Development and attributes of z/Architecture

by K. E. Plambeck
W. Eckert
R. R. Rogers
C. F. Webb

The IBM z/Architecture™ instruction set architecture (ISA) is an extension of the IBM Enterprise Systems Architecture/390® (ESA/390) ISA and features 64-bit general registers, 64-bit operations, and 64-bit virtual and real addressing. In addition, z/Architecture includes new instructions to optimize the handling of modern multi-byte character encodings and to improve the performance of programs written in high-level languages. It provides compatibility for ESA/390 application programs and increases the ease of development of new application programs. This paper presents an overview of the interesting aspects of z/Architecture and some of the associated decisions and tradeoffs made in its development.

Introduction

In early 1996, it was determined that the ESA/390 architecture had to be extended to provide 64-bit capabilities. At that point, ESA/390 could support large system images with more than two gigabytes of processor storage by augmenting the maximum of two gigabytes of main storage with additional gigabytes of expanded storage. Yet, the need for more main storage and larger address spaces was seen in the not-too-distant future. Some other platforms had already delivered 64-bit-capable systems, and it had been announced that others would be doing so. The extension to ESA/390, which was called z/Architecture*, was announced by IBM in October 2000.

Before presenting an overview of the z/Architecture, we summarize some history of predecessor architectures on which it is based.

Addressing and other history

The z/Architecture [1] continues a succession of architectures for IBM's large computers: the System/360* (1964), System/370* (1970), System/370 Extended Architecture (370-XA, 1983), Enterprise Systems Architecture/370* (ESA/370, 1988), and Enterprise Systems Architecture/390* (ESA/390, 1990) [2] ISAs. The principal evolutionary trait of the processor-related advances in this series has been an increase in both the storage usable by an individual application program and the main storage that can be attached to a model and shared by many programs being executed concurrently. The z/Architecture increases the 31-bit virtual and real (for main storage) address sizes of 370-XA, ESA/370, and ESA/390 to 64 bits, a size large enough to address approximately 1.8×10^{19} bytes (16 exabytes) of either a single virtual address space or the total main storage of the machine. This is 1.1×10^{12} times as much storage as is supported by ESA/390!

In System/360 through ESA/390, addressing proceeded from 24 bits to 31 bits for all addresses, with transitional support for 26-bit addressing of main storage. The dual-address-space facility initiated in System/370 and the access registers initiated in ESA/370 increased addressing horizontally by allowing multiple address spaces to be addressed concurrently. The architectures leading to ESA/370 are discussed in References [3–6], and ESA/370 is discussed in Reference [7].

0018-8646/02/$5.00 © 2002 IBM

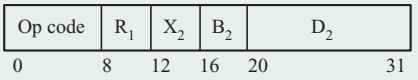

Figure 1

RX instruction format.

System/360

System/360 and its successors through ESA/390 contain 16 32-bit general registers (GRs) used as accumulators and as base and index registers for addressing. System/360 supported 24-bit effective addresses, allowing programs to access 16 megabytes of main storage. Several forms of address specification were provided in the instruction set, the most general of which used a 12-bit displacement in the instruction along with the contents of a base GR and an index GR, with each GR specified by a four-bit field in the instruction. A typical instruction using this "D(X, B)" address calculation uses the RX instruction format, as shown in **Figure 1**. As an example, an ADD instruction in this format forms the operand address as the sum of D_2, the contents of GR B_2, and the contents of GR X_2; reads four bytes from storage at that address; adds this value to the contents of GR R_1; and writes the result into GR R_1.

The principal controlling element of System/360 and its successors is the program status word (PSW), a 64-bit (before z/Architecture) register containing the current instruction address and other control information. The instruction address in the PSW is advanced during sequential instruction execution and is replaced by the branch-target address when a branch is taken. The entire PSW is stored and replaced during an interruption. The PSW formats for System/360 through ESA/390 are shown in **Figure 2**. Among the controls in the PSW is a bit (P) which controls whether the processor is in the *problem state* or the *supervisor state*. Certain system-control instructions, such as those for input/output operations, are classified as privileged and may be executed only in the supervisor state.

System/370 and DAT

The System/360 architecture was extended in System/370 by adding 16 32-bit control registers (CRs) and by supporting virtual storage accessed by means of virtual addresses. The virtual storage was mapped to *real storage* (a synonym for main storage, except for the effects of prefixing, as described below) by means of dynamic address translation (DAT). In System/370 and its successors, an instruction or operand address used by the program may be either real or virtual, with a bit (T) in the PSW controlling this for most addresses.

In System/370, the virtual storage consists of 64KB or 1MB units designated as *segments* and, within each segment, 2KB or 4KB units designated as *pages*, with these variations controlled by bits in CR0. Only the 1MB segments and 4KB pages are discussed here, since these became standard beginning with 370-XA. Address translation is performed on a page basis, with each virtual page being mapped to a *page frame* of real storage.

The DAT process for System/370 is illustrated in **Figure 3**. The address translation is performed by means of a lookup in segment and page tables. The segment-table designation (STD), which is contained in a CR, specifies the address of the segment table. The *segment index* (the four-bit high-order part of the virtual address) selects one entry from this table, which contains the address of the page table. The *page index* (the next eight bits of the virtual address) selects an entry from the page table, which contains the page-frame real address. A 12-bit byte offset (the low-order part of the virtual address) is appended to the page-frame real address to form the resulting real address, which is then used to access storage. If the page-table entry is marked as invalid, indicating that the virtual page is not currently backed in real storage, the control program is signaled by an interruption that it must allocate a real page frame for that page and copy the current contents of that page (e.g., from disk storage) into that real page frame. Other exceptional conditions encountered during DAT also are reported to the control program by interruptions.

Prefixing and extended real addressing

The System/370 architecture was also enhanced to support multiprocessing, allowing multiple processors to share main storage under the control of a single operating system. This required the introduction of *prefixing* to avoid conflicts among the processors in the use of real page frame 0, which is used for interruption processing. With prefixing, each processor has a prefix register specifying a unique 4KB block in main storage, which that processor will use as its real page frame 0, also known as its *prefix area*. Any reference by a processor to a real address in page frame 0 will be directed instead to that processor's prefix area; conversely, any reference by a processor to a real address in which the page-frame address matches the contents of the processor's prefix register will be directed instead to main-storage page frame 0. A real address to which prefixing has been applied is known as an *absolute address*; this is the type of address actually used by the hardware to access main storage.

Later, the System/370 architecture was further enhanced by extending the page-frame real address by two bits. This

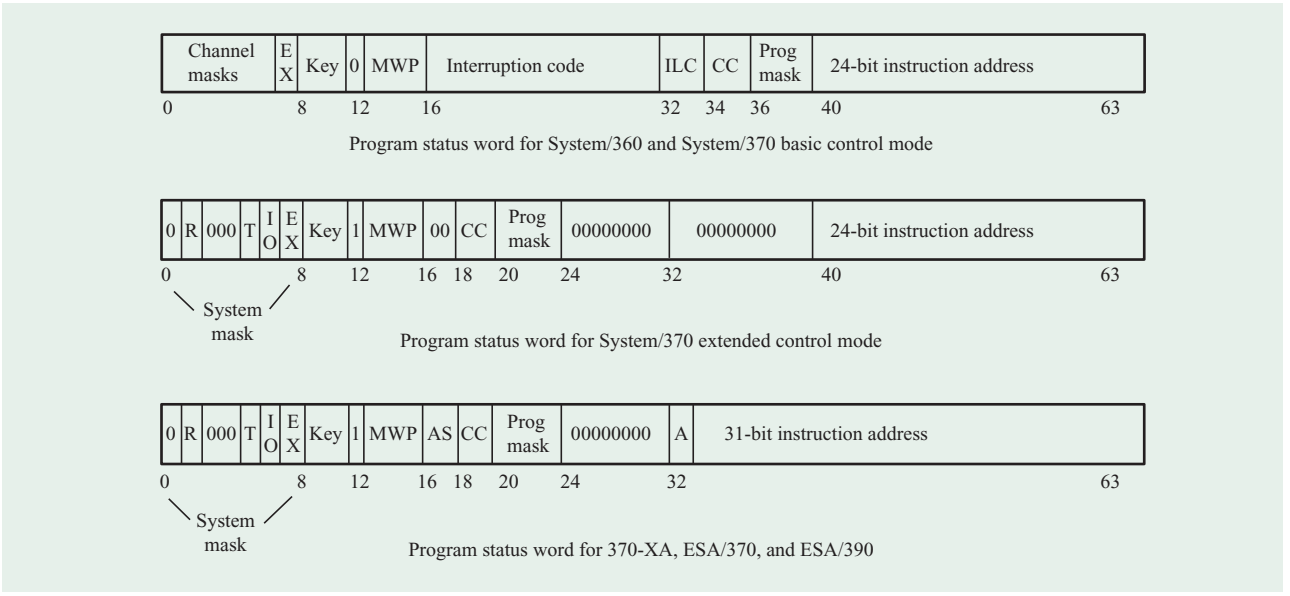

Figure 2

PSW formats for System/360 through ESA/390.

provided a 26-bit real address and allowed attachment of up to 64 MB of main storage.

Dual-address-space facility

The dual-address-space facility was added to the System/370 architecture to allow an application program, in a controlled manner, to operate on two address spaces concurrently and to pass control between one address space and any of many other address spaces. These different address spaces might be separate to achieve protection through isolation (since a virtual address of one space normally does not map to a real address for another space) or because of differing degrees of authority. The dual-address-space facility provided for two active STDs ("primary" and "secondary") in separate CRs (CR1 and CR7) and defined an address-space-control bit in the PSW to select between primary and secondary address-space modes. In these modes, instructions are in the primary address space, and operands are in the primary or secondary address space, as determined by the mode. Special instructions were added to move data between the primary and secondary address spaces, to change the address-space-control bit, and to transfer control from one address space to another. A target address space is identified by an *address-space number* (ASN). The ASN selects an ASN-second-table entry (ASTE) containing an STD that is loaded into CR1 as its new primary STD during the transfer of control. The dual-address-space

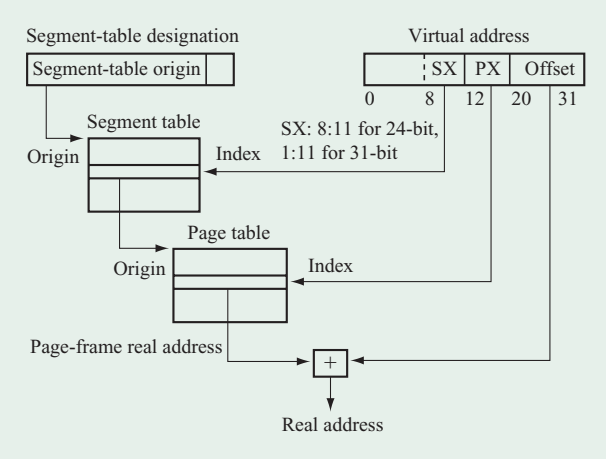

Figure 3

Dynamic address translation for System/370 through ESA/390.

facility and the enhancements to it introduced in ESA/370 are extensively described in Reference [7].

370-XA and 31-bit addressing

The 370-XA architecture introduced 31-bit virtual and real addressing. A bit (A) in the PSW was defined to specify whether effective addresses were 24 or 31 bits in size. To support this, DAT was extended by enlarging the page-

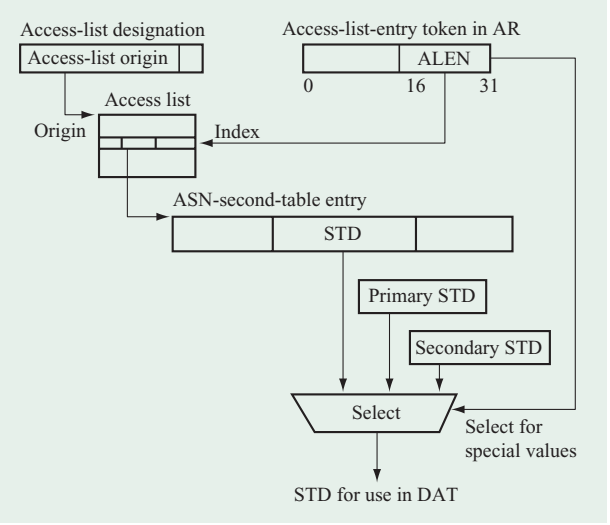

Figure 4

Access-register translation.

table entry to four bytes and the segment index to 11 bits. The instruction address in the PSW was also extended to 31 bits. New instructions were added to allow a program to switch between addressing modes: Branch and save and set mode (BASSM) and branch and set mode (BSM) use bit 0 of the branch-address GR to control the addressing mode after the branch. BASSM and BSM also save the old addressing mode in bit position 0 of a GR along with either the saved return address (BASSM) or a specified address (BSM). The dual-address-space instructions *program call* and *program transfer* were extended to support addressing-mode changes across program boundaries. The ability to change the addressing mode made it possible to mix 24-bit and 31-bit programs, allowing software reuse and accelerating the exploitation of 31-bit addressing. This ability, along with some lesser compatibility issues, was the rationale for defining 370-XA with 31-bit rather than 32-bit addressing. The success of this approach to compatibility and to mixing addressing modes greatly influenced the 64-bit addressing design for z/Architecture.

Expanded storage

The architecture was also extended to include a new form of electronic storage known as *expanded storage*. This was addressable in 4KB blocks through the use of a 32-bit block number by special privileged instructions. The expanded storage was originally intended to bridge the gap in cost and density between main storage and magnetic media; later it provided a means to relieve the performance constraint imposed by the 31-bit real-address size by serving as a high-speed backing store for paging and for large data buffers.

ESA/370

The ESA/370 architecture retained the address size and other attributes of 370-XA, added 16 new 32-bit access registers, and extended the address-space control to two bits to support two additional translation modes: home-space mode and access-register mode. In the home-space mode, instruction and operand addresses are translated using the home STD in CR13. This was originally used by the ESA/370 control program (MVS*) and now is used also for Linux** application programs. In the access-register (AR) mode, instructions are in the primary address space, and operands are in AR-specified address spaces. The instruction field specifying a base GR for address generation for a storage operand also selects the same-numbered access register. The access register is loadable by the application program and contains an access-list-entry token (ALET) that designates, by means of an access-list-entry number (ALEN) an entry in an access list. The access-list entry in turn points to an ASTE which contains an STD that is used by DAT to translate the virtual address for the storage-operand access. Special ALET values are defined as designating the current primary STD and current secondary STD. This architecture allows a program to associate a different address space with each base register used for storage-operand accesses. The mapping of ALETs to STDs, known as access-register translation, is depicted in **Figure 4**. This mapping allows for precise control of access to shared address spaces by various programs as well as simultaneous and efficient access to multiple address spaces by a single program.

ESA/390

ESA/390 initially was little changed from ESA/370. Over time, a number of features were added to ESA/390 which are not germane to this paper; these are described in detail in Reference [2].

Architectures and compatibility

It should be noted that a hallmark of the z/Architecture family is that it has provided, in its succession of architectures, upward compatibility for application programs. That is, if a program could be executed successfully by a machine that implemented an old architecture, it could also be executed successfully on a different machine built to conform to a replacement architecture. Such compatibility is crucial to mainframe customers since it guarantees that upgrading to the next machine will not render their application software obsolete. Preserving this characteristic was a key challenge in the development of the 64-bit z/Architecture.

This compatibility has not necessarily applied to a control program. The architecture of System/370 supported both "basic control" (System/360) and "extended control" (including CRs and DAT) modes of operation. The 370-XA architecture dropped the basic-control mode and changed the DAT-table formats, thus requiring changes to the control program. From 370-XA to ESA/370 to ESA/390, upward compatibility was maintained for control programs as well as for application programs. Although z/Architecture provides compatibility for ESA/390 application programs, it does not do so for control programs because of its changes to the DAT tables and other, lesser changes.

To minimize customer impact, the 370-XA machines up through the early ESA/390 machines were designed such that they could also operate in the System/370 mode. The architectural mode was established by Licensed Internal Code during machine initialization, known as initial machine load (IML).

Creating a 64-bit architecture and more

As might be expected of any 64-bit architecture, z/Architecture includes 64-bit general registers, 64-bit control registers, 64-bit real addressing, 64-bit virtual addressing, and instructions to operate on 64-bit binary integers. The z/Architecture also contains additional new instructions and facilities that facilitate modern e-business computing. The primary concern in extending ESA/390 to create z/Architecture was the need to maintain complete compatibility for application programs and nearly complete compatibility for middleware, tools, and utility programs. Nearly as important was to create an architecture which made it easy for programmers to selectively exploit a subset of the new capabilities without having to totally reprogram in order to use all of the new capabilities.

Access register/general register pairs vs. 64-bit general registers

In the late 1980s, an architecture proposal was made which would allow for the creation of and access to virtual-storage objects larger than two gigabytes. The intention was to accommodate large arrays in FORTRAN. The proposal would allow multiple 2GB spaces to be put together into a seamless address range using the access registers in a novel way. Some of the bits in the ALET (specifically, the ALEN) would, in a new addressing mode, be appended to the left of the 32 bits in the corresponding general register to create a 48-bit virtual address; bits 16–32 of this address, which might include a carry from bit 31 resulting from a D(X, B) addition, would then select an access-list entry specifying one of possibly many concatenated 2GB address spaces. The proposal also provided for 64-bit arithmetic instructions, operating on AR/GR pairs, by using two-byte prefixes placed in front of existing ESA/390 32-bit arithmetic instructions. The proposal had an important advantage: It did not create additional process-state information that had to be saved and restored across context switches and program linkages. This was an important consideration because of the performance cost of saving and restoring additional state information and because program changes are required in order to perform the saving and restoring. Nonetheless, this approach had to be dropped because it was not general enough. Since it used the access register to contain part of the address, it would not have been possible to access locations above two gigabytes while in the access-register mode. This capability would be needed by many operating-system elements that often access data in user address spaces while in the access-register mode. Also, it provided only 48 bits of addressability, while competitors were providing or would provide 64 bits. It was therefore decided that, despite the additional expenses, extending the 32-bit general registers to 64 bits would produce a better overall architecture than the use of AR/GR pairs. Shortly thereafter it was also decided, for reasons of simplicity, that all 16 control registers also should be extended to 64 bits, rather than extending only those which had to contain large addresses.

Modal vs. non-modal instruction-set architecture

One approach for extending an instruction-set architecture to 64-bit functionality is to create a new execution mode in which all addresses are 64 bits and all register operands are 64 bits. This allows reuse of most or all existing instructions, but it has the disadvantage of combining 64-bit addressing and 64-bit operations into a single option. This was deemed to be too disruptive an approach for this platform, given the strong requirement for application-program compatibility and the need for new 64-bit programs to inter-operate with existing 32-bit programs. There are also many programs that may want 64-bit addressing but prefer to continue to work with 32-bit integers and, conversely, many programs that could benefit from 64-bit arithmetic operations but have no need for 64-bit addressing. Therefore, in z/Architecture, the concepts of addressing mode and operand width are separated. The addressing mode is determined by bits in the PSW, while the operand width is determined by the instruction operation code. It is possible for a program to perform 64-bit arithmetic while being executed in the 31-bit addressing mode, and it is also possible to perform 32-bit arithmetic while being executed in the 64-bit addressing mode.

This approach provides the maximum flexibility to programmers (and compilers) in choosing when and how to take advantage of 64-bit capabilities. The cost of this approach is the introduction of about a hundred new

instructions to perform operations analogous to operations already performed on 32-bit operands by existing ESA/390 instructions. The instruction set of z/Architecture contains a new instruction to perform the 64-bit analog of almost every 32-bit arithmetic or logical operation provided by the ESA/390 instruction set. It also provides many additional arithmetic instructions that operate on one 32-bit operand and one 64-bit operand. These mixed-precision instructions are defined to make it easier to manage preexisting 32-bit data in a new 64-bit program.

We use the term *non-modal* to describe a situation in which instruction operation is defined solely by the instruction operation code and not by the addressing mode. The term *modal* is used when the operation of an instruction depends not only on the operation code but also on the current addressing mode. Almost all of the instructions of z/Architecture are non-modal and perform operations which are not influenced by the current addressing mode, other than the effect that the addressing mode has on locating operands in storage. However, z/Architecture does define a number of instructions for which the operation is influenced by the addressing mode. These instructions behave differently in the 24- and 31-bit addressing modes than they do in the 64-bit addressing mode. These instructions all return storage addresses in general registers. In the 24- or 31-bit addressing mode, they return a 24- or 31-bit address in the low-order half of the register and leave the upper half unchanged. In the 64-bit addressing mode, they return a full 64-bit address in the register. New instructions for the 64-bit behavior of these instructions were not introduced because the 64-bit behavior is not useful in the 24- or 31-bit addressing mode, nor is the 24- or 31-bit behavior useful in the 64-bit addressing mode. In a sense, each of these modally defined instructions comprises two instructions that share an operation code, each providing the appropriate behavior for the addressing mode in which it is executed. Many of the modally defined instructions also have length values in general registers. In the 24- and 31-bit addressing modes, these are 24- or 31-bit lengths in the low-order half of the general register. In the 64-bit addressing mode, they are 64-bit lengths.

To realize the full benefit of the non-modal approach to the instruction set, the operation of the existing (ESA/390) instructions in the z/Architecture mode was carefully defined such that an ESA/390 program being executed in the z/Architecture mode could not accidentally change any processor state which did not exist in ESA/390. Thus, not only do the ESA/390 32-bit instructions use 32-bit operands and operations, but also the results of these instructions modify only the low-order 32 bits of GRs, leaving the high-order 32 bits unchanged. Thus, a program being executed in the 64-bit addressing mode can call an old program to be executed in the 24- or 31-bit addressing mode without concern that the left halves of the GRs will be changed and then not restored (which assumes, of course, that the old program does not use the new instructions having 64-bit operands). Leaving the left halves of the GRs unchanged does impose some additional complexity on the execution hardware, particularly in the register-file design.

To facilitate branching between programs executed in different addressing modes including the 64-bit mode, z/Architecture extends the branch and save and set mode (BASSM) and branch and set mode (BSM) instructions so that a 1 in bit position 63 of the branch-address GR causes setting of the 64-bit addressing mode; when bit 63 is 0, bit 32 of the GR causes setting of the 24- or 31-bit addressing mode as in ESA/390. When bit 63 is 1, the branch address is generated as if the bit were 0 (because instructions are necessarily on halfword boundaries), but the bit is left 1 to indicate that the addressing-mode change occurred and, therefore, the addressing mode must be restored during the return linkage by using BSM instead of an unconditional *branch on condition* instruction. Similarly, BASSM and BSM set bit 63 of the return-address register to 1 when executed in the 64-bit addressing mode.

Instruction prefixes vs. split operation codes

The decision to add so many new instructions to z/Architecture created a problem. Since the roots of z/Architecture are more than 35 years old, it is difficult to find operation codes for so many new instructions. The idea to use instruction prefixes, as had been done in the 1980s proposal, was tempting. However, the problems that instruction prefixes present to processor design and software debugging required a different solution. A number of the few remaining one-byte operation codes would be used, along with another byte that could take values from 0 to 255, to create a large number of two-byte operation codes. The base architecture has had two-byte operation codes since the introduction of the System/370 architecture in the early 1970s. However, the approach taken then of using the first two bytes of the instruction for the operation code creates some difficulties. First, the base architecture defines an *execute* instruction that allows a target instruction to be executed with a temporary modification of the second byte of the instruction. If the second byte of an instruction is defined as part of the operation code, it cannot profitably be used as the target of the *execute* instruction, because any temporary modification of that byte would cause the execution of a completely different instruction. Also, it is advantageous to processor design if certain specifications within the instruction appear at the same relative location for instructions with two-byte operation codes as for instructions with one-byte operation codes. Therefore,

a novel approach was taken. The second byte of the operation code was placed not as the second byte but as the last byte. New six-byte forms of existing four-byte formats were created to accommodate the extended opcode without any loss of functionality. As an example, the RX format (Figure 1) was extended to the RXE format (**Figure 5**), retaining the position within the instruction of the R_1, X_2, B_2, and D_2 fields. These formats were first introduced in the binary-floating-point facility when this was added to ESA/390.

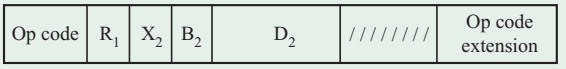

Figure 5

RXE instruction format.

Hashing vs. dynamic address translation

Since z/Architecture supports 64-bit virtual addressing, it must provide for the translation of 64-bit virtual addresses to real-storage addresses. Historically, since System/370, the architecture has employed an indexed-table scheme for accomplishing address translation, as described previously above. In extending from 31 bits to 64 bits, 33 additional addressing bits are introduced. An obvious way of dealing with these additional addressing bits would be to decompose them into three additional 11-bit indexes to three additional levels of translation tables, which might be called region tables. However, there was great concern that three additional levels of translation table—and three additional main storage references—during address translation would lead to significant processor overhead and would waste considerable main storage for predominantly empty region tables.

Some other architectures employ what is called "inverted page tables" rather than an indexed-table scheme. With inverted page tables, the virtual address is hashed, and the hash value is used to index into a table that contains a list of all the page-table entries designated by that particular value. The list is then searched to find a page-table entry that translates the specific virtual address being processed. In this sort of scheme, increasing the width of the virtual address to 64 bits does not necessarily lead to significant additional main-storage accesses to perform a translation. Switching completely to such a scheme would have driven an unacceptable level of change in the operating systems for z/Architecture, but a mixed approach appeared practical to pursue. The first 33 bits of the address would be hashed to search for a segment-table origin, and then the remainder of the translation would be done, as in ESA/390, by indexing to a segment-table entry and then a page-table entry. This idea seemed very promising, but as it was refined over time several problems arose. The largest problem was the possibility that a software error could put a loop in the synonym chain of the hash table and cause an unending internal-code loop. To avoid this possibility, a model-dependent limit was defined for the length of these chains. However, this led to the possibility that a page of virtual storage could be resident in real storage but, based on the pattern of valid virtual locations, could not be made valid because of the limitation on the number of synonyms allowed for a hash value. Another problem was that if the synonym chain exceeds a certain length, this scheme, which was to decrease main-storage accesses, might actually lead to more accesses than the indexed-table approach. These and other problems with the hashing technique made the approach unacceptable, particularly for a mainframe system that must be robustly reliable (both hardware and software) even under extraordinary conditions.

Out of this effort, an alternate idea arose, illustrated in **Figure 6**. Although the architecture would allow for a huge (16-exabyte) address space, it was known that most address spaces would be no larger than two gigabytes and that almost none would be larger than four terabytes for many years. The idea was to allow the translation process to bypass the initial translation tables if the address space was not large enough to require them. For example, for a 2GB address space, translation would start directly at the segment table, bypassing all three levels of region tables; for a 4TB address space, translation would start at the third region table, bypassing the first two levels of tables; and for larger address spaces, translation would start at the second or first region table as necessary. Two bits are used in the address-space-control element, the anchor for address translation for an address space in z/Architecture, to tell the translation hardware the type of table with which the translation process is to start (and, thus, the maximum allowed size of the virtual address). By using this approach, the additional main-storage accesses to translate the extra 33 bits in a 64-bit address are almost totally avoided. Furthermore, extending the indexed-table structure for address translation is a more natural extension of the architecture, and much less costly and complex to support in operating-system software. This approach also proved simpler to implement in hardware, since the additional translation steps are essentially identical to the first step (segment translation) of the ESA/390 DAT. This simplicity of design was further aided by making the formats of the region-table entries and segment-table entries nearly identical.

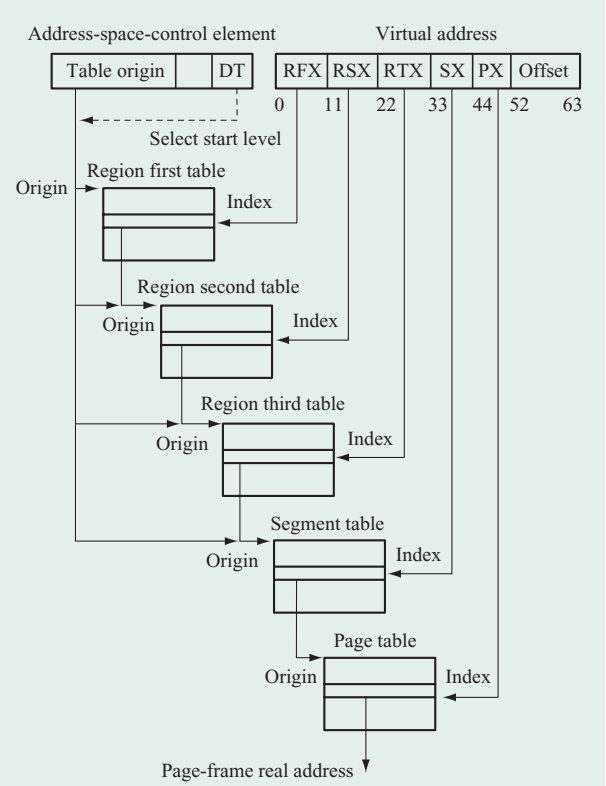

Figure 6

z/Architecture dynamic address translation.

The real space

For some time, the operating-system community has wanted a simple and safe way to access real storage directly. In ESA/390, a program must be executed with DAT turned off in order to access real storage (other than by using the instructions *LURA* and *STURA*). When address translation is turned off, the address-space isolation it provides is negated, so that an erroneous store can damage any application being executed, in any address space, in an unpredictable way and can lead to problems that are extremely difficult to debug. Execution with address translation turned off is complex, inconvenient, and dangerous. Furthermore, turning address translation on and off is expensive in terms of performance.

The first approach used to mitigate this situation was to designate specific access-list-entry-token (ALET) values which, with execution in the access-register mode with address translation turned on, could be used to address real storage directly. The problem with this is that a mechanism would be needed to prevent unauthorized programs from directly accessing real storage. Since the ALET values loaded into access registers are completely under the control of the application program, the only way to inhibit unauthorized access would be to limit the use of the special ALETs to supervisor-state execution. The base architecture has many means for limiting storage access, such as the storage key and, in CRs, the authorization index and extended authorization index, but it has no concept of requiring supervisor state for data access.

This first approach was refined by using specific address-space-control-element (ASCE) values to designate access to real storage, rather than using the ALET. Unlike the ALET, the ASCE is totally controlled by the operating system. The distinctive ASCE values either could be loaded into control registers (CR1, CR7, or CR13) or could be placed in an ASN-second-table entry (ASTE) that is attached to an access list for use in the access-register mode. By creating an ASTE containing a real-space ASCE and designating it from an entry in the access list of every address space, with the entry protected by a unique extended authorization index (EAX), convenient access to real storage is made available to programs being executed in any address space, but it is made available only to programs which have the authority to set the appropriate EAX. These techniques are used by both the z/OS* and z/VM* control programs to access real storage without the problems associated with turning off address translation.

An ASCE providing access to real storage as described above is called a real-space designation (RSD). An ASCE is like the segment-table designation (STD) of ESA/390, except that an ASCE may be an STD, a region-table designation (RTD), or an RSD.

Supporting two architectures

With each extension of the architecture, it has been required that the first hardware systems supporting the new architecture also fully support the older architecture, such that operating systems written for the older architecture could run on the new hardware without change. This is needed to minimize customer disruption when upgrading hardware and to allow for a phased adoption of software that exploits the new architecture features. For most of these architecture extensions, the changes have been upward-compatible, and it has been sufficient to indicate to the software which architecture is in effect. For the introduction of z/Architecture, however, incompatible changes (such as to the DAT-table formats) required a different approach. In the transition of S/370 to 370-XA, which entailed similar control-program incompatibilities, a machine was initialized to one of the two architecture modes, in some cases using different Licensed-Internal-Code loads. This approach was rejected for z/Architecture because of the disruptive nature of reinitializing the machine and because of increased development and testing costs. On the other hand, a

method in which the architectural mode is selected by a control-register or PSW bit was considered too vulnerable to a software error.

It was decided that the system would always be initialized in the ESA/390 architectural mode and the operating system could then switch into the z/Architecture mode under program control. In addition to avoiding the pitfalls of the previous solutions, this method would also make it easier to write an operating system that could run under either architecture (which was done with z/OS and z/VM). However, the method also posed some problems, such as having different processors in a multiprocessor configuration executing different architectures. While no need for such a hybrid configuration was ever identified, allowing for it would have required a complete definition and rigorous testing of the behavior in such a situation. To avoid this complexity, it was determined that the architectural mode had to be an attribute of the multiprocessor configuration rather than an attribute of each processor (another reason for not using a control-register bit). This was implemented by defining a special order of the *signal processor* instruction to change the architectural mode for all processors in the configuration. This order is unique in that it can only succeed if all processors in the configuration, other than the processor executing the instruction, are in a stopped or check-stop state. This rule makes it impossible to "pull the rug out from under" the program being executed by another processor by changing the architectural mode while that execution is ongoing.

In order to fully support the functions of the z/VM control program and the logical partitioning provided by the Processor Resource/Systems Manager* (PR/SM*) facility, the architecture for the interpretive-execution facility (the SIE instruction) was extended to allow virtualization of the architectural mode. Thus, any given logical partition under PR/SM on a machine supporting z/Architecture may be in either the ESA/390 or the z/Architecture mode, regardless of the architectural mode used by other partitions, though all of the processors within a multiprocessor partition must be in the same architectural mode, as described above. Similarly, z/VM allows for a mixture of ESA/390 and z/Architecture "guest" operating systems.

Staged architecture vs. a single and complete architecture

In order to provide flexibility for the initial software support for z/Architecture, and to a lesser extent for the initial hardware implementation, the initial definition of z/Architecture permitted separate enablement, by means of control-register bits, of the following features of the architecture:

- The basic z/Architecture, consisting of larger registers, larger segment- and page-table entries, and a larger prefix area. The instructions performing 64-bit operations could be executed, with this feature, only in the supervisor state, since the operating system would not necessarily save and restore the larger registers. This feature was to allow the operating system to use real storage above 2GB on behalf of itself and application programs.
- Allowance of execution of the new instructions in the problem state.
- The use of address-space-control elements designating region tables, thus providing virtual storage above 2 GB.
- Large instruction addresses, that is, a larger PSW containing a 64-bit instruction address.

It was intended that the operating system would enable the above four features as it became ready to support them. Only the last feature, the large instruction address, was intended as one that need not be in the initial hardware implementation.

As plans moved from the drawing board to implementation, however, several liabilities of this approach became apparent. First, the plurality of features greatly increased the amount and complexity of checking required in the hardware, and this in turn affected the circuit delay on critical paths in the control logic, potentially limiting the operating frequency of the processor. Each of the features required thorough testing in both simulation models and actual hardware, significantly increasing overall development costs and schedules. Furthermore, once these features were described as such in the externally published architecture, hardware support for their separate enablements would be required in future machine models even when the software had progressed to using all of them.

Another argument against this initial approach grew out of the hardware implementation plan that was finally chosen. One goal of this plan was that programs using the 64-bit functionality (both operations and addressing) should perform as well as if they were using only the ESA/390 31-bit functionality; this was necessary to avoid giving the 64-bit architecture a "black eye" because of weak performance of the initial product. Another goal was that the initial hardware should facilitate development of software which would exploit the full capabilities of z/Architecture, a target that is most easily hit by providing "native" hardware support for all architectural features.

These considerations led to a decision to combine the first three of the above features into a single architectural mode. Inclusion of 64-bit instruction addressing was less clear-cut, for two reasons:

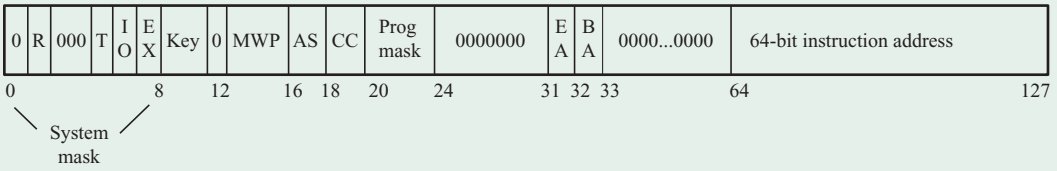

Figure 7

PSW format for z/Architecture.

1. The practical need for 64-bit instruction addressing in real applications is far less than for 64-bit operand addressing. Indeed, given the robust address-space architecture from ESA/390 and the availability of 64-bit addressing for data, a strong requirement for 64-bit instruction addressing is not likely for many years, if ever.
2. Expansion of the instruction address (IA) requires expansion of the program status word (PSW), which includes the IA, from a doubleword to a quadword (128 bits), and that in turn ripples into many parts of the operating system and even some middleware and applications.

On the other hand, having different architected sizes for operand and instruction addresses would create an architectural asymmetry new to the architectures commenced by System/360. An example of the difficulties arising from this is the case of the *execute* (EX) instruction, which specifies as its *operand* a single *instruction* to be executed with some modification by another operand of the EX. From a software viewpoint, the target instruction is a storage operand and should be subject to operand-addressing rules; in the hardware, the target must be processed as an instruction and is most naturally fetched from storage using instruction-fetching hardware, which operates according to instruction-addressing rules. While it would have been possible to define the rules and build the hardware to cover all such "corner" cases, the result would have been an unduly complex architecture and implementation. These considerations led to the development of a means to isolate the impact of the PSW format change to a small portion of the operating system, which opened the way for adoption of a single "64-bit everything" definition for z/Architecture.

By incorporating all of the 64-bit capabilities into a single architectural step, z/Architecture forms a stable base for software exploitation of these facilities in the years ahead and avoids a need to change the formal architecture or the hardware implementation. With the emergence of Linux as a viable operating system on the IBM zSeries* platform, this is all the more valuable since it allows different operating systems to follow different courses and schedules without having to adjust, or adjust to, the hardware implementation plan. Supporting all 64-bit features in the first implementation did require some additional dataflow hardware, but the support brought with it a simplification of some control logic and a robustness of performance which more than compensated for its cost.

One or two PSW formats

As mentioned above, changing the PSW format to accommodate a 64-bit instruction address would be very expensive in terms of software changes required to deal with such an incompatibility. On the other hand, there is very little need at this point to allow program execution in virtual storage above 2 GB. Once an application's data items can reside anywhere in a 16-exabyte address space, 2GB storage provides plenty of room for the programs themselves. However, introducing a "non-Von Neumann" architecture, with asymmetry between operand and instruction addressing, would have raised a number of subtle architecture issues and would have been a complication for all future z/Architecture implementations.

The compromise solution was remarkably simple. The architecture would allow program execution with a full 64-bit instruction address, and the PSW format would be incompatibly changed to accommodate the longer instruction address, as shown in **Figure 7**. However, this incompatible change would be absorbed and hidden by the operating system, at least in the initial implementations. Whenever a copy of a PSW is placed in storage, the operating system collapses the full z/Architecture PSW to the size and format of an ESA/390 PSW. Other software with PSW-format dependencies can be totally unaware of the actual z/Architecture PSW. The number of places in an operating system where this PSW-format manipulation would need to be done is small and does not present a performance problem. Since the PSW-format change loses the high-order 33 bits of the 64-bit instruction address,

an operating system that does this transformation for compatibility cannot, of course, support program execution above 2 GB even though the architecture allows it. This approach permits the z/Architecture to be defined as a full ("Von Neumann") 64-bit architecture while delaying the impact of the incompatible PSW format. When there is a true requirement for program execution above 2 GB, it will just be a matter of making all of the software changes and will not require an architecture revision or hardware change.

Since there are now, in a sense, two PSW formats—the true z/Architecture format and the collapsed ESA/390 format that software will generally be using—some accommodations were made in the z/Architecture definition. First, PSW bit 12, which has had to be 1 since 370-XA, is defined to be 0 in a z/Architecture PSW to avoid confusion. Second, a new *load PSW extended* (LPSWE) instruction is defined for loading a true z/Architecture PSW. The ESA/390 *load PSW* (LPSW) instruction is still supported and can be used to load an ESA/390-format PSW. When LPSW is executed, the processor expands the ESA/390-format PSW to the z/Architecture format, including inverting bit 12. This is the reverse of the collapsing that the operating system performs to create the ESA/390-format PSWs.

Doubling the prefix area

As was described in the historical background, each processor has a unique 4KB area at real address 0. This *prefix area* contains various hardware/software communication areas used during interruption processing and other activities. Thus, when different processors access real locations 0–4095, they are actually accessing different storage locations. Since the sizes of the z/Architecture PSW, general registers, and control registers are each doubled, more space is required for these communication areas in z/Architecture. The ESA/390 prefix area, given that a large part of it is used by software, does not contain enough free space for all the extra status that must be communicated in z/Architecture. Therefore, the size of the prefix area in z/Architecture was increased from 4 KB to 8 KB. Prior to z/Architecture, the entire prefix area was addressable without a base register. This was a convenience and sometimes a necessity for software, so the new mapping was laid out to keep the most frequently accessed fields in the first 4 KB by placing in the second 4 KB only fields which are seldom accessed.

Since the processor is always initialized in the ESA/390 mode and may switch to the z/Architecture mode, care had to be taken in defining what happens to the prefix register when the architectural mode is switched. Processor performance is very sensitive to the efficiency with which prefixing and reverse prefixing are done. To simplify matters, it is required that the prefix area address in the z/Architecture mode be a multiple of 8092. Therefore, the rightmost bit of the ESA/390 prefix, which specifies a location on a 4KB boundary, is set to zero when the prefix becomes a z/Architecture prefix due to a change of the architectural mode.

Onward to e-business

In addition to the main goal of defining z/Architecture as a 64-bit architecture, it was desirable to include some new capabilities to enable and facilitate modern business. These include globalization, cryptography, and interoperability.

Dealing with multiple character sets

For the most part, the processor architecture is ignorant of any character encodings such as ASCII or EBCDIC, yet applications must use different encodings (code-pages) for different languages. Because most character representations use one byte per character, the limitation to 256 characters caused many codepoints to be used multiple times in different languages—even within the different versions of the Latin alphabet for special characters such as monetary characters (e.g., $) and umlauts (e.g., ä, ö, ü). For more complicated characters such as Chinese, a two-byte encoding is needed.

To address these problems, an industry group, the UNICODE foundation, was created. The UNICODE foundation defined unique representations of all known characters, most of which are two bytes in length. In order to ease the conversion between different forms of UNICODE, the extended-translation facility 1 was introduced in ESA/390. The extended-translation facility 2, which is part of z/Architecture but has also been added back into ESA/390, enables conversion between different one- and two-byte character representations. Facility 2 provides table-driven conversions between one- and two-byte character representations, four conversions altogether: one-to-one, one-to-two, two-to-one, and two-to-two.

Besides these table-driven conversions, z/Architecture provides conversions of UNICODE decimal numbers, which are two bytes, or ASCII decimal numbers, which are bytes, into the computational packed-decimal format. It also provides an instruction to test the validity of packed-decimal numbers without causing a program interruption.

Toward more efficient cryptography

While for most commercial applications, 32-bit, and now 64-bit, signed arithmetic is sufficient and mathematically complete, newer applications such as cryptography require unsigned arithmetic and functions which allow easy calculation on arbitrary-length integers. For example, some cryptographic functions require 1024-bit or even longer arithmetic, and others require bit-wise rotation of

values. In order to satisfy these requirements, instructions for 32-bit and 64-bit add-with-carry, subtract-with-borrow, and rotate operations were included in z/Architecture.

Trading data with other platforms

Another requirement is to be able to convert between the native "big endian" integer format of z/Architecture and the reversed "little endian" format of many distributed platforms. Instructions are provided for byte-wise reversal of two-, four-, and eight-byte values as they are loaded into a general register or stored from a general register into storage.

More modern instructions

The definition of the z/Architecture also includes a number of new instructions that are not needed to support any particular initiative but simply allow for more efficient program execution. Most of these new instructions have immediate operands so that they can be used to reduce storage references and make more efficient use of processor cache. Here they are simply listed with a brief description:

- *Load immediate* loads a 16-bit immediate field into any one of the four halfwords of a 64-bit general register and clears the other three halfwords.
- *Insert immediate* is like *load immediate* but does not clear the remainder of the register.
- *And immediate* performs a logical AND operation between a 16-bit immediate field and any one of the four halfwords of a 64-bit general register and places the result in that halfword.
- *Or immediate* is like *and immediate* but performs a logical OR operation.
- *Test under mask immediate* performs a test under mask of any one of the four halfwords of a 64-bit general register using a 16-bit immediate mask. The instructions which perform this operation for the two low-order halfwords of a register already existed in ESA/390. Two new instructions were added to z/Architecture for the two high-order halfwords.
- *Branch relative on condition long* performs a relative branch similar to the ESA/390 instruction *branch relative on condition* except that the relative offset for the new instruction is 32 bits long and allows a branch to any halfword location within four gigabytes of the branch instruction.
- *Branch relative and save long* is like the ESA/390 instruction *branch relative and save*, except that the relative offset is 32 bits long.
- *Load address relative long* loads the address of any halfword location within four gigabytes of the instruction.

Out with the old

In the more than 35 years since the introduction of the IBM System/360, the architecture has grown in complexity, with hundreds of small and several large extensions. It was determined that some of these extensions would be of little or no value in z/Architecture, and they were dropped (i.e., not brought forward from ESA/390) in z/Architecture when this could be done in a way consistent with IBM's extreme commitment to compatibility in this architecture family. Some facilities having to do with expanded storage are not needed in z/Architecture because they were only used in z/OS, and z/OS does not support expanded storage when using the z/Architecture mode, since the 31-bit real-storage addressing constraint has been completely relieved by 64-bit real addressing. These facilities are asynchronous paging, the asynchronous data mover, and the ability of a page of virtual storage to be valid in expanded storage when referred to by the *move page* instruction. Control register 0 bit 15 (now bit 37), which was used to distinguish new control structures of ESA/370 from earlier dual-address-space structures, is now not defined because only the later structures are supported in z/Architecture. Finally, the vector facility was not carried forward into z/Architecture because it had not been built as part of ESA/390 machines for many years due to technology changes.

Future architecture steps

As described in this overview, z/Architecture extends the ESA/390 instruction-set architecture to obtain full support for 64-bit operations and addressing (real and virtual, operand and instruction) and enriches the instruction set to better support applications written in modern high-level languages. Even if the addressing needs of server applications continue to expand rapidly, these extensions should provide adequate growing space for decades to come, and operating-system support for exploiting them can be expected to track application needs. Future steps in the architecture, therefore, will come in new directions. As new languages, programming techniques, and applications arise, they will bring with them new opportunities for optimization of the architecture and hardware. The z/Architecture instruction formats, including several new ranges of extended opcodes, and the expanded control-register and PSW formats of z/Architecture, provide ample space for adding new features in a fully compatible and software-controllable manner.

Acknowledgments

The development of z/Architecture benefited from the contributions of many individuals throughout IBM. Particularly significant contributions came from M. S. Farrell, J. H. Mulder, D. L. Osisek, I. G. Redding, C. A. Scalzi, and T. W. Springer.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Linus Torvalds.

References

1. IBM Corporation, *z/Architecture Principles of Operation*, Order No. SA22-7832; available through IBM branch offices.
2. IBM Corporation, *Enterprise Systems Architecture/390 Principles of Operation*, Order No. SA22-7201; available through IBM branch offices.
3. R. P. Case and A. Padegs, "Architecture of the IBM System/360," *Commun. ACM* **21,** No. 1, 73–96 (January 1978).
4. A. Padegs, "System/360 and Beyond," *IBM J. Res. & Dev.* **25,** No. 5, 377–390 (September 1981).
5. A. Padegs, "System/370 Extended Architecture: Design Considerations," *IBM J. Res. & Dev.* **27,** No. 3, 198–205 (May 1983).
6. D. Gifford and A. Spector, "Case Study: IBM System/360–370 Architecture," *Commun. ACM* **30,** No. 4, 292–307 (April 1987).
7. K. E. Plambeck, "Concepts of Enterprise Systems Architecture/370," *IBM Syst. J.* **28,** No. 1, 39–61 (1989).

Received December 26, 2001; accepted for publication February 14, 2002

Kenneth E. Plambeck *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (plambeck@us.ibm.com).*

Mr. Plambeck is a Senior Programmer in the eServer Architecture Department. He received a B.E.E. degree from the Georgia Institute of Technology in 1956 and an M.S. degree in electrical engineering from the University of Illinois in 1958. After he joined IBM that same year, his initial work was on diagnostic programming for the IBM 709, 7090, and 7950 (an extension of the 7030 Stretch computer), the indexing and output phases of a FORTRAN compiler for the 7030, and various projects of the operating systems for System/360 and System/370. Mr. Plambeck began his systems architecture career in 1972, working first on the FS architecture, then on VSE, and, beginning in 1979, on extensions to System/370. He has been a principal developer and author of the CPU material in the *Principles of Operation* manuals for ESA/370, ESA/390, and z/Architecture and has also defined the separately documented compression facility begun in ESA/390. Mr. Plambeck received an IBM Outstanding Technical Achievement Award for ESA/370, an IBM Outstanding Innovation Award for each of compression and z/Architecture, and two Corporate Awards; he holds 15 issued patents.

Wolfgang Eckert *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (weckert@de.ibm.com).*

Mr. Eckert is an IBM Distinguished Engineer, working on eServer system design and hardware/software interfaces and architecture. He received an M.S. (German Diplom) degree in physics from the University of Hamburg, joining IBM in 1968. Mr. Eckert worked in product test and assurance until 1986, primarily developing hardware test systems. Since then he has worked on hardware system design, developing new System/390 and z/Architecture features, including translation facilities, 64-bit extensions, Hipersockets, and FCP.

Robert R. Rogers *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (rrrogers@us.ibm.com).*

Mr. Rogers is a Senior Technical Staff Member, working on zSeries software system design. He joined IBM in 1969 in Poughkeepsie as a computer operator. He received a B.A. in mathematics from Marist College in 1971 and subsequently became a computer programmer at the Poughkeepsie Programming Center, where he worked on the OS/360 operating system. Mr. Rogers has received an IBM Outstanding Technical Achievement Award and an IBM Outstanding Innovation Award for his design work; he was a recipient of the 2000 IBM Chairman's Award. His most recent assignment was as software architect and system designer for zSeries 64-bit software.

Charles F. Webb *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (cfw@us.ibm.com).*

Mr. Webb is an IBM Distinguished Engineer, working on server development. He received a B.S. degree in 1982 and an M.Eng. degree in 1983, both from Rensselaer Polytechnic Institute. He joined IBM in 1983 at the Product Development Laboratory in Poughkeepsie, where he has since remained. Mr. Webb has worked on the ES/9000 processor, the S/390 G4 and G5 CMOS processors, and the eServer z900 processor in the areas of performance analysis, architecture, and design. He has received thirteen IBM Invention Achievement Awards, four IBM Outstanding Innovation Awards, and an IBM Corporate Award. He is a member of the IBM Academy of Technology and the IEEE.