PowerPC 601 and Alpha 21064: A Tale of Two RISCs

James E. Smith, Cray Research, and Shlomo Weiss, Tel Aviv University

At this point many RISC purists will undoubtedly claim that this is not a RISC design . . . This second generation RISC design, representing a reasonable melding of RISC and CISC concepts is likely to be the direction for many future RISC designs.

P. Hester, *RISC System/6000 Hardware Background and Philosophies*

We reapplied the principles of RISC to processor design to get maximum clock speed.

R. Sites, *RISC Enters a New Generation — An Insider's Look at the Development of DEC's Alpha CPU*

Both PowerPC and Alpha are RISC architectures, but they have little in common beyond that. The design philosophy of one emphasizes powerful instructions, the other simplicity.

Virtually all microprocessor architectures developed in the past 10 years have followed the RISC (reduced instruction set computer) principles articulated by Patterson in 1985.[1] And, not surprisingly, the first-generation RISC implementations developed in the 1980s tended to look alike, with simple, five-stage instruction pipelines (see sidebar). In recent years, however, with more experience and more transistors at their disposal, designers have begun exploring a richly diverse set of architectures and implementations.

Nowhere is this diversity more apparent than in the recent RISC implementations from Digital Equipment Corporation, the Alpha 21064, and from IBM/Motorola/Apple, the PowerPC 601. Both are superscalar implementations; that is, they can sustain execution of two or more instructions per clock cycle. Otherwise, these two implementations present vastly different philosophies for achieving high performance. The PowerPC 601[2-5] focuses on powerful instructions and great flexibility in processing order, while the Alpha 21064[6-9] depends on a very fast clock, with simpler instructions and a more streamlined implementation structure. These two RISC microprocessors exemplify contrasting, but equally valid, implementation philosophies.

The next section, an overview of the instruction sets, emphasizes the differences in design: PowerPC uses powerful instructions so that fewer are needed to get the job done; Alpha uses simple instructions so that the hardware can be kept simpler and faster. The remainder of the article discusses the pipelined implementations of the two architectures; again, the contrast is between powerful and simple.

Architecture overview

Two PowerPC features — floating-point multiply-add instructions and update load/stores — illustrate the powerful-versus-simple approach of the two architectures, as do the differences in the way their instruction sets handle unaligned data, byte string operations, and branch instructions. There are other interesting differences, for example, the memory addressing architectures, but we focus only on those central to the

0018-9162/94/$4.00 © 1994 IEEE

"powerful" and "simple" philosophies.

Both the PowerPC and the Alpha are load/store architectures. That is, all instructions that access memory are either loads or stores, and all operate instructions are from register to register. They both have 32-bit fixed-length instructions and 32 integer and 32 floating-point registers. But they have little in common beyond these basic properties (see Table 1).

To describe the two architectures, we use notation and naming conventions that are mostly consistent with the PowerPC 601. For uniformity, we label bits beginning with 0 at the most significant bit in the left-to-right direction as defined in the PowerPC (and unlike the usual Alpha notation.) Doubleword, word, and halfword are also as defined in the PowerPC, that is, eight, four, and two bytes,

Table 1. Summary of architectural characteristics.

	PowerPC 601	Alpha 21064
Basic architecture	Load/store	Load/store
Instruction length	32 bit	32 bit
Byte/halfword load and store	Yes	No
Condition codes	Yes	No
Conditional moves	No	Yes
Integer registers	32	32
Integer register size	32/64 bit	64 bit
Floating point registers	32	32
Floating register size	64 bit	64 bit
Floating point format	IEEE 32bit, 64bit	IEEE, VAX 32bit, 64bit
Virtual address	52-80 bit	43-64 bit
32/64 mode bit	Yes	No
Segmentation	Yes	No
Page size	4 Kbytes	Implementation specific

The "classic" five-stage RISC pipeline

Pipelining achieves high performance by breaking instruction processing into a series of stages connected like stations in an assembly line. As instructions flow down this assembly line, called the pipeline, the hardware in each stage performs some processing, until instructions leaving the pipeline are completely processed. Pipelining achieves high performance through the parallelism of processing several instructions at once, each in a different pipeline stage.

Pipelining is the fundamental implementation technique used for most RISC architectures. Indeed, exposing certain characteristics of the pipeline's structure to software is the basis for many of the commonly enunciated RISC features. A typical first generation RISC pipeline consists of the following five stages.

- IF, instruction fetching: The program counter fetches the next instruction to be processed. Instructions are usually held in an instruction cache memory that is read during the fetch stage.
- ID, instruction decoding and operand fetching: The opcode and operands are inspected and control signals are generated. Register specifiers from the instruction are used to read operands from the register file(s).
- EX, instruction execution: The operation specified by the opcode is performed. For a memory access instruction, the operation forms the effective memory address.
- ME, memory access: Data is loaded from or stored to memory. A data cache memory is typically used.
- WB, result write-back: The result of the operation is written back into the register file.

Each pipeline stage consists of combinational logic and/or high-speed memory in the form of a register file or cache memory. The stages are separated by ranks of latches or flip-flops (see Figure A).

Figure B shows four sequential instructions flowing down the pipeline. Each row shows the stages the instruction passes through as it flows down the pipeline. In a well-designed pipeline, all stages contain logic with roughly the same signal propagation delay. The pipeline is designed to start a new instruction (and finish an instruction) each clock period. Therefore, the clock period determines throughput, that is, the rate at which instructions can be executed. The time it takes for a single instruction to traverse the entire length of the pipeline is referred to as its latency. Without pipelining, throughput would be the reciprocal of latency. In reality, the nonpipelined throughput might be a little better because the pipeline latches add some overhead to the overall latency.

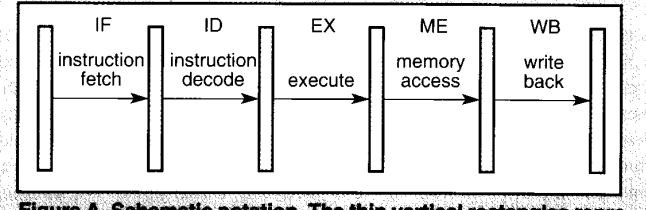


Figure A. Schematic notation. The thin vertical rectangles represent the latches, and the lines with arrows indicate the flow of information — data or instructions.

Clock Cycle:	0	1	2	3	4	5	6	7
Instruction 1	IF	ID	EX	ME	WB			
Instruction 2		IF	ID	EX	ME	WB		
Instruction 3			IF	ID	EX	ME	WB	
Instruction 4				IF	ID	EX	ME	WB

Figure B. An instruction sequence flowing down the pipeline.

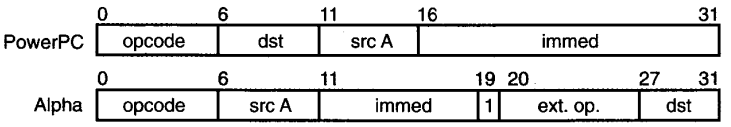
respectively. For examples, we use an assembly language very similar to that which the IBM RS/6000 C compiler produces when a flag is set to generate assembly output. This specific language was chosen for its readability.

Instruction sets. Figures 1 and 2 show the major PowerPC and Alpha instruction formats. (For additional PowerPC instruction formats, see Reference 3.) As you'd expect, since the two architectures have the same RISC underpinnings, the instruction formats are quite similar. The instructions themselves are also similar, but as we are about to see, they differ in "power."

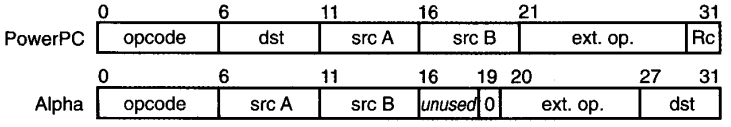
Load/store instructions. The PowerPC architecture has two primary addressing modes: register plus displacement and register plus register. Furthermore, the load and store instructions may automatically update the index register as a byproduct of their execution. That is, the instruction not only performs a simple load or store, as in most RISCs, but also modifies the index register by placing the just-computed effective address in it.

The Alpha architecture has only one primary addressing mode, register plus displacement. As we'll see in the implementation section, this simplifies the register file design. In Alpha, loads and stores do not update the index register.

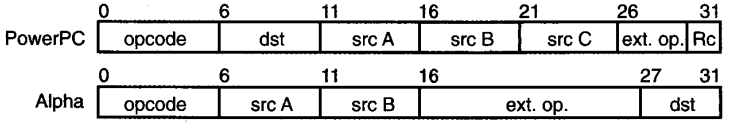
Floating-point multiply-add instructions. The PowerPC includes multiply-add instructions that take three input operands, A, B, and C, and form the result, $A \times C + B$ or $A \times C - B$. With these powerful instructions, one instruction passing through the pipeline can do the work of two, and instruction pipeline resources are used more efficiently. Merging the operations also reduces the latency, that is, the total time taken by a dependent multiply and add. And floating point accuracy can be increased by eliminating the rounding step between the multiply and the add.



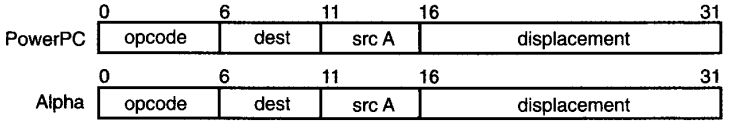
(a) Both architectures have instructions that take a source register (src A) and an immediate field to produce a result in a destination register (dst). The Alpha's immediate field is a little shorter because it has an extended opcode field (ext. op.). The immediate operand is a signed integer in PowerPC and a positive integer in Alpha.



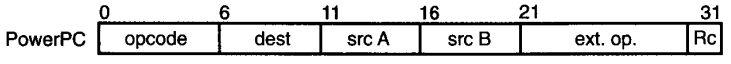
(b) Except for the locations of the fields, the instructions with two source registers (src A and src B) are similar. The PowerPC has a longer extended opcode field. The record bit (Rc) in the PowerPC instructions is to enable explicit updating of condition code flags.



(c) The floating point versions of register-register operate instructions are also similar to the integer version except the PowerPC has three source registers to support a combined floating point multiply-add instruction.



(d) The memory instructions for register + displacement addressing are virtually identical.



(e) Only the PowerPC has a register + register addressing format for loads and stores.

Figure 1. Operate and Memory instruction formats.

Example. Figure 2 illustrates the two instruction sets in general and the value of the update load/store and multiply-add instructions in particular. By design, this example plays into a strength of the PowerPC architecture, so it's not intended to indicate performance. The figure shows a simple C loop that operates on arrays of data and its compilation into PowerPC and Alpha instructions. Because the 601 floating-point pipeline is essentially a single-precision pipeline, which requires an extra pass for double precision computations, the PowerPC code in Figure 2b uses single-precision data. The PowerPC 620, currently under development, will be the first 64-bit PowerPC implementation.

In the PowerPC code in Figure 2b, the load with update at the top of the loop and the store with update near the end of the loop maintain the y and x array pointers, respectively. The PowerPC version needs only two floating point instructions: a multiply and a multiply-add.

The Alpha code in Figure 2c uses double precision. Because the Alpha does not have a register-plus-register addressing mode, pointers to the arrays are indi-

```
for (k = 0; k < 512; k++)
    x[k] = r * x[k] + t * y[k];
```

(a) C code.

```
                                    # r3 + 4 points to x.
                                    # r4 + 4 points to y.
                                    # fp1 contains t,
                                    # fp3 contains r, and
                                    # CTR contains the loop count (512).
LOOP: lfsu    p0 = y(r4 = r4 + 4)   # load floating single with update.
      fmuls   fp0 = fp0,fp1         # floating multiply.
      lfs     fp2 = x(r3,4)         # load floating single.
      fmadds  fp0 = fp0,fp2,fp3     # floating multiply-add single.
      stfsu   x(r3 = r3 + 4) = fp0  # store floating single with update.
      bc      LOOP,CTR ≠ 0          # decrement CTR, then branch if CTR ≠ 0.
```

(b) PowerPC code.

```
                                    # r1 points to x.
                                    # r2 points to y.
                                    # r6 points to the end y.
                                    # fp2 contains t.
                                    # fp4 contains r.
                                    # r5 contains the constant 1.
LOOP: ldt     fp3 = y(r2,0)         # load floating double.
      ldt     fp1 = x(r1,0)         # load floating double.
      mult    fp3 = fp3,fp2         # floating multiply double t * y.
      addq    r2 = r2,8             # bump y pointer.
      mult    fp1 = fp1,fp4         # floating multiply double, r * x.
      subq    r4 = r2,r6            # subtract y end from current pointer.
      addt    fp1 = fp3,fp1         # floating add double, r * x + t * z.
      stt     x(r1,0) = fp1         # store floating double to x[k].
      addq    r1 = r1,8             # bump x pointer.
      bne     r4,LOOP               # branch on r4 ≠ 0.
```

(c) Alpha code (using the PowerPC syntax).

Figure 2. Example of PowerPC and Alpha instructions.

vidually incremented each time through the loop. The pointer to array *y* also tracks the loop count; subtracting the pointer from the address of the end of array *y* generates the value tested by the loop-closing branch. The Alpha version of the loop uses 10 instructions, four more than the PowerPC.

Data alignment. Many RISC implementations simplify their memory load path by optimizing data alignment on natural boundaries. That is, doubleword 8-byte data must align on an address that is a multiple of 8; word data must be on an address that is a multiple of 4, and so on. If a load or store uses an address with an improper multiple, some RISC implementations trap to a trap handler that uses multiple instructions to implement the required memory operation. Not so in the PowerPC 601, which handles unaligned data entirely in hardware. It occasionally requires a second cache access when data crosses a four-word boundary, but this is a property of the cache implementation.

The Alpha architecture handles unaligned data in one of two ways, depending on how often it is actually unaligned. If the data is usually aligned, the compiler can use aligned versions of loads and stores. These will trap if an address should happen to be unaligned, and the trap handler takes care of the unaligned access. If the data is likely to be unaligned, then multiple instruction sequences of unaligned loads and stores can be combined with insert, mask, and extract instructions to get the job done.

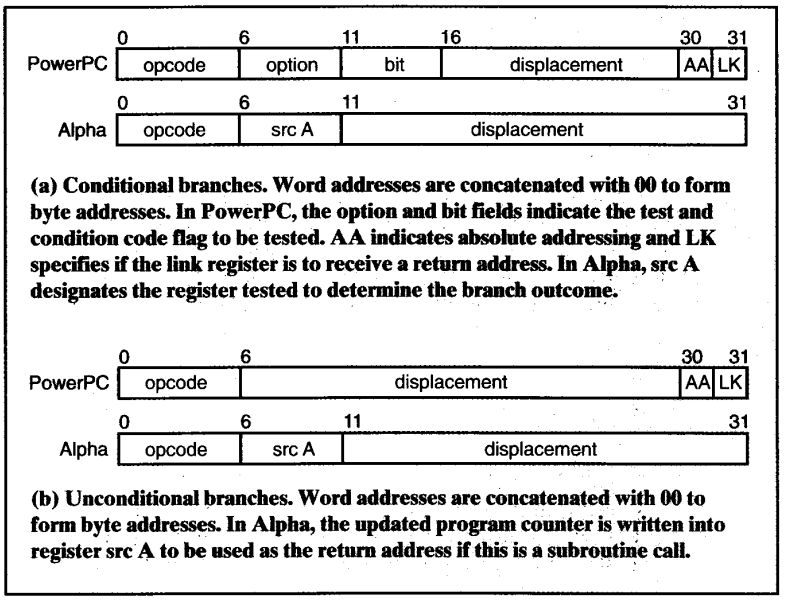
Byte-string operations. The two architectures' handling of byte operations is strikingly different. The PowerPC has byte load-and-store instructions. An Alpha characteristic is that load and store instructions transfer only 32- or 64-bit data between a register and memory; there are no instructions that load or store 8-bit or 16-bit quantities. The Alpha architecture does include a set of instructions to extract and manipulate bytes from registers. (The significance of not having to do select and alignment operations in the memory load path will be made apparent in the data cache implementation section.)

Branch instructions. There are significant differences in the way the two architectures handle branches. Figure 3 compares the format of conditional and unconditional branches. In both architectures, branch target addresses are usually determined by adding a displacement to the program counter (PC relative).

PowerPC includes a special set of registers architected for holding, operating on, and testing conditions. Conditional branches may test fields in the condition code register and the contents of a special register, the count register (CTR). Again using the theme of more powerful instructions, a single branch instruction can implement a loop-closing branch by decrementing the CTR, testing its value, and branching if it is nonzero. The code example in Figure 2b does this. Comparison instructions set fields of the condition code register explicitly, and most arithmetic and logical instructions may optionally set a condition field by using the record (Rc) bit.

In the Alpha architecture, conditional branches test a general-purpose register relative to zero or by odd/even register contents. (The odd/even tests allow for compilers that use the low-order bit to denote true or false logical values.) Thus, results of most instructions can be used directly by conditional branch instructions, as long as they are tested against zero (or odd/even). When needed, comparison instructions leave their result in a general-purpose register.

Certain control transfer instructions save the updated program counter to be used as a subroutine return address. In Alpha, these are special jump instructions that save the return address in a general-purpose register. The PowerPC does this in any branch by setting the link



(a) Conditional branches. Word addresses are concatenated with 00 to form byte addresses. In PowerPC, the option and bit fields indicate the test and condition code flag to be tested. AA indicates absolute addressing and LK specifies if the link register is to receive a return address. In Alpha, src A designates the register tested to determine the branch outcome.

(b) Unconditional branches. Word addresses are concatenated with 00 to form byte addresses. In Alpha, the updated program counter is written into register src A to be used as the return address if this is a subroutine call.

Figure 3. Branch instructions.

(LK) bit to one. The return address is saved in the link register.

Implementations

To compare the PowerPC 601 and Alpha 21064 implementations, we focus first on the overall pipelined implementations. Next, we describe critical areas: instruction fetching, branch processing, instruction dispatching, register files, and data caches. We illustrate important features with pipeline flow diagrams and conclude with a discussion of imprecise interrupts.

Table 2 compares the initial implementations of the PowerPC 601 and Alpha 21064 chips (faster versions using improved process technologies are on the way). Performance comparisons using the SPEC benchmarks (see sidebar) are included for completeness, but our primary interest is the contrasting styles in instruction sets and implementations.

Both the PowerPC 601 and Alpha 21064 use sophisticated pipelined implementations. The PowerPC 601 pipelines are relatively short with more buffering; the Alpha 21064 has deeper pipelines with less internal buffering and a much faster clock (by a factor of about three). The two implementations also use contrasting cache memory designs. The 601 has a unified 32-Kbyte cache; that is, instructions and data reside in the same cache. The 21064 has split data and instruction caches of 8 Kbytes each.

PowerPC 601 pipelines. As Figure 4 shows, the PowerPC 601 has three units, each implemented as a pipeline: the branch unit (BU), the fixed-point unit (FXU), and the floating-point unit (FPU). The BU fetches instructions, executes branches, and dispatches other instructions to the FXU and FPU. The FXU and FPU handle fixed-point and floating-point instructions, respectively, except that the FXU also decodes and executes floating memory instructions. Therefore, in addition to processing fixed-point instructions as its name implies, the FXU functions as a load/store unit.

Here are the pipeline stages.

- F, instruction fetch: The unified cache is accessed, and up to eight instructions are fetched into the instruction buffer.
- S, dispatch: Instructions are dispatched to the FXU and FPU.
- D, decode: Instructions are decoded. Source registers are read in this stage. Note that instructions going to the FXU may be dispatched and decoded in the same pipeline stage (labeled D).
- E, execute: Two different stages have this label. Branches execute in the one in the BU. The other E stage, in the FXU, is where fixed-point instructions execute and load/store instructions do their address processing and cache lookup.
- C, cache access: The cache is accessed, and fixed-point operands are sent directly to the FXU, floating-point operands to the FPU.
- W, write: Results are written to the register file.
- M, multiply: floating-point multiply.
- A, add: floating-point add.

The PowerPC 601 contains several buffers at strategic points in the pipelines: after fetching instructions and after dispatching. Instructions may be dispatched to the BU and FPU out of order relative to the program sequence. By getting instructions out of the instruction buffers even when a pipeline is blocked, instruction dispatching can continue to non-blocked units.

Alpha 21064 pipelines. Figure 5 illustrates the 21064 structure of three parallel pipelines: a fixed-point pipe, a floating-point pipe, and a load/store pipe. These relatively deep pipelines have 10 stages for floating point instructions and seven stages for integer and load/store instructions.

We describe the integer and load/store pipelines together.

Table 2. Summary of implementation characteristics.

	PowerPC 601	Alpha 21064
Technology	0.6-micron CMOS	0.75-micron CMOS
Levels of metal	4	3
Die size	1.09 cm square	2.33 cm square
Transistor count	2.8 million	1.68 million
Total cache (instructions + data)	32 Kbyte	16 Kbyte
Package	304-pin QFP	431-pin PGA
Clock frequency	50 and 66 Mhz	150 to 200 Mhz
Performance:		
SPECint92	63 @ 66 Mhz	117 @ 200 MHz
SPECfp92	72 @ 66 Mhz	194 @ 200 Mhz

- F, instruction fetch: The instruction cache is accessed, and two instructions are fetched.
- S, swap: Two instructions are inspected to see if they require the integer or floating-point pipelines. The instructions are directed to the correct pipeline; this might involve swapping their positions. Branch instructions are predicted in this stage.
- D, decode: Instructions are decoded in preparation for instruction issue; the opcode is inspected to determine the instruction's register and resource requirements. Unlike the 601, registers are not read during the decode stage.
- I, issue: Instructions are issued, and operands are read from the registers. Issue consists of checking register and resource dependencies to determine if the instruction should begin execution or be held back. Once they have passed the issue stage, instructions are no longer blocked in the pipelines; they flow through to completion.

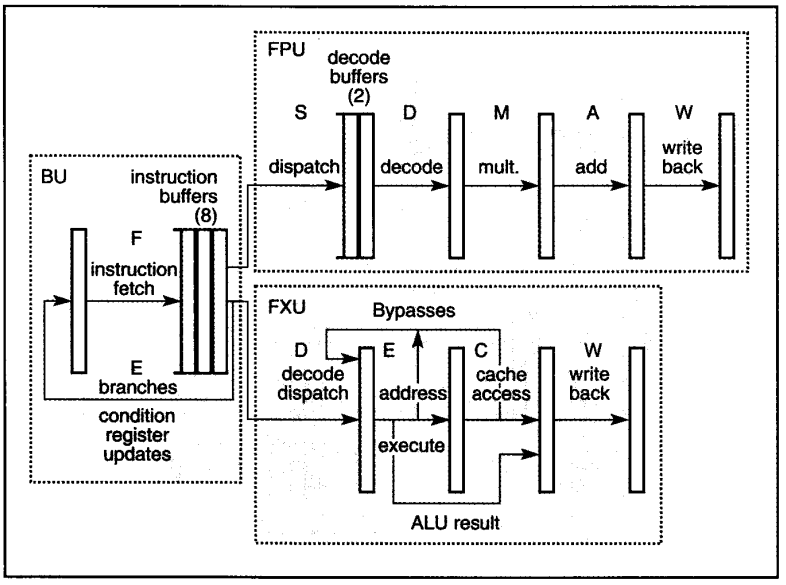


Figure 4. PowerPC 601 pipeline structure.

- A, ALU stage 1: Integer adds, logicals, and short-length shifts are executed. Their results can be immediately bypassed back, so they appear to be single-cycle instructions. Longer length shifts are initiated in

The tale of the tape: SPEC benchmarks

SPEC (Standard Performance Evaluation Corporation) was founded in 1988 by a number of computer companies with the goal of providing a representative set of benchmarks by which their products could be measured. Currently, there are two sets of benchmarks: one consisting of six integer programs and the other consisting of fourteen floating point benchmarks. These benchmarks are often distilled to two performance numbers SPECint92 and SPECfp92, 1992 being the year the benchmarks were introduced. The performance of each benchmark is given as the speedup versus a VAX 11/780 (primarily for historical reasons). SPECfp92 and SPECint92 are the geometric mean of these speedups (found by multiplying speedups on *n* benchmarks and taking the *n*th root of the product).

We give these numbers with a note of caution. Our article discusses processors and the design philosophies that went into them. The specific processor implementations discussed — the PowerPC 601 and the Alpha 21064 — are directed at systems covering different price ranges; there also may be differences in low-level implementation techniques and circuit design, compiler quality, and the system environment (for example, external cache characteristics) in which benchmarks are run.

To provide an overall picture of the PowerPC and Alpha product lines, the table contains SPEC benchmark performance for the 21064, the 21064A, a follow-on in a new process technology, and the 21066, a version with an integrated PCI bus interface. For PowerPC, we include performance for the 66-MHz 601, estimated performance for a 100-MHz 601+, using a new process technology, and the PowerPC 604, a recently announced processor that uses a more aggressive implementation intended for higher performance products.

The table shows that the 21064 appears to be ahead in the performance race, as measured by the SPEC benchmarks. It also shows that while the PowerPC has a slower clock period, the performance results are closer than the clock period alone would suggest. This is consistent with the "more work per clock period" philosophy used in the PowerPC designs.

SPEC performance for a sample of PowerPC and Alpha processors. (Source: *Microprocessor Report*)

Processor	Clock frequency	Availability	SPECint92	SPECfp92
Alpha 21066	166 MHz	4Q93	70	105
Alpha 21064	150 MHz	2Q92	84	128
Alpha 21064	200 MHz	2Q93	117	194
Alpha 21064A	275 MHz	3Q94	170 (est.)	290 (est.)
PowerPC 601	66 MHz	2Q93	63	72
PowerPC 601+	100 MHz	4Q94	105 (est.)	125 (est.)
PowerPC 604	100 MHz	4Q94	160 (est.)	165 (est.)

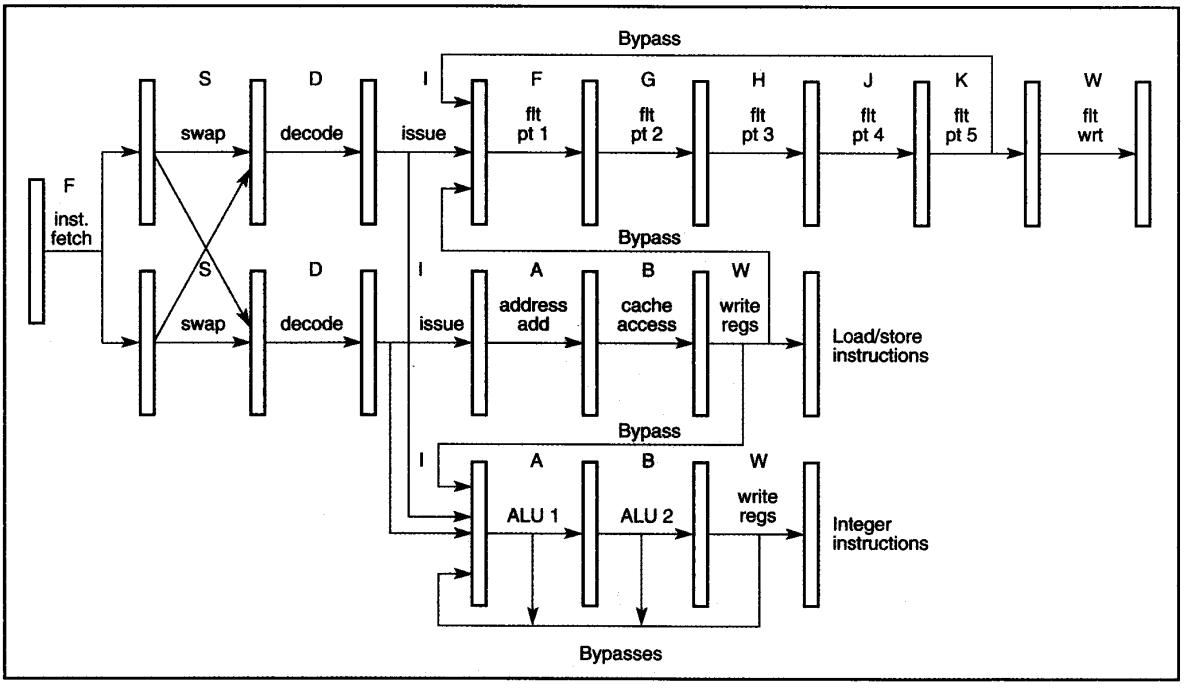


Figure 5. The 21064 pipeline complex. Only a small subset of the bypasses are shown.

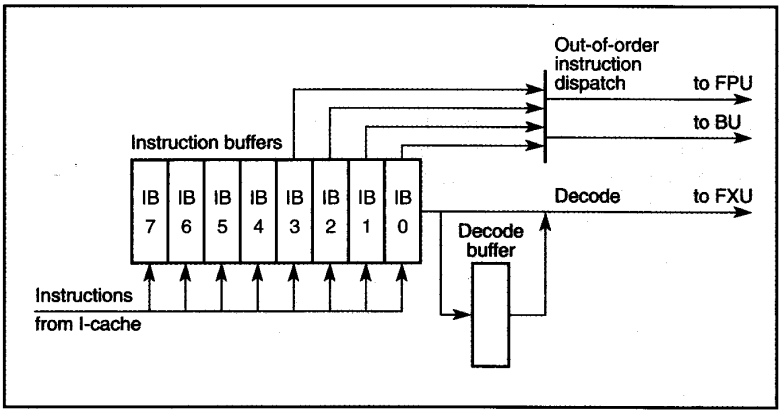


Figure 6. PowerPC 601 instruction dispatching.

this stage. Loads and stores do their effective address add in this stage.

- B, ALU stage 2: Longer length shifts complete in this stage; their results are bypassed back to ALU 1, so these are two-cycle instructions. For loads and stores, the data cache tags are read. Loads also read cache data.
- W, write: Results are written into the register file. Cache hit/miss is determined. Store instructions that hit have their data stored in a buffer. The buffer contents will be written into the cache during a following cycle when there is no load.

The 21064 integer pipeline relies on many bypasses for high performance. These are important in a deep pipeline to reduce apparent latencies. Figure 5 shows a few of the bypasses; there are a total of 38 separate bypass paths.

Referring again to Figure 5, floating-point instructions pass through F, S, D, and I stages just like the integer instructions. There are then five stages (F through K) where floating-point multiply and add instructions are performed. (Note that there are two pipeline stages labeled F: instruction fetch, and the first floating-point stage. We do this because in both cases, F provides the easiest way to remember the pipeline stage. Because the stages are so far apart, this shouldn't lead to any confusion in our diagrams.) The floating-point divide takes 31 or 61 cycles, depending on single or double precision.

The following subsections focus on different stages of instruction processing, proceeding in roughly the same order as instructions are processed.

Instruction fetching. Instruction fetching in the Alpha is very straightforward. Instructions are read from the instruction cache at the rate of two per cycle and are placed in the dual decode registers. In the PowerPC 601, however, the story is much longer. Instructions are fetched from the cache at a rate of up to eight instructions per clock cycle and held in an eight-slot instruction buffer (see Figure 6).

Unlike many RISC implementations, the PowerPC 601 uses a single, unified cache for both instructions and data. Instruction fetching and data accesses must contend for the cache resource. The unified cache has its lines divided into two 32-byte sectors. The sectors share the same address tag, but only one sector at a time is brought into the cache on a miss. This means that occasionally a line will have an invalid sector that holds no useful instructions or data.

Because the PowerPC cache is unified,

there is cache contention and arbitration between instruction and data accesses. Contention for the cache, with instructions having a lower priority than data, provides the reason for fetching up to eight instructions at a time, even though the absolute maximum processing rate is three per clock cycle. When the instruction fetch unit has a chance to fetch, it *fetches!* (It might not have another chance for a while.)

Branch prediction. Both processors predict branches in an effort to reduce pipeline bubbles. The PowerPC 601 uses a static branch prediction made by the compiler. The prediction is conveyed to the hardware by a bit in the branch instruction, which indicates whether the branch is expected to be taken or not. Also, as a hedge against a wrong prediction, the 601 saves (for a while) the contents of the instruction buffer following a branch-taken prediction; these instructions on the not-taken path are available immediately if a miss prediction is detected. The instruction buffer contents are kept until instructions from the taken path are delivered from memory.

The Alpha 21064 implements dynamic branch prediction with a 2048 entry table; one entry is associated with each instruction in the instruction cache. The prediction table is updated as a program runs. This table contains the outcome of the most recent execution of each branch. This predictor is based on the observation that most branches are decided the same way as on their previous execution. This is especially true for loop-closing branches.

This type of prediction does not always work well for subroutine returns, however, because a subroutine may be called from a number of places, and the return jump is not necessarily to the same address on two consecutive executions. Alpha takes this into account by having special hardware for predicting the target address for return-from-subroutine jumps. When the jump-to-subroutine instruction is executed, the return address is pushed on a four-entry prediction stack, so return addresses can be held for subroutines nested four deep. The stack is popped prior to returning from the subroutine, and the return address is used to prefetch instructions from the cache.

Branch processing. We are now ready to step through the pipeline flow for conditional branches; refer to Figure 7 for a PowerPC 601 example. Figure 7a assumes the branch was predicted not taken and the prediction was correct. Instructions are dispatched conditionally. The branch is resolved at time 2 after the compare executes, and the conditional instructions (B and C) are allowed to complete. The zero-cycle branch does not disrupt the instruction flow. The timing is similar for a correctly predicted taken branch, provided instructions are fetched immediately from the cache.

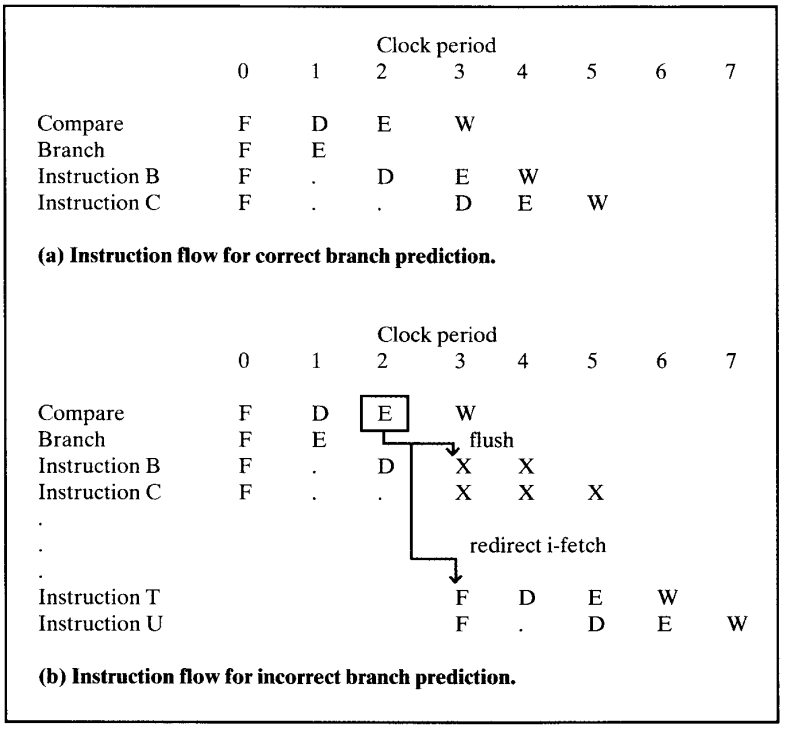


Figure 7. Timing for conditional branches in the PowerPC 601. X means instruction is flushed as a result of branch misprediction.

In Figure 7b the branch was predicted not taken, and the prediction was incorrect. Instructions B and C must be flushed, and instructions T and U are fetched from the branch target at time 3. There is a two-cycle bubble that sometimes can be reduced or entirely eliminated by moving the compare instruction earlier in the code sequence.

Regarding Alpha, Figure 8a is one (of several) cases where the branch prediction is correct; Figure 8b is a case where the prediction is wrong. The swap stage of the pipeline examines instructions in pairs. After the branch instruction is detected and predicted, it takes one clock cycle to compute the target address and begin fetching. This may lead to a one-cycle bubble in the pipeline. The pipeline is designed to allow "squashing" of this bubble. That is, if the instruction ahead of the bubble blocks and the instruction behind proceeds, the bubble is squashed between the two and eliminated. In some cases, when there is a simultaneous dispatch conflict, as in Figure 8a, the instruction preceding the branch must be split from it anyway. In this case, the branch instruction waits a cycle and naturally fills in the bubble (in effect, the branch fills its own bubble!). In other cases, if the pipeline stalls ahead of the branch, the bubble can be squashed by having an instruction behind the branch move up in the pipe (this happens later in Figure 11). If the bubble is squashed and the prediction is correct, the branch effectively becomes a zero-cycle branch.

Figure 8b shows the incorrect prediction case. The branch instruction has its registers read during issue stage. During the A stage, the register can be tested, and the correctness of the prediction can be determined. This is done quickly enough that if there is a misprediction, the instruction fetch stage can be notified while the branch is still in the A stage. Then, fetching the correct path can begin the next cycle. As a result, four stages of the pipeline must be flushed

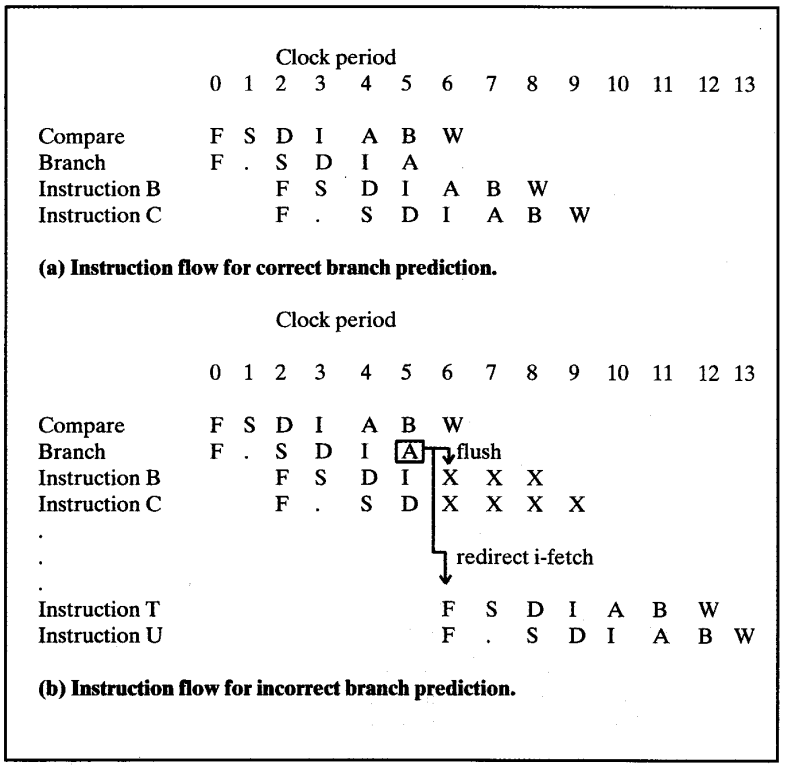


Figure 8. Timing for conditional branches in the Alpha 21064. X means instruction is flushed as a result of branch misprediction.

Table 3. Branch penalties. In the Alpha 21064, the penalty is generally independent of the distance between the compare and the branch. In the PowerPC 601, the penalty may be one cycle shorter (1 instead of 2, for example) when instructions are still in the buffer and the fetch cycle is saved.

Distance from Compare to Branch	Alpha 21064		PowerPC 601	
	Correct	Incorrect	Correct	Incorrect
1	0 to 1	4	0	2 or 1
2	0 to 1	4	0	1 or 0
>2	0 to 1	4	0	0

	Alpha 21064	PowerPC 601
Conditional branches	Dynamic prediction	Static prediction
Loop-closing branches	Dynamic prediction	Always zero-cycle
Subroutine returns	Stack prediction	Always zero-cycle

Table 4. Prediction method versus branch type.

when the prediction is found to be incorrect. For the jump-to-subroutine instruction, the penalty for a misprediction is five cycles.

For branches, the biggest architectural difference between the Alpha and PowerPC is that the Alpha uses general-purpose registers for testing and subroutine linkage. The PowerPC uses special-purpose registers held in the branch unit. This means the PowerPC can execute branch instructions in the branch unit, immediately after instructions are fetched. In fact, it looks back in the instruction buffer so that it can essentially execute, or at least predict, branches prior to dispatch. The Alpha 21064 implementation, on the other hand, must treat branch instructions like the other instructions. They decode in the D pipeline stage, read registers in I, and execute in A.

Table 3 compares the approximate branch penalties for integer conditional branches (far more common than floating-point branches). The penalties are expressed as a function of (1) the number of instructions separating the condition-determining instruction (for example, a compare) and the branch and (2) the correctness of the prediction. Instruction cache hits are assumed.

In summary, both the PowerPC 601 and Alpha 21064 dedicate special hardware to minimize the branch penalty. In the 601, the extra hardware is in the form of the PowerPC's special-purpose branch registers. These registers reduce the number of branches that must be predicted in the first place (see Table 4). In the 601, loop-closing branches that use the CTR register do not have to be predicted; in the Alpha, these are ordinary conditional branches, although loop-closing branches are easily predictable. In the PowerPC, return jumps can be executed immediately in the branch unit; there is no need for prediction. In Alpha, a subroutine return jump must read a general-purpose integer register, so these branches are predicted via the return stack.

In the 21064, the hardware resources are directed at improving branch prediction accuracy — history bits and the subroutine return stack. Not only does the 21064 have to predict more of the branches, the penalties (as measured in clock periods, not in absolute time) tend to be higher when it is wrong. As we have pointed out before, however, making performance comparisons based solely on clock periods does not give the 21064 proper credit. Its simple philosophy and

deeper pipelining leads to a clock that is about three times faster than the 601's.

Dispatch rules. Simultaneous dispatch rules are an important defining feature of superscalar architecture implementation.

The dispatch rules in the 601 are quite simple. The architecture has three units, integer (or fixed point), floating, and branch, and three instructions may issue simultaneously as long as all three go to different units. Integer operate instructions and all loads and stores go to the same pipeline (FXU), and only one instruction of this category may issue per clock cycle.

In the 21064, dispatch (in the 601 sense) occurs in the swap pipeline stage. Instructions *issue* two stages later. In the 21064, issue significantly affects dispatch because instructions must issue in their original program order, and dispatch (that is, the swap stage) helps enforce this order. A pair of instructions belonging to the same aligned doubleword (quadword in DEC parlance) can simultaneously issue. Consecutive instructions in different doublewords may not dual issue, and if two instructions in the same doubleword cannot simultaneously issue, the first in program sequence must issue first.

The 21064 implements separate integer and load/store pipelines, and several combinations of these instructions may be dual-issued, with the exception of integer operate/floating store and floating operate/integer store. These exceptions are due to a conflict in instruction paths, not shown in Figure 5. The load/store ports are shared with the branch unit. As a consequence, branches may not be simultaneously issued with any load or store instruction.

Tables 5 and 6 summarize the dispatch rules for both processors. In the PowerPC 601 table, two instructions can simultaneously issue if there is an X in the table's corresponding row/column. For three instructions, all three pairs must have X's. In the 21064 table, two instructions can simultaneously issue if there is an X in the table entry.

The ability of the 21064 to dual issue a load and an integer operate instruction is a definite strength with respect to the 601. Many applications (not to mention the operating system) use very little floating-point arithmetic, and the 21064 can execute these codes with high efficiency. For non-floating-point applications, the 601's floating-point unit sits idle while integer instructions dispatch at the rate of only one per clock cycle.

Table 5. PowerPC 601 instruction dispatch rules. Three mutually compatible instructions (marked with x) may issue simultaneously.

	Integer:			Floating:			Branch
	Load	Store	Operate	Load	Store	Operate	
Integer load						x	x
Integer store						x	x
Integer operate						x	x
Floating load						x	x
Floating store						x	x
Floating operate	x	x	x	x	x		x
Branch	x	x	x	x	x	x	

Table 6. Alpha 21064 instruction dispatch rules. Two compatible instructions (marked with x) may issue simultaneously. Integer branches depend on an integer register; floating branches depend on a floating register.

	Integer:			Floating:			Branch:	
	Load	Store	Operate	Load	Store	Operate	Integer	Floating
Integer load			x			x		
Integer store			x					
Integer operate	x	x		x		x	x	
Floating load			x			x		
Floating store						x		
Floating operate	x		x	x	x			x
Integer branch			x					
Floating branch						x		

Register files. It is interesting that quite different considerations in the two implementations led to register files that have almost the same number of ports (see Table 7).

A key aspect of the 21064 register file design is that integer register ports are judiciously allocated so that it can issue two integer instructions simultaneously. In the 21064, one write and two read ports are required to pipeline operate instructions, and there is a pair of read/write ports for load/store unit data. Branches share the load/store register ports. This brings the count up to 3R/2W for both integer and floating register files. A fourth integer read port is needed to get the address value for stores, and it is also used for load addresses. This fourth read port enables doing an integer store in parallel with an integer operate instruction. Finally, not allowing a register-plus-register addressing mode (see Figure 1e) eliminates the need for a fifth integer register read port.

Table 7. Register file ports.

	Alpha 21064	PowerPC 601
Integer registers:		
Read ports	4	3
Write ports	2	2
Floating registers:		
Read ports	3	3
Write ports	2	2

Looking at the 601, beginning again with one write and two read ports for operate instructions, a third integer read port is provided for single-cycle processing of the register-plus-register store indexed in-

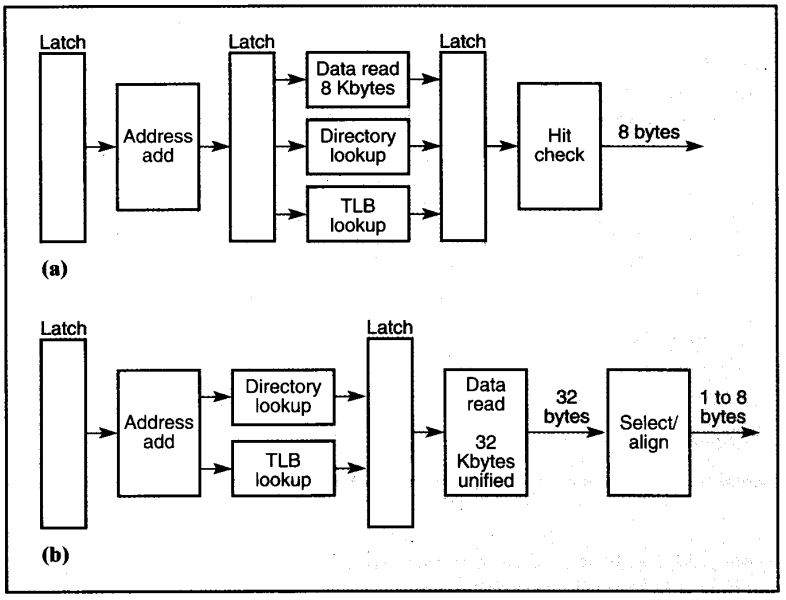


Figure 9. The Alpha 21064 (a) and PowerPC 601 (b) cache access paths.

```
                                        1 1 1 1 1 1 1 1 1 1
                          1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
lfsu    fp0 = y(r4 = r4 + 4)  F D E C W
fmuls   fp4 = fp0, fp1        F F . . DMAW
lfs     fp2 = x(r3,4)         F . DECW
fmadds  fp5 = fp4, fp2, fp3   F . S . . . . DMAW
stfsu   x(r3 = r3 + 4) = fp5  F . . . DE. . . . . C
bc      LOOP,CTR ≠ 0          F . S
lfsu    fp0 = y(r4 = r4 + 4)        F D E C W
fmuls   fp4 = fp0, fp1              F . . S . DMAW
lfs     fp2 = x(r3,4)               F . . . DECW
fmadds  fp5 = fp4,fp2,fp3           F . . . S . . . . DMAW
stfsu   x(r3 = r3 + 4) = fp5        F . . . . DE. . . . . C
bc      LOOP,CTR ≠ 0                F . . . S
```

Figure 10. PowerPC 601 pipelined processing example (two iterations of the loop). The arithmetic instructions are dispatched to the FPU pipeline. The load and store instructions are shown to go down the FXU pipe, but the store also decodes in the FPU pipe (not shown) right after the multiply-add instruction.

structions, which read three registers (two for the effective address, one for the result). The second integer write port allows writing the result of an operate instruction and data returned from the cache in the same clock cycle. The same consideration accounts for two write ports in the floating register file. The three floating-point read ports are needed by the combined floating multiply-add instruction.

Data caches. Yet another interesting contrast is the way the caches are implemented. Figure 9 shows the flow through the two data caches.

The 21064 uses separate instruction and data caches; the data cache is shown in Figure 9a. These are small (8 Kbyte), direct-mapped data caches designed for very fast access times. The address addition consumes one clock cycle. During the next clock cycle, the cache data and tag are read simultaneously. This is easily done with a direct-mapped cache where only one tag must be read and the data, if present, can only be in one place. Simultaneously, the translation lookaside buffer (TLB) is accessed for address translation information. During the next clock cycle, the TLB address translation completes, and the tag is compared with the upper address bits. A cache hit or miss is determined about half way through this clock cycle. The data is always delivered to the registers as an aligned 8-byte doubleword. Any alignment, byte selecting, etc. must be done with separate instructions.

This approach simplifies the cache interface, which does not have to include byte-level shift-and-mask logic in the cache access path. Clearly, the Alpha architects felt that the performance gained from shortening the cache load path was more important than the performance lost by using multiple instructions for byte loads and stores.

In the PowerPC 601, the unified data/instruction cache is much larger, 32 Kbytes, and is eight-way set associative, so it can be expected to have a higher hit rate than the 21064. In Figure 9b we can clearly see how much more "work" the 601 does in a clock cycle compared to the 21064. It does an address add and the cache directory/TLB lookup in the same cycle (see Weiss and Smith[10] for details). Then, during the next cycle, it accesses the 32-byte-wide data memory. During the same cycle, it selects and aligns the data field.

To summarize, the 601 implements the following features to improve performance: (1) a larger cache, (2) a higher associativity, and (3) automatic data alignment in hardware. The 21064 improves performance by not implementing such features, contributing to its very fast clock. When measured in nanoseconds, the two clock cycles in Figure 9b take longer than the three cycles in Figure 9a.

Example of pipeline flows. Figure 10 shows the 601 pipeline processing the code example given earlier in Figure 2b. The example is useful for emphasizing the basic differences in the two designs, but again, we caution against reaching any general conclusions regarding performance based on this one example. To simplify the presentation, we assume in all the examples below that instruction and data references hit in the cache.

Specifically in the 601, we also assume that the code is aligned with the cache sector, which allows fetching eight instructions. (Six instructions of the loop body are shown, the remaining two are irrelevant.)

The 601 predicts branches. However, as shown in the example, occasionally instructions cannot be fetched from the branch target address right after predicting the branch taken, because the unified cache is being accessed for data.

Figure 11 is the Alpha pipeline flow for the same example (Figure 2c). In-order issue and dual issue only for aligned instruction pairs is quite evident. The relatively long, six clock period, floating-point latency is also evident. We also observe that after the I stage, instructions are never blocked.

Starting with the first two instructions, which cannot dual issue because both are loads, we see the role the swap stage plays in ordering instructions for issue. The second instruction is held a cycle while the first moves ahead. The first dual issue occurs for the first addq-mult pair. Because the mult is the first instruction in the doubleword, the addq must wait, even though it has no dependencies holding it back. The sequence of dependent floating-point instructions paces instruction issue for most of the loop. Note that the floating store issues in anticipation of the floating point result. It does not wait six clock periods for the result. Rather, it waits only four, so it reaches the write stage just in time to have the floating point result bypassed to it.

Following the predicted branch at the end of the loop there is a bubble. Because other instructions in the pipeline are blocked, however, the bubble is squashed by the time the ldt following the branch is ready to issue.

Overall, the loop takes 16 clock periods per iteration in steady state. (The first ldt passes through the I stage at time 4, and during the second iteration it issues at time 20.) On the other hand, the PowerPC 601 takes six clock periods for single precision data (Figure 10) and eight clock periods for the double-precision version of the example. (Double precision takes longer because multiply and multiply-add instructions require two passes through the execute stages of the pipeline.) But don't forget, the 21064 is running its clock about three times faster.

The floating-point latencies are exposed and lead to delays when the 21064 executes this type of code. This is evident in the region of the timing diagram between clock periods 9 and 18. Also, because of in-order issue, the loops don't "telescope" like the 601's — in the 21064 there is very little overlap among consecutive loop iterations. Figure 12 illustrates the telescoping phenomenon. In the figure, each parallelogram illustrates the general shape of the pipeline flow for a single loop iteration.

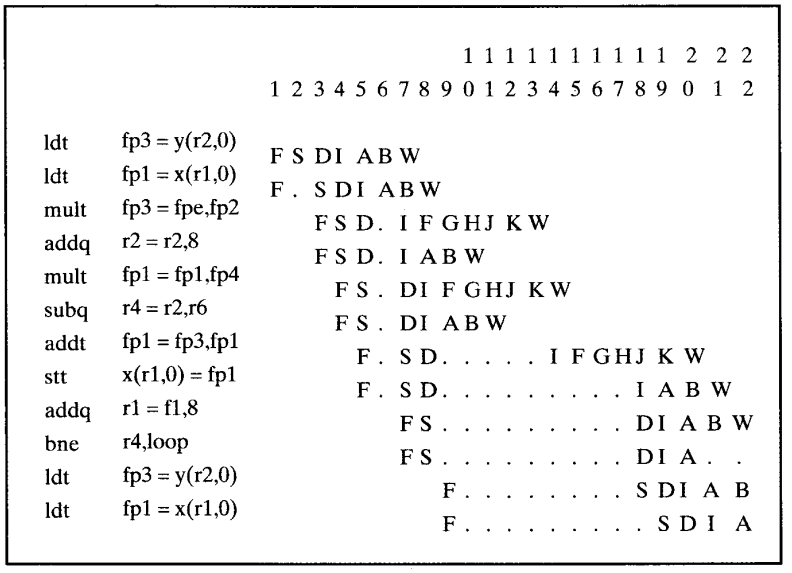


Figure 11. Alpha 21064 pipelined processing example.

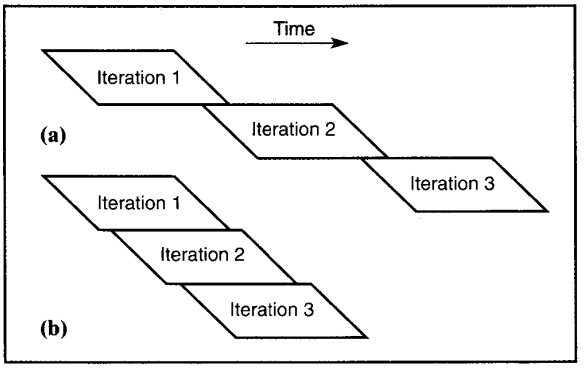


Figure 12. Comparison of loop overlap: (a) general pipeline flow with a 21064-like implementation; (b) "telescoped" pipeline flow with a 601-like implementation.

In an in-order issue processor like the 21064, there is little overlap between loop iterations, and branch prediction is a major contributor to the small amount of overlap that occurs.

In an implementation like the PowerPC 601, however, the out-of-order dispatch along with multiple buffers placed at key points allow the loop iterations to be compressed, like a folding telescope. Furthermore, with a branch processor as in the 601, branch prediction is not needed to achieve this effect.

Software techniques such as loop unrolling are likely to provide much better performance for a deeply pipelined implementation like the 21064. A loop is unrolled by combining multiple loop iterations into a single, larger loop. This not only gets rid of some loop closing branches but also provides more opportunities for the compiler to rearrange the instructions and increase overlap. The DEC compilers unroll loops.

When the example loop from Figure 2a is unrolled four times, the 21064 iteration takes 23 clock periods per unrolled iteration, or about six clock periods per original iteration — nearly a threefold improvement. Also, the performance advantage of dual issue is much more evident than with the rolled version (in our example, dual issue occurs in about a third of the clock periods).

Loop unrolling also improves the performance of the 601, but not nearly as

much. The unrolled loop (four iterations) takes 19 clock cycles, which corresponds to about five clock cycles per loop iteration (versus six clock periods in the rolled version). Now, with an unrolled loop where the deep pipelines can be used more efficiently, the 21064's clock period advantage translates into about a two-and-one-half-times performance advantage.

Imprecise interrupts. Both architectures support high-performance implementations with multiple pipelines. In such an implementation many instructions may be in the pipelines at any time, and it's difficult to precisely identify an interrupt-causing instruction without limiting the machine's performance.[10] Instead, an *imprecise* interrupt is signaled later, an arbitrary number of instructions after the interrupt-causing instruction.

A common problem occurs in the floating-point pipeline: It is usually longer than the integer pipe, so floating-point instructions finish late. When a floating-point interrupt is discovered, fixed-point instructions logically following the floating-point instruction may have already completed and modified a result register. This results in an imprecise state at the time of the interrupt. Allowing this to happen, however, leads to simpler implementations. Consequently, both Alpha and PowerPC allow imprecise floating point interrupts in their normal operating mode.

With imprecise interrupts, user software cannot "patch" an excepting floating-point result and continue. Imprecise interrupts can also make program debugging more difficult. Consequently, both architectures have provisions for precise operation, but at degraded performance. PowerPC does this in two ways. First, a bit in the machine state register may be set to make the machine enter a mode in which instructions execute serially and interrupts are precise. The second solution uses a compiler flag that inserts test code after each floating-point instruction that may cause an interrupt.

For implementing precise floating-point interrupts, Alpha has a trap barrier instruction that stalls instruction issuing until all prior instructions are executed without any interrupts. This instruction may be inserted after floating-point instructions to make floating-point interrupts precise. Of course, performance is degraded because the degree of instruction overlap is greatly reduced.

The PowerPC 601 and Alpha 21064 follow two remarkably different philosophies for achieving high performance implementations. The PowerPC architecture defines powerful instructions, such as floating-point multiply-add and update load/stores, that get more work done with fewer instructions. The Alpha architecture's simplicity, on the other hand, lends itself better to very high clock rate implementations. An Alpha processor can afford to execute more instructions if it can issue them faster. A typical example is load and store instructions that transfer only 32- or 64-bit quantities. As a result, Alpha implementations have a shorter cache load path, and the cache can be accessed with a faster clock.

The 601 uses independent pipelines, buffering, out-of-order dispatching, and it does a lot of computation in each pipe stage. Advanced branch handling and out-of-order dispatch lead to more efficient use of the pipes and more overlap among loop iterations. The 21064 has tightly coupled pipelines, little buffering, in-order issuing, and it does relatively less work in each pipe stage. Then again, it has a very fast clock. It also has fewer restrictions on multiple instruction dispatches, especially when doing integer code.

The 601 gains performance by design cleverness; the 21064 gains performance by design simplicity. This trade-off is a classic one, and the fact that both philosophies lead to viable processors is probably an indication that either choice is satisfactory as long as the implementation is done well. ■

Acknowledgment

We would like to thank the referees for their many suggestions and comments and Rick Kessler for a very helpful last-minute review.

References

1. D.A. Patterson, "Reduced Instruction Set Computers," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 8-21.
2. *PowerPC 601 RISC User's Manual*, Motorola, Pub. No. MPC601Um/Ad, 1992.
3. IBM Corp., *The PowerPC Architecture: A Specification for the New Family of RISC Processors*, Morgan Kaufmann, San Francisco, 1994.
4. C.R. Moore, "The PowerPC 601 Microprocessor," *Digest Compcon 93*, IEEE CS Press, Los Alamitos, Calif., Order No. 3400, 1993, pp. 109-116.
5. G. Paap and E. Silha, PowerPC: A Performance Architecture, *Digest Compcon 93*, IEEE CS Press, Los Alamitos, Calif., Order No. 3400, 1993, pp. 104-108.
6. *Alpha Architecture Handbook*, Digital Equipment Corporation, Maynard, Mass., 1992.
7. D. Dobberpuh et al., "A 200-MHz 64-Bit Dual-Issue CMOS Microprocessor," *IEEE J. Solid State Circuits*, Vol. 27, No. 11, Nov. 1992, pp. 1555-1567.
8. R.L. Sites, "RISC Enters a New Generation," *Byte*, Aug. 1992, pp. 141-148.
9. E. McLellan, "The Alpha AXP Architecture and 21065 Processor," *IEEE Micro*, Vol. 13, No. 3, June 1993, pp. 36-47.
10. S. Weiss and J.E. Smith, *Power and PowerPC: Principles, Architecture, and Implementation*, Morgan Kaufmann, San Francisco, 1994.

James E. Smith is with Cray Research in Chippewa Falls, Wisconsin, where he works on the development and analysis of future supercomputer architectures. Earlier, while on the faculty at the University of Wisconsin-Madison, he took two leaves of absence to work in industry on computer development projects. At the Astronautics Corporation Technology Center in Madison, he was principal architect for the ZS-1, a scientific computer employing a dynamically scheduled, superscalar processor architecture. At Control Data Corporation in Arden Hills, Minnesota, he participated in the design of the Cyber 180/990. In the early 1980s, he conducted research in high-performance processor implementations, including what are now referred to as "superscalar" architectures.

Smith is a coauthor of *Power and PowerPC: Principles, Architecture, Implementation* (Morgan Kaufmann, 1994) and is a member of IEEE and ACM.

Shlomo Weiss is a faculty member of the Department of Electrical Engineering/Systems at Tel Aviv University. He teaches computer architecture and leads a group of students working on the analysis of high-performance processor implementation alternatives and on performance evaluation of new systems. Before that, he was with Daisy Systems Corporation, Mountain View, California, and with the Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, where he worked on database management systems for VLSI CAD applications.

Weiss's current research interests are in computer architecture, with focus on the design of high-performance processors and memory systems. He is a coauthor of *Power and PowerPC: Principles, Architecture, Implementation* (Morgan Kaufmann, 1994) and a member of IEEE.

Readers can contact Smith at Cray Research, Inc., 900 Lowater Rd., Chippewa Falls, WI 54729.