Design considerations for the PowerPC 601 microprocessor

by M. T. Vaden
L. J. Merkel
C. R. Moore
T. M. Potter
R. J. Reese

The PowerPC 601™ microprocessor (601) is the first member of a family of processors that support IBM's PowerPC Architecture™. The 601 is a general-purpose processor based on a superscalar design point. As with any development effort, the 601 development program had several different, often conflicting, design goals. The most important requirements were support for the PowerPC Architecture, a short development cycle, competitive performance and cost, compatibility with existing POWER applications, and support for multiprocessing. This paper describes several aspects of the 601 design and discusses some of the design trade-offs considered in those areas.

Introduction—design goals and fundamental design decisions

The PowerPC 601™ microprocessor was developed as part of the PowerPC™ alliance between IBM, Motorola, and Apple. The original agreement specified an initial "road map" calling for the development of four microprocessors: the 603, for low-end desktop and portable computers; the 604, for desktop computers and low-end servers; the 620, for high-end servers; and the 601, which was intended to provide a competitive PowerPC processor to the marketplace very quickly [1].

The road map (**Figure 1**) established the basic ground rules and goals for the 601 design point: Implement the PowerPC Architecture™, bring it to market as quickly as possible, offer competitive performance and features, and make it cost-effective. These four goals formed the basic backbone for all design decisions related to the chip. Some of these decisions were made early and constitute fundamental principles that formed the core structure of the design. Others were made during the design process, and represent interesting trade-offs among various design alternatives.

The first goal of the 601 was to implement the PowerPC Architecture. In general, the PowerPC Architecture [2] was derived from the IBM POWER architecture [3]. Changes were made to add key missing features and to enable more efficient implementations by eliminating some instructions and relaxing the specifications of less significant "corner" cases [2].

In addition to implementing the PowerPC Architecture, the 601 was also required to support the user-level environment of the POWER Architecture™. This was necessary to provide a temporary bridge for the software development team as they migrated from designs providing the full POWER Architecture support to the new implementations of the PowerPC Architecture.

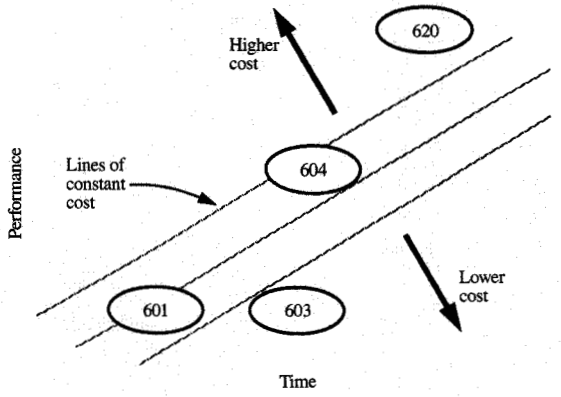


Figure 1

"Road map" of the PowerPC processor strategy. The locations of the processors on the plot show *qualitative*, not quantitative, relationships.

Table 1 PowerPC 601 microprocessor performance targets.

Benchmark	*50 MHz*	*66 MHz*
SPECint92	45	60
SPECfp92	60	80

The second and probably the most challenging goal was the requirement for a very short development cycle. Through customer negotiations and careful consideration, we set a goal of achieving working 601 modules by September 1992—just one year for a design cycle that included design, verification, and fabrication. To achieve this objective, several fundamental decisions were made early in the project. First, we decided to exploit a proven CMOS process technology to reduce manufacturing risk (a 0.6-μm minimum feature size and four levels of metal). Second, we chose to leverage the same structured custom design methodology that had been used successfully for the development of several previous generations of processors. This methodology blends the productivity advantage of design automation tools with the ability to address difficult problems with full custom design. The combination reduces the time and risk associated with the development of such complex devices. Third, we decided to take advantage of technology that existed at both IBM and Motorola. From IBM, the RISC single-chip (RSC) design served as a starting point for the 601 [4]. Although significant changes were made to this base design to achieve the performance and feature goals of the project, much of the logic in the fixed-point and the floating-point units was reusable. In addition, we were fortunate to retain much of the original RSC design team to work on the 601 processor. From Motorola, we borrowed many of the bus protocol concepts defined in the MC88110 processor as a starting point for the 601 bus interface.

A third objective of the 601 project was to offer competitive performance [5]. In order to achieve this, the processor core employs a superscalar machine organization with three execution units. These units can operate concurrently, so that up to three instructions can be executed in each processor cycle. The processor also includes an integrated 32KB unified cache, a high-performance bus interface [6], and support for multiprocessing. These features provide high-bandwidth access to memory and efficient support for cooperative memory sharing. **Table 1** summarizes the key performance goals of the 601.

The final objective of the 601 design was low cost. This was achieved primarily by the selection of a high-volume process and by minimizing the chip area. In addition, the cache on the 601 chip includes redundancy to increase the effective manufacturing yield. A fully static LSSD design approach is employed to achieve very high testability and accurate failure diagnostics. The cost of the 601 module was further reduced through the use of an economical package.

Figure 2 is a high-level block diagram of the PowerPC 601 processor, which illustrates how the different functional units interface with one another. The major elements are the three execution units (branch unit, fixed-point unit, and floating-point unit), the fetch and dispatch unit, the cache, and the bus interface unit (BIU). The BIU interfaces primarily with the cache. All data entering or leaving the chip do so via the BIU and cache. An eight-word bus from the cache feeds the instruction fetch and dispatch unit at a rate of up to eight instructions per processor clock. The dispatch unit has three unique buses for dispatching instructions—one for each execution unit. Two words of the eight-word cache data bus feed load data to the floating-point unit; one word feeds load data to the fixed-point unit. For store operations there is a corresponding data bus to the cache from the floating-point and fixed-point units.

Cache unit

Central to the design of the 601 is the cache, which is shown in a block diagram in **Figure 3**. At a high level, the 601 cache is a 32KB copy-back cache, and is used to hold both instructions and data. It has an eight-way set-associative organization and uses an LRU replacement policy. The line size is 64 bytes, which is divided into two 32-byte sectors. The cache contains a single read/write port which is the main access point for all cache operations. Cache operations can be initiated by the fixed-

point unit, the floating-point unit, the instruction fetcher, the bus interface unit, and the bus snooping* unit. Cache arbitration logic determines which requestor receives access to the cache on each cycle. The tag directory is accessed in parallel with the cache, and contains information used to determine whether the requested address currently resides in the cache. The cache directory has two access ports. The first one operates in synchronization with the main cache access port and can perform a read and a write each cycle. The second port is a read-only port which is used by the snoop unit to determine whether or not a particular address observed on the bus interface exists in the cache. In the event that it does, the snoop unit arbitrates for access to the main read/write port in order to perform the necessary state change and/or copy-back operation required by the coherency protocol. The trade-offs associated with the final design are described in the subsections that follow.

• *Cache size, organization, and geometry*

Many different cache structures have been used in the industry. The most obvious parameter that varies is the cache size itself. In general, the larger the cache, the better the processor performs; however, a larger cache also takes up more of the available circuit area for a particular die size. A second design consideration is the overall organization of the cache. A design can combine instructions and data into a single unified cache structure, or it can use separate, or split, caches for instructions and data. Split caches tend to provide higher bandwidths, since both an instruction and a data access can occur on the same cycle, but unified caches are more area-efficient. The associativity of the cache can also be varied [7]. In general, greater associativity increases the effective hit rate and reduces the probability of cache thrashing (which occurs when many neighboring accesses lie in the same congruence class), but caches with greater associativity can present difficult timing problems to the design. Line size is a fourth design consideration. Larger line sizes typically have greater hit rates; however, as the line size increases, the need for additional interface bandwidth also increases. This can increase the trailing-edge effect and reduce performance.

Although much of the 601 logic evolved from the RSC design, the RSC cache was not adequate to support the 601 performance goals; a larger cache was needed in order to lower the cache miss rate. The fact that larger caches yield lower miss rates is revealed by the measurements

*Snooping is a method of maintaining data coherency when there are several different memory locations in a system where the same data could reside, such as main memory, processor caches, and I/O device buffers. To maintain coherency, each device "listens" to the bus and follows a certain protocol to guarantee that only one device can modify memory at a time, and that when a given device requests data, it will receive the most recent copy [7]. There is more discussion of snooping in subsequent sections of this paper.

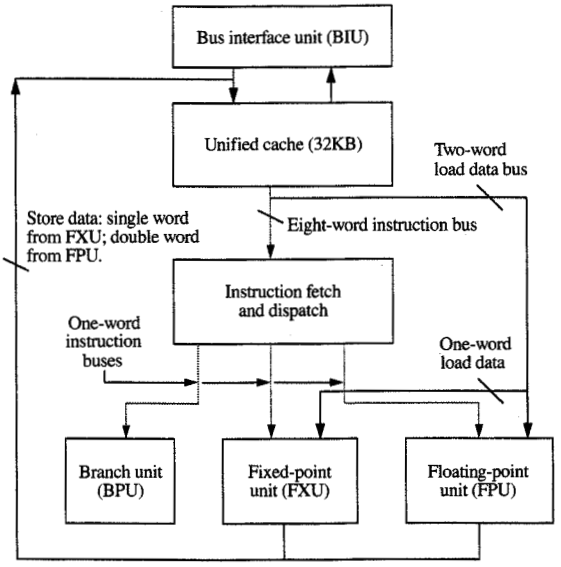
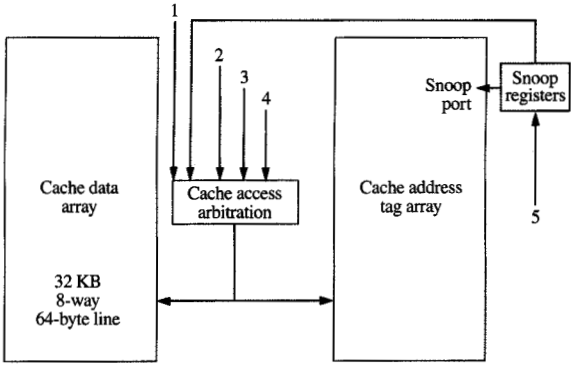


Figure 2

Block diagram of the PowerPC 601 processor showing the relationships among the major functional units.



1 Cache reload operations from the 601 bus
2 Floating-point stores
3 Fixed-point requests, including floating-point loads
4 Instruction fetches
5 From 601 address bus

Figure 3

PowerPC 601 microprocessor cache block diagram showing cache arbitration.

Table 2 Cache miss ratios related to the design of the 601 cache [8].

Cache configuration	*Overall miss rate*
Combined, 8 KB, 64-byte line, two-way set-associative (RSC-like)	0.0342
Split, 16KB inst. and 16KB data, 64-byte line, eight-way set-associative*	0.0109
Combined, 32 KB, 64-byte line, eight-way set-associative (601 cache)	0.0091

*The miss rates for a split instruction and data cache were combined assuming a 0.3 ratio of data accesses to instruction accesses.

shown in **Table 2**, which were made [8] against the SPEC™ benchmark suite. The RSC employed an 8KB, write-through, combined instruction/data cache. As Table 2 shows, moving from an RSC-like cache to a 601-like cache lowers the miss ratio from greater than 3% to less than 1%. Lower miss rates correspond to reduced cycles per instruction (CPI) averages for the machine and therefore higher overall performance.

We chose to use a unified cache structure (in which instructions and data both reside in a single cache) for two primary reasons. First, unified caches require less silicon area for a particular cache size than does a split cache organization. This was a very important factor in achieving the 601 die size. Second, a unified cache has slightly better performance in some cases, as shown in Table 2. This is because the combined instruction/data cache can automatically adjust for the varying demand of instructions versus data. As a result, the applications can effectively see larger available cache space than they do for the split cache organization. One serious drawback of a unified cache is that the available bandwidth is effectively halved (only one port for both data and instruction accesses). Several features were added to the design to compensate for this lost bandwidth. First, the access width from the cache was increased so that up to eight words could be fetched from the cache on each cycle. This additional bandwidth is especially important for instruction fetching, since the superscalar execution units can execute several instructions each cycle during peak operation. The additional bandwidth was also necessary because instruction fetches were prioritized low in the arbitration scheme. Another design feature allows the cache to perform a complete read-modify-write every cycle. This permits stores to execute using only one cycle of cache bandwidth. Finally, to prevent arbitration decisions from stalling instruction execution, queueing is provided between the cache and the execution units. These queues are used to temporarily hold lower-priority cache access requests while other, higher-priority cache operations take place.

We also chose to divide the 64-byte cache line into two sectors of 32 bytes each. The sectoring satisfies the micro-architectural requirement of 32-byte coherency size and also reduces the trailing-edge effect. The selection of an eight-way set-associative organization allows indexing into the cache using only address bits that are not translated. (Translation occurs on 4KB page boundaries. The bits that index into the cache come from the offset within the page.) This eliminates the difficult timing problem of translating a virtual address bit and attempting to index into the cache in the same cycle with that bit. **Figure 4** shows the pipeline for loads and stores and shows that the portion of the address used to index into the cache and cache tags does not go through the memory management unit to be translated.

As another design consideration, caches can be blocking or nonblocking after a miss is detected: Blocking caches do not allow further accesses to occur, while nonblocking caches do allow such accesses. Obviously, nonblocking caches have the ability to perform better if programmers choose to make use of the function by interleaving accesses to different addresses (so that if one is a miss, the other one has some chance of being a hit, since it references a different cache line). However, nonblocking caches are more complicated, which can lengthen the development cycle. The 601 was designed to be one-level nonblocking: One miss of a given type (load, fetch, or store) can be outstanding without blocking the cache, but a second miss of that type does block the cache. This requires only one target register to be held and one datum to be held, which is logically simple to manage and requires very few resources. Performance is increased by allowing cache hits to be processed after the first miss. Once a second miss occurs, the cache access point is occupied with the miss, and the cache becomes blocking.

• *Cache coherency*

The cache was also required to be coherent with respect to other caches in the system and main memory. The 601 bus interface was largely derived from the bus interface on the Motorola MC88110 microprocessor. This interface features a bus snooping mechanism as the memory coherency control mechanism, which allows multiple processors and other devices to cooperatively share system memory.

To achieve coherency, the 601 cache implements the standard MESI protocol (with states of modified, exclusive, shared, and invalid). The cache MESI state is maintained on a 32-byte line sector, rather than a whole line, to match the coherency size defined for the 601 bus interface. The addresses of operations on the 601 bus interface are snooped (monitored) and compared to the contents of the cache (and associated queues), and appropriate measures (as specified by the MESI coherency

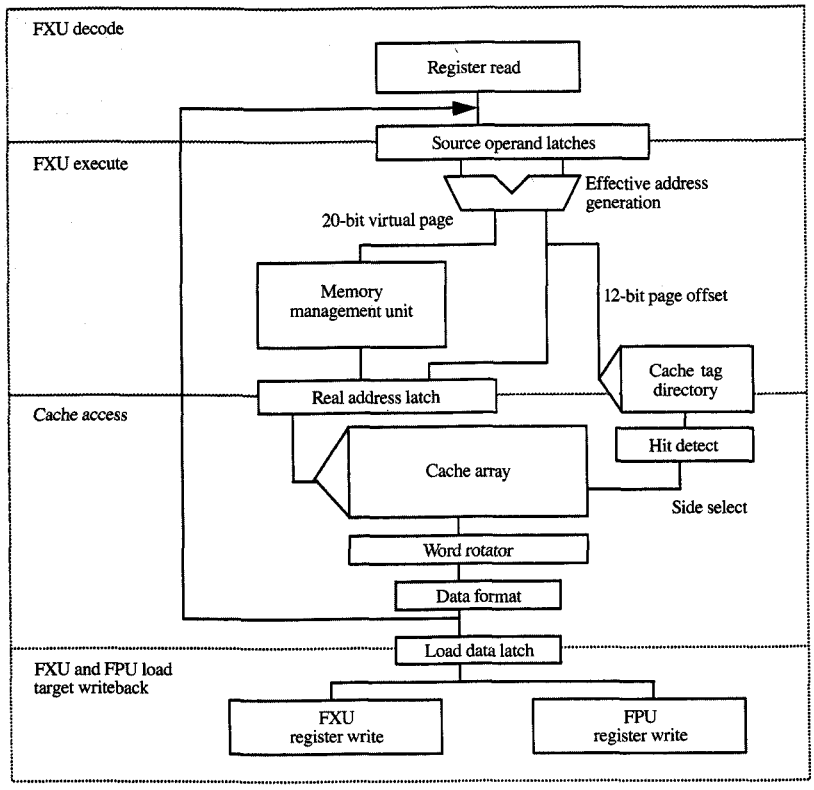


Figure 4

Load/store pipeline including the FXU, MMU, and cache.

protocol) are taken to ensure that the most recent copy of the data is provided to the requester, whether the requester is the local processor, or another processor or device on the 601 bus. To provide this function, the 601 cache was equipped with a second port (read-only), to which the snoop address has exclusive access. Processor activity continues through the normal (read–write) port, and is interrupted only when a snoop hit requires a state change in the cache or requires data to be pushed from the cache.

Fetcher and branch units

One of the areas of the chip which was significantly affected by the structure of the cache was the instruction flow logic, including the fetcher, dispatcher, and branch processing unit. Because the cache is unified, it was necessary to ensure that a continuous stream of instructions can be supplied to the dispatcher without having to access the cache every cycle. The fetcher accomplishes this with an eight-entry instruction queue between the cache and the execution units. This queue is fed by an eight-word bus from the cache so that up to eight sequential instructions can be fetched in a single cycle. The instruction queue can supply the dispatcher for several cycles, even at the peak dispatch rate of three instructions per cycle. A block diagram of the instruction queues is shown in **Figure 5**.

The term dispatch queue refers to the bottom four positions of the instruction queues. Branch and floating-point instructions can be dispatched from all four positions in the dispatch queues. Up to three instructions can be dispatched in a single processor cycle (a maximum of one to each execution unit). The branch processor executes the four main branch instructions (b, bc, bclr, and bcctr), including the absolute and link update forms of those instructions. Floating-point arithmetic operations are dispatched to the FPU; all other instructions are dispatched to the FXU.

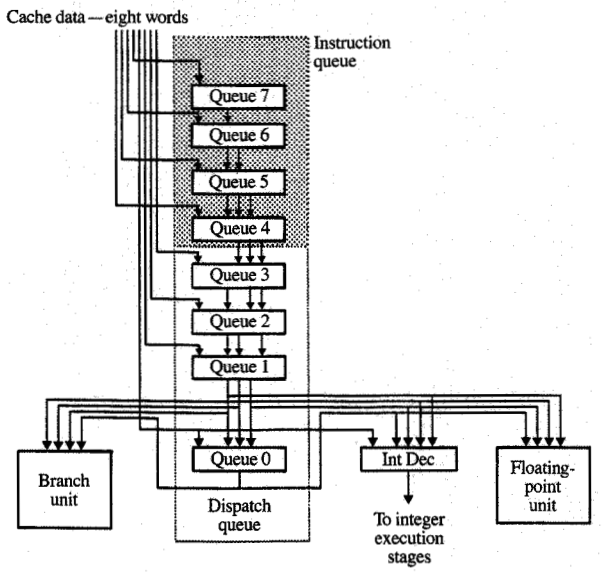


Figure 5

Block diagram of the data paths to and from the instruction queues and dispatch queues.

Branches can cause an increase in the cache bandwidth required by the fetcher. The 601 microprocessor cannot afford to fetch both the target and the sequential paths of a branch because of both cache bandwidth constraints and die size constraints. For performance reasons, the 601 cannot afford to simply stop fetching instructions when it reaches a conditional branch; thus, the 601 employs branch prediction in order to process branches efficiently. In order for branch-prediction mechanisms to help performance, they must either be very accurate or have a short recovery time when branches are mispredicted. Dynamic branch-prediction algorithms tend to be more accurate than static algorithms, but require large branch history tables in order to work. Because of the die size constraint on the 601, it was necessary to use a static prediction mechanism and to concentrate on reducing the misprediction recovery time. Another advantage of this approach is that it depends less on program behavior and is less subject to anomalous behavior.

The 601 branch processor uses a compiler-assisted branch-prediction mechanism with which the compiler can set a bit in the branch instruction to tell the processor whether the branch is likely to be taken or not taken. This mechanism achieves an accuracy which is similar to that of many dynamic schemes but uses significantly less hardware. Another advantage of this mechanism is that it allows the 601 to predict branches very quickly.

The 601 dispatcher can dispatch branches from any of the bottom four elements of the instruction queue. Branches can be dispatched ahead of their program order with respect to the other instructions in the dispatch queues. For example, a branch that follows an FXU instruction in program order can be dispatched before the FXU instruction is dispatched. (Since FXU instructions can be dispatched only from IQ0, this out-of-order dispatch occurs frequently). Branch dispatch, decoding, execution, prediction, and cache arbitration for the target instruction all occur on the same cycle. On the next cycle, branch target instructions are available from the cache (assuming that the fetch address has been placed in the cache and not been preempted by a higher-priority cache access). This quick turnaround of branches, in addition to the early dispatch of branches, helps contain the unified-cache contention problem by getting branch target fetches to the cache early enough that if contention does occur, the queue may still have instructions available for the dispatcher. Simulation has shown that in most cases, the instruction queue does not drain completely before a fetch can refill it. The one notable exception to this is when a stream of memory access instructions occurs (e.g., eight load instructions). In this case, the memory access instructions all access the cache before a fetch can access it (memory access instructions always have higher priority than fetches), so the instruction following the last memory access instruction cannot be fetched until the last memory access instruction has cleared the integer execute stage.

To minimize the impact of branch misprediction, the 601 processor was designed to have a fast misprediction recovery mechanism. Several key design features support misprediction recovery: condition register coherency checking, condition register forwarding, same-cycle prediction resolution, and delayed instruction queue purging. These features are now described in detail.

In order to resolve branches as soon as the condition register dependency is resolved, the pipeline control logic scans all instructions ahead of the branch, looking for condition register dependencies, and then signals the branch unit when the condition register is coherent. On the cycle when the condition register is coherent, the branch unit checks its prediction; if it is correct, it can predict another branch the next cycle. If the branch prediction is incorrect, any instructions from the predicted path are purged, and the fetch address is changed to the correct path.

One of the most important performance issues for branch processing is the time required to execute a compare instruction and get the results to the branch processing unit in order to resolve a dependent conditional branch. We refer to this time as the compare-branch

latency. In the 601, the results of compare instructions are forwarded to the branch processing unit from the integer execution stage. The integer unit contains a specialized compare unit (see **Figure 6**) which quickly calculates the results of a compare instruction and forwards the results to the branch processor. If the condition register is coherent, the dependent conditional branch is resolved on the same cycle. If the branch has been mispredicted, the request for the correct target address can be sent to the cache, also on the same cycle as that in which the compare was evaluated.

A case of interest is when the condition register is already coherent for a conditional branch which is being predicted. In this case, the 601 can actually resolve the branch in the same cycle in which it is being predicted. The branch processor can then execute another conditional branch on the next cycle. Even if the branch is mispredicted, there is no penalty.

Another way in which the 601 minimizes the penalty for mispredicted branches is through its purging mechanism. When a branch is predicted as taken, there are typically several instructions in the instruction queue from the sequential path. There is also a delay of at least one cycle (possibly more) before target instructions for the branch arrive at the instruction queue. The 601 does *not* purge the sequential instructions when a branch is predicted as taken. If the branch is determined to have been mispredicted before the target instructions come from the cache, the sequential instructions are kept, and the target instructions are thrown away. This is more efficient than immediately purging the sequential instructions and then having to refetch them after the branch has been resolved. If the target instructions arrive at the instruction queue before the branch is resolved, they overwrite the sequential instructions which have been left in the queue.

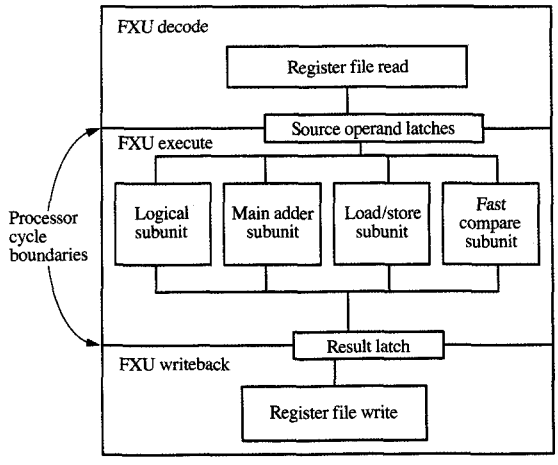


Figure 6

PowerPC 601 fixed-point unit (FXU) pipeline. The execute stage is divided into separate subunits.

• *Instruction address translation mechanism*

The instruction fetcher uses a four-entry, fully associative translation cache referred to as the translation shadow array (TSA). The TSA is used to provide fast translation ability to the fetcher without dual-porting the unified TLB and segment registers. Each TSA entry is capable of holding one translation object, either a block or a page translation.

The TSA behaves as a simple associative memory; a hit is determined by a comparison of the effective page of the address with the effective pages which are contained in the TSA. When a hit occurs, the real page number is returned—there is no knowledge of the actual translation mechanism imbedded in the TSA. In order to support variable page sizes, the comparators used for the effective page numbers have masks which cause only the appropriate bits to be compared (this function is trivial to add to a logical comparator).

As with any cache, coherency is a significant concern. The TSA is kept coherent with the page tables and segment registers by hardware. Whenever an instruction which affects the state of the virtual memory management subsystem (such as a TLB invalidate) occurs, the TSA is completely purged. When a tlbi is snooped on the bus, the tlbi address is sent to the fetcher, which purges the entire TSA and then forwards the snooped tlbi to the memory management unit. With these two mechanisms, hardware ensures that no changes can occur to the virtual memory environment without the TSA being updated (i.e., the TSA is completely coherent, and therefore invisible to the programmer).

• *Cycle-time constraints*

The 601 design point had several design constraints on it, including a performance and cycle-time constraint. The worst-case cycle time for the 601 was 20 ns, with nominal cycle time of 15 ns. With the short design cycle allotted for implementation, every design was carefully studied to see that cycle-time requirements would be met.

A potential timing problem in the instruction queue and dispatch logic was the ripple-hold effect, which is caused by stalls in lower pipeline stages that must hold upper pipeline stages. It was not reasonable for stalls in any of the execution units to cause a hold in the instruction queue, yet still expect the 601 to meet its targeted cycle

time. Three different solutions were employed to solve this problem, one for each execution unit.

The branch unit had the simplest solution. Since it is a single stage, checking all dependency information at dispatch time prevents most stalls from occurring. The only stalls that can occur involve dependencies on the condition register. These paths were carefully hand-tuned.

The floating-point unit has a separate queue element before its decode stage. This element is used to buffer the dispatch stage from stalls in the floating-point unit. Dispatch is held if and only if the floating-point queue is full. The floating-point queue-full signal is available at the start of the cycle, so it is not a problem for instruction dispatch and queue movement cycle timing.

The integer unit decode stage can be loaded directly from the cache or from the instruction queue. It is loaded in parallel with queue 0 (IQ0—the lowest element in the instruction queue), with the same instruction most of the time. However, in order to isolate the instruction queue movement from the movement of instructions in the FXU, IQ0 is allowed to advance on the first cycle in which FXU decode is held (i.e., FXU decode stalls). In the next cycle, IQ0 and FXU decode contain different instructions. Once they contain different instructions (a fact known at the beginning of the cycle), IQ0 is held if it contains an FXU instruction. When FXU decode is no longer held, it is loaded with the instruction in IQ0 (or, if IQ0 is empty, the instruction moving into IQ0). Now decode and IQ0 contain the same instruction again, and processing continues as usual. Thus, the ripple hold from the integer unit is broken at the interface between integer decode and the bottom of the dispatch queue.

Fixed-point unit (FXU)

The 601 fixed-point unit (FXU) is a single execution unit that handles all of the integer arithmetic, logical, rotate, and shift instructions. In addition, loads, stores, and cache control instructions are partially implemented in the FXU and partially implemented in the cache and memory subsystem. The FXU (Figure 6) is divided into subunits, each of which has logic dedicated to executing a particular class of FXU instructions. Add, subtract, multiply, and divide instructions are implemented in the main adder subunit. The logical subunit implements rotate, shift, and logical instructions. The fast-compare subunit is dedicated to comparing two operands quickly, so that early results can be forwarded to the branch unit for conditional branch resolution. The load/store subunit has logic which is used for calculating the effective address, control for unaligned accesses, control for string instructions and load/store multiple instructions, and interrupt detection for alignment interrupts and data storage interrupts. Some logic, such as the MQ register and the state counter for multicycle operations, is shared across subunits. These subunits are not separate execution units that can concurrently execute instructions. Only one instruction can be in FXU execute at a time. For example, if there is a load in FXU execute, the logical subunit and fast-compare subunit are idle.

Combining all of these functions in one functional unit simplifies the FXU design in several ways. First, there are no synchronization problems among FXU instructions. For example, the completion ordering between an add instruction and a load instruction is implicit in the 601, since they are dispatched to the FXU in program order and executed in order. If loads were implemented in a separate load/store execution unit, complicated logic would be needed to determine completion ordering. Knowledge of the instruction order at completion is required because the PowerPC Architecture supports a precise interrupt model for most types of interrupts. Second, the evaluation of register hazards becomes more complex when there are multiple execution units that read and write the same register file. New problems arise; for example, when there is more than one execution unit, the difficulty of maintaining proper instruction ordering is increased, as is the number of result-forwarding paths.

In addition, there is an overall saving of space when all of these functions are combined into one execution unit; certain logic elements can be shared in the merged 601 FXU that would have to be duplicated in a design with multiple execution units for FXU instructions. For example, the state counter for the multiply and divide instructions can also be used as the state counter for string loads and stores in the 601 FXU.

The penalty for using a single execution unit for all of these functions is that these instructions cannot be executed concurrently, thus reducing the instructions per clock figure of merit for the 601. However, it was felt that the overall performance objectives could be met without using separate FXU execution units. Given that belief, the decision was made to use the single FXU unit—a simpler design which would require less chip area and reduce the time to market. The next section introduces the 601 FXU design and presents some of the key design trade-offs.

- *Arithmetic and logical instructions*

Arithmetic and logical instructions have three stages in the FXU pipeline: the FXU decode stage, the FXU execute stage, and the FXU ALU writeback stage. All of the arithmetic and logical instructions flow through these stages in order; however, the multiply and divide instructions are held in the execute stage for several cycles of processing before moving to the writeback stage. (These instructions are called multicycle instructions, since they remain in execute for more than one cycle.) The decode stage is used for reading the register file and generating constants that are used as instruction source operands. The execute stage has the logic where the actual arithmetic or logical

manipulation of the data is performed. The writeback stage is used for writing the results back to the register file.

As previously mentioned, there are several subunits within the FXU execute stage. It is notable that there is no separate subunit for the multiply and divide instructions, as there was in some previous implementations of the POWER Architecture [3]. The multiply and divide instructions are implemented by holding the instructions in execute for several cycles and iteratively using the FXU main adder complex. Not implementing a separate subunit for the multiply and divide instructions saves chip area at the expense of performance. The following analysis looks at this trade-off in more detail.

The multiply instructions are implemented using a Booth-encoded four-bit step, so that four bits of the multiplier are processed every cycle. It takes nine cycles to complete the multiply for a 32-bit multiplier (one setup cycle and eight multiply steps). Completion in five cycles is supported for multipliers [8] that are in the range $-2^{15} \leqq X \leqq 2^{15} - 1$. Combining the multiply functions in the main adder subunit added hardware to the subunit: a 36-bit CSA, a 36-bit four-way mux (multiplexor), and some additional ports on the right mux and result mux. However, all of this added hardware is considerably less than a separate multiplier would require.

Including the divide instructions in the FXU adds only a single multiplexor port to the dataflow logic. There is a considerable amount of control logic, but that would still be required if implemented in a separate unit.

The performance cost of this approach can be analyzed to the first order by calculating a weighted clocks-per-instruction figure of merit. Examination of dynamic traces from the SPEC benchmark suites reveals that the total frequency of use of FXU multiply instructions is less than 0.4%, and the total frequency of use of all FXU divide instructions is less than 0.04%. The number of execution cycles for instructions other than multiply and divide instructions is one. The number of cycles required for a multiply instruction is usually five and sometimes nine. As a simplifying assumption, an average of seven processor cycles per multiply is assumed. The number of cycles for each divide is 36. Furthermore, we assume that the FXU is fed a continuous stream of instructions. For the 601, the weighted CPI would be

$$\text{Weighted CPI} = 0.9956 * 1 + 0.004 * 7 + 0.0004 * 36$$
$$= 1.038.$$

Now assume that a separate multiply and divide unit can execute multiply instructions in two cycles and divide instructions in 18 cycles. For a design with the separate multiply and divide unit, the weighted CPI would be

$$\text{Weighted CPI} = 0.9956 * 1 + 0.004 * 2 + 0.0004 * 18$$
$$= 1.0102.$$

The cost in performance in terms of weighted CPI for the 601 implementation versus the hypothetical design with a separate multiply and divide subunit is about 2.8%. This loss in performance was traded for smaller chip area.

- *Loads and stores*

Loads and stores are also executed within the FXU on the 601 chip. The execution is pipelined across four stages: the FXU decode stage, the FXU execute stage, the cache access stage, and a register writeback stage for loads only (see Figure 4). During the execute stage the effective address (EA) of the load or store is calculated, and the address is translated by the MMU. Simultaneously, the arbitration for cache access on the next cycle occurs while the load/store is in FXU execute. The translated address (real address) is available at the end of the cycle. In the following cycle, the cache access occurs for both loads and stores. Data are written into the cache for stores on this cycle; data are read from the cache on loads. Load data are also formatted on this cycle as specified by the instruction. For example, load algebraic instructions do the sign extension of the target data at this point. The access of the cache tag directory straddles the boundary between the execute stage and the cache access stage.

The load/store subunit in the FXU has its own 32-bit adder to generate the effective address (EA). Because this adder is dedicated to generating the effective address, it does not have the burden of being surrounded by unrelated logic, and it is therefore faster than the main adder subunit. The dedicated 32-bit EA adder provides the address about one third of the way through the cycle, allowing two thirds of the cycle for translation.

Also, access to the tag directory can start during execute, because the index into the cache and tag is limited to bits within the page offset and need not be translated. This is a benefit of organizing the cache as at least eight-way set-associative. The cycle stealing for access to the cache tag directory and the completion of the translation in execute are key in permitting a two-cycle, pipelined load and store access. Consequently, load target data are available for use by a subsequent instruction with only a single cycle of latency (i.e., an instruction immediately following a load that wants to use the load data stalls for only one cycle). This is true for any fixed-point load; no extra cycles are required for loads with sign extension or byte reversal, because the data formatting is completed in the cache access stage. The cache access and formatting of the data make up one of the longest timing paths in the 601.

- *Support of user-level POWER instructions*

Some user-level POWER instructions were not carried over to the PowerPC architecture; however, these

instructions were implemented on the 601 to provide binary compatibility with the POWER architecture. These include the absolute value instructions (abs and nabs), the difference or zero instructions (doz and dozi), and the MQ instructions.

The abs and nabs were implemented using the rotate/logical subunit and the main adder subunit. The main adder always generated the two's complement of the operand, while the logical subunit just forwarded the operand unchanged. The results were selected according to the sign bit of the original operand.

The difference or zero instructions were implemented using the main adder subunit and the fast comparator subunit. The main adder always provides the difference result. The select between 0's and the difference result is determined from the output of the fast comparator.

While implementing these instructions on the 601 did not have much impact, we must note that implementations with much more aggressive cycle times might have difficulty implementing these instructions as single-cycle instructions.

Floating-point unit (FPU)

The floating-point unit (**Figure 7**) is a pipelined execution unit that implements all of the nonoptional floating-point instructions in the PowerPC architecture. There are four primary stages in the pipeline: decode, multiply, add, and writeback. There is also a queue position before decode that allows the dispatcher to dispatch a floating-point instruction even when the decode stage is busy. Every instruction passes through each stage (from decode to writeback); however, some instructions (such as a double-precision multiply operation) spend more than one cycle in a given stage. Floating-point load instructions are executed in the FXU, however, rather than the FPU, because the FXU has the effective address-generation logic and the interfaces to the memory management unit and cache. Floating-point store operations are jointly implemented in the FPU and the FXU. The FXU provides the address, and the FPU provides the data.

The 601 implementation of floating-point stores deserves closer examination. Floating-point stores have to go through the FXU to generate the effective address (which is calculated from fixed-point GPRs). The data, of course, come from the floating-point FPRs, so floating-point stores are also dispatched to the FPU. Several things were done to avoid synchronizing the two pipelines. First, dispatch of the floating-point store occurs separately (independently) to the FXU and the FPU, and the flow of the instruction through the two pipelines is not interlocked. However, because of the difference in pipeline length and dispatch latency between floating-point instructions and fixed-point instructions, floating-point store data may not be available for cache access when the store address is available. In order to keep from blocking the FXU pipeline with a floating-point store while waiting for the store data, there is a queue between the FXU and the cache that holds the floating-point store request until the data are available from the floating-point unit. This allows the FXU to continue execution of subsequent instructions. When the store data are available from the FPU, the address and control information is taken from this queue and sent to the cache. The presence of a floating-point store in this queue blocks other stores (fixed or floating) from accessing the cache. However, loads (fixed or floating) may "go around" the store in the floating-point store queue as long as they do not reference the same address. This allows loads at the top of a loop to be issued for the next iteration of the loop while the processor is waiting for store data from the FPU. In this way the 601 logic reduces the effect of load latency on the execution of subsequent floating-point instructions.

The 601 FPU also has the ability to compress a floating-point store operation that is storing the result of the previous floating-point instruction. An example of this is shown in the double-precision Linpack loop described in the next paragraph. The fmadd produces a result in F03, and the stfdu places the contents of F03 in memory. For most floating-point operations, an instruction dependent on the result of the previous instruction has to wait in the decode stage until the result is available. However, the 601 can complete the store on the same cycle as the previous instruction, if that previous instruction is generating the data to be stored. This provides the floating-point data sooner, reducing the chance of stalling the FXU pipeline.

The design of the floating-point multiplier involved one of the more significant trade-offs between processor performance and die size. The floating-point multiplier is only wide enough to perform half of a double-precision multiply operation in a single cycle. Therefore, double-precision multiply operations must pass through the multiplier twice in order to complete the multiply. However, a single-precision multiply operation need only pass through the multiplier once and can complete in one cycle. By using a half-wide multiplier, the peak throughput for multiply and accumulate instructions is half of what it would be with the wider multiplier. Lowering the peak throughput degrades the overall floating-point performance, but not as significantly as one might think at first. As an example, consider the execution of the following code sequence (the core Linpack loop):

Double-precision Linpack loop

```
loop:  lfdu   F01,  0x8 (G05)
       lfdu   F02,  0x8 (G04)
       fmadd  F03,  F01,  F02,  F03
       stfdu  F03,  0x8 (G06)
       bc     loop    /* until counter=0 */
```

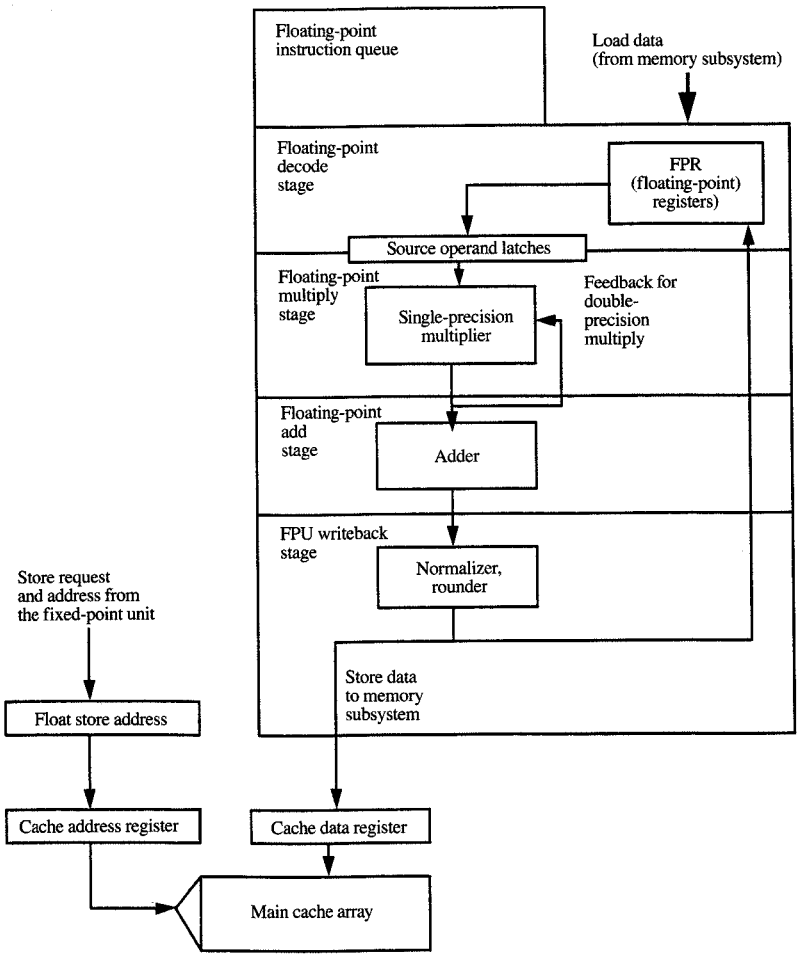


Figure 7

Logical block diagram of the PowerPC 601 floating-point unit (FPU).

Only two of the five instructions are executed in the FPU. Three instructions (the loads and the store) are executed in the FXU. The branch, of course, is executed in the branch processor. This loop takes six cycles per iteration on the 601. When the same loop is recoded with single-precision instructions and operands, it also takes six cycles per iteration on the 601. The timing of the loop is dominated by the fact that the branch cannot access the cache because it has a lower priority than the two loads and the store. If the loop is unrolled, the effect of the branch is minimized and the longer execution time of the double-precision fmadd instruction can be revealed. Code with a higher concentration of floating-point multiply and accumulate instructions is affected more by the decision to have a half-wide multiplier.

The benefit of using a half-wide multiplier is a savings in chip area required for the floating-point unit. The multiply–add unit of the 601 takes up about 4 mm^2. A double-precision unit would take up approximately twice the space (8 mm^2). The space saved is between three and four percent of the die size.

Bus interface unit (BIU)

The bus interface unit (BIU) in the 601 processor consists of the queueing registers and control logic to connect the unified cache to external memory and to the I/O interface.

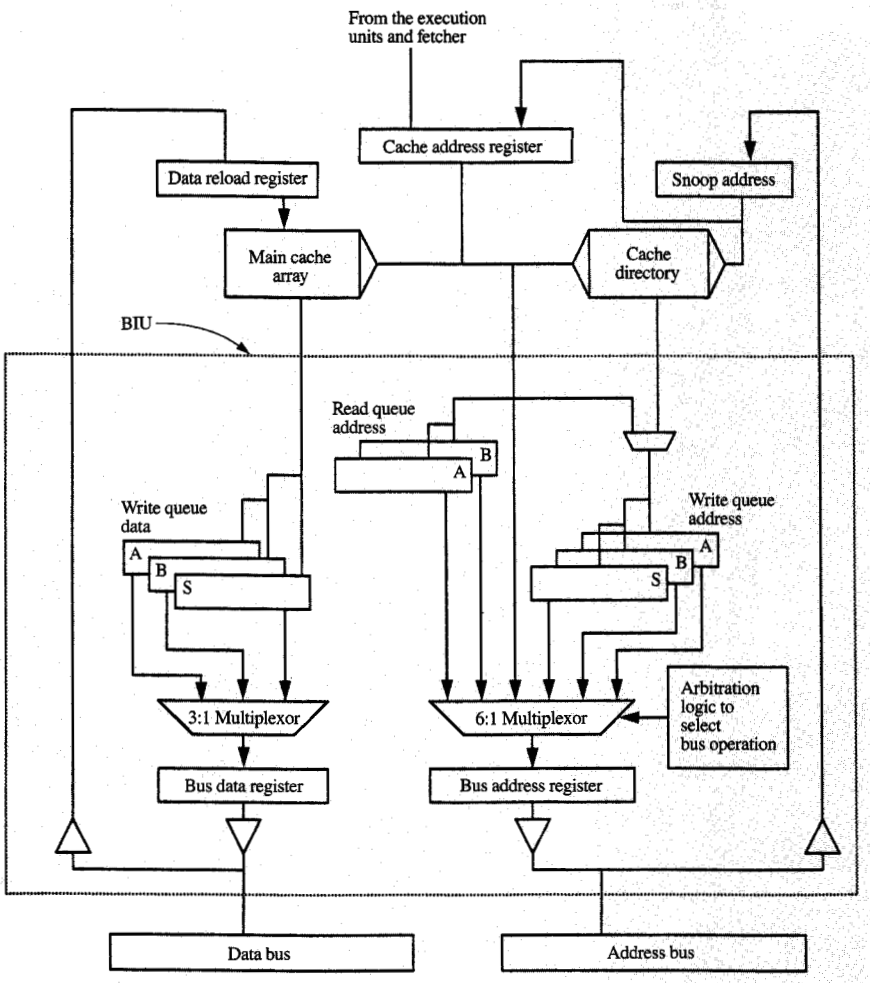


Figure 8

Block diagram of the PowerPC 601 bus interface unit (BIU).

A logical block diagram of the BIU is shown in **Figure 8.**

The 601 interface design had the following goals and constraints.

First, the 601 had to support a general-purpose, high-performance memory interface capable of high-speed instruction and data transfer using burst reads and writes. A queueing mechanism that would allow processor performance optimization by supporting out-of-order memory references was needed. Customer needs dictated that 601 memory operations generally follow the Motorola MC88110 RISC microprocessor interface protocol.

Second, I/O device support using strongly ordered read/write operations was needed. A separate I/O address space, as opposed to memory-mapped I/O, was needed to support existing I/O architectures. Because of pin-count limitations, this I/O bus protocol had to be mapped onto the memory bus signal lines.

Third, multiprocessor support, including cache coherency, was needed.

• *Bus description*

The 601 bus consists of a 32-bit address bus and a 64-bit data bus with control signals that allow a split-transaction bus protocol: Address bus transfers can complete on the address bus, followed later by the associated data bus transfer. While the data bus transfer for one operation is

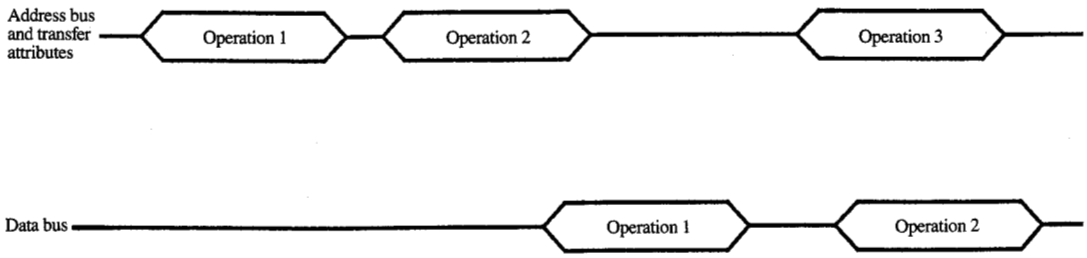


Figure 9

Pipeline operation of the PowerPC 601 address and data buses.

pending, the address bus can be used to initiate a subsequent transfer, thus allowing address and data transfers to be "pipelined" on the bus, as shown in **Figure 9**. This allows the 601 bus to have multiple operations outstanding on the bus: The address bus tenure for several operations from different bus master devices can be placed on the bus, awaiting data bus tenure to complete the transfer. The 601 places a maximum of two operations on the address bus and does not start subsequent operations until the data bus tenure for one of the previous operations has completed.

High data bandwidth on the bus is maintained by operating the data bus in a burst mode, in which four data transfers of sequential data (32 bytes total) can be made for each address placed on the bus. This allows a peak bandwidth of more than 400 megabytes per second in a 66-MHz 601 system.

The 601 bus further incorporates an additional bus protocol for input/output device support on the bus. This protocol provides strongly ordered bus operations and synchronous error-reporting capability. This is implemented on the 601 bus by placing two 32-bit address words on the bus during address bus tenure, allowing a to/from device address and additional device control information to be appended to the normal 32-bit address. Synchronous error reporting is accomplished by having the I/O device respond to a transfer with an address-only positive or negative acknowledgment before the processor continues execution of the instruction stream.

A block diagram of the 601 BIU is shown in Figure 8. Central to the operation of the BIU is the cache address register (CAR), which holds the address and operation codes for all accesses to the cache or to the 601 external bus. The CAR is used to simultaneously access the cache directory and the cache data array, and, depending on the state of the cache, the CAR contents can be copied into one of two read queues or one of three write queues.

The read queues are used to queue noncacheable data loads and instruction fetches and to queue cacheable loads, fetches, and store operations which miss in the cache.

The write queues are used to queue noncacheable and cacheable write-through stores, cache sector replacement copy-back operations (cast-outs), and cache sector-push operations caused either by cache manipulation instructions or by snoop activity on the bus.

This read and write queueing mechanism is used to improve overall processor performance by freeing cache bandwidth when a cache miss occurs and prioritizing operations placed on the bus. As an example, consider the case of an instruction prefetch which misses in the cache. The operations which must occur are the following:

1. Accessing the cache directory to determine the cache miss.
2. Selection of the least recently used line in the cache. If this line contains modified data, it must be written back to memory before that cache location can be used.
3. The read operation must be completed on the bus, allowing the cache to be updated with the read data.

If these operations were to be performed directly from the CAR, it would essentially block all cache accesses until both the write and read operations were completed on the bus. With the read and write queueing mechanism, the following can occur: The copy-back data and address can be copied from the cache to a write queue while the read

operations address is placed in the read queue. The CAR is then freed for other loads, fetches, stores, etc., while the priority arbiter in the BIU allows the read operation to precede the lower-priority copy-back operation.

• *Memory queue-to-bus priority arbitration*

The priority for placing operations from the memory queue onto the bus is important for two reasons: First, careful prioritization can improve processor performance by getting instructions and data to the appropriate function unit in the processor or attached device to avoid stalls. Second, proper prioritization alleviates potential bus deadlock in coherent memory systems.

In general, the priority used in the 601, and the reasoning behind it, is as listed below in declining order of priority:

1. Snoop copy-back (snoop push caused by the bus coherency logic). The snoop copy-back is of highest priority for two reasons:
   - To avoid the deadlock situation described below.
   - For overall system performance: In the snoop copy-back case, another processor or bus device is accessing the data which are modified in the 601 cache. This operation is retried by the other device on the bus until the 601 has written the modified data back into the bus memory.
2. Program-initiated push caused by data cache block flush (dcbf) or store (dcbst) operations. These operations are not considered critical for performance, but were placed high in the priority to simplify the logic.
3. Programmable input/output (PIO) operations which have not been retried. PIO operations, especially read operations, may take a long time to complete, and will stall the processor. By placing the initial portions of a PIO high on the priority list, the PIO operation can be started. Once it has been placed on the bus, subsequent portions of the PIO operation are assigned a low priority so that other regular memory operations pending in the read queue can be placed on the bus.
4. Address-only cache operations (data cache block flush, store, invalidate, or zero). These operations must be placed on the bus and complete execution there before the fixed-point unit in the processor logic can progress, and are therefore given relatively high priority.
5. Read operations caused by a load or a fetch. These operations may stall the processor because of register dependencies or lack of instructions, and are therefore of relatively high priority.
6. Noncacheable write operations. These operations must complete in order and must complete before any data cache synchronization operation (sync) listed below.
7. Synchronizing operations. The sync is placed here to ensure that all noncacheable load/store operations are executed and clear the processor before the sync completes.
8. Dynamic reload of the other sector of a cacheable read operation. These can be considered optional bus operations, which may help processor performance by prefetching instructions or data which are in the same cache line as data or instructions that the processor is actually using, and are therefore of low priority.
9. Cast-out writes. Cast-out data can be written back to the cache whenever it is convenient, and this operation is therefore of very low priority.
10. Programmable input/output (PIO) operations which have been retried. Programmable I/O operations, in particular PIO load operations, consist of a request, followed by data transfers, followed by an acknowledgment. For comparatively slow I/O systems, the PIO read operations may be retried on the bus many times before the data become available. By toggling retried I/O operations to the lowest priority on the bus, any other normal memory operations waiting in the queue can be executed, improving bus utilization and potentially improving performance.

• *Bus deadlocks*

In multi-master bus systems, which can include either multiprocessor systems or uniprocessor systems with other bus master devices, coherency between the memory system and caches located in the various bus devices is maintained by the basic MESI protocol. During an address tenure, the transfer can be terminated with the snoop response signal ADDRESS RETRY. A normal response may terminate with the SHARED response. When the SHARED response is given in response to a bus read operation, the cached data are marked shared. When the ADDRESS RETRY response is given, the operation is terminated without data transfer and the snooped device is given, through address bus arbitration, the opportunity to use the bus to copy back the modified data from its cache to main memory.

A significant bus deadlock occurs in systems in which a device adapts a nonpended bus to the 601 bus. The nonpended bus can initiate an operation which locks up the nonpended bus until the operation is complete. If the operation from the nonpended bus causes a snoop-push operation by the 601, a deadlock may occur if the 601 has simultaneously initiated bus operations to the nonpended bus adapter.

Figure 10 illustrates an example of the deadlock: The 601 can issue a read operation to the bus adapter, where the operation is queued until the adapter can acquire access to the nonpended bus. At the same time, a device on the nonpended bus can issue a read operation through the bus adapter to the memory on the 601 bus. With a coherent memory system on the 601 bus, the 601 may retry this read operation in order to write modified data from its cache back to the memory. A deadlock can occur: The 601-to-nonpended bus read cannot complete until the nonpended bus is available. The nonpended bus is not available until the nonpended bus-to-601 memory read operation is completed. This memory read cannot complete until the 601 can complete a snoop-push write from its cache to the 601 memory. The snoop-push write is not able to complete because the 601 write queues may be full from previous operations, and, even if a write queue is available, the data bus tenure for the write operation cannot complete until the data bus tenure for the previous 601-to-nonpended bus read operation is complete.

This deadlock is resolved by using three mechanisms.

First, a special, high-priority snoop-push write queue is reserved in the 601. Use of this queue is controlled by using an input to the 601 (HIGH-PRIORITY SNOOP REQUEST), which can be asserted by the nonpended bus adapter as part of the attributes transferred during address bus tenure. This allows a snoop-push write to be placed in the 601 memory queue as the highest-priority operation.

Second, the 601 implements a DATA BUS WRITE ONLY (DBWO) signal which, when asserted by the 601 bus arbitration logic, allows out-of-order data bus tenure, thereby permitting the snoop-push write data bus tenure to precede the data bus tenure for the pending read operation, whose address bus tenure preceded the write tenure.

Third, restrictions are placed in the 601 bus arbitration logic and the bus adapter logic to limit the number of reads the 601 can place on the 601 bus once a read from the bus adapter by the 601 is pending. This allows the snoop-push write from the 601 to be placed on the bus if needed.

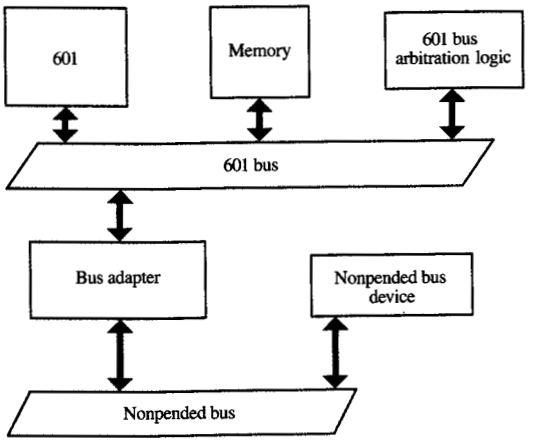


Figure 10

System configuration in which bus deadlocks might occur.

Summary

The 601 microprocessor is the first implementation of a family of PowerPC microprocessors. The goals of the 601 product were to provide a PowerPC implementation that would also serve as a software bridge from the POWER architecture, and it had to be developed on a very tight schedule in order to bring a PowerPC processor to the marketplace quickly. The 601 was held to the specified die size of 10.95 × 10.95 mm^2.

The 601 also meets its original performance objectives in terms of processor cycle time and performance against industry standard benchmarks. The SPEC benchmarks for some IBM products that use the 601 are shown in **Table 3**.

Table 3 SPEC benchmark ratios for 601-based computers.

Benchmark name	*SPEC ratios for IBM RISC System/6000 POWERstation™ models*	
	Model 250	*Model C10*
008.espresso	58.4	76.9
022.li	74.2	104.0
023.eqntott	76.9	107.8
026.compress	41.2	67.4
072.sc	85.2	121.8
085.gcc	51.6	77.7
Geometric mean: SPECint92	62.6	90.5
013.spice2g6	43.4	68.9
015.doduc	56.4	79.1
034.mdljdp2	85.7	114.0
039.wave5	48.6	63.6
047.tomcatv	94.3	129.9
048.ora	61.1	74.3
052.alvinn	160.9	220.3
056.ear	143.8	173.6
077.mdljsp2	47.6	63.2
078.swm256	63.9	81.5
089.su2cor	72.6	123.3
090.hydro2d	53.6	80.0
093.nasa7	72.1	117.2
094.fpppp	89.7	128.5
Geometric mean: SPECfp92	72.2	100.8

Obviously, performance of a computer system is a function of all the components—not just the processor. However, the SPEC benchmark numbers provide some indication of the performance of the PowerPC 601 microprocessor.

The effort of designing the 601 processor involved many design trade-offs that tried to balance chip area, performance, and design complexity (time to market). The experience of the design team, the ability to extend and reuse parts of the RISC single-chip design, and a design philosophy of "keeping it simple" all contributed to the success of the 601 processor, which is now being used [9] in products by IBM, Apple, and even other companies that were not part of the original PowerPC alliance.

Acknowledgments

Acknowledgment must first be made to the entire 601 design team. The design and delivery of the 601 microprocessor on the tight design schedule would not have been possible were it not for the dedication and personal sacrifice of the entire team. Special acknowledgment is made to Chris Olson for his assistance in writing the section on the floating-point unit.

References

1. Charles R. Moore and Robert L. Mansfield, "PowerPC Alliance," *PowerPC and Power2: Technical Aspects of the New IBM RISC System/6000*, IBM Corporation, Austin, TX, 1994, pp. 69–72.
2. G. Paap and E. Silha, "PowerPC: A Performance Architecture," *Proceedings of COMPCON SPRING 1993*, IEEE, Piscataway, NJ, February 1993, pp. 104–108.
3. G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 37–58 (January 1990).
4. C. R. Moore, D. M. Balser, J. S. Muhich, and R. E. East, "IBM Single Chip RISC Processor (RSC)," *Proceedings of the 1992 International Conference on Computer Design*, IEEE Computer Society Press, Los Alamitos, CA, 1992.
5. C. R. Moore, "The PowerPC 601 Microprocessor," *Proceedings of COMPCON SPRING 1993*, IEEE, Piscataway, NJ, February 1993, pp. 109–116.
6. M. S. Allen and M. C. Becker, "Multiprocessing Aspects of the PowerPC 601," *Proceedings of COMPCON SPRING 1993*, IEEE, Piscataway, NJ, February 1993, pp. 117–126.
7. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, San Mateo, CA, 1990.
8. J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro* **13,** No. 4, 17–27 (1993).
9. *PowerPC 601 RISC Microprocessor User's Manual*, IBM Microelectronics Division, Essex Junction, VT, 1993.

Received October 8, 1993; accepted for publication March 7, 1994

Michael T. Vaden *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (MTVADEN at AUSVM6).* Mr. Vaden received a B.S.E.E. degree from Texas A&M University in 1983 and an M.S.E. in electrical engineering from the University of Texas at Austin in 1990. He began his career with IBM Austin in 1983 working in a hardware systems test organization, specifically working with data communications adapters and software. He worked on the RISC single-chip (RSC) microprocessor before beginning work on the 601 project, where he has worked on the design of the fixed-point unit. Mr. Vaden is currently employed by IBM as a staff engineer on the PowerPC processor design team at the Somerset Design Center in Austin, Texas.

Lawrence J. Merkel *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (MERKEL at AUSTIN).* Mr. Merkel received a B.S. in electrical engineering from the University of Texas at Austin in 1990. He has been employed at IBM since then, working in processor development in both verification and design capacities. Mr. Merkel worked on the RISC single-chip processor before coming to the 601 project. He is currently working on future processor development at the Somerset Design Center.

Charles R. Moore *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (CMOORE at AUSTIN).* Mr. Moore is a Senior Engineer currently on assignment in the Somerset Design Center, working in the High-End Processor Development group. He was the co-lead architect of the PowerPC 601 microprocessor. Previously, he was one of the key designers of the branch unit of the original RISC System/6000® chip set; he was also heavily involved in the development of the RISC single-chip (RSC) microprocessor. Mr. Moore received a B.S.E.E. in 1984 from Rensselaer Polytechnic Institute and an M.S.E.E. from the University of Texas at Austin in 1991. He is a member of the IEEE.

Terence M. Potter *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (POTTER at AUSTIN).* Mr. Potter received a B.S. in electrical engineering from the University of Texas at Austin in 1991. That same year he joined IBM Austin, where he designed the branch processing unit and instruction fetcher/dispatcher on the 601 microprocessor. He holds one IBM Invention Achievement Award, and has applied for three patents. Mr. Potter is currently a Senior Associate Engineer working in the PowerPC microprocessor design group at the Somerset Design Center.

Robert James Reese *IBM RISC System/6000 Division, 11400 Burnet Road, Austin, Texas 78758 (REESE at AUSVM6).* Mr. Reese received a B.S. in 1976 and an M.S. in 1978, both in electrical engineering, from Utah State University. In 1977 he joined IBM in Boulder, Colorado, where he worked on signal processor design, high-speed page printer control unit logic, and digital image processing. In 1989 he moved to Austin, where he worked on the RSC microprocessor and the bus interface unit of the 601 microprocessor. He holds two IBM Invention Achievement Awards and has filed six patent applications. Mr. Reese is currently an Advisory Engineer in PowerPC processor development at the Somerset Design Center.