POWER2 floating-point unit: Architecture and implementation

by T. N. Hicks
R. E. Fry
P. E. Harvey

The POWER2™ floating-point unit (FPU) extends the concept of the innovative multiply–add fused (MAF) ALU of the RISC System/6000® processor to provide a floating-point unit that sets new standards, not only for computation capability but for data throughput and processor flexibility. The POWER2 FPU achieves a performance (MFLOPS) rate never accomplished before by a personal workstation machine by 1) integrating dual generic MAF ALUs, 2) doubling the instruction bandwidth and quadrupling the data bandwidth over that of the POWER FPU, 3) adding support for additional functions, and 4) using dynamic instruction scheduling techniques to maximize instruction-level parallelism not only among its own internal units but with the rest of the CPU.

Introduction

The original version of the RISC System/6000® (RS/6000) floating-point unit (FPU) set a new standard for floating-point performance. Its innovative *multiply–add fused* (MAF) dataflow minimizes latency, rounding error, and chip busing [1]. The MAF unit performs a double-precision multiply in a single cycle and a double-precision add in the following cycle. A single round occurs in the final and bypassable stage of the pipeline. The FPU combines, in a single two-stage pipeline, capabilities which many other processors, such as the SuperSPARC Microprocessor [2], provide with two units, usually a separate multiplier and adder. The simultaneous use of multiple execution units requires additional data buses as well as control logic for detecting dependencies across units. The architecture supports the exploitation of the MAF capability through a set of multiply–add instructions. The RS/6000 processor support of these instructions allows execution of a dependent pair of operations with a combined latency of only two cycles. This feature is unique in the industry.

The POWER2™ FPU design goal is to build upon these strong points to provide a FPU that sets new standards not only for computation capability but also for data throughput and processor flexibility. The POWER2 FPU achieves an MFLOPS rate never accomplished before by a personal workstation machine [3] by

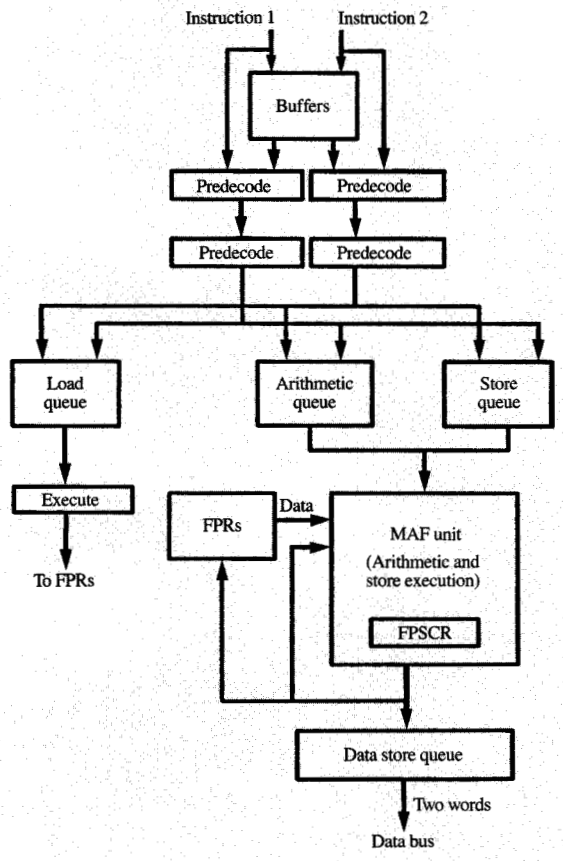

Figure 1

Block diagram of RS/6000 FPU.

- Integrating dual generic MAF ALUs.
- Doubling the instruction bandwidth and quadrupling the data bandwidth over that of the RS/6000 FPU.
- Adding support for additional functions.
- Using dynamic instruction scheduling techniques [4] to maximize instruction-level parallelism, not only between its own internal units but also across the rest of the CPU.

System perspective

Floating-point computation had a very revolutionary role in the evolution of computer processing. First, in the early systems, fixed-point arithmetic was used to perform numerical computation. The necessity for a floating-point representation grew from the dynamic range limitations and portability concerns associated with the various fixed-point work lengths available in the industry. Integer emulation of floating-point numbers became standard.

Second, as silicon became cheaper, it became practical to dedicate hardware to the task of floating-point computation. This dedicated hardware could perform the standard arithmetic operations in significantly less time than the integer processor, which was customized for its own specific tasks. The first attempts involved a coprocessing element which was fed instructions once the core processor determined that the instructions were floating-point operations. In early versions, the FPU and the fixed-point unit (FXU) could not run simultaneously.

The third evolutionary step was incorporating this dedicated hardware into the rest of the CPU in a way which maximized floating-point performance and minimized processor overhead. As an example, processors such as the Intel 8087 coarsely overlapped floating-point and nonfloating-point operations. As floating-point capabilities increased, migration of floating-point-dominated applications further accelerated the demand for more advances. Integrating the floating-point processor with the rest of the CPU became imperative.

Various methods were used in attempting to integrate these units [5]. The RS/6000 processor achieved much of its floating-point performance by tightly coupling the FPU to the rest of the CPU, particularly the FXU. Although this design point significantly advanced the state of the art in floating-point computation, the POWER2 FPU has since taken a further step by removing interlocks and increasing the autonomy of the multiple functional units.

RS/6000 FPU overview

Figure 1 shows a block diagram of the RS/6000 FPU. The FPU receives two instructions from the instruction cache unit (ICU). These two instructions go through a predecode stage in which the FPU discards nonfloating-point instructions, followed by a register-renaming stage [6]. Register renaming allows hardware to remove any read-before-write or write-before-write conflicts between arithmetic and subsequent load operations. Register renaming, along with the pending store queue buffer, greatly increases the potential for the FXU and FPU to operate independently. The rename stage forwards the two instructions to the execution unit responsible for that class of instruction. The load unit receives load operations, while the MAF execution unit receives both arithmetic and store operations.

This MAF unit performs all of the floating-point arithmetic instructions, such as the multiply–add fused operation, as well as all floating-point store operations. All internal data representations use the IEEE [7] double-precision format (with an extended exponent field).

Dual unit motivation for POWER2

Three factors determine the time required by a processor to complete a program [8]:

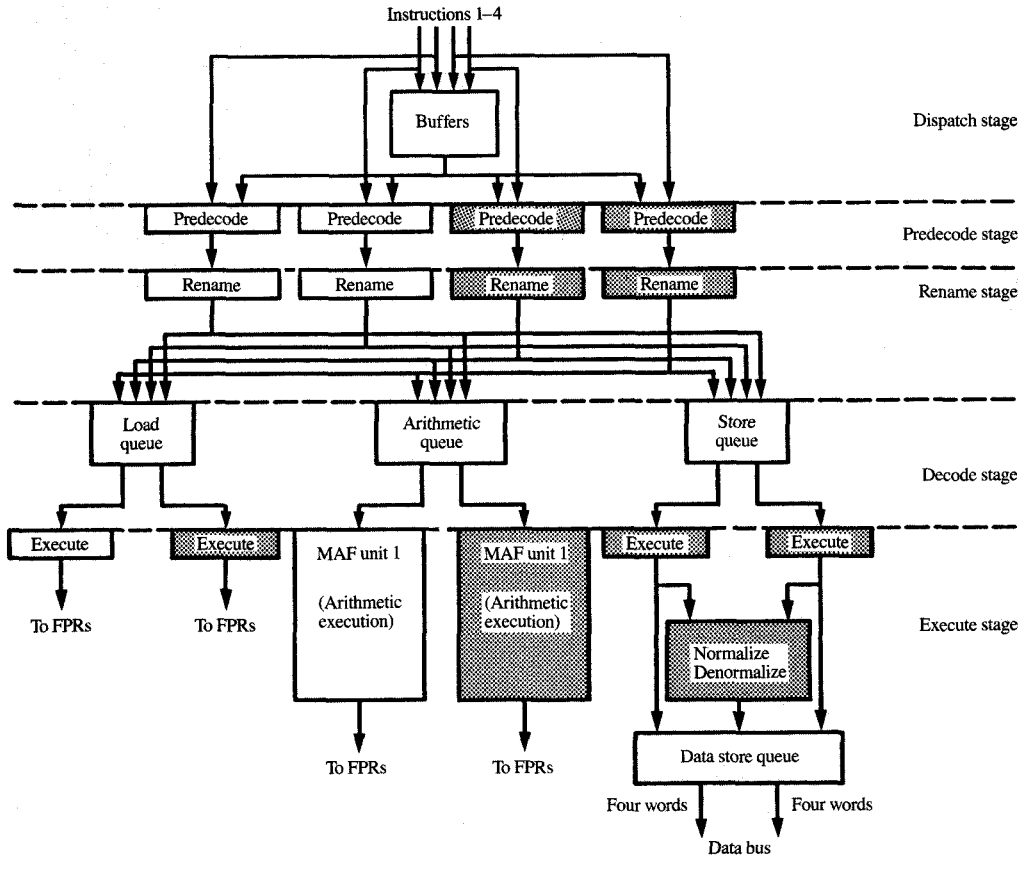

Figure 2

Block diagram of POWER2 FPU.

- The number of instructions required to execute the program.
- The processor cycle time.
- The average number of processor cycles required to execute an instruction.

Compiler capabilities determine the total number of instructions required for a given program. The second and third factors are under the CPU designers' control. The POWER2 FPU targets both factors. To decrease the cycle time, the POWER2 processor employs 0.5-μm CMOS technology. This process allows processor clock rates that are more than twice that of the initial versions of the RS/6000 processor. In decreasing the average number of cycles required to execute an instruction, one can either decrease the latency of the execution unit or add more execution units. Given a two-cycle latency for dependent multiply–add instructions, decreasing the latency for a single FPU instruction is unlikely. However, increasing instruction-level parallelism to decrease the average time for execution of a group of instructions is viable. POWER2 achieves this by doubling the number of floating-point execution units. A fundamental challenge confronting the POWER2 FPU design team was how to feed both units simultaneously to achieve maximum performance.

The POWER2 FPU

Figure 2 is a block diagram of the POWER2 FPU. The figure highlights the functional units that were not present in the RS/6000 FPU design. The FPU receives four instructions from the ICU, double that of the RS/6000. It pipes them through the predecode and register-rename stages, both twice the width of the analogous RS/6000 stages. Once past the rename stage, the instructions split up into one of three types of execution units: a load execution unit, an arithmetic execution unit, and a store

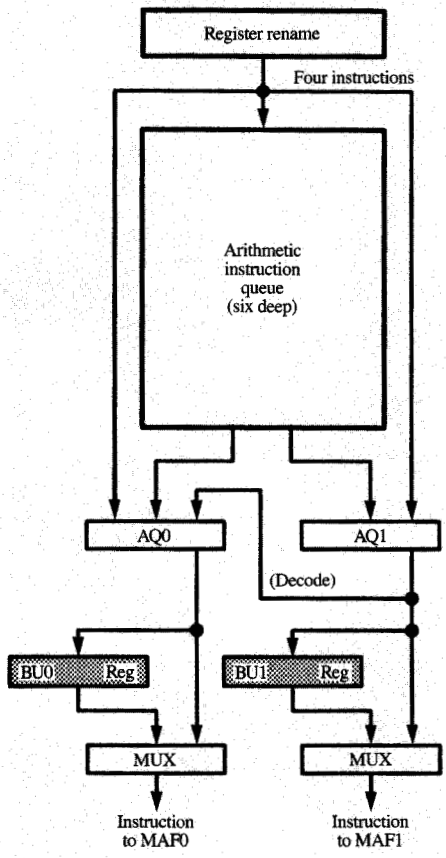

Figure 3

Arithmetic instruction queue.

execution unit. Each one of these units is actually a plurality of units; two instructions of each class can execute simultaneously. The arithmetic pipeline can start two arithmetic operations simultaneously, the store pipeline can perform two store operations simultaneously, and the load pipeline can perform two load operations simultaneously.

Each of these units is capable of executing independently of the others, barring data dependencies. However, the FXU, which does all address calculations, can perform only two loads/stores at any one time (in any combination: ld/ld, st/st, ld/st, or st/ld), because it has only two arithmetic logic units (ALUs).

Unlike the RS/6000 processor, the POWER2 FXU and FPU receive only the instructions that they will execute, except for interruptible instructions. After the instruction stream has passed through the ICU, only two types of instructions can cause an interrupt: memory-accessing instructions (loads and stores) and fixed-point trap instructions. POWER2 systems perform synchronization of the FXU and FPU units at these interruptible boundaries. Each unit may "get ahead" of the other (with respect to the instruction stream) until it encounters a potentially interrupting instruction. At this point, synchronization must occur. If the FPU encounters the interruptible operation (IOP) first, it waits until the FXU performs the address calculation (or executes the trap) and signals the FPU that execution can continue. If the FXU encounters the IOP first, the FXU informs the FPU that instructions can continue past the IOP. Once the FPU reaches the IOP, it ignores its interruptible capability (having already been cleared for execution) and execution continues without any stalls.

This synchronization scheme allows each unit to operate at its peak performance regardless of stalls within the other unit. For example, the FXU can proceed to execute register-to-register and load/store operations even when the FPU is working on multicycle arithmetic operations, such as divide or square root.

The POWER2 FPU targets five areas for improving floating-point performance:

- Maximize parallelism between dual arithmetic units.
- Maximize parallelism between arithmetic and store operations.
- Quadruple data bandwidth and process cache hits under miss.
- Double instruction bandwidth.
- Migrate operations to hardware.

Maximizing parallelism between dual arithmetic units

One of the limiting factors in issuing multiple instructions in a superscalar processor is the number of floating-point pipelines. As previously mentioned, the POWER2 FPU contains two generic multiply–add ALUs, each of which is capable of executing all floating-point register-to-register operations. The two primary concerns for avoiding bottlenecks are how to avoid unit interlock when both units are fed from a common queue, and how to minimize delays for unit-to-unit dependencies.

While one MAF unit is consumed for many cycles because of multicycle operations, such as divide or square-root instructions, the instruction queue should be able to feed the other unit with independent instructions. POWER2 allows this by providing a *backup* register above each of the execution units. This register provides sufficient instruction buffering and pathways to allow one unit to continue while the other unit processes a multicycle operation.

This register also allows one dependent operation without stalling the other pipe. For example, consider the following code:

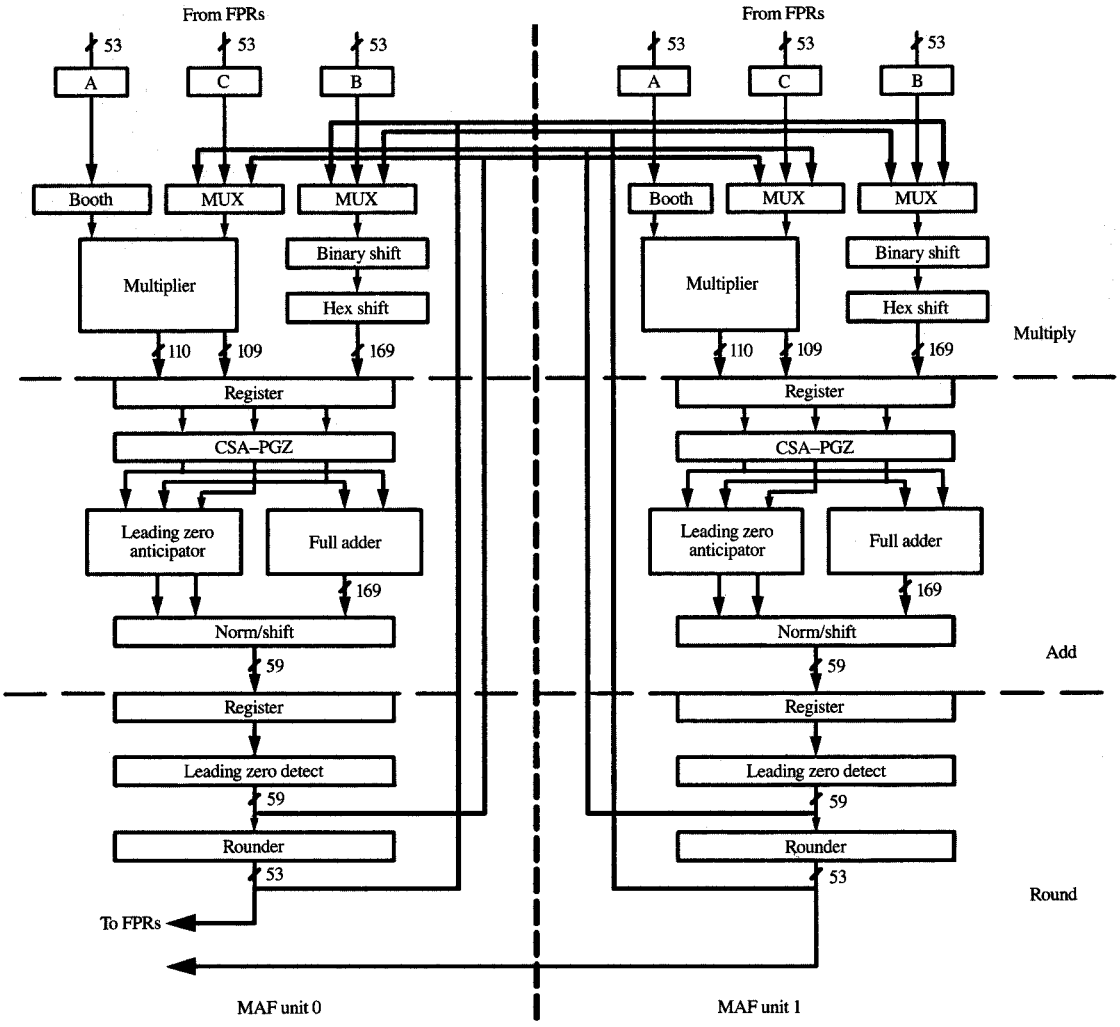

Figure 4

Execution unit bypasses.

fd **3** ← 1/2

fa 5 ← **3** + 4

fma 6 ← (1*2) + 4

Since the divide instruction is held in the backup register (and the dependent add is held in decode for MAF0), the independent multiply–add instruction (fma), and any following instructions, may execute in MAF1. **Figure 3** shows the instruction queue that feeds both units and also highlights the backup registers.

Bypass buses between the two units minimize unit-to-unit dependencies. Instead of requiring that the result data be written into the floating-point registers (FPRs) prior to subsequent use, the RS/6000 FPU has the capability to bypass the registers and pass the resultant data (leaving the IEEE rounder) directly into the add operand and (from before the rounder, which was required to make the setup time for the multiplier) directly into the multiplier. Both POWER2 units can bypass in the same manner (that is, no cycle delay is required for passing through the register file). **Figure 4** highlights these bypasses.

When an instruction is in the arithmetic instruction queue preparing for execution, the logic compares the operand registers of the instruction with the target of the instructions already in execution to detect dependencies. If

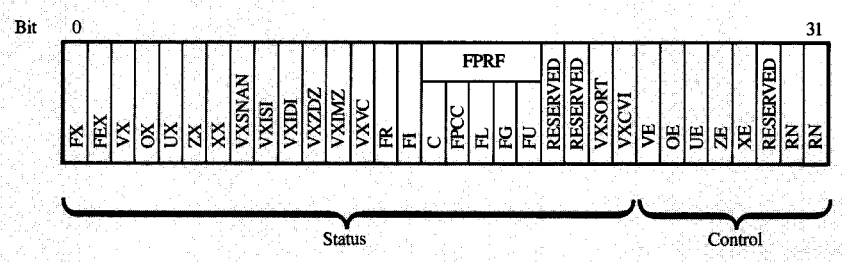

Figure 5

FPU status and control register.

the logic detects a dependency, the multiplexers select the appropriate bypass bus. To utilize these bypass buses, the result must be a normalized number. Zero, infinity, NANs, and unnormalized and denormalized numbers cannot use the data bypasses.

One floating-point architected register, the FPSCR (floating-point status and control register), controls rounding, dictates exception handling, and records exception status of executed arithmetic instructions. When an arithmetic instruction is in the last cycle of execution, the FPU reads the rounding mode (bits that specify one of four IEEE standard [7] rounding modes) from the FPSCR and updates status flags such as overflow, underflow, and inexact. The FPSCR holds the rounding mode, floating-point trap enables, condition codes, and IEEE exception status. A detailed description can be found in the *AIX® Version 3.2 for RISC System/6000 Assembler Language Reference Manual* [10]. The FPSCR register is shown in **Figure 5**.

Arithmetic operations may complete differently from the sequential ordering of the program; an obstacle to optimizing arithmetic performance is maintaining the integrity of the FPSCR register when this occurs.

The POWER2 FPU, like its predecessor, does not support precise floating-point interrupts within the hardware itself. All exceptions are recorded within the FPSCR register, and execution status is checked by polling this register and trapping on a software comparison of the exception flags. As in the RS/6000 architecture, a mechanism does exist within the instruction unit to put the system into a state in which precise floating-point interrupts are possible (the setting of the FE bit within the MSR register). However, this severely degrades performance and is only recommended for software debugging. In POWER2, hardware support for a floating-point imprecise interrupt was also added in order to aid in software debugging. The enabled exception summary bit within the FPSCR register was tied into the external interrupt line within the instruction unit. The external interrupt handler is then responsible for polling the FPSCR register to determine whether the interrupt was due to a floating-point exception.

Since the FPU handles all data adjustments due to exceptions "on the fly," the polling of the FPSCR register is a viable method of handling exceptions in a superscalar machine. However, the sequential appearance of the FPSCR register must be maintained, despite the fact that the arithmetic operations whose status is being recorded may complete out of order. When the programmer polls the FPSCR, it must reflect the state of the FPU at that instant as if all instructions were executing sequentially. Also, at the execution time of each arithmetic operation, the control portion of the FPSCR (such as the rounding mode) must reflect that of sequential execution.

The RS/6000 floating-point arithmetic instruction set can be divided into two classes of FPSCR operations:

- Type1 instructions report the progress of their own execution to the FPSCR and depend on the control portion in order to execute properly [e.g., a floating-point add instruction (FA) which reports the occurrence of an overflow during execution (UX bit) and whose results depend upon the IEEE underflow trap enable flag (UE bit) set in the control portion].
- Type2 instructions are instructions whose target is the FPSCR itself [e.g., a move to or from status registers instructions (MTFSF and MFFS, respectively]. These instructions are used to save and restore the state of the FPU upon task switching or to poll the FPSCR for exception checking.

The FPSCR register is divided into a sticky field and a nonsticky field. The bits within the sticky field have the characteristic that once they have been set (switched to an active state), they stay set until a type2 instruction resets them. Bits within the nonsticky field are evaluated and allowed to switch every cycle.

Before explaining the mechanism used, one observation must be made: At the time of execution of a type2 instruction, all operations sequentially preceding it must be complete (or guaranteed to complete before the type2 instruction). This is to guarantee that if the type2 operation is to move (read) the status/control register, it will be reading the sticky status of ALL the operations before it, and the nonsticky status of the single instruction preceding the type2 operation. Therefore, the first rule of this mechanism is that execution synchronization occurs on all operations of type 2.

Once this is accomplished, sequential instruction execution such as this:

$type1_1\ type1_2\ type1_3\ type1_4\ type2_1\ type1_5\ type1_6\ type1_7\ type2_2$

can be viewed as this:

$$\boxed{\begin{matrix} type1_1 \\ type1_2 \\ type1_3 \\ type1_4 \end{matrix}}\ type2_1\ \boxed{\begin{matrix} type1_5 \\ type1_6 \\ type1_7 \end{matrix}}\ type2_2,$$

where out-of-order optimization can be performed within the type1 instruction boxes.

As mentioned previously, the FPSCR register is divided into a sticky field and a nonsticky field. Maintaining the sticky field is not a problem, since order of execution is not important to a sticky bit (once it is set, it stays set). However, when the type2 instruction executes, the nonsticky field must reflect the execution of the previous instruction. In the above example, when instruction $type2_1$ executes, the nonsticky field must be updated as a result of instruction $type1_4$.

The mechanism consists of two apparatuses, a state machine pointer and a tagging mechanism. The state machine pointer is used to keep track of which execution unit has the most recent type1 (sequential) instruction. The state machine monitors the dispatch control unit (since this is the last place serialism is preserved) in order to track the order of execution. The tagging mechanism is used to tag the instruction that comes before a type2 instruction. When a type2 instruction is ready to be dispatched, the state machine pointer identifies the execution unit that has the instruction preceding it. The instruction in the earliest pipeline stage of that execution unit is then tagged. When an instruction identified by the tag pointer finishes execution, it is allowed to update the FPSCR. If the tag pointer does not identify any instruction that has finished execution, then the instruction identified by the state machine pointer is allowed to update the FPSCR. Therefore, if the type2 instruction is not fetched for several cycles (and therefore no tag has been set), it is still guaranteed that the instruction preceding this type2 instruction will be reflected in the FPSCR register. Since execution synchronization occurs on all type2 instructions, it is guaranteed that they finish alone; therefore, type2 instructions override the state machine pointer. **Figure 6** shows the mechanism. Notice that the sticky bits are just ORed together regardless of execution order, while the nonsticky bits are selected by a multiplexor controlled by the state machine pointer.

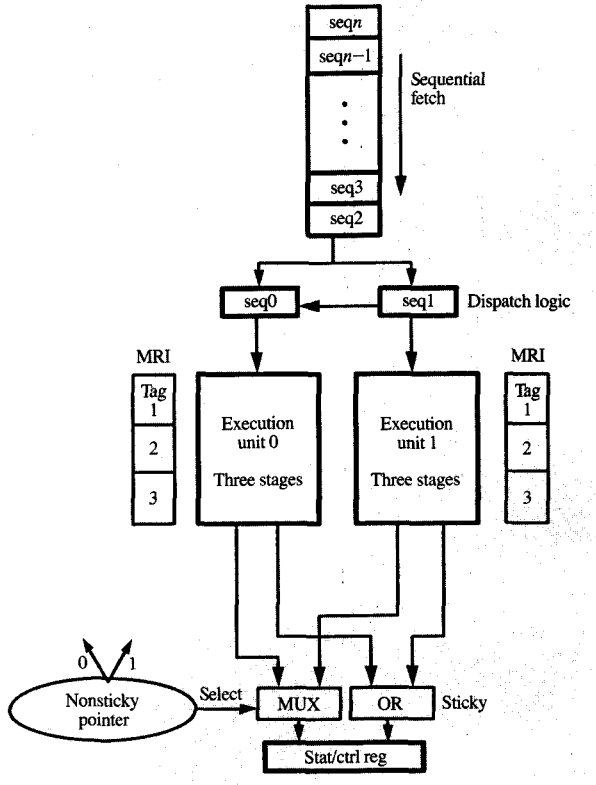

Figure 6

FPSCR register mechanism.

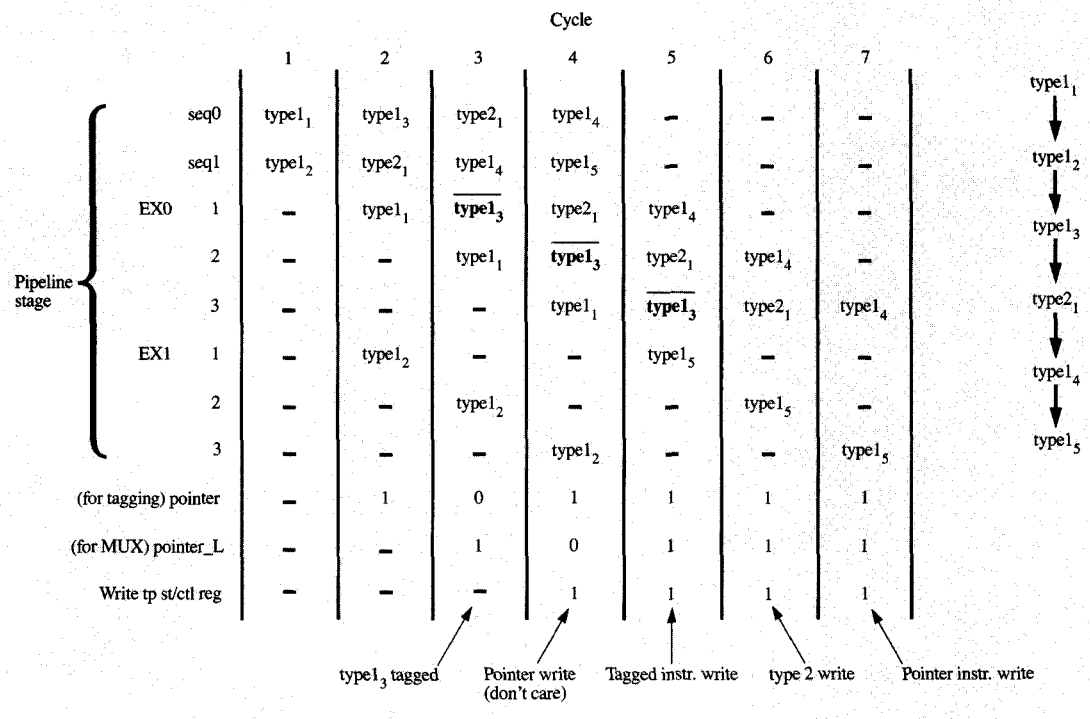

Figure 7

Instruction execution timing diagram.

Figure 7 illustrates the execution of the following stream of code:

$type1_1$, $type1_2$, $type1_3$, $type2_1$, $type1_4$, $type1_5$

1. In cycle one, the first two instructions are in sequential buffers 0 and 1, ready to execute.
2. In cycle two, the first two instructions begin execution in execution units 0 and 1, respectively. The third and fourth instructions ($type1_3$ and $type2_1$) move into sequential buffers 0 and 1. Also in cycle two, the pointer is pointing at the $type1_2$ instruction, since it is the most recent in execution.
3. In cycle three, the $type1_3$ instruction begins to execute. Since synchronization must occur on all type2 instructions, $type2_1$ cannot begin executing in cycle three. Therefore, it is shifted into sequential buffer 0, allowing $type1_4$ to enter sequential buffer 1. Also in cycle three, since the next instruction is type2, tagging is done. The instruction highest in the execution unit to which the pointer is pointing (in this case execution unit 1) is tagged ($type1_3$ above).
4. In cycle four, the type2 instruction begins execution. Again, $type1_4$ cannot start with the type2 instruction. Therefore, it is shifted into sequential buffer 0, allowing $type1_5$ to move into sequential buffer 1. Also in cycle four, the first two instructions are completing execution. Since the pointer is pointing to execution unit 0, $type1_1$ has access to update the nonsticky portion of the FPSCR register, even though it is not the next sequential instruction ($type1_2$ is). This anomaly is irrelevant because there is another instruction following (to which the pointer is actually pointing). When it ($type1_3$) completes, the status/control register will be corrected.
5. In cycle five, the last two instructions begin execution simultaneously. The tagged instruction ($type1_3$) is in the final stage of execution, and even though the pointer is not pointing to it, it writes to the nonsticky FPSCR register, because of the tag.

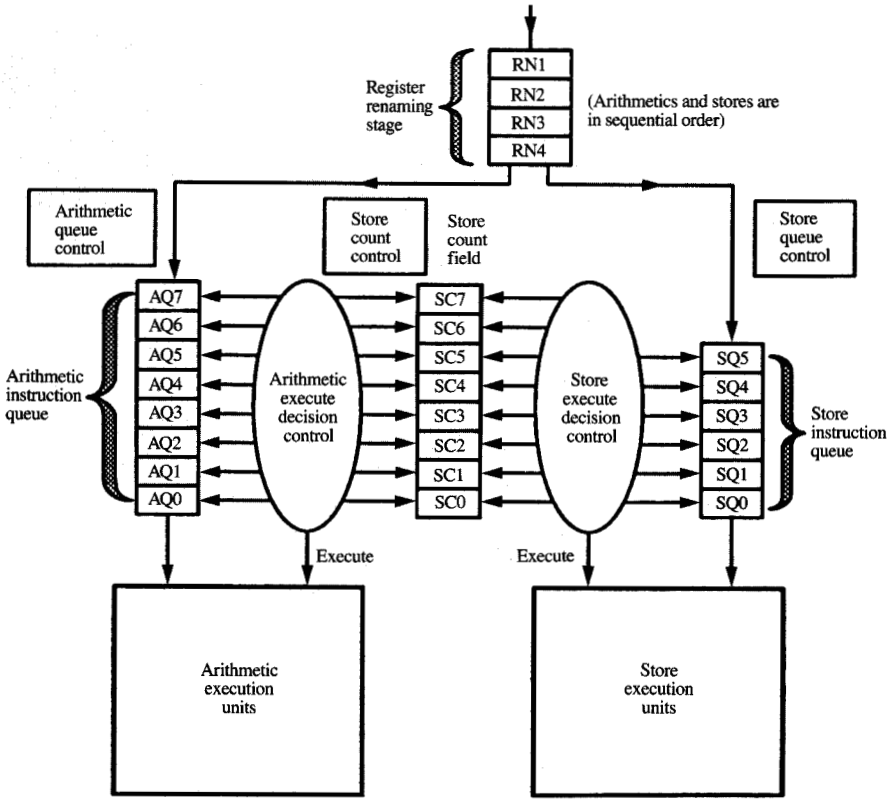

Figure 8

Instruction queues and store count fields.

6. In cycles six and seven, the last two instructions complete execution. Since there are no more instructions, the pointer simply follows $type1_5$ down through the pipeline stages, allowing it to write the nonsticky FPSCR register in cycle seven.

Maximizing parallelism between arithmetic and store operations

In POWER, floating-point arithmetic and store operations compete for the single MAF resource. Normalization and denormalization of store data require the MAF unit. The POWER2 FPU separates the store and arithmetic execution units. This allows arithmetic operations to operate in parallel with independent store operations. With this implementation, synchronizing the two units to detect true data dependencies while allowing out-of-sequence execution is a challenge. The solution utilizes two mechanisms: a means of preserving precedence and a means of detecting dependencies.

Unlike some other RISC implementations [9], the RISC System/6000 implementation allows instructions to proceed all the way through the pipeline to the execution stage before checking operand availability. This increases the opportunity for instruction-level parallelism. However, the FPU must preserve the serialization of dependent arithmetic and store instructions. The POWER2 design point requires a mechanism to optimize parallel independent execution while detecting and serializing dependent execution.

The FPU includes a serializing mechanism, called the "store count (STC) field," associated with the instructions in the arithmetic queue. This field indicates the number of store instructions in the store queue that are logically ahead of a particular arithmetic instruction. A bank of comparators compare the target operand of the arithmetic

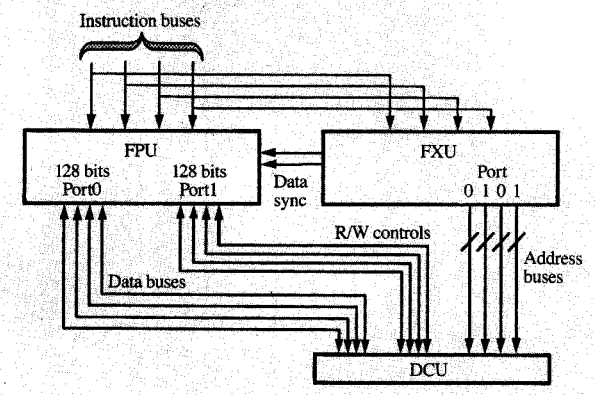

Figure 9

Floating-point buses and control structure.

instruction (in AQ0) with the source operands of the instructions in the store queue. This information is then combined with the information from the STC to determine whether the pipeline can execute the arithmetic. The FPU uses this same mechanism to decide when store instructions can begin execution. **Figure 8** illustrates the STC mechanism.

The figure shows the renaming stage of the instruction pipeline, at which point instructions are in the sequential order received from the ICU. Four instructions at a time move from the rename stage into their respective queues (arithmetic or store). As an arithmetic instruction moves into the arithmetic queue (AQ), the serializing mechanism adds the count of all store instructions in the rename stage ahead of the arithmetic instruction to the count of instructions currently in the store queue (SQ), then subtracts the number of store instructions starting execution. The serializing mechanism places the count in the store count field associated with the arithmetic instruction in the AQ. The STC data move down the queue as the arithmetic instruction moves down through the AQ. The FPU decrements each store count entry in the queue each time a store instruction executes.

When the STC is zero, there are no store instructions which could cause a dependency ahead of that arithmetic. The pipelines can execute arithmetic instructions that have an STC of zero immediately, regardless of store instruction comparisons. When the STC is nonzero, the FPU must compare the arithmetic target register with the source operands of each store, up to and including the number of stores indicated by the STC. If none of these store operands match the arithmetic target, the arithmetic instruction can begin executing, even though these store instructions are lined up sequentially before the arithmetic. If the targets do match, the arithmetic instruction must wait until the store executes, or the operation will destroy the data being moved to memory by the store instruction.

A store (in SQ0) may execute immediately if all of the instructions in the AQ have a nonzero STC. This case indicates that the store came before all of the arithmetic instructions in the AQ. The FPU must compare the store operand with the target of all arithmetic instructions in the AQ that have an STC of zero. If the operands do not match, the store can execute, even though the store is lined up sequentially after each arithmetic instruction with an STC of zero. If the targets do match, a true data dependency exists, and the store must wait for the arithmetic instruction to generate the data to be stored in memory. The fact that POWER2 can execute two stores and two arithmetic operations per cycle further complicates this mechanism. However, this mechanism allows the POWER2 FPU to separate arithmetic and store execution units while requiring synchronization only on true data dependencies.

Quadrupling data bandwidth and processing cache hits under miss

Modern packaging technology permits chips with large pin counts. The POWER2 FPU has 506 signal I/Os. The designers allocated more than half of these to the dual quadword (16 bytes) data buses which move data to and from the data cache unit (DCU). Supporting dual quadword buses (compared to POWER's single doubleword bus) gives POWER2 a fourfold increase in data bandwidth. Storage-reference-limited codes, such as Linpack, often obtain a substantial performance boost from dual units, wide buses, and quadword storage instructions [3]. **Figure 9** shows the data and instruction bus organization. Synchronization between the FXU and the FPU during the execution of floating-point load operations is necessary to ensure that the data are routed to the correct FPRs and aligned properly. **Figure 10** shows the stages of execution for a floating-point load operation.

During the first two stages, the FXU generates and translates the address and forwards the request to the data cache. During stages three and four, the FXU loads a data register into the FPU and tells the FPU which port the data are on and how to align the data. The FPU then determines the load instruction to which the data correspond (depending on previous signals) and moves the data into the appropriate FPR (or FPRs in the case of a load quad instruction).

A key design point of the POWER2 implementation is the ability to continue processing and executing loads and stores through one fixed-point execution unit while there is a cache miss in progress (because of a load or store which

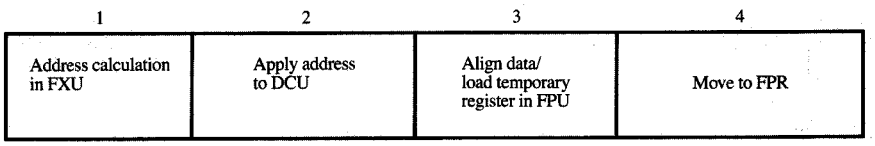

Figure 10

Floating-point load execution pipeline.

Cycle	1	2	3	4	5
Execution unit0	LFD0 (EA gen)	Cache miss	Cache miss	Cache miss	**DATA0**
Execution unit1	LFD1 (EA gen)	**DATA1** LFD2 (EA gen)	**DATA2** LFD3 (EA gen)	**DATA3**	

Figure 11

Overlapping load execution with data cache misses.

executed in the other execution unit). The two data ports to the FPU are *tied* to the two execution units in the FXU. Therefore, the order for the data ports is the same as that of the fixed-point execution units (that is, port 0 is the oldest and port 1 is the newest). However, depending on the availability of the data in the cache, the loading of data into the FPU can *complete* out of order. This means that the FPU may load data into the FPRs in a different order from the load instructions. For the sequence of instructions in **Figure 11**—which assumes a cache miss for LFD0 (load double)—LFD1, LFD2, and LFD3 complete before LFD0.

Migrating operations to hardware

Through RS/6000 application performance analyses, the FPU targeted two operations for enhancement. The POWER2 FPU hardware implements the square-root and convert-to-integer operations, which are both math library subroutine calls in POWER. The square-root function improves from a software implementation of roughly 53 cycles to a hardware implementation of 27 cycles. The square-root implementation uses a state machine controller for the MAF unit similar to the divide implementation.

The architecture adds two forms of convert-floating-point-to-integer instructions. One performs the rounding specified in the FPSCR control register; the other forces a truncation (round-to-zero mode) which provides support for the FORTRAN intrinsic INT. This eliminates the need for machine cycles associated with loading the FPSCR register when truncation is appropriate.

Status and interrupt support

A hardware mechanism in POWER2 maintains the integrity of the single FPSCR architected register despite simultaneous updates from the dual MAF units. The only instructions that force unit-to-unit synchronization are those specifically directed at the FPSCR registers, such as MTFSF (move to FPSCR). These instructions cause a stall in the MAF1 unit for one or two cycles.

The POWER2 FPU, like its predecessor, does not support precise floating-point interrupts within the hardware itself. The processor records all exceptions

within the FPSCR register and checks execution status by polling the register and trapping on a software comparison of the exception flags. As in the RS/6000 implementation, a mechanism does exist within the ICU to put the system into a state that will make precise floating-point interrupts possible (the setting of the FE bit within the MSR register). The processor implements this mode by restricting the ICU's issue rate so that only one floating-point instruction is outstanding. This severely degrades performance and is only recommended for software debugging. POWER2 adds hardware support for a floating-point imprecise interrupt to aid in software debug. This enables an "abort on error" (such as divide by zero) while the processor is running at full speed.

Summary

The POWER2 FPU advances the state of the art in superscalar floating-point architecture by doubling the number of execution units while supporting the industry's fastest dependent multiply–add operation. Its flexible integration with the rest of the CPU and dynamic instruction scheduling make its theoretical peak performance sustainable within loops. Its cache miss bypassing allows execution to continue during cache reloads. The status and exception recording mechanism helps avoid ALU-to-ALU interlocks. All of these improvements contribute to a peak, and often realizable, performance of four MFLOPS per MHz (286 MFLOPS at 71.5 MHz).

Acknowledgments

The POWER2 FPU development was a collaborative effort of many skilled people. Among them, the authors would like to thank Rudy Pirovitz, George Lerom, and Don Mehaffey for their managerial direction. The authors also thank Steve White and John Reysa for reviewing this manuscript and providing many useful suggestions.

References

1. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. Develop.* **34,** 61–62 (January 1990).
2. G. Blanck and S. Krueger, "The SuperSPARC Microprocessor," Technical White Paper, Sun Microsystems, Inc., Mountainview, CA, 1992.
3. S. W. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM J. Res. Develop.* **38,** 493–502 (1994, this issue).
4. J. E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," *Computer* **22,** 21–35 (July 1989).
5. Vojin G. Oklobdzija, "Issues in CPU–Coprocessor Communication and Synchronization," *Microprocess. & Microprogram.* **24,** 695–700 (1988).
6. G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 47–52 (January 1990).
7. *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, STD 754-1985, IEEE, New York, August 12, 1985.
8. Mike Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991, pp. 2–3.
9. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*, Digital Equipment Corp., Maynard, MA, April 29, 1992, pp. 2–15.
10. *AIX Version 3.2 for RISC System/6000 Assembler Language Reference Manual*, Order No. SA23-2197, 1992; available through IBM branch offices.

Received June 3, 1993; accepted for publication May 31, 1994

Troy N. Hicks *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (TROYHIX AT AUSVM6, troy@austin.ibm.com).* Mr. Hicks is currently a Staff Engineer in the RISC System/6000 Division of IBM in Austin. He received a B.S. in electrical engineering from the University of South Florida in 1986. In 1987 he joined the AWS division of IBM in Austin and has since worked in the field of floating-point design, contributing to the design of the RS/6000 and POWER2 FPUs. Mr. Hicks holds a IBM Invention Achievement Award and has been awarded several patents for floating-point and processor invention.

Richard E. Fry *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (RFRY AT AUSVM6).* Mr. Fry is currently an Advisory Engineer in the RISC System/6000 Division of IBM in Austin. He received his B.S. degree in electrical engineering from the University of Texas in 1976, after which he worked in the area of microprocessor emulation for Texas Instruments in Austin. Since he joined IBM in 1984, his field of expertise has been in floating-point design.

Paul E. Harvey *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (PHARVEY AT AUSVM6).* Mr. Harvey is currently a Senior Engineer in the RISC System/6000 Division of IBM in Austin. He received his B.S. in electrical engineering from Texas A & M University in 1981. He then joined the Motorola Semiconductor Group and microcoded the MC68881 floating-point coprocessor, later serving as technical team leader on the MC68882. Since joining IBM in 1987, he has contributed to the designs of the RS/6000 fixed-point and POWER2 floating-point chips.