translation levels; the other one, the Page Table Entry (PTE) TLB2, covers the lowest translation level. An advantage of this scheme is that the output of the CRSTE TLB2 is a valid page-table origin when a match is found for the higher address bits and a valid entry was built before. In this case, since all accesses to the higher-level translation tables (region and segment tables) are bypassed, there is a considerable performance

[3] Self-modification of code is rarely done.

gain when there is a hit in the CRSTE TLB2 but a miss in the PTE TLB2. With this feature, the start address of the page table can be found within one cycle and can be used for the last table access to obtain the absolute address. A diagram of the TLB2 is shown in **Figure 5**.

The linkage of the CRSTE to the PTE TLB2 is established by means of seven bits of the segment index from the full 64-bit virtual address. These bits serve as an index address covering the address range of the CRSTE TLB2; the same information is used as tag information in the PTE TLB2 and is used as a quick reference for any lookup operation in order to find the absolute address of the relevant virtual address translation.

### *Programmable translator*

The z/Architecture provides software with numerous different modes for defining virtual address spaces. These modes offer tremendous flexibility and make software running on zSeries processors very robust, but they also impose complexities on a processor implementing the z/Architecture. The translator unit comprises a new control concept to ease the implementation of the complicated algorithms for dynamic address and access-register translation in a virtual guest environment. Instead of a hardware state machine that has been used in prior processors, the overall control of the unit is accomplished by an embedded programmable processor called a *picoengine* with its control program stored in a small RAM, called *picocode RAM*. The programmable translator has, essentially, the same performance as prior translator designs that were implemented purely in hardware.

The main advantages of the picoengine-based translator are as follows:

- All dataflow control functions are programmable. If a bug is found late in the development cycle, it can easily be fixed with a simple change to the contents of the picocode RAM. This RAM is loaded during the power-on reset phase of the system.
- New translation modes can be added to the instruction set architecture after the processor is committed to silicon.
- Design changes do not affect the cycle time of the control logic; in general, they can be implemented as picocode change.
- The picoengine is composed of standard logical building blocks (picocode RAM, branch decoder, etc.), which simplifies the problem analysis.
- Error checking is much easier to implement than in a complex state machine, since simple parity checking of the picocode RAM provides good coverage of the state controls. The remainder of the engine is checked using traditional methods.

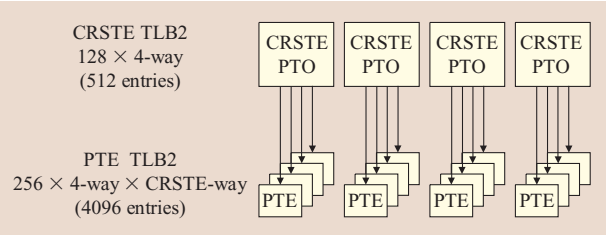


**Figure 5**

TLB2 diagram. (PTO: page table origin.)

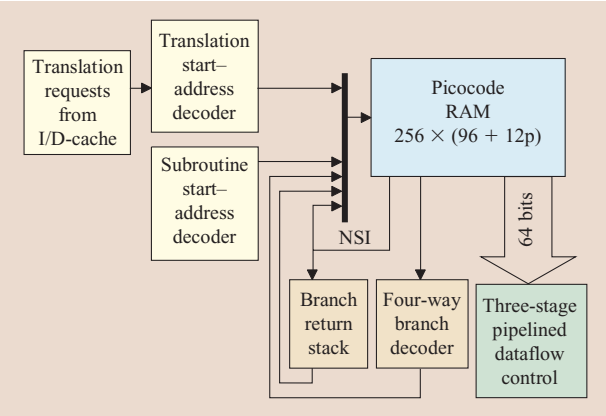


**Figure 6**

Diagram of picocode engine.

A diagram of the picocode engine is shown in **Figure 6**. When the caches miss in their TLB1s for a virtual address, the request is sent to the TLB2 and translator unit. If the request also misses in the TLB2, the translator unit decodes the request to obtain the starting address for one of the numerous translation algorithms stored in the picocode RAM. The picocode instructions are horizontally organized and are executed in one microprocessor clock cycle. They are of two different types: Either they control the multiplexers of the dataflow part (three-stage pipelined dataflow control) or they are used as branch instructions to transfer control to one out of four different branch targets (four-way branch decoder) if a preset condition is met. If no condition is met, or if the instruction type is a control instruction, the next sequential instruction (NSI) address stored in RAM is used to access the next instruction. Subroutine execution is supported by means of a hard-wired branch address decoder combined with a branch-return stack. When the translation operation is completed, the results are

returned and are stored in the TLB2 and the TLB1 from which the request originated.

### *TLB purge operations*

One of the drawbacks associated with large TLBs is the inordinately large performance loss when they must be purged. Several instructions in the z/Architecture require the TLBs to be purged of all entries or of selected entries. The Purge TLB (PTLB) and Compare And Swap And Purge (CSP) instructions respectively cause the TLBs to be completely purged on this processor or on all processors in the system. The Invalidate Page Table Entry (IPTE), Set Storage Key Extended (SSKE), and the new Invalidate DAT Table Entry (IDTE) instructions cause TLB entries to be selectively purged on all processors in the system. When TLBs on all processors have to be purged, it causes the entire system to be quiesced; this quiesced state is necessary so that the TLBs on all processors can be updated atomically with the resource being modified. This is called a broadcast purge operation.

A z990 system can have up to 64 physical processors installed (with up to 48 being normal processors accessible to a customer) and up to 60 logical partitions (LPARs). It has been shown on a prior-generation 16-way z900 system that up to ten percent of all time was spent idling by processors in the quiesced state or waiting for the last processor to reach a quiesced state. This problem had been evident for some time and was partially solved on the G5 processor [3]. Although those earlier mechanisms were implemented on the z900, it still had this very significant performance loss due to quiesce effects. To make matters worse, the performance loss grows with the square of the number of processors. Therefore, the z990 processor implements several new features to combat this system-quiesce performance loss for TLB purge operations.

The first new feature is that each TLB2 entry stored in the higher-level subunit is tagged with an identifier to indicate which LPAR partition created that entry. This allows several improvements to purge instructions:

- It is possible to keep the entries for several different LPARs in the TLB2 at one time. This significantly improves performance when numerous z/OS or Linux images are running on the system.
- A PTLB requires only those entries in the TLB2 that were formed by the currently active LPAR partition to be purged. On broadcast purge CSP instructions, only those entries must be purged in which there is a match between the LPAR identifier stored in the TLB2 and the LPAR partition of the quiesce initiator processor. Similar limited purges are implemented for IPTE and SSKE.

Another new feature is LPAR partition filtering for broadcast purge operations. Previously, all processors had to wait for the last processor to respond to the broadcast quiesce. The enhancement added to the G5 processor generation was that after responding, a processor could continue with normal work subject to the restriction that it had to stop if it missed in its TLB. After the last processor responds to the quiesce request, the restrictions are lifted. With partition filtering, when a processor initiates a broadcast purge operation, only those other processors which are currently operating in the same LPAR partition as the initiator respond to the broadcast immediately. Other processors perform the TLB purge operation when they are not doing other useful work. But the real gain is that fewer processors have to respond and hence overall system performance is increased.

Finally, the z990 implements address filtering for certain broadcast purge operations. When a processor receives a broadcast IPTE purge request, it saves the page index portion of the address; then it resumes normal processing subject to the restriction above. However, if it misses in the TLB and has to translate a virtual address, it is allowed to continue as long as the page index (or indices in the event of a pageable guest) needed for translation does not match the page index that was saved from the prior broadcast IPTE. If they do not match, the processor continues executing instructions. A similar mechanism is implemented for broadcast SSKE purge operations, but here a portion of the absolute address is saved and compared in the event of a storage key miss in the TLB.

## Cryptographic coprocessor

IBM mainframes have supported hardware-accelerated execution of cryptographic operations since the introduction of the Integrated Cryptographic Facility (ICRF) for the ES/9000 Type 9021 in bipolar technology. The ICRF provided support for Data Encryption Standard (DES), Message Authentication Code (MAC), and Personal Identification Number (PIN) algorithms. Encrypted keys were managed and stored by means of an external piece of hardware.

When IBM mainframes moved to CMOS technology, the ICRF was migrated and enhanced to include state-of-the-art algorithms and to provide support for the public key security architecture (PKA), the base of almost all secure Internet applications. Prior to the advent of the Internet, hardware assistance for encryption was practical only on specialized servers inside large commercial institutions, primarily in the banking industry. With the increasing prevalence of encrypted electronic communications on the Internet, microprocessors performing encryption without hardware assistance spend a very significant amount of processing time executing the software routines for these operations. However, all of

the IBM hardware assist engines developed previously had the downside of significant startup latency costs to communicate with the hardware external to the processor. This was particularly costly for short cryptographic operations in which the startup latency would be a significant fraction of the overall transaction.

For the z990, it was decided that the microprocessor core itself had to include clear key cryptographic engines to make the execution of the most frequently used functions more efficient. PKA requires three types of cryptographic functions. Those functions best suited to implementation on the core are symmetric key encryption/decryption and secure hashing. DES was chosen as the symmetric function and (Secure Hash Algorithm) SHA-1 was selected for secure hash. The third component of the PKA, an asymmetric encryption function that allows the symmetric keys to be transmitted securely via a network not guaranteed to be secure, does not provide appreciable advantage because the function is used relatively infrequently compared to the others.

New z/Architecture instructions were defined that allow software to access these functions. These include Convert Message In CBC Mode (KMC), Convert Message In ECB Mode (KM), Convert Message Authentication Code (KMAC), Compute Middle Message Digest (KIMD), and Computer Last Message Digest (KLMD).

**Figure 7** shows a diagram of the cryptographic coprocessor. Dataflow to and from the cryptographic engines makes use of components shared with the compression and character-translation engines. Shared 32-DW-deep first-in, first-out (FIFO) registers are used for all of the coprocessor functions.

When a cryptographic instruction is encountered, the corresponding millicode routine is executed. A coprocessor command register is written that activates the appropriate cryptographic engine. Millicode then writes the input FIFO with initial vectors and (for DES operations) one, two, or three keys. The engine has a defined handshake protocol to interact with the FIFO, and loads that startup information into corresponding registers within the engine. Millicode provides the input FIFO with operand data from storage and polls the output FIFO for results to be stored. Millicode writes the control register to let the engine know when the last operand has been delivered to the FIFO. After the last data has been processed, the engine sets the status register to indicate that the final results are available in the output FIFO. This, along with the indication in the status register that the FIFO is depleted, marks the end of the instruction execution.

To achieve a high degree of RAS, the SHA engine has parity checking on the dataflow. For the DES engine, due to the nonregular structure of the algorithm, it was more practical to duplicate the logic and compare results. In the event that a hardware error is detected during a cryptographic operation, the instruction can be re-executed starting from the last checkpoint.

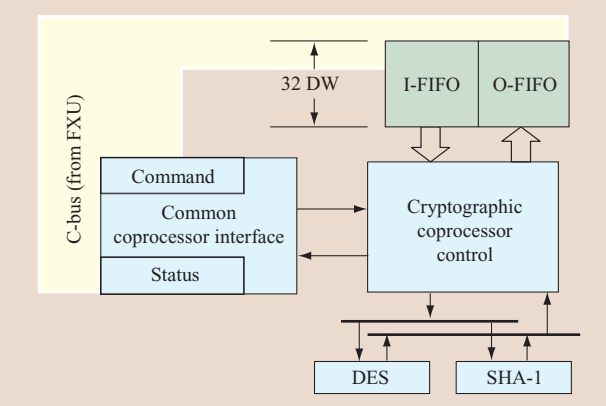


Figure 7

Diagram of cryptographic coprocessor.

The DES engine bandwidth is 3.8 Gb/s and the SHA-1 engine bandwidth is 3.1 Gb/s. These performance levels compare favorably with ASIC designs, and are unprecedented in a microprocessor core.

## Concluding remarks

The z990 microprocessor is manufactured in IBM CMOS 9S technology. This 0.13-$\mu$m technology includes silicon-on-insulator (SOI) transistors and eight levels of copper interconnection wiring. The chip, containing two complete processors, measures approximately 19 mm $\times$ 14 mm. It contains roughly 121 million transistors, of which 84 million are in array structures and the remainder in combinatorial logic. At its 1.2-GHz shipping frequency, the chip (with both cores operating) consumes 55 W of power. A micrograph of the chip is shown in **Figure 8**, with the various units labeled for microprocessor core 0. Core 1 is an exact replica, but is rotated 180 degrees with the two cores arranged in a "staircase" layout around the shared L2 interface logic.

The processor provides a dramatic performance increase over its predecessor, the z900, and excels at both traditional mainframe applications and newer applications. It also maintains its leadership over all competitive servers by providing virtually complete error-detection and error-recovery capabilities. The logical and physical design points of the microprocessor are also very flexible; we anticipate that they will be usable for the next-generation IBM eServer after it has been remapped into an even more advanced CMOS technology and further functional

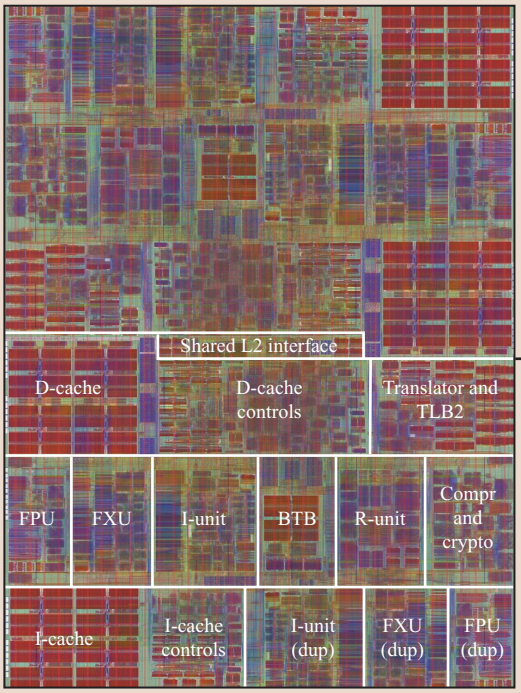


**Figure 8**

Micrograph of z990 processor chip.

and performance enhancements have been added to the basic design point.

## Acknowledgments

A great many individuals have been involved in the development of the z990 processor from its earliest conceptual stages, through logic and physical design, performance analysis, verification, and finally system test. Klaus Getzlaff, Allan Hartstein, Thomas Puzak, Charles Webb, Dale Hoffman, and Tom McPherson were instrumental in the early high-level concept and design phases. Key logic designers included Brian Prasky, Mark Check, Bruce Giamei, John Liptay, Sam Thomas, Eric Schwarz, Fadi Busaba, John Rell, Steven Carlough, Chris Krygowski, Kevin Shum, Jennifer Navarro, Aaron Tsai, Barry Krumm, Mark Mayo, Gerhard Doettling, Ute Gaertner, Guenter Gerwig, Michael Kroener, Juergen Haess, Hans-Juergen Muenster, Michael Campbell, Thomas Koehler, Siegmund Schlechter, Thomas Fuchs, Michael Billeci, Anthony Saporito, Anthony Cullen, and Douglas Balazich. Other team leaders in verification, millicode, circuit, and physical design included Dean Bair, Lisa Heller, Robert Averill, Sean Carey, Leon Sigal, and Yuen Chan. Unfortunately, space does not permit us to mention the numerous other individuals who contributed to the development of the z990 processor, but their efforts are certainly appreciated.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., or Linus Torvalds.

## 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. IBM Corporation, *z/Architecture Principles of Operation* (SA22-7832); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.
6. K. E. Plambeck, W. Eckert, R. R. Rogers, and C. F. Webb, "Development and Attributes of z/Architecture," *IBM J. Res. & Dev.* **46,** No. 4/5, 367–379 (July/September 2002).
7. J. S. Liptay, "Design of the IBM Enterprise System/9000 High-End Processor," *IBM J. Res. & Dev.* **36,** No. 4, 713–731 (July 1992).
8. L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries Processor," *IBM J. Res. & Dev.* **48,** No. 3/4, 425–434 (May/July 2004, this issue).
9. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard No. 754-1985*, Institute of Electrical and Electronics Engineers, Inc., New York, August 1985.
10. G. Gerwig, H. Wetter, E. M. Schwarz, J. Haess, C. A. Krygowski, B. M. Fleischer, and M. Kroener, "The IBM eServer z990 Floating-Point Unit," *IBM J. Res. & Dev.* **48,** No. 3/4, 311–322 (May/July 2004, this issue).
11. C. F. Webb, "S/390 Microprocessor Design," *IBM J. Res. & Dev.* **44,** No. 6, 899–907 (November 2000).
12. IBM Corporation, *Enterprise Systems Architecture/390: Data Compression* (SA22-7208); see *http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/*.

*Received September 19, 2003; accepted for publication January 8, 2004; Internet publication April 27, 2004*

**Timothy J. Slegel** *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (slegel@us.ibm.com).* Mr. Slegel received his B.S.E.E. and M.S.E.E. degrees from Lehigh University in 1980 and 1982, respectively, joining IBM in 1982. He has worked in many areas of processor design, including floating-point units, vector processors and cache design. He was the chief architect and overall team leader for the G5 and z990 microprocessors. Mr. Slegel has received an IBM Corporate Award, two IBM Outstanding Innovation Awards, two IBM Outstanding Technical Achievement Awards, and a Ninth-Plateau IBM Invention Achievement Award with 36 U.S. patents. He is a Distinguished Engineer, currently working on the design of future IBM systems.

**Erwin Pfeffer** *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (pfeffer@de.ibm.com).* Mr. Pfeffer studied electrical engineering at the Johannes-Kepler-Polytechnikum in Regensburg, Germany, receiving his graduate degree in 1971 and joining IBM the same year. He worked in areas in printer and inspection tool development before joining microcode development in the S/390 processor group in 1984. In 1993 he moved to the CMOS processor design group and was responsible for the design of the execution unit. Since 1998 he has been working on the TLB2 and Programmable Translator design. Mr. Pfeffer has received an IBM Outstanding Innovation Award, two IBM Outstanding Technical Achievement Awards, two IBM Exceptional Achievement Awards, and a 6th-Plateau IBM Invention Achievement Award.

**Jeffrey A. Magee** *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (jm5@us.ibm.com).* Mr. Magee received his B.S. degree in electrical, computer, and systems engineering from Rensselaer Polytechnic Institute and joined IBM in 1998. He previously worked in cache design and was the lead engineer on the cryptographic coprocessor for the z990 system. He presented this work at the Hot Chips conference in 2002 and has filed a patent application on the concept. Currently Mr. Magee is working on high-performance interfaces for a future IBM microprocessor.