THE ALPHA 21264 MICROPROCESSOR

THE ALPHA 21264 OWES ITS HIGH PERFORMANCE TO HIGH CLOCK SPEED, MANY FORMS OF OUT-OF-ORDER AND SPECULATIVE EXECUTION, AND A HIGH-BANDWIDTH MEMORY SYSTEM.

R. E. Kessler
Compaq Computer Corporation

Alpha microprocessors have been performance leaders since their introduction in 1992. The first generation 21064 and the later 21164[1,2] raised expectations for the newest generation—performance leadership was again a goal of the 21264 design team. Benchmark scores of 30+ SPECint95 and 58+ SPECfp95 offer convincing evidence thus far that the 21264 achieves this goal and will continue to set a high performance standard.

A unique combination of high clock speeds and advanced microarchitectural techniques, including many forms of out-of-order and speculative execution, provide exceptional core computational performance in the 21264. The processor also features a high-bandwidth memory system that can quickly deliver data values to the execution core, providing robust performance for a wide range of applications, including those without cache locality. The advanced performance levels are attained while maintaining an installed application base. All Alpha generations are upward-compatible. Database, real-time visual computing, data mining, medical imaging, scientific/technical, and many other applications can utilize the outstanding performance available with the 21264.

Architecture highlights

The 21264 is a superscalar microprocessor that can fetch and execute up to four instructions per cycle. It also features out-of-order execution.[3,4] With this, instructions execute as soon as possible and in parallel with other nondependent work, which results in faster execution because critical-path computations start and complete quickly.

The processor also employs speculative execution to maximize performance. It speculatively fetches and executes instructions even though it may not know immediately whether the instructions will be on the final execution path. This is particularly useful, for instance, when the 21264 predicts branch directions and speculatively executes down the predicted path.

Sophisticated branch prediction, coupled with speculative and dynamic execution, extracts instruction parallelism from applications. With more functional units and these dynamic execution techniques, the processor is 50% to 200% faster than its 21164 predecessor for many applications, even though both generations can fetch at most four instructions per cycle.[5]

The 21264's memory system also enables high performance levels. On-chip and off-chip caches provide for very low latency data access. Additionally, the 21264 can service many parallel memory references to all caches in the hierarchy, as well as to the off-chip memory system. This permits very high bandwidth data access.[6] For example, the processor can sustain more than 1.3 GBytes/sec on the Stream benchmark.[7]

The microprocessor's cycle time is 500 to 600 MHz, implemented by 15 million transistors in a 2.2-V, 0.35-micron CMOS process with six metal layers. The 3.1 cm^2 processor

0272-1732/99/$10.00 © 1999 IEEE

comes in a 587-pin PGA package. It can execute up to 2.4 billion instructions per second.

Figure 1 shows a photo of the 21264, highlighting major sections. Figure 2 is a high-level overview of the 21264 pipeline, which has seven stages, similar to the earlier in-order 21164. One notable addition is the map stage that renames registers to expose instruction parallelism—this addition is fundamental to the 21264's out-of-order techniques.

Instruction pipeline—Fetch

The instruction pipeline begins with the fetch stage, which delivers four instructions to the out-of-order execution engine each cycle. The processor speculatively fetches through line, branch, or jump predictions. Since the predictions are usually accurate, this instruction fetch implementation typically supplies a continuous stream of good-path instructions to keep the functional units busy with useful work.

Two architectural techniques increase fetch efficiency: line and way prediction, and branch prediction. A 64-Kbyte, two-way set-associative instruction cache offers much-improved level-one hit rates compared to the 8-Kbyte, direct-mapped instruction cache in the Alpha 21164.

Line and way prediction

The processor implements a line and way prediction technique that combines the advantages of set-associative behavior and fetch bubble elimination, together with the fast access time of a direct-mapped cache. Figure 3 (next page) shows the technique's main features. Each four-instruction fetch block includes a line and way prediction. This prediction indicates where to fetch the next block of four instructions, including which way—that is, which of the two choices allowed by two-way associative cache.

The processor reads out the next instructions using the prediction (via the wraparound path in Figure 3) while, in parallel, it completes the validity check for the previous instructions. Note that the address paths needing extra logic levels—instruction decode, branch prediction, and cache tag comparison—are outside the critical fetch loop.

The processor loads the line and way predictors on an instruction cache fill, and

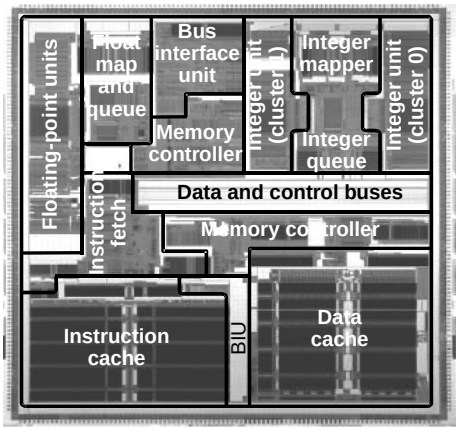

Figure 1. Alpha 21264 microprocessor die photo. BIU stands for bus interface unit.

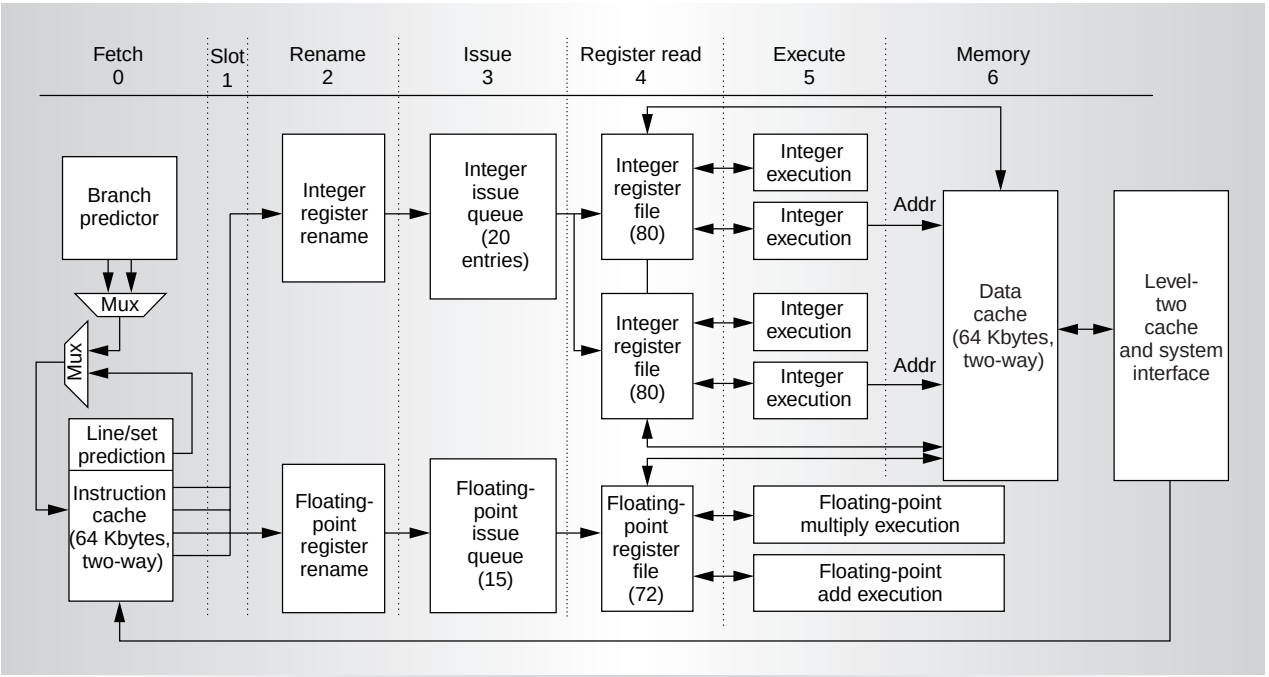

Figure 2. Stages of the Alpha 21264 instruction pipeline.

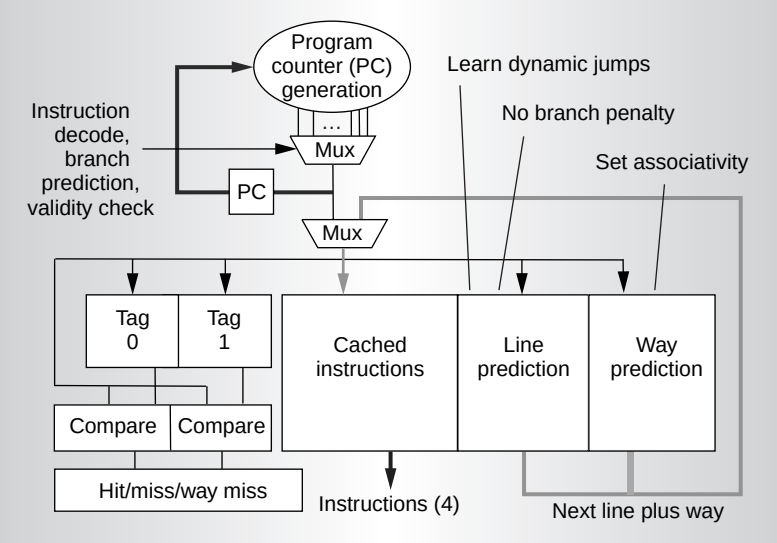

Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.

dynamically retrains them when they are in error. Most mispredictions cost a single cycle. The line and way predictors are correct 85% to 100% of the time for most applications, so training is infrequent. As an additional precaution, a 2-bit hysteresis counter associated with each fetch block eliminates overtraining—training occurs only when the current prediction has been in error multiple times. Line and way prediction is an important speed enhancement since the mispredict cost is low and line/way mispredictions are rare.

Beyond the speed benefits of direct cache access, line and way prediction has other benefits. For example, frequently encountered predictable branches, such as loop terminators, avoid the mis-fetch penalty often associated with a taken branch. The processor also trains the line predictor with the address of jumps and subroutine calls that use direct register addressing. Code using dynamically linked library routines will thus benefit after the line predictor is trained with the target. This is important since the pipeline delays required to calculate the indirect (subroutine) jump address are eight cycles or more.

An instruction cache miss forces the instruction fetch engine to check the level-two (L2) cache or system memory for the necessary instructions. The fetch engine prefetches up to four 64-byte (or 16-instruction) cache lines to tolerate the additional latency. The result is very high bandwidth instruction fetch, even when the instructions are not found in the instruction cache. For instance, the processor can saturate the available L2 cache bandwidth with instruction prefetches.

Branch prediction

Branch prediction is more important to the 21264's efficiency than to previous microprocessors for several reasons. First, the seven-cycle mispredict cost is slightly higher than previous generations. Second, the instruction execution engine is faster than in previous generations. Finally, successful branch prediction can utilize the processor's speculative execution capabilities. Good branch prediction avoids the costs of mispredicts and capitalizes on the most opportunities to find parallelism. The 21164 could accept 20 in-flight instructions at most, but the 21264 can accept 80, offering many more parallelism opportunities.

The 21264 implements a sophisticated tournament branch prediction scheme. The scheme dynamically chooses between two types of branch predictors—one using local history, and one using global history—to predict the direction of a given branch.[8] The result is a tournament branch predictor with better prediction accuracy than larger tables of either individual method, with a 90% to 100% success rate on most simulated applications/benchmarks. Together, local and global correlation techniques minimize branch mispredicts. The processor adapts to dynamically choose the best method for each branch.

Figure 4, in detailing the structure of the tournament branch predictor, shows the local-history prediction path—through a two-level structure—on the left. The first level holds 10 bits of branch pattern history for up to 1,024 branches. This 10-bit pattern picks from one of 1,024 prediction counters. The global predictor is a 4,096-entry table of 2-bit saturating counters indexed by the path, or global, history of the last 12 branches. The choice prediction, or chooser, is also a 4,096-entry table of 2-bit prediction counters indexed by the path history. The "Local and global branch predictors" box describes these techniques in more detail.

The processor inserts the true branch direction in the local-history table once branches

Local and global branch predictors

The Alpha 21264 branch predictor uses local history and global history to predict future branch directions since branches exhibit both local correlation and global correlation. The property of local correlation implies branch direction prediction on the basis of the branch's past behavior. Local-history predictors are typically tables, indexed by the program counter (branch instruction address), that contain history information about many different branches. Different table entries correspond to different branch instructions. Different local-history formats can exploit different forms of local correlation. In some simple (single-level) predictors, the local-history table entries are saturating prediction counters (incremented on taken branches and decremented on not-taken branches), and the prediction is the counter's uppermost bit. In these predictors, a branch will be predicted taken if the branch is often taken.

The 21264's two-level local prediction exploits pattern-based prediction; see Figure 4 in the main text. Each table entry in the first-level local-history table is a 10-bit pattern indicating the direction of the selected branch's last 10 executions. The local-branch prediction is a prediction counter bit from a second table (the local-prediction table) indexed by the local-history pattern. In this more sophisticated predictor, branches will be predicted taken when branches with the local-history pattern are often taken.

Figure A1 shows two simple example branches that can be predicted with the 21264's local predictor. The second branch on the left (b == 0) is always not taken. It will have a consistent local-history pattern of all zeroes in the local-history table. After several invocations of this branch, the prediction counter at index zero in the local-prediction table will train, and the branch will be predicted correctly.

The first branch on the left (a % 2 == 0) alternates between taken and not taken on every invocation. It could not be predicted correctly with simple single-level local-history predictors. The alternation leads to two local-history patterns in the first-level table: 0101010101 and 1010101010. After several invocations of this branch, the two prediction counters at these indices in the local prediction table will train—one taken and the other not taken. Any repeating pattern of 10 invocations of the same branch can be predicted this way.

With global correlation, a branch can be predicted based on the past behavior of all previous branches, rather than just the past behavior of the single branch. The 21264 exploits global correlation by tracking the path, or global, history of all branches. The path history is a 12-bit pattern indicating the taken/not taken direction for the last 12 executed branches (in fetch order). The 21264's global-history predictor is a table of saturating counters indexed by the path history.

Figure A2 shows an example that could use global correlation. If both branches (a == 0) and (b == 0) are taken, we can easily predict that a and b are equal and, therefore, the (a == b) branch will be taken. This means that if the path history is xxxxxxxxxx11 (ones in the last two bits), the branch should be predicted taken. With enough executions, the 21264's global prediction counters at the xxxxxxxxxx11 indices will train to predict taken and the branch will be predicted correctly. Though this example requires only a path history depth of 2, more complicated path history patterns can be found with the path history depth of 12.

Since different branches can be predicted better with either local or global correlation techniques, the 21264 branch predictor implements both local and global predictors. The chooser (choice prediction) selects either local or global prediction as the final prediction.[1] The chooser is a table of prediction counters, indexed by path history, that dynamically selects either local or global predictions for each branch invocation. The processor trains choice prediction counters to prefer the correct prediction whenever the local and global predictions differ. The chooser may select differently for each invocation of the same branch instruction.

Figure A2 exemplifies the chooser's usefulness. The chooser may select the global prediction for the (a == b) branch whenever the branches (a == 0) and (b == 0) are taken, and it may select the local prediction for the (a == b) branch in other cases.

Reference

1. S. McFarling, *Combining Branch Predictors*, Tech. Note TN-36, Compaq Computer Corp. Western Research Laboratory, Palo Alto, Calif., June 1993; http://www.research.digital.com/wrl/techreports/abstracts/TN-36.html.

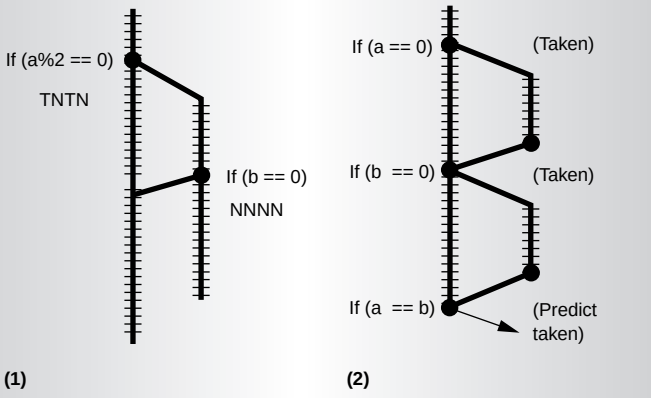

Figure A. Branch prediction in the Alpha 21264 occurs by means of local-history prediction (1) and global-history prediction (2). "TNTN" indicates that during the last four executions of the branch, the branch was taken twice and not taken twice. Similarly, NNNN indicates that the branch has not been taken in the last four executions—a pattern of 0000.

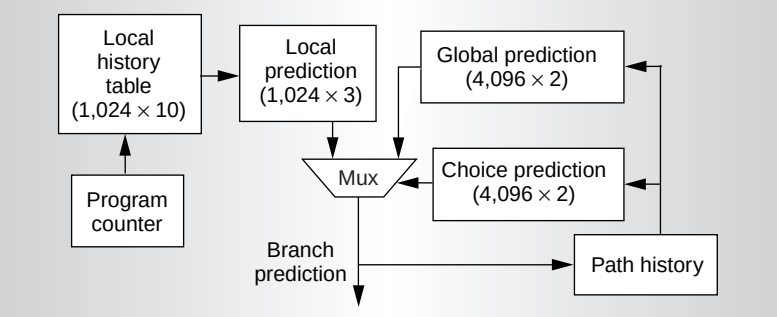

Figure 4. Block diagram of the 21264 tournament branch predictor. The local history prediction path is on the left; the global history prediction path and the chooser (choice prediction) are on the right.

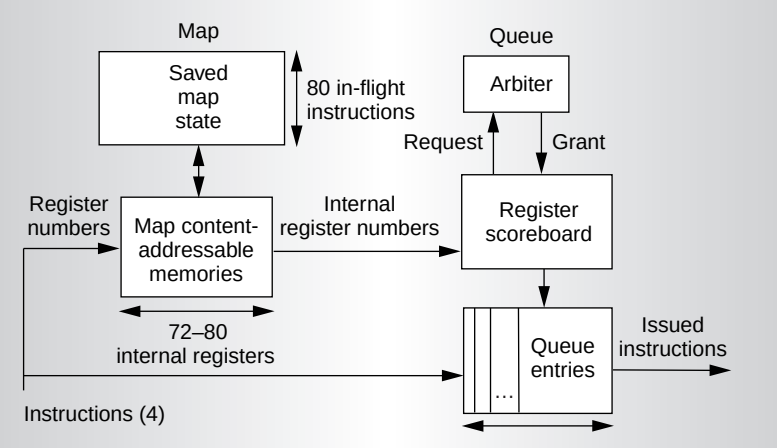

Figure 5. Block diagram of the 21264's map (register rename) and queue stages. The map stage renames programmer-visible register numbers to internal register numbers. The queue stage stores instructions until they are ready to issue. These structures are duplicated for integer and floating-point execution.

issue and retire. It also trains the correct predictions by updating the referenced local, global, and choice counters at that time. The processor maintains path history with a silo of 12 branch predictions. This silo is speculatively updated before a branch retires and is backed up on a mispredict.

Out-of-order execution

The 21264 offers out-of-order efficiencies with higher clock speeds than competing designs, yet this speed does not restrict the microprocessor's dynamic execution capabilities. The out-of-order execution logic receives four fetched instructions every cycle, renames/remaps the registers to avoid unnecessary register dependencies, and queues the instructions until operands or functional units become available. It dynamically issues up to six instructions every cycle—four integer instructions and two floating-point instructions. It also provides an in-order execution model to the programmer via in-order instruction retire.

Register renaming

Register renaming exposes application instruction parallelism since it eliminates unnecessary dependencies and allows speculative execution. Register renaming assigns a unique storage location with each write-reference to a register. The 21264 speculatively allocates a register to each instruction with a register result. The register only becomes part of the user-visible (architectural) register state when the instruction retires/commits. This lets the instruction speculatively issue and deposit its result into the register file before the instruction retires. Register renaming also eliminates write-after-write and write-after-read register dependencies, but preserves all the read-after-write register dependencies that are necessary for correct computation.

The left side of Figure 5 depicts the map, or register rename, stage in more detail. The processor maintains storage with each internal register indicating the user-visible register that is currently associated with the given internal register (if any). Thus, register renaming is a content-addressable memory (CAM) operation for register sources together with a register allocation for the destination register. All pipeline stages subsequent to the register map stage operate on internal registers rather than user-visible registers.

Beyond the 31 integer and 31 floating-point user-visible (non-speculative) registers, an additional 41 integer and 41 floating-point registers are available to hold speculative results prior to instruction retirement. The register mapper stores the register map state for each in-flight instruction so that the machine architectural state can be restored in case a misspeculation occurs.

The Alpha conditional-move instructions must be handled specially by the map stage. These operations conditionally move one of two source registers into a destination register. This makes conditional move the only instruction in the Alpha architecture that requires three register sources—the two

sources plus the old value of the destination register (in case the move is not performed).

The 21264 splits each conditional move instruction into two and maps them separately. These two new instructions only have two register sources. The first instruction places the move value into an internal register together with a 65th bit indicating the move's ultimate success or failure. The second instruction reads the first result (including the 65th bit) together with the old destination register value and produces the final destination register result.

Out-of-order issue queues

The issue queue logic maintains two lists of pending instructions in separate integer and floating-point queues. Each cycle, the queue logic selects from these instructions, as their operands become available, using register scoreboards based on the internal register numbers. These scoreboards maintain the status of the internal registers by tracking the progress of single-cycle, multiple-cycle, and variable-cycle (memory load) instructions. When functional-unit or load-data results become available, the scoreboard unit notifies all instructions in the queue that require the register value. These dependent instructions can issue as soon as the bypassed result becomes available from the functional unit or load. The 20-entry integer queue can issue four instructions, and the 15-entry floating-point queue can issue two instructions per cycle. The issue queue is depicted on the right in Figure 5.

The integer queue statically assigns, or slots, each instruction to one of two arbiters before the instructions enter the queue. Each arbiter handles two of the four integer pipes. Each arbiter dynamically issues the oldest two queued instructions each cycle. The integer queue slots instructions based on instruction fetch position to equalize the utilization of the integer execution engine's two halves. An instruction cannot switch arbiters after the static assignment upon entry into the queue. The interactions of these arbitration decisions with the structure of the execution pipes are described later.

The integer and floating-point queues issue instructions speculatively. Each queue/arbiter selects the oldest operand-ready and functional-unit-ready instructions for execution each cycle. Since older instructions receive priority over newer instructions, speculative issues do not slow down older, less speculative issues. The queues are collapsable—an entry becomes immediately available once the instruction issues or is squashed due to misspeculation. New instructions can enter the issue queue when there are four or more available queue slots, and new instructions can enter the floating-point queue when there are enough available queue slots.[9]

Instruction retire and exception handling

Although instructions issue out of order, instructions are fetched and retired in order. The in-order retire mechanism maintains the illusion of in-order execution to the programmer even though the instructions actually execute out of order. The retire mechanism assigns each mapped instruction a slot in a circular in-flight window (in fetch order). After an instruction starts executing, it can retire whenever all previous instructions have retired and it is guaranteed to generate no exceptions. The retiring of an instruction makes the instruction nonspeculative—guaranteeing that the instruction's effects will be visible to the programmer. The 21264 implements a precise exception model using in-order retiring. The programmer does not see the effects of a younger instruction if an older instruction has an exception.

The retire mechanism also tracks the internal register usage for all in-flight instructions. Each entry in the mechanism contains storage indicating the internal register that held the old contents of the destination register for the corresponding instruction. This (stale) register can be freed for other use after the instruction retires. After retiring, the old destination register value cannot possibly be needed—all older instructions must have issued and read their source registers; all newer instructions cannot use the old destination register value.

An exception causes all younger instructions in the in-flight window to be squashed. These instructions are removed from all queues in the system. The register map is backed up to the state before the last squashed instruction using the saved map state. The map state for each in-flight instruction is maintained, so it is easily restored. The registers allocated by the squashed instructions become immediately available. The retire mechanism has a large, 80-instruction in-

Table 1. Sample 21264 retire pipe stages.

Instruction class	Retire latency (cycles)
Integer	4
Memory	7
Floating-point	8
Branch/jump to subroutine	7

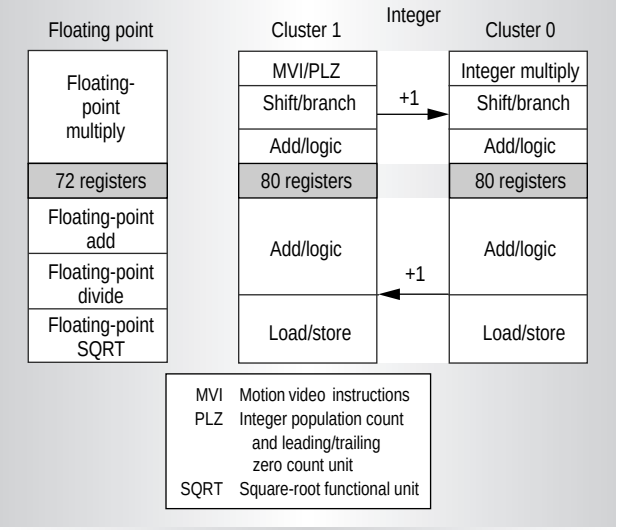

Figure 6. The four integer execution pipes (upper and lower for each of a left and right cluster) and the two floating-point pipes in the 21264, together with the functional units in each.

flight window. This means that up to 80 instructions can be in partial states of completion at any time, allowing for significant execution concurrency and latency hiding. (This is particularly true since the memory system can track an additional 32 in-flight loads and 32 in-flight stores.)

Table 1 shows the minimum latency, in number of cycles, from issue until retire eligibility for different instruction classes. The retire mechanism can retire at most 11 instructions in a single cycle, and it can sustain a rate of 8 per cycle (over short periods).

Execution engine

Figure 6 depicts the six execution pipelines. Each pipeline is physically placed above or below its corresponding register file. The 21264 splits the integer register file into two clusters that contain duplicates of the 80-entry register file. Two pipes access a single register file to form a cluster, and the two clusters combine to support four-way integer instruction execution. This clustering makes the design simpler and faster, although it costs an extra cycle of latency to broadcast results from an integer cluster to the other cluster. The upper pipelines from the two integer clusters in Figure 6 are managed by the same issue queue arbiter, as are the two lower pipelines. The integer queue statically slots instructions to either the upper or lower pipeline arbiters. It then dynamically selects which cluster to execute an instruction on, left or right.

The performance costs of the register clustering and issue queue arbitration simplifications are small—a few percent or less compared to an idealized unclustered implementation in most applications. There are multiple reasons for the minimal performance effect. First, for many operations (such as loads and stores) the static-issue queue assignment is not a restriction since they can only execute in either the upper or lower pipelines. Second, critical-path computations tend to execute on the same cluster. The issue queue prefers older instructions, so more-critical instructions incur fewer cross-cluster delays—an instruction can usually issue first on the same cluster that produces the result. This integer pipeline architecture as a result provides much of the implementation simplicity, lower risk, and higher speed of a two-issue machine with the performance benefits of four-way integer issue. Figure 6 also shows the floating-point execution pipes' configuration. A single cluster has the two floating-point execution pipes, with a single 72-entry register file.

The 21264 includes new functional units not present in prior Alpha microprocessors. The Alpha motion-video instructions (MVI, used to speed many forms of image processing), a fully pipelined integer multiply unit, an integer population count and leading/trailing zero count unit (PLZ), a floating-point square-root functional unit, and instructions to move register values directly between floating-point and integer registers are included. The processor also provides more complete hardware support for the IEEE floating-point standard, including precise exceptions, NaN and infinity processing, and support for flushing denormal results to zero. Table 2 shows sample instruction latencies (issue of producer to issue of consumer). These latencies are achieved through result bypassing.

Internal memory system

The internal memory system supports many in-flight memory references and out-of-order operations. It can service up to two memory references from the integer execution pipes every cycle. These two memory references are out-of-order issues. The memory system simultaneously tracks up to 32 in-flight loads, 32 in-flight stores, and 8 in-flight (instruction or data) cache misses. It also has a 64-Kbyte, two-way set-associative data cache. This cache has much lower miss rates than the 8-Kbyte, direct-mapped cache in the earlier 21164. The end result is a high-bandwidth, low-latency memory system.

Table 2. Sample 21264 instruction latencies (s-p means single-precision; d-p means double-precision).

Instruction class	Latency (cycles)
Simple integer operations	1
Motion-video instructions/integer population count and leading/trailing zero count unit (MVI/PLZ)	3
Integer multiply	7
Integer load	3
Floating-point load	4
Floating-point add	4
Floating-point multiply	4
Floating-point divide	12 s-p,15 d-p
Floating-point square-root	15 s-p, 30 d-p

Data path

The 21264 supports any combination of two loads or stores per cycle without conflict. The data cache is double-pumped to implement the necessary two ports. That means that the data cache is referenced twice each cycle—once per each of the two clock phases. In effect, the data cache operates at twice the frequency of the processor clock—an important feature of the 21264's memory system.

Figure 7 depicts the memory system's internal data paths. The two 64-bit data buses are the heart of the internal memory system. Each load receives data via these buses from the data cache, the speculative store data buffers, or an external (system or L2) fill. Stores first transfer their data across the data buses into the speculative store buffer. Store data remains in the speculative store buffer until the stores retire. Once they retire, the data is written (dumped) into the data cache on idle cache cycles. Each dump can write 128 bits into the cache since two stores can merge into one dump. Dumps use the double-pumped data cache to implement a read-modify-write sequence. Read-modify-write is required on stores to update the stored SECDED ECC that allows correction of single-bit errors.

Stores can forward their data to subsequent loads while they reside in the speculative store data buffer. Load instructions compare their age and address against these pending stores. On a match, the appropriate store data is put on the data bus rather than the data from the data cache. In effect, the speculative store data buffer performs a memory-renaming function. From the perspective of younger loads, it appears the stores write into the data cache immediately. However, squashed stores are removed from the speculative store data buffer before they affect the final cache state.

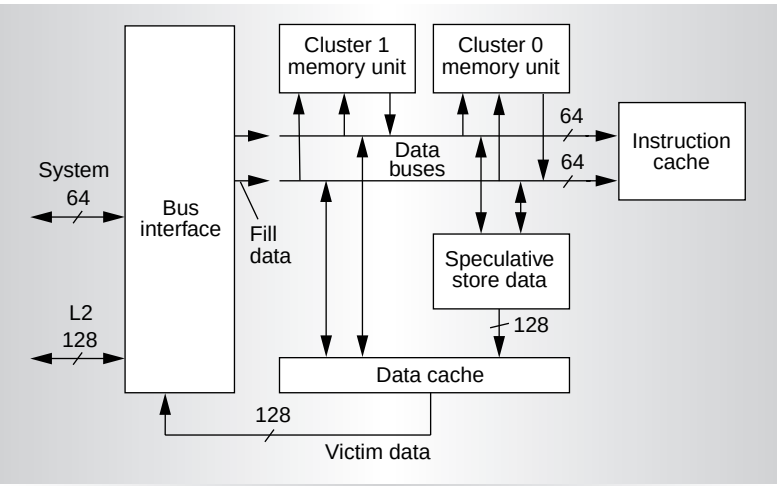

Figure 7. The 21264's internal memory system data paths.

Figure 7 shows how data is brought into and out of the internal memory system. Fill data arrives on the data buses. Pending loads sample the data to write into the register file while, in parallel, the caches (instruction or data) also fill using the same bus data. The data cache is write-back, so fills also use its double-pumped capability: The previous cache contents are read out in the same cycle that fill data is written in. The bus interface unit captures this victim data and later writes it back.

Address and control structure

The internal memory system maintains a 32-entry load queue (LDQ) and a 32-entry

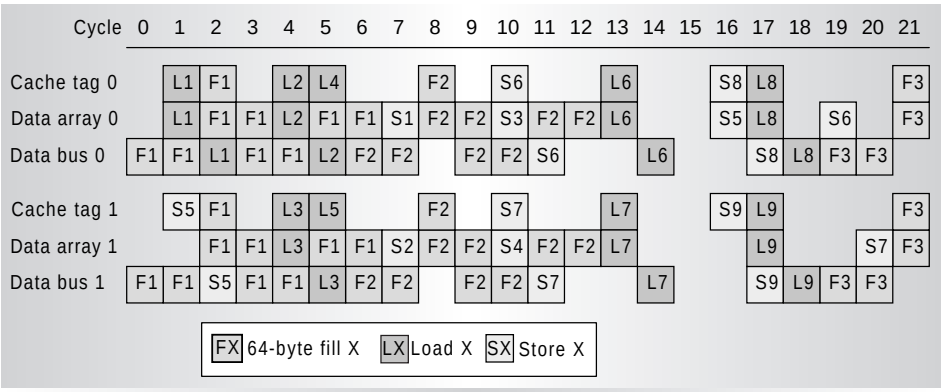

Figure 8. A 21264 memory system pipeline diagram showing major data path elements—data cache tags, data cache array, and data buses for an example usage containing loads, stores, and fills from the L2 cache.

store queue (STQ) that manage the references while they are in-flight. The LDQ (STQ) positions loads (stores) in the queue in fetch order, although they enter the queue when they issue, out of order. Loads exit the LDQ in fetch order after the loads retire and the load data has been returned. Stores exit the STQ in fetch order after they retire and dump into the data cache.

New issues check their address and age against older references. Dual-ported address CAMs resolve the read-after-read, read-after-write, write-after-read, and write-after-write hazards inherent in a fully out-of-order memory system. For instance, to detect memory read-after-write hazards, the LDQ must compare addresses when a store issues: Whenever an older store issues after a younger load to the same memory address, the LDQ must squash the load—the load got the wrong data. The 21264's response to this hazard is described in more detail in a later section. The STQ CAM logic controls the speculative store data buffer. It enables the bypass of speculative store data to loads when a younger load issues after an older store.

Figure 8 shows an example scheduling of the memory system, including both data buses, the data cache tags, and both ports to the data cache array itself. Nine load issues, L1–L9, pieces of nine stores, S1–S9, and pieces of three fills (each requiring four data bus cycles to get the required 64 bytes), F1–F3, are included. Loads reference both the data cache tags and array in their pipeline stage 6, and then the load result uses the data bus one cycle later in their stage 7. For example in Figure 8, loads L6 and L7 are in their stage 6 (from Figure 2) in cycle 13, and in their stage 7 in cycle 14.

Stores are similar to loads except they do not use the cache array until they retire. For example, S5 uses the tags in cycle 1 in Figure 8 but does not write into the data cache until cycle 16. The fill patterns from the figure correspond to fills from the fastest possible L2 cache. The data that crosses the data buses in cycle 0 gets skewed to cycle 2 and written into the cache. This skew is for performance reasons only—the fill data could have been written in cycle 1 instead, but pipeline conflicts would allow fewer load and store issues in that case. Fills only need to update the cache tags once per block. This leaves the data cache tags more idled than the array or the data bus.

Cycles 7, 10, and 16 in the example each show two stores that merge into a single cache dump. Note that the dump in cycle 7 (S1 and S2 writing into the data cache array) could not have occurred in cycles 1–6 because the cache array is busy with fills or load issues in each case. Cache dumps can happen when only stores issue, such as in cycle 10. Loads L4 and L5 at cycle 5 are a special case. Note how these two loads only use the cache tags. These loads decouple the cache tag lookup from the data array and data bus use, taking advantage of the idle data cache tags during fills. This decoupling is particularly useful for loads that miss in the data cache, since in that case the cache array lookup and data bus transfer of the load issue slot are avoided. Decoupled loads maximize the available bandwidth for cache misses since they eliminate unnecessary cache accesses.

The internal memory system also contains an eight-entry miss address file (MAF). Each entry tracks an outstanding miss (fill) to a 64-byte cache block. Multiple load and store misses to the same cache block can merge and be satisfied by a single MAF entry. Each MAF entry can be either an L2 cache or system fill. Both loads and stores can create a MAF entry immediately after they check the data cache tags (on initial issue). This MAF entry is then for-

Table 3. The 21264 cache prefetch and management instructions.

Instruction	Description
Normal prefetch	The 21264 fetches the 64-byte block into the L1 data and L2 cache.
Prefetch with modify intent	The same as the normal prefetch except that the block is loaded into the cache in a writeable state.
Prefetch and evict next	The same as the normal prefetch except that the block will be evicted from the L1 data cache on the next access to the same data cache set.
Write-hint 64	The 21264 obtains write access to the 64-byte block without reading the old contents of the block
Evict	The cache block is evicted from the caches.

warded for further processing by the bus interface unit—before the load or store is retired.

The Alpha memory model is weakly ordered. Ordering among references to different memory addresses is required only when the programmer inserts memory barrier instructions. In the 21264, these instructions drain the internal memory system and bus interface unit.

Cache prefetching and management

The memory system implements cache prefetch instructions that let the programmer take full advantage of the memory system's parallelism and high-bandwidth capabilities. These prefetches are particularly useful in applications with loops that reference large arrays. In these and other cases where software can predict memory references, it can prefetch the associated 64-byte cache blocks to overlap the cache-miss time with other operations. The prefetch can be scheduled far in advance because the block is held in the cache until it is used.

Table 3 describes the cache prefetch and management instructions. Normal, modify-intent, and evict-next prefetches perform similar operations but are used in different specific circumstances. For each, the processor fills the block into the data cache if it was not already present in the cache. The write-hint 64 instruction resembles a prefetch with modify intent except that the block's previous value is not loaded. This is useful, for example, to zero out a contiguous memory region.

Bus interface unit

The 21264 bus interface unit (BIU) interfaces the internal memory system and the external (off-chip) L2 cache and the rest of the system. It receives MAF references from the internal memory system and responds with fill data from either the L2 cache on a hit, or the system on a miss. It forwards victim data from the data cache to the L2 and from the L2 to the system using an eight-entry victim file. It manages the data cache contents. Finally, it receives cache probes from the system, performs the necessary cache coherence actions, and responds to the system. The 21264 implements a write-invalidate cache coherence protocol to support shared-memory multiprocessing.

Figure 9 shows the 21264's external interface. The interface to the L2 (on the right) is separate from the interface to the system (on the left). All interconnects on and off the 21264 are high-speed point-to-point channels. They use clock-forwarding technology to maximize the available bandwidth and minimize pin counts.

The L2 cache provides a fast backup store for the primary caches. This cache is direct-mapped, shared by both instructions and data, and can range from 1 to 16 Mbytes. The BIU can support a wide range of SRAM part variants for different size, speed, and latency L2, including late-write synchronous, PC-style, and dual-data for very high speed operation. The peak L2 transfer rate is 16 bytes every 1.5 CPU cycles. This is a bandwidth of 6.4 Gbytes/sec with a 400-MHz transfer rate. The minimum L2 cache latency is 12 cycles using an SRAM part with a latency of six cycles—nine more than the data cache latency of three.

Figure 9 shows that the 21264 has split

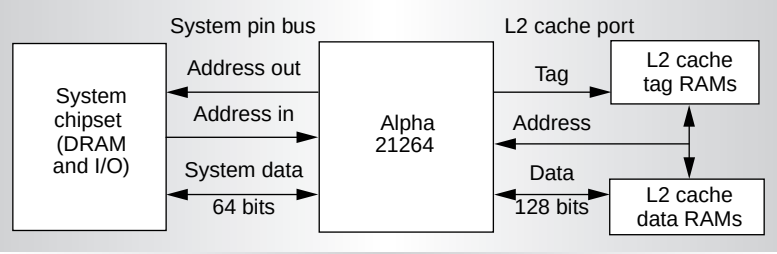

Figure 9. The 21264 external interface.

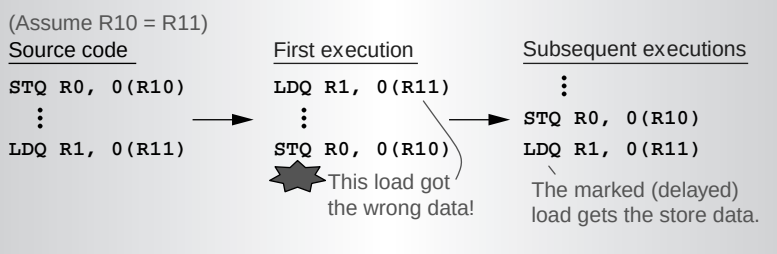

Figure 10. An example of the 21264 memory load-after-store hazard adaptation.

address-out and address-in buses in the system pin bus. This provides bandwidth for new address requests (out from the processor) and system probes (into the processor), and allows for simple, small-scale multiprocessor system designs. The 21264 system interface's low pin counts and high bandwidth let a high-performance system (of four or more processors) broadcast probes without using a large number of pins. The BIU stores pending system probes in an eight-entry probe queue before responding to the probes, in order. It responds to probes very quickly to support a system with minimum latency, and minimizes the address bus bandwidth required in common probe response cases.

The 21264 provides a rich set of possible coherence actions; it can scale to larger-scale system implementations, including directory-based systems.[4] It supports all five of the standard MOESI (modified-owned-exclusive-shared-invalid) cache states.

The BIU supports a wide range of system data bus speeds. The peak bandwidth of the system data interface is 8 bytes of data per 1.5 CPU cycles—or 3.2 Gbytes/sec at a 400-MHz transfer rate. The load latency (issue of load to issue of consumer) can be as low as 160 ns with a 60-ns DRAM access time. The total of eight in-flight MAFs and eight in-flight victims provide many parallel memory operations to schedule for high SRAM and DRAM efficiency. This translates into high memory system performance, even with cache misses. For example, the 21264 has sustained in excess of 1.3 Gbytes/sec (user-visible) memory bandwidth on the Stream benchmark.[7]

Dynamic execution examples

The 21264 architecture is very dynamic. In this article I have discussed a number of its dynamic techniques, including the line predictor, branch predictor, and issue queue scheduling. Two more examples in this section further illustrate the 21264's dynamic adaptability.

Store/load memory ordering

The 21264 memory system supports the full capabilities of the out-of-order execution core, yet maintains an in-order architectural memory model. This is a challenge when multiple loads and stores reference the same address. The register rename logic cannot automatically handle these read-after-write memory dependencies as it does register dependencies because it does not have the memory address until the instruction issues. Instead, the memory system dynamically detects the problem case *after* the instructions issue (and the addresses are available).

This example shows how the 21264 dynamically adapts to avoid the costs of load misspeculation. It remembers the first misspeculation and avoids the problem in subsequent executions by delaying the load.

Figure 10 shows how the 21264 resolves a memory read-after-write hazard. The source instructions are on the far left—a store followed by a load to the same address. On the first execution of these instructions, the 21264 attempts to issue the load as early as possible—before the older store—to minimize load latency. The load receives the wrong data since it issues before the store in this case, so the 21264 hazard detection logic squashes the load (and all subsequent instructions). After this type of load misspeculation, the 21264 trains to avoid it on subsequent executions by setting a bit in a load wait table.

Figure 10 also shows what happens on subsequent executions of the same code. At fetch time the store wait table bit corresponding to the load is set. The issue queue then forces the issue point of the marked load to be delayed until all prior stores have issued, thereby avoiding this store/load order violation and also allowing the speculative store buffer to bypass the correct data to the load. This store wait table is periodically cleared to avoid unnecessary waits.

This example store/load order case shows how the memory system produces a result that is the same as an in-order memory system while capturing the performance advantages

of out-of-order execution. Unmarked loads issue as early as possible, and before as many stores as possible, while only the necessary marked loads are delayed.

Load hit/miss prediction

There are minispeculations within the 21264's speculative execution engine. To achieve the minimum three-cycle integer load hit latency, the processor must speculatively issue the consumers of the integer load data before knowing if the load hit or missed in the on-chip data cache. This early issue allows the consumers to receive bypassed data from a load at the earliest possible time. Note in Figure 2 that the data cache stage is three cycles after the queue, or issue, stage, so the load's cache lookup must happen in parallel with the consumers issue. Furthermore, it really takes another cycle after the cache lookup to get the hit/miss indication to the issue queue. This means that consumers of the results produced by the consumers of the load data (the beneficiaries) can also speculatively issue—even though the load may have actually missed.

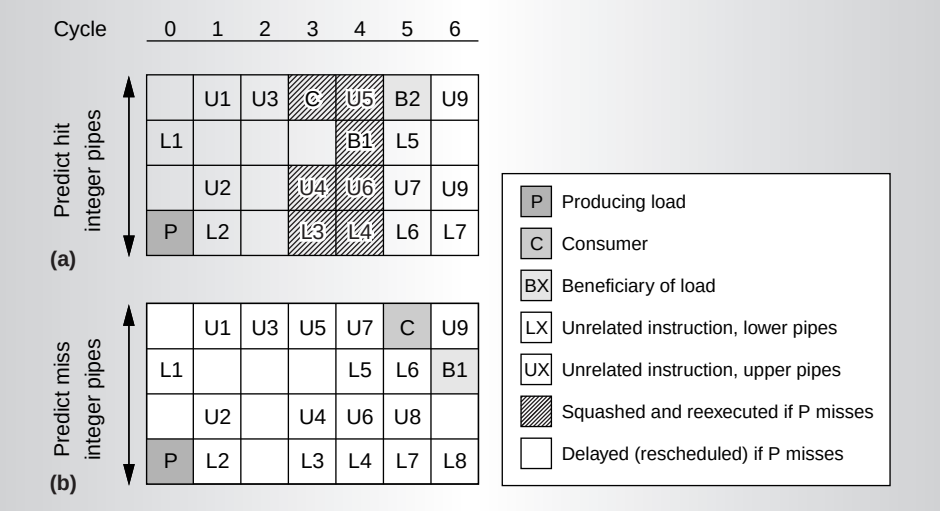

Figure 11. Integer load hit/miss prediction example. This figure depicts the execution of a workload when the selected load (P) is predicted to hit (a) and predicted to miss (b) on the four integer pipes. The cross-hatched and screened sections show the instructions that are either squashed and reexecuted from the issue queue, or delayed due to operand availability or the reexecution of other instructions.

The processor could rely on the general mechanisms available in the speculative execution engine to abort the integer load data's speculatively executed consumers; however, that requires restarting the entire instruction pipeline. Given that load misses can be frequent in some applications, this technique would be too expensive. Instead, the processor handles this with a minirestart. When consumers speculatively issue three cycles after a load that misses, two integer issue cycles (on all four integer pipes) are squashed. All integer instructions that issued during those two cycles are pulled back into the issue queue to be reissued later. This forces the processor to reissue both the consumers and the beneficiaries. If the load hits, the instruction schedule shown on the top of Figure 11 will be executed. If the load misses, however, the original issues of the unrelated instructions L3–L4 and U4–U6 must be reexecuted in cycles 5 and 6. The schedule thus is delayed two cycles from that depicted.

While this two-cycle window is less costly than fully restarting the processor pipeline, it still can be expensive for applications with many integer load misses. Consequently, the 21264 predicts when loads will miss and does not speculatively issue the consumers of the load data in that case. The bottom half of Figure 11 shows the example instruction schedule for this prediction. The effective load latency is five cycles rather than the minimum three for an integer load hit that is (incorrectly) predicted to miss. But more unrelated instructions are allowed to issue in the slots not taken by the consumer and the beneficiaries.

The load hit/miss predictor is the most-significant bit of a 4-bit counter that tracks the hit/miss behavior of recent loads. The saturating counter decrements by two on cycles when there is a load miss, otherwise it increments by one when there is a hit. This hit/miss predictor minimizes latencies in applications that often hit, and avoids the costs of over-speculation for applications that often miss.

The 21264 treats floating-point loads differently than integer loads for load hit/miss prediction. The floating-point load latency is four cycles, with no single-cycle operations, so

there is enough time to resolve the exact instruction that used the load result.

Compaq has been shipping the 21264 to customers since the last quarter of 1998. Future versions of the 21264, taking advantage of technology advances for lower cost and higher speed, will extend the Alpha's performance leadership well into the new millennium. The next-generation 21364 and 21464 Alphas are currently being designed. They will carry the Alpha line even further into the future. MICRO

Acknowledgments

The 21264 is the fruition of many individuals, including M. Albers, R. Allmon, M. Arneborn, D. Asher, R. Badeau, D. Bailey, S. Bakke, A. Barber, S. Bell, B. Benschneider, M. Bhaiwala, D. Bhavsar, L. Biro, S. Britton, D. Brown, M. Callander, C. Chang, J. Clouser, R. Davies, D. Dever, N. Dohm, R. Dupcak, J. Emer, N. Fairbanks, B. Fields, M. Gowan, R. Gries, J. Hagan, C. Hanks, R. Hokinson, C. Houghton, J. Huggins, D. Jackson, D. Katz, J. Kowaleski, J. Krause, J. Kumpf, G. Lowney, M. Matson, P. McKernan, S. Meier, J. Mylius, K. Menzel, D. Morgan, T. Morse, L. Noack, N. O'Neill, S. Park, P. Patsis, M. Petronino, J. Pickholtz, M. Quinn, C. Ramey, D. Ramey, E. Rasmussen, N. Raughley, M. Reilly, S. Root, E. Samberg, S. Samudrala, D. Sarrazin, S. Sayadi, D. Siegrist, Y. Seok, T. Sperber, R. Stamm, J. St Laurent, J. Sun, R. Tan, S. Taylor, S. Thierauf, G. Vernes, V. von Kaenel, D. Webb, J. Wiedemeier, K. Wilcox, and T. Zou.

References

1. D. Dobberpuhl et al., "A 200 MHz 64-bit Dual Issue CMOS Microprocessor," *IEEE J. Solid State Circuits*, Vol. 27, No. 11, Nov. 1992, pp. 1,555–1,567.
2. J. Edmondson et al., "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro*, Vol. 15, No. 2, Apr. 1995; pp. 33–43.
3. B. Gieseke et al., "A 600 MHz Superscalar RISC Microprocessor with Out-of-Order Execution," *IEEE Int'l Solid-State Circuits Conf. Dig.*, *Tech. Papers*, IEEE Press, Piscataway, N.J., Feb. 1997, pp. 176–177.
4. D. Leibholz and R. Razdan, "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor," *Proc. IEEE Compcon 97*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 28–36.
5. R.E. Kessler, E.J. McLellan, and D.A. Webb, "The Alpha 21264 Microprocessor Architecture," *Proc. 1998 IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors*, IEEE Computer Soc. Press, Oct. 1998, pp. 90–95.
6. M. Matson et al., "Circuit Implementation of a 600 MHz Superscalar RISC Microprocessor," *1998 IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors*, Oct. 1998, pp. 104–110.
7. J.D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High-Performance Computers," Univ. of Virginia, Dept. of Computer Science, Charlottesville, Va.; http://www.cs.virginia.edu/stream/.
8. S. McFarling, *Combining Branch Predictors*, Tech. Note TN-36, Compaq Computer Corp. Western Research Laboratory, Palo Alto, Calif., June 1993; http://www.research.digital.com/wrl/techreports/abstracts/TN-36.html.
9. T. Fischer and D. Leibholz, "Design Tradeoffs in Stall-Control Circuits for 600 MHz Instruction Queues," *Proc. IEEE Int'l Solid-State Circuits Conf. Dig.*, *Tech. Papers*, IEEE Press, Feb. 1998, pp. 398–399.

Richard E. Kessler is a consulting engineer in the Alpha Development Group of Compaq Computer Corp. in Shrewsbury, Massachusetts. He is an architect of the Alpha 21264 and 21364 microprocessors. His interests include microprocessor and computer system architecture. He has an MS and a PhD in computer sciences from the University of Wisconsin, Madison, and a BS in electrical and computer engineering from the University of Iowa. He is a member of the ACM and the IEEE.

Contact Kessler about this article at Compaq Computer Corp., 334 South St., Shrewsbury, MA 01545; richard.kessler@compaq.com.