The microarchitecture of the IBM eServer z900 processor

by E. M. Schwarz
M. A. Check
C.-L. K. Shum
T. Koehler
S. B. Swaney
J. D. MacDougall
C. A. Krygowski

The recent IBM ESA/390 CMOS line of processors, from 1997 to 1999, consisted of the G4, G5, and G6 processors. The architecture they implemented lacked 64-bit addressability and had only a limited set of 64-bit arithmetic instructions. The processors also lacked data and instruction bandwidth, since they utilized a unified cache. The branch performance was good, but there were delays due to conflicts in searching and writing the branch target buffer. Also, the hardware data compression and decimal arithmetic performance, though good, was in demand by database and COBOL programmers. Most of the performance concerns regarding prior processors were due to area constraints. Recent technology advances have increased the circuit density by 50 percent over that of the G6 processor. This has allowed the design of several performance-critical areas to be revisited. The end result of these efforts is the IBM eServer z900 processor, which is the first high-end processor based on the new 64-bit z/Architecture™.

1. Introduction

The microarchitecture of the IBM eServer z900 processor is based on that used in the G4 processor, introduced in 1997 [1]. The G4 processor supported the ESA/390 architecture and was a single-issue 32-bit processor with a unified cache and a short six-stage pipeline. This design was expanded in 1998 to create the G5 processor. The G5 processor was the first IBM 390 mainframe to implement the IEEE 754 binary floating-point standard [2–5] as well as the traditional hexadecimal floating-point architecture. The G5 processor doubled the performance of the G4 by implementing more instructions in hardware versus low-level software and improving branch prediction with a branch target buffer (BTB). The 1999 G6 processor was primarily a remap of the G5, but with the expansion of some facilities to lead to a multiprocessor system of twelve central processors, versus ten central processors on G5, with two extra processors for I/O processing, coupling, or simply as spares [6]. This type of configuration is called a 12 + 2-way system. The z900, introduced in December 2000 [7], continued the enhancements of the G4 through G6 design, and is available in a 16 + 4-way system.

The G6 processor had some performance bottlenecks which were recognized as key targets for future improvements. The fetching of data operands could conflict with instruction fetches. The level 1 (L1) cache was interleaved, which allowed multiple requests at the same time, but conflicts were still possible because it was unified, containing both data and instructions. The BTB had some delays due to conflicts between searches and history field updates. Branch performance was greatly

0018-8646/02/$5.00 © 2002 IBM

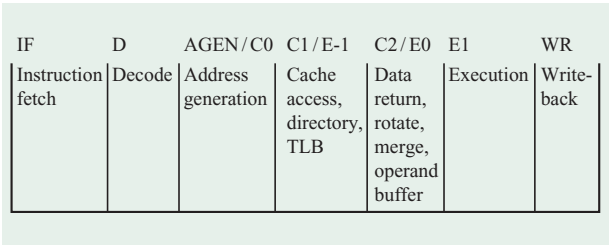


Figure 1

Instruction pipeline of the z900 processor.

aided by the use of a BTB, but its 2K size was a concern. Also, database programmers had concerns about the data compression performance. On the G3 processor, data compression was implemented in a hardware compression engine [8]. In the G4, G5, and G6 processors, only a set of compression hardware assist instructions were implemented in the fixed-point unit (FXU), and much of the functionality came from low-level software called millicode. The cycle time of the G4 was half that of the G3 in 1997, and thus the relative compression performance was acceptable. However, within three years there was a concern about hardware compression performance. Also, our COBOL customers are continually asking for increased decimal performance, so we wanted to make some substantial changes in this area. With the latest circuit and packaging advances, circuit densities are far greater than in previous generations. Thus, there was plenty of room to enhance the functionality and performance of the prior-generation (G6) processor. It was possible in 2000 to consider large-area-consuming enhancements such as doubling the L1 cache, quadrupling the BTB, and adding a hardware compression engine.

The z900 processor is the first processor to implement the 64-bit z/Architecture*. It also retains support for all older 390 to 360 architectures, thereby providing binary compatibility for programs written in the 1960s. Thirty-four new instructions were added to the ESA/390 architecture mode and 139 new instructions for the z/Architecture mode, for a total of 173 new opcodes. The new ESA/390 instructions are 32-bit-long immediate relative branches; nondestructive rotates, which use a different source and target register; conversion instructions for Unicode to/from UTF-8 format and *store clock extended*; and *translate extended*, as well as *set clock field* and *store system information* instructions. The new z/Architecture instructions include many new arithmetic instructions which operate on 64-bit binary integers [9, 10]. Of the additions to the general instruction set, the most notable are some new 16-bit immediate operations, and nondestructive shift operations. Architected facilities have also changed. The general-purpose register file (GPR) has increased from 32 bits to 64 bits; 32-bit instructions operate on only the low-order data in bits 32 to 63. The control registers have also been widened, but there was no need to widen the access registers or floating-point registers. The processor status word (PSW) was also widened from 64 bits to 128 bits, so that it could easily hold the 64-bit instruction address (IA). The z900 processor supports 24-, 31-, and 64-bit addressing and 32- and 64-bit arithmetic.

The underlying hardware supports all facilities for both architectures and all addressing mode possibilities. The architecture level supported by hardware execution cannot be changed readily and requires a *signal processor* (SIGP) or *start interpretive execution* (SIE) privileged instruction. The architecture level is important to the operating system for determining translation table sizes and other underlying information, but it is not very important to user programs. The newer the architecture level, the greater the numbers of instructions that are enabled. The addressing mode, on the other hand, is very important to user programs and is controlled by bits 31 and 32 of the PSW. The addressing mode can be rather dynamic. Other architectures have allowed transitions to 64-bit architectures which have provided only minimal support for old user programs. Other platforms which have made the transition to 64-bit architectures have forced the user to recompile, have allowed one addressing mode to be active at a given run time, or have forced old binaries into slow software emulation. The z900 platform allows old subroutines to run with new programs, each using different addressing modes, and they run efficiently together. Several instructions such as *branch and save* or *set address mode* can change the addressing mode. In z/Architecture mode, all three addressing modes are possible: 24-, 31-, or 64-bit addressing.

Since the architecture is relatively complex, it is implemented by a combination of hardware and low-level software called millicode. Millicode on the z900, as well as for G4, G5, and G6, is vertical code consisting of a combination of 390 assembler and additional assist instructions. The millicode operates on working registers called millicode general-purpose registers (MGRs) rather than on the architected GPRs. For the z900 there is a new interface between hardware and millicode which enhances performance and simplifies code. Working registers are bigger, for a total of 16 by 64-bit MGRs. Register interlocks for millicode control registers are handled primarily in hardware. System operation commands such as *move page* or *store with padding* are handled with an asynchronous controller. This allows the launch of these special instructions to be decoupled from the instruction pipeline, easing the handling of pipeline hazards and

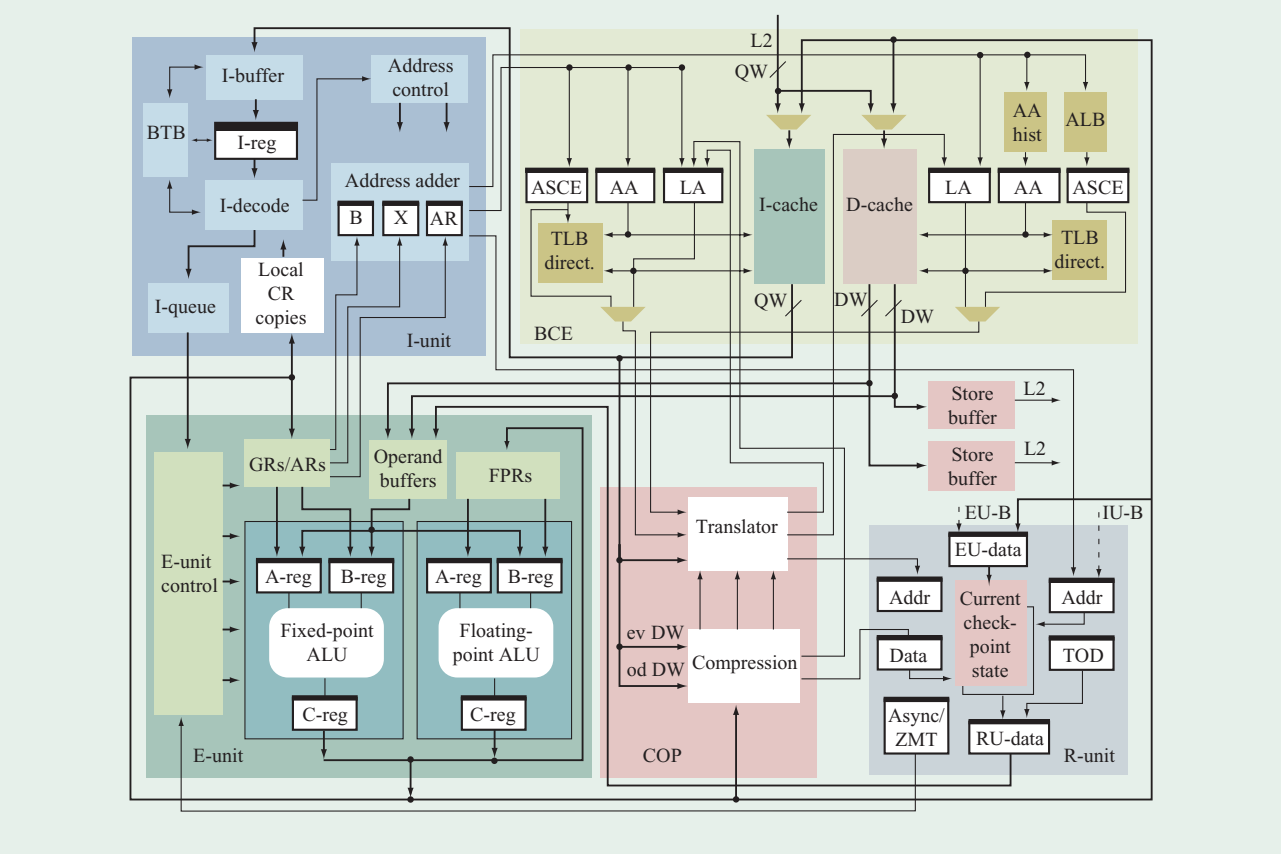


Figure 2

Dataflow of the z900 processor.

exceptional conditions. Also, with a large instruction cache (I-cache) there is no need to have a special internal code instruction memory, as for the G4, G5, and G6 processors. With the z900, millicode is fetched into the I-cache using the same mechanisms as for other instructions. To help some routines where it is critical that millicode instructions be in cache, a new millicode assist instruction was added which creates an I-cache line touch mechanism.

The instruction pipeline of the z900 processor is shown in **Figure 1**. The first cycle shown is the instruction fetch, which can actually take several cycles and is not generally counted when referring to pipeline depth. The next cycle begins at the instruction register, where a single instruction is decoded and the operands for address generation are read from the general-purpose register file (GPR). The pipeline is optimized for instructions that require one operand from the register file and one from memory. These instructions are referred to as RX-type instructions. The address generation cycle is used to add a base and index which are from the GPRs and a displacement from the instruction text to form a memory address. In the following cycle, the L1 cache, directory, and TLB are accessed in parallel. In the subsequent cycle, the memory data is sent to the execution unit, and it is rotated and potentially merged with previously read cache lines in the operand buffers. Then, the execution begins: typically one cycle for most fixed-point arithmetic instructions and three cycles for most floating-point instructions, but these cycles are pipelineable. The last cycle is the stage in which write-back to the register file takes place. Thus, the z900 (as well as the G4, G5, and G6 processors) has a six-stage instruction pipeline with single instruction issue.

The overall dataflow is shown in **Figure 2**. At the upper left is the instruction unit. A hit in the BTB results in an instruction fetch. The returned instruction text is placed in the instruction buffers. The buffers feed the instruction register. The instruction is decoded and placed in the instruction queue until being issued to the execution unit. Also during decode, the base and index registers are read and latched. In the following cycle, the address adder computes the address.

The cache and cache controller referred to as the buffer control element (BCE) are shown in the upper right portion of the figure. The BCE is given a logical (virtual) address and performs a hashed lookup of the upper bits in the absolute (real) address history table. This provides an absolute address to access the cache array. In parallel with the array access is a directory lookup and TLB access. If there is a hit in the directory and the TLB, the operand data is returned. Two doublewords (one fetch and one store) can be accessed from the data cache (D-cache) per cycle. The I-cache provides data for the I-unit as well as the compression and translator units. One quadword can be fetched from the I-cache per cycle.

The execution unit (E-unit), which consists of a fixed-point unit (FXU) and a floating-point unit (FPU), is shown in the lower left portion of the figure. The operands from the D-cache are buffered in the E-unit operand buffers, where they are rotated and merged with other cache lines to form a doubleword of aligned operand data. The FXU executes most of its instructions in one cycle, though some multicycle instructions such as *divide decimal* can take hundreds of cycles. The FPU has a five-stage pipeline, but most instructions execute with a latency of three cycles with a throughput of one per cycle.

Shown to the right of the E-unit are the translator and the compression unit, which together are referred to as the coprocessor unit (COP). Both can make table fetches directly from the I-cache. The recovery unit (R-unit) is shown in the lower right portion of the figure. The R-unit holds the machine state with ECC protection and provides this data in the event of an error. The R-unit also contains system-pervasive functions and miscellaneous functions such as the time-of-day clock (TOD) and trace and instrumentation functions.

Thus, the overall dataflow of the processor has been described. The overall CP chip is 17.9 mm by 9.9 mm and contains 47 million transistors, of which 13 million are in logic circuits and 34 million are in arrays. The L1 caches have a capacity of 256 KB. The chip is implemented in a CMOS process with minimum dimensions of 0.18 μm and seven-metal-layer copper technology. The cycle time of the first products shipped was 1.3 ns, with a junction temperature of 0°C, requiring the use of a chiller (a self-contained cooling unit) to remove the almost 30 W per chip. For more details on the technology or circuits, see [11, 12].

The z900 processor contains some major enhancements by comparison with the G6, both in terms of functionality and performance, while retaining some of its simplicity, such as a simple six-stage pipeline with in-order execution, as well as the issuing of a single instruction per cycle. But the z900 has increased functionality with the implementation of the new z/Architecture, and it also has increased performance. The rest of the paper describes these enhancements in more detail. Section 2 discusses the new split I/D cache, which increases bandwidth as well as doubling the L1 cache size. Enhancements made to the instruction unit, including the BTB, are discussed in Section 3. Section 4 presents a new hardware compression engine which quadruples the performance of the G6 compression implementation. The hardware translator, which deals with the new complexities of 64-bit architecture, is discussed in Section 5. Section 6 describes new features of the execution unit, such as enhanced decimal arithmetic algorithms that aid performance. The overall processor enhancements are summarized in the final section.

2. Buffer control element

The buffer control element (BCE) of the z900 processor consists of the split L1 (I/D) caches and the interfaces with the unified L2 cache. To reduce rework and promote reuse, the I- and D-caches were designed by the same team of logic and circuit designers. The pipeline operations were generally kept the same for both caches. Thus, most of the logical and physical entities, e.g., cache/directory/TLB arrays, are shared between the two cache units. A high-level diagram of the BCE is provided in **Figure 3** to show the similarities between the two cache designs.

The decision to change from a unified cache (in previous z-processors) to split I/D caches resulted from a desire to provide greater bandwidth to the processor by separating the instruction and operand fetch requests. The split design also allows faster access time for larger cache sizes. The caches are both 256 KB in size, with a line size of 256 bytes; both require an access time of only one cycle. The z900 contains an L1 cache twice as big as that of its predecessor, the G6 processor, which had a unified 256KB L1 cache. The z900 L1 caches are managed by grouping the contents into 256 congruence classes of four lines (four-way set-associative). The I-cache is managed as single-quadword (16 bytes)-accessible, while the D-cache is managed as two-doubleword (8 bytes)-accessible by interleaving on even and odd doubleword addresses. Since the D-cache is interleaved, it is possible to process a fetch in one of its interleaves while processing a store in the other interleave.

The z900 BCE required changes in many areas to support the new 64-bit z/Architecture. Logical address paths have been increased from 32 bits to 64 bits, while physical address paths have also been increased to access the larger physical storage provided with the new z900 machines. A new interface has been added between the coprocessor and the I-cache to allow both compression dictionary/table fetches and translation table fetches. A new content-access-memory (CAM)-structured TLB has been added to the pipeline to speed up the selective entry purging required by the z/Architecture. To fully support the split-cache structure, the L2 cache (off chip) and the

L2 interfaces have been changed to support concurrent misses from both I- and D-caches. To support the z/Architecture requirement of self-modifying instructions, the address-compare logic (called program-store-compare, or PSC) has been redesigned and distributed among the L1 and L2 caches. Communications between the I- and D-caches of an individual processor are performed through the L2 using the multiprocessor cache-coherency protocols. To modify an instruction, the D-cache must send a cross-interrogate (XI) command to the L2 requesting the line exclusive. The I-cache must eventually give up the requested line to L2, which then notifies the D-cache that it has the line exclusive. Rather than creating another protocol between an individual processor's I- and D-caches, the cache-coherency protocol among the 20 processors (or 40 L1 caches) is used.

During the address generation cycle, the I-unit provides the logical address requested and its corresponding address information to the caches. Since the cache is organized as an absolute-addressed structure (instead of logical-addressed), the logical address cannot be used directly to index the cache or directory. In the D-cache, the I-unit maintains the existing hash index used in looking up in the absolute address history table (AAHT) for predicting the absolute address bits needed to access the directory and cache. For instruction fetches, the BTB in the I-unit manages absolute address bits rather than using an AAHT. As in the G6 processor BCE, if misprediction occurs, the AAHT is corrected, and the initial request is recycled with the correct guess based on the result from TLB lookup. The corresponding address space control element [(ASCE), the translation table origin, detailed in Section 5] is also looked up using the control register files in the I-cache for instruction fetching and the ALB in the D-cache.

During the array lookup cycle, the translation lookaside buffer (TLB) is indexed with the logical address, using the appropriate ASCE. The corresponding absolute address is obtained if the translation hits in the TLB. If not, the logical request address is forwarded to the coprocessor's translator to perform hardware translation. During the cycle in which the TLB is addressed, the directory and cache are also accessed; if a match is detected in the directory, the data is obtained with a late selection multiplexor from the parallel cache lookup.

During the data return cycle, if data is obtained successfully through the cache, the I-unit is notified of the presence of requested data. If data is missing from the L1 cache, a line fetch request is formulated to be sent through the L2 interface to the L2 cache. In the meantime, the I-unit is notified of the miss and is advised to wait for the return of requested data.

The D-cache has a write-through policy, and the L2 cache uses a write-back policy. Since the reliability of

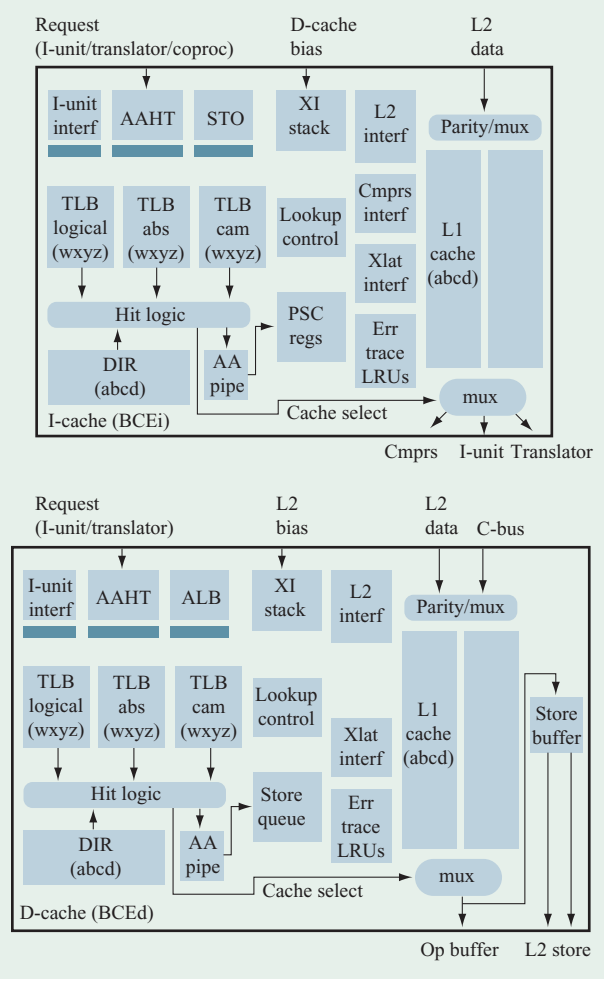


Figure 3

Major components of the buffer control element.

cache array cells was a chief concern, we chose a write-through L1 design with parity protection but with an ECC-protected L2. This also simplified recovery from errors, since the L1 caches contained no architected state that was not already available in the L2. The L1 caches were simply flushed for recovery. Write-back caches have the advantage of reducing bus traffic, but instead we chose to use dedicated wires from each processor to the L2. The processor-to-L2 structure is a 2×10-way binodal set of dedicated bidirectional buses.

Operand stores are written into the L1 cache as each store is executed, and are merged with current data from the cache to form the doubleword. Since the connection to the L2 cache is by two-quadword bidirectional buses, any storage updates are forwarded to the L2 cache when no cache-miss traffic is returning from the L2 cache to the L1 caches. The z/Architecture requirement to support self-

modifying instructions is more difficult with a split I/D-cache design than with a unified L1 cache. To simplify the design and reduce buses in the chip, the multiprocessor cache coherency protocol is used to communicate between the I- and D-caches.

To increase multiprocessor performance, changing a page key (SSKE instruction) or invalidating a selective page (IPTE instruction) had to be very fast. In the G6 processor, the TLB had to be searched for the corresponding absolute address in every entry, which required one cycle per entry for a total of 256 cycles. In the z900, the TLB has 128 entries four-way set-associative, which takes 128 cycles to invalidate a page or change its key. The TLB is indexed by bits 45 to 51 of the logical address, and each TLB entry has three addresses: 1) the logical address bits 0 to 44, 2) the ASCE bits 24 to 51, and 3) the absolute address bits 24 to 51. Also, a TLB entry contains miscellaneous control information bits such as the key, the change bit, and valid bits. For the z900, an additional array is implemented which speeds the lookup of an absolute address; it is called the content-addressable memory (TLB-CAM). The TLB-CAM contains bits 42 to 51 of the absolute address rather than all of the absolute address bits (24 to 51). This provides speed of lookup at the cost of overpurging entries in the TLB. Rather than taking hundreds of cycles to find an entry which matches an absolute address, only one cycle is required. When a selective purge is needed, the pipeline requires four to six cycles to invalidate the requested keys or translations. This speeds up the overall machine performance, since most of the selective purges involve the coordination of many processors.

3. Instruction unit

The instruction unit (I-unit) for the z900 processor required changes to almost all areas to accommodate the new 64-bit z/Architecture. However, much of the fundamental structure did not change. The unit had to expand its entire dataflow to handle the full 64-bit addressing of the new architecture. As mentioned in the Introduction, support for 173 new instructions was added. The majority of these new instructions are executed completely in hardware with no millicode assistance. Many changes were also required to support the dynamic modal nature of the three addressing modes. To improve performance, the fetch interface to the instruction cache and the address generation interlocks were enhanced, the number of outstanding branch paths was increased, and the size and performance of the BTB were increased considerably.

The z/Architecture adds new instructions that allow for programs to switch address modes easily among 24-bit, 31-bit, and 64-bit. One set of instructions is *set addressing mode* (SAM24, SAM31, SAM64), which executes in hardware. This allows program execution to dynamically switch addressing modes without even the need to branch to a different subroutine. To ensure that address calculation for these different modes for both instruction and operand fetching can flow through the instruction unit simultaneously, all addresses in the instruction unit keep mode tags and handle the proper truncation of addresses to the correct mode. This allows for a single 64-bit address and mode tag interface to both of the L1 caches which is independent of the addressing mode of the processor.

When the processor is running in the new z/Architecture mode, program execution is allowed to mix 32-bit and 64-bit arithmetic, since there are separate instructions for the 64-bit operations. This required special handling of updates to the GPR set for address generation. Since it is now possible to update an entire 64-bit register, the lower 32 bits or the upper 32 bits via different instructions, careful attention must be given to which addressing mode the processor is in when these updates are made. This affects the address generation interlock (AGI) logic. Care must now be taken to allow bypass of only data having a width which matches that required for the different addressing modes.

In the G5/G6 processor, the early resolution of data from *load address* (LA) and *load* (L) was allowed in the instruction unit for address calculation before the fixed-point unit would execute and update the GPR file for these instructions. For the z900 processor, additional instructions have been added which also exploit early address generation interlock (AGI) resolution for performance. These include all of the index branches and some linking branches for which the instruction unit has a result that updates the GPR at the execution time of these branches. The branches that are supported for early AGI generation include *branch on index*, *branch and link*, and *branch and save*. The index branches are often used in floating-point loops to increment the register used to index a data array. For linking branches, the register value is often used as a base for storage reference by subsequent instructions.

The G5/G6 processor contained a single unified cache. All operand data and instruction fetch requests to the cache were prioritized in the I-unit. In some cases this resulted in conflicts between the need to fetch operand data and to fetch more instruction text. With the split I/D cache design of the z900, there are now two cache interfaces. While this does not significantly change the priority in the operand fetch logic, it does remove conflicts of branch addresses. The greater change is in the instruction fetch performance. With its own interface, instruction fetching has many more fetch cycles available, allowing more aggressive fetching of instruction text.

With more fetch cycles and the expectation of running more new 64-bit instructions, most of which are 4- or

6-byte instruction formats, more instruction text must be buffered. Thus, the six-quadword (32-byte) buffers of the G5/G6 design have been increased to six double-quadword (64-byte) buffers for the z900. In addition to making the buffer larger, the number of predicted branch paths fetched has been increased by one to allow three outstanding branches.

The z900 processor has the same basic pipeline as the G4 through G6 processors. Note that the G5 processor was the first to use a BTB in its design to predict the target address of branches. Most branches in old workloads for z/Architecture are not relative addressed branches, so a simple direction guess is not very useful. Instead, each BTB entry is very wide to contain the address of the target and so that it can be prefetched into the instruction buffers. The penalties for branches are the same as in the G4 through G6 processors. A branch for which the prediction is in the wrong direction costs five cycles, since the target can be decoded two cycles after execution of the branch on this short six-stage pipeline. A "surprise-guessed-taken" branch (a branch which is not found in the BTB and is statically predicted taken) has a three-cycle penalty, since decode of the target can occur on the execute cycle of the branch. A branch with a wrong target address predicted costs six cycles, since the addresses are compared in the cycle after address generation but the instruction fetch of the target is not performed until the direction is determined.

The most significant performance enhancement in the I-unit was the change to the BTB. In the G5/G6 design it is 1K by two-way associative for a total of 2K entries. In the z900 it is a 2K by four-way associative design for a total of 8K entries. In addition to holding more entries, the BTB can search 32 bytes per cycle instead of 16. Also, the history and least-recently-used (LRU) portions of the BTB are a two-port design. Every time a predicted branch is executed, an update cycle is made to the branch history and LRU sections of the array to update it. Note that the history has three states: invalid, weakly taken, and strongly taken; a surprise-taken branch starts in the strongly taken state. The two-port design simultaneously allows the update of these fields for a predicted branch and a search for future branches. In the G5/G6 single-port design, a search cycle had to be taken away to perform the update. The data portion of the array is still a single port, but it grew in width to accommodate the larger addresses. With these changes, 11 bits of the branch address are used to index the array. The array holds 19 additional bits for branch address for comparison purposes. The array also holds a 30-bit target address. The BTB array can be searched modulo 2 GB on the instruction address of a branch. For the predicted target address, the upper bits are assumed to be the same as the address of the branch itself. Only taken branches are maintained in the BTB.

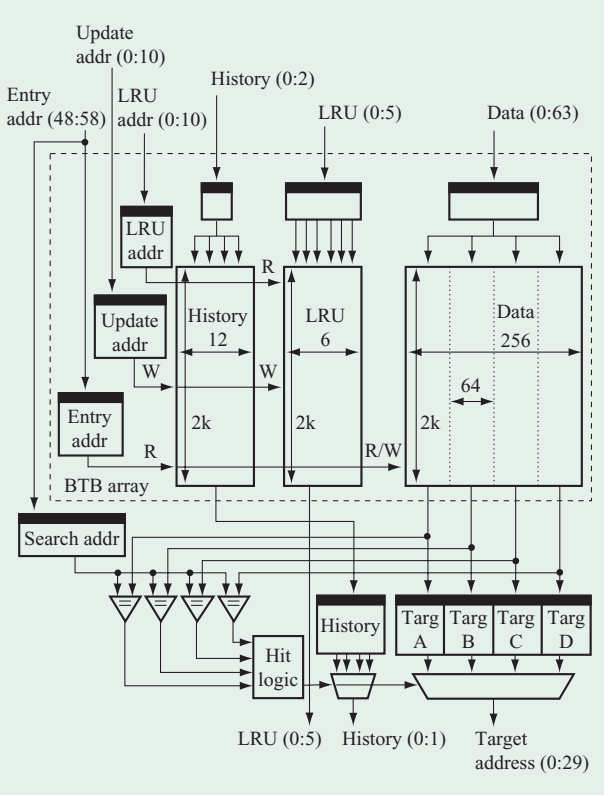


Figure 4

Dataflow of the branch target buffer.

There is an advantage to saving taken and not-taken branches if only a history table is maintained, but the added area of target addresses in a synchronous BTB makes it a disadvantage to save not-taken branches. The BTB is shown in **Figure 4**.

With the much larger size of the BTB in the z900, changes were made in how the BTB was protected for reliability. For the G5/G6, it was part of the I-unit and was replicated. Now it is outside the replicated I-unit and is protected with parity, saving considerable silicon area while still providing soft array failure protection. On hard fails, each of the sets may be disabled independently, allowing the BTB to continue to operate in a degraded mode.

4. Compression unit for data compression and character translation

The compression unit for the z900 implements data compression and character translations. Data compression is commonly used in database applications to save memory. The compression unit performs table loads from the I-cache and then iterates on a compression or translation. The prior G6 processor does not have a

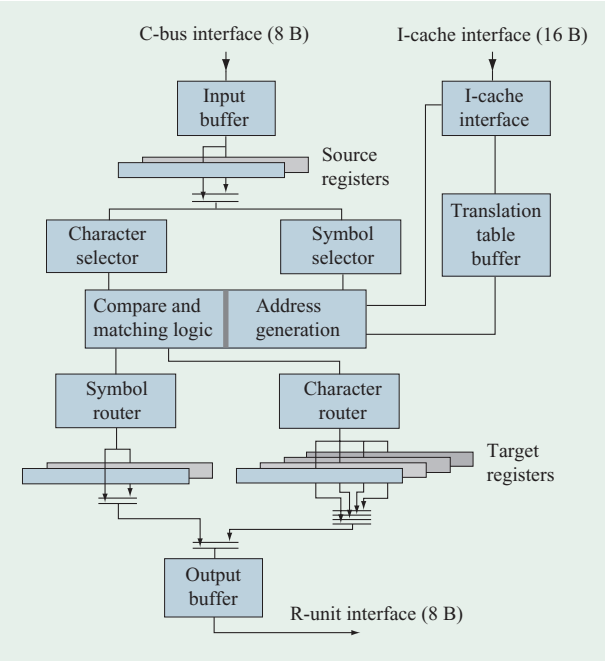


Figure 5

Overview of coprocessor for compression and character translations.

dedicated compression unit and instead performs some millicode assist instructions in the FXU. Thus, the z900 has significantly higher performance than the G6 processor, taking of the order of one third the number of cycles of the G6; also, the cycle time is significantly shorter.

Hardware data compression

The z/Architecture incorporates support for data compression in the processor based on the Lempel–Ziv 2 (LZ2) method with static dictionaries [13–15]. Data compression is invoked by the *compression call* (CMPSC) instruction, which specifies via general registers the source operand address and length, target operand address and length, dictionary address, and a few bits for controlling the algorithm (such as a bit for compression versus expansion). The operands and the dictionary reside in main storage and are accessed through the regular cache hierarchy.

CMPSC is a synchronous processor instruction, which means that the processor cannot continue executing other sequential instructions as long as CMPSC is running. From a system performance perspective, processor utilization and memory/cache utilization (for dictionaries and compressed data) are increased in the short term in order to save disk space and I/O bandwidth in the long term. The net effect on system performance depends on whether the environment is more processor-bound, memory-bound, or I/O-bound. In the case of database queries, the expanded data often has a very short lifetime before being overwritten with new expanded data in the L1/L2 caches and therefore may never propagate beyond L2 cache toward main memory. This effect can reduce overall data traffic in the system and in many cases substantially improves system performance as a result of using hardware data compression.

Character translations

The z/Architecture defines the instructions TR, TRT, and TRE for translating an input stream of single-byte characters bytewise into an output stream of single-byte characters by means of a 256-byte translation table. Input data, translation table, and output data reside in main memory and are accessed through the regular cache hierarchy. The three translate instructions are differentiated by format and ending conditions.

Coprocessor implementation

Predecessor processors of the z900 had implemented data compression and character translates mainly in millicode with extensive hardware assists in the processor. Increasing performance requirements for data compression were the main reason for the choice of a coprocessor-based implementation. In addition, the hardware assists for data compression and character translations or “translates” in the processor turned out to severely restrict efforts for further reduction of logical path lengths.

Both data compression and character translates convert source operand data sequentially into target operand data by means of a translation table or dictionary. This analogy was the motivation to implement those functions in a common coprocessor to exploit the hardware resources efficiently. The coprocessor is attached to the processor via three interface buses, as shown in **Figure 5**. Input data and control information are sent to the coprocessor via the 8-byte-wide C-bus (upper left), from which they are loaded either to the coprocessor control registers or to the 256-byte input buffer. Output data is loaded from the coprocessor to the 256-byte output buffer and from there is sent to the processor via the 8-byte-wide R-unit bus (shown in the lower middle portion of the figure). This bus is operated in multiplex mode and also transmits status information back to the processor. The data transfer between processor and coprocessor in general is under control of millicode, which fetches source operand data from memory and sends it to the input buffer; reads data from the output buffer and stores it to the target operand in memory; and reads and interprets status information from the coprocessor (which primarily indicates the number of empty input buffer slots, the number of data units ready to be read from the output

Table 1 Data compression performance measurements for the z900 and G6 processors.

Record length	*Compression ratio (%)*	*Cycles*		*Speedup compression*	*Cycles*		*Speedup expansion*
		z900 compression	*G6 compression*		*z900 expansion*	*G6 expansion*	
80 B	54.5	1041	2960	2.84	410	767	1.87
1 KB	54.5	11301	36459	3.23	3011	8163	2.71
4 KB	54.5	44722	145579	3.26	11484	32252	2.81
80 B	70.9	1042	3000	2.88	355	526	1.48
1 KB	70.9	11303	36873	3.26	2270	5051	2.23
4 KB	70.9	44680	147052	3.29	8498	19769	2.33

buffer, and, at the end of the operation, the type of ending condition encountered).

The coprocessor reads data from the input buffer, performs the required processing, and stores the result data to the output buffer. While both input data and output data may be considered as sequential data streams, the dictionaries and translation tables are accessed randomly. Therefore, the coprocessor has a logical interface to memory, consisting of an 8-byte-wide address bus, a 16-byte-wide data bus, and a few control lines for the "handshake" protocol shown at the upper right in Figure 5 as the I-cache interface. Because this interface is read-only, it is physically connected to the I-cache rather than to the D-cache, since wiring congestion and timing there are less critical. For CMPSC with dictionary sizes up to 64 KB, every entry is accessed individually when it is needed. For the character translates with a table size of 256 bytes, a fast-streaming mode has been implemented which allows loading of the entire translation table into the coprocessor at the beginning of the operation and eliminates delays on the I-cache interface during processing.

The coprocessor itself comprises the following main functional components:

- *Source registers* are two 8-byte-wide registers that are loaded alternately from the input buffer.
- *Character selector* selects up to five characters from the input registers for processing by CMPSC compression or character translates.
- *Symbol selector* selects one compressed symbol from the input registers for processing by CMPSC expansion.
- *Address generator* calculates a logical address for fetching a dictionary entry or a translation table.
- *Compare logic* performs the compare and matching processes required for CMPSC compression.
- *Translation table buffer* holds the 256-byte translation table for the character translates. It is implemented logically as a register file with a 16-byte write port for loading the translation table with 16 consecutive data shots, and two single-byte read ports allowing translation of 2 bytes every cycle.
- *Character router* loads the output characters of a translate or of CMPSC expansion to the output registers.
- *Symbol router* loads a compressed symbol to the output registers.
- *Target registers* are six 8-byte-wide registers (two for compression, four for expansion) that are stored alternately to the output buffer. Separate registers are used for compression and expansion for timing reasons. Expansion requires four registers, since processing inside an expanded symbol may be either from left to right or right to left.

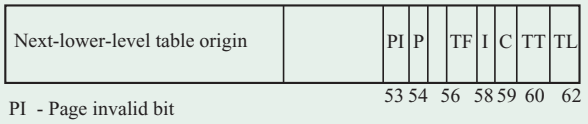
Coprocessor performance

The data compression performance is strongly data- and dictionary-dependent. Basically, a better compression ratio yields better performance, since more expanded characters can be processed with every compressed symbol. Long records show a better per-byte performance than short records because of the start-up overhead of the coprocessor. **Table 1** shows measurements in terms of cycles for the z900 and its predecessor machine, the G6, for different record lengths and compression ratios. All numbers are based on the assumption that all data needed are in the L1 cache. Compared to the G6, the z900 processor requires about a third of the number of cycles to compress data and about half the number of cycles to expand data. This is a significant performance improvement above and beyond the usual cycle time improvements for successive processors.

The raw performance for the character translates is 2 bytes per cycle. In addition, the start-up overhead for initially loading the translation table at the beginning of the execution of every character translate instruction is 20 cycles.

5. Translation unit

The translation unit performs virtual-to-real address translation in hardware. This function has expanded with



PI - Page invalid bit
P - Page protection bit
TF - Table offset
I - Invalid entry bit (all tables except page)
C - Common segment bit
TT - Table type
TL - Table length

Figure 6

Typical translation table entry for z/Architecture.

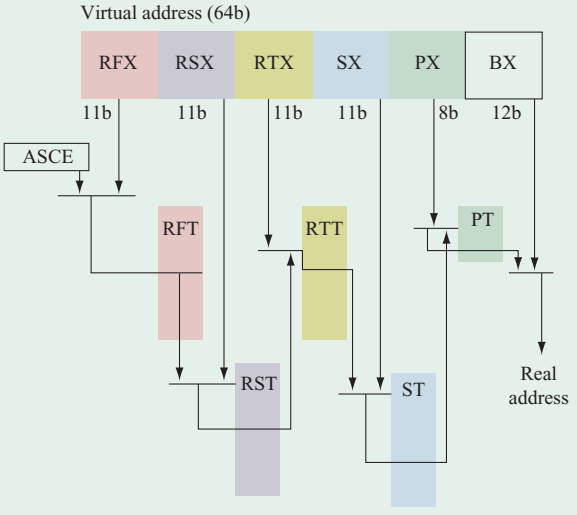


Figure 7

Virtual-to-real translation of a 64-bit address.

the introduction of 64-bit addressing in the z/Architecture. The basic function is rather simple: Find a translation table origin, add an offset to it, find the table entry, and determine where to look next. The translator is started up when either the instruction or data cache translation lookaside buffer (TLB) misses, or for special processor millicode operations. The cache control forwards the missed virtual address, along with any operand access control information (OACR) such as whether the fetch is in primary, secondary, access register, or home mode (which affects the location of the initial translation table origin). The translator performs table fetches by requesting the entries from the I-cache using the same interface as the compression unit.

A typical translation table entry for the z/Architecture is shown in **Figure 6**. Bits 0–51 or 0–52 designate the table origin of the next-lower-level table. Bit 54 on the segment table entry (STE) and page table entry (PTE) is a page protection bit. Bits 56–57 designate a table offset of the next-lower-level table. Bit 53 of the page table entry or bit 58 of the other table entries designates an invalid bit, meaning that this entry is invalid. Bit 59 on the STE designates whether the common segment bit is on, which indicates that this STE entry allows multiple mappings. Bits 60–61 designate the table type, and bits 62–63 designate the table length. Formats for a particular table level may not define all of these bits; for more complete details, see Reference [10].

The additional effort required to complete a z/Architecture translation can be very involved for a 64-bit address. For 31-bit addresses, the first step is to identify which control register holds the segment table origin (STO) for the requested address space controls. The 11-bit segment table index (SX) is used to offset the segment table origin to find the segment table entry, which contains the page table origin (PTO). The 8-bit page table index (PX) is added to the page table origin, and the resulting page table entry (PTE) is read from the I-cache. The contents of the PTE gives the page frame real address (PFRA). The real address is formed from the adding of the base index (BX) to the PFRA. For a 31-bit address translation, there is a maximum of only two table lookups. In access register mode there can be several more table lookups to determine the initial segment table origin.

Figure 7 depicts the translation of a 64-bit address. There are three additional tables for 64-bit translation called the region tables. The upper 33 bits of the address are divided into three 11-bit fields called the region first index (RFX), region second index (RSX), and region third index (RTX). They are used to look up table entries in the region first, second, and third tables (RFT, RST, RTT). The initial table origin is called the address space control element (ASCE) and can specify table lookups starting at the RFTO, RSTO, RTTO, or STO. In this way, only the necessary translation tables have to be built by an operating system, and software can save memory and speed translation by using only the minimal number of tables.

The ASCE can be specified in a control register by primary, secondary, home, or address space mode, or it can be specified in a more complex manner using access register mode. Access register mode effectively creates 2^{11} address spaces. The z/Architecture has a 64-bit virtual address, but with additional addressability created by multiple address spaces. Note that all of these table lookups, including access register mode, are handled in hardware by the translator unit.

In addition to the complexities that access register mode creates, the translator additionally handles SIE

architecture. This architecture defines the host, guest 1, and guest 2 virtual environments, and the z900 maintains many of the architected facilities for each environment in hardware rather than maintaining just the current level. A guest 2 translation has the potential to go through a guest 1 and host translation. By design, one out of three translations is guaranteed not to require table lookups and is a direct map of virtual to real mode with possibly an offset addition required. But even with two levels of translation, many table lookups are possible. For instance, if in the middle of a guest 1 translation the RTE misses in the I-cache TLB, a host translation must be started on this entry, which may very well have up to five table lookups. In a worst-case scenario, a translation may require five host table lookups for all five guest table entries, for a total of 25 table lookups to complete one translation. The guests and hosts can be in different architecture modes, addressing modes, and OACR modes, all of which must be handled by the translator for each translation. The zSeries* and 390 architectures are very rich, which gives the operating system vast capabilities to efficiently run one operating system on top of another (e.g., Linux on z/VM*). All of these translations are handled with the speed of hardware execution, since the translator unit is designed to handle all possible combinations.

Additionally, millicode has the capability of issuing requests to the translator. These instructions become very useful for system operations in which an absolute or real address must be sent to the L2 or I/O processor to perform an operation on memory. Millicode can send an instruction to the translator unit to translate a virtual address into a real one and put the result in a system register accessible by millicode. This millicode special instruction is called *perform translator operation* (PXLO) and is decoded by the I-unit. The I-unit issues the instruction to the D-cache on its operand fetch control bus. The D-cache then transmits the command to the translator. The translator performs the operation and then signals the D-cache, which subsequently signals the E-unit that the operation has been completed. When the E-unit receives the PXLO instruction, it waits for the operand fetch controls to signal completion. When the D-cache signals that the processing is done, it issues an end-of-instruction signal and proceeds to the next instruction.

6. Execution unit

The execution unit (E-unit) of the z900 processor shares the same fundamental design as its predecessor [2]. As mentioned earlier, the E-unit comprises two subunits: the fixed-point unit (FXU) and the floating-point unit (FPU). In order to support the z/Architecture, the FXU required the most extensive changes of the two units. The sheer number of new fixed-point arithmetic instructions required a massive effort to implement the new hardware instruction algorithms. This effort primarily affected the control logic. However, there were additional changes, which included enhancements to the dataflow of the FXU to efficiently perform 32-bit, 64-bit, and mixed 32-/64-bit arithmetic. Specialized functional elements were added to the FXU dataflow to allow complete hardware execution of decimal multiplication and division, and to increase the performance of conversion between decimal and binary formats. There were also innovative changes made to the controls of the FXU to accommodate the aggressive processor clock frequency. The FPU unit was relatively unchanged from that of the G6 processor, but it did receive minor changes to support the new instructions included in the z/Architecture, as well as additional hardware support of instructions previously implemented in millicode.

FXU dataflow enhancements to accommodate the z/Architecture

The hardware implementation of the many new instructions included in the z/Architecture did not require massive changes to the existing FXU dataflow elements. The existing FXU dataflow taken from the G6 processor was already 64 bits wide, since the internal architecture of the G6 processor allowed a limited amount of 64-bit arithmetic. As a review, the FXU dataflow consists of three input registers: the A, B, and E registers, which are each 64 bits wide. The input controls to these registers include independent selection on each 32-bit half, allowing for a robust variety of data selection possibilities. These registers are fed from multiple sources: the general-purpose register array (GPR), the access register array (AR), the operand buffers, or a multitude of result wrap-back paths, as shown in **Figure 8**. The binary adder is 64 bits wide and has byte granularity on its primary controls, thus permitting zeroing and inversion of input data. The bit logical unit/and-insert-under mask (BLU/AIM) is also 64 bits wide and remains unchanged from the G6. The BLU/AIM function is fed from the rotated A-register, the B-register, and the mask generator. The decimal adder, which was only 32 bits wide in the G6, has been enhanced to be 64 bits wide to increase the performance of decimal arithmetic. There is also a new decimal support unit, which contains hardware that allows for performance improvements in the conversion between decimal and binary formatted data.

A significant change to the FXU dataflow for the z/Architecture was made to the GPR array. In the previous ESA/390 architecture, this architected register file had sixteen 32-bit entries. The new IBM z/Architecture defines sixteen 64-bit registers in order to accommodate the arguments and results of either 32-bit or 64-bit arithmetic. In addition to the sixteen architected

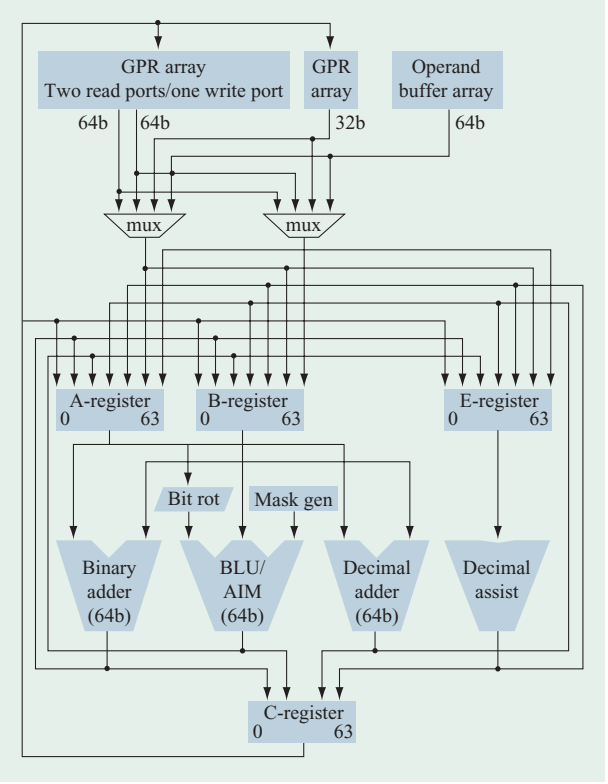


Figure 8

Dataflow of the fixed-point unit.

GPRs, there are also sixteen millicode general registers (MGRs). For implementation purposes, the latter are combined into a single register file which is 128 bits wide and 16 entries deep (see Figure 8), and in which each 128-bit entry consists of an even/odd register pair. The GPR array has two 64-bit read ports and a single 64-bit write port. The write operation is rather complex and allows two 32-bit words of data to be written to any word within a 128-bit quadword. This is accomplished with separate two-bit write flags and enables for each word and combined address bits defining the quadword address.

Changes to the FXU controls: The E-1 pipeline stage

At the onset of the z900 design effort, it was determined that the existing instruction pipeline structure from the predecessor design would not meet an aggressive processor clock cycle time target. Therefore, a redesign of the fundamental instruction staging from the instruction queue in the I-unit to the E-unit was needed. The goal of this redesign was to create a new instruction dispatch mechanism which achieved the cycle time target and also required no additional stages to be added to the overall processor instruction pipeline. The result was the creation of the concept of the E minus one (E-1) stage inside the E-unit.

The E-1 stage can be considered a "virtual" pipeline stage, since its existence does not necessarily imply that another pipeline stage was inserted between the instruction dispatch cycle and the register access, or E0, cycle. Instead, the E-1 cycle in the execution unit overlaps with the instruction dispatch cycle in the I-unit and the cache access cycle. Another reason for using "virtual" as a description of the E-1 stage is that the validity of the E-1 stage is not established until the following (E0) cycle. The main benefit of the E-1 stage is that it allows most of the timing-critical controls that are needed in the E0 cycle to originate directly from latch outputs, thereby allowing the most time for the control signal to propagate through the dataflow. The E-1 stage was most beneficial in the formation of the register array access controls (which control the outgating of operand data to the dataflow) and the mask generation controls (which control the operation of the logical unit). The register array read address path was one of the more timing-critical logic paths in the previous designs, since the access of the register array and the input register selection consumed the majority of the machine cycle. This left very little time in the E0 cycle to form the GPR read address. In the z900 design, the instruction text is available at the beginning of the E-1 cycle, which is much sooner than in the G6. The formation of the register array read address consists of a selection among the timing-critical new instruction text, multiple hold paths, and a control-logic-formed address for multi-cycle instructions.

Another section of control logic which benefited from the E-1 cycle is the mask generation logic, as shown in **Figure 9**. The mask generation signals specify the starting and ending positions of a consecutive string of 1s in a 64-bit string. This mask is used by the bit logical unit to bitwise mask the results of the logical operation during the E1 cycle. Because of the complexity involved in forming the mask, it is preferable to have the starting and ending positions of the mask available at the E0 cycle, which means that the formation of the mask specification is required during the E-1 cycle. Since this is primarily a function of the instruction, it is feasible to form the starting and ending positions of the mask during the E-1 cycle. However, as with the formation of the register addresses, the real challenge is to predict what mask controls should be on the following cycle.

Decimal instruction performance enhancements

The FXU required many changes to improve the performance of decimal instruction execution. In most cases, instructions are moved from millicode emulation

to an algorithm implemented in hardware. These hardware algorithms required the creation of a dedicated multi-cycle state machine which controls preexisting dataflow units and the functionally enhanced decimal dataflow units. As mentioned earlier, the existing decimal adder is widened from 32 bits to 64 bits, which improves the performance of decimal addition, subtraction, and compare. It also aids the performance of decimal multiplication and division. A new decimal assist function is also added to the dataflow to allow high-performance format conversions between decimal and binary.

Decimal adder

In the G4 and G5, the FXU decimal adder was limited to a width of 32 bits primarily because of area and timing constraints. The z900 decimal adder was redesigned to perform single-cycle additions, subtractions, and compares on 64-bit or 16-digit binary coded decimal (BCD) numbers. This adder design uses a common algorithm which carries out a binary addition followed by a decimal correction; however, it essentially performs the correction in parallel rather than serially after the binary add. Decimal carry-outs can be generated either by a binary carry from the digit position or by a sum digit which is larger than 9. The decimal correction is performed by adding a value of 6 to each sum digit which received a carry-in. The adder consists of two pre-sum logic adders, a digit-carry network which produces the hot carries for the 16-BCD-digit pre-sums, and a ±6 increment/decrement correction logic. For each digit for decimal addition or subtraction, four sums are created and selected on the basis of a carry network. For addition, A+B+0, A+B+1, A+B+6+0, and A+B+6+1 are calculated, and for subtraction, A+B'+0, A+B'+1, A+B'+0−6, and A+B'+1−6 are calculated. There are four different possible results for each digit; two are the outputs of the pre-sums and two are the outputs of the −6 corrections. For more details, see Reference [16].

Decimal conversion hardware

A specialized function has been added to the FXU dataflow to facilitate the hardware implementation of decimal conversion instructions and other decimal support instructions. This function can quickly perform decimal conversions between packed and zoned format and can also perform the conversion between decimal and binary formats. It can convert three decimal digits to a 10-bit binary number in one cycle by using three lookup tables and a three-input binary adder. The reverse binary-to-decimal conversion consists of converting a 12-bit binary value to three BCD digits through the use of three table lookups and a BCD adder.

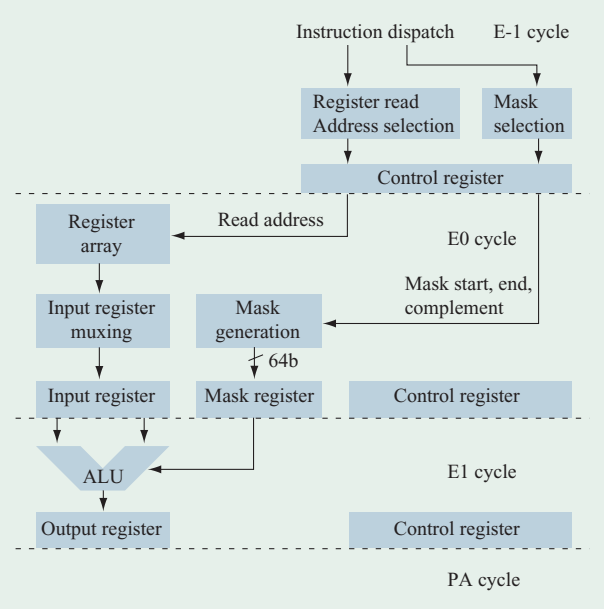


Figure 9

E-1 cycle.

Decimal multiplication and division

In the G4, G5, and G6 processors, decimal multiplication and division are implemented using a hardware-assisted millicode routine. The z900 processor is the first of the CMOS S/390 processors to implement decimal multiplication and division entirely in hardware. The algorithm to perform the decimal multiply depends on the number of significant digits in both operands. It first generates multiples of the multiplicand and stores them in the register file. It then iterates on the multiplier by examining the value of each digit and performs an accumulate and shift operation of the partial product.

Like the decimal multiply algorithm, the divide algorithm is implemented using multiple iterations depending on the number of significant digits of both operands. One quotient digit is generated during each iteration by subtracting the dividend from the divisor using a restoring division algorithm. In some cases, nine subtractions may be required to find the correct quotient digit. By moving the algorithm from millicode to complete hardware execution on the z900, the number of cycles on average was reduced to one half to one third that of the G6 processor.

7. Summary

The z900 processor contains substantial architecture enhancements for supporting the 64-bit z/Architecture as well as increases in cache size, bandwidth, number of

multiprocessors in a configuration, branch target buffer, and decimal performance. This paper has described the enhancements to the L1 cache design, with its new split instruction and data cache, and to the I-unit, with its multiported quadruple-sized BTB. Also, its compression unit is new and requires one half to one third the number of cycles required by its predecessor processors. Its translator unit implements in hardware the complexity of 64-bit z/Architecture translation, including AR translation and SIE modes. Its enhanced execution unit achieves aggressive cycle time goals by adding a virtual E-1 cycle, and performance enhancements to decimal arithmetic instructions have been achieved.

From the start of the IBM mainframe CMOS system line with the G4 processor (introduced in 1994) to the G6 processor (introduced in 1999), performance has doubled every 12 to 24 months. The z900 system continues this performance growth. The performance of the G6 multiprocessor (1999) was 1600 MIPS.[1] By comparison, that of the z900 processor (2000) is more than 2500 MIPS. Uniprocessor performance also increased substantially, even on 32-bit performance benchmarks. In general, uniprocessor performance increased from about 200 MIPS for the G6 to about 250 MIPS for the z900, continuing performance increases over prior generations while adding functional enhancements via a 64-bit architecture.

Acknowledgments

Processor development involves a considerable team effort, from logic design to millicode, to circuit design, and to verification. Most notable in the work described here were the chip integrator, Peter Camporese, the circuit and timing leader, Brian Curran, and the millicode leader, Mark Farrell. Logic design efforts were carried out by Wen Li, Fadi Busaba, John Rell, Michael Kroener, Bruce Giamei, Brian Prasky, Thomas Pflueger, Doug Balazich, Jennifer Navarro, Aaron Tsai, Farhan Mansoor, Siegmund Schlechter, Thomas Streicher, Ute Gaertner, and Gerhard Doettling. Key contributing advisors were Charles Webb, Timothy Slegel, John Liptay, Barry Krumm, Bill Lewis, and Dean Bair. Circuit design efforts were carried out by Yuen Chan, Bill Huott, Leon Sigal, Rainer Clemen, Frank Tanzi, Rocco Crea, Yiu-Hing Chan, Mark Mayo, and Greg Northrop.

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. 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).
3. 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).
4. G. Northrop, R. Averill, K. Barkley, S. Carey, Y. Chan, Y. H. Chan, M. Check, D. Hoffman, W. Huott, B. Krumm, C. Krygowski, J. Liptay, M. Mayo, T. McNamara, T. McPherson, E. Schwarz, L. Sigal, C. Webb, D. Webber, and P. Williams, "600 MHz G5 S/390 Microprocessor," *1999 International Solid-State Circuits Conference*, *Digest of Technical Papers*, February 1999, pp. 88–89.
5. E. Schwarz and C. Krygowski, "The S/390 G5 Floating-Point Unit," *IBM J. Res. & Dev.* **43,** No. 5/6, 707–721 (September/November 1999).
6. T. McPherson, R. Averill, D. Balazich, K. Barkley, S. Carey, Y. Chan, Y. H. Chan, R. Crea, A. Dansky, R. Dwyer, A. Haen, D. Hoffman, A. Jatkowski, M. Mayo, D. Merrill, T. McNamara, G. Northrop, J. Rawlins, L. Sigal, T. Slegel, and D. Webber, "760 MHz G6 S/390 Microprocessor Exploiting Multiple Vt and Copper Interconnects," *2000 International Solid-State Circuits Conference*, *Digest of Technical Papers*, February 2000, pp. 96–97.
7. M. Check, "z900—A 64-bit Processor," presented at the Microprocessor Forum, San Jose, CA, 2000.
8. G. Doettling, K. J. Getzlaff, B. Leppala, W. Lipponer, T. Pflueger, T. Schlipf, D. Schmunkamp, and U. Wille, "S/390 Parallel Enterprise Server Generation 3: A Balanced System and Cache Structure," *IBM J. Res. & Dev.* **41,** No. 4/5, 405–428 (July/September 1997).
9. 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 (2002, this issue).
10. IBM Corporation, *z/Architecture Principles of Operation*, Order No. SA22-7832-00, December 2000; available through IBM branch offices.
11. B. Curran, P. Camporese, S. Carey, Y. Chan, Y. H. Chan, R. Clemen, R. Crea, D. Hoffman, T. Koprowski, M. Mayo, T. McPherson, G. Northrop, L. Sigal, H. Smith, F. Tanzi, and P. Williams, "A 1.1GHz First 64b Generation Z900 Microprocessor," *2001 International Solid-State Circuits Conference*, *Digest of Technical Papers*, February 2001, pp. 238–239.
12. B. W. Curran, Y. H. Chan, P. T. Wu, P. J. Camporese, G. A. Northrop, R. F. Hatch, L. B. Lacey, J. P. Eckhardt, D. T. Hui, and H. H. Smith, "IBM eServer z900 High-Frequency Microprocessor Technology, Circuits, and Design Methodology," *IBM J. Res. & Dev.* **46,** No. 4/5, 631–644 (this issue, 2002).
13. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Info. Theory* **23,** 337–343 (May 1977).
14. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. Info. Theory* **24,** 530–536 (September 1978).
15. IBM Corporation, *Enterprise Systems Architecture/390: Data Compression*, Order No. SA22-7208-00, November 1996; available through IBM branch offices.
16. F. Busaba, C. Krygowski, E. Schwarz, W. Li, and S. Carlough, "IBM z900 Decimal Arithmetic Unit," *Proceedings of the 35th Asilomar Conference on Signals, Systems, and Computers*, November 2001; in press.

[1] MIPS are measured in terms of the geometric mean of several commercial benchmarks which are called LSPR MIPS, and are used to describe relative performance differences between IBM machines.

Received September 21, 2001; accepted for publication March 6, 2002

Eric M. Schwarz *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (eschwarz@us.ibm.com).* Dr. Schwarz received a B.S. degree in engineering science from Pennsylvania State University in 1983, an M.S. degree in electrical engineering from Ohio University in 1984, and a Ph.D. degree in electrical engineering from Stanford University in 1993. He joined IBM in 1984 in Endicott, New York, and in 1993 transferred to Poughkeepsie. He is a Senior Technical Staff Member and was the z900 microprocessor logic design leader. Dr. Schwarz provides support to IBM floating-point unit teams and is on the program committee of the IEEE Symposium on Computer Arithmetic. He is also actively researching computer architecture and heads two IBM/CMU microarchitecture work groups. Dr. Schwarz is currently working on future microarchitectures for IBM zSeries, iSeries, and pSeries processors. He is an author of 26 U.S. patents, 15 journal articles, and many conference proceedings and technical reports.

Mark A. Check *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (check@us.ibm.com).* Mr. Check received a B.S.E.E. degree from the University of Wisconsin at Madison in 1988 and an M.S.C.E. degree from Syracuse University in 1993. He joined IBM in Poughkeepsie, New York, in 1988. He is a Senior Engineer and was microarchitecture and instruction unit logic leader for the z900 processor. He has received a fifth-level IBM Invention Achievement Award, two IBM Outstanding Innovation Awards, and an IBM Outstanding Technical Achievement Award. Mr. Check is currently working on the design of future IBM server processors and systems.

Chung-Lung Kevin Shum *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (cshum@us.ibm.com).* Mr. Shum received a B.S. degree and an M.S. degree in electrical engineering from Columbia University in 1987 and 1988, respectively. He is one of the leading processor design engineers with the IBM Server Division in Poughkeepsie, New York. Mr. Shum joined IBM in 1988 and has had various logic and high-frequency assignments in the development of IBM zSeries central processor units, focusing mostly on I/D caches and load/store units. He is also a lead advisor on the front-end design methodology for the recent zSeries microprocessor designs. His current research interests include design methodology and high-frequency microprocessor design. Mr. Shum is the author of several patents and papers; he has received several IBM Outstanding Technical Achievement Awards.

Thomas Koehler *IBM Server Group, Boeblingen Development Laboratory, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (thkoehler@de.ibm.com).* Mr. Koehler received his B.S. and M.S. degrees in electrical engineering from the University of Stuttgart, Germany, in 1980 and 1984, respectively. He joined IBM in 1986 to work on memory and I/O adapter development for IBM server systems. He is currently the technical leader for the data compression and cryptographic design for the IBM zSeries processor. Mr. Koehler holds several patents; he has received two IBM Outstanding Technical Achievement Awards related to processor design and data compression.

Scott B. Swaney *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (sswaney@us.ibm.com).* Mr. Swaney joined IBM in 1988 in Poughkeepsie, New York, after receiving a B.S.E.E. degree from Pennsylvania State University. He has worked on many aspects of advanced processor design, including timing verification, logic verification, and logic design, in bipolar and CMOS technologies. Mr. Swaney's work is currently focused on reliability, availability, and serviceability (RAS) in enterprise computing.

John D. MacDougall *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (jmacdo@us.ibm.com).* Mr. MacDougall is a Senior Engineering Manager. He received a B.S. degree in computer engineering from Worcester Polytechnic Institute in 1990, joining the IBM S/390 division that same year to work as a logic designer of the buffer control element. He has continued his career in processor development, involved in logic design, floorplanning, and physical design. He was responsible for the definition and design of the z900 translator logic, and has since moved into the management position of the logic team. Mr. MacDougall has received IBM Outstanding Technical Achievement Awards for his work on the S/390 G5 and z900 processors.

Christopher A. Krygowski *IBM Server Group, 2455 South Road, Poughkeepsie, New York 12601 (cakryg@us.ibm.com).* Mr. Krygowski received a B.S. degree in electrical engineering from Clarkson University in 1989 and an M.S. degree from the National Technological University in 1999. He joined IBM in 1989 and has had various logic design and simulation responsibilities in the development of IBM zSeries central processor units. He is currently a Senior Engineering Manager of a department responsible for the verification of future zSeries microprocessor designs. Mr. Krygowski's current research interests include floating-point arithmetic and high-frequency microprocessor design and verification. He is an author of several patents and papers and has received four IBM Outstanding Technical Achievement Awards.