



# Superscalar Instruction Execution in the 21164 Alpha Microprocessor

**The 21164 is a new quad-issue, superscalar Alpha microprocessor that executes 1.2 billion instructions per second. Available this January, the 300-MHz, 0.5- $\mu$ m CMOS chip delivers an estimated 345/505 SPECint92/SPECfp92 performance. The design's high clock rate, low operational latency, and high-throughput/nonblocking memory systems contribute to this performance.**

**John H. Edmondson**

**Paul Rubinfeld**

**Ronald Preston**

*Digital Equipment Corporation*

**Vidya Rajagopalan**

*Quantum Effect Design Inc.*

**I**n September 1994, Digital Equipment Corporation introduced the 21164 Alpha microprocessor. This processor exceeded the performance level of existing Alpha microprocessors by over 50 percent and delivered exceptional performance on computing-intensive applications such as large database manipulation, scientific and technical simulation, CAD, and powerful user interfaces. The first microprocessor to achieve over 300/500 SPECint92/SPECfp92 in a uniprocessor system, the 21164 has been shipping to customers since January 1995.

This superscalar (quad-issue) processor is the second completely new implementation of Alpha.<sup>1-3</sup> (Digital Semiconductor introduced the 200-MHz 21064 in 1992 and the 275-MHz 21064A in 1994.) Designed in 0.5- $\mu$ m CMOS technology, the 21164 currently achieves an estimated 345/505 SPECint92/SPECfp92 performance when operating at 300 MHz. Figure 1 (next page) shows a photomicrograph of the 16.5 $\times$ 18.1-mm<sup>2</sup> chip, which contains 9.3 million transistors.

Key to the Alpha 21164 architectural performance are its four-way superscalar instruction issue; low latencies in functional units; high-throughput, nonblocking memory subsystem with low-latency primary caches; and large second-level, on-chip write-back cache.<sup>2,5</sup>

The chip operates 10 percent faster than the previous 21064 implementation<sup>5</sup> would if scaled

into the 0.5- $\mu$ m CMOS technology. To enable the extremely high CPU clock speed, we made certain design trade-offs in the choice of cache hierarchy, instruction-issue algorithm, and execution unit clocking strategy.

## Architecture

Figure 2's block diagram on page 35 shows the chip's five functional units: instruction, integer, floating-point, memory, and cache control/bus interface (called the CBOX). Note the three on-chip caches and the pipeline stages in which each unit and cache operates.

**The functional units.** The four-way superscalar instruction unit forms the control center for fetching, decoding, issuing, and retiring instructions. It controls all data bypasses and register file writes, and handles exceptions, traps, and interrupts. This four-stage pipelined unit implements sophisticated control flow prediction algorithms to minimize the cost of control flow changes.

The integer unit executes integer instructions, calculates virtual addresses for all load and store instructions, and executes all control instructions except floating-point conditional branches. It includes the integer register file and several integer functional units, most of which are contained in two parallel pipelines. Each pipeline operates in four stages and contains an adder and a Boolean logic unit. The first pipeline also contains a shifter,

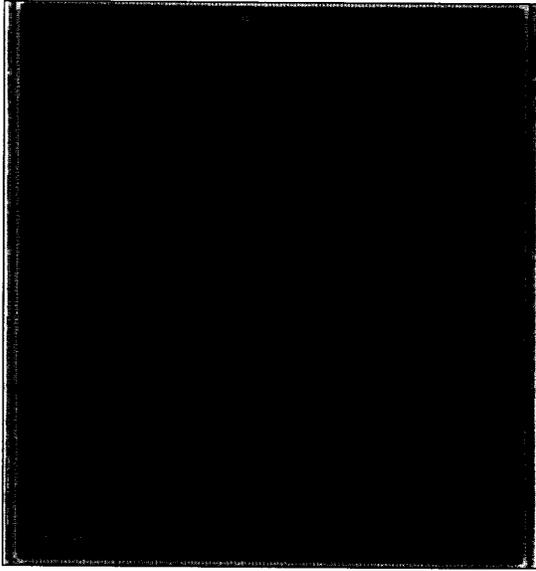


Figure 1. Photomicrograph of the 16.5x18.1-mm<sup>2</sup> Alpha 21164 microprocessor.

### The AXP architecture

Alpha's AXP architecture,<sup>1</sup> first introduced in 1992 as a scalable RISC architecture, supports 64-bit addresses and data types, and deeply pipelined, superscalar designs that operate with a very high clock rate. The architecture includes conditional instructions that help avoid branches that disrupt deep pipelines. It does not contain the branch delay slots, register windows, or resource bottlenecks found in older RISC architectures, which hinder efficient superscalar implementations.

The architecture eliminates complex operations and simplifies fundamental control loops to allow fast cycle time implementations. The deeply pipelined, superscalar internal organization and 300-MHz operating frequency of the Alpha 21164 support these claims.

### References

1. R.L. Sites, "Alpha AXP Architecture," Special Issue, *Digital Technical J.*, Vol. 4, No. 4, 1992, pp. 19-34.

while the second pipeline holds the control instruction execution unit. The first pipeline connects to the partially pipelined integer multiplier, which operates in the background. Except

for the issue cycle and a cycle to return the result, the first pipeline and integer multiplier operate in parallel.

The floating-point unit consists of the floating-point register file and two pipelined execution units: an add pipeline that executes all floating-point instructions except for multiply, and a multiply pipeline that executes floating-point multiply instructions. Both pipelines can operate on IEEE and VAX format data. A floating-point divider connects to the add pipeline, and, except for the issue cycle and a cycle to return the result, the add pipeline and the divider operate in parallel.

The memory unit translates virtual addresses to physical addresses and executes load, store, and memory barrier instructions. It can execute two load instructions or one store instruction per cycle and detects and manages all memory access conflicts. This unit buffers load instructions that miss in the data cache in a structure called the miss address file (MAF) and buffers store instructions in the write buffer.

The CBOX completes read and write commands received from the memory unit, and manages the second- and third-level caches as multiprocessor-coherent, write-back caches.<sup>2</sup> It interfaces with memory and I/O, and implements the multiprocessor cache coherence protocol.

**Caches.** An on-chip second-level cache and an optional off-chip third-level cache back up first-level, 8-Kbyte, direct-mapped instruction and data caches. The 96-Kbyte second-level cache is three-way set associative; the direct-mapped, off-chip cache can range from 1 Mbyte to 64 Mbytes in size. The write-back second- and third-level caches hold instructions and data. The dual-read-ported, write-through data cache supports dual-issue and parallel execution of load instructions.

The instruction and data caches contain 32-byte blocks. The second-level cache contains one tag per 64-byte block and supports full 64-byte block and 32-byte subblock fills. System designers can configure the optional third-level cache to 32- or 64-byte block sizes.

Low-order physical address bits address each cache. The instruction cache tags store virtual addresses for non-PAL mode code, while the data and second-level cache tags store physical addresses. The instruction cache tags support caching PAL code through a tag bit that indicates the tag contains a physical address. (PAL, or Privileged Architecture Library, refers to physically addressed code executed in a privileged hardware mode that implements an architecturally defined interface between the operating system and the hardware.)

The second-level cache tags contain bits to record which 16-byte subblocks have been modified since the block was brought on chip. When a block is copied back to the off-chip cache, the chip transfers only modified 16-byte subblocks to reduce the average time required to write back a block.

We invested considerable effort into improving load latency in the 21164 relative to the 21064. Reducing this latency is important for performance because loads are used frequent-

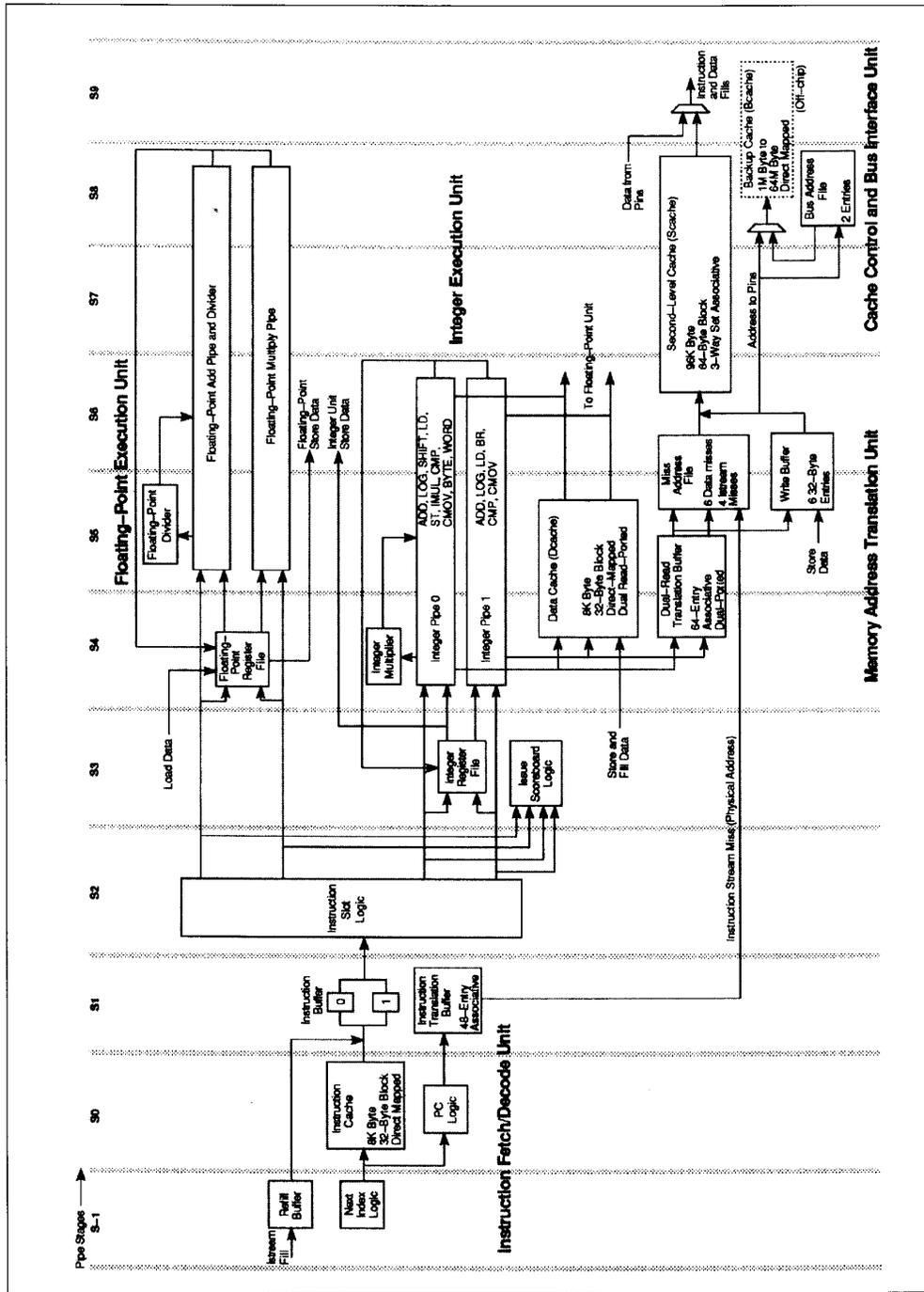


Figure 2. Block diagram of the 21164. Note the five functional units: instruction, integer, floating-point, memory, and the CBOX (cache control/bus interface unit).

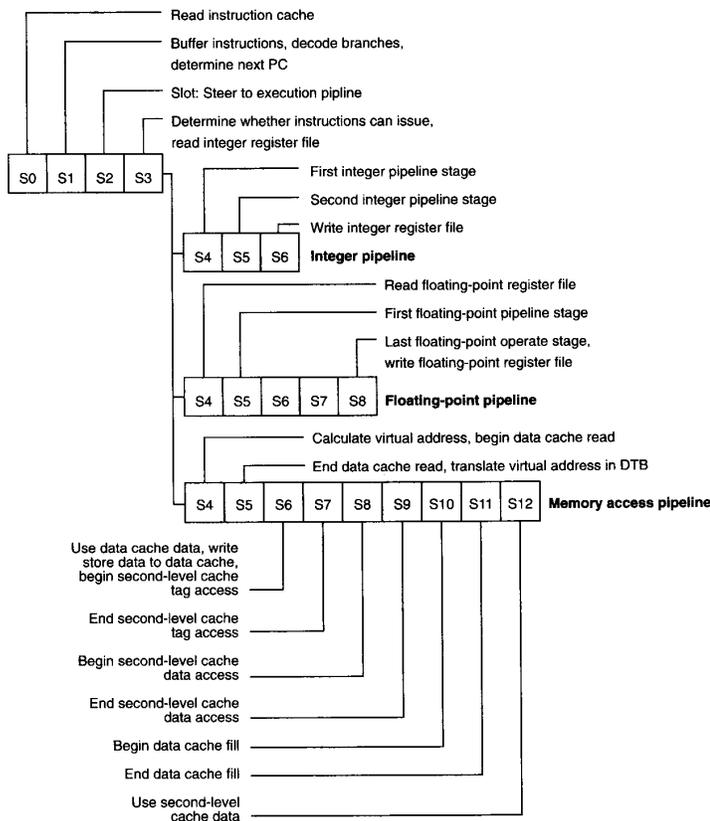


Figure 3. The 21164 instruction pipeline, which executes in seven integer, nine floating-point, and up to 12 on-chip memory instruction stages.

ly and load latency can be a dominant factor in program execution time. By improving the circuit design, we reduced the 21164's latency of load instructions that hit in the data cache relative to the 21064 from three to two cycles.

The two-level, on-chip cache hierarchy enabled us to reduce latency without a cycle-time penalty because it kept the data cache relatively small. We considered a single-level data cache design in which the data cache was much larger. But such a design would not have achieved the reduced data-cache latency realized in the current design: For circuit reasons, physically larger caches are slower than smaller caches. Performance evaluation using our trace-driven performance model showed that the effective latency of reads is better in the two-level design with its faster primary cache and large second-level cache than in the single-level design. The final performance of the current design is better because

it simultaneously reduces the effective latency and cycle time.

**Address translation.** The chip provides two fully associative address translation buffers: a 64-entry data translation buffer in the memory unit and a 48-entry instruction translation buffer in the instruction unit. Each buffer entry can be configured to map 1, 8, 64, or 512 contiguous 8-Kbyte pages.

**Data path widths.** The integer and floating-point execution units are 64-bits wide, and the instruction pipeline is 128-bits wide at each stage. The buses that carry data to and from the second-level cache are 128-bits wide, including the bus used to fill the first-level caches. Finally, access to off-chip cache and memory occurs through the chip's 128 data pins.

### Instruction pipeline

The 21164 pipeline is seven stages long for integer execution, nine stages for floating-point execution, and up to 12 stages for on-chip memory instruction execution. Additional stages are needed for off-chip memory instruction execution. Figure 3 depicts the pipeline for integer, floating-point, and memory operations.

**Cache stages.** The pipeline begins with instruction cache access in stage S0. We normally consider the logic in S-1 (the stage before S0) to operate in S1 of the pipeline. Calculation of a

new instruction cache address takes place in S-1, either by incrementing the previous address or selecting a new address in response to a predicted or actual flow change operation.

During S0, the instruction cache returns a naturally aligned block of four instructions (16 bytes) along with 20 bits of precalculated instruction decode information (5 bits per instruction). This extra information is used in S1 for branch and jump processing and in S2 for instruction slotting. (Instruction slotting is the process of steering each instruction to the proper function unit).

**Prefetcher and refill buffer.** The instruction prefetcher operates in parallel with the instruction cache. When an instruction is not in either the instruction cache or refill buffer, the prefetcher generates a stream of 32-byte instruction block fetch requests. Each 32-byte instruction block contains eight instructions. The instruction unit writes fetched instruction data into

one entry of the four-entry refill buffer when it is returned. When needed, the integer unit moves 16-byte subblocks of instruction data from the refill buffer to the instruction buffer and updates the instruction cache. If this data movement empties a refill buffer entry, an additional fetch request starts through the memory pipeline. We decided to buffer fetched instruction data in the refill buffer rather than the instruction cache to avoid evicting valid cache blocks unnecessarily.

The refill buffer is a type of stream buffer. Each entry stores a virtual address and has a comparator so the refill buffer can be probed for instruction data on a cache miss. Instruction fetching begins only if an access misses in both the instruction cache and the refill buffer. Fetching stops when any instruction flow change occurs (that is, branch, jump, exception). It also stops if at any time the instructions needed in S1 can be found in the instruction cache.

The combination of the on-chip second-level cache and the instruction prefetcher significantly reduces the benefit of enlarging the instruction cache beyond its current size. The prefetcher generates requests at a high rate. Because it is on chip, the second-level cache has the bandwidth to handle these quickly and with relatively little effect on data stream requests. In general, the performance benefit from making the instruction cache larger is very small. This is one of the benefits of the two-level, on-chip cache hierarchy.

**Buffer stage.** In S1, the chip copies a 16-byte instruction block into one entry of the two-entry instruction buffer and calculates instruction cache and refill buffer hits.

Each individual instruction in the 16-byte block in the instruction buffer receives a valid bit tag. When a program branches into the middle of a block, the instructions prior to that instruction are marked invalid.

The branch prediction logic examines the block of instructions coming from the instruction cache or refill buffer during S1. It checks the block for control instructions (conditional branches, jumps, subroutine return instructions, and other flow change instructions), calculates the new fetch address, and sends it to the instruction cache. If there is more than one control instruction, the first unconditional flow change instruction or predicted-taken conditional branch takes priority. Since the new fetch address is available at the end of S1, the read of the instruction cache for the target instruction occurs in the next cycle. This means the control instruction is in S2 at the same time as the target instruction is in S0, resulting in a one-cycle branch delay that creates an empty cycle in the pipeline.

We designed the instruction buffer to opportunistically eliminate the empty cycle following a branch. Whenever any valid instructions in S2 stall, the instruction buffer quashes an empty cycle. In essence, an S2 stall lets the instruction buffer get one instruction block ahead. A taken branch causes the instruction buffer to fall one block behind.

The branch prediction logic predicts conditional branch

instructions using a branch history table addressed by low-order bits of the program counter (PC) that contains 2,048 two-bit saturating counters. Each counter increments on taken branches and decrements on not-taken branches. A branch is predicted to be taken if the current counter value is one of the top two counts and is not predicted to be taken if it is one of the bottom two counts. Four separate displacement adders produce potential target program counters in parallel with the evaluation of the branch predictions. The logic then selects the program counter corresponding to the earliest predicted-taken branch.

A 12-entry return address stack predicts the target address on subroutine returns (RET, JSR\_COROUTINE) and returns from PAL code. Each entry stores 11 bits of address, which is sufficient to address the 8-Kbyte instruction cache. The value stored in the instruction cache tag that is addressed by the return address stack predicts the upper 32 bits of the target address. The same basic mechanism predicts the full target address of jump and jump-type subroutine call instructions since the Alpha architecture provides a "hint" field in these instructions to indicate the target cache address.

The 21164 recovers from incorrect branch and program counter predictions by taking a "mispredict" trap when the incorrectly predicted branch or jump instruction actually executes in the integer or floating-point unit.

**Instruction translation buffer access.** During S1, the instruction unit checks these buffer entries for a match with the program counter. If it finds a page table entry (PTE), it checks the page's protection bits against the current operating mode. If it does not find a PTE, a miss trap occurs. If it finds the PTE in the instruction translation buffer but the access is an instruction cache miss, the instruction translation buffer supplies the physical page translation to the prefetcher.

**Slotting stage.** The main function of the S2 stage is steering each instruction to an appropriate function unit, a process called instruction slotting. Instruction slotting resolves all static execution conflicts. The instruction slotter places the next four-instruction block from the instruction buffer into a staging register and routes the individual instructions to the appropriate instruction pipelines.

The slotter can slot all four instructions in a single cycle if the block contains a mix of integer and floating-point instructions that can issue together. It slots instructions strictly in program order. In the first cycle in which S3 is not stalled, S2 advances to S3 as many instructions from the instruction block as can possibly issue together. The slotter slots the remaining instructions in the block and advances them in subsequent cycles. A new four-instruction block enters S2 when every instruction in the prior block has been advanced to S3.

**Issue stage.** S3 performs all dynamic conflict checks on the set of instructions advanced from S2. S3 contains a complex register scoreboard to check for read-after-write and write-after-write register conflicts. This stage also detects

function unit busy conflicts, which can occur because the integer multiplier and floating-point divider are not fully pipelined. The register scoreboard logic detects all integer and floating-point operand bypass cases and drives the necessary bypass control signals.

This stage issues instructions to the execution units if they encounter no conflicts. If there is a conflict, S3 does not issue the conflicting instruction or any logically subsequent instruction. The advance of the next set of slotted instructions from S2 to S3 stalls until all of the instructions in the current slotted set have been issued.

To check conflicts and handle exceptions (including traps and interrupts), the instruction unit tracks issued instructions from stages S4 through S8. The instruction unit sends register file write strobes and register addresses to the integer and floating-point register files for instructions that reach the retire point (S6) without an exception. The unit disables write strobes to prevent instructions that don't complete from updating the register file. Instructions wouldn't complete either because they cause an exception or because they are in the "shadow" of an exception. The shadow of an exception covers instructions that must not execute because they are in the pipeline when an exception is recognized but are logically subsequent to its occurrence.

If necessary, S3 stalls for a single cycle to permit the integer multiplier or floating-point divider to return a result into their associated pipelines. This is necessary when another instruction would have issued in the associated pipeline because the register files do not have extra write ports dedicated to receiving these results. S3 also stalls for a single cycle in similar cases so data fills for loads that missed in the data cache can write to the register file and data cache. The issue stage stalls indefinitely when necessary for correct execution of trap barrier and memory barrier instructions.

**No-ops in instruction schedules.** New instructions move into the slotting and issue stages when a given stage becomes completely empty. Compared to the ideal design in which instructions are shifted in to partially fill a given stage, this design slightly increases the average cycles-per-instruction ratio. We considered the alternative in which instructions are shifted in as slots become available. This alternative would have created critical paths that would add at least 10 percent to the CPU cycle time. The performance model showed the alternative did not improve the cycles-per-instruction metric enough to compensate for the increase in cycle time, so we chose the simpler and faster design.

Compilers and assembly language programmers can insert no-op instructions to minimize, and in most cases eliminate, any negative performance effect. To facilitate this process, the 21164 handles three different kinds of no-ops.

The first two are the integer no-op (NOP) and the floating-point no-op (FNOP). NOPs (BIS R31,R31,R31) can issue in either integer execution pipeline, and FNOPs (CPYS

F31,F31,F31) can issue in either floating-point execution pipeline. The compiler uses these to improve performance whenever a pair of instructions would be slotted together even though they cannot issue in the same cycle. If one instruction in a pair depends on the other, issuing them together guarantees the second will stall in the issue stage and prevent later instructions from entering that stage. The compiler inserts a NOP or FNOP to delay the issue of the second instruction. With this improvement, the second instruction can issue along with later instructions.

The slotting stage detects the third kind of no-op, the universal no-op [UNOP, LDQ\_U R31,0(Rnn)]. It discards UNOPs so that they don't require an issue slot in either pipeline. UNOPs allow compilers to align instructions without using pipeline issue slots unnecessarily. For example, the compiler can align the target of a branch without necessarily slowing execution of the fall-through path to that branch.

### Execution pipelines

The pipeline for integer instruction execution begins with the register file read in S3. Execution begins in S4 and completes at the end of S4, except for the integer multiply and conditional move instructions. The second and last stage for conditional move instructions is S5. The integer unit writes all results to the integer register file in S6. An integer multiply continues execution in the background and later requests that the issue stage insert a free cycle in the first integer pipeline to allow its result to be written.

The pipeline for floating-point instruction execution begins with the register file read in S4. Execution occurs in S5 through S8, except for floating-point divides. The floating-point unit writes results to the floating-point register file at the end of S8. A floating-point divide continues execution in the background and later requests that the issue stage insert a free cycle in the floating-point add pipeline to allow its result to be written.

**Register files and bypasses.** The integer register file contains 40 registers: the 32 integer registers specified by the architecture (R0 through R31) with R31 always reading as zero; and eight shadow registers available to PAL code as scratch space. Four read ports (two for each pipeline) and two write ports (one for each pipeline) can access the register file.

The integer unit provides bypass paths that allow all subunits other than the multiplier to receive and use the result of a previous instruction from any stage (S4-S6) of either pipeline. Due to implementation constraints, the multiplier can only receive bypassed data from stage S6 of the pipeline. This increases multiply latency by up to two cycles when preceding integer operate instructions produce multiply input operands.

The floating-point register file contains nine ports. Two read ports and one write port per functional unit permit source and destination operand accesses, one read port sup-

ports floating-point stores, and two write ports support two floating-point loads per cycle.

The floating-point unit provides bypass paths that permit data to be forwarded from each of the four floating-point register file write buses to each of the five read buses.

**Latencies.** Most instructions executed in the integer execution unit incur a latency of one cycle. Conditional moves have a latency of two cycles.

Integer multiply latencies depend on the operation and data size. Each 32-bit integer multiply has an eight-cycle latency, and the multiplier can start a second multiply after four cycles provided that the second multiply does not depend on data from the first. For 64-bit signed and unsigned multiplies, the latencies are 12 and 14 cycles. For both of these instructions, the multiplier can start a nondependent multiply after eight cycles. However, multiplies can only receive data from the register file or from S6 of a pipeline. This adds two cycles of additional latency to a multiply that requires data from S4 of a pipeline, and one cycle to a multiply that requires data from S5 of a pipeline.

Compare and Boolean logic instructions produce their result with effectively no delay when a conditional move or a branch test input operand is the result of an immediately preceding compare or Boolean logic instruction. The integer unit uses a special bypass to issue the producer and consumer instructions at the same time in this case.

The latency of all floating-point instructions except floating-point divide is four cycles. The floating-point divider latency depends on the data, determining from 1 to 4 quotient bits per cycle (2.4 on average). With start-up and completion overhead, single-precision divides take from 15 to 31 cycles (19 cycles on average) and double-precision divides take from 22 to 60 cycles (31 cycles on average).

To realize full benefit from the increased issue width relative to the 21064, we had to reduce operational latencies. As the issue width increases, the cost in instruction execution opportunities for a given latency increases. Relative to the 21064, the integer unit reduces the following latencies: shifter (from two cycles to one), byte and word operation (from two cycles to one), and multiplier (from 19 to 23 cycles in the 21064 to eight to 16 cycles in the 21164). Also the special bypass for conditional instructions reduces that latency from one to zero cycles. The floating-point unit designers reduced the latency for all operations except divide from six to four cycles. For single-precision divide, they reduced latency from 34 cycles to an average of 19 and, for double-precision, from 63 to an average of 31.

For the most part, designers improved circuit designs to reduce latency. In the case of integer multiply, the designers implemented a larger multiplier array to reduce latency. For floating-point divide, they implemented an algorithm that required an improved adder circuit design that can complete an add in half a cycle.<sup>6</sup>

## Memory instructions

Memory access instructions proceed as follows in the pipeline. Virtual address calculation occurs and data cache access begins in S4 (for up to two loads or one store); data cache access completes, address translation occurs, and a cache hit is calculated in S5. If the access hits, the memory unit writes data to the register file (load case) or the cache (store case) in S6.

The pipeline continues in the case of a data cache miss as follows. Second-level cache access begins in S6 and ends in S9; the data cache fill begins in S10; the integer register file write occurs, and the data cache fill completes in S11. A dependent instruction can begin execution in S12.

One key to performance is that the memory unit and the CBOX process cache misses in a nonblocking fashion at all hierarchy levels: The pipelines don't stall simply because of a cache miss. The pipelines stall if an instruction is ready to issue but its data isn't ready or if needed instruction stream data is not available. The advantages of this nonblocking design are

1. exploiting available parallelism in the memory subsystem by working on more than one access at a time (hit under miss and miss under miss), and
2. allowing code to be scheduled for longer latency when appropriate (for example, software pipelining techniques).

Load instructions access the data cache and return data to the appropriate register file if there is a hit. For loads that miss, the memory unit holds the physical addresses in the miss address file (MAF) where they are buffered while waiting to access the second-level cache. Store instructions write the data cache if there is a hit and are always buffered in a 32-byte write buffer entry where they wait to update the second-level cache.

For integer load instructions that hit in the data cache, the memory unit multiplexes the result of the load into the output of stage S5 of the pipeline in which the load issued, and writes to the register file through the write port associated with that pipeline. Later, the CBOX returns the data for loads that miss in the data cache. The memory unit multiplexes the data into the output of stage S5 as before, and the issue stage inserts a properly timed NOP cycle by stalling instruction issue for one cycle to make the pipeline's register write port available.

Floating-point load instructions issue to an integer unit for address calculation. The floating-point unit converts the data from its memory format to the standard floating-point register format. The floating-point unit reads its register file source operands one cycle later in the pipeline than the integer unit reads its register file source operands. This skew provides an extra cycle for floating-point load format conversion and simplifies the bypass control logic.

Floating-point stores read the register file and convert the data to memory format while the address calculation occurs in the integer unit. They then write to the data cache.

**Data translation buffer access.** The memory unit receives up to two virtual addresses from the integer unit each cycle. The dual-read-ported data translation buffer supports execution of two loads per cycle, and accesses two buffer entries in late S4 and early S5. If PTEs are found, the translated addresses complete the data cache accesses and the memory unit checks the protection bits against the current operating mode. If either PTE is not present, the memory unit requests a data translation buffer miss trap.

**Miss address file (MAF).** This file consists of two sections: one for load misses (called DREADs) and one for instruction fetches and prefetches (called IREF accesses). For DREADs, the miss address file stores the physical address, destination register, and instruction type (for example, 4-byte integer, 8-byte integer, IEEE-S-type, VAX-G-type, and so on). For IREFs, the miss address file stores the physical address.

The memory unit transfers buffered accesses to the CBOX from the miss address file and write buffer at a peak rate of one every two cycles. An arbiter in the memory unit determines which pending request to send next. DREADs have higher priority than writes from the write buffer, and IREFs have the lowest priority of all. The CBOX checks the second- and third-level caches and initiates a memory access if necessary to fetch the requested data.

When the CBOX returns DREAD data, the memory unit uses the information in the corresponding miss address file entry to update the correct register(s) and the register scoreboard in the instruction unit. When the second half of the data arrives, the memory unit removes the DREAD entry from the miss address file.

The CBOX returns IREF data directly to the instruction unit's cache and refill buffer. The memory unit removes the IREF entry from the miss address file as soon as the CBOX accepts the command.

An important performance feature of the 21164 is that the miss address file merges multiple load misses that access the same 32-byte block of memory into a single DREAD request. Each load instruction requests at most 8 bytes of a 32-byte memory block. The miss address file can merge up to four load misses into one DREAD request, improving latency and reducing unnecessary bandwidth consumption in the second-level cache.

To implement merging, the miss address file merge logic detects when a load miss addresses a block that has already been buffered in the DREAD section of the miss address file and adds the new destination register to the existing request. Merging is limited to one load miss per naturally aligned 8-byte portion of the 32-byte block. Also, merging is only permitted for load misses with identical instruction type. The memory unit allocates a new DREAD entry in the miss address file only for load misses that do not merge.

The merge logic supports the peak load instruction issue rate. It can merge up to two load misses per cycle into the

DREAD section and can merge loads that issue together.

The miss address file can hold up to six DREADs, representing up to 21 loads when their addresses allow best-case merging. (The 21 loads are three less than the theoretical maximum. This limit is a by-product of the overflow prevention algorithm.) Requests transfer to the CBOX in the order they were allocated in the miss address file.

**Write buffer.** The write buffer contains six entries that hold up to 32 bytes of data and a physical address. This buffer accumulates stores that write to the same 32-byte block by merging them into one entry. Capable of merging up to one store instruction per cycle, the buffer speed matches the peak store instruction issue rate. There are no restrictions to merging in the write buffer until the actual write of the second-level cache begins. At that time the write buffer stops merging to the entry sent by the memory unit to the cache.

Once a write buffer entry transfers to the CBOX, several steps may be required to complete the write, depending on the presence of the memory block in the second-level cache and its cache coherence state. The CBOX signals the memory unit upon completion of a store operation, and then the memory unit removes the corresponding entry from the write buffer.

**Access ordering.** The memory unit guarantees that all memory accesses to the same address execute in the correct order. Load misses that conflict with stores, and vice versa, set conflict bits in the miss address file or write buffer. These conflict bits prevent the issue of a DREAD or a write until all conflicts have been cleared. If a load accesses a location for which there is an outstanding miss, the later load must be trapped and replayed in the pipeline (as explained later). If a store matches a valid entry in the write buffer and cannot merge with that entry, the memory unit allocates a new entry to the store and prevents it from being sent to the CBOX until the earlier write completes.

**Memory barrier instructions.** The memory unit implements a memory barrier (MB) instruction by retiring all previous load misses and writes before sending the memory barrier to the CBOX. The CBOX optionally sends the memory barrier to the external system before signaling its completion, and the instruction unit stalls new memory instructions until the memory barrier completes its operation.

The memory unit handles a write memory barrier (WMB) instruction by setting a control bit in the write buffer. This bit prevents the memory unit from sending any new writes to the CBOX before the CBOX completes all the writes that were pending at the time the WMB executed. The memory unit also disables further merging to entries that are on the write buffer when a WMB executes.

**Second-level cache transaction flows.** DREADs, IREFs, and writes from the memory unit access the second-level cache after winning the CBOX arbitration. (The CBOX arbitration chooses between the memory unit request, second-level cache fills, operations to complete writes in the

second-level cache, write-back requests, and coherence requests from the external system.) The second-level cache is fully pipelined. Figure 4 shows a case in which a read is followed by a write and both hit in the cache.

The read access in Figure 4 wins arbitration in S5 of the pipeline. The second-level cache tag store is read in S6, and hit is determined in S7. The requested data is read from the cache data store in S8 and driven onto the 128-bit-wide read data bus in S9. The second half of the 32-byte block is read and driven in the next pipeline cycle. The data transfers to an execution unit or the instruction unit, depending on the access type.

The write access in Figure 4 wins arbitration in S5, and the tag store is read in S6. A hit is determined, and the CBOX transfers the data to the cache on the 128-bit write data bus in S7. The cache is written in S8. As before, the second half of the 32-byte write happens in the next pipeline cycle.

The case of a second-level cache miss that evicts a modified block is interesting because of the need to determine which set to fill and the need to remove the block before the off-chip cache returns the data. The modified block that is evicted is called a victim. Figure 5 shows an example in which a DREAD misses in the second-level cache, creating a victim, and hits in the off-chip cache. The figure shows the address being driven immediately to the off-chip cache using a special bypass path. In this example, the off-chip cache access takes four CPU cycles.

In Figure 5, the DREAD wins arbitration in S5, and the miss is detected in S7. The set picked by the random replacement algorithm contains modified data (a victim). The block size in the second-level cache is 64 bytes, so two 32-byte victim

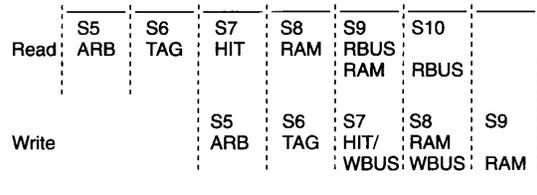


Figure 4. Example of a second-level read and write flow, with both hitting in the cache.

read sequences are needed to copy the entire victim into the on-chip victim buffer. The CBOX gives the two victim reads priority to ensure the victim is copied before the fills from off-chip overwrite the locations.

The 21164 begins driving the off-chip cache address in S8. The tag and data clock into the input pins at the beginning of S12. Before it is possible to determine if the off-chip cache access hits, the CBOX allocates a single cycle in the second-level cache pipeline to reserve a cycle on the read data bus. This cycle allows the incoming data to enter the floating-point and integer pipelines and write the data cache. If the off-chip cache access misses, the cycle allocated by the CBOX becomes a null cycle.

If the access hits in the off-chip cache, the CBOX fills the second-level cache. The fill takes a single cycle in the pipeline, which writes the tag store in S6 and the data store in S8. Note that the second victim read occurs after the first fill. The first victim read sequence always reads the data location overwritten by the first fill.

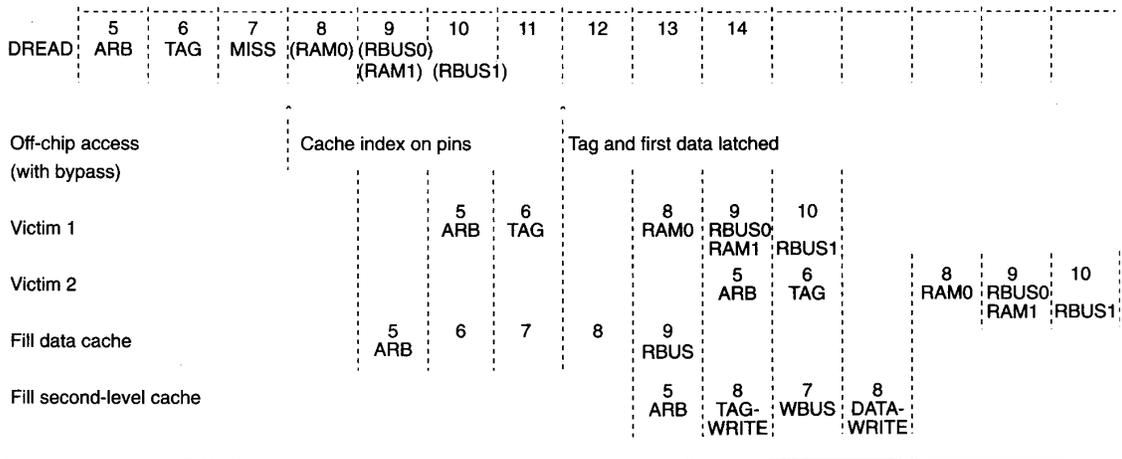


Figure 5. Second-level cache miss sequence with fastest fill possible. A DREAD command misses in the second-level cache, creates a victim, and hits in the off-chip cache.

## Exception handling

In a number of situations, instruction execution must be aborted in the 21164 pipeline. These events are called traps. A trap cancels an instruction and all its successors in the pipeline. Then it causes instruction execution to begin again from a specified address. Traps occur for operation exceptions (arithmetic overflow, translation buffer miss), interrupts, control flow mispredicts, and replay traps (described later).

The instruction unit disables register file writes and the signals that initiate memory operations for the number of cycles necessary to drain the execution pipelines. Disabling starts when the instruction taking the trap is about to write the register file or initiate a memory operation. Superscalar execution complicates trap implementation because, generally, some of the instructions in the same stage as the instruction taking the trap must complete and some must not. Only the instruction taking the trap and all logically subsequent instructions are disabled.

For performance reasons, we allow different traps to signal in different pipeline stages. Priority goes to traps taken on the logically earliest instruction, but sometimes the logically earliest trap is recognized later in the pipeline and therefore later in time. For this reason, the trap mechanism keeps track of the instruction that caused the currently recognized trap and, if an earlier instruction causes a trap later in time, the earlier instruction's trap supercedes the currently recognized trap.

Another complication is that a single instruction may cause multiple traps. The instruction unit uses a priority scheme to decide between traps that may be signaled simultaneously. Once a given instruction has taken a trap, later traps caused by that instruction are ignored. This is because traps that are signaled later in the pipeline for a given instruction are always lower priority than those recognized earlier in the pipeline for that same instruction.

Traps for exceptions are sometimes precise (meaning that the instruction that caused the trap actually takes the trap). The precise exceptions are instruction translation buffer miss, instruction stream access violation, illegal opcode, floating-point instruction issued with floating-point unit disabled, data translation buffer miss, unaligned data stream access, and data stream access violation. The imprecise exceptions are reset, interrupts, machine check, and all arithmetic exceptions. For imprecise exceptions, the instruction unit traps on a later instruction than the instruction that caused the trap. For all exceptions, the instruction unit dispatches control to PAL code. For an instruction translation buffer and data translation buffer miss, PAL code fills the appropriate translation buffer and restarts the program. In other cases, PAL code passes the exception to the operating system.

We treat interrupts as imprecise exceptions. The trap is taken on the next valid instruction entering the pipeline.

The instruction unit handles branch mispredicts and program counter mispredicts, cancelling the incorrect instruc-

tions and immediately restarting execution at the correct target program counter.

## Replay traps

Replay traps handle various pipeline and resource conflicts. In a replay trap, the instruction unit prevents completion of a given instruction by trapping the instruction and then immediately restarting execution with that instruction. This effectively replays the instruction from the beginning of the pipeline.

Replay traps occur in the following cases: correctable ECC error from off-chip cache or memory (giving time for the corrected data to be produced), load-miss-and-use (described later), and various unusual memory unit conflicts and buffer overruns.

The most frequent replay trap is the load-miss-and-use, which occurs whenever a load misses in the data cache and a dependent instruction issues exactly two cycles after the load. The instruction unit makes the issue decision for such a dependent instruction prior to the memory unit determining the load hit in the cache, effectively predicting the load hits. If this prediction is wrong, the dependent instruction restarts from the front of the pipeline and will arrive at the issue stage one cycle before data would typically arrive from the second-level cache. The replayed instruction stalls at the issue stage until the data arrives. Because the replayed instruction arrives at the issue stage before its data arrives, no performance loss occurs due to the trap itself.

Replay traps occur in a number of infrequent memory unit cases. The memory unit forces replay traps to prevent write buffer overflow and miss address file overflow. It also forces a replay trap for certain sequences of loads or stores to the same location when necessary to ensure correct access order and correct results.

In a trade-off that favors high-speed clocks, we used the replay trap mechanism when a stall after S3 would otherwise be required. The ability to stall adds complexity to clocking circuits and latches, particularly in execution unit data paths. Also, stall logic would create many critical paths in detecting the need for a stall and stalling the pipeline. Both of these problems can limit clock speed. Eliminating stalls beyond S3 and instead using the replay trap mechanism solves these problems.

Using replay traps instead of stalls has a theoretical cost: The design with replay traps has a higher average cycles-per-instruction ratio than the design with stalls. The occurrence of a replay trap adds a fixed number of cycles of execution delay. If the cause of the replay trap is resolved in a smaller number of cycles, the remainder of that delay is an unnecessary cost. Most replay traps are very rare, so the negative performance effect is very small.

The load-miss-and-use replay trap does not reduce performance significantly because the condition causing it is

seldom resolved in fewer cycles than it takes to complete the replay trap. There is an obscure case in which its cause is resolved more quickly, but this occurs rarely. Our performance model verified that replay traps do not significantly affect delivered performance.

### Performance

Estimated performance at 300 MHz based on measurements in prototype systems is 345 SPECint92, 505 SPECfp92, SPEC benchmarks. By achieving these performance results, the 21164 delivered the highest performance of any commercially available microprocessor.

CPU PERFORMANCE CONTINUES TO INCREASE rapidly, as it has for many years. This performance increase has enabled rapid growth in the sophistication and power of user interface and application programs. Since 1992, Alpha microprocessors have held the position of highest performance in the industry. The 21164 continues this record of achievement. Applications such as databases, simulation, modeling, CAD, and powerful user interfaces all benefit from the increased CPU performance the 21164 offers.

A number of high-performance RISC microprocessors have been announced recently. Of this new generation, the 21164 is first to be available commercially (shipping since January 1995) and is positioned to remain the performance leader as the others become available.

Digital Equipment Corporation will continue to improve the Alpha line of microprocessors. Future revisions of the 21164 will benefit from advances in process technology, delivering higher performance and lower cost. New Alpha microprocessor designs will extend Digital's performance leadership into the next century. □

### Acknowledgments

We acknowledge the following for their work in producing estimates of SPEC benchmark performance: George Chrysos, Robert Cohn, Zarka Cvetanovic, Kent Glossop, Lucy Hamnett, Steve Hobbs, John Shakshober, and Paula Smith.

### References

1. *Alpha Architecture Reference Manual*, R. Sites, ed., Digital Press, Burlington, Mass., 1992.
2. J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz, 64-Bit, Quad-Issue, CMOS RISC Microprocessor," *Digital Technical J.*, Vol. 7, No. 1, 1995.
3. S. Bell et al., "Circuit Implementation of a 300-MHz, 64-Bit, Second-Generation CMOS Alpha CPU," *Digital Technical J.*, Vol. 7, No. 1, 1995.

4. D.W. Dobberpuhl et al., "A 200-MHz 64-b Dual-Issue CMOS Microprocessor," *IEEE J. Solid-State Circuits*, Vol. 27, No. 11, Nov. 1992, pp. 1555-1564.
5. O.L. MacSorley, "High-Speed Arithmetic in Binary Computers," *Proc. IRE*, Vol. 49, 1961, pp. 67-91.

**John H. Edmondson**, a consulting engineer in the Semiconductor Engineering Group, was the architecture leader on the Alpha 21164 design team. Before that, he was a member of the VAX 6000 Model 600 CPU chip design team. Prior to joining Digital, he worked at Canaan Computer Corporation and Massachusetts General Hospital. Edmondson received a BSEE degree from the Massachusetts Institute of Technology.

**Paul Rubinfeld**, a consulting engineer within Digital's Semiconductor Business Group, was the engineering manager on the Alpha 21164 project. He has worked on VAX and PDP-11 CPU development projects and a SIMD massively parallel processing system at Digital over the last 15 years. Rubinfeld received his BSEE and MSEE degrees from Carnegie Mellon University, where he helped build the Cm\* multiprocessor.

**Ronald Preston**, a principal engineer with Digital's Semiconductor Engineering Group, was the implementation leader for the instruction issue unit of the Alpha 21164. Previously, he worked at Signetics Corporation, Sunnyvale, California. Preston received BSEE and MEE degrees from Rensselaer Polytechnic Institute. He has coauthored several articles on design and reliability analysis of CMOS circuits.

**Vidya Rajagopalan**, currently with Quantum Effect Design Inc., worked on the Alpha 21164 and 21064 designs when with the Semiconductor Engineering Group at Digital Equipment Corporation. She received her MSEE degree from the University of Maryland, College Park, and her BE degree from Visvesvaraya Regional College of Engineering, Nagpur, India.

Direct comments about this article to John H. Edmondson, Digital Equipment Corporation, HLO2-3/D11, 77 Reed Road, Hudson, MA 01749-2809; jhe@tgate.enet.dec.com.

### Reader Interest Survey

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

Low 159

Medium 160

High 161