Design of the IBM Enterprise System/9000 high-end processor

by J. S. Liptay

The "high-end" water-cooled processors in the IBM Enterprise System/9000™ product family use a CPU organization and cache structure which depart significantly from previous designs. The CPU organization includes multiple execution elements which execute instructions out of sequence, and uses a new virtual register management algorithm to control them. It also contains a branch history table to remember recent branches and their target addresses so that instruction fetching and decoding can be directed more accurately. These models also use a two-level cache structure which provides a level 1 cache associated with each processor and a level 2 cache associated with central storage. The level 1 cache uses a store-through organization, and is split into two separate caches, one used for instruction fetching and the other for operand references. The level 2 cache uses a store-in method to handle stores.

Introduction

As the circuit and packaging technology for implementing large mainframe processors has advanced and new design techniques have evolved, it has become possible to build more complex processors. The advantage gained from this complexity is an increase in performance well beyond the performance gained by increased circuit speed. Many such techniques have been introduced over the years, and two of the major ones, out-of-sequence execution and cache storage, have had an interesting interaction.

About 25 years ago, before caches had been invented, one of the major limiting factors in processor performance was the time needed to access main storage, perhaps ten times greater than the cycle time of a processor. The major approach of that era in dealing with this problem was to increase the number of instructions being processed at any given time, and to allow operations to take place as soon as the required data were available, which implies allowing operations to occur out of their logical sequence. During that period, IBM designed and introduced the System/360™ Model 91, which had exactly those characteristics [1].

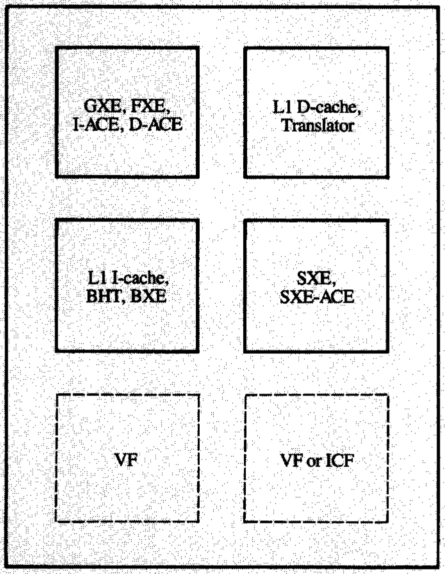

Figure 1

Physical layout of ES/9000 Model 520-based processors.

Immediately after the Model 91 was designed, a new idea was conceived: cache storage. A cache is a high-speed buffer which lies between the processor and main storage and holds recently accessed main storage data. This reduces the average access time for main-storage data by a large factor. Cache storage on a mainframe processor was introduced by IBM on the System/360 Model 85 in 1968 [2], although the particular cache organization used on the Model 85 was abandoned on subsequent IBM processors in favor of the organization described in this paper.

Since a cache is a much more effective technique for dealing with the disparity between main-storage access time and processor speeds than out-of-sequence operation, all large IBM processors since the Model 85 have had caches, and until now, out-of-sequence execution has not been used again. These processors, of which the IBM 3090™ [3] was typical, have processed a number of instructions at the same time, using logic which is specialized for particular stages of processing, but always keeping the instructions in sequence. This is usually called a pipelined design.

Now, with the high-end Enterprise System/9000™ (ES/9000™) Models 520, 640, 660, 740, 820, 860, and 900 (referred to as the "520-based models"), IBM is again using out-of-sequence execution, and has also enhanced the cache structure by introducing a two-level cache to better match the speed of main storage and the processor. These and other enhancements to the design have been made possible by the higher density of the technology available to implement the system, and have been made necessary by the need for higher performance.

The physical packaging of a processor is illustrated in **Figure 1**, which depicts a single board containing six thermal conduction modules (TCMs). Four of the modules are standard, and the other two are for optional features. Both optional positions are used when the Vector Facility (VF) is installed, and one of them is used when the Integrated Cryptographic Feature (ICF) is installed. Because of their common usage of one of these TCM positions, the VF and ICF cannot be installed at the same time on a processor.

The four standard TCMs provide the logic which processes the instructions, and a portion of the cache structure. These TCMs, the board on which they are placed, and their associated power supplies compose what is generally referred to as a processor. However, designers of this logic will often make a distinction, using the term processor or CPU to mean only the logic which processes the instructions, distinguishing it from the logic which implements the cache; that is the way the term CPU is used in this paper.

The purpose of this paper is to describe the CPU and the cache on a logical level. No attention is given to the technology, the physical package, multiprocessing considerations, the vector or cryptographic features, or the rest of the system, except to the extent needed to understand the logical operation of the CPU and cache. Figure 1 identifies where particular elements are located, and **Table 1** defines the meaning of the acronyms used in this paper.

General comments on cache structure

A cache is a hardware mechanism for holding data from central storage so that the data can be retrieved quickly. This is done because most programs make repeated references to the same area in storage within a relatively short time. A cache allows all of these references, except the first one, to be satisfied by a quick reference to the cache rather than a more time-consuming central storage reference.

The type of cache organization described in this section has been in use for about 20 years. It is used for the caches in these models, and is also the basis for the translation lookaside buffer (TLB) and branch history table (BHT) in these models.

The basic element of central storage which is managed is called a line (or sometimes a block), and consists of a contiguous set of bytes which begin at an address which is an integral multiple of the line size. The line size is always a power of two. For example, if the line size is 128 bytes, then lines begin at addresses 0, 128, 256, 384, etc. In binary, these addresses have in common that their low-order bits (25–31) are all zero. Also, all bytes within a particular line have addresses with the same value in their high-order bits (1–24), which is referred to as the line number.

The lines in a cache are divided among a number of partitions, each of which contains the same number of lines. The number of partitions is a power of 2, but the number of lines per partition, called the associativity, may be any integer. Typical numbers might be for a cache to contain 131 072 bytes, divided into 1024 lines of 128 bytes each, which are organized into 256 partitions with four lines each. These are the numbers that apply to the L1 D-cache and L1 I-cache described later, and are used as an example in the rest of this section.

A rule is established that each line of data from central storage is only allowed to be present in a single partition, although it may be in any of the cache lines within that partition. Each partition is assigned a binary number, and the lines that are allowed to occupy that partition are those for which the low-order bits of the line number match the partition number. In the example, the line number consists of bits 1–24 of the address, and there are eight bits in the partition number because there are 256 partitions; therefore, bits 17–24 of the address define the partition a line may occupy. Lines with 00000000 in those bits go to partition 0, lines with 00000001 go to partition 1, and so forth.

There are two main elements which implement a cache: a directory and a data array. The directory records which central storage lines are in the cache, and the data array holds the actual data. The directory is implemented using an array which has a word for each cache partition, with each word long enough to contain an entry for each line in the partition. In the example, the directory contains 256 words, each long enough to hold four line entries. The contents of a line entry are a valid bit and those bits from the central storage address of the line which are above the partition bits. When it is desired to fetch a doubleword, the partition bits in the address (17–24) are used to read the line entries for that partition from the directory, and the high-order bits of the address (1–16) are compared to each of the line entries. If none of the entries matches, the line is not in the cache. If one of them matches (and is valid), that cache line contains the desired line from central storage. The address of the doubleword in the cache data array is then formed by encoding the outputs of the comparators into two bits, and concatenating the address bits which define the partition and the doubleword (17–28). This address is then used to read the doubleword from the data array on the following CPU cycle.

Table 1 Nomenclature and definitions.

ABC	A and B conditional path flags
ACE	address computation element
ACL	array control list
BHT	branch history table
BRAL	backup register assignment list
BXE	branch execution element
CPU	central processing unit, exclusive of cache implementation logic
D-ACE	data address computation element (a subsection of the ACE)
DRAL	decode-time register assignment list
FR	floating-point register
FXE	floating-point execution element
GR	general register
GXE	general execution element
I-ACE	instruction address computation element (a subsection of the ACE)
I-reg	instruction register
ICF	Integrated Cryptographic Feature
IID	instruction identification number
L1 D-cache	level 1 data cache
L1 I-cache	level 1 instruction cache
LRU	least recently used replacement algorithm
SXE	system execution element
TCM	thermal conduction module
TLB	translation lookaside buffer
VF	Vector Facility
VXE	vector execution element

Another method, illustrated in **Figure 2**, allows faster access to the data in the cache. Notice that in the method described above, all except two of the bits needed to address the data array come from the original address; therefore, it is possible to determine which doubleword to read from the data array to within four possibilities just using those bits (17–28). In particular, the four possibilities are the addressed doublewords in each of the four lines in the partition. The faster method of accessing the cache is to read all four of those doublewords at the same time the directory is being read; then, when the cache line containing the desired data is determined, the appropriate doubleword is selected. This often makes it possible to complete a cache access in a single CPU cycle.

When a line is not found in the cache, it is loaded from central storage, and the directory is changed to reflect the address of the new line. The new line is placed in the appropriate partition, taking the place of one of the lines already there. The method of selecting a line to be replaced is known as the replacement algorithm; an algorithm commonly used for this procedure is the least recently used (LRU) algorithm. This algorithm maintains a list for each partition, with the most recently used line at the top, followed by the next most recently used line, and with the least recently used one at the bottom. This list is updated whenever a line is referenced, and when a line is replaced, the least recently used line is selected. The LRU algorithm can be implemented by using one bit to reflect

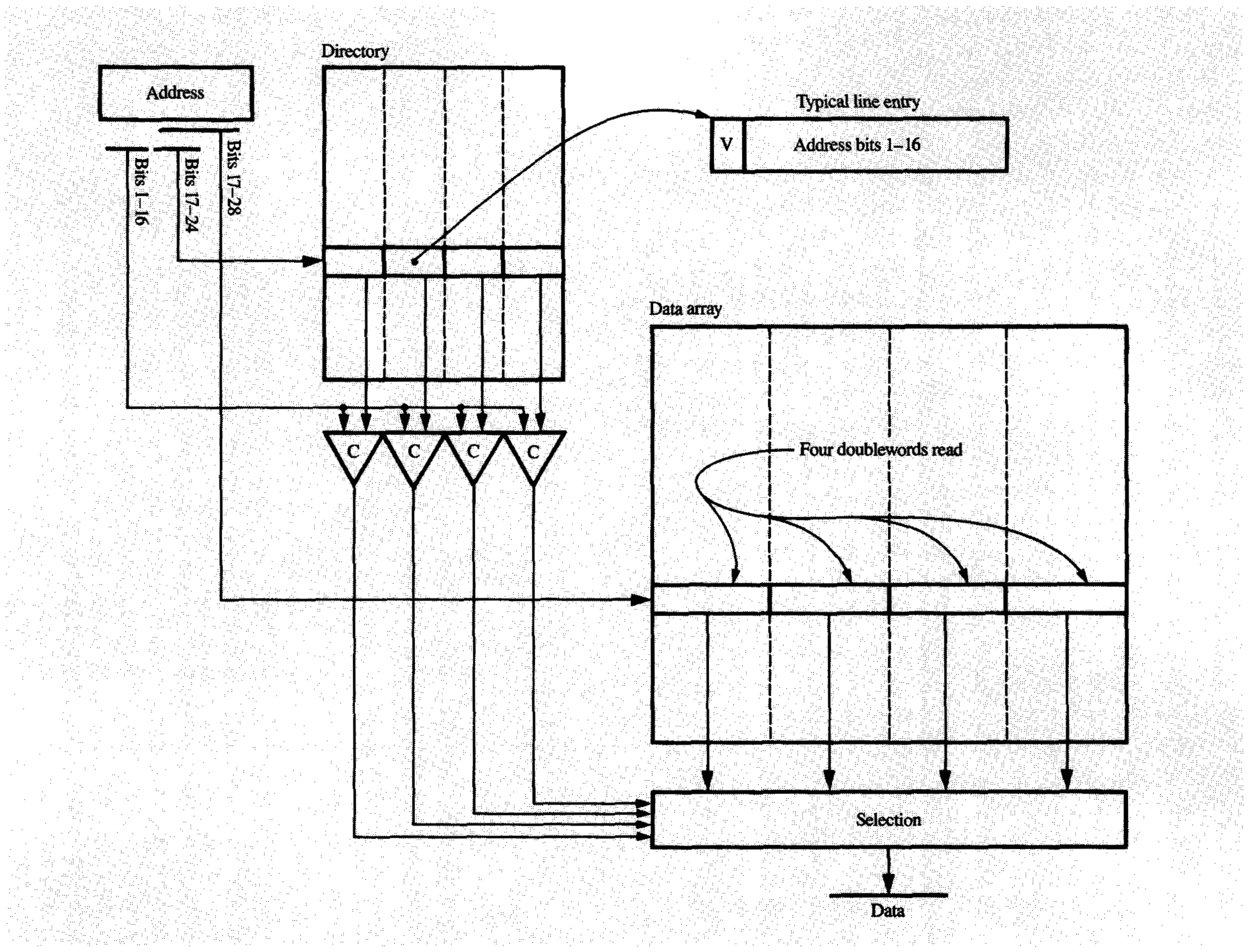

Figure 2

Diagram of typical cache structure.

the relative position in the list of each unique pair of lines. For an associativity of 4, 6 bits are required, and for an associativity of 2, only a single bit is required. Other replacement algorithms are also used which require fewer bits and work approximately as well as the LRU algorithm.

Cache implementations can be divided into two categories on the basis of the way in which stores are handled: one is called a store-through cache, and the other is called a store-in cache. In a store-through cache, every time a store occurs, in addition to updating the cache, the store is also passed to the next level of storage. Then, when a line is replaced in the cache, it need only be overwritten by the new line. In a store-in cache, stores only update the line in the cache, and the next level of storage is left with old values. In this case, when a line which has been changed is replaced, it must be written to the next level of storage before it may be overwritten by the new line. Both methods have been used by computer designers, and there are a variety of considerations in making the decision between the two methods. Some of these considerations relate to how cache integrity is maintained in a multiprocessing system, others relate to performance, and still others relate to the specific characteristics of the technologies available for building the system. Neither method is best in all cases, and the discussion of the considerations involved is beyond the scope of this paper.

There are two other elements in high-end ES/9000 processors which are structured similarly to caches: the translation lookaside buffer (TLB) and the branch history table (BHT). Instead of caching lines of storage, they cache information about a portion of storage—in particular, about how addresses translate, and about the

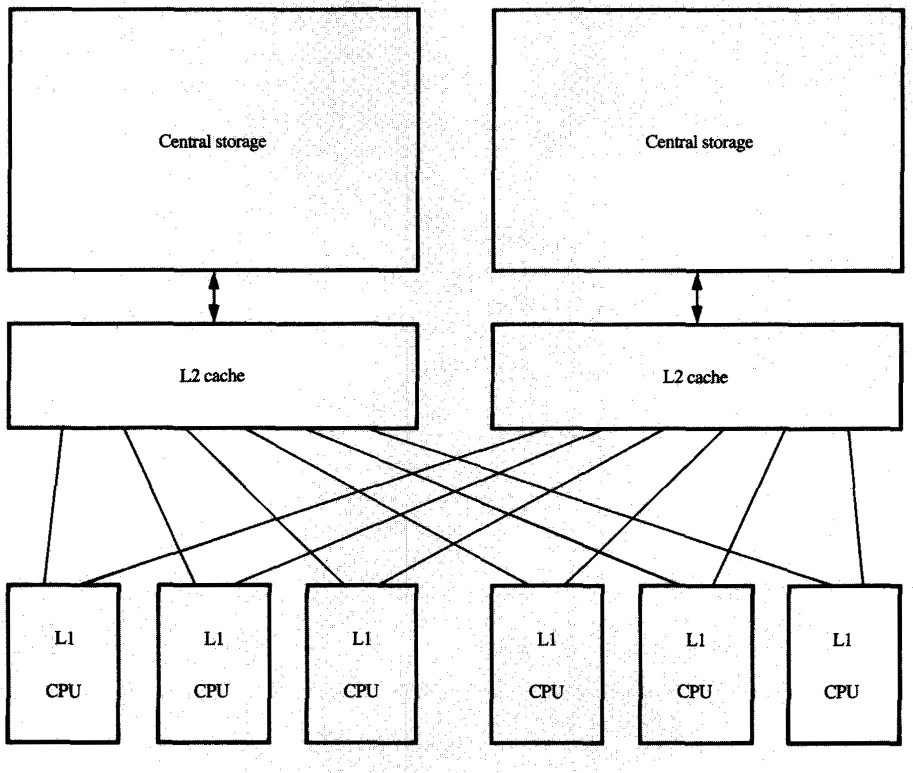

Figure 3

ES/9000 Model 900 system diagram.

presence of branches. They do not have data arrays, but instead have larger directory entries which hold this additional information. Otherwise they are the same.

System and L2 cache structure

The high-end ES/9000 models contain a two-level cache system. The two cache levels are referred to as L1 (level 1) and L2 (level 2). The L1 cache is closer to the CPU, and each processor contains its own L1 cache. The L2 cache is (conceptually) a single cache which serves all of the processors. These two caches provide two levels of speed matching between the CPUs and central storage for high performance.

Figure 3 illustrates the ES/9000 Model 900 system, showing a two-sided structure with three processors, half of central storage, and half of the L2 cache on each side. Each half of the L2 cache operates independently, and serves to buffer lines from the central storage on its side. Central storage is interleaved between the two sides on four-megabyte boundaries; that is, the first four megabytes are on one side, the next four megabytes are on the other side, and so forth. When a processor needs to access a piece of data, it determines from the central storage address which side has the data, and directs its request to that side of the system. Each processor has a full set of control signals and data buses connecting it to each side, and can access data through each side with equal speed. Since there is no performance bias based on which side of the system contains a piece of data, the performance of the system is essentially the same as if it contained only a single L2 cache.

The L2 is a store-in cache which contains a total of four megabytes. Each side contains 8192 lines of 256 bytes each, and is four-way associative. The replacement algorithm uses only three bits and is a variant of the LRU algorithm. The buses between the L2 cache and central storage operate at a rate of a quadword (16 bytes) per cycle. The L2 cache data arrays are four-way interleaved by quadword, thus allowing a high rate of data transfer both between L2 and central storage and between the L2 and L1 caches. As just one example, it is possible for an L2 cache to be reading four different lines and transferring them to L1 caches in four different processors at the same time.

By contrast, the L1 cache is a store-through cache; thus, data transfers to the L2 cache originate in the CPU and consist of individual store operations. Stores can occur at a rate of one per cycle from each CPU, and their size is a doubleword (except that the vector execution element can store a quadword per cycle, subject to certain restrictions). Because of the potentially high rate of store traffic to the L2 cache, it includes a 16-quadword store buffer for each processor connected to it. When an instruction creates stores to successive doublewords, the CPU tags those stores appropriately, and they are organized into quadword stores as they are put into the L2 store buffer.

Data transfers from the L2 cache to the L1 cache consist of L1 lines (128 bytes), with the transfer occurring at a quadword per cycle.

L1 cache structure

The L1 cache actually consists of two separate and independent caches, as depicted in **Figure 4**: an instruction cache (L1 I-cache) devoted to holding lines from which instruction fetches have been made, and a data cache (L1 D-cache) devoted to holding lines for operand references. Several motivations led to a split-cache design, all centered on improved CPU performance. It was desired to reduce the number of cache misses, because each one requires a relatively lengthy access to fetch a new line. Although the CPU does not have to stop operation on a cache miss, there is usually a sequence of dependencies on the data which result in the CPU losing a significant portion of the time required for the line fetch. The way to reduce the number of cache misses is to make the cache larger so that it can hold more data. The desired design point was a 256KB cache, but the technology available limited the cache size to 128 KB on a single TCM when the associated control and data path circuitry was considered. While not as good as a single 256KB cache from a cache-miss point of view, two 128KB caches are nevertheless considerably better than a single one.

Another consideration was a desire to improve the cache fetch bandwidth—the amount of data which can be fetched in a given amount of time. On previous large IBM processors with only a single cache, only one fetch could be made each cycle, either for instructions or for data. It was the practice on those processors to give operand fetches priority over instruction fetches, and there were times when the CPU was unable to decode instructions because it had been unable to make the needed instruction fetches. By implementing two caches, one devoted to each type of request, two requests can be made each cycle, and there is no further interference between the two types of requests. This is of particular value in a CPU which has a relatively high-performance design point, and needs to make more requests within a given number of CPU cycles.

A further consideration was the number of places in the CPU which need to connect to the cache, and their distance from the cache. If there is only a single cache, all requests must have priority determined among them, all addresses must come together in one place, and the data output from the cache must be powered up with enough copies to go to all of the places that need storage data. With two caches that are specialized in their usage, fewer requests go to each cache, and the output from each cache goes to fewer places. Moreover, there is a potential for placement of the caches such that the distances involved are shorter. Without doing two complete designs, it is difficult to measure the value of this, but the designers believe that having two caches has reduced the time required for paths that might limit CPU cycle time.

For these reasons a split-cache design was chosen, with each cache as large as would fit on a TCM (that is, 128 KB). Both caches have lines of 128 bytes, are four-way associative, and use the LRU replacement algorithm. The same central storage line can be in both L1 caches if it has been accessed both for an operand and an instruction.

The data array in the L1 D-cache is two-way interleaved by doubleword. This permits a store and a fetch operation to occur at the same time, and it permits a quadword to be fetched or stored (by accessing the same address in both interleaves), which is useful when lines are loaded from the L2 cache, and for accesses by the vector execution element. It is not possible for the processor to fetch two doublewords from the cache at the same time because only one address can be looked up in the cache directory on each cycle, and because data paths to return two independent pieces of data are not provided.

The L1 D-cache includes a line-fetch buffer which receives lines that are being loaded from the L2 cache and holds them until cycles are available to write the data into the cache without interfering with CPU fetches. This improves performance by eliminating a potential source of interference, and does not introduce any performance penalty because a piece of data in the line-fetch buffer can be fetched by the CPU as fast as if it were in the cache.

The addressing structure associated with the L1 caches is considerably more complicated than that associated with the L2 cache. In the L2 cache, a piece of data is identified by its address in central storage, and that is the only address that need be considered. However, the System/390® architecture (ESA/390™) includes a rich address-translation structure which allows the physical central storage address at which a piece of data is located to be different from the address generated by the program (the logical address). Relocation of data is done in units of 4096 bytes called pages, based on entries in a set of tables

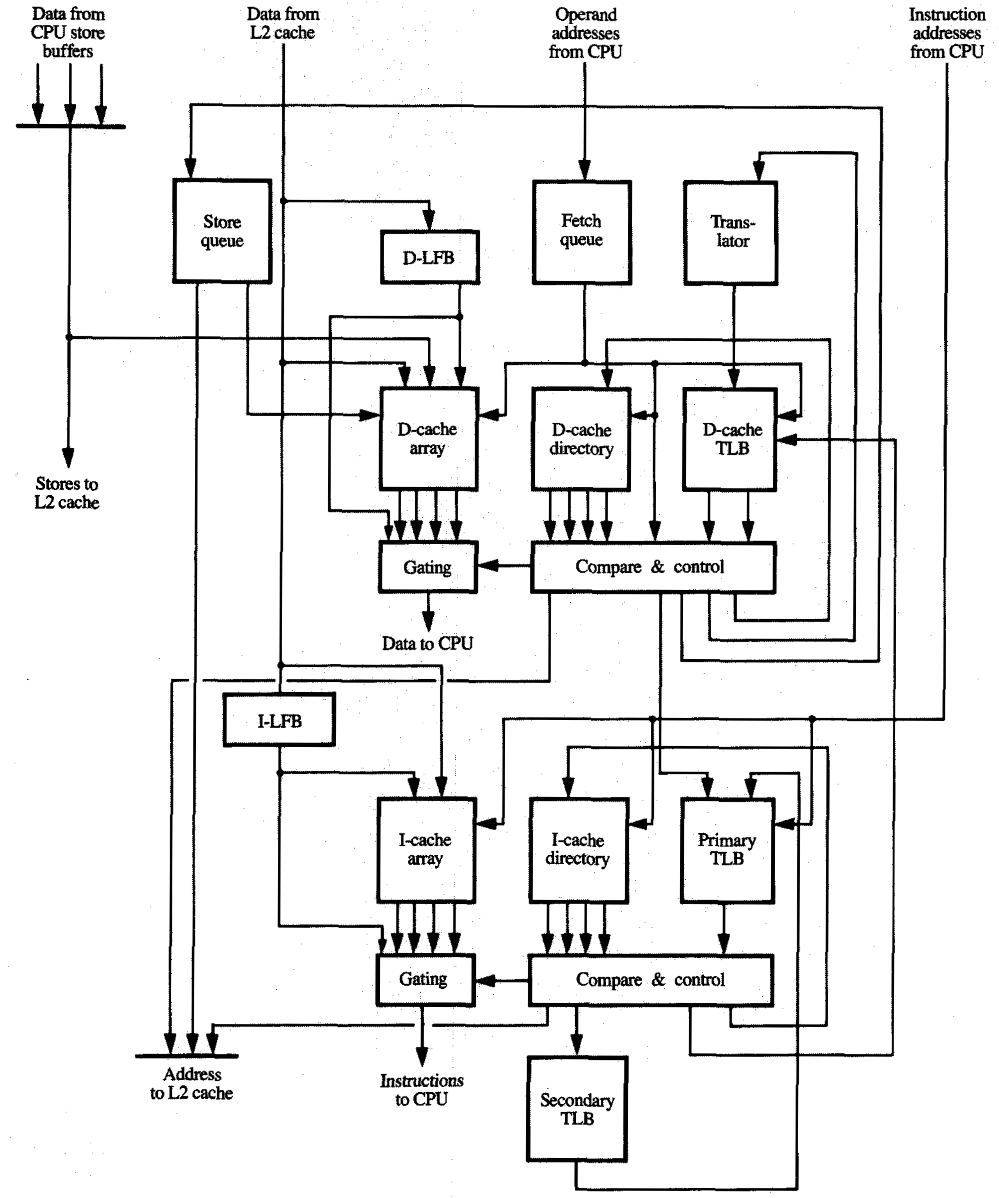

Figure 4

L1 split cache structure.

in central storage. This address translation is carried out by the logic associated with the L1 caches. First, there is a mechanism called a translator which takes a logical address and carries out the required operations to determine its central storage address. Because this process takes a number of cycles and would greatly degrade performance if it were used for every storage reference, there is also a translation lookaside buffer (TLB) which remembers recently performed translations, so that the result can be found quickly if it is needed again. The TLB is structured similarly to a cache, contains 512 entries, and is two-way associative. Each entry contains the logical address of a page, the central storage address to which it translates, the storage protection key associated with that page, and some control information.

The CPU places its fetch requests and their associated logical addresses into an eight-position fetch queue between the CPU and the L1 D-cache in the logical order of the instructions with which they are associated, and they are taken from the queue in the same order. When the L1 D-cache processes one of these requests, it reads the two entries from the TLB partition addressed by bits 12–19, reads the four entries from the cache directory partition addressed by bits 17–24, and reads the doubleword within each of those cache lines defined by address bits 25–28. The L1 D-cache control logic examines the TLB output and the addresses of the lines in the cache to determine whether an entry in each matches the address being looked for. If so, the data can be accessed, and the appropriate doubleword of data is gated to the output bus. All of this occurs in one processor cycle. If neither logical address from the TLB matches, the logical address is sent to the translator, and after the central storage address is determined, a new entry is made in the TLB. If none of the four addresses from the cache directory matches, a line is loaded from the L2 cache.

The first step in processing stores is to make a test request to the store address. This request is the same as a fetch except that no data are returned from the L1 D-cache to the CPU, data are loaded into the cache in a special store mode, and the checking for access exceptions is based on the requirements for storing. The result of this fetch is that it is known whether the address can be accessed for storing, where the data are in the cache, and what the central storage address is. This and other information, such as the number of bytes to be stored and the store buffers which are assigned to the data, are placed into one of the six positions in the store queue. When a store-type instruction changes multiple doublewords, but all of them are within a single L1 D-cache line, only a single test request is made, and only a single store-queue position is used, although multiple doublewords of store buffering are assigned. If the stores extend across multiple L1 D-cache lines, a test fetch is made and a store-queue position is assigned for each line affected. Later, as the instruction executes, the data to be stored are placed in the appropriate store buffers. Still later, after completion of the instruction, the data from the store buffers are written into the L1 D-cache, and sent to the L2 cache. Writing into the L1 D-cache does not require another access to the TLB or the cache directory because this information is already in the store queue. This is what allows stores to take advantage of the interleaving of the data arrays and to operate at the same time as another fetch.

The size, line size, and partition structure of the L1 I-cache are the same as the L1 D-cache. Also, the L1 I-cache has a line-fetch buffer, can receive a quadword each cycle during a line fetch, and provides a doubleword in response to a fetch. There are two areas of significant difference—the handling of stores and the TLB structure. Stores are never performed to the L1 I-cache; if a line in the L1 I-cache is changed by a store, it is invalidated rather than updated. The TLB associated with the L1 I-cache has two levels, referred to as a primary TLB and a secondary TLB. The primary TLB contains 32 entries and is one-way associative, and the secondary TLB contains 256 entries and is four-way associative. If a desired fetch address is found in the primary TLB, the access requires only a single cycle. If an address is not in the primary TLB but is in the secondary TLB, two extra cycles are required for the access, and the address is moved into the primary TLB. If an address is not found in the secondary TLB, the L1 I-cache logic is not able to translate the address because it does not contain a translator. Instead, the address is sent to the L1 D-cache translator for processing, and before translating it the L1 D-cache looks first in its own TLB. Therefore, logical addresses presented to the L1 I-cache have the benefit of being looked up in three TLBs before they are translated by the translator. This rather different TLB structure is used because of the different patterns of addresses presented to an instruction cache, and for other reasons related to dealing with some special cases, which are beyond the scope of this paper.

Instruction decoding and branch processing

Figure 5 depicts the main elements involved in instruction fetching, instruction decoding, and branch processing. Most of the connections between the elements in Figure 5 are control signals, and only the major ones are shown.

The instruction stream which is expected to be executed is fetched from the L1 I-cache as doublewords and is buffered in a five-doubleword instruction buffer. From there instructions are gated to the instruction registers (I-regs) and decoded. Decoding consists of examining an instruction, determining what resources it needs, and breaking it into parts which are sent to the appropriate places to have their required operations performed. The processor contains two I-regs to hold instructions being decoded; it can decode two instructions at a time, subject to some restrictions. Decoding is always in order; only consecutive instructions are considered for simultaneous decoding, and if the first one cannot decode, the second one is not permitted to decode either. Certain instructions must decode by themselves: all six-byte-long instructions, and those requiring more complicated and unusual processing. A branch may decode at the same time as its preceding instruction, but the following instruction may not decode with the branch. There are limits on the data paths available to transmit instructions to various parts of

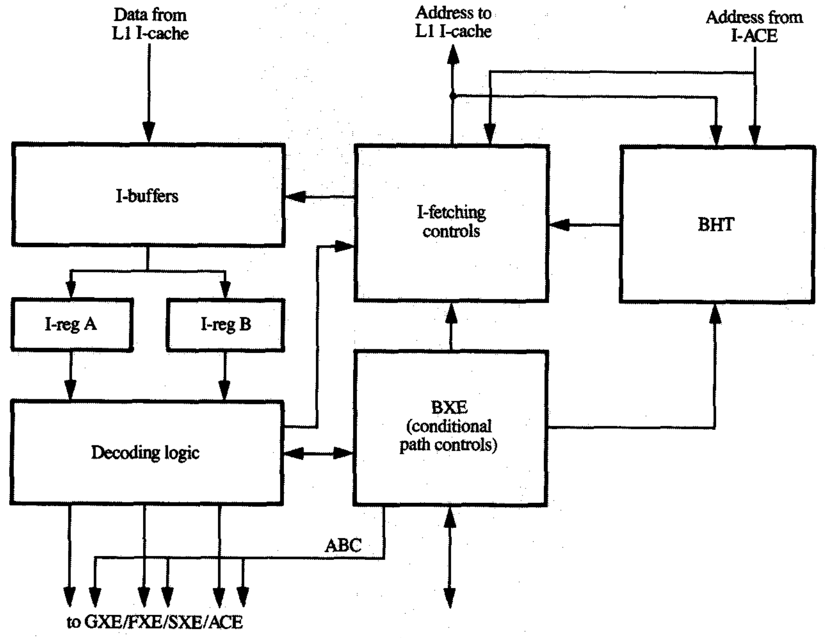

Figure 5

Instruction decoding and branch processing logic.

the processor which impose restrictions on decoding, and instructions may not decode if needed resources are unavailable.

When a branch occurs, the processor makes a judgment as to which way it is most likely to go, and both instruction fetching and instruction decoding proceed according to that judgment. The processor does not fetch instructions along the alternate path. To improve branch processing, the CPU has a branch history table (BHT) which remembers branches recently taken, and the address to which they branched. The BHT is structured similarly to a cache, contains 4096 entries, and is two-way associative. Each entry can record information about one branch. Each time a branch is executed, a decision is made about whether it should be in the BHT. The criterion is simple: If the branch should be judged taken when encountered again, it should be in the BHT; if it should be judged not taken, it should not be in the BHT. If it is decided that a branch which is in the BHT should not be there, its entry is marked invalid; if it is decided that a branch which is not in the BHT should be there, it is written into the appropriate partition.

To discuss how this criterion is applied to particular branch instructions, it is useful to divide them into three categories: those that are always taken, BRANCH ON CONDITION, and the rest. Using the criterion given, those branches which are always taken obviously belong in the BHT. On the other hand, BRANCH ON CONDITION instructions are variable in how they are used, and when looked at as a group, roughly half are taken, and half are not taken. However, when individual BRANCH ON CONDITION instructions are examined, there is a greater degree of consistency, with many taken most of the time and many not taken most of the time. Therefore, a good predictor of what a BRANCH ON CONDITION will do on its next execution is what it did on its last execution; consequently, if one is taken, it is kept (or placed) in the BHT, and if one is not taken, it is left out of (or removed from) the BHT. The remaining branches, BRANCH ON COUNT and BRANCH ON INDEX, are used in a more regular way, almost always to close a loop. Therefore, it is a good guess that one of these branches will be taken on its next execution, even if it was not taken on its last execution; consequently, they are always kept (or placed) in the BHT. That is, they are treated the same as branches that are always taken.

To understand the criterion for placing branches in the BHT, it is worth noting that the BHT serves two purposes, first to improve the judgment of the CPU about which branches are taken, and second to make the target address of the branch available sooner so that fetching of the target instruction stream can begin sooner. In the case of branches which are always judged to be taken, that judgment could be made by examining the branch, with no need for it to take up space in the BHT. However, that examination could not occur until the branch reaches the I-register, and the target address would not be known until it is calculated. By being recorded in the BHT, the instruction-fetching controls are alerted to the presence of a branch several cycles earlier, during the instruction-fetching process, and are provided with the address to which it branched the last time; this address is used to redirect instruction fetching. This often makes it possible for the target instruction of a branch to be available for decoding on the cycle immediately after the branch decodes, thus avoiding the discontinuity in processor operation that is usually associated with branches.

Fundamentally, a BHT entry contains two pieces of information—the address of a branch and its target address from the last time it executed. The address of the branch is used to determine where the branch entry is placed in the BHT and to identify it in the manner that was described for caches, except that bits 1–11 are not implemented because simulation indicates that just assuming that they match would not significantly affect performance. As discussed above, the target address is used to redirect instruction fetching.

Although the BHT works well to improve branch processing, there is no guarantee that it is right, and there are a number of reasons why it can be wrong. A branch may be erroneously recognized because bits 1–11 of the branch address are not recorded in the BHT. The instruction stream in storage may have changed since a branch was found there. A branch may go to a different address than it did the last time it executed, or it may not be taken at all. A taken branch may be found in the instruction stream that was not recorded in the BHT. The processor contains logic which checks that all of the decisions based on information from the BHT were correct, and, if not, takes corrective action. Primarily, this involves canceling instructions decoded along a wrong path that have been sent throughout the processor for execution.

When a branch decodes, the instructions which follow it are said to be part of a conditional path. If a second branch decodes, the instructions following it are still part of the original conditional path, and are also part of a new conditional path. Two conditional paths, designated A and B, can be active at one time. A two-bit ABC (A and B conditional path) field is maintained to reflect the conditional status of each instruction decoded. The first bit is on if the instruction is part of path A, and the second bit is on if it is part of path B; this information is carried with all operations performed for the instruction. At some point, for each branch, it is decided either that all of the decisions were made correctly, or that something was wrong and the path must be canceled. If all of the decisions were correct, a path-correct signal is sent out, and in all of the ABC fields the bit associated with that path is turned off. That path may then be used for another branch. If something was wrong, a path-wrong signal is sent, all operations which are part of that path are canceled, and instruction fetching and decoding are restarted at the correct instruction address. The control logic for the A and B conditional paths, and some other branch-resolution logic, are collectively referred to as the branch execution element (BXE).

Execution elements

The phrase "executing an instruction" means performing the principal operation called for by the instruction. In the case of ADD, it means adding the two operands together, as distinguished from other operations such as fetching the instruction, decoding it, calculating its address, fetching its operand, and putting the result away. Most instructions are executed in one of three execution elements containing specialized logic. They are the general execution element (GXE), the floating-point execution element (FXE), and the system execution element (SXE). A vector execution element (VXE) is added when the optional Vector Facility (VF) is installed. Also, there are other places in the CPU where a few instructions are executed. One of these is the address computation element (ACE), whose principal purpose is to compute addresses and make storage requests, but which also executes a few instructions such as LOAD ADDRESS. The branch execution element (BXE) controls conditional execution, resolves whether branches are taken, and operates with other execution elements to execute most branches; for example, for BRANCH ON COUNT the BXE controls the conditional path and resolution, but the GXE performs the necessary processing and testing of the general register involved. Only one instruction, BRANCH ON CONDITION, is executed by the BXE operating alone. The data flow diagram in **Figure 6** shows the execution elements.

The GXE executes most instructions which perform operations on the GRs, and most of the other simple instructions. Its instructions typically take only a single cycle to execute. Some examples are ADD (A, AR), EXCLUSIVE OR (X, XR, XI), STORE (ST), and COMPARE (C, CR). Some of the longer instructions executed in the GXE are MULTIPLY (M, MR) and COMPARE AND SWAP (CS, CSD). In some respects the GXE is a single execution element, and in other respects it is two separate but closely related execution elements. It has a single six-position queue to hold instructions

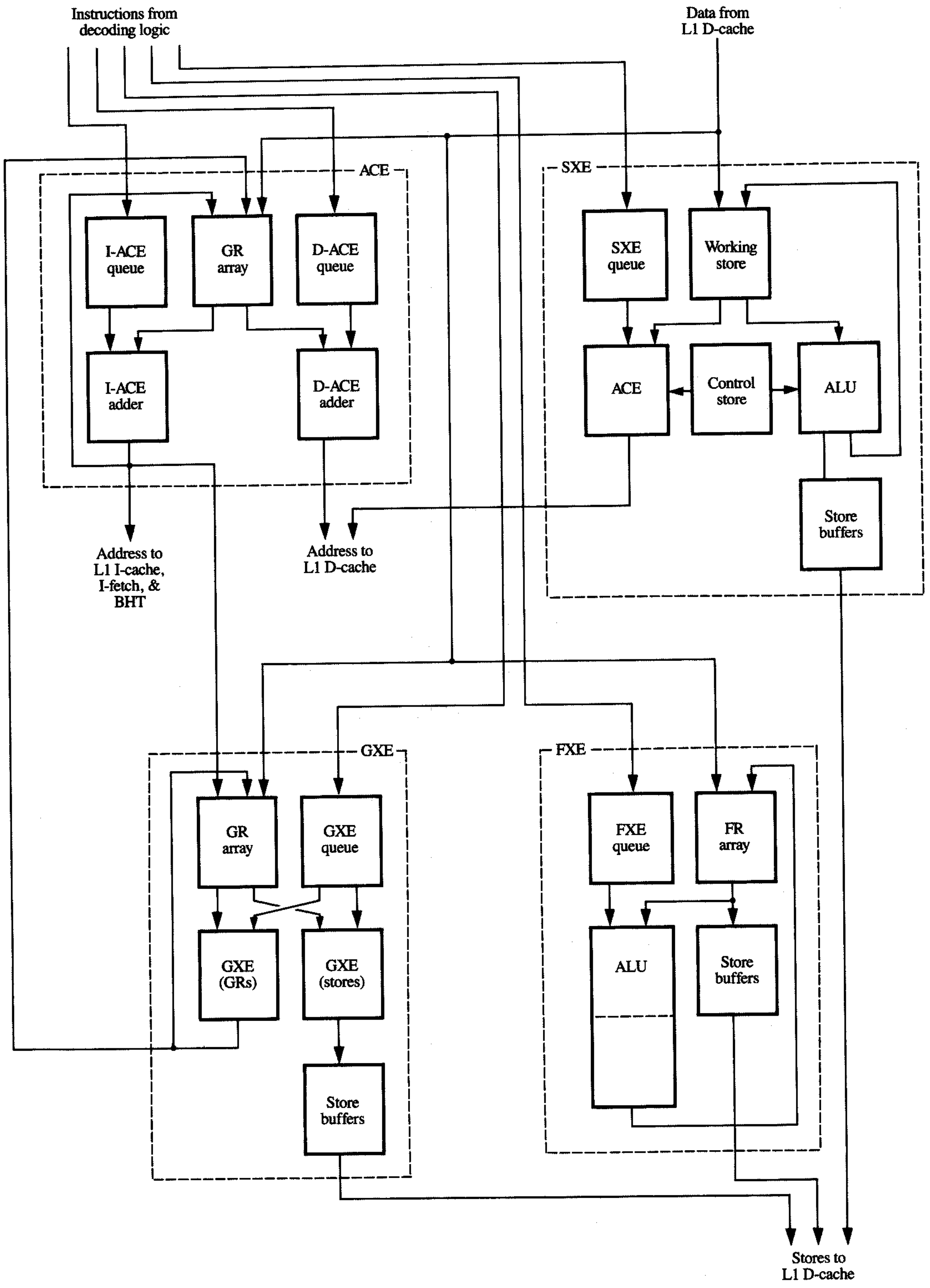

Figure 6

Execution element data flow.

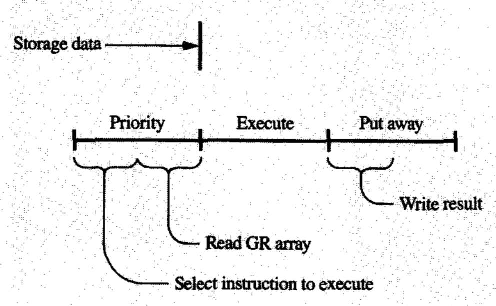

Figure 7

Timing for a typical GXE instruction.

awaiting execution, but it has two sets of execution logic which operate in parallel and independently. The division of work between these two sets of execution logic is determined by where the result is placed. One set is for instructions which put their result in a GR, and the other set is for instructions which put their result in storage. A few instructions (such as COMPARE) do not put a result in either place and can execute in either set of execution logic. COMPARE AND SWAP, which puts a result in both places, requires both sets of execution logic. It is possible for two instructions to be taken out of the instruction queue at the same time, with one going to each set of execution logic, and producing results at the same time (which go to different places). Since most instructions require only a single cycle to execute, an instruction execution rate of two instructions per cycle can be maintained for a period of time. Instructions in the queue can execute as soon as their input operands are ready, and thus may be executed out of sequence with respect to their order in the program. If two instructions are eligible for execution at the same time, the older one gets priority. The GXE includes a 32-word array to hold GR values and to buffer operands fetched from storage. It also contains 16 doubleword store buffers to hold results until the proper time for them to be stored. These, along with others associated with the FXE and SXE, constitute the store buffers previously described.

Timing for a typical GXE instruction is illustrated in **Figure 7**. On the cycle before its execution, the instruction is selected as having the highest priority among those which are ready, and its operands are read from the register array. An instruction using an operand from storage is considered ready for execution the cycle before the operand returns, so that it can execute as soon as the operand arrives. Then, during the execution cycle the required operation is performed, and on the following cycle the result is written into the GR array or store buffers. If there are two instructions awaiting execution, one of which needs the result produced by the other, they may execute on successive cycles because data paths are available to take a result from the output of the execution logic and return it directly to its input for use by another instruction on the next cycle. Because the execution of GXE instructions does not involve complicated sequences of operations, the GXE is controlled with combinatorial logic circuits.

The FXE executes the floating-point instructions. It also has a six-position instruction queue, and instructions may execute out of sequence, but only a single instruction may be selected for execution each cycle. The execution logic in the FXE has a pipeline structure for multiply, add, and subtract. The pipeline is two cycles long, and when a result is produced it can be gated directly to the input registers for use in another operation. In the case of the add and subtract, if postnormalization is not required, the result is available as an input to a new operation after a single cycle. In addition to this pipelined execution logic, there is separate logic to perform divide and square root, which iterate for a number of cycles, and special logic to control extended-precision operations. The FXE contains a 16-doubleword array to hold the floating-point registers (FRs) and to buffer storage operands, and 32 doublewords of store buffers to hold store data until the proper time for storing. The timing of FXE instructions is similar to that of GXE instructions, except that the execution time is two cycles long. The FXE is controlled by combinatorial logic circuits.

The SXE executes the storage-to-storage instructions (including decimal) and the instructions which deal with system control functions, such as the timers, the program status word, and the control registers. These instructions take multiple cycles to execute and involve a richer set of control sequences than those executed in the other elements. Accordingly, the SXE contains a 16 384-word control storage which controls several arithmetic and logical elements and a number of registers. The SXE also contains a working storage of 32 doublewords to hold architected status information and provide a work area for some instructions, and eight doublewords to buffer storage operands. Their implementation and timing are similar to those of the GRs and FRs. The SXE contains 64 doublewords of store buffering.

The ACE is composed of three independent sections of logic, known as the I-ACE, the D-ACE, and the SXE-ACE, which collectively have the responsibility for the calculation of addresses and for manipulations that must be performed on them. The I-ACE and the D-ACE each contain a three-input adder which is used to perform address computations; each is capable of calculating an address each cycle. Each has a queue which holds operations waiting to be performed; the queue for the I-ACE is two positions deep, and that for the D-ACE is four positions deep. They share access to a copy of the GR array from which each can read two GR values each cycle. This copy of the GR array is kept exactly synchronized with the copy used by the GXE. The distinction between the I-ACE and the D-ACE is that the I-ACE calculates addresses used to access the L1 I-cache (branch target addresses), and the D-ACE calculates addresses used to access that L1 D-cache. The I-ACE also calculates the address for LOAD ADDRESS and the shift instructions which are not used to reference storage at all.

Because instructions which are executed in the GXE and FXE have their storage operands accessed by the D-ACE, the GXE and FXE do not contain any logic to make accesses themselves. On the other hand, the SXE deals with some instructions of sufficient complexity that decisions about storage accesses and the calculation of addresses must be handled during the course of execution; therefore, the SXE needs the ability to access the L1 D-cache itself. From a conceptual point of view, the D-ACE must stop its normal operation and allow the SXE to control it during the execution of such instructions. However, there is a problem with this because the SXE, the D-ACE, and the L1 D-cache are located on different modules, and there would be excessively long delays in the control paths. Therefore, the SXE contains an SXE-ACE with the necessary logic to manipulate addresses and make access requests directly to the L1 D-cache. Usually the SXE-ACE is initialized with addresses computed in the D-ACE, which then stops making requests, allowing the SXE-ACE to make the requests needed for the instruction. When the instruction finishes, the D-ACE resumes operation. In this way, at any given moment the L1 D-cache is receiving requests from only a single source.

Controlling out-of-sequence operation

To give a clear explanation of how out-of-sequence operation is controlled, it is necessary to define two words which by dictionary definition are synonyms, but to which specific and distinct meanings are ascribed. One of these words is *finished*, which means that the execution of an instruction has finished everything which needs to be done, and all of its results have been placed in their proper places. The other word is *completed*, which means that all of the actions carried out by the instruction have become irrevocable. Before an instruction is completed, it is in a tentative state, and sufficient status must be retained in the processor to make it disappear if that is required. After an instruction is completed, it becomes part of the amorphous past which brought the processor to its present state, and any specific information about it can be forgotten. An instruction is always finished before it is completed, and whereas the finishing of instructions can happen in any sequence, their completion must always occur in their logical sequence.

The essential first step in controlling a processor with out-of-sequence operation is to keep a record of the logical sequence of the instructions. To do this, a five-bit instruction identification number (IID) is assigned to each instruction when it is decoded. IIDs are assigned in numerical order, wrapping around from 31 to 0; a limit is placed on instruction decoding so that a newly decoded instruction is not assigned an IID which is still in use. When a branch is found to have been guessed wrong, the IIDs assigned to instructions along the wrong path cease to be in use and are reused for the next instructions that are decoded. For example, if IIDs 4–20 are in use, and IIDs 11–20 are for instructions along a conditional path which is canceled, IIDs 11–20 cease to be in use, and the next instruction decoded is assigned an IID of 11. The IID of an instruction is carried with the instruction, and with most operations done on its behalf, as it passes through the processor.

The completion control logic uses these IIDs to bring order to the out-of-sequence operations taking place in the processor. These controls receive "finished reports" from the parts of the processor which execute instructions. These are reports that everything required for an instruction has been done, and include the condition code produced (if any) and any interruption conditions generated. An instruction is identified in a finished report by its IID. The completion controls record the status of the 32 possible outstanding instructions in the processor, they know whether a particular IID is in use, and they record the information received in the finished reports. They continually monitor the oldest outstanding instruction, and when it is finished they generate a completion report for it. This completion report, which is distributed through the processor, identifies the instruction by its IID and causes the processor to finalize the results generated by the instruction. If an instruction creates an interruption condition, the interruption is processed at completion time, and no further instructions are allowed to complete. The completion controls can complete two instructions in each machine cycle.

A number of pieces of data may be changed by an instruction, and all values of these pieces of data become final when the instruction completes. These data can be divided into two categories: those that change frequently as a program executes, and those that change infrequently. Those pieces of data which are considered to change frequently are the instruction address, the condition code, storage, the general registers, and the floating-point registers. These are the principal object of the design described here, and the purpose of this design is to control changes to these data items without restricting operation of the CPU. Changes to other data items are controlled in

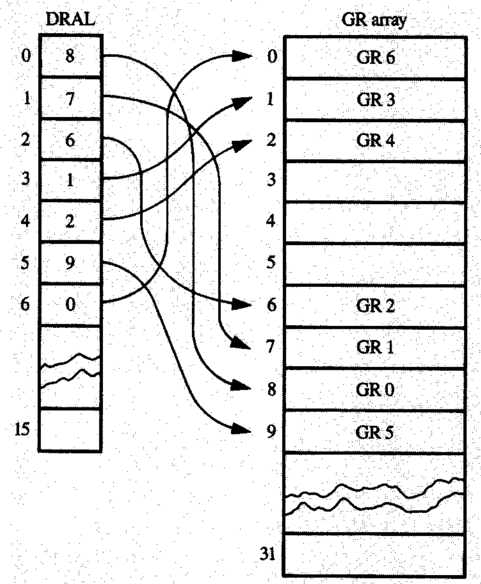

Figure 8

Decode-time register assignment list (DRAL).

less dynamic ways, often by processing the instructions which change them with overlap disabled or restricted.

The instruction address is updated at completion time by the length of the instruction(s) being completed, and the condition code is updated with the value (if any) from the finished report which has been saved in the completion controls.

Storage updating is handled by buffering the results generated until completion time; this is the primary purpose of the store buffers described earlier, and is the reason why they are large enough to hold the largest result that can be generated by an instruction. At completion time, the buffered data are released for storing, and although it requires several cycles to carry out, the transfer of the data to storage becomes irrevocable. In this way, storage retains its old value until completion, and is updated only after the instruction completes. Although this entails some delay in updating storage, it is normally not a problem, because there are usually a moderate to large number of instructions between one which changes a storage location and the next one which fetches from that same location. However, when only a few instructions intervene, the one which fetches from the location incurs a delay while it waits for the one that stores to complete and for the store to actually take place. MOVE LONG (MVCL) is a special case because it can store arbitrarily long results and is handled in segments with overlap disabled.

To understand the mechanism used for the GRs and FRs, it is necessary to understand that the underlying objective of the mechanism is to minimize the number of times data are moved. It is desired that an instruction result be placed in its final location as soon as it is generated; however, it is also necessary that the old contents of a register be preserved until completion in case the instruction is canceled, and it is desired to accomplish this without moving the data. This creates a conflict. If there is one piece of circuitry in the processor which always holds, for example, GR 5, it is desired to keep the old contents of GR 5 in that place until completion, and also to put the new value there as soon as it is generated, which is before completion.

This conflict is resolved by breaking the fixed relationship between a register and a specific piece of circuitry. Instead, arrays are used which contain more locations than there are GRs and FRs, array positions are assigned to hold particular GRs and FRs, and those assignments change dynamically as execution proceeds. For example, if an instruction changes GR 5, the old value of GR 5 is found in the array position which has been assigned to it, and a different, unused, array position is assigned to receive the result generated by the instruction. Until completion, the array position with the old value of GR 5 continues to hold it, and as part of the completion process that array position becomes unused and the one with the new value becomes the one that represents GR 5. Thus, for a period of time two array positions represent GR 5, but at different logical points in the execution process. If multiple instructions which change GR 5 were being processed, different array positions would hold each of the possible values GR 5 can have. In summary, instead of moving data around to save needed values, the data are left in one place and status information is updated. The way this is done is referred to as the virtual register management algorithm.

The 16 GRs are supported by a 32-position array, and the four FRs are supported by a 16-position array; these two arrays are physically and logically separate. In the remainder of this paper, only the GRs are described, but most of what is said applies to the FRs also.

Figure 8 illustrates a table called the decode-time register assignment list (DRAL) and its relationship to the register array. The DRAL has 16 entries, each corresponding to a GR and containing a pointer to the array location which contains that GR. For example, position 2 in the DRAL contains a 6; this means that GR 2 is in array position 6. The DRAL contains the phrase "decode-time" in its name because its contents correspond to the point in the program which has been reached by instruction decoding; its contents do not correspond to the array positions assigned to various GRs as of completion.

During decoding, the GRs referenced by an instruction are looked up in the DRAL, and the register numbers in the instruction are replaced by the number of the array position containing the GR. When an unused array position is assigned to receive a new value of a GR, the DRAL is updated to point to the newly assigned array position. **Figure 9** illustrates this process. The RX ADD instruction at the top is shown as it appears in storage. In this case, GR 3 and GR 4 are the index and base registers which are added to the displacement, X'408', to form an address. A word fetched from this address is added to the contents of GR 1, and the sum is placed in GR 1. During decoding, these GR references are replaced by array positions 1, 2, and 7, which are found by looking in the DRAL. In this case, the unused array position selected to receive the new value of GR 1 is array position 3, which is added to the instruction as a new field. This forms the conceptual internal instruction shown at the bottom which says to add array positions 1, 2, and the displacement together to form an address, fetch a word from that location, add it to array position 7, and put the sum in array position 3. Also, the DRAL is updated so that GR 1 is shown as being in array position 3; in this way, a subsequent instruction which uses GR 1 will use array position 3, which is the place to find the value generated by this instruction. This internal instruction is only conceptual because other things are also happening to it. It is being divided into two parts, of which one goes to the ACE to cause the address computation and fetch, and the other goes to the GXE to cause the required execution. In addition, fields such as the IID and ABC fields are being added to it, and the operation code is being changed to an internal form. There are also (though not described here) status bits and control logic in the processor which prevent operations from taking place until the required data are available.

Figure 10 illustrates two backup register assignment lists (BRAL-A and BRAL-B) and their relationship to the DRAL. Their structure is identical to that of the DRAL, and each has the capability of having the entire DRAL copied into it in a single machine cycle, or of having its entire contents copied into the DRAL in a single machine cycle. One BRAL is associated with each conditional path; when the path is started the BRAL copies the contents of the DRAL, and if the path is canceled, the BRAL is copied back to the DRAL. Thus, when a path is canceled the DRAL is restored to the value it had when the path was started. To provide the capability of copying an entire DRAL or BRAL into one another, it is necessary to implement the DRAL and BRALs using logic circuits instead of arrays.

Figure 11 illustrates a table called the array control list (ACL) which contains 32 entries, one for each position in

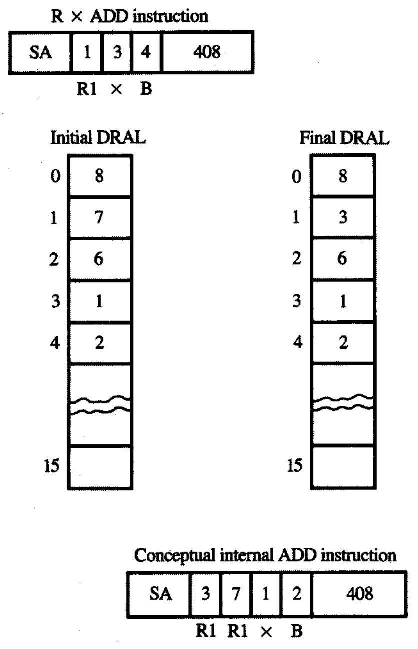

Figure 9

Transformations during the decoding process.

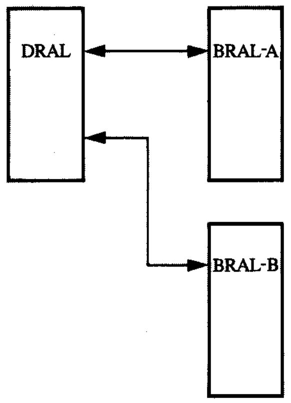

Figure 10

DRAL/BRAL structure.

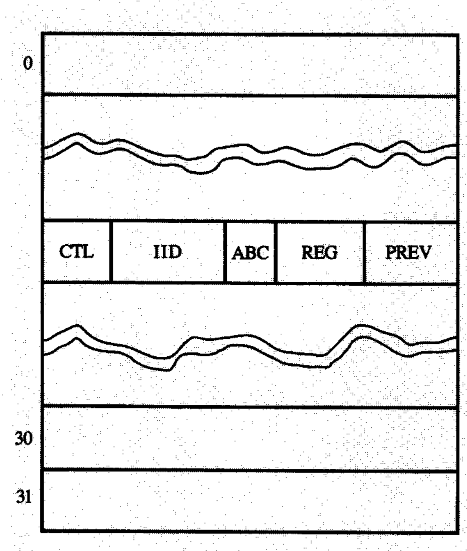

Figure 11

Array control list.

the array, of which a representative one is shown in detail. The CTL field contains two bits which record the status of the array position. They have the following meanings:

- 00: Available.
- 01: Pending but not loaded.
- 11: Pending and loaded.
- 10: Assigned.

If an array position is in the "assigned" state, it is holding the architecturally official copy of the GR identified in its REG field. There are always 16 array positions in the assigned state, with one assigned to each GR.

If an array position is in the "available" state, it is not being used for anything. When an array position is needed to receive a new value, it is selected from these available array positions. To simplify the logic which makes this selection, even-numbered array positions are assigned to even-numbered GRs, and odd-numbered array positions to odd-numbered GRs. If the CPU is not processing any instructions which change GRs, all array positions not in the assigned state are available.

The two pending states are for array positions assigned to hold values generated by instructions which have not yet completed. The "pending-but-not-loaded" state signifies that the final value has not yet been placed in the array, whereas the "pending-and-loaded" state signifies that it has. Only in these two states do the IID, ABC, and PREV fields have significance. The IID identifies the instruction which is generating the value for this array position, and the ABC field reflects the conditional status of that instruction. The PREV field implements a chained list of the array positions assigned to a particular GR by pointing to the one which holds the logically preceding value. It is set at decode time by using the value in the DRAL which is replaced when the number of this array position is written into the DRAL.

Figure 12 is a state transition diagram which shows the states in which an ACL entry can be, and the state changes which are possible. An ACL entry normally cycles from the available state to the pending-but-not-loaded state, to the pending-and-loaded state, to the assigned state, and back to the available state, that is, transitions A, B, C, and D.

Transition A from the available state to the pending-but-not-loaded state occurs when the array position is selected to receive a new GR value to be generated by an instruction which is decoding. The next transition, B, to the pending-and-loaded state, occurs when the instruction has executed and the value is written into the array. For most instructions this coincides with the generation of its finished report. Transition C, to the assigned state, occurs when the instruction completes. This state change is accomplished by comparing the completion report IID with the IID in each ACL entry, and changing all ACL entries with matching IIDs to the assigned state. Multiple entries may be changed to the assigned state at the same time because some instructions change multiple GRs, and because two instructions can complete each cycle.

The ACL entry may then remain in the assigned state for an extended period of time. Transition D, back to the available state, occurs when a new instruction, one which changes the GR to which this array position is assigned, completes execution. The way in which this happens is that whenever an array position makes transition C into the assigned state, the array position pointed to by its PREV field (which must be in the assigned state) is changed to the available state (transition D). In this way, there is always exactly one array position in the assigned state associated with each GR.

Transition E, from either of the pending states directly back to the available state, occurs when instructions which have been decoded are canceled without being allowed to complete. There are two circumstances in which this occurs: wrongly judged branches and interruptions. Array positions which are in a pending state and are associated with a conditional path are marked by a bit in the ABC field. If a path-wrong signal is received, all array positions associated with that path return to the available state.

When the completion controls reach an instruction which has had an interruption reported, they take the appropriate action for the type of interruption reported and then cancel all instructions which are still in progress. As part of that process, all array positions which are in either pending state return to the available state (transition E). When this happens, the DRAL reflects the state of the array at the point decoding had reached, which no longer has any meaning. It must have its contents reconstructed through a search of the ACL to find the array position associated with each GR. This takes a number of cycles but is overlapped with the initial interruption processing when no GRs are referenced, and therefore effectively takes no time.

As was mentioned earlier, the GR array also provides buffering for operands which have been prefetched from storage. This is accomplished by using the array position which has been assigned to receive the result of an instruction to also buffer its storage operand. Since that array position is assigned at the time the instruction decodes, but does not receive any value until the instruction executes, it is available for use during the time in which an operand buffer would be needed. To do this, it is necessary to provide an independent port for writing into the GR array from the L1 D-cache data bus; this is easy to do because the technology used to implement the GR array includes that capability. When the operand is needed for execution, it is read from the GR array using the same data paths used for reading register values.

Using the GR array for operand buffering presents a minor problem with those instructions, such as RX COMPARE, which do not need an array position assigned to receive their result (because there is no result) but which do fetch operands from storage and therefore may need an operand buffer. For those instructions, an array position is assigned anyway. Then, at completion time, instead of the array position entering the assigned state, it goes directly back to the available state (transition F in Figure 12). From a control point of view, this is accomplished by setting its PREV field in the ACL to point to itself; then the natural operation of the controls at completion will cause it to move directly to the available state.

The way in which operand buffering is done also presents an opportunity for the implementation of RX LOAD. On many previous processors, LOAD fetches its operand from storage to an operand buffer and then takes an execution cycle to move it from the operand buffer to the register. In this design, when the operand is fetched from storage it is placed in the array position selected to receive the final result. Since a LOAD instruction does not change the value from storage, and since the operand has already been placed in the correct place, there is nothing further that could be done during an execution cycle. Therefore, LOAD is not executed by any execution element, reducing interference with other operations and

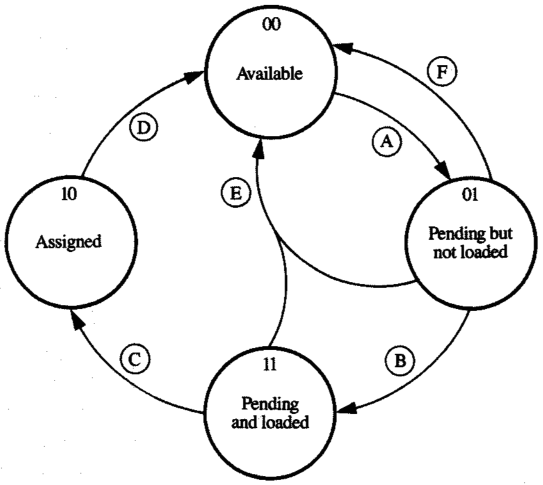

Figure 12

State transition diagram for an entry in the array control list (ACL).

```
Program:                 CYCLE         1    2    3    4    5    6    7
                         DECODE     -- +A,B+C,D+E,F+-G-+---+---+---+
A-A    4,ADDR1           D-ACE      -- +-A-+-D-+-F-+---+---+---+---+
B-AR   5,6               I-ACE      -- +---+---+---+-C-+---+---+---+
C-LA   5,8(5)            L1 D-CACHE --+---+-A-+-D-+-F-+---+---+--
D-NI   ADDR2,15          GXE (GR)   - +---+---+-B-+-A-+-E-+-G-+---+
E-SR   4,7               GXE (ST)   - +---+---+---+---+-D-+---+---+
F-L    8,ADDR3           WRITE GRs  +---+---+---+-B-+-A-+CEF+-G-+
G-AR   8,4               WRITE ST B +---+---+---+---+---+-D-+---+
                         FINISH REP +---+---+---+-B-+-A-+CDEF+-G-+
```

Figure 13

Timing example for a sequence of instructions.

making its result available a cycle sooner than would otherwise be the case. The finished report for LOAD instructions is created by the L1 D-cache. LOAD MULTIPLE and floating-point LOAD (long) operate similarly.

Because of the number of changes which can occur at one time, and their relatively complex relationships, the ACL is built with logic circuitry rather than an array.

Timing example

Figure 13 is a timing diagram which shows the interaction of several instructions. The code which is timed does not represent any useful computation and is not meant to be representative of code which runs on the CPU; rather, it is an example which was constructed to illustrate some common aspects of CPU operation for discussion purposes. It is assumed that nothing is happening in the CPU at the beginning of the example, and that needed operands are found in the cache. The instructions are designated by the letters A–G, and are considered one at a time.

Instruction A is an RX ADD instruction. On cycle 1 it decodes and has its address calculated in the D-ACE. When nothing is queued up for the D-ACE, it can take a request from the decoding logic, give it immediate priority, read the required general registers, and calculate the address, all in the same cycle that the instruction decodes. At the beginning of cycle 2, the address goes to the L1 D-cache. Because the L1 D-cache is on a different TCM, it takes a little less than a half cycle for the address to get to it, and the same time for the data to get back. For that reason, the L1 D-cache operates on a clock which is displaced by about a half cycle from the clock that controls the decoding and execution logic. The data are read from the cache during cycles 2–3, and are available for use by the GXE during cycle 4. Cycle 4 is the execution cycle, during which the addition takes place. During cycle 5, the result is written into the GR array, and the finished report is sent to the completion controls.

Instruction B is an RR ADD instruction, and does not use an operand from storage. The operands it uses are independent of those used by instruction A, and there is nothing to prevent it from decoding on cycle 1 also. On cycle 3 it executes, and on cycle 4 it writes its result to the GR array and creates a finished report. It does not execute on cycle 2, because that cycle is required for it to get priority in the GXE queue and to have its operands read out of the GR array. This is an example of two instructions decoding on the same cycle, of two instructions going to the GXE queue at the same time, and of out-of-sequence execution.

Instruction C is an RX LOAD ADDRESS instruction. It has its address calculated in the I-ACE and written from there into the GR array. It decodes on cycle 2 and goes to the I-ACE queue. If the required GR values were available, it would be able to have its address calculated on cycle 2 also; however, one of its operands is GR 5, whose value is being calculated by instruction B. Instruction B executes on cycle 3, and its result can go directly from the GXE output to the I-ACE; therefore, the value can be used in an address calculation on cycle 4 at the same time the operand is being written into the GR array. Following the address calculation, there is a one-cycle delay before the result can be written into the GR array, which takes place on cycle 6.

Instruction D is an SI AND IMMEDIATE, which sets the high-order four bits of the byte at location ADDR2 to 0s. It can decode on cycle 2 along with instruction C, and also has its address calculation performed on that cycle in the D-ACE. A request to ADDR2 on cycles 3–4 serves to fetch the data, to test whether a store access to the location is possible, and to assign a store queue and store buffer position. The instruction executes on the store side of the GXE on cycle 5, and on cycle 6 the value is written into the appropriate GXE store buffer and a finished report is created.

Instruction E is an RR SUBTRACT instruction. On cycle 3 it decodes, on cycle 5 it executes, and on cycle 6 it writes its result into the GR array and creates a finished report. It should be noted that one of its operands, GR 4, is generated by instruction A on cycle 4. This is an example of the result of an instruction being gated from the output of the GXE right back to its input so that it can be used by another instruction while it is being written into the GR array.

Instruction F is an RX LOAD instruction. On cycle 3 it decodes and has its address calculated, and on cycles 4–5 its operand is fetched from the L1 D-cache. Because no operation is required on the data, and because there is a data path from the L1 D-cache directly into the GR array, on cycle 6 the operand is written into the GR array and a finished report is created. The instruction never goes to the GXE queue and never takes a GXE execution cycle.

It is worth noting that on cycle 6 three different values are written into the GR array. This is possible because they come from three different sources, and each of those sources has an independent write capability into the GR array. The operand for instruction C comes from the I-ACE, that for instruction E comes from the GXE, and that for instruction F comes from the L1 D-cache.

Also on cycle 6, four different finished reports are created. This is possible because each one comes from a different source which has an independent capability to generate finished reports. Instruction C is reported finished by the I-ACE, instruction D by the store side of the GXE, instruction E by the GR side of the GXE, and instruction F by the L1 D-cache.

Instruction G is an RR ADD instruction. On cycle 4 it decodes, on cycle 6 it executes, and on cycle 7 it writes its result to the GR array and creates a finished report. It should be noted that one of its operands is the value loaded by instruction F, and that instruction G is able to make use of the value as soon as it comes from the L1 D-cache.

Summary

The ES/9000 520-based models reflect a number of advances in processor organization. They implement a two-level cache to better buffer the cycle time difference between the CPU and central storage, with one level associated primarily with central storage and the other with the CPU. The cache associated with the CPU is split into two separate caches which allow two pieces of data to be obtained each cycle. The CPU employs an organization which involves a substantially greater degree of parallel operation than its predecessors. Different instructions can be executing at the same time in different parts of the CPU, and instructions can execute out of sequence. This requires a substantially more complex control structure than has been used previously, and the use of a virtual register management algorithm which breaks the relationship between an architected register and a specific piece of hardware.

Acknowledgments
In any large project, the final result is the sum of the contributions of many people, and it is impossible to recognize everyone who has made a contribution. Nevertheless, the author would like to note that the following people made significant contributions to the portion of the design described in this paper: Henry Brandt, Neal Christensen, Leo Clark, Steven Comfort, Stanley Dobrzynski, Jun Fong, Patrick Gannon, Clifford Hayden, Hung Le, Donald McCauley, Jay Murdock, Thomas Rathje, Steven Risch, James Shelly, and Charles Webb. Some of these individuals also made contributions to this paper by reviewing early drafts of the manuscript.

References
1. R. M. Tomasulo, "The IBM System/360 Model 91: An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Develop.* **11**, No. 1, 25–33 (1967).
2. J. S. Liptay, "Structural Aspects of the System/360 Model 85: II. The Cache," *IBM Syst. J.* **7**, No. 1, 15–21 (1968).
3. S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Syst. J.* **25**, No. 1, 4–19 (1986).

Received February 6, 1991; accepted for publication September 18, 1991

John S. Liptay *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (LIPTAY at PK705VMG).* Mr. Liptay received his B.E.E. degree in 1962 and his M.E.E. degree in 1966, both from Rensselaer Polytechnic Institute. He joined IBM in 1965 at the Thomas J. Watson Research Center, transferring shortly thereafter to the IBM Product Development Laboratory in Poughkeepsie, where he has remained since. Mr. Liptay has worked on the System/360 Models 65 and 85, the System/370™ Model 168, the 3033, and the high-end ES/9000 processor, all in the area of CPU design. He has received three IBM Invention Achievement Awards, an IBM Outstanding Contribution Award, an IBM Division Award, an IBM Technical Excellence Award, and an IBM Outstanding Innovation Award, the last for his work on the virtual register management algorithm described in this paper. Mr. Liptay is an IBM Senior Technical Staff Member and a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, the Institute of Electrical and Electronics Engineers, and the Association for Computing Machinery.