POWER2 instruction cache unit

by J. I. Barreh
R. T. Golla
L. B. Arimilli
P. J. Jordan

This paper describes the instruction cache unit (ICU) of the IBM POWER2™ processor, with emphasis on improvements over the original POWER ICU design. The POWER2 ICU incorporates a new compare-branch scheme that minimizes processing time penalties, a second branch processor, increased branch look-ahead capability, and doubled instruction-fetch and instruction-dispatch bandwidth.

Introduction

IBM introduced the POWER-based RISC System/6000® (RS/6000) workstation in February of 1990. This system was well received in the industry and helped IBM capture a sizable share of the workstation market. The POWER2™ processor goals were to build on the strengths of the original POWER design and to overcome its shortcomings.

The POWER and POWER2 systems partition instruction processing across three units: the instruction cache unit (ICU), the fixed-point unit (FXU), and the floating-point unit (FPU). This paper describes the overall organization of the POWER2 ICU, as well as the improvements over the previous design. **Figure 1** is a block diagram of the POWER2 ICU. The primary functions of the ICU are the following:

- Fetch all instructions.
- Execute branch and logic on condition register instructions.
- Dispatch instructions to the FXU and FPU.
- Process interrupts.
- Maintain the architected condition, count, and link registers.
- Maintain interrupt control registers.
- Provide engineering support processor (ESP) functions.

To improve performance over that of the existing POWER-based RS/6000 systems, the ICU designers focused on enhancing branch and superscalar performance while minimizing the cache miss penalty. The POWER2 ICU design addressed these challenges by

- Designing a new compare-branch scheme.
- Adding a second branch processor to allow processing of two branches in a single cycle.
- Increasing branch look-ahead capability.
- Doubling the instruction fetch bandwidth both from cache and from memory.
- Doubling the instruction buffers.
- Enabling instruction cache accesses during cache miss processing.
- Doubling the dispatch bandwidth to the other functional units.

Architected registers

The ICU maintains the architected registers involved in processing branch instructions and interrupts. User-accessible registers include the condition register (CR), the link register (LR), and the count register (CTR). The other registers include the machine state register (MSR), the save restore registers, and the segment registers.

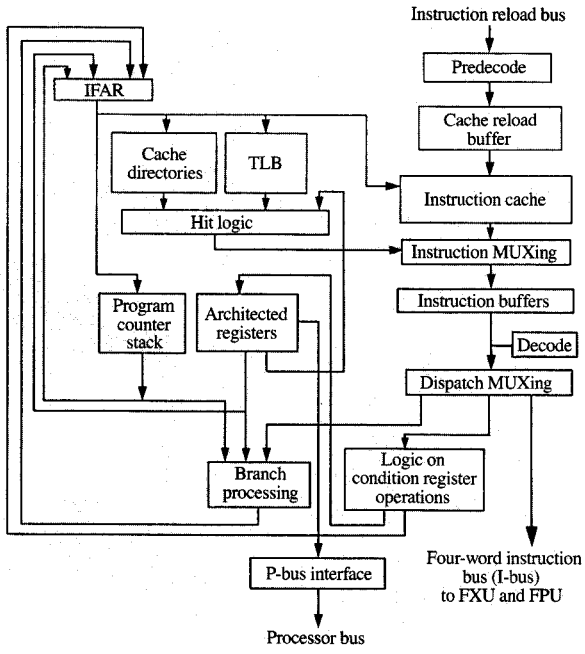

Figure 1

ICU internal logical partitioning.

• *Condition register (CR)*

The CR consists of eight 4-bit fields. The ICU tests these fields to determine the outcome of conditional branch instructions. Many events can modify the CR bits. Compare instructions can alter any field. Most register-to-register instructions include a record bit; setting the bit causes an update to a predefined CR field. Move to condition register (MCR) instructions can alter any combination of the eight fields. Logic on condition register (LCR) instructions alter a single CR bit. Finally, the ICU can restore a previous CR value if an interrupt occurs.

The CR interlock logic provides a lock bit for each of the eight CR fields. Lock bits are set (reserving the corresponding fields) when the ICU dispatches CR setting instructions to the FXU or FPU; the lock bits are reset upon instruction completion. The ICU dispatches or holds instructions that read or update a CR field based on the setting of the lock bit associated with the source or target field.

• *Link register (LR)*

Subroutine calls use the LR extensively. A subroutine call usually translates into a branch instruction with its link bit set; the link bit specifies that the ICU must save the address of the next sequential instruction (the return address) in the LR. A special form of the branch instruction, which uses the contents of the LR as the target address, supports returns from subroutine calls. Because the entire 32-bit value becomes the target address, this instruction also allows branching beyond the limitations of the normal branch displacement fields.

• *Count register (CTR)*

The 32-bit CTR serves multiple purposes. As its name implies, this register can hold the iteration count for a loop. A form of the branch conditional instruction decrements the count and tests the resulting value; the outcome affects whether the conditional branch is taken. Second, like the LR, the CTR also can supply the target address for long branches. Finally, across supervisor call boundaries, the save and restore process for the MSR involves the CTR.

• *Machine state register (MSR)*

Bits in the MSR enable or disable processor features such as external (and other) interrupt enables, privileged instruction access, FPU access, interrupt addressing, and instruction and data relocation.

• *Save restore registers*

Save restore register 0 saves the return address on interrupts. Save restore register 1 saves the contents of the MSR and the interrupt status on interrupts; it restores the MSR during a return from interrupt instruction.

• *Segment registers*

The FXU maintains the sixteen segment registers which define the current effective-to-virtual address translation. The FXU broadcasts any segment register changes using the processor bus (P-bus). The ICU monitors the P-bus and maintains a shadow copy of these registers to improve the performance for instruction address translation.

• *Program counter stack (PCS)*

To allow precise interrupts, the ICU maintains multiple copies of the link, count, and condition registers. The program counter stack (PCS) retains up to four backup copies of each of these registers and restores their contents if required because of an interrupt. When the ICU dispatches an interrupt-causing (IC) instruction, the PCS logic flags the instruction. If the ICU dispatches an ICU instruction that modifies any of these three registers after the flagged IC instruction, the ICU backs up that particular register on its corresponding backup stack. In general, only the first ICU instruction to modify one of these registers after an IC instruction requires a backup of that register; additional operations on that register do not. If an IC instruction does cause an interrupt, the ICU uses these

backup stacks to restore the registers to their state prior to the IC instruction, and then the ICU removes the backup copies.

The ability to restore snapshots of these registers using the backup copies allows the ICU to execute past the interrupt-causing instructions. However, the FXU and FPU do not execute past an instruction that causes an interrupt; therefore, any of their instructions which modify these ICU registers do not require register backup copies (because they will not have to back out of any changes). If the FXU (or FPU) finishes executing a marked IC instruction successfully and the backup is no longer needed, the backup copy is removed.

Instruction fetching

Most computers execute programs in a manner which provides the same results as if all instructions executed one at a time. Early computers included a program counter that pointed to the current instruction. In concept, all instructions prior to the current instruction were complete, and all instructions following the current instruction had yet to start.

In many of today's high-performance systems, especially in superscalar designs, the high degree of instruction-level parallelism blurs the concept of a program counter, possibly to the point of nonexistence. However, in a processor which supports precise interrupts, an effective program counter usually exists for interrupt processing; the interrupt processing mechanism receives a single current address; prior instructions are complete, and following instructions have not begun.

However, aside from interrupts, each pipeline stage in a superscalar processor effectively has its own view of the current instruction. In the POWER2 ICU, two hardware registers maintain the program counter equivalents for the fetch and dispatch stages. The instruction fetch address register (IFAR) points to the current instruction being fetched along an expected, or guessed, instruction stream path. The instruction address register (IAR) points to the current instructions being dispatched. The ICU usually increments the IAR each cycle by the number of dispatched instructions.

On the basis of the IAR contents and branch instructions which logically follow the current dispatch instruction, the branch unit determines the expected instruction stream path based on a guess algorithm. (For POWER and POWER2, the guess algorithm for unresolved, conditional branches is not taken.) The ICU adjusts the IFAR each cycle to prefetch instructions which the dispatch stage might need. By prefetching instructions along an expected path, the ICU overlaps instruction cache accesses with useful work. Usually the IFAR is many instructions ahead of where the IAR is currently pointing.

To fetch instructions, the ICU must generate addresses, translate addresses, and access the instruction cache. The instruction fetch mechanism attempts to fetch eight instructions per cycle, including fetch requests which span cache-line boundaries. The fetch logic places the instructions in one of the two instruction buffers (target or sequential), as space allows. Later sections of this paper describe these buffers and the subsequent dispatch of instructions.

- *Fetch address generation*

The fetch logic prefetches instructions along an expected, or guessed, instruction stream path. The fetch logic attempts to have at least six instructions in the sequential instruction buffer at all times to facilitate multiple instruction dispatch. The ICU normally increments the IFAR by the number of instructions stored in the instruction buffer by the fetch logic; this normal operation occurs while sequential code is being executed. If there is a branch, the ICU may load the IFAR with the target address of the branch. The ICU treats interrupts and returns from interrupts as special cases of branches; the interrupt logic provides the target address for an interrupt, and save restore register 0 provides the address for a return from interrupt instruction.

- *Address translation*

The ICU must translate fetch addresses in order to access the cache arrays and cache directories. The address translation logic provides a mechanism for translating the effective address of an instruction to a virtual address and then to a real memory address. (If translation is not enabled, the real address is simply the effective address.) When a translation cannot complete because the instruction translation lookaside buffer (I-TLB) does not contain the necessary translation information, the ICU uses the P-bus to request that the FXU complete the translation. The FXU translates the virtual address and returns the real address on the P-bus. The ICU uses the translation to update an entry in the TLB. The ICU does not request a translation until the ICU is sure that the dispatch stage will require the associated instruction. For a more detailed description of POWER2 address translation, see the paper by Shippy and Griffith in this issue [1].

- *Translation lookaside buffer (TLB)*

This small cache provides a mechanism for improving the speed of translating a virtual address to a real page number. This two-way set-associative table contains 64 entries (congruence classes) in each set. Each two-word entry contains the segment ID, bits 4 through 13 of the virtual address, the real page number, page protection bits, a valid bit, and parity bits. The TLB replacement policy is least recently used (LRU).

Instruction cache

The 32KB instruction cache is a two-way set-associative design with 128 lines (congruence classes) per set; each line is 128 bytes, or 32 instructions. The cache allows the fetching of eight words in parallel, even when the access crosses cache-line boundaries, as long as the instructions are in the cache and the fetch does not cross a page boundary. The cache implements an LRU replacement algorithm.

In the POWER implementation, the instruction cache is not accessible while a cache line is being filled from memory. The POWER2 design alleviates this problem by adding a cache reload buffer (CRB). This ICU CRB is similar to the CRB which is present in the POWER and POWER2 data cache units; it holds a single cache line of information, and it permits access to the cache during a cache-line reload. When a cache request is made, the cache logic searches both the cache directories and the CRB.

- *Cache directories*

Logically, for each cache request, the cache logic searches a directory to determine whether the data are present and, if present, where the data reside. POWER2 physically implements two distinct cache directories—one for even cache lines and one for odd cache lines—as did the original POWER implementation [2]. This allows access to directory entries for two adjacent lines (odd and even) during one cycle, enabling the prefetching of instructions across cache-line boundaries.

A cache directory contains one entry (or tag) per cache line; a one-to-one mapping exists between a cache directory tag and its associated cache line in the cache storage array. The cache directory entry consists of the 20-bit real page number, a valid bit, and parity bits. Since the cache is a two-way set-associative design, the cache logic compares the 20 high-order address bits of a cache request with two directory tags. A match signals a cache hit; the directory tag that matches dictates which cache line contains the required data.

- *Cache arrays*

The cache arrays provide local storage for instructions fetched from main storage (and for some instruction predecode bits which are described later). The 32KB cache consists of 16 cache arrays of 2 KB each. Eight arrays supply data for even cache lines; the other eight supply data for odd cache lines. The organization of the arrays is such that any eight sequential addresses in a cache line access one word from each of the eight associated arrays.

Addressing for the eight storage arrays for the even cache lines is independent of the addressing for the eight arrays for odd cache lines. By addressing one group of arrays using the IFAR contents and addressing the other group with the address of the first word in the next cache line, a fetch request can access one word from each of the 16 arrays. As a result, the ICU can fetch eight sequential instructions (within the same page) every cycle, including eight-word blocks that span cache-line boundaries.

- *Cache miss processing*

When the fetch logic requests an instruction that is not in the cache or CRB, the ICU uses the P-bus to request a cache-line reload from the storage control unit (SCU). The CRB address register (IFAR_R) holds the address of the cache line currently in the CRB. At the beginning of a cache miss, the cache logic copies the CRB contents (which were filled during the previous cache miss) into the cache. The IFAR_R addresses the cache and directories as the cache receives the copy of the line from the CRB. After the CRB contents have been emptied into the cache, the ICU copies the current cache miss address from the IFAR into the IFAR_R in preparation for the reload data. This also frees the IFAR to access the cache, if necessary, during the cache miss sequence once the initial reload data return. The cache miss address in the IFAR_R routes the reload data into the proper CRB locations.

As instructions come into the CRB, they are also forwarded (or bypassed) to the fetch unit. The IFAR keeps track of the reload data and adjusts the address by the number of instructions bypassed so that instruction fetching can continue from the point where bypass has stopped. Instruction bypass can be stopped by a cache-line wrap, full instruction buffers, a machine check, an SVC-type instruction, or a branch which changes the sequential instruction stream.

During a cache-line reload, as instructions arrive on the instruction reload bus, the ICU partially decodes the instructions. The predecode logic creates a 6-bit tag which identifies invalid opcodes, FXU instructions, FPU instructions, instructions that affect cache miss processing, and ICU instructions. The ICU stores these predecode tags with the associated instruction in the instruction cache.

Instruction buffers

The instruction buffers hold instructions prefetched from the cache by the prefetch mechanism (IFAR and associated logic). These buffers supply up to six instructions within a single cycle for dispatch and allow inspection of two subsequent instructions for use in branch prefetching. Sixteen sequential instruction buffer entries hold prefetched instructions from the sequential stream; eight target instruction buffer entries hold instructions from a guessed branch taken path. Each buffer entry contains 40 bits of instruction, predecode tag, and parity information.

When the ICU encounters an unresolved conditional branch, the fetch logic uses one instruction buffer for

sequential instruction stream fetching and the other buffer for fetching instructions along a branch path. If the branch is not taken, the ICU discards instructions fetched along the branch path, the sequential buffer supplies instructions to the dispatch stage, and instruction fetching along the sequential path resumes. When a branch is taken, the ICU dumps the target buffer into the sequential stream buffer; dispatch continues along this new sequential stream.

The instruction buffer logic controls instruction buffer updates and supplies instructions to the decode and dispatch multiplexers. The information necessary to manage these buffers includes the following:

- Which branches are taken.
- When synchronizing operations must occur.
- How many instructions are being fetched from the cache.
- How many instructions are being dispatched.
- How many instructions are in the instruction buffer.
- Where the next instruction for dispatch is in the instruction buffer.
- How much space exists in the FXU and FPU instruction buffers.
- Whether special-purpose register contention exists between instructions.

The ICU can steer any sequential or target buffer entry to the first dispatch port and the entries immediately following to the remaining dispatch ports.

Instruction dispatch

The ICU dispatches instructions from either the sequential or target instruction buffers, depending on whether the ICU is in sequential dispatch mode or target dispatch mode. In sequential dispatch mode, the ICU fully decodes six instructions and inspects two more for branches, assuming the sequential buffers have valid instructions. Full decoding identifies the instruction type from the instruction tags, opcodes, and extended opcodes. If the seventh or eighth instruction is a branch, and the target address is available, the ICU can prefetch the target. The ICU can dispatch up to six instructions from the sequential buffers: four FXU/FPU instructions and two ICU instructions (either two branches or one branch and one LCR instruction). The dispatch logic evaluates all six instructions in parallel.

In target dispatch mode, the ICU can fully decode and dispatch up to four instructions, none of which may be ICU instructions. In some cases, the fetch logic can prefetch branches in target buffers. To determine whether the ICU can dispatch a particular instruction, the dispatch logic examines all conditions and interlocks established by previous instructions, up to and including the given instruction. The ICU routes instructions that meet the conditions for dispatch to the CR control logic, the branch processors, or the four-word instruction bus.

The ICU conditionally dispatches instructions following unresolved conditional branches. When the ICU or FXU resolves the branch, either the FXU and FPU execute the conditionally dispatched instructions (if the branch is not taken), or the ICU or FXU cancels the instructions (if the branch is taken). The ICU can dispatch only one conditional instructional stream. The ICU cannot dispatch a second unresolved branch nor the instructions that follow.

- *Branch processing*

The ICU routes branches in the dispatch window to the branch processors. The ICU has two independent branch processors; therefore, it can handle up to two branches per cycle. When the POWER2 ICU encounters two branch instructions in the current dispatch cycle, one of the following cases is true:

- *First branch is taken* — No processing of the second branch is necessary.
- *First branch is unresolved* — No further branch processing is possible that cycle.
- *First branch is not taken and second branch is not taken* — The ICU treats both branches as NO OPs and the branches cause no lost cycles.
- *First branch is not taken and the second branch is taken* — The ICU generates the target address of the second branch using the second branch processor.
- *First branch is not taken and second branch is unresolved* — Same as the preceding case.

When the ICU encounters unresolved branches, it sends a branch packet to the FXU. The packet specifies the condition register bit on which the branch depends, as well as whether the branch should be taken if the condition is true. The ICU dispatches instructions past the unresolved branch and marks them as conditional. When the ICU has conditionally dispatched four instructions, or when the sequential stream becomes interlocked for any reason other than FXU/FPU lack of buffer space, the ICU switches to target dispatch. The ICU marks these instructions as target instructions, and holds them on the instruction bus (I-bus) until the branch is resolved.

If the FXU resolves the branch, the FXU notifies the ICU of the outcome of the branch. Similarly, if the ICU decides the branch, the ICU notifies the FXU. The ICU resolves the branch if the ICU or FPU computes the dependent CR value (or if the ICU receives the CR change on the same cycle in which the packet is sent, a boundary case). Depending on whether the branch is taken, the FXU and FPU either execute the conditional instruction stream

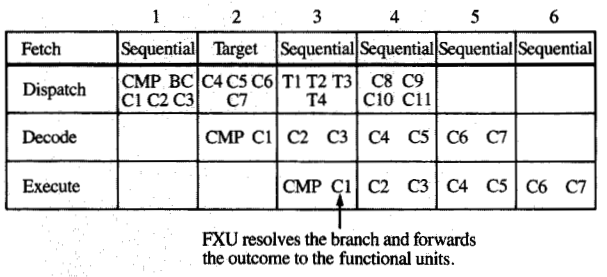

Figure 2

POWER2 timing: Branch not taken.

or latch the target instructions from the I-bus. **Figures 2** and **3** illustrate the timing characteristics for the taken and not-taken cases for the following code:

```
CMP
BC
C1 ──────┐
C2       │
C3       T1
C4       T2
C5       T3
C6       T4
C7       T5
```

Figure 2 shows that there is no penalty for a correctly guessed (not taken) conditional branch. As Figure 3 shows, the POWER2 implementation requires only one idle pipeline cycle after a mispredicted (taken) branch. The POWER implementation incurred as much as a three-cycle penalty. By fetching the target path and placing the target instructions on the I-bus, the POWER2 design eliminates two of these guessed wrong branch penalty cycles.

Interrupt processing

Two categories of interrupts exist. Some interrupts (for example, divide by zero, storage protection exceptions, or invalid operation) occur as the result of an instruction. In contrast, asynchronous interrupts (for example, system reset, machine check, external, and floating-point imprecise) occur with no predictable relation to the instructions being executed.

As a result of the high degree of instruction-level parallelism, POWER2 systems may recognize interrupt conditions earlier than a sequential machine. For all recoverable interrupts, when the ICU services an interrupt, the machine state requires a single *current* address at which execution is to resume upon the return from the interrupt. Instructions prior to this current instruction are complete; future instructions have not begun. When an instruction causes an interrupt, the offending instruction becomes the current instruction when the ICU services the interrupt. Therefore, the ICU cannot recognize an interrupt associated with an instruction until all previous instructions have completed, thereby ensuring that the previous instructions will not later cause interrupts.

Furthermore, hardware must suppress an interrupt condition if it occurs solely as an artifact of the superscalar implementation. For example, consider an instruction storage interrupt which might occur when instructions are prefetched. If the fetched instruction is on a guessed instruction stream (beyond an unresolved conditional branch), or if a previously fetched branch instruction has not yet been executed, a possibility exists that this interrupt-causing fetch would not occur in a sequential machine. Therefore, the instruction storage interrupt is not valid until the offending instruction is the next instruction to execute.

In the case of multiple interrupts, the hardware must select a single interrupt to service. The architecture defines the system reset interrupt to be the highest-priority interrupt, so that it will reset the machine even when other interrupts are pending. In POWER2, the system reset and machine check interrupts can occur at any time. Because of the serially reusable nature of the save and restore registers, the ICU recognizes all other interrupts sequentially. A predefined priority list orders these interrupts.

Because the ICU cannot recognize an interrupt associated with an instruction until all preceding instructions have completed, the interrupt ordering gives priority to instructions that are earlier in the instruction stream (and therefore further through the pipelines). In the list that follows, notice that, among interrupts caused by an instruction, the ICU gives preference to interrupt conditions that do not occur until the execute stage over interrupt conditions that are detectable in the dispatch stage.

Although the ICU can select the current instruction in an arbitrary manner for the asynchronous floating-point imprecise and external interrupts, all previous instructions must have completed when the ICU services the interrupt. Therefore, the asynchronous floating-point imprecise and external interrupts must wait until all previously initiated instructions complete to ensure that outstanding operations cannot produce an interrupt.

Some types of interrupts are known to be mutually exclusive; because they cannot be present simultaneously, the priority list does not differentiate among them. The list groups them together and assigns them a common priority level. The POWER2 interrupt priority ordering is as follows:

1. System reset.
2. Machine check.
3. Program counter interrupt—execution. These interrupts are mutually exclusive and assume a priority based on the instruction sequence. They are trace, trap, alignment, data storage, and floating-point enabled exception.
4. Instruction dispatch interrupts—dispatch. These interrupts are mutually exclusive and have no priority order. They are the program interrupts (privilege, invalid operation), supervisor call, and floating-point unavailable.
5. Floating-point imprecise interrupt.
6. External interrupt.
7. Instruction storage interrupt—fetch.

• *Interrupt state machine*

A state machine validates and prioritizes interrupts. The machine starts in a "no interrupts pending" state. Occurrences of interrupt conditions (requests or signals) cause transitions based on the conditions and their respective priorities. Once a pending state is entered, the ICU accepts and processes the interrupt, or a higher-priority interrupt becomes pending (overriding the lower-priority interrupt), or the interrupt condition is cleared (canceling the pending interrupt).

• *Interrupt process execution*

When the ICU accepts an interrupt, the ICU saves the state and transfers control to the interrupt handler software. Saving the state involves storing the current MSR and the address at which the interrupt occurred (or the address of the next instruction if the interrupt is asynchronous). For some interrupts, the ICU stores the cause of the interrupt. The interrupt state machine returns to the "no interrupts pending" state. The MSR and the type of interrupt determine the interrupt vector to which the machine branches. The ICU alters the MSR according to type of interrupt.

P-bus interface

The ICU, the FXU, and the SCU use the P-bus to communicate. P-bus interface logic monitors the P-bus to detect commands that the ICU must execute or co-execute with the FXU. It also generates and receives service requests from the other chips on the P-bus. These services include address translation, segment register update, special-purpose register access, cache operations (for example, cache-line invalidate or cache-line flush), TLB invalidates, or cache-line reloads. The P-bus state machine controls the P-bus interface on the ICU. Because the P-bus state machine reloads the I-cache and I-TLB, it also controls the instruction buffers, the IFAR, the cache arrays and directories, and the TLB.

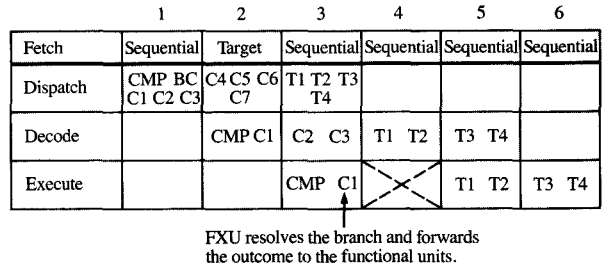

Figure 3

POWER2 timing: Branch taken.

Summary

The POWER2 ICU builds on the strengths of the POWER ICU and improves important functions such as branch processing. It doubles the instruction reload bandwidth, instruction fetch bandwidth, and instruction dispatch to FXU and FPU. The design adds a second branch processor to facilitate the execution of two branches in one cycle. It also implements a new scheme to improve compare-branch code sequences.

Acknowledgments

The logic designers of the ICU would like to thank Greg Grohoski and Charles Moore for their contribution to the definition of the POWER2 ICU. We would also like to thank John Groot, who did the physical layout and wiring of the chip, and Pat Pena, who handled the overall simulation of the ICU.

References

1. D. J. Shippy and T. W. Griffith, "POWER2 Fixed-Point, Data Cache, and Storage Control Units," *IBM J. Res. Develop.* **38,** No. 5, 503–524 (September 1994, this issue).
2. G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore, "Branch and Fixed-Point Instruction Execution Units," *IBM RISC System/6000 Technology*, Order No. SA23-2619, IBM Corporation, 1990, pp. 24–32; available through IBM branch offices.

Received September 3, 1993; revised manuscript received June 20, 1994; accepted for publication June 21, 1994

Jama I. Barreh *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (JAMA at AUSVM6).*
Mr. Barreh received B.S. and M.S. degrees in electrical engineering from the Illinois Institute of Technology in 1985 and 1987, respectively. In 1987 he joined IBM, initially working on the RISC System/6000 processor. He also worked on the instruction cache unit of the POWER2 processor. He is currently a Staff Engineer working on a memory controller for the PowerPC 620™. Mr. Barreh has received an IBM First Invention Achievement Award.

Robert T. Golla *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (GOLLA at AUSVM6).*
Mr. Golla received his B.S. degree in electrical engineering from the University of Houston in 1986, and his M.S. degree, also in electrical engineering, from the University of Texas at Austin in 1990. He joined IBM in 1986 as a logic design engineer, and initially worked on the RISC System/6000 microprocessor. From 1989 to 1992, he acted as team leader on the instruction cache chip for the POWER2 microprocessor. Currently, Mr. Golla is a Staff Engineer working on the PowerPC 603™ microprocessor. He has received an IBM First Invention Achievement Award.

L. Baba Arimilli *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (BABA at AUSVM6).*
Mr. Arimilli received an M.S. degree in electrical engineering from Louisiana State University in 1986. He joined Texas Instruments' Semiconductor Division in 1986 in Houston, where he worked on VRAM design. In 1987 he joined IBM in Austin, where he currently works on high-end RISC processor designs. Mr. Arimilli has received an IBM Invention Achievement Award.

Paul J. Jordan *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (PJORDAN at AUSVM6).*
Mr. Jordan graduated cum laude from Rice University in May 1990 with a B.S. degree in electrical engineering. On joining IBM, he worked as an Associate Engineer on the instruction cache unit design team for the POWER2 processor during 1991 and 1992. He is currently a Senior Associate Engineer working in PowerPC™ processor development.