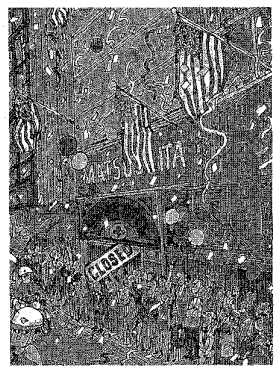
THE MIPS R10000 SUPERSCALAR MICROPROCESSOR

Kenneth C. Yeager

Silicon Graphics, Inc.



Out-of-order superscalar microprocessors execute instructions beyond those stalled by cache misses. This minimizes the time lost due to latency by completing other instructions and initiating subsequent cache refills early.

The Mips R10000 is a dynamic, superscalar microprocessor that implements the 64-bit Mips 4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically issues them to five fully-pipelined, low-latency execution units. Instructions can be fetched and executed speculatively beyond branches. Instructions graduate in order upon completion. Although execution is out of order, the processor still provides sequential memory consistency and precise exception handling.

The R10000 is designed for high performance, even in large, real-world applications with poor memory locality. With speculative execution, it calculates memory addresses and initiates cache refills early. Its hierarchical, nonblocking memory system helps hide memory latency with two levels of set-associative, write-back caches. Figure 1 shows the R10000 system configuration, and the R10000 box lists its principal features.

Out-of-order superscalar processors are inherently complex. To cope with this complexity, the R10000 uses a modular design that locates much of the control logic within regular structures, including the active list, register map tables, and instruction queues.

Design rationale

Memory bandwidth and latency limit the performance of many programs. Because packaging and system costs constrain these resources, the processor must use them efficiently.

The R10000 implements register mapping and nonblocking caches, which complement each other to overlap cache refill operations. Thus, if an instruction misses in the cache, it must wait for its operand to be refilled, but other instructions can continue out of order. This increases memory use and reduces effective latency, because refills begin early and up to four refills proceed in parallel while the processor executes other instructions. This type of cache design is called "nonblocking," because cache refills do not block subsequent accesses to other cache lines.

Processors rely on compiler support to optimize instruction sequencing. This technique is especially effective for data arrays, such as those used in many floating-point applications. For these arrays, a sophisticated compiler can optimize performance for a specific cache organization. However, compiler optimization is less effective for the scalar values of many integer applications, because the compiler has difficulty predicting which instructions will generate cache misses.

The R10000 design includes complex hardware that dynamically reorders instruction execution based on operand availability. This hardware immediately adapts whenever cache misses delay instructions. The processor looks ahead up to 32 instructions to find possible parallelism. This instruction window is large enough to hide most of the latency for refills from the secondary cache. However, it can hide only a fraction of main memory latency, which is typically much longer.

It is relatively easy to add nonblocking caches to an out-of-order processor, because it already contains mechanisms that coordinate dependencies between instructions.

Implementation

We implemented the initial R10000 microprocessor using 0.35-μm CMOS technology on a 16.64×17.934-mm chip. This 298-mm^2 chip contains 6.8 million transistors, including 4.4 million in its primary cache arrays. We implemented data paths and time-critical control logic in full custom design, making wide use of dynamic and latch-based logic. We synthesized the less critical circuits using static register-based logic.

0272-1732/96/$5.00 © 1996 IEEE

Petitioner Lenovo (United States) Inc. - Ex. 1015

Mips R10000

This processor features a four-way superscalar RISC processor that

- fetches and decodes four instructions per cycle,
- speculatively executes beyond branches, with a four-entry branch stack,
- uses dynamic out-of-order execution,
- implements register renaming using map tables, and
- achieves in-order graduation for precise exceptions.

Five independent pipelined execution units include

- a nonblocking load/store unit,
- dual 64-bit integer ALUs,
- 64-bit, IEEE Std 754-1985 floating-point units,
- a pipelined adder with two-cycle latency, and
- a pipelined multiplier with two-cycle latency.

The hierarchical, nonblocking memory subsystem includes

- on-chip, two-way-associative primary caches—
  32-Kbyte instruction cache and
  32-Kbyte, two-way interleaved data cache,
- an external, two-way-associative secondary cache—
  128-bit-wide, synchronous static RAM, and
- a 64-bit multiprocessor system interface with split transaction protocol.

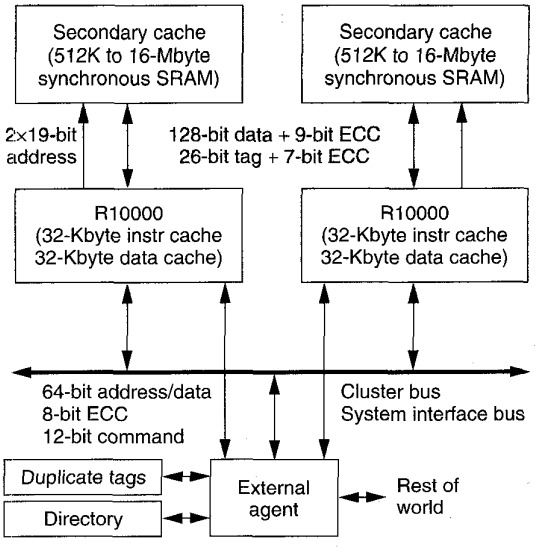


Figure 1. System configuration. The cluster bus directly connects as many as four chips.

System flexibility

Alternate configurations allow the R10000 to operate in a wide range of systems—as a uniprocessor or in a multiprocessor cluster. The system maintains cache coherency using either snoopy or directory-based protocols. The R10000's secondary cache ranges from 512 Kbytes to 16 Mbytes.

Operation overview

Figure 2 (next page) shows a block diagram and a pipeline timing diagram for the R10000. There are six nearly independent pipelines.

The instruction fetch pipeline occupies stages 1 through 3. In stage 1, the R10000 fetches and aligns the next four instructions. In stage 2, it decodes and renames these instructions and also calculates target addresses for jump and branch instructions. In stage 3, it writes the renamed instructions into the queues and reads the busy-bit table to determine if the operands are initially busy. Instructions wait in the queues until all their operands are ready.

The five execution pipelines begin when a queue issues an instruction in stage 3. The processor reads operands from the register files during the second half of stage 3, and execution begins in stage 4. The integer pipelines occupy one stage, the load pipeline occupies two, and the floating-point pipelines occupy three. The processor writes results into the register file during the first half of the next stage.

The integer and floating-point sections have separate instruction queues, register files, and data paths. This separation reduces maximum wire lengths and allows fully parallel operation. Together, the two register files need more free registers than would a combined unit, but they are physically smaller, because each register has fewer read and write ports.

Instruction fetch

For good performance, the processor must fetch and decode instructions at a higher bandwidth than it can execute them. It is important to keep the queues full, so they can look ahead to find instructions to issue out of order. Ultimately, the processor fetches more instructions than it graduates, because it discards instructions occurring after mispredicted branches.

The processor fetches instructions during stage 1, as shown in Figure 3. The instruction cache contains address tag and data sections. To implement two-way set associativity, each section has two parallel arrays. The processor compares the two tag addresses to translated physical addresses to select data from the correct way. The small, eight-entry instruction translation look-aside buffer (TLB) contains a subset of the translations in the main TLB.

The processor fetches four instructions in parallel at any word alignment within a 16-word instruction cache line. We implemented this feature with a simple modification to the cache's sense amplifiers, as shown in Figure 4. Each sense amplifier is as wide as four bit columns in the memory array, and a 4-to-1 multiplexer selects one column (which represents one instruction) for fetching. The R10000 fetches unaligned instructions using a separate select signal for each instruction. These instructions rotate, if necessary, so that they are decoded in order. This ordering reduces the amount of dependency logic.

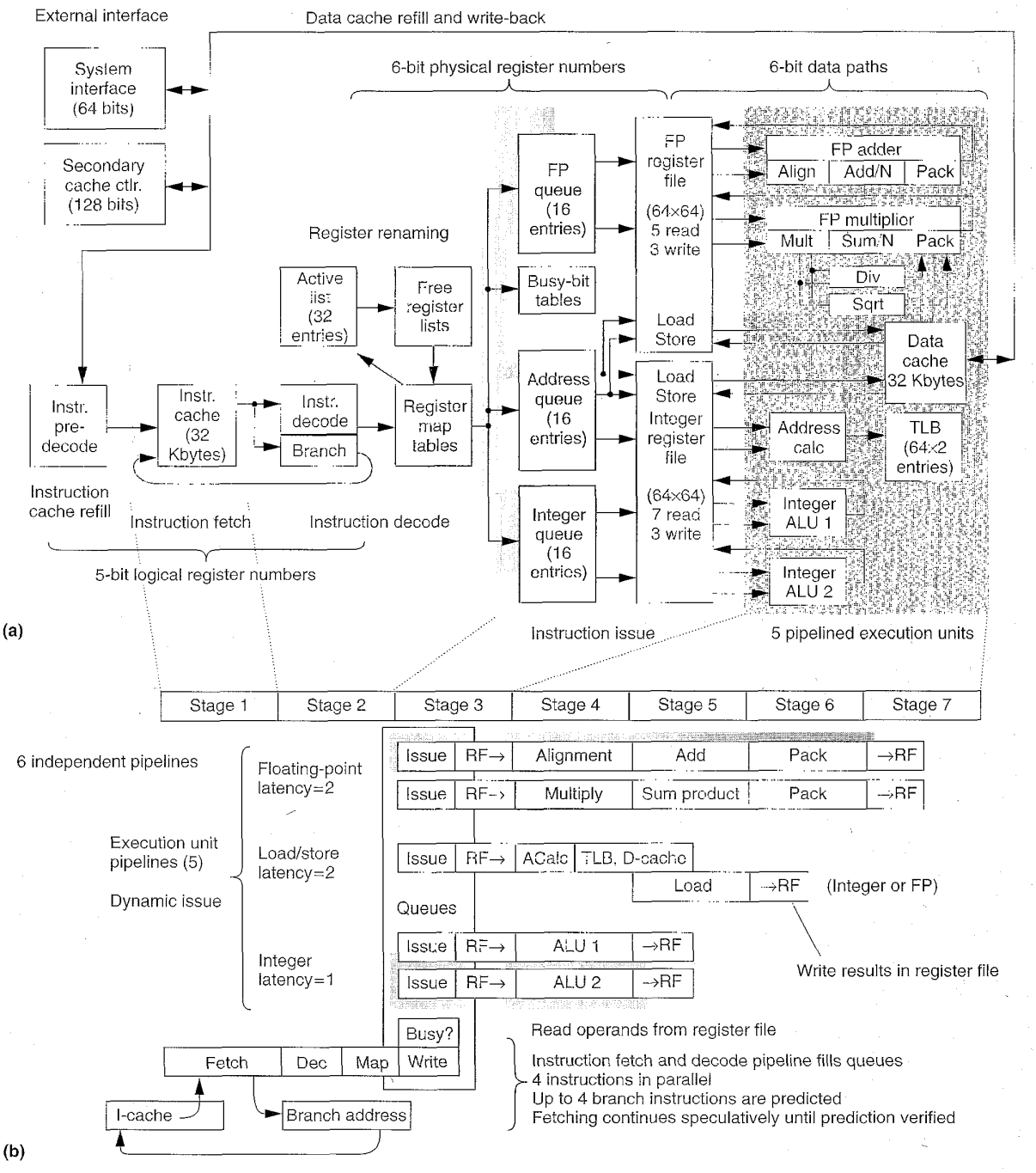


Figure 2. R10000 block diagram (a) and pipeline timing diagram (b). The block diagram shows pipeline stages left to right to correspond to pipeline timing.

Usually, the processor decodes all four instructions during the next cycle, unless the queues or active list is full. Instructions that are not immediately decoded remain in an eight-word instruction buffer, simplifying timing for sequential fetching.

Branch unit

Branch instructions occur frequently and must execute quickly. However, the processor cannot usually determine the branch direction until several or even many cycles after

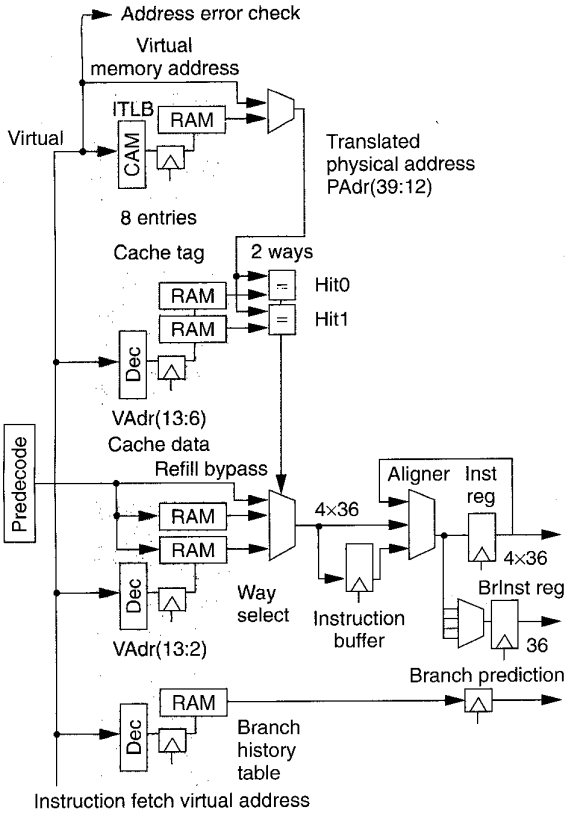


Figure 3. Instruction fetch, pipeline stage 1.

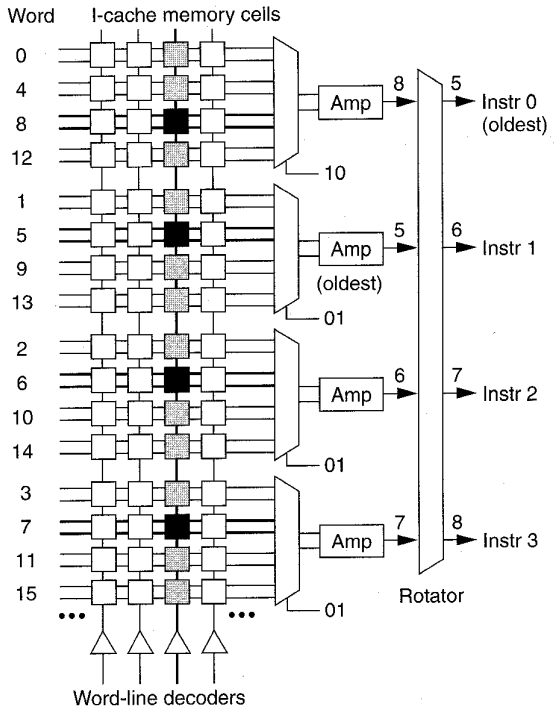


Figure 4. Unaligned fetching from instruction cache.

decoding the branch. Thus, the processor predicts the direction a conditional branch will take and fetches instructions speculatively along the predicted path. The prediction uses a 2-bit algorithm based on a 512-entry branch history table. This table is indexed by bits 11:3 of the address of the branch instruction. Simulations show an 87 percent prediction accuracy for Spec92 integer programs.

In the Mips architecture, the processor executes the instruction immediately following a jump or branch before executing instructions at the target address. In a pipelined scalar processor, this delay slot instruction can be executed for free, while the target instruction is read from the cache. This technique improved branch efficiency in early RISC microprocessors. For a superscalar design, however, it has no performance advantage, but we retained the feature in the R10000 for compatibility.

When the program execution takes a jump or branch, the processor discards any instructions already fetched beyond the delay slot. It loads the jump's target address into the program counter and fetches new instructions from the cache after a one-cycle delay. This introduces one "branch bubble" cycle, during which the R10000 decodes no instructions.

Branch stack. When it decodes a branch, the processor saves its state in a four-entry branch stack. This contains the alternate branch address, complete copies of the integer and floating-point map tables, and miscellaneous control bits. Although the stack operates as a single logical entity, it is physically distributed near the information it copies.

When the branch stack is full, the processor continues decoding only until it encounters the next branch instruction. Decoding then stalls until resolution of one of the pending branches.

Branch verification. The processor verifies each branch prediction as soon as its condition is determined, even if earlier branches are still pending. If the prediction was incorrect, the processor immediately aborts all instructions fetched along the mispredicted path and restores its state from the branch stack.

Fetching along mispredicted paths may initiate unneeded cache refills. In this case, the instruction cache is nonblocking, and the processor fetches the correct path while these refills complete. It is easier and often desirable to complete such refills, since the program execution may soon take the other direction of the branch, such as at the end of a loop.

A 4-bit branch mask, corresponding to entries within the branch stack, accompanies each instruction through the queues and execution pipelines. This mask indicates which pending branches the instruction depends on. If any of these branches was mispredicted, the processor will abort the instruction when that branch decision is reversed. Whenever the R10000 verifies a branch, it resets the corresponding mask bits throughout the pipeline.

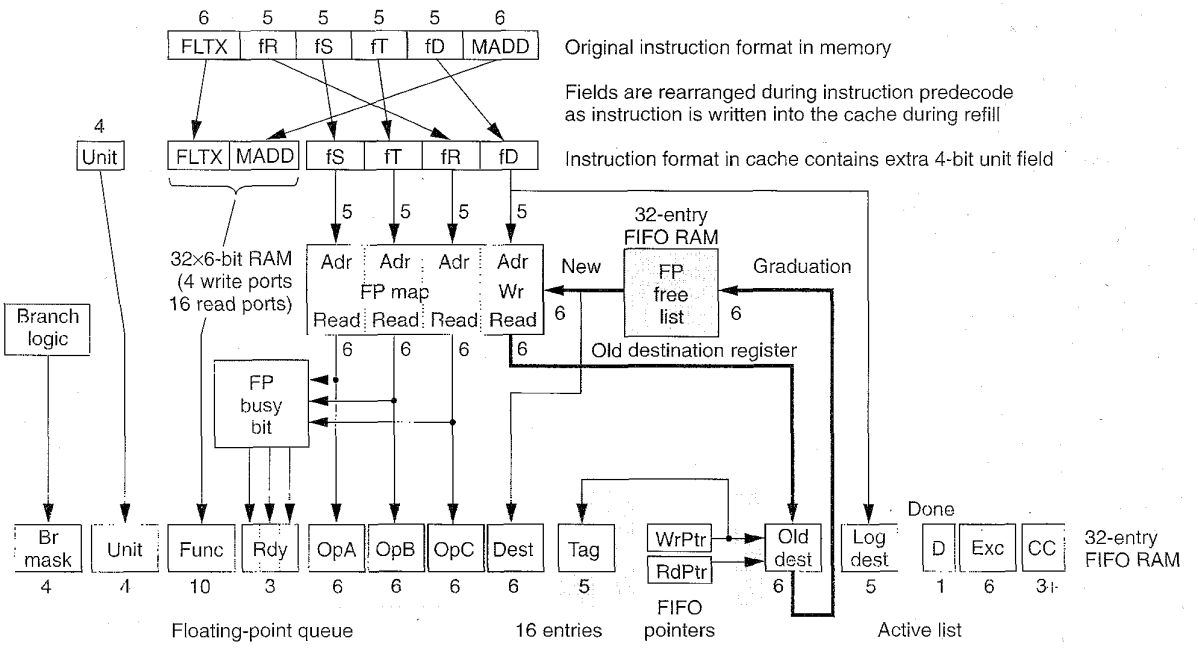


Figure 5. Register renaming, pipeline stage 2. The R10000 rearranges fields during instruction predecode as it writes the instruction into the cache during refill. The instruction format in the cache contains an extra 4-bit unit field.

Decode logic

The R10000 decodes and maps four instructions in parallel during stage 2 and writes them into the appropriate instruction queue at the beginning of stage 3.

Decoding stops if the active list or a queue becomes full, but there are very few decode restrictions that depend on the type of instructions being decoded. The principal exception involves integer multiply and divide instructions. Their results go into two special registers—Hi and Lo. No other instructions have more than one result register. We did not add much logic for these infrequently used instructions; instead, they occupy two slots in the active list. Once the processor decodes such an instruction, it does not decode any subsequent instructions during the same cycle. (In addition, it cannot decode an integer multiply or divide as the fourth instruction in a cycle.)

Instructions that read or modify certain control registers execute serially. The processor can only execute these instructions, which are mostly restricted to rare cases in the kernel operating system mode, when the pipeline is empty. This restriction has little effect on overall performance.

Register mapping

Figure 5 illustrates the R10000's register-mapping hardware. To execute instructions out of their original program order, the processor must keep track of dependencies on register operands, memory addresses, and condition bits. (The condition bits are eight bits in the status register set by floating-point compare instructions.) To determine register dependencies, the R10000 uses register renaming. It determines memory address dependencies in the address queue. It sets each condition bit operand during decode if its value is known. If not, it renames the bit with the tag of the floating-point compare instruction that will eventually set its value.

From a programmer's perspective, instructions execute sequentially in the order the program specifies. When an instruction loads a new value into its destination register, that new value is immediately available for subsequent instructions to use. However, a superscalar processor performs several instructions simultaneously, and their results are not immediately available for subsequent instructions. Frequently, the next sequential instruction must wait for its operands to become valid, but the operands of later instructions may already be available.

The R10000 achieves higher performance by executing these later instructions out of order, but this reordering is invisible to the programmer. Any result it generates out of order is temporary until all previous instructions have completed. Then this instruction graduates, and its result is committed as the processor's state. Until it graduates, an instruction can be aborted if it follows an exception or a mispredicted branch. The previous contents of its logical destination register can be retrieved by restoring its previous mapping.

In most processors, there is no distinction between logical register numbers, which are referenced within instruction fields, and physical registers, which are locations in the hardware register file. Each instruction field directly addresses the corresponding register. Our renaming strategy, however, dynamically maps the logical-register numbers into physical-register numbers. The processor writes each new

result into a new physical register. After mapping, the processor determines dependencies simply by comparing physical-register numbers; it no longer must consider instruction order. Again, the existence of these physical registers and the mapping of logical registers to physical registers are invisible to the programmer.

The R10000 executes instructions dynamically after resolving all dependencies on previous instructions. That is, each instruction must wait until all its operands have been computed. Then the R10000 can execute that instruction, regardless of the original instruction sequence. To execute instructions correctly, the processor must determine when each operand register is ready. This can be complicated, because logical-register numbers may be ambiguous in terms of operand values. For example, if several instructions specifying the same logical register are simultaneously in the pipeline, that register may load repeatedly with different values.

There must be more physical than logical registers, because physical registers contain both committed values and temporary results for instructions that have completed but not yet graduated. A logical register may have a sequence of values as instructions flow through the pipeline. Whenever an instruction modifies a register, the processor assigns a new physical register to the logical destination register and stores these assignments in register map tables. As the R10000 decodes each instruction, it replaces each of the logical-register fields with the corresponding physical-register number.

Each physical register is written exactly once after each assignment from the free list. Until it is written, it is busy. If a subsequent instruction needs its value, that instruction must wait until it is written. After the register is written, it is ready, and its value does not change. When a subsequent instruction changes the corresponding logical register, that result is written into a new physical register. When this subsequent instruction graduates, the program no longer needs the old value, and the old physical register becomes free for reuse. Thus, physical registers always have unambiguous values.

There are 33 logical (numbers 1 through 31, Hi, and Lo) and 64 physical integer registers. (There is no integer register 0. A zero operand field indicates a zero value; a zero destination field indicates an unstored result.) There are 32 logical (numbers 0 through 31) and 64 physical floating-point registers.

Register map tables. Separate register files store integer and floating-point registers, which the processor renames independently. The integer and floating-point map tables contain the current assignments of logical to physical registers. The processor selects logical registers using 5-bit instruction fields. Six-bit addresses in the corresponding register files identify the physical registers.

The floating-point table maps registers f0 through f31 in a 32×6-bit multiport RAM. The integer table maps registers r1 through r31, Hi, and Lo in a 33×6-bit multiport RAM. (There is special access logic for the Hi and Lo registers, the implicit destinations of integer multiply and divide instructions.)

These map tables have 16 read ports and four write ports which map four instructions in parallel. Each instruction reads the mappings for three operand registers and one destination register. The processor writes the current operand mappings and new destination mapping into the instruction queues, while the active list saves previous destination mappings.

The R10000 uses 24 five-bit comparators to detect dependencies among the four instructions decoded in parallel. These comparators control bypass multiplexers, which replace dependent operands with new assignments from the free lists.

Free lists. The integer and floating-point free lists contain lists of currently unassigned physical registers. Because the processor decodes and graduates up to four instructions in parallel, these lists consist of four parallel, eight-deep, circular FIFOs.

Active list. The active list records all instructions currently active within the processor, appending each instruction as the processor decodes it. The list removes instructions when they graduate, or if a mispredicted branch or an exception causes them to abort. Since up to 32 instructions can be active, the active list consists of four parallel, eight-deep, circular FIFOs.

Each instruction is identified by 5-bit tag, which equals an address in the active list. When an execution unit completes an instruction, it sends its tag to the active list, which sets its done bit.

The active list contains the logical-destination register number and its old physical-register number for each instruction. An instruction's graduation commits its new mapping, so the old physical register can return to the free list for reuse.

When an exception occurs, however, subsequent instructions never graduate. Instead, the processor restores old mappings from the active list. The R10000 unmaps four instructions per cycle—in reverse order, in case it renamed the same logical register twice. Although this is slower than restoring a branch, exceptions are much rarer than mispredicted branches. The processor returns new physical registers to the free lists by restoring their read pointers.

Busy-bit tables. For each physical register, integer and floating-point busy-bit tables contain a bit indicating whether the register currently contains a valid value. Each table is a 64×1-bit multiport RAM. The tables sets a bit busy when the corresponding register leaves the free list. It resets the bit when an execution unit writes a value into this register. Twelve read ports determine the status of three operand registers for each of four newly decoded instructions. The queues use three other ports for special-case instructions, such as moves between the integer and floating-point register files.

Instruction queues

The R10000 puts each decoded instruction, except jumps and no operation NOPs, into one of three instruction queues, according to type. Provided there is room, the queues can accept any combination of new instructions.

The chip's cycle time constrained our design of the queues. For instance, we dedicated two register file read ports to each issued instruction to avoid delays arbitrating and multiplexing operand buses.

Integer queue. The integer queue contains 16 entries in no specific order and allocates an entry to each integer instruction as it is decoded. The queue releases the entry as soon as it issues the instruction to an ALU.

Instructions that only one of the ALUs can execute have priority for that ALU. Thus, branch and shift instructions have priority for ALU 1; integer multiply and divide have priority

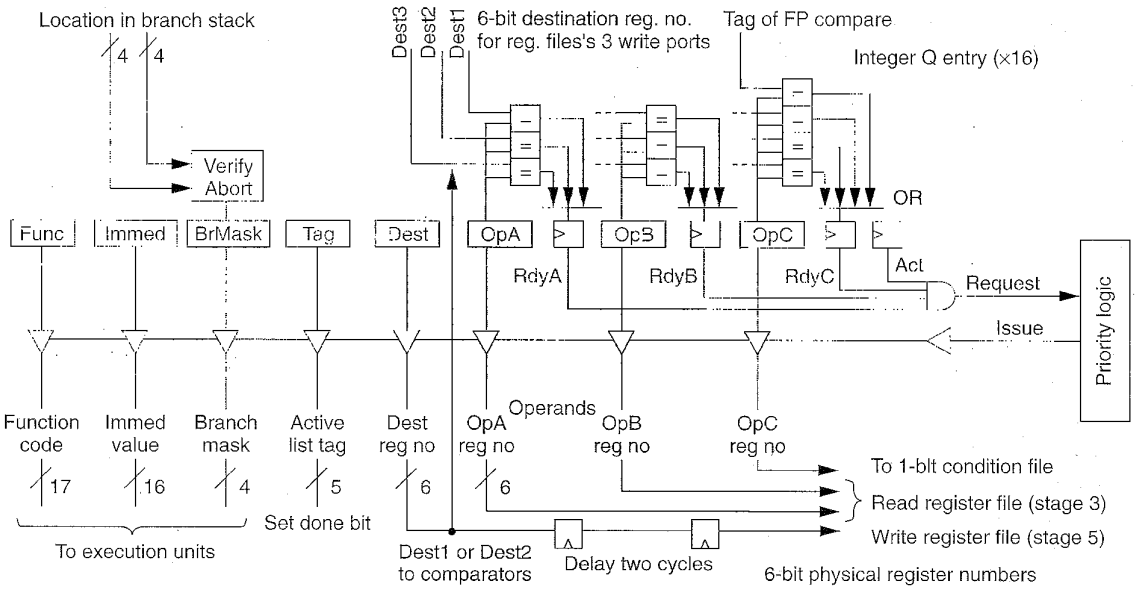


Figure 6. Integer instruction queue, showing only one issue port. The queue issues two instructions in parallel.

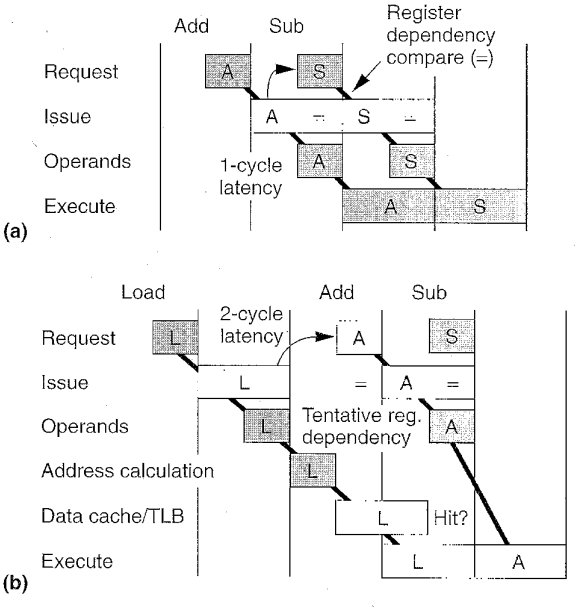


Figure 7. Releasing register dependency in the integer queue (a) and tentative issue of an instruction dependent on an earlier load instruction (b).

for ALU 2. For simplicity, location in the queue rather than instruction age determines priority for issue. However, a round-robin request circuit raises the priority of old instructions requesting ALU 2.

Figure 6 shows the contents of an integer queue entry. It contains three operand select fields, which contain physical-register numbers. Each field contains a ready bit, initialized from the busy-bit table. The queue compares each select with the three destination selects corresponding to write ports in the integer register file. Any comparator match sets the corresponding ready bit. When all operands are ready, the queue can issue the instruction to an execution unit.

Operand C contains either a condition bit value or the tag of the floating-point compare instruction that will set its value. In total, each of the 16 entries contains ten 6-bit comparators.

The queue issues the function code and immediate values to the execution units. The branch mask determines if the instruction aborted because of a mispredicted branch. The tag sets the done bit in the active list after the processor completes the instruction.

The single-cycle latency of integer instructions complicated integer queue timing and logic. In one cycle, the queue must issue two instructions, detect which operands become ready, and request dependent instructions. Figure 7a illustrates this process.

To achieve two-cycle load latency, an instruction that depends on the result of an integer load must be issued tentatively, assuming that the load will be completed successfully. The dependent instruction is issued one cycle before it is executed, while the load reads the data cache. If the load fails, because of a cache miss or a dependency, the issue of the dependent instruction must be aborted. Figure 7b illustrates this process.

Address queue. The address queue contains 16 entries. Unlike the other two queues, it is a circular FIFO that preserves the original program order of its instructions. It allocates an entry when the processor decodes each load or store

instruction and removes the entry after that instruction graduates. The queue uses instruction order to determine memory dependencies and to give priority to the oldest instruction.

When the processor restores a mispredicted branch, the address queue removes all instructions decoded after that branch from the end of the queue by restoring the write pointer. The queue issues instructions to the address calculation unit using logic similar to that used in the integer queue, except that this logic contains only two register operands.

The address queue is more complex than the other queues. A load or store instruction may need to be retried if it has a memory address dependency or misses in the data cache.

Two 16-bit×16-bit matrixes track dependencies between memory accesses. The rows and columns correspond to the queue's entries. The first matrix avoids unnecessary cache thrashing by tracking which entries access the same cache set (virtual addresses 13:5). Either way in a set can be used by instructions that are executed out of order. But if two or more queue entries address different lines in the same cache set, the other way is reserved for the oldest entry that accesses that set. The second matrix tracks instructions that load the same bytes as a pending store instruction. It determines this match by comparing double-word addresses and 8-bit byte masks.

Whenever the external interface accesses the data cache, the processor compares its index to all pending entries in the queue. If a load entry matches a refill address, it passes the refill data directly into its destination register. If an entry matches an invalidated command, that entry's state clears.

Although the address queue executes load and store instructions out of their original order, it maintains sequential-memory consistency. The external interface could violate this consistency, however, by invalidating a cache line after it was used to load a register, but before that load instruction graduates. In this case, the queue creates a soft exception on the load instruction. This exception flushes the pipeline and aborts that load and all later instructions, so the processor does not use the stale data. Then, instead of continuing with the exception, the processor simply resumes normal execution, beginning with the aborted load instruction. (This strategy guarantees forward progress because the oldest instruction graduates immediately after completion.)

Store instructions require special coordination between the address queue and active list. The queue must write into the data cache precisely when the store instruction graduates.

The Mips architecture simulates atomic memory operations with load-link (LL) and store-conditional (SC) instruction pairs. These instructions do not complicate system design, because they do not need to lock access to memory. In a typical sequence, the processor loads a value with an LL instruction, tests and modifies it, and then conditionally stores it with an SC instruction. The SC instruction writes into memory only if there was no conflict for this value and the link word remains in the cache. The processor loads its result register with a one or zero to indicate if memory was written.

Floating-point queue. The floating-point queue contains 16 entries. It is very similar to the integer queue, but it does not contain immediate values. Because of extra wiring delays, floating-point loads have three-cycle latency.

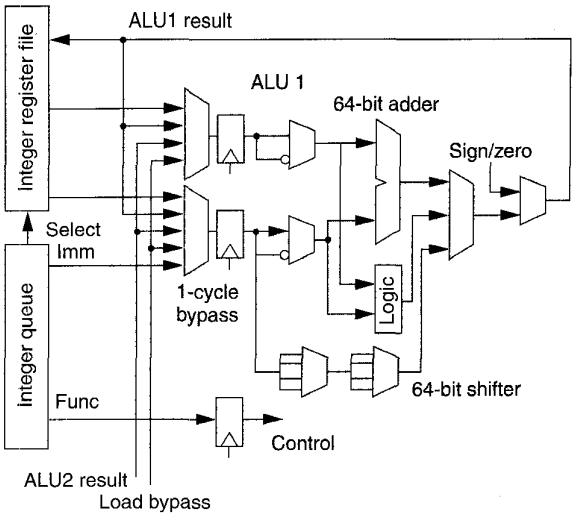


Figure 8. ALU 1 block diagram.

Register files

Integer and floating-point register files each contain 64 physical registers. Execution units read operands directly from the register files and write results directly back. Results may bypass the register file into operand registers, but there are no separate structures, such as reservation stations or reorder buffers in the wide data paths.

The integer register file has seven read ports and three write ports. These include two dedicated read ports and one dedicated write port for each ALU and two dedicated read ports for the address calculate unit. The integer register's seventh read port handles store, jump-register, and move-to-floating-point instructions. Its third write port handles load, branch-and-link, and move-from-floating-point instructions.

A separate 64-word×1-bit condition file indicates if the value in the corresponding physical register is non-zero. Its three write ports operate in parallel with the integer register file. Its two read ports allow integer and floating-point conditional-move instructions to test a single condition bit instead of an entire register. This file used much less area than two additional read ports in the register file.

The floating-point register file has five read and three write ports. The adder and multiplier each have two dedicated read ports and one dedicated write port. The fifth read port handles store and move instructions; the third write port handles load and move instructions.

Integer execution units

During each cycle, the integer queue can issue two instructions to the integer execution units.

Integer ALUs. Each of the two integer ALUs contains a 64-bit adder and a logic unit. In addition, ALU 1 contains a 64-bit shifter and branch condition logic, and ALU 2 contains a partial integer multiplier array and integer-divide logic. Figure 8 shows the ALU 1 block diagram. Each ALU has two 64-bit operand registers that load from the register file. To

Table 1. Latency and repeat rates for integer instructions.

Unit	Latency (cycles)	Repeat rate (cycles)	Instruction
Either ALU	1	1	Add, subtract, logical, move Hi/Lo, trap
ALU 1	1	1	Integer branches
ALU 1	1	1	Shift
ALU 1	1	1	Conditional move
ALU 2	5/6	6	32-bit multiply
	9/10	10	64-bit multiply (to Hi/Lo registers)
ALU 2	34/35	35	32-bit divide
	66/67	67	64-bit divide
Load/store	2	1	Load integer
	—	1	Store integer

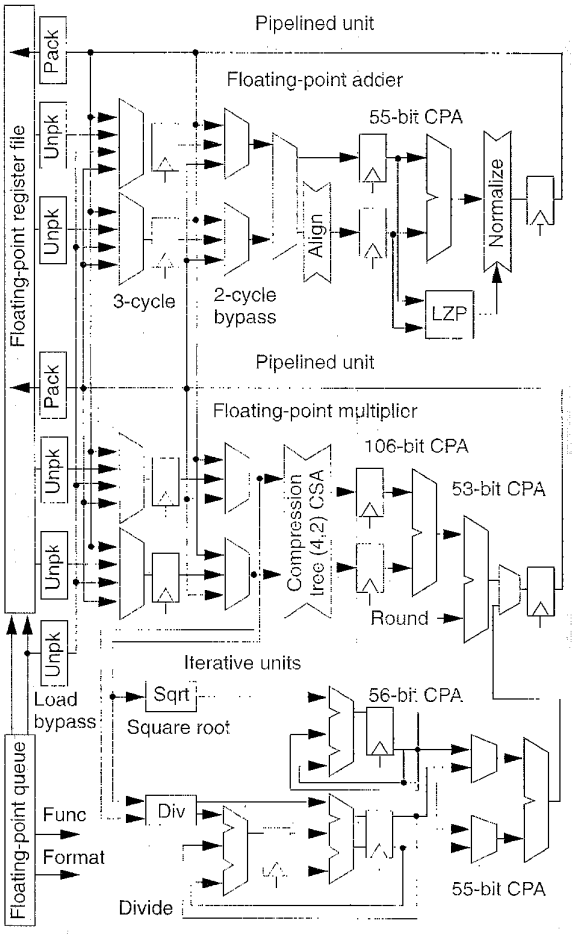


Figure 9. Floating-point execution units block diagram.

achieve one-cycle latency, the three write ports of the register file bypass into the operand register.

The integer queue controls both ALUs. It provides function codes, immediate values, bypass controls, and so forth.

Integer multiplication and division. ALU 2 iteratively computes integer multiplication and division. As mentioned earlier, these instructions have two destination registers, Hi and Lo. For multiply instructions, Hi and Lo contain the high and low halves of a double-precision product. For divide instructions, they contain the remainder and quotient.

ALU 2 computes integer multiplication using Booth's algorithm, which generates a partial product for each two bits of the multiplier. The algorithm generates and accumulates four partial products per cycle. ALU 2 is busy for the first cycle after the instruction is issued, and for the last two cycles to store the result.

To compute an integer division, ALU 2 uses a nonrestoring algorithm that generates one bit per cycle. ALU 2 is busy for the entire operation.

Table 1 lists latency and repeat rates for common integer instructions.

Floating-point execution units

Figure 9 shows the mantissa data path for these units. (Exponent logic is not shown.) The adder and multiplier have three-stage pipelines. Both units are fully pipelined with a single-cycle repeat rate. Results can bypass the register file for either two-cycle or three-cycle latency. All floating-point operations are issued from the floating-point queue.

Floating-point values are packed in IEEE Std 754 single- or double-precision formats in the floating-point register file. The execution units and all internal bypassing use an unpacked format that explicitly stores the hidden bit and separates the 11-bit exponent and 53-bit mantissa. Operands are unpacked as they are read, and results are packed before they are written back. Packing and unpacking are implemented with two-input multiplexers that select bits according to single or double precision formats. This logic is between the execution units and register file.

Floating-point adder. The adder does floating-point addition, subtraction, compare, and conversion operations. Its first stage subtracts the operand exponents, selects the larger operand, and aligns the smaller mantissa in a 55-bit right shifter. The second stage adds or subtracts the mantissas, depending on the operation and the signs of the operands.

A magnitude addition can produce a carry that requires a one-bit shift right for post normalization. Conceptually, the processor must round the result after generating it. To avoid extra delay, a dual, carry-chain adder generates both +1 and +2 versions of the sum. The processor selects the +2 chain if the operation requires a right shift for post normalization.

On the other hand, a magnitude subtraction can cause massive cancellation, producing high-order zeros in the result. A leading-zero prediction circuit determines how many high-order zeros the subtraction will produce. Its out-

put controls a 55-bit left shifter that normalizes the result.

Floating-point multiplier. The multiplier does floating-point multiplication in a full double-precision array. Because it is slightly less busy than the adder, it also contains the multiplexers that perform move and conditional-move operations.

During the first cycle, the unit Booth-encodes the 53-bit mantissa of the multiplier and uses it to select 27 partial products. (With Booth encoding, only one partial product is needed for each two bits of the multiplier.) A compression tree uses an array of (4, 2) carry-save adders, which sum four bits into two sum and carry outputs. During the second cycle, the resulting 106-bit sum and carry values are combined using a 106-bit carry-propagate adder. A final 53-bit adder rounds the result.

Floating-point divide and square root. Two independent iterative units compute floating-point division and square-root operations. Each unit uses an SRT algorithm that generates two bits per iteration stage. The divide unit cascades two stages within each cycle to generate four bits per cycle.

These units share register file ports with the multiplier. Each operation preempts two cycles. The first cycle issues the instruction and reads its operands from the register file. At the end of the operation, the unit uses the second cycle to write the result into the register file.

Table 2 lists latency and repeat rates for common floating-point instructions.

Table 2. Latency and repeat rates for floating-point instructions.

Unit	Latency (cycles)	Repeat rate (cycles)	Instruction
Add	2	1	Add, subtract, compare
Multiply	2	1	Integer branches
Divide	12	14	32-bit divide
	19	21	64-bit divide
Square root	18	20	32-bit square root
	33	35	64-bit square root
Load/store	3	1	Load floating-point value
	—	1	Store floating-point value

Memory hierarchy

Memory latency has a major impact on processor performance. To run large programs effectively, the R10000 implements a nonblocking memory hierarchy with two levels of set-associative caches. The on-chip primary instruction and data caches operate concurrently, providing low latency and high bandwidth. The chip also controls a large external secondary cache. All caches use a least-recently-used (LRU) replacement algorithm.

Both primary caches use a virtual address index and a physical-address tag. To minimize latency, the processor can access each primary cache concurrently with address translation in its TLB. Because each cache way contains 16 Kbytes (four times the minimum virtual page size), two of the virtual index bits (13:12) might not equal bits in the physical address tag. This technique simplifies the cache design. It works well, as long as the program uses consistent virtual indexes to reference the same page. The processor stores these two virtual address bits as part of the secondary-cache tag. The secondary-cache controller detects any violations and ensures that the primary caches retain only a single copy of each cache line.

Load/store unit. Figure 10 contains a block diagram of the load/store unit and the data cache. The address queue issues load and store instructions to the address calculation unit and the data cache. When the cache is not busy, a load instruction simultaneously accesses the TLB, cache tag array, and cache data array. This parallel access results in two-cycle load latency.

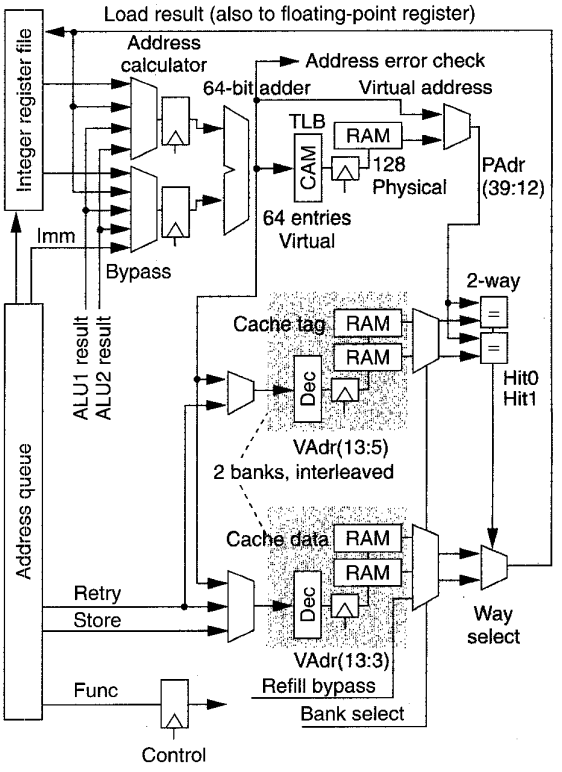


Figure 10. Address calculation unit and data cache block diagram.

Address calculation. The R10000 calculates virtual memory addresses as the sum of two 64-bit registers or the sum of a register and a 16-bit immediate field. Results from the ALUs or the data cache can bypass the register files into the operand registers. The TLB translates these virtual addresses to physical addresses.

Memory address translation (TLB). The Mips-4 architecture defines 64-bit addressing. Practical implementations

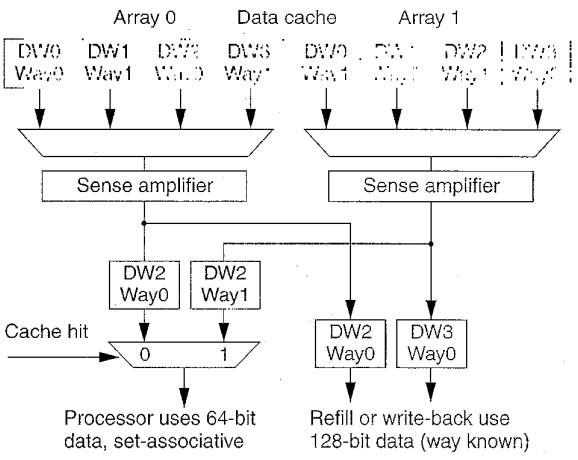


Figure 11. Arrangement of ways in the data cache.

reduce the maximum address width to reduce the cost of the TLB and cache tag arrays. The R10000's fully-associative translation look-aside buffer translates 44-bit virtual addresses into 40-bit physical addresses. This TLB is similar to that of the R4000, but we increased it to 64 entries. Each entry maps a pair of virtual pages and independently selects a page size of any power of 4 between 4 Kbytes and 16 Mbytes. The TLB consists of a content-addressable memory (CAM section), which compares virtual addresses, and a RAM section, which contains corresponding physical addresses.

Primary instruction cache. The 32-Kbyte instruction cache contains 8,192 instruction words, each predecoded into a 36-bit format. The processor can decode this expanded format more rapidly than the original instruction format. In particular, the four extra bits indicate which functional unit should execute the instruction. The predecoding also rearranges operand- and destination-select fields to be in the same position for every instruction. Finally, it modifies several opcodes to simplify decoding of integer or floating-point destination registers.

The processor simultaneously fetches four instructions in parallel from both cache ways, and the cache hit logic selects the desired instructions. These instructions need not be aligned on a quad-word address, but they cannot cross a 16-word cache line (see Figure 3).

Primary data cache. The data cache interleaves two 16-Kbyte banks for increased bandwidth. The processor allocates the tag and data arrays of each bank independently to the four following requesting pipelines:

- external interface (refill data, interventions, and so on),
- tag check for a newly calculated address,
- retrying a load instruction, and
- graduating a store instruction.

To simplify interfacing and reduce the amount of buffering required, the external interface has priority for the arrays it needs. Its requests occur two cycles before cache reads or writes, so the processor can allocate the remaining resources among its pipelines.

The data cache has an eight-word line size, which is a con-

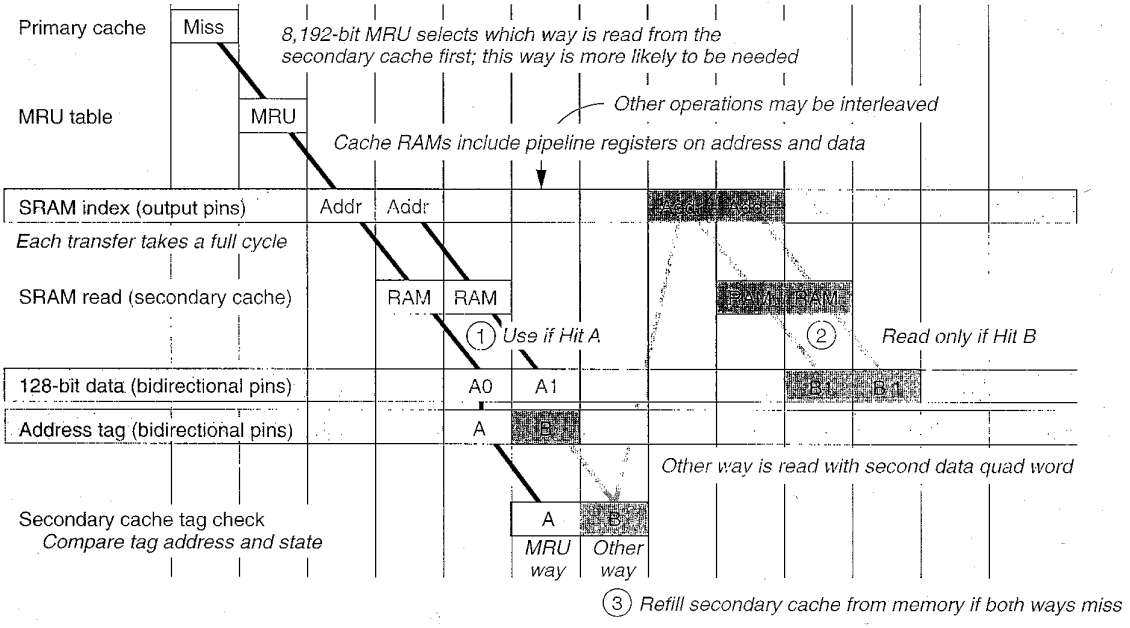


Figure 12. Refill from the set-associative secondary cache. In this example, the secondary clock equals the processor's internal pipeline clock. It may be slower.

venient compromise. Larger sizes reduce the tag RAM array area and modestly reduce the miss rate, but increase the bandwidth consumed during refill. With an eight-word line size, the secondary-cache bandwidth supports three or four overlapped refills.

Each of the data cache's two banks comprises two logical arrays to support two-way set associativity. Unlike the usual arrangement, however, the cache ways alternate between these arrays to efficiently support different-width accesses, as Figure 11 shows.

The processor simultaneously reads the same double word from both cache ways, because it checks the cache tags in parallel and later selects data from the correct way. It discards the double word from the incorrect way. The external interface refills or writes quad words by accessing two double words in parallel. This is possible because it knows the correct cache way in advance.

This arrangement makes efficient use of the cache's sense amplifiers. Each amplifier includes a four-to-one multiplexer anyway, because there are four columns of memory cells for each amplifier. We implemented this feature by changing the select logic.

Secondary cache. We used external synchronous static RAM chips to implement the 512-Kbyte to 16-Mbyte, two-way set-associative secondary cache. Depending on system requirements, the user can configure the secondary-cache line size at either 16 or 32 words.

Set associativity reduces conflict misses and increases predictability. For an external cache, however, this usually requires special RAMs or many more interface pins. Instead, the R10000 implements a two-way pseudo-set-associative secondary cache using standard synchronous SRAMs and only one extra address pin.

Figure 12 shows how cache refills are pipelined. The single group of RAMs contains both cache ways. An on-chip bit array keeps track of which way was most recently used for each cache set. After a primary miss, two quad words are read from this way in the secondary cache. Its tag is read along with the first quad word. The tag of the alternate way is read with the second quad word by toggling of the extra address pin.

Three cases occur: If the first way hits, data becomes available immediately. If the alternate way hits, the processor reads the secondary cache again. If neither way hits, the processor must refill the secondary cache from memory.

Large external caches require error correction codes for data integrity. The R10000 stores both a 9-bit ECC code and a parity bit with each data quad word. The extra parity bit reduces latency because it can be checked quickly and stop the use of bad data. If the processor detects a correctable error, it retries the read through a two-cycle correction pipeline.

We can configure the interface to use this correction pipeline for all reads. Although this increases latency, it allows redundant lock-step processors to remain synchronized in the presence of correctable errors.

System interface

The R10000 communicates with the outside world using a 64-bit split-transaction system bus with multiplexed address and data. This bus can directly connect as many as four R10000 chips in a cluster and overlaps up to eight read requests.

The system interface dedicates substantial resources to support concurrency and out-of-order operation. Cache refills are nonblocking, with up to four outstanding read requests from either the secondary cache or main memory. These are controlled by the miss handling table.

The cached buffer contains addresses for four outstanding read requests. The memory data returned from these requests is stored in the four-entry incoming buffer, so that it can be accepted at any rate and in any order. The outgoing buffer holds up to five "victim" blocks to be written back to memory. The buffer requires the fifth entry when the bus invalidates a secondary-cache line.

An eight-entry cluster buffer tracks all outstanding operations on the system bus. It ensures cache coherency by interrogating and, if necessary, by invalidating cache lines.

Uncached loads and stores execute serially when they are the oldest instructions in the pipeline. The processor often uses uncached stores for writing to graphics or other peripheral devices. Such sequences typically consist of numerous sequentially or identically addressed accesses. The uncached buffer automatically gathers these into 32-word blocks to conserve bus bandwidth.

Clocks. An on-chip phase-locked loop (PLL) generates all timing synchronously with an external system interface clock. For system design and upgrade flexibility, independent clock divisors give users the choice of five secondary-cache and seven system interface clock frequencies. To allow more choices, we base these clocks on a PLL clock oscillating at twice the pipeline clock. When the pipeline operates at 200 MHz, the PLL operates at 400 MHz, and the user can configure the system interface to run at 200, 133, 100, 80, 66.7, 57, or 50 MHz. In addition, the user can separately configure the secondary-cache to frequencies between 200 and 66.7 MHz.

Output drivers. Four groups of buffers drive the chip's output pins. The user can configure each group separately to conform to either low-voltage CMOS or HSTL standards. The buffer design for each group has special characteristics.

Figure 13 (next page) illustrates how these buffers connect. The system interface buffer contains additional open-drain pull-down transistors, which provide the extra current needed to drive HSTL Class-2 multidrop buses.

We designed the secondary-cache data buffer to reduce overlap current spikes when switching, because nearly 200 of these signals can switch simultaneously.

The cache address buffer uses large totem pole transistors to rapidly drive multiple distributed loads. The cache clock buffer drives low-impedance differential signals with minimum output delay. A low-jitter delay element precisely aligns these clocks. This delay is statically configured to adjust for propagation delays in the printed circuit board clock net, so the clock's rising edge arrives at the cache simultaneously with the processor's internal clock.

Test features. For economical manufacture, a microprocessor chip must be easy to test with high fault coverage. The R10000 observes internal signals with ten 128-bit linear-feedback shift registers. These internal test points partition

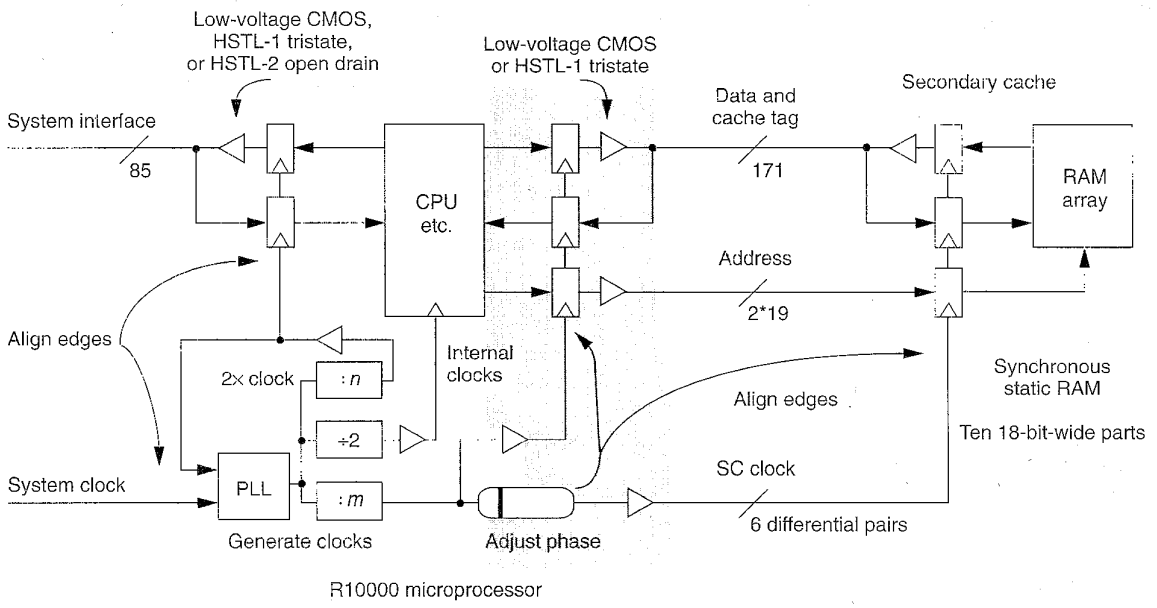


Figure 13. Clocks and output drivers.

the chip into three fully observed sections.

The registers are separate structures that do not affect the processor's logic or add a noticeable load to the observed signals. They use the processor's clock to ensure synchronous behavior and avoid any special clock requirements. They are useful for debugging or production testing.

Performance

We are currently shipping 200-MHz R10000 microprocessors in Silicon Graphics' Challenge servers, and several vendors will soon ship the R10000 in systems specifically designed to use its features. We project that such a system—with a 200-MHz R10000 microprocessor, 4-Mbyte secondary cache (200 MHz), 100-MHz system interface bus, and 180-ns memory latency—will have the following performance:

- SPEC95int (peak) 9
- SPEC95fp (peak) 19

We scaled these benchmark results from the performance of an R10000 running in an actual system in which the processor, cache, and memory speeds were proportionately slower. We compiled the benchmarks using early versions of the Mips Mongoose compiler.

AN AGGRESSIVE, SUPERSCALAR MICROPROCESSOR, the R10000 features fast clocks and a nonblocking, set-associative memory subsystem. Its design emphasizes concurrency and latency-hiding techniques to efficiently run large real-world applications.

Acknowledgments

The R10000 was designed by the Mips "T5" project team, whose dedication made this chip a reality. The figures in this article are derived from the author's design notes, and are used with permission from Mips Technologies, Inc.

Kenneth C. Yeager is a microprocessor designer at Mips Technologies Inc. (a subsidiary of Silicon Graphics Inc.), where he participated in the conception and design of the R10000 superscalar microprocessor. His research interests include the architecture, logic, and circuit implementation of high-performance processors. Yeager received BS degrees in physics and electrical engineering from the Massachusetts Institute of Technology.

Direct questions concerning this article to the author at Silicon Graphics Inc., M/S 10L-175, 2011 N. Shoreline Blvd., Mountain View, CA 94043; yeager@mti.sgi.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 153 Medium 154 High 155