IBM POWER6 accelerators: VMX and DFU

L. Eisen
J. W. Ward III
H.-W. Tast
N. Mäding
J. Leenstra
S. M. Mueller
C. Jacobi
J. Preiss
E. M. Schwarz
S. R. Carlough

The IBM POWER6™ microprocessor core includes two accelerators for increasing performance of specific workloads. The vector multimedia extension (VMX) provides a vector acceleration of graphic and scientific workloads. It provides single instructions that work on multiple data elements. The instructions separate a 128-bit vector into different components that are operated on concurrently. The decimal floating-point unit (DFU) provides acceleration of commercial workloads, more specifically, financial transactions. It provides a new number system that performs implicit rounding to decimal radix points, a feature essential to monetary transactions. The IBM POWER™ processor instruction set is substantially expanded with the addition of these two accelerators. The VMX architecture contains 176 instructions, while the DFU architecture adds 54 instructions to the base architecture. The IEEE 754R Binary Floating-Point Arithmetic Standard defines decimal floating-point formats, and the POWER6 processor—on which a substantial amount of area has been devoted to increasing performance of both scientific and commercial workloads—is the first commercial hardware implementation of this format.

Introduction: VMX unit

A major purpose of high-performance microprocessors is the manipulation of data through complex calculations. Current designs allow for a wide array of calculations on very large and very precise operands. While the current standard is 64 bits, many types of calculations do not require this high amount of precision and thus are unable to realize the full potential of the microprocessor design through conventional methods. Some examples include graphics (such as calculations made by computer-aided design tools), scientific visualizations, data encryption, real-time video processing, and calculations used in the evaluation of seismic data.

One alternative way to address this potential deficiency is to use a single-instruction multiple-data (SIMD) design. Instead of working on a single piece of data and offering a high degree of precision, SIMD designs offer data parallelism by performing the same operation on multiple, lower-precision operands at the same time. This provides multiple virtual pipelines where only a single pipeline exists physically.

The vector multimedia extension (VMX) instruction set—announced as AltiVec** [1]—is a SIMD extension of the IBM Power Architecture* technology. It was jointly developed by the IBM Corporation, Apple, Inc., and Freescale Semiconductor, Inc., formerly the Semiconductor Products Sector of Motorola. The AltiVec instruction set was designed to provide RISC (reduced instruction set computing) type of instructions that allow for rapid data manipulation. The AltiVec architecture operates on 128-bit vector registers that can represent four 32-bit single-precision floating-point or integer operands, eight 16-bit integers, or sixteen 8-bit integers.

The first IBM-designed processor to implement the AltiVec instruction set was the IBM PowerPC* 7400 [2]. The IBM POWER6* processor VMX unit is the latest incarnation of this vectorized design. The paramount challenge of the POWER6 processor design was its aggressive frequency target of 13 FO4 (fanout of 4-inverter delay). This high-frequency goal put stress on the logic design, layout, and fine-tuning of the final design. These difficulties are discussed in more detail in the following sections.

Unique to the POWER6 processor design was the introduction of a recovery unit (RU) to the processor core. The RU is used to maintain a shadow copy of the

0018-8646/07/$5.00 © 2007 IBM

current state of the core. If an error is detected within the core, execution in the core is halted, and the RU can be used to restore the core to a known good state before execution is resumed. To support this, the VMX unit had to present its results to the RU for storage as well as support the restoration of state in the event of a recovery.

The following sections give an overview of the POWER6 processor VMX implementation. Particular focus is given to the challenges encountered in implementing a high-frequency design and how those challenges were overcome. The POWER6 processor support of the RU is discussed as well.

VMX unit overview

The VMX unit is treated as a separate execution unit by the POWER6 processor core. The VMX inputs are driven by the instruction dispatch unit (IDU), which also feeds instructions to the binary and decimal floating-point units (BFU and DFU, respectively). Instructions are presented to the VMX unit in program order and must be retired in program order. The IDU tracks the order of instruction execution across execution units, and the VMX control logic is responsible for tracking the instructions within the unit.

The VMX unit contains seven distinct execution subunits that control the dataflow pipelines shown in **Figure 1**. They are two load subunits, one store subunit, and four execution subunits. The execution subunits are the arithmetic logic unit (ALU) simple unit (XS), the vector floating-point unit (VF), the complex unit (XC), and the permute unit (PM). Each load unit controls its own dataflow pipeline (two total). The remaining subunits are divided across two execution pipelines. The first pipeline consists of the XS, VF, and store units, and the second pipeline consists of the XC and PM units.

While each of the four dataflow pipes may contain a valid instruction on any given cycle, they are tracked in parallel to ensure that program-order execution is preserved. This also simplifies the tracking of dependencies between the pipes; if they were allowed to move independently, the degree of complexity would significantly increase.

The vector register file (VRF) contains a writeback port for each of the four dataflow pipes and is capable of writing from all four simultaneously on any given cycle. The POWER6 processor VMX uses two instances of the VRF: one per execution pipeline. In the VMX floorplan, the VRFs are placed close to their corresponding pipeline; this allows for a fast register file access. Loads and writes update both copies of the VRF.

Instruction flow control details

Instruction flow through the VMX unit begins at the top of the pipeline with the instruction and load data circular queues. Under optimal operation (no stalls within the VMX unit), the queues are bypassed and cause no delay for the instruction. Instructions are maintained in the queue until they are issued to their respective subunit.

The read pointer for the instruction and data queues is controlled by two primary mechanisms: stalls and rejects. Stalls are generated when a read-after-write (RAW) dependency is detected. If an instruction has a RAW dependency, it will be stalled the minimum number of cycles required to have the data available. The primary reason for rejects is write-after-write (WAW) dependencies.

The different reactions to the above dependencies are due to the stage of the VMX pipeline in which they are detected. Instructions are allowed to stall within the top four stages of the VMX pipeline; beyond that, they will proceed down the pipe unimpeded. RAW dependency detection is done early enough to catch the dependent instruction within one of the early stalled stages.

The loading on the stall generation was a primary concern in the unit design. To mitigate this loading as much as possible, a reject capability was added to handle cases that are not performance critical. Part of this load reduction was done by moving WAW dependency checking to a later pipeline stage. This detection is done beyond the stall pipeline boundary. To allow for this, a WAW-dependent instruction is invalidated from the instruction pipeline and is reread from the queue. This reject results in a seven-cycle delay in the instruction. The WAW–reject pairing was done because a compiler should be able to avoid WAW dependencies (without an intervening RAW dependency, or otherwise an intervening RAW dependency will generate a stall).

An instruction reject may also occur following an estimate-form floating-point instruction. These estimate instructions require additional pipeline stalls within the floating-point unit (FPU). Rather than further load the VMX stall detection logic, these instructions were fed through the reject path. In addition, any stall condition that would last for seven cycles or longer is fed through the reject detection instead, since rejecting the instruction does not result in a performance penalty.

Operands are presented to each of the execution subunits from one of five possible sources: from the VRF, bypassed from the load result bus, bypassed from the remote pipe, bypassed from another subunit on the local pipe, or bypassed from the result bus of the subunit. Operands that are read from the VRF have no dependencies so their results are available from the register file. The VRF write operation requires two cycles to complete.

Operands bypassed from the load result bus are used to cover the two-cycle window required to write the results into the VRF. Operands bypassed from the remote execution pipe cover the last cycle of the VRF write

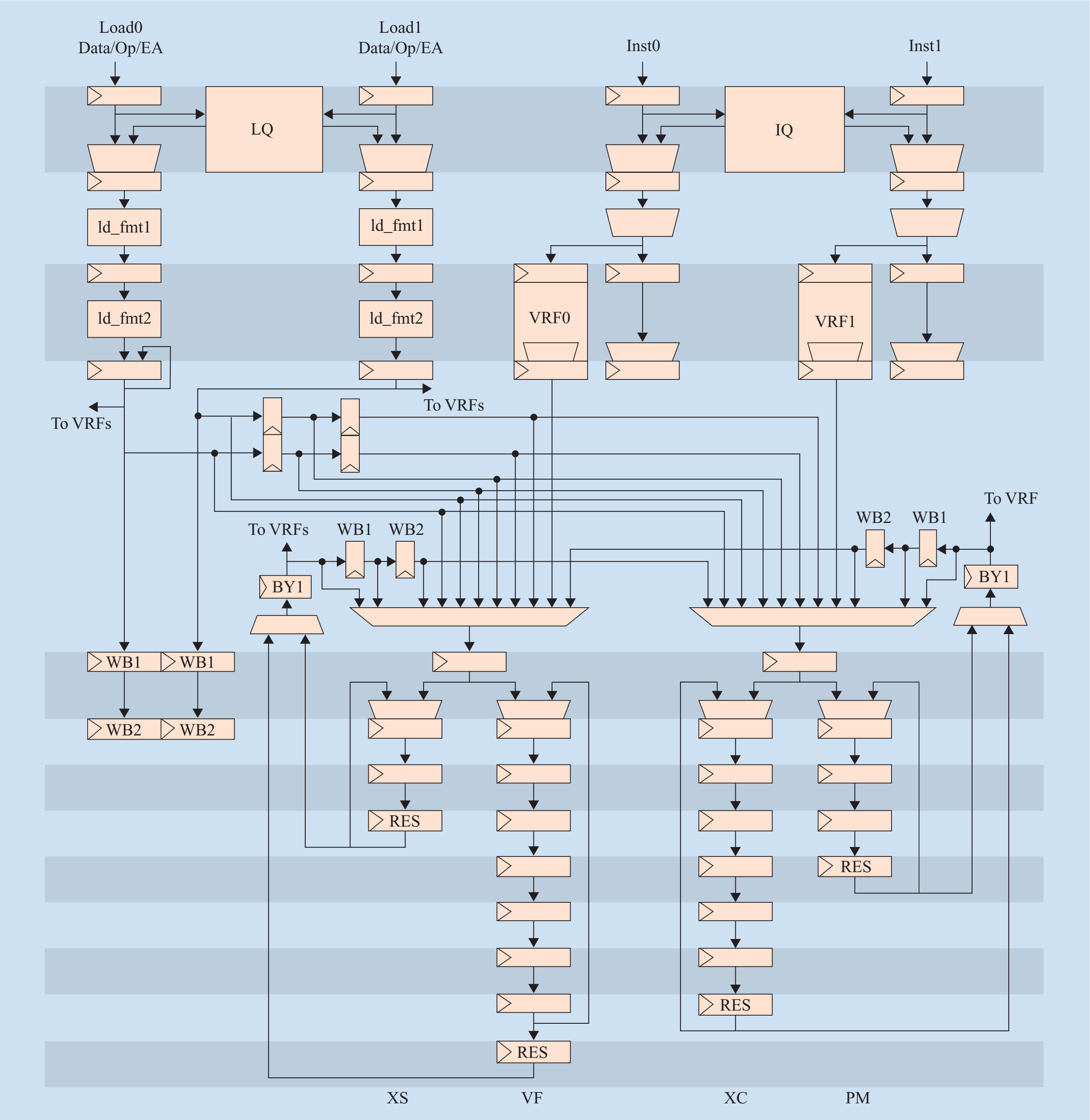


Figure 1

VMX pipeline. (IQ: instruction queue; LQ: load queue; BY1: first bypass stage; WB: writeback cycle; RES: result available. Execution cycles are distinguished by shaded and unshaded bars.)

window; floorplan restrictions prevent the data from being available sooner. Operands bypassed from the local execution pipe cover both cycles of the VRF write window.

Operands bypassed from the subunit result bus cover three to four cycles—the two-cycle VRF write window and the one (for XS, XC, and PM) or two (VF) cycles preceding. These additional cycles are gained because of the proximity between the subunit result bus and the operand macro.

The two execution pipes are grouped as follows: Pipe A contains the XS, VF, and store formatter, and pipe B contains the XC and PM. This selection was done for optimal instruction-type grouping and to have the two pipes mirror each other as closely as possible.

The final function of the control logic is to track the valid instructions as they pass through the execution subunits toward completion. This tracking is done to control the writeback ports of the VRF, to track dependencies as they flow through the pipes, and to signal valid VMX results to the RU.

Dependency tracking

Compromises in dependency detection proved to be one of the most difficult aspects of the POWER6 VMX design. Each VMX instruction can have up to three source operands.

Dependencies of an incoming instruction had to be done by comparing against 14 stages of each execution pipe (28 stages total) and four stages of each load pipe (8 stages total). Accounting for 36 pipeline stages × 3 source operands × 2 destinations for the incoming instructions required 216 comparisons each cycle for RAW dependencies.

WAW dependencies add another 68 comparisons. The two execution pipes must compare the incoming target register against the target registers of seven stages of each pipe ($2 \times 2 \times 7 = 28$ compares). The incoming load targets must be compared to ten stages of each pipe because loads write back to the VRF much earlier than the execution pipes (2 incoming load targets × 2 pipes × 10 stages = 40 compares).

Doing these compares on a single pipe cycle was impossible from a loading, timing, and floorplanning perspective. To mitigate this, compares were spread out across three pipeline stages, with the most critical compares done first.

The first comparisons done were to generate stalls for RAW dependencies. If a RAW dependency exists, the instruction will be stalled for the minimum number of cycles required until all of its operands are available.

The second set of compares was done to calculate the bypass controls for each operand. These bypass controls are calculated once; if the instruction stalls because of a RAW dependency, the bypass controls shift every cycle so they are aligned properly when the stall is released.

The third set of compares was done to calculate reject scenarios, such as WAW dependencies or RAW dependencies whose stalls are greater than the reject penalty. Instructions reach this final comparison only after they have passed the final stall stage. Thus, any WAW dependency that is detected is one without an intervening read (which would have stalled because of a RAW dependency). The compiler should not allow this to happen, so any performance penalty from this is not pertinent.

A reject causes the instruction to be invalidated in the execution pipe, and the instruction is then reread from the queue at the top of the VMX pipe. When the instruction exits the queue, it passes through the same comparison stages a second time. The reject penalty is seven cycles; if a WAW dependency still exists when the rejected instruction reaches the final comparison stage, it is rejected again. Because of the length of the VMX execution pipes, any given instruction can be rejected up to two times for WAW dependencies.

PM

The vector PM is 128 bits wide; therefore, only one instantiation of this entity exists. The vector PM performs instructions of the following types:

- Permute.
- Merge.
- Shift (by octet and by bit).
- Splat (repeat a part of the input operand).
- Pack (modulo and saturate).
- Unpack.

The permute, merge, shift, and splat instructions are used to efficiently replicate or align data of a storage operand after it is loaded into a register. The pack and unpack instructions compress and uncompress data.

The vector PM can be issued one instruction per cycle and has a back-to-back latency of four cycles. During the first and second stages of the pipeline, the operands and controls are modified and applied to the crossbar switch, which forms the third pipeline stage. The forwarding of the result to the input register is done in the fourth pipeline stage.

Crossbar switch

The central macro of the PM is the crossbar switch. All instructions performed in the vector PM deliver their results through it into the target register. The crossbar switch is built as 16 separate multiplexers (one per byte of the target register). Each one of these multiplexers can select any byte of a 32-byte input vector and deliver it to the target register.

The functionality of the crossbar is identical to the requirements of the permute instruction. Registers A and B (16 bytes each) deliver the 32-byte inputs to the crossbar switch, while register C (16 bytes wide, but only 5 bits per byte used) defines the permutation for the result value put into target register T. All other instructions executed by vector PM are mapped to the permute instruction. This is done by manipulating the operands in such a way that executing a permute in the crossbar delivers the correct result. The manipulation of the operands is done in the first two stages of vector PM.

Figure 2(a) depicts the classical layout of a permute crossbar. It shows how byte *n* of register T depends on byte *n* of register C. The five bits in register C select one byte of the 2 × 16 bytes in the source registers A and B.

Like the permute instruction, the instructions pack modulo, merge, and splat deliver permutations of the bytes in operands A and B. In contrast to the permute instruction, the instructions pack modulo, merge, and splat depend on the operation code (opcode) rather than operand C to control the crossbar switches. For these instructions, it is, therefore, sufficient to compute the proper register C value for the crossbar based on the instruction.

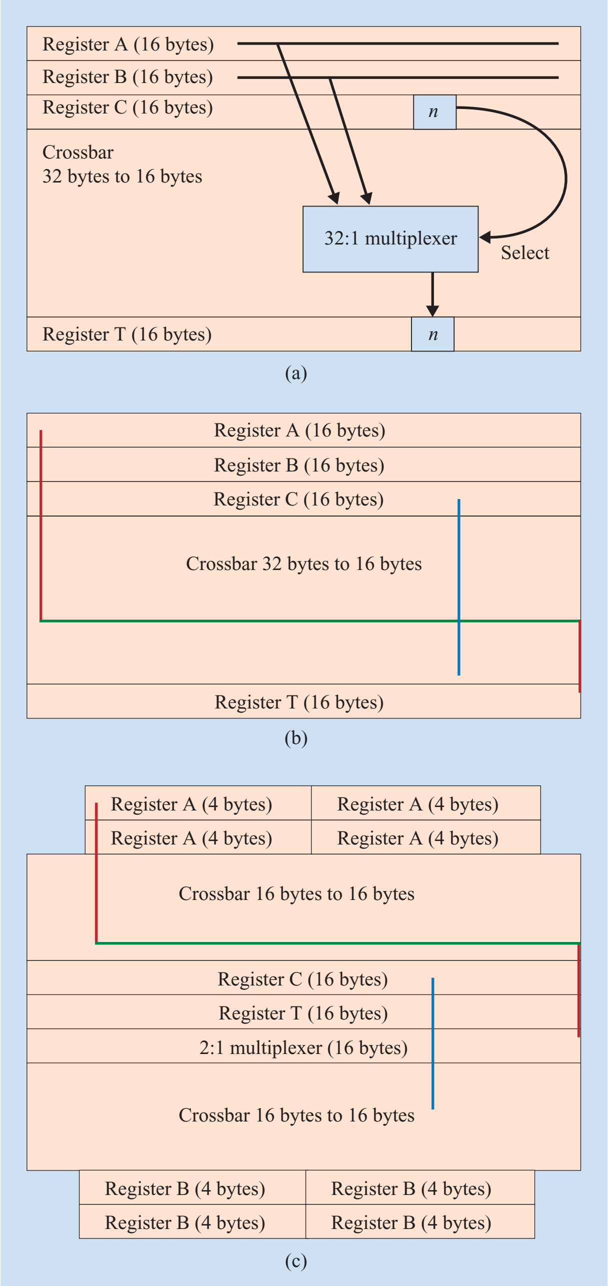
There are other instructions performed in the vector PM that must modify the A and B input operands before the values are applied to the crossbar. These modifications are done in the two pipeline stages before the crossbar switch. The pack pixel, for example, picks predefined bits from a word. The pack instructions with saturation force the saturation value depending on the pack result and the opcode. For the unpack instructions, predefined bits are distributed over wider areas of target register T.

Implementation of the crossbar switch

With the POWER6 processor high-frequency design goal, the main challenge for the implementation of vector PM was to design an efficient crossbar switch. A design in which the crossbar switch uses two pipeline stages would have increased the back-to-back latency to five cycles and, more importantly, would have needed a register bank for the intermediate result. The intermediate results of a crossbar contain several times more data than the final result. The challenge was to get the shortest possible back-to-back latency with a minimum of silicon area and power consumption.

The classical permute crossbar layout shown in Figure 2(a) does not achieve the required cycle time. It implies the wiring distances shown in **Figure 2(b)**. The vertical blue line is the control vector distribution. The red (vertical) and green (horizontal) lines are the distance to travel from the leftmost byte of operand register A to the rightmost byte of the target register T. Along this distance there are the logic gates forming the 32:1 multiplexer. In our technology, the fastest circuit to implement the 32:1 multiplexer is a combination of 4:1 and 2:1 multiplexers. Although the propagation delay of these logic gates consumes part of the cycle time, the majority of the delay inside the crossbar is caused by the length of the wiring and the buffers needed to drive these long wires.

The first step to reduce the amount of vertical wire is to move register B to the bottom of the crossbar and register T to the vertical center. The 32:1 multiplexer is split into two 16:1 (each constructed out of 4:1 multiplexers) and a 2:1 multiplexer near the target register. This change reduces the red line to approximately half the length of the implementation shown in Figure 2(b). Moving register C near register T cuts the effective length of the blue line in half, which is the distance from the source (register C) to the most distant sink.



(a)

(b)

(c)

Figure 2

Crossbar switch: (a) classical layout; (b) classical wiring; (c) POWER6 processor core layout wiring.

Table 1 VMX simple instructions.

Instruction	*SIMP* (bits)	*Subunit*	*Description*
Vector add and subtract modulo	8, 16, 32	Adder	Result modulo maximum value; computes signed and unsigned
Vector add and subtract saturate	8, 16, 32	Adder	Result saturates to minimum or maximum value, signed and unsigned
Vector add carryout unsigned	32	Adder	Carryout value of most significant bit is given in the least significant bit of result
Vector average	8, 16, 32	Adder	Average of operands, signed and unsigned
Vector logical	1	Adder	Boolean AND, OR, XOR, NOR
Vector select	1	Logical	Selects bitwise between vector A and vector B based on vector C
Vector integer compare	8, 16, 32	Adder	$>$, $=$ [+ record of all 0s and 1s]
Vector single-precision floating-point compare	32	Logical	$>$, $=$, $\geq$, bounds [+ record of all 0s and 1s]
Vector integer minimum and maximum	8, 16, 32	Adder	Minimum or maximum of the A and B integer operands
Vector single-precision floating-point minimum and maximum	32	Logical	Vector single-precision minimum or maximum of the A and B floating-point operands
Vector rotate left	8, 16, 32	Rotator	Rotate left of the operand A according to the shift amount B
Vector shift left or right	8, 16, 32	Rotator	Shift left or right operand A according to the shift amount B
Move to or from the VSCR	n/a	Logical	Move to the VSCR

Note: VSCR: vector status and control register.

The actual layout of vector PM is shown in **Figure 2(c)**. The total wiring length is reduced further by cutting off 25% of the horizontal green line. With this layout, the crossbar switch reaches the cycle-time goal of the POWER6 processor design with a minimum amount of silicon area and power consumption.

ALU XS

The XS is responsible for fixed-point calculations of a simple nature (i.e., not multiplies or divides). Many of the applications for VMX acceleration have to perform the same calculation on multiple fixed-point data elements. An example of such a use would be for matrix representation of graphics. For typical applications, the number of bits needed for each data element is usually 8, 16, or 32. The data elements are signed or unsigned integers, and the results can be modulo arithmetic or saturating arithmetic. Furthermore, rotates and logical instructions are sometimes required.

The broad range of simple instructions is shown in **Table 1**. Each of these instruction types has a separate instruction opcode for each supported data element width. Note that both integer and floating-point compares are implemented by this unit. As a result, 86 different instructions are supported by the XS.

The biggest challenge for implementing the XS was the high-frequency target (13 FO4). In previous designs, such as the PowerPC 970, the XS is implemented as a two-cycle pipeline. Mapping that design to the POWER6 processor frequency target would have doubled the pipeline depth, significantly impacting performance. Therefore, a new design was needed for the POWER6 processor. In this new XS, the actual execution pipeline is limited to two cycles. The third cycle is used for the result distribution, as shown in the VMX overview (Figure 1).

The ALU XS is built out of three subunits: the adder (ADD), logical (LOG), and rotator (ROT). In Table 1, the subunit column shows how the instructions are mapped to each subunit. The mapping was done in such a way that equally balanced the delays. For example, the integer and floating-point compares are implemented in separate subunits to support the required cycle time. The following section describes the specifics of each subunit to enable the two-cycle execution of the POWER6 processor simple unit implementation.

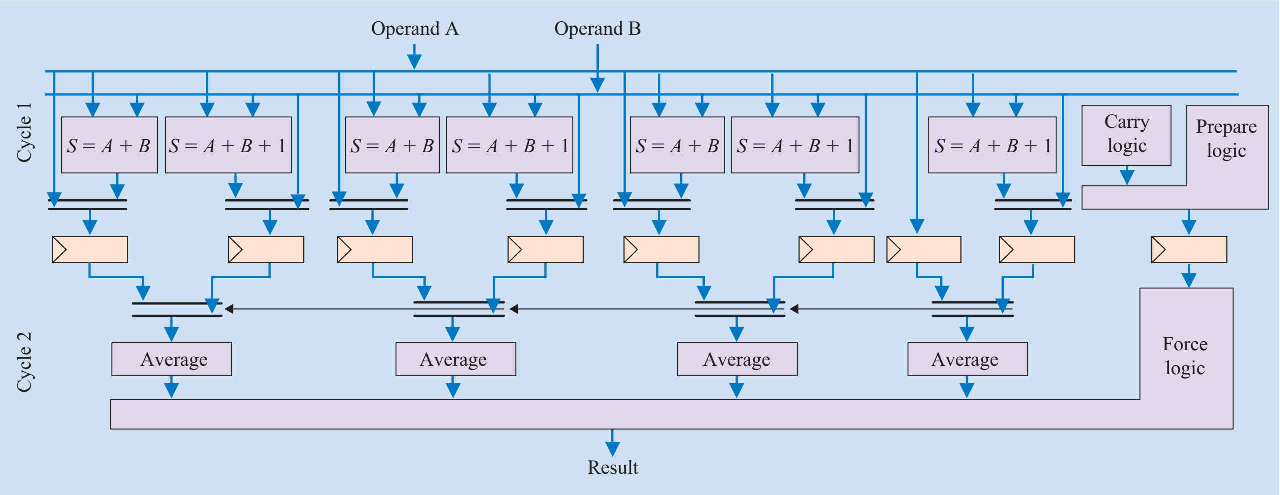


Figure 3

ADD macro.

Adder

The simple unit adder subunit is built out of four ADD macro instances with a 32-bit-wide dataflow. The four instances of the ADD macro implement a 128-bit-wide datapath that enables SIMD instructions for add, subtract, average, and integer compare to execute simultaneously on multiple data elements. **Figure 3** shows the new two-cycle 32-bit-wide ADD macro implementation.

The first cycle is divided into three portions: A static adder [3] using a carry select structure with 8-bit-wide adder blocks, the carry generation, and the preparation logic for compare, minimum and maximum, and average operations. The carry logic is shared between the sum and compare logic [4]. The compare logic handles 8-bit, 16-bit, and 32-bit signed and unsigned integers as well.

The critical paths are in the carry logic outputs, and the implementation of this logic is highly optimized to fit within the first cycle. To enable the SIMD operations of addition, subtraction, and compare, the carry network is extended for the additional functionality needed for the various operand lengths. The actual sum select of the carry select adder (or minimum and maximum selection) is done at the beginning of the second cycle. The two 32-bit intermediate values are selected based on 8-bit portions. The selection is done for add and sum as well as minimum and maximum operations.

For average instructions, the result is shifted by one to the left for the $(a+b)/2$ calculation. The preparation logic in the first cycle handles saturation and the setting of the multiplexer select signals for the second cycle generation of the add, subtract, minimum, maximum, and integer compare results. Finally, the force logic (as shown in Figure 3) generates the compare result `TRUE` (all bits "1") and `FALSE` (all bits "0").

Logical

The logical subunit performs four types of instructions: Boolean logic, vector select, single-precision floating-point compare, and floating-point minimum and maximum. The 128-bit dataflow handles these instructions with four instances of a 32-bit LOG macro. The critical paths are contained in the floating-point compare operations. Because of the POWER6 processor cycle time, an additional carry network was needed for the floating-point compares. The network is a carry select structure (as in the ADD macro), but it is optimized for floating-point compares.

Rotator

The rotator is capable of performing rotate left, shift left, and signed and unsigned shift right SIMD operations. As with the ADD and LOG macros, there are four instances of the ROT macro, each 32 bits wide, to implement the 128-bit dataflow.

There are two approaches commonly used in state-of-the-art SIMD rotator and shifter macros. Independent logarithmic rotator arrays for each of the supported data types can be used with a select between the different results, or a more complex rotator array structure that takes the width of the different data types into account can be used. In either case, the rotate array block is followed by a masking stage for the shift results. Neither of these approaches can be used for the POWER6 VMX because of area and cycle-time limitations.

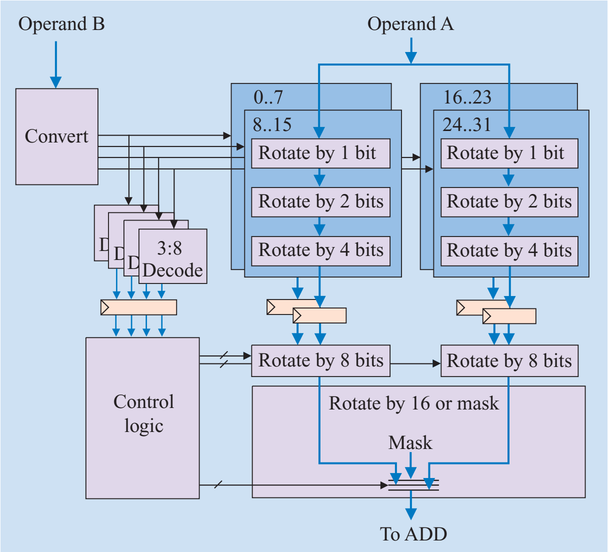


Figure 4

ROT macro.

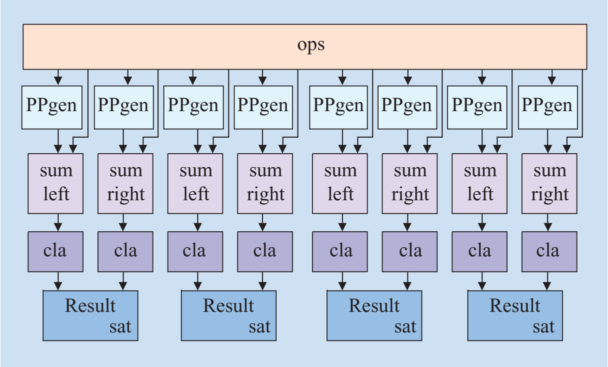


Figure 5

XC dataflow.

Figure 4 shows the organization of one of the 32-bit rotator macros. All rotates and shifts are executed as rotate left within the first execution cycle and corrected and masked in the second cycle. The byte left rotators rotate up to 7-bit positions. The inputs for the byte rotators are the four operand A bytes, and the rotate matrix is controlled by the rotate and shift amounts in operand B. This generates the correct rotate results for the 8-bit-wide data elements and a simple masking is done in the second cycle for shifts instead of rotates. However, in the case of 16-bit data elements (halfwords) or a 32-bit data element (word), the rotator result of the first cycle needs to be corrected by crossing bit ranges. This crossing of bits is performed by the halfword and word-cross correction logic based on the masks generated in the first cycle. The use of this rotator structure enabled us to balance the dataflow rotate path delay with the mask control delay. This balancing is critical in supporting the POWER6 processor cycle time.

XC

The POWER6 processor XC executes SIMD multiply, multiply–add, multiply–sum, and sum-across instructions. The unit is broken into four 32-bit datapaths, each containing two multiply structures (left and right). Each of these halfword multipliers can do simultaneous multiples supporting both byte and halfword multiples (8 × 8 and 16 × 16).

The XC comprises five different blocks (**Figure 5**). At the top of the complex pipe is the operand (ops) macro. The ops macro handles selecting and multiplexing the operands. Sources include a local feedback path from a previous complex instruction or from the VMX bypass macro. The VMX bypass macro multiplexes result data from other VMX subunits and the register files. The first true pipe stage for the unit is contained in the PPgen (partial-product generation) macro. It generates partial products for the booth multiplier in later stages.

Pipe stages 2 through 4 encompass the sum blocks. Each sum block contains the partial-product (multiply) adder and has additional support logic to add the *B* and *C* operands. The left and right versions are identical except for the sum and carry interfaces; the right transmits and the left receives them.

The cla (carry lookahead adder) macro comprises the fifth stage of the pipe. It contains a 36-bit cla whose inputs are the 36-bit sum and carry results from the sum block. The output of this macro is the sum, which is broken into a 32-bit result and 4-bit overflow.

The final stage is the result macro, which selects the correct result for the appropriate instruction being executed: an add, even 8 × 8 multiply, odd 8 × 8 multiply, even 16 × 16 multiply, or an odd 16 × 16 multiply. The products and overflows from the cla macro are tested for positive and negative saturation (labeled *sat* in Figure 5) conditions and can affect the final result (depending on the instruction).

Vector floating-point unit

The vector floating-point unit (VFU) operates in a four-way SIMD fashion on 4 × 32-bit binary single-precision data. The POWER6 VFU is a fully pipelined, 6.5-cycle, fused multiply–add design with a 13-FO4 cycle time.

As with the other units, the biggest challenge of the POWER6 processor VFU design was the high-frequency, low-latency design point. This required many optimizations, most of which are similar to those used in the BFU [5].

For the VFU, the aggressive design target also resulted in a special design style and floorplan.

As part of the VMX accelerator, the VFU had some constraints that did not apply to the BFU. First, as an accelerator, the VMX has a more restrictive interface to the core. Once an instruction has reached the POWER6 processor VFU, it has to proceed unimpeded (as explained above in the instruction flow details section). As a consequence, no data-dependent stalls, rejects, or traps are supported for the VFU; therefore, some design tricks used in the BFU—e.g., optimizing for the common case and adding extra cycles for corner cases using stalls—were not applicable to the VFU.

Second, the vector register file is shared by all of the functional subunits of the VMX. This design has its pros and cons for the VFU. To reduce hardware requirements and speed up execution, each instruction is executed in the subunit that is best suited for it, independent of the data type. To enable this sharing, the data in the register file is in memory format, which is another drawback for VFU design. For register file data, it is preferable that FPUs use an intermediate format that provides extra information, like the integer bit or tags for special operands like NaN (not a number), infinity, and zero [5]. The VFU is limited to the memory format. Thus, the decoding and packing of the operands—which, for the BFU, happens on the load and store interface—becomes part of the VFU pipe.

The rest of this section describes how each of these challenges and constraints had an impact on the VFU design, its instruction execution, pipeline, design style, and floorplan.

VFU instructions

Since all VMX subunits share the vector register file, each instruction is executed in the subunit that can support it in the most efficient way. Thus, in the POWER6 processor design, most of the VMX binary floating-point instructions are executed in the VFU, including the following types of instructions:

- Add and subtract.
- Fused multiply–add ($A \cdot C + B$) and fused negative multiply–subtract ($-(A \cdot C - B)$).
- Converts to and from integer. The integer can be signed or unsigned.
- Round to integral value with the four rounding modes as defined by the IEEE Standard for Binary Floating-Point Arithmetic [6].
- Estimate operations for $1/x$, $1/\sqrt{x}$, $\log(x)$, and $\hat{2}x$.

Floating-point compares and the minimum and maximum functions are executed in the XS. These instructions can be executed much faster with special integer arithmetic than with a floating-point multiply–add unit [7].

Denormal number support

The floating-point memory format for single-precision data divides the 32 bits into a sign s (bit 0), an exponent e (bits 1 to 8), and a fraction f (bits 9 to 31). Based on the exponent, the data has to be interpreted in three different ways:

1. If e consists of all 1s, the data represent either infinity or NaN, depending on the value of the fraction.
2. If e consists of all 0s, the represented number is $(-1)^s \cdot 2^{e+1-127} \cdot 0.f$; this is either a zero ($f = 0$) or a denormal number.
3. In all other cases, the represented number is a normal number with value $(-1)^s \cdot 2^{e-127} \cdot 1.f$. Note that denormal numbers have a different integer bit value, and their exponent needs to be incremented by one.

Since denormal numbers are rare, especially in media applications, previous VMX implementations [1] handle only normal data and, in cases of denormal numbers, trap to software-assist code. This simplified the design of the floating-point pipeline.

The POWER6 processor core does not support data-dependent stalls, rejects, or traps for VMX operations. Thus, the POWER6 processor VFU has to process denormal numbers at full speed within the regular floating-point pipeline. This requires modifications to the FPU pipeline, as described in the next sections.

Operation modes

The VMX implements two modes, a Java** program mode and a fast non-Java program mode. In Java program mode, the VFU conforms to the Java program specifications [8], which are a subset of the IEEE 754-1985 standard [6]. It differs from the IEEE standard in the following ways:

- Except for the convert and round instructions, all VFU instructions use the rounding mode round to nearest even.
- Trapping on floating-point exceptions is not supported, and exception information is not collected.

This special handling of IEEE exceptions is essential for the POWER6 processor design. It enabled a VMX design without data-dependent stalls, rejects, or traps and thus allowed for a much simpler interface to the POWER6 processor core.

In non-Java mode, denormal operands and results are forced to zero. This feature was originally introduced to allow for faster, imprecise processing of denormal data. Some graphics applications can tolerate the loss of precision for these tiny numbers but cannot afford the performance degradation for their precise processing. In the POWER6 processor VFU design, there is no performance difference between the two modes because of special hardware support for denormal numbers.

High-precision estimates

To support estimates with a high precision, the VFU performs a table lookup together with a modified multiply–add operation. The algorithms and tables for the high-precision estimates are taken from a previous VMX design but had to be extended to support denormal numbers. For example, for the reciprocal, reciprocal square root, and log estimate, this was done by adding a normalization stage before the table lookup.

Normalization and table lookup take four cycles. During this time, no other VFU instruction may be started so that the pipeline can be reused for the multiply–add operation. Since this stall depends only on the instruction word, it can be supported by the processor core.

Pipeline

By default, the datapath executes the multiply–add instruction $A \cdot C + B$. Other instructions are executed as special multiply–adds, e.g., add is executed as $A \cdot 1 + B$.

As described above, the operands of the VFU are in memory format. Extra packing and unpacking circuits before and after the VFU datapath would have increased the latency of the VFU and were, therefore, not an option. Instead, the unpacking and packing are done as part of the datapath, hiding most of their latency.

The VFU speculatively starts the execution assuming that the operands are normal numbers. If one or more of the operands are denormal numbers, a late correction has to be performed. The correction mechanisms work as follows.

Multiplier correction—The multiplier uses radix-4 Booth encoding and supports 14 partial products using three levels of 4:2 compressors. Two partial products bypass the first level of 4:2 compressors and are, therefore, less timing critical. These partial products are used for late correction of denorm inputs.

The idea for the denorm correction is to split the multiplication $ia.fa \cdot ic.fc$ (for integer bits ia,ic and fractions fa,fc) into the two terms $0.fa \cdot ic.fc + ia \cdot ic.fc$. This increases the number of partial products by one, to 14. Since $ic.fc$ is Booth recoded, only one of the partial products of $0.fa \cdot ic.fc$ depends on the integer bit ic. This partial product and the term $ia \cdot ic.fc$ bypass the first compression stage.

Aligner correction—In parallel to the multiplication, the fraction of the B operand is aligned to the product. The timing-critical path of the aligner is the shift amount computation. The implicit bit of the B operand is not timing critical.

The alignment shift amount is the difference between the exponents of the product $A \cdot C$ and the addend B. In the VFU, it is computed assuming normal inputs. Consequently, for denormal inputs the exponent is assumed to be -127 instead of -126. Thus, the shift amount can be off by either $+1$, -1, or -2, depending on which of the operands are denormal.

An error of -2 in the shift amount can occur only if both multiplicands are denormal. In this case the product is much smaller than the addend; therefore, the product contributes only to the sticky computation and not to the sum. For this special case, the exact shift amount does not matter. For the other two cases, the correction by ± 1 is done by a post-shift at the beginning of the adder.

Fast carryout detection

The adder computes either the sum or the absolute difference of the aligned addend and the product. This is done using an end-around-carry scheme [9, 10] for which the carryout of the addition is timing critical. The carryout computation, therefore, needs special attention.

For the improved carryout computation, it is essential to determine the position of the leading one of the aligned addend relative to that of the product. For normal operands, the leading one is identical to the integer bit. Thus, the position of the leading one can be derived easily from the shift amount.

For denormal numbers, this conventional mechanism does not work, because the integer bit is zero; the denormal number can actually have up to 23 leading zeros. The new solution implemented in the VFU works as follows: The VFU counts the number of leading zeros of the B operand. It then corrects the position computed by the conventional mechanism by this number.

Design style

In a high-frequency design, a lot of circuit effort and tuning is needed to meet all of the design checks. To reduce the overall design effort, and to lighten the burden of the circuit designers, only the most timing-critical macros were implemented in full custom style [**Figure 6(a)**]. The remaining macros were implemented as synthesized random logic macros.

For the synthesized macros to meet the timing target, a semicustom approach was used. The synthesis was guided by writing gate-level VHDL (Very high-speed integrated circuit Hardware Description Language). If needed, we

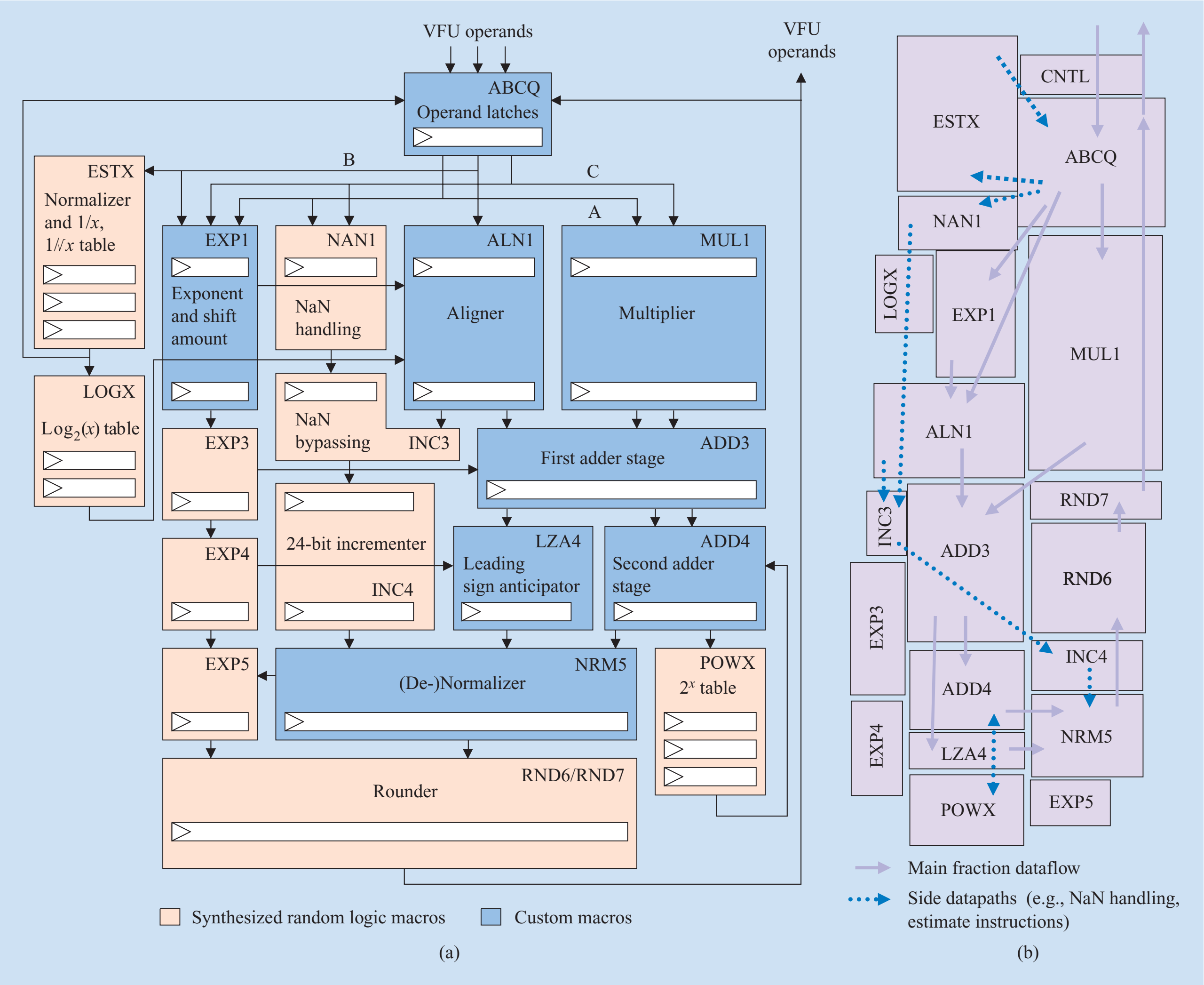


Figure 6

VFU (a) dataflow and (b) floorplan of one of the VFU slices. [The macros ESTX, LOGX, and POWX are the special hardware used for the estimate instructions 1/x, 1/sqrt(x), log(x), and 2^x. The macro CNTL decodes the instruction word and generates the control signals for the other floating-point unit macros.]

explicitly specified the gate type and drive strength. This process was carried out by the logic design team.

With this approach, it was possible to synthesize half the VFU macros and still meet the aggressive timing target. In addition to the control, most of the exponent datapath, the special case handling, the incrementer part of the fraction adder, the rounder, and all of the support macros for the estimate instructions were synthesized.

Pipeline latency

With its 6.5-cycle pipeline, the VFU supports seven-cycle *back-to-back* (fully pipelined) issuing of VFU instructions. Compared to the POWER6 processor BFU design with its six-cycle back-to-back issuing of VFU instructions [5], the seven-cycle VFU design seems less aggressive.

The BFU achieves the six-cycle back-to-back issuing of VFU instructions by forwarding an unrounded intermediate result and corresponding correction terms; the fully rounded result is available only at the end of cycle seven. In some corner cases, the early bypass cannot provide the correct result; this leads to data-dependent stalls in the BFU. Because the POWER6 processor VMX interface does not support data-dependent latencies, this forwarding approach was not applicable to the VFU design.

To achieve six-cycle back-to-back issuing, the VFU pipeline would have to be reduced to fewer than six stages. This would require the whole dataflow to be implemented as highly tuned custom macros and would have increased the design effort, power, and area of the VFU beyond budget.

Floorplan

In a high-frequency design, the wire reach puts severe constraints on the unit floorplan. With a default wire, it takes a whole cycle to send signals from one corner of a VFU slice to the opposite corner. As a result, many critical intermacro paths require good wire to close timing, but those wires are limited. Thus, the floorplan becomes a challenge of its own.

To ease the VMX-level wiring of the wide operand and result buses, the operand and result ports of the VFU should all be located near the top of the VFU. This also helps to reduce the latency of the data distribution. Within a VFU slice, it was essential to find a macro placement that keeps the intermacro wires short and reduces wire congestion so that critical connections can afford wider wire.

The four VFU slices of the POWER6 processor VMX are placed side by side, similar to other vector FPU designs [11]. However, each slice uses a special two-stack, U-shaped floorplan, as indicated in **Figure 6(b)**, to satisfy the constraints mentioned above.

According to the pipeline diagram [Figure 6(a)], the first four stages of the VFU have two parallel dataflows for the main fraction path, passing through the aligner and multiplier and the adder and leading-zero anticipator (LZA). For the last three pipe stages, there is just one fraction dataflow. This maps very well onto the U-shaped floorplan of the VFU. The aligner and multiplier are equally timing critical; therefore, they are placed side by side below the operand latch macro, which holds the operand latches. The outputs of the aligner and multiplier then pass down the right stack through the adder and LZA, switch to the left stack, and pass up through the normalizer and rounder. The result then jumps over the multiplier to the top of the VFU, where it is latched. The paths to the estimate macros and through the incrementer are less timing critical. They can be placed at the side and can afford longer wire, as shown by the dotted blue lines in Figure 6(b).

Load and store formatters

The vector instruction set contains numerous loads and stores that require manipulation of the data used in the operation. Some of this data manipulation is done within the VMX unit. This breakdown in formatting was done to limit the VMX-specific formatting requirements to the VMX unit itself rather than forcing further complexity into the load/store unit (LSU).

The load formatters perform a variety of functions, such as masking off particular bits within the destination register or shifting the loaded data left or right into the destination register. In addition, doubleword swapping is required to accommodate little-endian data formatting.

The load formatters also take advantage of the fact that load data for a single load instruction is presented across two cycles. This is because the remainder of the POWER6 core operates on 64-bit operands, while the VMX operates on 128-bit operands. The two-cycle data presentation allows the formatters to be 64 bits wide (instead of 128 bits). The results of the staging of the formatter are then folded back on themselves to present all 128 bits of load data on the same cycle to the VRF. This allows for a reduced area design point that aided in achieving a high frequency.

Recovery actions

The POWER6 processor design introduces an RU, which maintains a shadow copy of the processor state. The VMX unit supports the RU by presenting its results to the RU in program order and by supporting data returned from the RU during a recovery.

Two cycles after the results are written into the VRF, the VMX unit sends results of its execution pipelines to the RU. This additional delay is due to floorplan constraints.

To save resources in routing VMX load results to the RU, these results go directly to the RU from the LSU, bypassing the VMX unit entirely. This presented a difficulty because the LSU performs only some of the data manipulation required by the AltiVec architecture; the remainder of this formatting is performed by the VMX unit itself. As a result, the unformatted data is stored in the RU along with other information that is required by the VMX unit to recreate the formatted VMX load results. This data consists of 4 bits of the load instruction opcode and the bottom-most 4 bits of the load target address. Using this information and the unformatted load data, the VMX load formatters are able to generate the proper results following a recovery.

In the event of a recovery, the RU sends an indication to the VMX unit that a recovery is taking place. This indication is used to block any incoming instructions from the IDU, and instead the VMX unit waits for inputs from the RU. When the RU begins presenting the recovery data, it is handled by the VMX control logic so that the recovery data looks like incoming VMX load instructions. The recovery data is then fed through the load formatters before being written into the VRF.

Because the RU stores the load formatting information in addition to the unformatted load data, the load formatters are able to generate the proper results and write them into the VRFs. For data that is the result of a

VMX logical operation, the RU forces the formatting information to a value that results in no formatting shifts of the results by the VMX. So while these results still appear to be from a VMX load, the load formatters allow the data to pass through untouched.

Introduction: DFU

Transaction processing is one of the major uses of computers. Financial transactions require exact decimal arithmetic. Typically, these transactions involve many decimal multiplications, such as multiplying the cost per minute or the tax rate per charge. These decimal calculations must be rounded to a decimal radix point. The decimal arithmetic component of these financial transactions is becoming more prevalent given that other more general components are continually being improved.

Decimal arithmetic is natural to humans and has been the standard numeric system for thousands of years. With the advent of the computer age, binary arithmetic has become popular. There are two common binary number systems in computers: fixed point, or integer, and floating point. Decimal calculations cannot be directly implemented with binary floating point because fractions such as 0.1 cannot be represented exactly. Instead, decimal floating-point operations have been emulated with binary fixed-point integers. Binary integers have performance problems because of their limited range and their difficulty in scaling and rounding. Binary integer implementations keep the exponent and coefficient partitioned and operate on them separately. Rounding and scaling are more difficult with integers than a decimal format. Rounding to a decimal radix point is unnatural in binary format and requires many operations, such as leading-ones detection, table lookup, division, or an equivalent reciprocal multiplication, and a further detection of trailing zeros.

In prior computer systems, decimal formats have been limited to fixed-point decimal implementations that were implemented on mainframe and minicomputers. There also has been limited support for binary-coded decimal (BCD) arithmetic instructions on desktop systems, such as the x86 architecture in which two-digit arithmetic is supported. The bigger implementations support fixed-point decimal format in a packed BCD format in which each nibble (4 bits) represents a BCD digit, and there are up to 31 digits and a sign. These systems provide arithmetic operations for the coefficients, but they provide no implicit rounding. The range is limited to the number of coefficient digits or a separate exponent is maintained in an integer format. The fixed-point decimal instruction set of the IBM z/Architecture* technology was defined in the 1960s when high-speed register files were very costly or even impossible to build, and it is defined to use data directly from memory. Today, these operations create performance bottlenecks in high-speed processors because of the scalar nature of the workloads. Many computations depend on the result of the prior computation. Rather than resolving dependencies in the execution unit, the dependency is a memory interlock and is resolved in the cache or the LSU. Memory interlocks typically require more cycles than register interlocks, which in some systems can be eliminated with register renaming.

A new decimal floating-point system is needed, and the proposed IEEE 754R standard [12] defines the formats, possible encodings, and the execution of arithmetic operations. The POWER6 processor design introduces a new architecture to support this proposed standard and, for the first time, implements this decimal floating-point architecture in hardware. A decimal floating-point number system is implemented that implicitly rounds operations to a decimal radix point. This decimal floating-point architecture is implemented completely in hardware for both a 64-bit and a 128-bit format.

The following sections of this paper describe the architecture, including formats, encodings, status information, and instructions, and then describe the hardware implementation. The basic dataflow is described, followed by details concerning the operations of addition, multiplication, and division.

Architecture

Given that commercial databases have more than half of their numeric data in a decimal format [13], a BCD-like format is desired. BCD encoding is not very efficient and utilizes only 62.5% of the encoding space, but it allows quick conversion from a database and is optimal for shifting, scaling, and extracting fields of data. Binary integer encoding provides 100% compression and fast execution of high-order arithmetic operations, but it is slow in reading and writing data from databases and performing simple operations and rounding.

Binary integer encodings have their disadvantages, so another encoding was desired. This encoding is a BCD compressed format called *densely packed decimal* (DPD) [14]. Three BCD digits, which would normally require 12 bits to represent, are compressed to 10 bits in what is called a *declet*. This provides greater than 97.6% efficiency (or 1,000 out of 1,024 possible representations). It also has the advantage of requiring only three logic gate delays to convert from BCD to DPD and from DPD to BCD format. The DPD format has the same advantages of the BCD format, but with the additional benefit of being more compact and allowing more digits to be represented in a given data width.

Formats

For a 64-bit data width, a BCD format can represent only 16 digits of coefficient without an exponent, or about 14

Table 2 Decimal floating-point combo field encoding.

Most significant coefficient digit	*Most significant two bits of the exponent value*			*Special values*	
	00	01	10		
0	00000	01000	10000	Infinity	11110
1	00001	01001	10001	NaN	11111
2	00010	01010	10010		
3	00011	01011	10011		
4	00100	01100	10100		
5	00101	01101	10101		
6	00110	01110	10110		
7	00111	01111	10111		
8	11000	11010	11100		
9	11001	11011	11101		

digits with a 7-bit exponent. With DPD encoding, it is possible to represent 15 or more digits and an exponent. A coefficient of 50 bits (or five declets) can encode 15 decimal digits with DPD encoding. This leaves 14 remaining bits, which is a little excessive for a base-10 exponent. Encoding one more coefficient digit in BCD format is possible, but it is inefficient. Instead, the IEEE 754R committee suggested combining a BCD coefficient digit with a 2-bit exponent field of limited range, in which it can have the value 0, 1, or 2, but not 3 [12]. This combined field is 5 bits and is called the *combo field*. The other 9 bits are composed of a sign bit and an 8-bit exponent continuation field that comprises the lower 8 bits of the 10-bit exponent. The combo field encoding is shown in **Table 2**. If the most significant digit is less than eight, then the first two bits are the most significant exponent bits, and the remaining three bits represent the most significant coefficient digit. If the most significant digit is eight or nine, then the first two bits of the combo field are 11, followed by the two most significant exponent bits, and the remaining bit indicates whether the most significant digit of coefficient is eight or nine. There are two remaining encodings possible that start with four ones (1111), and these are used to encode the special numbers infinity and NaN.

The IEEE 754R standard provides two basic decimal formats: decimal64 and decimal128, which are, respectively, a 64-bit doubleword and a 128-bit quadword. It also provides one storage format: decimal32, which is a 32-bit word in length.

A decimal floating-point number A can be represented by

$$A = (-1)^{(As)} \cdot 10^{(Ae-\mathrm{bias})} \cdot Ac,$$

which includes a sign bit (As), a biased exponent (Ae) represented as an unsigned binary integer similar to the binary floating-point format, and a coefficient (Ac). The coefficient is not normalized and it is an integer. The IEEE 754R standard allows the coefficient to be represented in binary integer format or DPD format [6]. The POWER6 processor design uses the DPD format for its quick conversion of BCD databases.

The exponent of the least significant bit of a coefficient is referred to as the *quantum*. The quantum indicates the magnitude of the unit of measurement, such as millimeters (e.g., 10^{-3}) or pennies (e.g., 10^{-2}). The concept of representation includes more than value; it includes quantum as well. The set of multiple representations of the same value are called *cohorts*. Each operation is defined to have a preferred quantum. If a result is exact, the member of the cohort with its exponent equal to or closest to the preferred quantum is chosen. For addition and subtraction, the preferred quantum is the minimum of the two operand quantums, which is written as min(Q(X), Q(Y)). Even though there may be several representations of a value, for each operation there is only one acceptable representation of the result. For instance, $1.0 + 1.00$ is represented in the decimal format as $10\mathrm{E} - 1 + 100\mathrm{E} - 2 = 200\mathrm{E} - 2$. The result is represented using the smaller quantum, and there is only one acceptable result.

Table 3 gives some parameters of the different decimal formats. First, the precision of the coefficient in decimal digits is given, followed by the number of bits in the biased exponent continuation field, the range of the signed unbiased exponent, the maximum normal number ($N_{\max}$), and the minimum normal number ($N_{\min}$). The formats follow the guidelines of the IEEE 854 standard [15] for making the next larger format at least $2p + 2$ digits, where p is the precision. Decimal32 format is not very useful since it has only seven digits, and many calculators have more digits. Its only purpose is a reduction of the storage requirements for constants. Therefore, it is implemented only with limited support.

The decimal floating-point formats provide a greater range than their binary floating-point counterparts. For binary, the three formats range from $10^{\pm 38}$, $10^{\pm 308}$, and $10^{\pm 4,932}$.

Floating-point register and FPSCR

To reduce the amount of area dedicated to decimal floating-point operations, the register file is reused from the BFU. Because of the fundamental differences in the requirements met between these two radices, a program is unlikely to require both binary and decimal floating-point computations simultaneously. The floating-point register (FPR) files, located in the BFU, are used by both. Quadword operands take up an even–odd pair of FPRs

Table 3 Decimal floating-point format parameters.

	Format		
	Decimal32	*Decimal64*	*Decimal128*
Coefficient precision (p)	7	10	34
Bits of exponent continuation	6	8	12
Exponent range	−101 to 90	−398 to 369	−6,176 to 6,111
N_{max}	$(10^7 - 1) \times 10^{90}$	$(10^{16} - 1) \times 10^{369}$	$(10^{34} - 1) \times 10^{6,111}$
N_{min}	1×10^{-95}	1×10^{-383}	$1 \times 10^{-6,143}$

and are addressed by the even register. The even register holds the most significant bits of the quadword, and the odd register holds the least significant bits. There are 32 doubleword registers that can also be addressed as 16 quadword registers. Both doubleword and quadword data is loaded using the binary floating-point load doubleword instructions. No new load or store instructions are added for decimal floating point.

The floating-point status and control register (FPSCR) is also used by both binary and decimal floating-point architectures. Only the rounding mode is separated for decimal floating point. The decimal rounding mode field is 3 bits and allows eight different rounding modes. The first four rounding modes are the same as binary: round to nearest even, truncate, round toward positive infinity, and round toward negative infinity. The additional four rounding modes are round to nearest ties away from zero, round to nearest ties toward zero, round away from zero, and round to prepare for shorter precision. The last rounding mode needs a little explanation. It creates a result that is a rounding equivalent of the infinitely precise intermediate result for further rounding to $p - 1$ digits. This is accomplished by truncating the intermediate result for all cases except when the least significant digit is 0 or 5, and in this case the result is incremented if inexact. This makes a least significant zero or five occur only when the intermediate result is truly exact. Without this perturbation, a zero would appear to be an exact result and a five would appear to be exactly halfway between two $p - 1$ digit representations. This new rounding mode for an arithmetic instruction, coupled with a new instruction called *reround*, allows for exact variable precision rounding.

For decimal as well as binary, the FPSCR also records the class of the result for arithmetic instructions. In some cases this results in a couple of cycles of additional latency since subnormal numbers are difficult to detect. For binary floating-point numbers, a subnormal number can be detected quickly by examining the exponent since arithmetic operations normalize the result. For decimal floating-point numbers, detection of a subnormal number is dependent on the number of coefficient digits, which is determined by a leading-digit detect and a subtraction of the precision, followed by an addition of the exponent and a comparison with a constant. The classes of results for decimal floating-point numbers are subnormal, normal, zero, infinity, quiet NaN, and signaling NaN, which is recorded in an encoded flag field of the FPSCR after every decimal arithmetic operation.

Instruction set

The POWER6 core supports decimal64 and decimal128 basic formats directly with arithmetic operations and provides support for converting to and from the decimal32 storage format. Operations are provided for basic arithmetic, test, quantum adjustment, conversion, and format instructions. All operations are defined as register-to-register operations to make interlock resolution easier and closer to the execution unit. Load and store instructions are borrowed from the binary floating-point architecture to also support decimal floating point.

The basic arithmetic instructions are add, subtract, multiply, and divide. They are defined to have two source operands and one target operand. Each source operand specifies an FPR for doubleword instructions or a pair of FPRs for quadword instructions, and the target operand designates a destination FPR or pair of FPRs. Other, more complex arithmetic operations must be implemented in software. The preferred quantum for add and subtract is the minimum of the quantum of the two operands. The preferred quantum for multiply and divide is, respectively, the sum and difference of the quantum of the two operands.

The test instructions provide a method for determining the data class of the operand. They also provide mechanisms for testing whether the number has been rounded or could possibly have a quantum that differs from one if greater precision were used. This is especially useful for implementing programming languages such as Java that support greater precision than the hardware precision. The test data group instruction does this by testing for extreme exponents and for the most significant

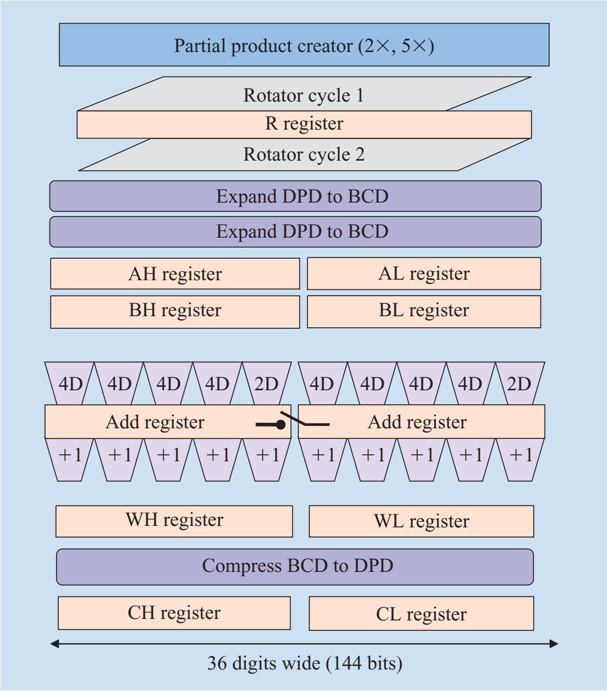


Figure 7

DFU dataflow.

digit nonzero. An extreme exponent is an indication that the result quantum may have differed from the preferred quantum and had been forced to be within the exponent range. The test for most significant digit nonzero is an indication of whether rounding could have occurred due to the intermediate result being different in a larger precision. It is particularly important to be able to emulate a different precision or different exponent range than that provided directly in hardware. Eventually many programming languages and applications will be optimized to the high-performance hardware decimal formats, but flexibility is needed to support all possibilities. The Java BigDecimal format allows exponents and precision to be greater than these format limits. Financial applications require the ability to round to any precision, especially a smaller precision. The emulation of a greater exponent range and precision is accomplished by the use of the test data group instruction.

The quantum adjustment instructions include quantize, round to floating-point integer, and reround. These instructions have a separate field designating the rounding mode to use or indicating whether to use the current FPSCR rounding mode. There are also separate instructions that affect or do not affect the inexact flag. The IEEE 754R standard defines *quantize* as forcing a quantum (such as pennies) for a value. This is especially useful prior to storing data in BCD format to a database. The round-to-integer instruction provides a mechanism for performing a ceiling or floor operation. *Reround* is an instruction that provides a way to round an arithmetic operation to a variable precision with only one rounding error. This is especially important for programming languages that provide the ability to round each arithmetic operation to a specified precision. It is also necessary for tax calculations that insist that a result be rounded to a specific precision or for programming languages that may have a slightly different precision than the hardware precision, for example, 32 digits instead of 34 digits.

The conversion instructions provide means to round or convert to and from the three decimal floating-point formats, as well as to and from the fixed-point or integer formats. These are the only instructions that support decimal32 format since it is only a storage format. In addition, it is possible to emulate decimal32 arithmetic using these instructions in conjunction with the arithmetic instructions.

The format instructions include instructions for inserting or extracting the coefficient to BCD format or the exponent to binary integer format. Also included are operations to shift the coefficient left or right. These format instructions are useful for fast conversion to and from existing commercial databases.

Altogether 54 instructions were added to the Power Architecture to support decimal floating-point formats. The POWER6 architecture for decimal floating point is optimized for implementing a BCD-like format in hardware. It provides instructions that support programming languages with greater precision and range than hardware-based formats such as Java or formats that are matched to hardware such as C and C++. Best performance is possible when the programming language provides data types that are identical to those in hardware.

Hardware implementation

The POWER6 processor DFU is rather small but very wide. Its main component is a wide 36-digit (or 144-bit) adder, shown in **Figure 7**. The POWER6 processor cycle time is approximately 13 FO4 and can support only a 64-bit binary add in one cycle without complementation. The widest decimal adder that could be built with complementation in one cycle is four digits. A four-digit decimal adder is actually four conditional one-digit adders, in which the sum of $A + B$, $A + B + 1$, $A + B + 6$, or $A + B + 7$ is chosen based on the carry into each digit.

Many replicated four-digit adders were used to construct the larger adder. At a four-digit group level, the carry into each of these adders is set to zero such that the final sum will be equal to this *sum* or *sum* + 1. In the next cycle, a four-digit increment is implemented to calculate *sum* + 1 while the carry into each group is determined and then used to select between *sum* and *sum* + 1. This is a relatively simple, replicated building-block design. The groups are actually 4, 4, 4, 4, 2 and 4, 4, 4, 4, 2, where the upper 18 digits can be separated from the lower 18 digits. To be able to include a guard digit and a round digit, this width is ideal since adders must be $p + 2$, where p is the precision (16 or 34 digits). Note that the adder has a two-cycle latency but can start a new computation every cycle. A control signal allows the adder to be reconfigured on the fly between one 36-digit adder or two 18-digit adders. Quad-precision operations typically use the full 36-digit adder, whereas double-precision operations tend to use two 18-digit adders.

The other components of the dataflow are two DPD-to-BCD expanders, one BCD-to-DPD compressor, a two-cycle pipelined rotator, a partial product creator, and eight 18-digit registers organized into four pairs (AH–AL, BH–BL, WH–WL, and CH–CL), where each pair is separated across the high-order and low-order half of the dataflow. The compressor and expanders convert to and from BCD and DPD formats, because the FPRs hold the data in DPD format to reduce space requirements. This takes only three gate levels, but additional delay is needed to replicate the doubleword data to both the high and the low half of the registers. Horizontal wire is particularly slow in the technology used and causes large delays in select lines that are repowered for 144-bit multiplexers. This is especially evident in the rotator, which could not be completed in one cycle; instead, it required two cycles to rotate to any of 36 digits. The rotator also includes a mask function to act as a shifter that shifts out data. The partial product creator provides a doubler (2×) and a quintupler (5×) to provide easy-to-create multiples for multiplication. Both doubling and quintupling in BCD format are digit-independent operations and do not require any carry propagation.

The operations of addition and multiplication are described for this dataflow. A description of division is also available [16].

Addition

Floating-point binary addition typically involves alignment of the operand with the smaller exponent, addition, normalization, and rounding. Floating-point decimal addition is slightly different and is separated into three cases: exponents equal, aligning to the operand with the smaller exponent, and shifting both operands.

Case 1—Exponents equal: When the exponents are equal, the following steps are performed:

1. Expand DPD data to BCD.
2. Add the coefficients.
3. If there is a carryout from the adder, increment the exponent, shift the coefficient right, and round.
4. Compress the result to DPD format.

No normalization is necessary in decimal format. At most, a shift by one digit right is necessary if there is a carryout. Rounding is performed by incrementing the result on a second pass through the adder. If rounding is not necessary, this operation requires only five cycles.

Decimal floating-point operands are read from the FPR in the BFU and are transmitted to the DFU. In more detail, and referring to Figure 7, the two operands are latched into the AL register and the BL register. In cycle 1, AL is driven to one expander to convert the DPD coefficient into BCD format and is latched in both BL and AH. The other operand in BL is driven to the other expander and is expanded and latched in both AL and BH. Both high and low parts are used in case the effective operation is subtraction, and then A–B and B–A would be calculated in parallel. Also in cycle 1, the exponent is driven to the exponent dataflow, and the exponent difference is calculated. In cycles 2 and 3, the BCD data in AL and BL is added in the low part of the adder, and AH and BH in the high part of the adder. The carries are not propagated between each 18-digit portion of the adder because this is a doubleword operation. The result is latched in WH and WL. In cycle 4, WL drives the compressor of BCD to DPD format and then is latched in the CL register. In cycle 5, the CL register can drive the result to the FPRs.

A carryout of the adder will require an additional cycle to shift the data right one digit and possibly additional cycles for rounding. Subnormal number detection to update the class flags in the FPSCR can also add delay. Additionally, overflow and underflow result in a rebiased exponent.

Case 2—Aligning to the operand with the smaller exponent: When the exponent difference is less than or equal to the number of leading zeros in the operand with the bigger exponent, the operand with the larger exponent can be shifted left to properly align it with the smaller exponent value. For this case the following steps are performed:

1. Expand to BCD and in parallel compare the exponents.
2. Swap the operands, creating two operands called *big* and *small*.
3. Shift the operand with the larger exponent left by the exponent difference.
4. Add aligned *big* to *small*.
5. Round, if necessary.
6. Compress to DPD format.

Table 4 Execution times of addition, subtraction, multiplication, and division.

	Cycles required for execution	
	Doubleword operands	*Quadword operands*
Case 1 add/sub	9 to 13	11 to 15
Case 2 add/sub	11 to 15	13 to 17
Case 3 add/sub	13 to 17	15 to 19
Multiplication	$19 + N$	$21 + 2N$
Division	82	154

Note: N is the number of digits in the first operand excluding leading zeros.

Similar to case 1, no rounding is necessary unless there is a carryout. Results are aligned to the preferred exponent, which is the smallest exponent of the two operands.

Case 3—Shifting both operands: If the exponent difference is greater than the number of leading zeros in the operand with the bigger exponent, then both operands are shifted. This could be avoided if the adder is $2p$ digits wide, but this is prohibitive to implement for the 34-digit format. The steps are the following:

1. Expand to BCD and, in parallel, compare the exponents.
2. Swap the operands.
3. Shift the operand with the larger exponent left by the exponent difference (D).
4. Reshift the operand with the larger exponent left by the number of leading zeros in its coefficient (Z).
5. Compute $D - Z$ and shift the operand with the smaller exponent right by the result.
6. Add the now-aligned coefficient.
7. Round.
8. Compress the result to DPD format.

These three cases are executed concurrently, and in the event of a conflict, the faster case is given precedence. This is evident for step 3 of case 3, in which we first incorrectly shift left by the exponent difference and then have to reshift by the number of leading zeros. This is done because the leading-zero count is not finished in time to set the shift amount by step 3; the hardware, therefore, assumes it a case 2 and discards the result if it is wrong. For subtraction of a doubleword format, both the high and low adder halves are used, so $A - B$ and $B - A$ cannot be computed in parallel. Instead, they are computed serially, and the result is selected based on the carryout. The number of cycles necessary to complete each of the three cases of addition is shown in **Table 4**.

Multiplication

Multiplication of the decimal floating-point coefficients is performed by a serial sequence of a digit multiplication, a shift, and summations. If P is the product of M times N, where M contains p digits of precision, then

$$P = \sum_{i=0}^{p} (M_i N) \times 10^i.$$

Quadword operands are processed in this straightforward manner and require two cycles for each iteration, necessitated by the two-cycle latency through the adder. To reduce the number of multiples maintained in the partial product creator block of Figure 7, partial products are formed with summations of easy-to-generate multiples 1×, 2×, 5×, and 10× [17]. This provides less storage of multiples, since only 1× needs to be stored and 2× and 5× can be calculated very quickly with the BCD doubler and BCD quintupler. Although the adder is needed to create each required multiple from this subset of multiples, no latency is added to the iteration because these computations can be interleaved in the adder every other cycle with the partial product summations.

For doubleword operands, a technique of summing pairs of partial products is used to reduce the average multiplication iteration to one cycle. This is shown by

$$P = \sum_{i=0}^{p/2} [(M_i N) + 10(M_{i+1} N)] \times 100^i.$$

To accomplish this, the dataflow is split into two separate 72-digit halves. The lower half of the adder alternates between generating multiples $M_i N$ and $10(M_{i+1}N)$, where the multiplication by 10 is a simple one-digit shift left. These multiples are then passed to the upper half of the dataflow. The upper half of the adder interleaves the computations for generating the partial product pairs $(M_i N) + 10(M_{i+1}N)$ and partial product accumulations $P_{i+1} = P_i + 100 \cdot [(M_i N) + 10(M_{i+1}N)]$, where the multiplication by 100 is a simple two-digit shift left. The number of cycles necessary to complete each of the multiplication operations is dependent on the number of significant digits in the first operand and is shown in Table 4.

Division

The divide algorithm chosen for the POWER6 processor design uses a nonrestoring division algorithm with prescaling. Nonrestoring division iteratively generates quotient digits using the following steps:

1. Quotient selection based on a partial remainder q_{sel}.
2. Multiplication of the divisor D by the quotient digit.
3. Computation of the next partial remainder $P_{(i+1)}$, as shown by

$$P_{(i+1)} = P_{(i)} - q_{\mathrm{sel}(i)} \cdot D.$$

Each iteration takes four cycles to complete: one cycle for quotient selection, one cycle for digit multiplication, and two cycles to compute the next partial remainder. Correctly rounding the common case of an inexact quotient requires that the number of iterations to complete the division be one greater than the target precision.

The divisor and numerator are prescaled in such a way that the divisor is greater than 1 and strictly less than 1.11. This simplifies the quotient selection process by simply extracting the most significant digit of the partial remainder. This simplification in the quotient selection process of the division iteration comes with an upfront cost of 12 cycles to complete the prescaling of the numerator and divisor. This prescaling is a multiplication of the numerator and denominator by a two-digit number generated from a 90-entry by 8-bit programmable logic array lookup table.

Multiplication of the selected quotient by the now-prescaled divisor is done by selecting the appropriate multiple of the divisor. To reduce the number of divisor multiples that must be maintained in the partial products creator, quotient selections are made from a redundant set of $\{-5 \text{ to } +5\}$. This is done by adjusting the quotient digits on the fly after they are selected and before they are put into the final result register. This on-the-fly quotient adjustment is done in parallel with the next partial remainder computation and does not affect the critical path in the divisor iteration. Divisor multiples 1×, 3×, and 4× are precomputed and stored in the partial product creator, and 2× and 5× are generated on the fly in the BCD doubler and BCD quintupler logic in the partial product creator block. Table 4 shows the number of cycles necessary to complete the common case for the two division algorithms.

Summary

The VMX unit provides acceleration of graphics and scientific workloads and operates on multiple fixed-point or floating-point operands with a single instruction. Vectors of 128 bits are separated into 16×8-bit, 8×16-bit, or 4×32-bit operands, and the arithmetic is computed in parallel. The architecture is very rich and robust and supports hundreds of instructions. An overview of the unit is given here, including insights into each of its execution subunits. One of the primary challenges of this design was tracking dependencies through the pipelines given the high-frequency requirements. Details of how this challenge was solved are given. Design additions that were necessary to support the POWER6 processor RU are discussed.

The POWER6 processor DFU accelerates financial transactions and provides the first hardware implementation of the IEEE 754R standard decimal data types. Decimal floating point requires alignment and rounding, and it has unnormalized coefficients. The architecture and implementation are robust to adapt to applications or languages that require larger, smaller, or identical precision and exponent range. This supports the varying requirements of decimal floating-point applications. The architecture and implementation are designed to be small but sufficient to accelerate current software implementations. Size is minimized by reusing the large BFU register file and by eliminating the need for a separate load and store interface. Instead, hardware is invested in the most frequent operations, such as format manipulation and basic arithmetic.

The two POWER6 processor accelerators have been discussed in detail. Both are designed at the POWER6 processor clock speed of more than 5 GHz with a technology-independent cycle time of 13 FO4. The POWER6 processor design is targeted for performance, and the VMX and DFU add two application-specific accelerators to boost both scientific and commercial transaction performance.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Freescale Semiconductor, Inc., Sun Microsystems, Inc., or Sony Computer Entertainment, Inc., in the United States, other countries, or both.

References

1. Freescale Semiconductor, *AltiVec™ Technology Programming Environments Manual*, 2006; see *http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf*.
2. M. S. Schmookler, M. Putrino, C. Roth, M. Sharma, A. Mather, J. Tyler, H. Van Nguyen, M. N. Pham, and J. Lent, "A Low-Power, High-Speed Implementation of a PowerPC™ Microprocessor Vector Extension," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, 1999, pp. 14–16.
3. M. M. Ziegler and M. R. Stan, "A Unified Design Space for Regular Parallel Prefix Adders," *Proceedings of the Conference on Design, Automation and Test in Europe*, Paris, France, 2004, pp. 1386–1387.
4. N. Mäding, J. Leenstra, J. Pille, R. Sautter, S. Buttner, S. Ehrenreich, and W. Haller, "The Vector Fixed Point Unit of the Synergistic Processor Element of the Cell Architecture Processor," *Proceedings of the 31st European Solid-State Conference*, Grenoble, France, 2005, pp. 203–206.
5. S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener, "POWER6 Binary Floating-Point Unit," *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH18)*, Montpellier, France, 2007, pp. 77–86.
6. *ANSI/IEEE Standard 754-1985*, "IEEE Standard for Binary Floating-Point Arithmetic," ©1985 IEEE; see *http://754r.ucbtest.org/standards/754xml.html*.
7. S. M. Mueller and W. J. Paul, *Computer Architecture: Complexity and Correctness*, Springer-Verlag, Berlin, Germany, 2000, pp. 351–436.

8. J. Gosling, B. Joy, and G. Steele, *The Java™ Language Specification*, Addison-Wesley, Boston, MA, 1996.
9. E. M. Schwarz, “Binary Floating-Point Unit Design: The Fused Multiply–Add Dataflow,” *High-Performance Energy-Efficient Microprocessor Design*, V. G. Oklobdzija and R. K. Krishnamurthy, Eds., Springer, Dordrecht, The Netherlands, 2006, pp. 189–208.
10. X. Y. Yu, Y.-H. Chan, M. Kelly, E. Schwarz, B. Curran, and B. Fleischer, “A 5GHz+ 128-bit Binary Floating-Point Adder for the POWER6 Processor,” *Proceedings of the European Solid-State Circuits Conference*, Montreux, Switzerland, 2006, pp. 166–169.
11. H.-J. Oh, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, et al., “A Fully Pipelined Single-Precision Floating-Point Unit in the Synergistic Processor Element of a Cell Processor,” *IEEE J. Solid-State Circuits* **41**, No. 4, 759–771 (2006).
12. ANSI/IEEE,“DRAFT Standard for Floating-Point Arithmetic P754,” Draft 1.2.5, see “Working Group Records,” at *http://754r.ucbtest.org/*.
13. M. F. Cowlishaw, “Decimal Floating-Point: Algorism for Computers,” *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, 2003, pp. 104–111.
14. M. Cowlishaw, “Densely Packed Decimal Encoding,” *IEE Proceedings—Computers and Digital Techniques* **149**, No. 3, 102–104 (May 2002).
15. *ANSI/IEEE Standard 854-1987*, “IEEE Standard for Radix-Independent Floating-Point Arithmetic,” ©1987 IEEE; see *http://754r.ucbtest.org/standards/854xml.html.*
16. E. M. Schwarz and S. Carlough, “POWER6 Decimal Divide,” submitted to the 18th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Montreal, Canada, July 2007.
17. R. K. Richards, *Arithmetic Operations in Digital Computers*, D. Van Nostrand Company, Inc., New York, 1955, pp. 247–285.

Received January 17, 2007; accepted for publication February 21, 2007; Internet publication October 23, 2007

Lee Eisen *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (leisen@us.ibm.com)*. Mr. Eisen is a Senior Technical Staff Member in the high-performance processor design team. He received a B.S. degree in electrical engineering from Texas A&M University. He has worked on the PowerPC 602, PowerPC 603*, PowerPC 603ev, PowerPC 750* (all Somerset Design Center), POWER4*, POWER4+*, and PowerPC 970 processors. Mr. Eisen was the lead power engineer, VMX leader, and core hardware bring-up lead for the POWER6 processor design.

John Wesley (Wes) Ward III *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (wesward@us.ibm.com)*. Mr. Ward is an Advisory Engineer in the high-performance processor design team. He received a B.S. degree in electrical engineering from the University of Texas. Mr. Ward has worked on verification, timing, and logic design on numerous processor core designs—including Power PC 620*, POWER4, POWER4+, PowerPC 970, and POWER5* processors. For the POWER6 processor design, his responsibilities included design and implementation of the VMX issue queue and instruction fetch unit (IFU) prefetch logic.

Hans-Werner Tast *IBM Systems and Technology Group, Schoenaicherstrasse 220, D-71032 Boeblingen, Germany (tast@de.ibm.com)*. Mr. Tast received a Dipl.-Ing. degree in electrical engineering from the Fachhochschule Ulm, Germany. Mr. Tast was the Logic Design Leader of the cache systems for several generations of IBM zSeries* processors. He worked on the Cell Broadband Engine** (Cell/B.E.) processor synergistic processor elements (SPEs) and the POWER6 processor VMX unit. In 2004, he assumed a lead position in the concept of a new level of cache hierarchy for future zSeries processors.

Nicolas Mäding *IBM Systems and Technology Group, Schoenaicherstrasse 220, D-71032 Boeblingen, Germany (nmaeding@de.ibm.com)*. Mr. Mäding is currently an Advisory Development Engineer for vector fixed-point unit developments. He received an M.S. degree from the Technical University of Chemnitz, Germany. He worked in several areas of the Cell/B.E. processor development, including logic design, integration, and system bring up. He later worked on the vector fixed-point execution unit of the POWER6 processor VMX. His interests are in computer architecture, high-frequency design, low power, and design for testability and reliability.

Jens Leenstra *IBM Systems and Technology Group, Schoenaicherstrasse 220, D-71032 Boeblingen, Germany (leenstra@de.ibm.com)*. Dr. Leenstra received an M.S. degree from the University of Twente and a Ph.D. degree from the University of Eindhoven, both of The Netherlands. He has worked in several areas of development, including logic design and verification of I/O chips, multiprocessor system verification of the IBM S/390* G2 and G3 mainframe computers, the Cell/B.E. processor SPEs, and the POWER6 processor VMX unit. He is currently working on next-generation IBM microprocessors. Dr. Leenstra's current interests focus on computer architecture, high-frequency design, low power, and design for testability.

Silvia M. Mueller *IBM Systems and Technology Group, Schoenaicherstrasse 220, D-71032 Boeblingen, Germany (smm@de.ibm.com)*. Dr. Mueller is a Senior Technical Staff

Member in the high-performance processor design team. She received B.S. degrees in mathematics and computer science, an M.S. degree in mathematics, and a Ph.D. degree in computer science from the University of Saarland, Germany. In 1998, she became a Privatdozent at the computer science department of the University of Saarland and still holds a teaching assignment there. Dr. Mueller joined IBM at Boeblingen in late 1999. From 2001 to 2003 she was on an international assignment in Austin, Texas, joining the Sony–Toshiba–IBM Design Center developing the Cell/B.E. processor. She led the team for the floating-point units for the Cell/B.E. processor and for the POWER6 processor VMX.

Christian Jacobi *IBM Systems and Technology Group, Schoenaicherstrasse 220, D-71032 Boeblingen, Germany (cjacobi@de.ibm.com)*. Dr. Jacobi received an M.S. degree (Diplom-Informatiker) and a Ph.D. degree, both in computer science, from Saarland University, Germany. He has worked on formal verification techniques, mainly for FPUs, floating-point logic design, logic verification, and physical implementation for various IBM microprocessors. He is now working on cache designs for future zSeries processors.

Jochen Preiss *IBM Systems and Technology Group, Schoenaicherstrasse 220, D-71032 Boeblingen, Germany (preiss@de.ibm.com)*. Dr. Preiss is a Staff Hardware Engineer for FPUs in the microprocessor design team. He received a B.S. degree in mathematics and M.S. and Ph.D. degrees in computer science, all from the University of Saarland, Germany. For his master's thesis, he received the Guenther Hotz Award. Dr. Preiss worked on the development of the PowerPC 970 series IFU and the POWER6 processor VMX FPU.

Eric M. Schwarz *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (eschwarz@us.ibm.com)*. Dr. Schwarz is a Distinguished Engineer in IBM zSeries, iSeries*, and pSeries* processor development. He received a B.S. degree in engineering science from the Pennsylvania State University and M.S. and Ph.D. degrees in electrical engineering from Ohio University and Stanford University, respectively. He worked on the successors to the 4381 and 9370 computers as well as on the IBM G4, G5, G6, z900, z990, z9*-109, and POWER6 processor-based computers. He led the FPU development for these computers and was Chief Engineer of the IBM z900 system in 2000. Dr. Schwarz is active in the IEEE Symposium on Computer Arithmetic and has been on the program committee since 1993.

Steven R. Carlough *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (scarloug@us.ibm.com)*. Dr. Carlough is a Senior Engineer in zSeries, iSeries, and pSeries processor development. He received B.S. and M.S. degrees, both in electrical engineering, and a Ph.D. degree in electrical engineering, all from Rensselaer Polytechnic Institute. He has worked on fixed-point units for z990 and z9-109 servers and the decimal FPU in the POWER6 processor-based computer.