The PowerPC 604 RISC Microprocessor

The PowerPC 604 RISC microprocessor uses out-of-order and speculative execution techniques to extract instruction-level parallelism. Its nonblocking execution pipelines, fast branch misprediction recovery, and decoupled memory queues support speculative execution.

S. Peter Song

Marvin Denman

Joe Chang

Somerset Design Center

The 604 microprocessor is the third member of the PowerPC family being developed jointly by Apple, IBM, and Motorola. Developed for use in desktop personal computers, workstations, and servers, this 32-bit implementation works with the software and bus in the PowerPC 601 and 603 microprocessors.[1-3] While keeping the system interface compatible with the 601 microprocessor, we improved upon it by incorporating a phase-locked loop and an IEEE-Std 1149.1 boundary-scan (JTAG) interface on chip. In addition, an advanced machine organization delivers one and a half to two times the 601's integer performance.

Performance strategy

Processor performance depends on three factors: the number of instructions in a task, the number of cycles a processor takes to complete the task, and the processor's frequency.[4,5] Our architecture, which we optimized to produce compact code while adhering to the reduced instruction set computer (RISC) philosophy, addresses the first factor. The high instruction execution rate and clock frequency addresses the other two factors. The 604 provides deep pipelines, multiple execution units, register renaming, branch prediction, speculative execution, and serialization.

Six-stage superscalar pipeline. As shown in Figure 1, this deep pipeline enables the 604 to achieve its 100-MHz design. The stages are

- *Fetch.* This stage translates an instruction fetch address and accesses the cache for up to four instructions.
- *Decode.* Instruction decoding determines needed resources, such as source operands and an execution unit.
- *Dispatch.* When the resources are available, dispatch sends instructions to a reservation station in the appropriate execution unit. A reservation station permits an instruction to be dispatched before all of its operands are available.[6] As they become available, the reservation station forwards operands to the execution units. Dispatch can send up to four instructions in program order (in-order dispatch) to four of six execution units: two single-cycle integers, a multicycle integer, a load/store, a floating point, and a branch.
- *Issue/execute.* In each execution unit, this stage issues one instruction from its reservation station and executes it to produce results. The instructions can execute out of program order (out-of-order execution) across the six execution units as well as within an execution unit that has an out-of-order issue reservation station. Table 1 lists the latency and throughput of the execution stages.

0740-7475/94/$04.00 © 1994 IEEE

- *Completion.* An instruction is said to be *finished* when it passes the execute stage. A finished instruction can be *completed* 1) if it does not cause an exception condition and 2) when all instructions that appear earlier in program order complete. This is known as in-order completion.
- *Write back.* This stage writes the results of completed instructions to the architectural state (or the state that is visible to programmer). Bypass logic permits most instructions to complete and write back in one cycle.

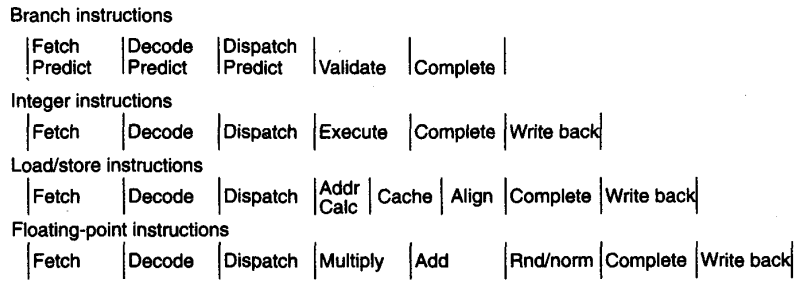


Figure 1. Pipeline description.

Although some designs use even deeper pipelines to achieve higher clock frequencies than the 604 does, we felt that such a design point does not suit today's personal computers. It relies too heavily on one of, or a combination of, a very large on-chip cache, a wide data bus, or a fast memory system to deliver its performance. It would be less than competitive in today's cost-sensitive personal computer market.

Precise interrupts and register renaming. Most programmers expect a pipelined processor to behave as a nonpipelined processor, in which one instruction goes through the fetch to write-back stages before the next one begins. A processor meets that expectation if it supports precise interrupts, in which it stops at the first instruction that should not be processed. When it stops (to process an interrupt), the processor's state reflects the results of executing all instructions prior to the interrupt-causing instruction and none of the later instructions, including the interrupt-causing instruction. This is not a trivial problem to solve in multiple, out-of-order execution pipelines. An earlier instruction executing after a later instruction can change the processor's state to make later instruction processing illegal. Sohi gives a general overview of the design issues and solutions.[7]

The 604 uses a variant of the reorder buffer described by Smith and Pleszkun to implement precise interrupts.[8] The 16-entry reorder buffer keeps track of instruction order as well as the state of the instructions. The dispatch stage assigns each instruction a reorder buffer entry as it is dispatched. When the instruction finishes execution, the execution unit records the instruction's execution status in the assigned reorder buffer entry. Since the reorder buffer is managed as a first-in/first-out queue, its examining order matches the instruction flow sequence. To enforce in-order completion, all prior instructions in the reorder buffer must complete before an instruction can be considered for completion. The reorder buffer examines four entries every cycle to allow completion of up to four instructions per cycle.

Table 1. 604 execution timings.

Instruction	Latency	Throughput
Most integer	1	1
Integer multiply (32×32)	4	2
Integer multiply (others)	3	1
Integer divide	20	19
Integer load	2	1
Floating-point load	3	1
Store	3	1
Floating-point multiply-add	3	1
Single-precision floating-point divide	18	18
Double-precision floating-point divide	31	31

Unlike Smith and Pleszkun's reorder buffer, the 604's reorder buffer does not store instruction results. Temporary buffers hold them until the instructions that generated them complete. At that time, the write-back stage copies the results to the architectural registers. The 604 renames registers to achieve this; instead of writing results directly to specified registers, they are written to rename buffers and later copied to specified registers. Since instructions can execute out of order, their results can also be produced and written out of order into the rename buffers. The results are, however, copied from the buffers to the specified registers in program order. Register renaming minimizes architectural resource dependencies, namely the output-dependency (or write-after-write hazard) and antidependency (or write-after-read hazard), that would otherwise limit opportunities for out-of-order execution.[9]

Figure 2 (next page) depicts the format of a rename buffer entry. The 604 contains a 12-entry rename buffer for the general-purpose registers (GPRs) that are used for 32-bit integer operations. The 604 allocates a GPR rename buffer entry upon dispatch of an instruction that modifies a GPR. The dispatch stage writes a destination register number of the

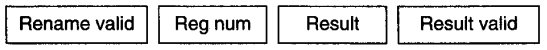


Figure 2. Rename buffer entry format.

instruction to the Reg num field, sets a Rename valid bit, and clears the Result valid bit. When the instruction executes, the execution unit writes its result to the Result field and sets the Result valid bit. After the instruction completes, the write-back stage copies its result from the rename buffer entry to the GPR specified by the Reg num field, freeing the entry for reallocation. For a load-with-update instruction that modifies two GPRs, one for load data and another for address, the 604 allocates two rename buffer entries.

Register renaming complicates the process of locating the source operands for an instruction since they can also reside in rename buffers. In dispatching an integer instruction, the dispatch stage searches its source operands simultaneously from the GPR file and its rename buffer. If a source operand has not been renamed, the processor uses the value read from the GPR file. If a rename exists (indicated by an entry with the Rename valid set and its Reg num field matching the source register number), the Result in the rename buffer is used. It is, however, possible that the result is not yet valid because the instruction that produces the GPR has not yet executed. The dispatch stage still dispatches the instruction since the operand will be supplied by the reservation station when the result is produced. The dispatched instruction contains the rename buffer entry identifier in place of the operand. The GPR file and its rename buffer can use eight read ports for source operands to support dispatching of four integer instructions each cycle.

The 604 also uses a rename buffer for floating-point registers (FPRs) and one more for the condition register (CR). The FPR rename buffer has eight 90-bit-wide entries to hold a double-precision result with its data type and exception status. The FPR file and its rename buffer access three read ports for dispatching one floating-point instruction per cycle. In addition to compare instructions, most integer and floating-point instructions can also generate negative, positive, zero, and overflow condition results. One of the eight fields in the 32-bit CR stores these 4-bit condition results. The 604 treats each field as a 4-bit register and applies register renaming using an eight-entry CR rename buffer.

Branch prediction and speculative execution. Because today's application software contain a high percentage of branch instructions, correctly predicting the outcome of these instructions is crucial to keeping the multiple instruction pipelines flowing and for achieving two to three times the execution rate of scalar processors. The 604 uses dynamic branch prediction in the fetch, decode, and dispatch stages to predict as well as correct branch instructions early.

The 604's speculative execution strategy complements its branch prediction mechanisms. The strategy is to fetch and execute beyond two unresolved branch instructions. The results of these speculatively executed instructions reside in rename buffers and in other temporary registers. If the prediction is correct, the write-back stage copies the results of speculatively executed instructions to the specified registers after the instructions complete.

Upon detection of a branch misprediction, the 604 takes quick action to recover in one cycle. It selectively cancels the instructions that belong in the mispredicted path from the reservation stations, execution units, and memory queues. It also discards their results from the temporary buffers. In addition, the processor resumes its previous state to start executing from the correct path even before the mispredicted branch and its earlier instructions have completed. Since the 604 detects a branch misprediction many cycles before the branch instruction completes, its fast recovery scheme helps to maintain performance of those applications with high data cache miss rates and whose branches are difficult to predict.

Serialization. A serialization mechanism delays execution of certain instructions that would otherwise be expensive to execute speculatively in the 604's multiple-pipeline, out-of-order execution design. This mechanism delays infrequently used instructions until they can safely execute while permitting later instructions to execute. Some examples are the move to and from special-purpose register instructions, the extended arithmetic instructions that read the carry bit, and the instructions that directly operate on the CR, which the PowerPC architecture provides for calculating complex branch conditions. This mechanism also controls store instructions since it is difficult to undo stores.

The dispatch stage sends a serialized instruction to the proper execution unit with an indication that it should not be executed. When all prior instructions have completed and updated their results to the architectural state, the completion stage allows the serialized instruction to execute. Once the serialized instruction is dispatched, dispatch continues to dispatch the following instructions so they can execute before the serialized instruction. When the serialized instruction is completed, the later instructions also complete upon finishing execution. This minimizes the penalty of serialized instructions.

Machine organization

Figure 3 shows the fetch address generation logic. The fetch stage selects an address from the addresses generated in the different pipeline stages each cycle. Since an address generated in a later stage belongs to an earlier instruction, its selection precedes an address from an earlier stage.

The completion stage detects exception conditions and generates an exception handler address. This stage also

updates the program counter (PC) with the target address of a taken branch instruction, or advances it by the number of instructions being completed. The branch execute stage may correct the instruction fetch with the branch target address if the branch is mispredicted as not taken and with the sequential address if the branch is mispredicted as taken. The dispatch and decode stages may change the fetch address with either the target or sequential address of a branch instruction being predicted. There are two copies of the target and sequential address registers in the decode, dispatch, and execute stages, since there can be up to two branch instructions in each stage. The completion stage also has two target registers to handle up to two finished branch instructions.

If the fetch address hits in the branch target address cache (BTAC), the target address becomes the fetch address. Otherwise, the instruction fetch continues sequentially. The 64-entry, fully associative BTAC holds the target addresses of the branches that are predicted to be taken. If a branch is predicted as not taken for its next encounter, the branch execute stage removes it from the BTAC. The BTAC is accessed with the fetch address, and not with a branch instruction address, providing a zero-cycle fetch penalty for taken branches. Although there may be multiple branch instructions in the four instructions being fetched, the BTAC provides the target address of the first-predicted taken branch instruction.

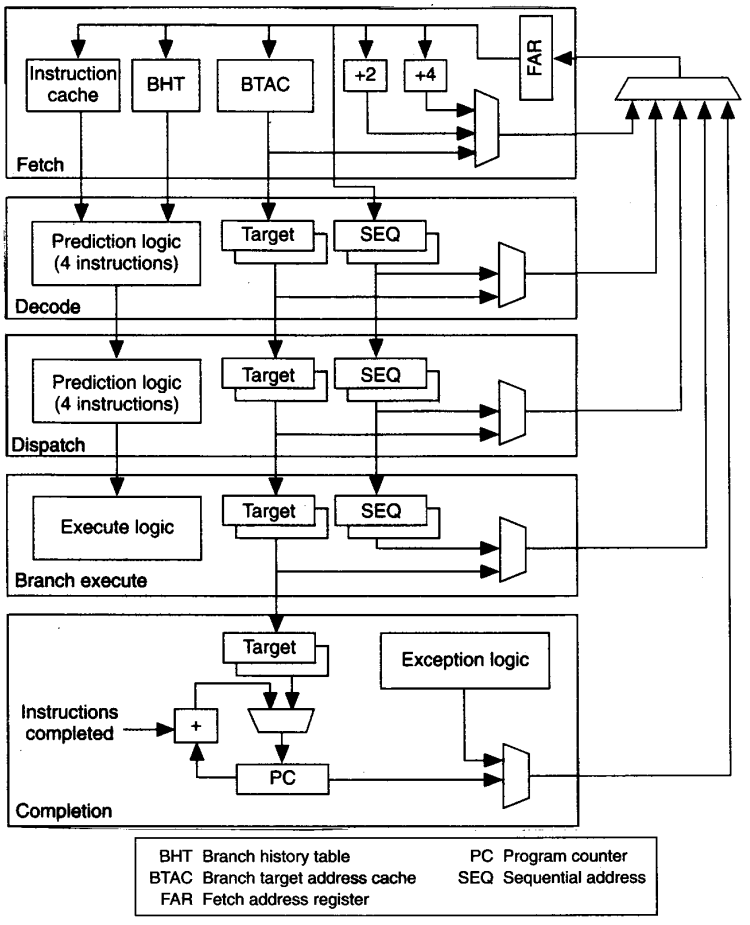


Figure 3. Instruction fetch address generation logic.

Instruction decode and dispatch. The pipeline decodes four instructions every cycle to determine exception conditions, as well as the resources needed by the instructions. The resources include the execution unit, source operands, and destination registers. Decoding the instructions before the dispatch stage simplifies the dispatch logic without using predecoded bits in the instruction cache. Predicting branch instructions in the first two entries of the decode buffer minimizes the performance penalty of adding the decode stage.

When the decode stage detects an unconditional branch that was not in the BTAC, it corrects the instruction fetch to the target address of the branch. It also predicts conditional branches with the execution history found in the branch history table (BHT).[10] Each entry of the 512-entry BHT denotes one of the four history states: strong not taken, weak not taken, weak taken, and strong taken. The table updates the history state with the actual outcome of the branch that is mapped to the entry. To simplify the design, each entry in the BHT maps to every 512th instruction address. This allows the BHT to be accessed with the fetch address and to return the four entries mapped to the four instructions starting with the fetch address.

Not all conditional branch instructions use the BHT. The architecture provides a count register (CTR) value as a branch condition to support loops in programs. Only the conditional branch instructions that do not depend on the CTR value use, as well as update, the BHT. Those that do depend on the CTR are predicted, based on the value of the shadow CTR. The shadow CTR has a future state of the CTR that is

updated by speculatively executed move-to-CTR or branch-and-decrement instructions. This prediction scheme is very effective on branches that control loop iteration.

The dispatch buffer sends up to four instructions to four of the six execution units each cycle. As space allows, more instructions advance from the decode stage into the four-entry dispatch buffer. The 604 places only a few restrictions on dispatch to enable a high-speed implementation. They are

1) *One instruction per execution unit.* Since each execution unit can start only one instruction per cycle and an instruction can bypass the reservation station if the execution unit is available, dispatching one instruction per unit simplifies the logic without imposing an undue performance penalty. Two identical single-cycle integer units handle the more frequent instructions.
2) *Resources available.* Each instruction needs a reorder buffer entry, a reservation station entry in the appropriate execution unit, and rename buffer entries to hold its results. Available resources depend on the state of the instructions previously dispatched as well as those currently being dispatched.
3) *Stop dispatch after branch.* Instructions following a branch instruction are not dispatched in the same cycle as the branch is dispatched. This restriction simplifies saving the processor state, which allows immediate canceling of speculatively executed instructions that follow predicted branches.
4) *In-order dispatch.* Dispatching instructions in order results in only a small cost in performance and greatly simplifies resource allocation and dispatch logic. Out-of-order execution is introduced with six independent execution pipelines and out-of-order issue reservation stations to achieve performance comparable to an out-of-order dispatch design.

Reservation stations and result forwarding. A two-entry reservation station on every execution unit allows instructions to be dispatched before obtaining all of their operands. Without a reservation station, an instruction cannot be dispatched until all of its source operands become available, either in the register file or in its rename buffer. Without reservation stations, the 604's in-order dispatch design would be more complex, since it would have to detect data dependencies and would frequently stall. The reservation stations in the three integer units can issue instructions out of order to allow a later instruction to bypass an earlier stalled instruction. The branch, load/store, and floating-point unit reservation stations may only issue instructions in order.

Each execution unit provides one result bus for each type of result it produces. For instance, the multicycle integer unit has one result bus for the GPR and another for the CR data types. Figure 4 shows the four GPR result buses and the reservation stations and GPR rename buffer that are connected to them. Each GPR reservation station entry monitors all four GPR result buses for any missing A or B operands, denoted as A op and B op in the figure. When an execution unit returns a result and the associated GPR rename buffer entry identifier, the reservation station compares the identifier against those in its entries. When a match is found, it forwards the result to the waiting instruction. For returning the update address of a load-with-update instruction while executing one load instruction per cycle, the load/store unit shares the result bus of the less frequently used multicycle integer unit.

It is interesting to note why the 604 uses a reorder buffer, rename buffers, and reservation stations to provide the same functions that a DRIS (deferred-scheduling, register-renaming instruction shelf) in the Metaflow architecture provides.[11] A DRIS entry consists of instruction status fields that a reorder buffer entry would have, source operand fields that a reservation station entry would have, and destination fields that a rename buffer entry would have. (The 604's reservation station entry uses a separate source operand to store either an immediate or a copy of the source operand. Although the DRIS figure in the Metaflow article shows only the ID field to indicate the DRIS entry with the source operand, it is likely that another field is needed to store an immediate operand.)

Two disadvantages of the DRIS had we used it in the 604 design are

- *Scheduling overhead.* The DRIS instruction scheduling is more complicated than the 604's dedicated reservation stations since the next instruction for an execution unit must be the first "ready" instruction of the "right" type in all DRIS entries.
- *Single result type.* DRIS supports renaming of only one register type, whereas the 604 needs three. Say that more than one DRIS is used, as described in Popescu et al.,[11] to support separate integer and floating-point registers. One of them would have to house all instructions to provide precise interrupts while not being able to provide register renaming. An alternative is to design one DRIS to accommodate the largest register type.

Execution units. The branch execution unit can hold two branch instructions in its reservation station and two more finished branch instructions. It serves to validate branch predictions made in earlier stages, and also verifies that the predicted target matches the actual target address. If a misprediction is detected, the branch execution unit redirects the fetch to the correct address and starts the branch misprediction recovery.

The 604 has a three-stage complex integer unit (CFX) to execute integer multiply, divide, and all move to and from special-purpose register instructions. The CFX can sustain one multiply instruction per cycle for 32×16-bit and those 32×32-bit multiplies whose B operand is representable as a

17-bit signed integer. It can sustain one multiply per two cycles for larger 32×32-bit operands. The CFX also uses the multiply pipeline stages to execute a divide instruction in 20 cycles. The 604's two simple integer units execute all other integer instructions in one cycle.

The three-stage floating-point unit can sustain one double-precision multiply-add per cycle, one single-precision divide every 18 cycles, or one double-precision divide every 31 cycles. It complies fully with the IEEE-Std 754 floating-point arithmetic standard. The 604 provides hardware support for denormalization, exceptions, and three graphics instructions. It also provides a non-IEEE mode for graphics support. The non-IEEE mode converts a denormalized result to zero to avoid prenormalization in subsequent operations.

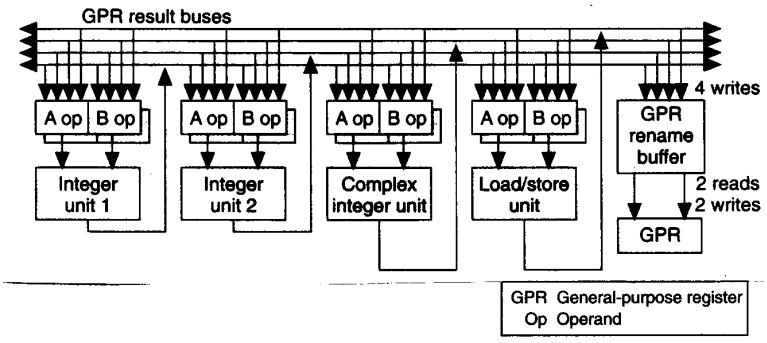


Figure 4. GPR result buses, reservation stations, and rename buffer.

Instruction completion and write back. After an instruction executes, the execution unit copies results to its rename buffer entries and the execution status to its reorder buffer entry. Among other things, the execution status indicates whether the instruction finished execution without an exception. Of the four reorder buffer entries examined every cycle, up to four instructions that finished without an exception complete in program order.

Other than the in-order completion necessary to support precise exceptions, the 604 imposes only a few additional restrictions on instruction completion. They are

1) *Stop before a store instruction.* Since a store data operand is read from the register file in the completion stage, a store instruction cannot complete if its store operand is still in the rename buffer. Stopping completion before a store instruction allows the store operand to be written to the register file, even if it is produced by an instruction currently being completed.
2) *Stop after a taken branch instruction.* Since a taken branch instruction sets the program counter to its target address, it is speed critical to advance the program counter from the new target address by the number of instructions completed after the taken branch in one cycle. Stopping completion after a taken branch instruction avoids this logic altogether.

To minimize effects of long execution latency on in-order completion, the completion stage overlaps with the last execution cycle for those instructions with multicycle execution stage. These include the multiply, divide, store, load miss, and execution serialized instructions. A store instruction completes as soon as it is translated without an exception. Similarly, a load instruction that misses in the data cache completes upon translation without an exception. Since the reservation stations can forward the load data when it is available to the dependent instructions, the load miss can safely be completed.

Most superscalar designs impose additional restrictions due to a limited number of ports to register files. For instance, four write ports would be required to complete up to four instructions if each one can update one register. The 604 GPR file would require eight write ports to complete four load-with-update instructions per cycle. The 604 avoids this problem by decoupling instruction completion from register file updates using the write-back stage. Instructions complete without regard to the type or number of registers they update. The completion stage updates their results if ports are available; if not, the write-back stage updates them. The rename buffer entries function as temporary buffers for those instructions that are not completed and as write-back buffers for those that are. All three GPR, FPR, and CR rename buffers contain two read ports for write back. Correspondingly, the three register files have two write ports for write back.

Memory operations

High-speed superscalar processors require a greater memory bandwidth to sustain their performance. The 604 meets the increased demand with large on-chip caches, nonblocking memory operations, and a high-bandwidth system interface. The 604 takes advantage of the weakly ordered memory model, to which the PowerPC architecture subscribes, to offer efficient memory operations. Although loads and stores that hit in the data cache can bypass earlier loads and stores, program order memory access can be enforced with instructions provided for this purpose.

Load/store unit. Figure 5 (next page) shows a block diagram of the load/store unit and the memory queues. This unit has a two-cycle execution stage. It calculates the memory address and translates that address with a 128-entry, two-

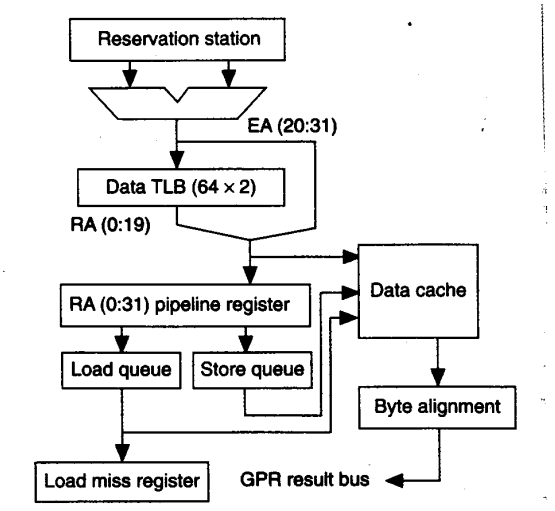


Figure 5. Load/store unit and memory queues.

way set-associative translation lookaside buffer (TLB) in the first cycle. The second cycle processes loads making a speculative cache access and aligns bytes when the access hits in the cache. The pipelined execution stage executes one load or store instruction per cycle.

In the first half cycle, the load/store unit calculates a load instruction's memory address, denoted as EA in the figure. It translates the real address, denoted as RA in the figure, and the data cache access begins in the second half cycle. If the access hits in the cache, the unit aligns the data and forwards it to the rename buffer and the execution units in the second cycle. If the access misses, the unit places the instruction and its real address in the four-entry load queue. When a load miss completes, it accesses the cache a second time. If the load is still a miss, the unit moves it to the load miss register while reloading the missing cache line. This permits a second load miss to access the cache and to initiate the second cache line reload before the first is brought in.

The unit calculates the memory address of a store and translates it in the first cycle. It does not write the data to the cache, however, until after the store instruction completes. The unit places the instruction and its real address in the six-entry store queue. Since the data cache is not accessed in the second cycle, it is available for an earlier store from the store queue (if necessary) or load miss from the load queue (if necessary).

When a store instruction completes, the load/store unit marks it *completed* in the store queue so that instruction completion can continue without waiting for storage to the cache or memory. If the store hits in the cache, the unit writes it to the cache and removes it from the store queue. If the store is a miss, the unit will bypass it in the store queue to allow later stores to take place while cache reloading proceeds. Multiple store misses can be bypassed in the store queue.

Figure 6 shows the store queue structure. Four pointers identify the state of the store instructions in the circular store queue. When a store has finished execution (or successful translation), the load/store unit places it in the finished state. When it completes, the finish pointer advances to place it in the completed state. When it is committed to cache or memory, the completion pointer advances to place it in the committed state. If the store hits in the cache, advancing the commit pointer removes it from the queue. If the store is a miss, the commit pointer does not advance until the missing cache line is reloaded and the store is written to the cache. While the cache line is being reloaded, the next store indicated by the completion pointer can access the cache. If this store hits in the cache, the unit removes it from the queue. If it misses, another cache line reload begins.

Caches. The 604 provides separate instruction and data caches to allow simultaneous instruction and data accesses every cycle. Both 16-Kbyte caches provide byte parity protection and a four-way set-associative organization with 32-byte lines. They are indexed with physical addresses, have physical tags, and make use of the least recently used replacement algorithm.

The instruction cache provides a 16-byte interface to the fetch unit to support the four-instruction dispatch design. This nonblocking cache allows subsequent instructions to be fetched while a prior cache line is being reloaded. This design is particularly beneficial if the missing cache line belongs in a mispredicted path, since it allows the correct instructions to be accessed immediately after a branch misprediction recovery. The instruction cache also provides streaming, a mechanism to forward instructions as they are received from off-chip cache or memory.

The instruction cache does not maintain coherency; instead, the architecture provides a set of instructions for software to manage coherency. In particular, the instruction cache block invalidate (ICBI) instruction causes all copies of the addressed cache line to be invalidated in the system. The ICBI generates an invalidation request, to which all coherent caches must comply.

The data cache contains a 64-bit data interface to the load/store unit for data access, MMU for tablewalking, and bus interface unit (BIU) for cache line reloading and snooping. (The architecture specifies an algorithm to traverse page table entries that define the virtual-to-physical memory mappings. "Tablewalk" refers to a hardware implementation of the algorithm.) The data cache's two copy-back buffers support nonblocking cache operations. A copy-back buffer holds a dirty (modified) cache line that is being replaced or that hits on a snoop request. The data cache moves an entire cache line into a copy-back buffer in one cycle to minimize the

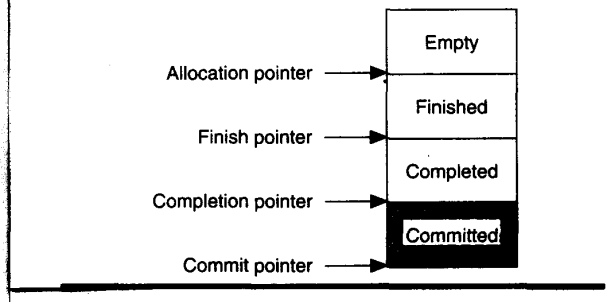


Figure 6. Store queue structure.

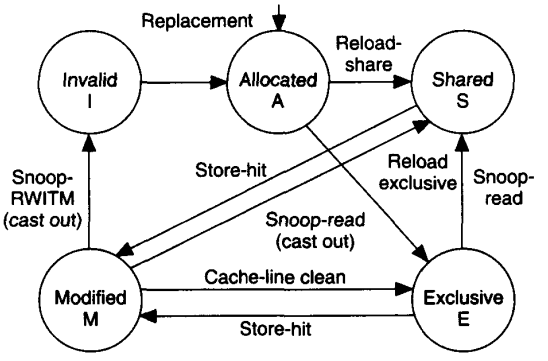


Figure 7. Simplified data cache state diagram.

cycles the cache is unavailable.

The MESI (modified, exclusive, shared, invalid) coherency protocol defined in the *PowerPC 60x Processor Interface Specification* keeps the data cache coherent.[12] A duplicated data tag array supports two-cycle snoop response with minimum performance impact to the normal cache operations.

To further support nonblocking cache operations, we've extended the MESI coherency protocol with one more state. As illustrated in a simplified state diagram in Figure 7, the protocol assigns the new state, Allocated, to the block selected to hold the missing cache line. All necessary information for this particular miss, including the address and set number, remains in the memory queue and later completes the cache line reload. The cache line becomes either shared or exclusive, based on the coherency response.

The software can individually disable, invalidate, or lock both instruction and data caches. While a cache is disabled, all accesses bypass it and directly access the off-chip cache or memory. While a cache is locked, it is accessible but its contents cannot be replaced. All cache misses, in this case, are accessed from off chip as cache-inhibited accesses. Coherency is, however, maintained even when a cache is locked. The data cache supports the cache touch instructions, which initiate reloading of the specified cache line if it is not in the cache. These instructions can effectively shorten cache miss rates and latency.

Bus interface unit and memory queues. The 604's BIU implements the *PowerPC 60X Processor Interface Specification* to ensure bus compatibility with the 601 and 603 microprocessors. A split transaction mode allows the address bus to operate independently of the data bus, freeing the data bus during memory wait states. To support the split transaction mode, the BIU uses the address and data buses only during what are known as address tenure and data tenure cycles. The BIU also provides a pipelined mode, in which up to three address tenures can be outstanding before data for the first address is received. If permitted, the BIU will complete one or more write transactions between the address and data tenures of a read transaction. Byte parity protects its 32-bit address and 64-bit data buses.

Figure 8 (next page) shows the address and data queues that implement split and pipelined transaction modes. Four types of memory queues support the four types of operations: line fill, write, copy back, and cache control. For a line-fill operation, the line-fill address from either the instruction or data cache remains in the memory address queue until the address can be sent out in an address tenure. After the address tenure, the address transfers to the line-fill address queue. This releases the address bus for other transactions in the split transaction mode. As each double word for the cache line returns, it moves to the line-fill buffer and also forwards to the load/store unit.

During a write operation the address stays in the memory address queue, and the data in the write buffer, until both the address and data can move out in a write transaction. The size of a write transaction can vary from 1 to 8 bytes to handle nondouble-word aligned writes. Similarly during a copy-back operation, the address remains in the copy-back address queue, and the data in the copy-back buffer, until both the address and data transfer in a 32-byte burst write transaction. For a cache control instruction or a store to a shared cache line, the address stays in the cache control address queue until an address-only transaction broadcasts the cache control command. Since all address queues in the 604 are considered part of the coherent memory system, the BIU checks them against data cache and snoop addresses to ensure data consistency and maintain MESI coherency protocol.

System support features

The 604 provides several features to support robust system design such as instruction and data address breakpoints and single-step and branch instruction tracing facilities for software debugging. It also provides performance-monitoring functions for the system to profile software performance

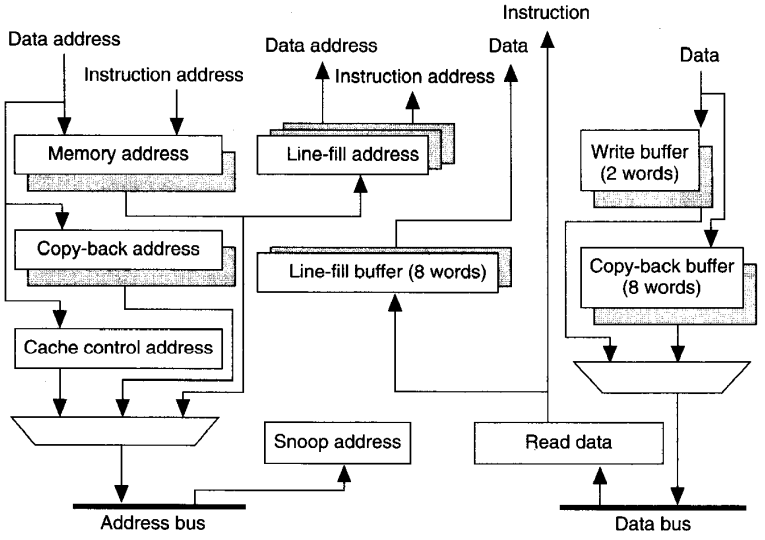


Figure 8. Address and data queue organization.

Table 2. The 604 physical characteristics.

Item	Characteristics
Technology	0.5-µm CMOS, 4 metal layers
Die size	196 mm², 12.4×15.8 mm
Transistor count	3.6 million
Cache size	16-Kbyte I-cache and D-cache
Voltage	3.3V, 5V I/O tolerant
Power dissipation	Less than 10W at 100 MHz
Signal I/Os	171; CMOS/TTL compatible
Package	304-pin CQFP

without additional hardware. These functions can determine many key performance parameters, such as instruction execution rate, branch prediction rate, cache hit rates, and average cache miss latency.

The 604 design follows the level-sensitive scan design methodology to provide high test coverage. As required by LSSD rules, every storage element, except in arrays, connects to a scan chain that starts with a chip input pin and ends on a chip output pin. During test mode, storage elements in a scan chain behave as a shift register that can also capture inputs to exercise a sequential digital network in a combinational manner. The 604's common on-chip processor (COP) provides many functions to control and observe the storage elements. Some of the functions useful for chip and system debugging include setting instruction or data address breakpoints, single stepping, running *N* cycles, and reading and writing system memory locations as well as any storage element within the processor. The COP functions are implemented as an extension to the IEEE-1149.1 specification, and are controlled entirely through that interface.

System designers can configure the 604 processor operating frequency as one, one-and-a-half, two, or three times the system bus frequency. The on-chip phase-locked loop generates the necessary processor clocks from the bus clock. The 604 also provides a nap mode, which clocks only external interrupt detection logic and the phase-locked loop. It enters nap mode under software control and exits from the mode upon detecting an interrupt. The 604 can still service snoop requests if the system asserts the RUN pin to run the clocks while in the nap mode. We estimate nap mode power consumption at less than 0.4 watts.

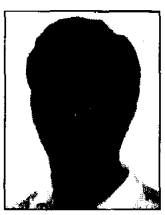
Table 2 lists some of key physical characteristics of the 604. Figure 9 shows the 604 die photo.

DESIGNED TO MEET LOW-COST needs of the personal computer market, the 604 performs well with inexpensive, as well as expensive, memory systems. The 604's large on-chip caches help to maintain performance of well-behaved applications that exhibit localities. For those with erratic behaviors and access patterns, speculative execution guided by dynamic branch prediction helps to reduce on-chip cache miss latency. The nonblocking execution pipelines and the memory queues that decouple the pipelines from memory access further help to reduce the effects of cache misses. The split and pipelined modes use the system bus to provide greater bandwidth while maintaining compatibility with the 601 and 603 microprocessors.

References

1. E. Silha, "The PowerPC Architecture," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, Austin, Tex., 1993.
2. C. Moore, "The PowerPC 601 Microprocessor," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993.
3. B. Burgess et al., "The PowerPC 603 Microprocessor: A High Performance, Low Power, Superscalar RISC Microprocessor,"

Proc. Compcon, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 300-306.
4. S. White et al., "How Does Processor MHz Relate to End-User Performance? Part 1 of 2," *IEEE Micro*, Aug. 1993, pp. 8-16.
5. S. White et al., "How Does Processor MHz Relate to End-User Performance? Part 2 of 2," *IEEE Micro*, Oct. 1993, pp. 79-89.
6. R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J.*, Vol. 11, Jan. 1967, pp. 25-33.
7. G. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Function Unit, Pipelined Computers," *IEEE Trans. Computers*, Mar. 1990, pp. 349-359.
8. J. Smith and A. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. Computer Architecture*, IEEE, Piscataway, N.J., 1985, pp. 36-44.
9. M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
10. J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Jan. 1984, pp. 6-22.
11. V. Popescu et al., "The Metaflow Architecture," *IEEE Micro*, June 1991, pp. 10-13, 63-73.
12. M.S. Allen, et al., *Overview of the PowerPC Bus Interface*, *IEEE Micro*, this issue, pp. 42-51.



S. Peter Song is a senior engineer in the Systems Technology and Architecture Division of IBM. He led the definition of the 604 microarchitecture and later designed the speculative execution, completion, and exception control logic. Song holds BS, MS, and PhD degrees in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.

Marvin Denman is a principal staff engineer in the RISC Microprocessor Division of Motorola, Inc. He has contributed to the definition of the 604 microarchitecture and later designed the fetch and branch-processing logic. Denman holds a BS degree in computer science from Texas A&M University and an MS in electrical engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society.

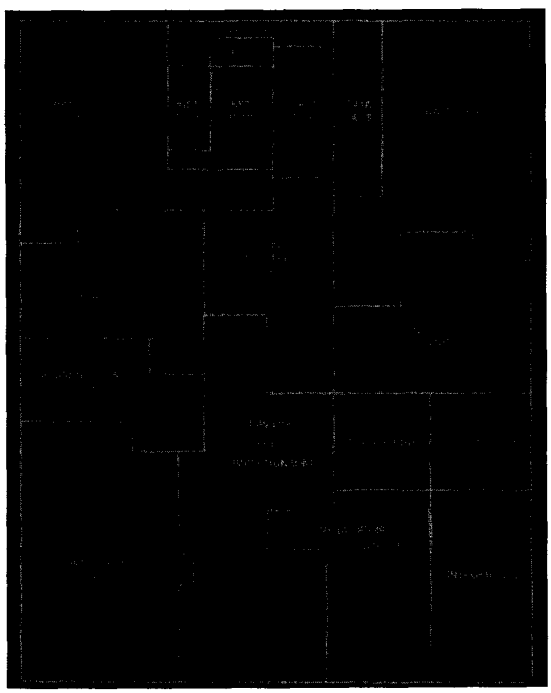


Figure 9. Die photo of the PowerPC 604.

Joe Chang's biography, photograph, and address appear on p. 51 of this issue.

Direct questions concerning this article to S. Peter Song, Somerset Design Center, 11400 Burnet Road, M/S 973, Austin, TX 78758; spsong@ibmoto.com.