DEC's Alpha Architecture Premiers

64-Bit RISC Architecture Streamlined for Speed

By Brian Case

It has been just about ten years since the ideas of RISC first started getting widespread attention. Since then, many RISC and RISC-like architectures have been proposed and almost as many have been introduced. A few of the architectures are now being offered in second-generation implementations using advanced hardware organizations. Most of the architectures have found niches, and their vendors are busy solidifying positions in the markets where they have found success. Competition has even claimed its first casualties. In short, the market is exhibiting the first signs of a phase of consolidation.

It is into this climate that DEC has chosen to introduce its new RISC architecture, called Alpha. With so many existing and "standardized" RISCs already crowding the market, it seems strange that DEC decided to introduce yet another architecture. Accompanying each new RISC are claims of architectural superiority and claims that the superior features will lead to superior implementations. For a while, the claims seem to be substantiated, but without exception, a new implementation of an existing architecture has always supplanted the front runner. In fact, the 486 and forthcoming 586 give some observers reason to doubt the significance of the difference between RISC and CISC.

Carrying on the tradition, Alpha incorporates subtle architectural features that DEC claims have a substantial effect on the effectiveness of implementations.

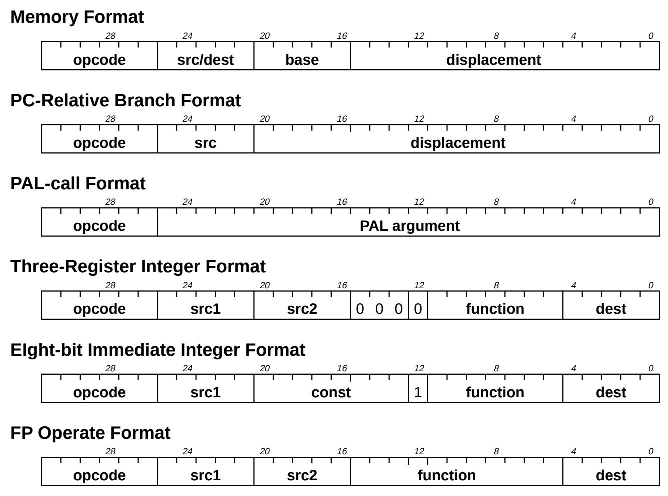

Figure 1. Alpha instruction formats.

Alpha is designed to facilitate multiple-issue (superscalar) implementations and multiprocessor system organizations so that the architecture will be competitive for at least 25 years. (The VAX architecture began to run out of gas after less than 10 years.) While the Alpha architecture may or may not have significant advantages for high-performance designs, it is an extremely clean architecture.

Architecture Overview

Although "alpha" is usually an adjective that means "first version," DEC's architecture is complete and fully specified with a sheer volume of documentation that only a large computer vendor can produce.

The most significant new aspect of Alpha is that it is strictly a 64-bit architecture. Unlike the MIPS-III architecture as implemented in the R4000, Alpha has no 32-bit "compatibility mode" and no mode bits; addresses are always 64 bits long. Like the R4000, however, Alpha implementations will not permit a full-64-bit user-mode virtual address space at first. Planned Alpha implementations will have 43-, 47-, 51-, and 55-bit virtual address lengths. The desire for a modeless 64-bit address space reflects the large-system requirements at DEC.

In nearly every respect, Alpha is a traditional RISC architecture. It has 32 general-purpose integer registers and 32 floating-point registers. R31 in each register file is the "zero" register that reads as zero and discards results written there. All registers are 64 bits long, and all instructions are encoded in 32 bits.

Alpha has floating-point operations not only for IEEE single and double precision but also for a subset of VAX FP formats. VAX F (single) and G (double) formats have full support while VAX D (longer double) is minimally supported and suffers a loss of 3 bits of precision.

Like the IBM POWER architecture, which is also a relatively new RISC architecture, Alpha does not have delayed branches. Since delayed branches add some complexity to superscalar implementations, it makes sense to leave them out of a new architecture.

Conventional RISC and CISC architectures complicate processor designs, especially superscalar implementations, with the requirement that exceptions be precise. An exception is precise when it is reported for the causing instruction and when no following instructions are allowed to even partially execute. Alpha allows exceptions to be reported in an imprecise way. This allows instruction-issue logic to dispatch instructions as soon as the necessary operands are available. In the majority of cases, an imprecise exception is as useful as a precise one since the program will be aborted anyway. When precise exception reporting is desired for an instruction in Alpha programs, such as during software debugging, the instruction can be surrounded with TRAPB (trap barrier) instructions.

To aid compatibility with the VAX software base, Alpha is a little-endian machine. The endianness only affects the extract, insert, and mask instructions that are used to operate on data smaller than 64 bits.

Instruction Set

Alpha has only four basic instruction formats with three variations for computation instructions, as shown in Figure 1. PC-relative branches have an offset of 21 bits, yielding an effective displacement range of 23 bits (since the two LSBs of instruction addresses are always zero). Integer computation instructions can specify either three registers or two registers and a zero-extended, eight-bit constant. The PAL-call format is used to invoke Privileged Architecture Library routines, which implement system functions and complex, atomic operations required for backward compatibility with VAX/VMS. Privileged architecture libraries will also be written to facilitate the implementation of various other operating systems such as Windows NT and OSF/1 UNIX.

The instruction set, shown in Figure 2, consists of 160 instructions, a number that is comparable to other RISCs aimed at general-purpose computing. The instruction set is fairly conventional, with the exception of the extensive set of byte, word (16-bit data), and

Load/Store, Byte Manipulation

LDA	Load address
LDAH	Load address high
LDL	Load sign-extended longword
LDQ	Load quadword
LDQ_U	Load unaligned quadword
LDL_L	Load sign-extended longword, locked
LDQ_L	Load quadword, locked
STL_C	Store longword, conditional
STQ_C	Store quadword, conditional
STL	Store longword
STQ	Store quadword
STQ_U	Store unaligned quadword
EXTBL	Extract byte low
EXTWL	Extract word low
EXTLL	Extract longword low
EXTQL	Extract quadword low
EXTWH	Extract word high
EXTLH	Extract longword high
EXTQH	Extract quadword high
INSBL	Insert byte low
INSWL	Insert word low
INSLL	Insert longword low
INSQL	Insert quadword low
INSWH	Insert word high
INSLH	Insert longword high
INSQH	Insert quadword high
MSKBL	Mask byte low
MSKWL	Mask word low
MSKLL	Mask longword low
MSKQL	Mask quadword low
MSKWH	Mask word high
MSKLH	Mask longword high
MSKQH	Mask quadword high

FP Load/Store

LDF	Load F format (VAX single)
LDG	Load G format (VAX double)
LDS	Load S format (IEEE single)
LDT	Load T format (IEEE double)
STF	Store F format (VAX single)
STG	Store G format (VAX double)
STS	Store S format (IEEE single)
STT	Store T format (IEEE double)

Address/Constant

LDA	Load address
LDAH	Load address high

Integer Computation & Cond. Move

ADDL	Add longword
S4ADDL	Add longword, scale by 4
S8ADDL	Add longword, scale by 8
ADDQ	Add quadword
S4ADDQ	Add quadword, scale by 4
S8ADDQ	Add quadword, scale by 8
CMPEQ	Compare signed quadword ==
CMPLT	Compare signed quadword <
CMPLE	Compare signed quadword <=
CMPULT	Compare unsigned quadword <
CMPULE	Compare unsigned quadword <=
MULL	Multiply longword
MULQ	Multiply quadword
UMULH	Multiply quadword high, unsigned
SUBL	Subtract longword
S4SUBL	Subtract longword, scale by 4
S8SUBL	Subtract longword, scale by 8
SUBQ	Subtract quadword
S4SUBQ	Subtract quadword, scale by 4
S8SUBQ	Subtract quadword, scale by 8
AND	AND logical
BIS	OR logical
XOR	XOR logical
BIC	AND-NOT logical
ORNOT	OR-NOT logical
EQV	XOR-NOT logical
SLL	Shift left, logical
SRL	Shift right, logical
SRA	Shift right, arithmetic
CMOVEQ	Conditional move if register == 0
CMOVNE	Conditional move if register != 0
CMOVLT	Conditional move if register < 0
CMOVLE	Conditional move if register <= 0
CMOVGT	Conditional move if register > 0
CMOVGE	Conditional move if register >= 0
CMOVLBC	Conditional move if register low bit clear
CMOVLBS	Conditional move if register low bit set
CMPBGE	Compare bytes, unsigned
ZAP	Clear selected bytes
ZAPNOT	Clear unselected bytes

Integer Branch

BEQ	Branch if register == 0
BNE	Branch if register != 0
BLT	Branch if register < 0
BLE	Branch if register <= 0
BGT	Branch if register > 0
BGE	Branch if register >= 0
BLBC	Branch if low bit clear
BLBS	Branch if low bit set
BR	Branch
BSR	Branch to subroutine
JMP	Jump
JSR	Jump to subroutine
RET	Return from subroutine
JSR_COROUTINE	Jump to subroutine return

FP Branch

FBEQ	FP branch if == 0
FBNE	FP branch if != 0
FBLT	FP branch if < 0
FBLE	FP branch if <= 0
FBGT	FP branch if > 0
FBGE	FP branch if >= 0

FP Computation & Conditional Move

CPYS	Copy sign
CPYSN	Copy sign, negate
CPYSE	Copy sign and exponent
CVTQL	Convert quadword to longword
CVTLQ	Convert longword to quadword
FCMOVEQ	FP conditional move if register == 0
FCMOVNE	FP conditional move if register != 0
FCMOVLT	FP conditional move if register < 0
FCMOVLE	FP conditional move if register <= 0
FCMOVGT	FP conditional move if register > 0
FCMOVGE	FP conditional move if register >= 0
MF_FPCR	Move from FP control register
MT_FPCR	Move to FP control register
ADDF	Add F format (VAX single)
ADDG	Add G format (VAX double)
ADDS	Add S format (IEEE single)
ADDT	Add T format (IEEE double)
CMPGEQ	Compare G format == (VAXdouble)
CMPGLT	Compare G format < (VAX double)
CMPGLE	Compare G format <= (VAX double)
CMPTEQ	Compare T format == (IEEE double)
CMPTLT	Compare T format < (IEEE double)
CMPTLE	Compare T format <= (IEEE double)
CMPTUN	Compare T format unordered (IEEE double)
CVTGQ	Convert G format to quadwored (VAX double)
CVTQF	Convert quadword to F format (VAX single)
CVTQG	Convert quadword to G format (VAX double)
CVTDG	Convert D format to G format (VAX double/double)
CVTGD	Convert G format to D format (VAX double/double)
CVTGF	Convert G format to F format (VAX double/single)
CVTTQ	Convert T format to quadword (IEEE double)
CVTQS	Convert quadword to S format (IEEE single)
CVTQT	Convert quadword to T format (IEEE double)
CVTTS	Convert T format to S format (IEEE double/single)
DIVF	Divide F format (VAX single)
DIVG	Divide G format (VAX double)
DIVS	Divide S format (IEEE single)
DIVT	Divide T format (IEEE double)
MULF	Multiply F format (VAX single)
MULG	Multiply G format (VAX double)
MULS	Multiply S format (IEEE single)
MULT	Multiply T format (IEEE double)
SUBF	Subtract F format (VAX single)
SUBG	Subtract G format (VAX double)
SUBS	Subtract S format (IEEE single)
SUBT	Subtract T format (IEEE double)

System

CALL_PAL	Call privileged architecture library
TRAPB	Trap barrier (precise exception)
FETCH	Prefetch (cache) data hint
FETCH_M	Prefetch (cache) data, modify hint
MB	Memory barrier (serialize)
RPCC	Read process cycle counter
RC	Read and clear
RS	Read and set
PALRES0	PAL reserved opcode 0
PALRES1	PAL reserved opcode 1
PALRES2	PAL reserved opcode 2
PALRES3	PAL reserved opcode 3
PALRES4	PAL reserved opcode 4

Figure 2. Alpha instruction set summary.

longword (32-bit data) manipulation instructions shown under the Load/Store heading. (Note that DEC uses “word” to mean a 16-bit quantity, “longword” for 32 bits, and “quadword” for 64 bits, and we’ll follow their usage in this article.)

Alpha has a longword-oriented load/store architecture, so unlike every other significant RISC architecture, it has no loads or store for bytes or 16-bit words. As shown in Figure 2, loads and stores are available only for longwords and quadwords. This means that to load a byte or word, the longword or quadword containing the data must first be loaded; the desired byte or word is then extracted from the loaded data. To store a byte or word, the longword or quadword containing the data is first loaded, the byte or word inserted into the loaded data, and then the modified longword or quadword is stored to memory.

Alpha doesn’t have load-word left or right like MIPS, because these instructions require byte-write capability into the register file. This is another case of DEC eliminating strangeness that would have helped only one or two instructions.

This longword orientation imposes quite a penalty in instruction count for Alpha programs that manipulate 8- or 16-bit data. The only other well-known RISC architecture to implement word-orientation—which was proposed as advantageous in a RISC-oriented paper from Stanford in 1982—is the 29000, but shortly after its introduction, AMD augmented the architecture with regular byte and 16-bit loads and stores to correct the oversight. The Stanford MIPS architecture also lacked 8- and 16-bit memory operations, but these functions were added for the commercial MIPS architecture. It is interesting to note that even the original 29000 had slightly better support for storing byte and 16-bit data than does Alpha: the respective insert instructions placed the data to be stored directly into the proper place in the 32-bit memory word. Alpha requires a three-instruction sequence to perform the same job: the data to be stored must be shifted up with an “insert” instruction, the memory word must be masked to create a “hole” for the data to be stored, and finally an OR (BIS) instruction is required to combine the two. Figure 3 shows a byte store in Alpha code.

```
LDL        R1,0(R9)
INSBL      R5,#1,R3
MSKBL      R1,#1,R1
BIS        R3,R1,R1
STL        R1,3(R9)
```

Figure 3. A byte store in Alpha code. The address in R9 is known by the compiler to be a longword address, and so the offset of the desired byte is encoded in the INSBL and MSKBL instructions (#1). The sequence loads the longword (LDL), shifts the byte into the right spot (INSBL), makes a hole in the longword for the byte (MSKBL), ORs the two together (BIS), and then stores the longword to memory.

The one strength of this approach is in handling data that is not naturally aligned (e.g., a 32-bit word not on a 32-bit boundary). The LDQ_U and STQ_U instructions cause the low three address bits to be ignored on a load or store. By using one or two LDQ_U instructions and an appropriate sequence of extract-low (EXTxL) and extract-high (EXTxH) instructions, a word, longword, or quadword can be loaded regardless of its alignment. A similar algorithm works for stores. While the instruction sequences involved are somewhat long, they are still very much shorter than what would be required with other RISC instruction sets.

Handling misaligned data is important for Alpha since it will have to run programs that operate on old VAX data structures. DEC is implementing a binary compiler and interpreter suite to help VAX customers make the transition from VAX-family to Alpha-family computers, but even old code that is recompiled may have to access misaligned structures. Given that handling misaligned data is necessary in Alpha, a software implementation eliminates some major complexity from the critical path between the processor and the cache/main-memory system.

A more important issue for DEC is the impact on write-back caches. It is a DEC policy that ECC be used wherever non-transient copies of data are held. To implement a byte-writable write-back cache with ECC requires check/correct bits with every byte or a read-modify-write state machine. Both are costly and can be avoided by leaving out byte and 16-bit loads and stores.

Alpha has the same set of memory addressing modes as MIPS: register, register-plus-signed-16-bit-offset, and signed-16-bit-absolute. Load-address (LDA) and load-address-high (LDAH) can be used to form a 32-bit constant in two instructions.

Alpha’s integer computation instructions are fairly traditional except for a few unusual operations. Add, subtract, and multiply are provided in both 64-bit (Q suffix) and 32-bit (L suffix) versions. The 32-bit versions treat overflow and carry differently, and they also use only the low 32 bits of their source operands and sign-extend the 32-bit result to 64 bits. The shift-and-add (SxADDy) and shift-and-subtract (SxSUBy) instructions are used in subscript addressing mode calculations for arrays of longwords and quadwords. They are also useful for computing integer multiplies when one operand is a small constant.

The UMULH instruction, which produces the upper 64-bits of a full 64 $\times$ 64-bit multiplication, is used to implement integer division quickly. Integer division by a constant is performed by multiplying by the reciprocal, which is easily provided by a compiler. Integer division by a variable is performed by a routine that does a lookup in a reciprocal table for numbers less than 1000 and performs an iterative algorithm for larger numbers. The algorithm needs a maximum of nine multiplies with an average of five. This performance compares favorably with dedicated hardware.

The conditional-move instructions are unique to Alpha. They copy a source register to a destination register if a second source register satisfies a condition. The conditions are either signed comparisons with zero or the state of the least-significant bit. These instructions implement a fairly frequent case without requiring a pipeline-disrupting branch. Where the IBM POWER architecture includes a couple of instructions specifically targeted at maximum and minimum functions (the difference-or-zero instructions), the Alpha conditional-moves implement these functions with equal efficiency and much more generality. (The POWER instructions that perform these functions have been deleted in the PowerPC architecture.)

The compare-bytes (CMPBGE) and "byte-zero" (ZAP, ZAPNOT) instructions work together. CMPBGE does eight pair-wise byte comparisons for greater-or-equal and stores an eight-bit vector reflecting the outcomes in the low byte of its destination operand. The ZAP instructions then clear selected bytes based on this eight-bit vector. Another use of CMPBGE is in speeding up C-language string-handling library routines, which has an especially beneficial effect on Dhrystone performance.

Alpha has a very clean conditional-execution architecture. The same set of comparisons against zero is used throughout the integer and FP conditional moves and the integer and FP conditional branches. One of the most common compare-and-branch uses in programs is comparing an integer value or data structure address pointer to zero. For example, in linked-list code, it is necessary to check a pointer against "NULL" before dereferencing it. Like MIPS, Alpha can perform this compare-and-branch function in one instruction.

Another very common operation is comparing two registers for equal or not-equal and then branching, and the MIPS architecture has instructions for these common cases. Despite the fact that these compare-and-branch instructions are extremely useful, Alpha left them out because they create a very critical speed path.

Alpha has a straightforward FP instruction set that provides the basic arithmetic operations and format conversions. Unlike other architectures, there is no square-root instruction. It is surprising to see that there are no composite multiply-add and multiply-subtract instructions, given that they have demonstrated their value in graphics and benchmark code for the IBM POWER, HP PA-RISC, and Intel 860 machines. DEC claims that these instructions would not be fully utilized in Alpha implementations because the functional units are already able to absorb all the operand bandwidth available.

As with the integer computation instruction subset, FP conditional-move instructions are available. A unique feature of the instruction set is that the FP compares generate data values in the FP registers. This allows compound conditional expressions, such as "IF (x < y AND y < z)," to be executed completely within the FP unit. In this example, the two less-than comparisons are performed with FP compare instructions, the logical AND is performed with a floating-point multiply, and the branch of the "IF" is performed with an FP branch. Similarly, a logical OR construct can be implemented with an FP add instruction.

Alpha system instructions are few since sensitive operations are performed by short routines that are invoked with a CALL_PAL instruction. Privileged Architecture Library routines have the advantage that they are entered and exited with little overhead, they execute with interrupts disabled, and they operate with user-mode instead of supervisor-mode address translation. Other RISC architectures have similar facilities, but CALL_PAL instructions have a large, 26-bit argument field encoded directly in the instruction.

DEC's system software uses PAL routines to implement functions such as TLB miss handling and interrupt acknowledge. By defining a standard software interface for these functions at the PAL call level, future implementations can use different hardware structures but provide an identical interface, since the PAL code handles the low-level details.

The TRAPB instruction aids a program in implementing precise exceptions when needed in, say, a debugging environment. To make multiprocessor shared memory systems easier to build, Alpha specifies a weakly ordered model of memory operations. Weak ordering allows loads and stores to complete in the order most convenient for the hardware. The memory-barrier (MB) instruction allows a program to force loads and stores to be performed in a deterministic order when necessary. This instruction guarantees that all preceding loads and stores will be completed before any subsequent loads or stores are allowed to access memory.

FETCH and FETCH_M supply cache pre-fetch hints to implementations that can make use of them. These instructions specify an aligned 512-byte block of data; an implementation can choose to ignore the instruction, to move all or part of the block to a faster part of the storage hierarchy, or to move an even larger surrounding area. FETCH_M performs the same function, except it indicates the intent to modify some or all of the data.

RPCC accesses the hardware cycle counter so that low-level performance analysis can be very accurate. RC and RS help implement some VAX-compatible instruction semantics, but they are not guaranteed to be present in all future implementations. The PALRESx instructions provide the PAL routines with access to low-level hardware resources such as the processor status register and address-translation hardware.

Architecture Evaluation

One of the main goals of Alpha is to make sure that its implementations are as fast as technology will allow. This is nothing new, since RISC was invented to exploit hardware, but Alpha does indeed go a couple of steps beyond other RISCs in catering to fast implementations. The lack of byte and 16-bit loads and stores helps to minimize the logic between the processor and first-level cache and simplifies ECC and sequencing in write-back caches. Superscalar implementations should be easier to design because there are no delayed branches, no shared resources, and no precise exceptions. Condition codes (as in IBM POWER) and multiply-quotient registers (as in MIPS) are examples of shared resources that can complicate superscalar implementations.

Still, Alpha has its drawbacks. The tremendous overhead paid for loads and stores of small data is the most glaring. One of the golden rules of architecture and implementation is that anything that makes the basic instruction cycle faster is probably worthwhile. Nonetheless, the improvement in basic instruction timing allowed by eliminating byte and 16-bit loads and stores must be quite significant to overcome the cost of the long instruction sequences required. DEC says that fewer than 8% of VAX/VMS memory references deal with quantities smaller than 32 bits, and that many of these are string searches or copies that can be performed with longword or quadword accesses. For byte and word I/O operations, DEC's plan is to shift I/O addresses left four bits and use the four lower address bits as byte enables.

Alpha does indeed facilitate fast, superscalar implementations better than other RISCs, but it is still possible to design and build superscalar implementations of other RISCs in a reasonable amount of time and chip area. The original RISC characteristics (load/store architecture, three-address operations, single-sized instructions, simple formats, instructions that fit in a uniform pipeline, etc.) have a first-order effect on implementation and software quality. While the advantages of the Alpha architecture are real, they are of only second-order significance.

What is more important for DEC is the architectural support for misaligned data and VAX FP operands—features that are not found in other RISCs. DEC's motivations for creating a new architecture are best understood in light of its need to provide an efficient VAX replacement. For applications where VAX compatibility is not a concern, it does not appear that Alpha's advantages are great enough to overcome the advantages of the software base and momentum of existing architectures. ♦

Alpha Implementation

Continued from page 9

plans to stop MIPS-based development—R4000 system development is said to be well underway—but they also point out that Alpha products will have to be competitive. This seems to imply that customers will have a choice of MIPS- and Alpha-based workstations in similar price ranges. DEC's aggressive promotion of Alpha could considerably weaken the marketability of DEC's MIPS-based workstations, despite assurances of a smooth migration path.

Despite Alpha's impressive performance and architectural edge, each of the leading RISC architectures has important advantages. SPARC and MIPS both have the advantage of a wide range of implementations now available or expected soon, and a similar range of implementations is promised for PowerPC. It remains to be seen how quickly DEC will be able to develop a range of implementations and how aggressively it will pursue the low end of the market, which is where the volume is.

For the MIPS architecture, in comparison, there are low-end chips such as IDT's R3081 and Performance Semiconductor's PIPER that will compete effectively with the 486; today's R4000, which is a good mid-range device; future R4000s with higher clock rates and larger caches; and future R4000s aimed at lower-cost systems.

PowerPC remains the only RISC architecture with a clear path to the personal productivity computer market, thanks to Apple's commitment to migrate its Macintosh line. MIPS may have the second-best chance because of Windows NT, but its ability to compete with the x86 remains in question. SPARC continues to do well in the technical workstation market because of its software edge, and Sun has begun penetrating some commercial applications where a wide variety of personal productivity software is not required. Alpha seems unlikely to blunt any of these advantages.

In the workstation arena, HP is likely to be the most threatened by Alpha. HP has taken a very slow, cautious approach to licensing its architecture, and it has been unwilling to take on the support burden of selling its chips on the merchant market. DEC's more aggressive approach could create considerable trouble for HP, which remains outside the mainstream. HP expects its latest implementation, the 7100, to run at 100 MHz with a performance level of 120 SPECmarks—about the same as the 21064 at 150 MHz.

From a business perspective, it seems a shame that DEC did not apply its chip design and fabrication capabilities to the MIPS architecture, perhaps with extensions to better support VAX software. That they did not do so is perhaps evidence of the difficulty a company such as DEC has with basing its business on an architecture from an outside supplier. ♦