Alpha AXP Architecture

By Richard L. Sites

Abstract

The Alpha AXP 64-bit computer architecture is designed for high performance and longevity. Because of the focus on multiple instruction issue, the architecture does not contain facilities such as branch delay slots, byte writes, and precise arithmetic exceptions. Because of the focus on multiple processors, the architecture does contain a careful shared-memory model, atomic-update primitive instructions, and relaxed read/write ordering. The first implementation of the Alpha AXP architecture is the world's fastest single-chip microprocessor. The DECchip 21064 runs multiple operating systems and runs native-compiled programs that were translated from the VAX and MIPS architectures.

Thus in all these cases the Romans did what all wise princes ought to do; namely, not only to look to all present troubles, but also to those in the future, against which they provided with the utmost prudence. — Niccolo Machiavelli, The Prince

Historical Context

The Alpha AXP architecture grew out of a small task force chartered in 1988 to explore ways to preserve the VAX VMS customer base through the 1990s. This group eventually came to the conclusion that a new reduced instruction set computer (RISC) architecture would be needed before the turn of the century, primarily because 32-bit architectures will run out of address bits. Once we made the decision to pursue a new architecture, we shaped it to do much more than just preserve the VMS customer base.

This paper discusses the architecture from a number of points of view. It begins by making the distinction between architecture and implementation. The paper then states the overriding architectural goals and discusses a number of key architectural decisions that were derived directly from these goals. The key decisions distinguish the Alpha AXP architecture from other architectures. The remaining sections of the paper discuss the architecture in more detail, from data and instruction formats through the detailed instruction set. The paper concludes with a discussion of the designed-in future growth of the architecture. An Appendix explains some of the key technical terms used in this paper. These terms are highlighted with an asterisk in the text.

Architecture Distinct from Implementations

From the beginning of the Alpha AXP design, we distinguished the architecture from the implementations, following the distinction made by the IBM System/360 architects:

> Computer architecture is defined as the attributes and behavior of a computer as seen by a machine-language programmer. This definition includes the instruction set, instruction formats, operation codes, addressing modes, and all registers and memory locations that may be directly manipulated by a machine-language programmer.
>
> Implementation is defined as the actual hardware structure, logic design, and data-path organization of a particular embodiment of the architecture.[1]

Thus, the architecture is a document that describes the behavior of all possible implementations; an implementation is typically a single computer chip.[2] The architecture and software written to the architecture are intended to last several decades, while individual implementations will have much shorter lifetimes. The architecture must therefore carefully describe the behavior that a machine-language programmer sees, but must not describe the means by which a particular implementation achieves that behavior.

A similar approach has been used with much success in specifying the PDP–11 and VAX families of computers. An alternate approach is to design and build a fast RISC* chip, then wait to see if it is successful in the marketplace. If so, successive implementations are often forced to reproduce accidents of the initial design, or to introduce slight software incompatibilities. This approach works, but with varying success.

Architectural Goals

When we started the detailed design of the Alpha AXP architecture, we had a short list of goals:

1. High performance
2. Longevity
3. Capability to run both VMS and UNIX operating systems
4. Easy migration from VAX and MIPS architectures

These goals directly influenced our key decisions in designing the architecture.

In considering performance and longevity, we set a 15- to 25-year design horizon and tried to avoid any design elements that we thought could become limitations during this time. In current architectures, a primary limitation is the 32-bit memory address. Thus we adopted a full 64-bit architecture, with a minimal number of 32-bit operations for backward compatibility.

We also considered how implementation performance should scale over 25 years. During the past 25 years, computers have become about 1,000 times faster. Therefore we focused our design decisions on allowing Alpha AXP system implementations to become 1,000 times faster over the coming 25 years. In our projections of future performance, we reasoned that raw clock rates would improve by a factor of 10 over that time, and that other design dimensions would have to provide two more factors of 10.

If the clock cannot be made faster, then more work must be done per clock tick. We therefore designed the Alpha AXP architecture to encourage multiple instruction issue* implementations that will eventually sustain about ten new instructions starting every clock cycle. This aggressive technique of starting multiple instructions distinguishes the Alpha AXP architecture from many other RISC architectures.

The remaining factor of 10 will come from multiple processors. A single system will contain perhaps ten processors and share memory. We therefore designed a multiprocessor memory model and matching instructions from the beginning. This early accommodation for multiple processors also distinguishes the Alpha AXP architecture from many other RISC architectures, which try to add the proper primitives later.

To run the OpenVMS AXP and the DEC OSF/1 AXP—and now the Microsoft Windows NT—operating systems, we adopted an idea from a previous Digital RISC design called PRISM.[3] We placed the underpinnings for interrupt delivery and return, exceptions, context switching, memory management, and error handling in a set of privileged software subroutines called PALcode. These subroutines have controlled entry points, run with interrupts turned off, and have access to real hardware (implementation) registers. By including different sets of PALcode for different operating systems, neither the hardware nor the operating system is burdened with a bad interface match, and the architecture itself is not biased toward a particular computing style.

To run existing VAX and MIPS binary images, we adopted the idea of binary translation,* as described in a companion paper.[4,5,6] The combination of PALcode and binary translation gave us the luxury of designing a new architecture. Other than the fundamental integer and floating-point data types, there are no specific VAX or MIPS features carried directly into the Alpha AXP instruction-set architecture for compatibility reasons.

Key Design Decisions

This section presents the design decisions that distinguish the Alpha AXP architecture from others.

RISC

The Alpha AXP architecture is a traditional RISC load/store architecture. All data is moved between registers and memory without computation, and all computation is done between values in registers. Little-endian byte addressing and both VAX and IEEE floating-point operations* are carried over from the VAX and MIPS architectures.[7] We assumed that most implementations would pipeline instructions, i.e., they would start execution of a second, third, etc. instruction before the execution of a first instruction completes. We assumed that the implementation latency of many operations would be important. Latency is the number of cycles a program must wait to use the result of a preceding instruction. We assumed that the vast majority of memory operands would be aligned. An aligned operand of size 2***N* bytes* has an address with *N* low-order zeros. Other memory operands are termed unaligned.

Full 64-bit Design

The Alpha AXP architecture uses a linear* 64-bit virtual address space. Registers, addresses, integers, floating-point numbers, and character strings are all operated on as full 64-bit quantities. There are no segmented addresses.*

Register File

In choosing the register file design, we considered both a single combined register file and split integer and floating-point register files. We chose a split register file to support aggressive multiple issue. A combined file is somewhat more flexible, especially for programs that are heavily skewed toward integer-only or floating-point-only computation. A combined file also makes it easier to pass a mixture of integer and floating-point subroutine parameters in registers. However, split files allow graceful two-chip implementations and smaller integer-only implementations. They also need fewer read/write ports per file to sustain a given amount of multiple instruction issue.

We also considered whether each file should contain 32 or 64 registers. We chose 32, largely because

1. Thirty-two registers in each file are enough to support at least eight-way multiple issue

2. Two valuable instruction bits are better used to make a 16-bit displacement field in memory-format instructions.

More registers might seem better, but excess registers consume chip area and access time, save/restore speed across subroutines and context switches, and instruction bits that might be put to better use. Compilers can deliver substantial performance gains when given 32 registers instead of 16, but there is no clear evidence of similar gains with 64 registers. Demand for registers is likely to increase slowly in the future, but a number of implementation techniques, such as short latency pipelines and register renaming, should satisfy this demand.

Multiple Instruction Issue

Our design sought to eliminate any mechanism that would hinder aggressive multiple instruction issue implementations. Therefore we tried to avoid all special or hidden processor resources.[8] Thus, the Alpha AXP architecture has no condition codes, no global exception enables, no multiplier-quotient or string registers, no branch delay slots, no suppressed instructions or skips, no precise arithmetic exceptions, and no single-byte writes to memory. All of these features, found in some RISC architectures, have the effect of hindering multiple instruction issue, or hindering pipelining of multiple instances of the same instruction. For example, a dedicated string register makes it hard to have three unrelated string operations in the pipeline at once.

To illustrate the performance loss associated with special or hidden processor resources, consider a dual-issue implementation with a four-cycle-deep pipeline. At the beginning of each cycle, up to six prior instructions are partially executed and two more are about to be issued. Six prior instructions can have six pending writes to result registers, plus six sets of side effects on special or hidden processor resources. The next two instructions can specify a total of four operand registers, two more result registers, and two more sets of side effects on special or hidden resources. The decision to issue 0, 1, or 2 of the next instructions involves 36 simple comparisons of pairs of register numbers and 12 complex comparisons of sets of side effects. The number of such comparisons increases as a function of the issue width, the pipeline depth, and the number of special or hidden processor resources. The complexity of these comparisons can limit the clock rate. The register-number comparisons are unavoidable, therefore we tried to limit special or hidden processor resources.

Branch Delay Slots. The Alpha AXP architecture has no branch delay slots. The branch delay slots found in some RISC architectures require exactly one following instruction to be executed after a conditional branch. In 1988 this was, perhaps, a good idea for overlapping branch latency in a single-issue chip with a one-cycle instruction cache. In 1995, however, it will not scale well to a four-way issue chip with a two-cycle instruction cache. Instead of one instruction, up to eight instructions would be needed in the delay slot. Branch delay slots also introduce a restart problem if the instruction in the delay slot faults: one restart program counter is needed for the delay slot and another one for the actual branch target.

Suppressed Instructions. The Alpha AXP architecture has no suppressed instructions, whereby the execution of one instruction conditionally suppresses a following one. Suppressed (or skipped) instructions are found in other RISC architectures. The suppression bit(s) represent nonreplicated hidden state, so multiple instruction issue is difficult for more than one potential suppressor. If an interrupt is taken between a suppressor and suppressee, or if the suppressee takes a restartable exception (e.g., page fault), the correct version of the suppression state must be saved and restored. There are also definitional problems with this approach: Are exceptions ever reported for suppressed instructions? What happens if the suppressed instruction suppresses a third instruction?

Byte Load or Store Instructions. The Alpha AXP architecture has no byte load or store instructions and no implicit unaligned accesses. There also are no partial-register writes. The byte load/store instructions and unaligned accesses found in some RISC architectures can be a performance bottleneck. They require an extra byte shifter in the speed-critical load and store paths, and they force a hard choice in fast cache design. The partial-register writes found in other RISC architectures can also be a performance bottleneck because they require masking and shifting in the fundamental operation of accessing a register.

On a previous project involving a MIPS implementation, we found the shifter for the load-left/load-right instructions to be a direct cycle-time bottleneck. Also, the VAX 8700 implementation (circa 1986) removed the byte shifter in the load/store hardware in favor of a faster microcycle, with 2 cycles for a byte load and 6 cycles for an unaligned 32-bit access. This decision achieved a net performance gain. Our experience encouraged us to avoid byte load/store.

An additional problem with byte stores is that an implementer may easily choose only two of the three design features: fast write-back cache, single-bit error correction code (ECC), or byte stores.

Byte stores are straightforward in simple byte-parity write-through cache implementations. Except for the expensive design of four or five ECC bits for every eight bits of data, a byte store to a fast ECC write-back cache involves

1. Reading an entire cache word*
2. Checking the ECC bits and correcting any single-bit error
3. Modifying the byte
4. Calculating the new ECC bits
5. Writing the entire cache word

This read-modify-write sequence requires hidden sequencer hardware and hidden state to hold the cache word temporarily. The sequencer tends to slow down ordinary full-cache-word stores. The need for byte stores tends to ripple throughout the memory subsystem design, making each piece a little more complicated and a little slower. With non-replicated hidden state, it is difficult to issue another byte store until the first one finishes. Finally, the existence of a byte store instruction has led to programs and library routines for other RISC implementations with single-byte move and compare loops. String manipulation on Alpha AXP implementations is up to eight times faster by processing eight bytes at a time.[9]

Instead of including byte load/store, we followed the RISC philosophy of exposing hidden computation as a sequence of many simple, fast instructions. In the Alpha AXP architecture, a byte load is done as an explicit load/shift sequence; a byte store as an explicit load/modify/store sequence. We tuned the instruction set to keep these sequences short. The instructions in these sequences can be intermixed, scheduled, and issued as multiples with other computation, as can the rest of the instructions in the architecture. Table 1 gives a summary of the Alpha AXP instruction set.

Arithmetic Exceptions. The Alpha AXP architecture has no precise arithmetic exceptions. Reporting an arithmetic exception (e.g., overflow, underflow) precisely means that instructions subsequent to the one causing the exception must not be executed. This is straightforward in a slow implementation that runs a single instruction to completion before starting the next one, but becomes substantially more difficult to do quickly in a pipelined four-way issue implementation. There are standard techniques available for delivering precise exceptions while running quickly (checking exponents, suppressing register writes, exception silos and backout), but these techniques consume substantial design time and can cost some performance. They appear not to scale well with wider multiple issue or faster clocks.

Exceptional cases are just that—exceptional, or rare, events. Based partly on customer requests, we decided to emphasize the performance of normal operations at the expense of exceptional cases. Rather than an implicit exception ordering between every pair of instructions, we adopted the Cray-1 model of arithmetic exceptions—in which exceptions are reported eventually—plus an explicit trap barrier (TRAPB) instruction that can be used to make exception reporting as precise as desired.[10] We also documented a code-generation design that needs one trap barrier per branch (at most) to give precise reporting. Using TRAPB instructions in the first Alpha AXP implementation lowers performance 3 percent to 25 percent in real floating-point programs and less than 1 percent in integer programs, but improves cycle time approximately 10 percent.

In contrast to arithmetic exceptions, memory management exceptions, such as page faults, are reported precisely. This is not as much of a burden on implementers as precise arithmetic exceptions would be, and lack of precise memory management faults would be a severe burden on software writers.

Table 1 Alpha AXP Architecture Instruction Set Summary

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

Floating Point 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 single)
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 and Conditional 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 reg = 0
CMOVNE	Conditional move if reg ≠ 0
CMOVLT	Conditional move if reg < 0
CMOVLE	Conditional move if reg ≤ 0
CMOVGT	Conditional move if reg > 0
CMOVGE	Conditional move if reg ≥ 0
CMOVLBC	Conditional move if reg, low bit clear
CMOVLBS	Conditional move if reg low bit set
CMPBGE	Compare bytes, unsigned
ZAP	Clear selected bytes
ZAPNOT	Clear inselected bytes

Integer Branch	
BEQ	Branch if reg = 0
BNE	Branch if reg ≠ 0
BLT	Branch if reg < 0
BLE	Branch if reg ≤ 0
BGT	Branch if reg > 0
BGE	Branch if reg ≥ 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

Floating Point 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

Continued on next page

Table 1 Alpha AXP Architecture Instruction Set Summary (continued)

Floating Point Computation and 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 reg = 0
FCMOVNE	FP conditional move if reg ≠ 0
FCMOVLT	FP conditional move if reg < 0
FCMOVLE	FP conditional move if reg ≤ 0
FCMOVGT	FP conditional move if reg > 0
FCMOVGE	FP conditional move if reg ≥ 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 = (VAX double)
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 quadword (VAX double)
CVTQF	Convert quadword to F format (VAX single)
CVTQG	Convert quadword to G format (VAX double)
CVTDG	Convert D to G format (VAX double/double)
CVTGD	Convert G to D format (VAX double/double)
CVTGF	Convert G 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 to S format (IEEE double/single)
CVTST	Convert S to T format (IEEE single/double)
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) date hint
FETCH_M	Prefetch (cache) data, modify hint
MB	Memory barrier (serialize)
WMB	Memory barrier (serialize) write
RPCC	Read process cycle counter
RC	Read and Clear
RS	Read and set
PALRES0	PALcode reserved opcode 0
PALRES1	PALcode reserved opcode 1
PALRES2	PALcode reserved opcode 2
PALRES3	PALcode reserved opcode 3
PALRES4	PALcode reserved opcode 4

Shared-memory Multiprocessing

The Alpha AXP architecture's shared-memory multiprocessing model is an integral part of the design. It is not the add-on found in other RISC architectures.

The underlying primitive for safe updating of a multiprocessor-shared memory location is a sequence of RISC instructions: load-locked, in-register modify, store-conditional, test. If this sequence completes with no interrupts, no exceptions, and no interfering write from another processor, then the store-conditional stores the modified result, and the test indicates success: an atomic update was in fact performed.

If anything goes wrong, the store-conditional does not store a result, and the test indicates failure. The program must then retry the sequence until it succeeds. We chose this primitive sequence (quite similar to the MIPS R4000 chip design[5]) because it can be implemented in a way that scales up with processor performance. In the absence of an interfering write, the entire sequence can be done in an on-chip write-back cache, and hundreds of chips can do noninterfering sequences simultaneously. The sequence can also be used to achieve byte granularity* of writes in shared memory.[6]

The Alpha AXP architecture has no strict multiprocessor read/write ordering, whereby the sequence of reads and writes issued by one processor in a multiprocessor configuration is delivered to all other processors in exactly the order issued. Strict order is simple, but has a problem similar to that of byte stores. An implementer may easily choose only two of the three design features: pipelined writes, bus retry, or strict read/write ordering.

If one processor starts a write to location A and a write to location B, then discovers that the write to A has failed (bus parity error, etc.) and retries it successfully, then a second processor will observe the writes out of order: B, then A.

Before Alpha AXP implementations, many VAX implementations avoided pipelined writes to main memory, multibank caches, write-buffer bypassing, routing networks, crossbar memory interconnect, etc., to preserve strict read/write ordering. The Alpha AXP architecture's shared-memory model instead specifies no implicit ordering between the reads and writes issued on one processor, as viewed by a different processor. This programming model is an enabling technology for a wide variety of high-performance implementation techniques. Strict ordering can be specified when needed by insertion of explicit memory barrier (MB) instructions, quite similar to the IBM System/370 serialization design.[11]

Data Representation and Processor State

This section describes the fundamental Alpha AXP data types and their representation in memory and registers. It also describes the complete hardware register state for each processor and outlines the additional state maintained by operating-system-specific PALcode routines. The Alpha AXP architecture differs from other RISC architectures by carefully specifying a canonical form for 32-bit data in 64-bit registers. A canonical form is a standardized choice of data representation for redundantly encoded values. Since 32-bit operations assume canonical operands and give canonical results, very few explicit conversions between 32- and 64-bit representations are needed.

The fundamental unit of data in the Alpha AXP architecture is a 64-bit quadword.* As shown in Figure 1, quadwords may reside in memory or registers. For backwards compatibility, 32-bit longwords* may also be stored in memory.

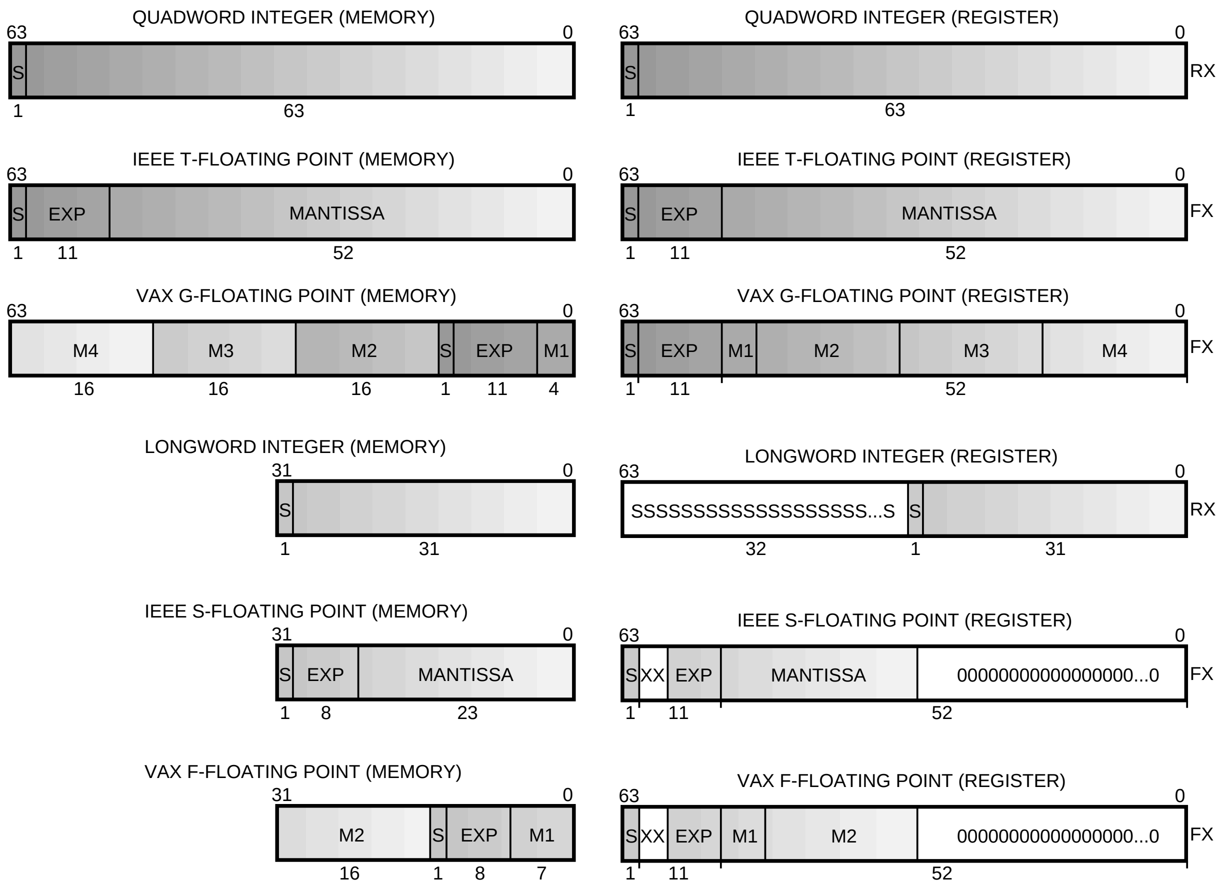
QUADWORD INTEGER (MEMORY)
63 0 S 1 63
QUADWORD INTEGER (REGISTER)
63 0 S RX 1 63
IEEE T-FLOATING POINT (MEMORY)
63 0 S EXP MANTISSA 1 11 52
IEEE T-FLOATING POINT (REGISTER)
63 0 S EXP MANTISSA FX 1 11 52
VAX G-FLOATING POINT (MEMORY)
63 0 M4 M3 M2 S EXP M1 16 16 16 1 11 4
VAX G-FLOATING POINT (REGISTER)
63 0 S EXP M1 M2 M3 M4 FX 1 11 52
LONGWORD INTEGER (MEMORY)
31 0 S 1 31
LONGWORD INTEGER (REGISTER)
63 0 SSSSSSSSSSSSSSSSSSSS...S S RX 32 1 31
IEEE S-FLOATING POINT (MEMORY)
31 0 S EXP MANTISSA 1 8 23
IEEE S-FLOATING POINT (REGISTER)
63 0 S XX EXP MANTISSA 00000000000000000...0 FX 1 11 52
VAX F-FLOATING POINT (MEMORY)
31 0 M2 S EXP M1 16 1 8 7
VAX F-FLOATING POINT (REGISTER)
63 0 S XX EXP M1 M2 00000000000000000...0 FX 1 11 52

Figure 1 Data Representation

There are three fundamental data types: integer, IEEE floating point, and VAX floating point; each is available in 32-bit and 64-bit forms.[4,12] VAX floating-point values in memory have 16-bit words swapped, for compatibility with VAX (and PDP–11) formats. The VAX floating-point load and store instructions do word swapping* to give a common register order. The 32-bit load instructions expand values to 64-bit canonical form, and the 32-bit store instructions contract 64-bit values back to 32.[13] All register-to-register operations are thus done on full 64-bit values in a common integer or floating-point format. No partial-register reads or writes are done.

The canonical form of a 32-bit value in a 64-bit integer register has the most significant 33 bits all equal to bit<31>. In essence, bit<31> is kept as a "fat bit." This allows signed integer values to be used directly in 64-bit arithmetic and branches. This canonical form is maintained as a closed system (even for 32-bit data considered to be "unsigned") by using a combination of 64-bit operates, 32-bit add/subtract/multiply, and two-instruction sequences for shifts.

The canonical form of a 32-bit value in a 64-bit floating-point register has the 8-bit exponent field expanded to 11 bits and the 23-bit mantissa field expanded to 52 bits. Except for IEEE denormals,* this allows single-precision floating-point values to be used directly in double-precision arithmetic and branches. This canonical form is maintained as a closed system by using single-precision instructions.

Bytes and words (16-bit quantities) are not fundamental data types. They may be transferred between memory and registers with short sequences of instructions and manipulated in registers using normal arithmetic and the byte-manipulation instructions described in the Operate Instructions section.

The hardware processor state, shown in Figure 2, includes 32 integer registers R0..R31 of 64 bits each; R31 is always zero. There are also 32 floating-point registers F0..F31 of 64 bits each; F31 is always zero. Writes to R31 and F31 are ignored.

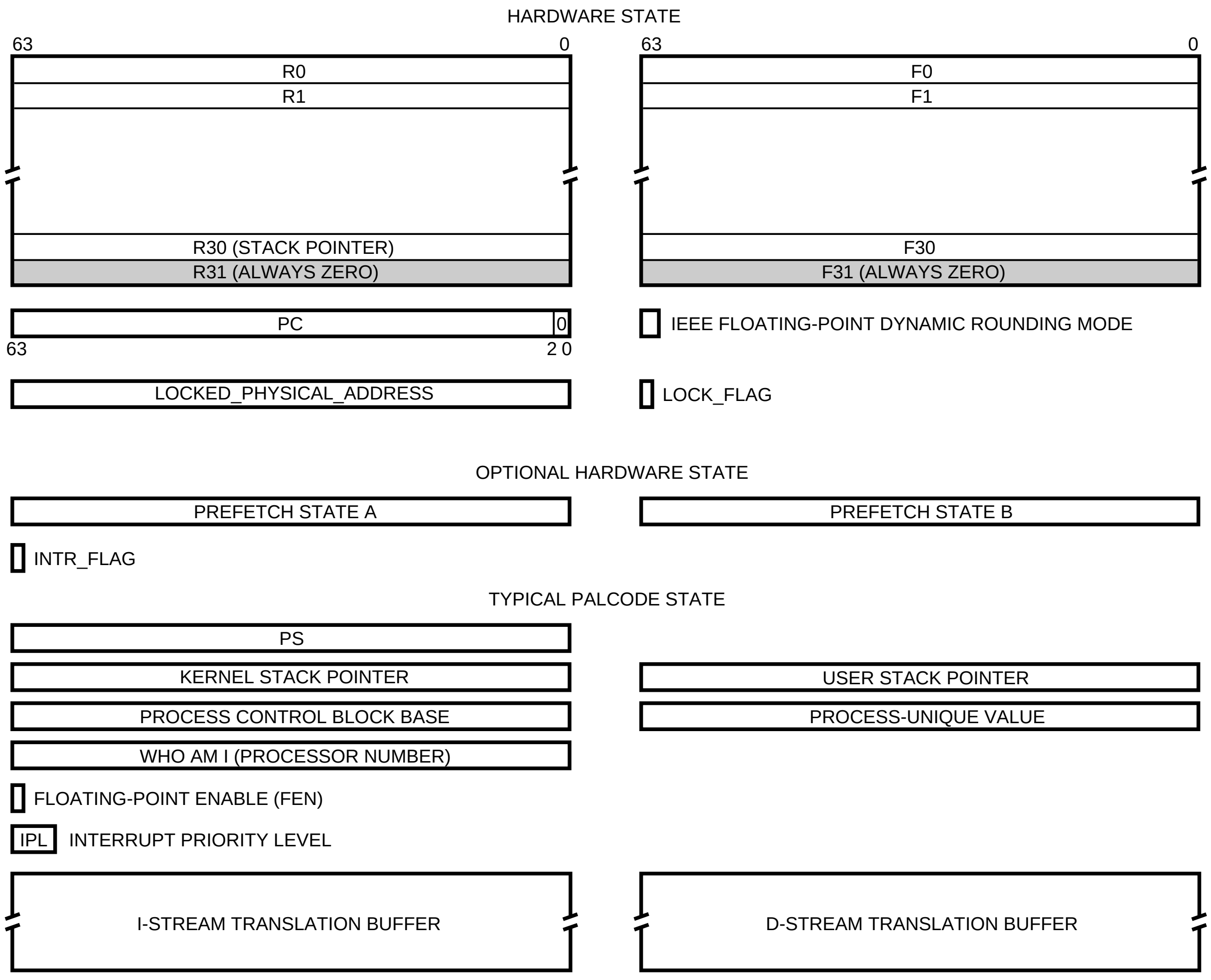

Figure 2 Per-processor State

A 64-bit program counter (PC) contains a longword-aligned virtual byte address (i.e., the low 2 bits of the PC are always zero). The VAX architecture keeps the PC in general register 15, where it is directly used for PC-relative memory addressing. In the Alpha AXP architecture, however, code and data pages are usually separated by 64 kilobytes (KB) or more to allow separate memory protection, but the 16-bit displacement in load/store instructions cannot span more than 64KB.

The hardware processor state includes a lock flag and a locked physical address for the load-locked /store-conditional sequence. It also has a floating-point control register containing the IEEE dynamic rounding mode.*

Hardware implementations may optionally include a pair of state registers for memory prefetching (FETCH/FETCH_M instructions), and an optional interrupt flag for use only by translated VAX OpenVMS AXP programs that reproduce complex instruction set computer (CISC*) instruction atomicity using a sequence of RISC instructions.[6]

In addition to the above hardware state, the privileged architecture library routines for the various operating systems implement additional state. This state may be maintained by hardware or (PALcode) software, at the option of the implementer, and it varies from one operating system to another. Typical PALcode state includes a processor status (PS) word, kernel and user stack pointers, a process control block base for context switching, a process-unique value for threads, and a processor number for multiprocessor dispatching. Additional PALcode state may include a floating-point enable bit, interrupt priority level, and translation look-aside buffers for mapping instruction-stream and data-stream virtual addresses. All of this state is soft in the sense that it is defined only in relationship to the PALcode routines for a specific operating system. In a multiprocessor implementation, all of the above is replicated for each processor.

Memory Access

Alpha AXP memory is byte addressed, using the lowest-numbered byte of a datum. Only aligned longwords or quadwords may be accessed: an aligned longword is a four-byte datum whose address is a multiple of four; an aligned quadword is an eight-byte datum whose address is a multiple of eight. Normal load or store instructions that specify an unaligned address take a precise data alignment trap to PALcode (which may do the access using two aligned accesses or report a fatal error, depending on the operating system design).

Alpha AXP implementations allow data to be accessed using either a little-endian* view (byte 0 is the low byte of an integer), or a big-endian* view (byte 0 is the high byte of an integer). As described in the Load/store Instructions section, there is a one-instruction bias in the sequences for little- and big-endian byte manipulation.

Virtual addresses are a full 64 bits; implementations may restrict addresses to have some number of identical high-order bits, but must always distinguish at least 43 bits. Virtual addresses are mapped in an operating-specific way to physical addresses, using fixed-size pages. Memory protection is done on a per-page basis. Address mapping errors (e.g., protection, page faults) take precise traps to PALcode. Each page may also be marked to provide a fault on each read, write, or instruction-fetch.

Virtual addresses may be further qualified by address space numbers (ASNs), to allow multiple disjoint addresses spaces. The choice of disjoint or common mapping across all processes is done on a per-page basis.

The virtual- to physical-address mapping is done on a per-page basis. Each implementation may have a page size of 8KB, 16KB, 32KB, or 64KB. The 64KB upper bound allows a linker to allocate blocks of memory with differing protection or ASN properties far enough apart to work on all implementations. The virtual- to physical-address mapping can be many to one, i.e., synonyms are allowed. In a multiprocessor implementation, shared main memory locations have the same physical address on all processors. Per-processor unshared locations are also allowed.

Memory has longword granularity: two processors may simultaneously access adjacent longwords without mutual interference. The load-locked/store-conditional sequence discussed previously can be used to achieve multiprocessor byte granularity.

Input/output is memory mapped: some physical memory addresses may refer to I/O device registers whose access triggers side effects (such as the transfer of data). Side effects on reads are discouraged.

Instruction Formats

Four fundamental instruction formats—operate, memory, branch, and CALL_PAL—are shown in Figure 3. All instructions are 32 bits wide and reside in memory at aligned longword addresses. Each instruction contains a 6-bit opcode field and zero to three 5-bit register-number fields, RA, RB, and RC. The remaining bits contain function (opcode extension), literal, or displacement fields. To minimize register file ports in fast implementations, RB is never written, and RC is never read.

All the operate instructions are three-operand register-to-register, calculating *RC = RA operate RB*. In integer operates, the opcode and a 7-bit function field specify the exact operation. Integer operates may have an 8-bit zero-extended literal instead of RB. In floating-point operates, the opcode and an 11-bit function field specify the exact operation. There are no floating-point literals.

Memory format instructions are used for loads, stores, and a few miscellaneous operations. Loads and stores are two-operand instructions, specifying a register RA and a base-displacement virtual byte address. The effective address calculation sign extends the 16-bit displacement to 64 bits and adds the 64-bit RB base register (ignoring overflow). The resulting virtual byte address is mapped to a physical address. The miscellaneous instructions make other uses of the RA, RB, and displacement fields.

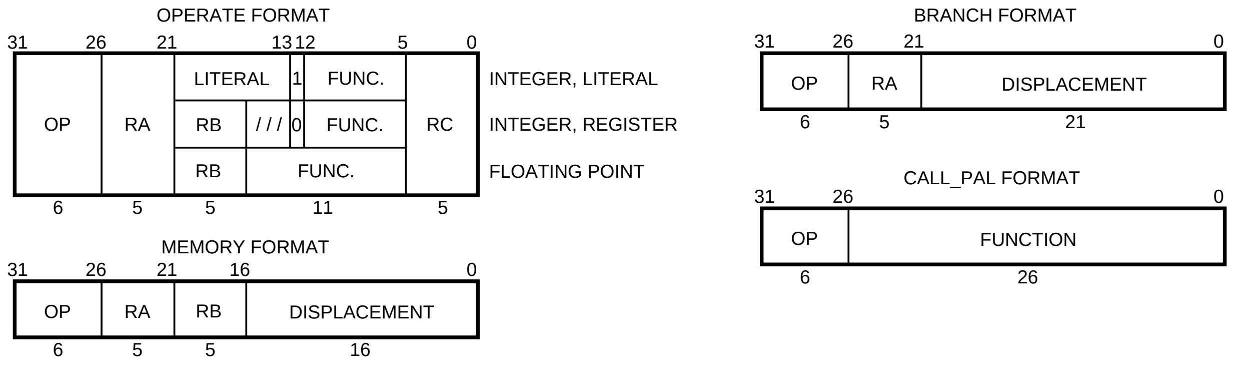

Figure 3 Instruction Formats

Branch format instructions specify a single register RA and a signed PC-relative longword displacement. The branch target calculation shifts the 21-bit displacement left by 2 bits to make it a longword (not byte) displacement, then sign extends it and adds it to the updated PC. Conditional branch instructions test register RA, and unconditional branches write the updated PC to RA for subroutine linkage. The large longword displacement allows a range of ±4MB, substantially reducing the need for branches around or to other branches.

The CALL_PAL instruction has only a 6-bit opcode and a 26-bit function field. The function field is a small integer specifying one of a few dozen privileged architecture library subroutines.

Operate Instructions

There are five groups of register-to-register operate instructions: integer arithmetic, logical, byte-manipulation, floating-point, and miscellaneous. All instructions operate on 64-bit quadwords unless otherwise specified.

Integer Arithmetic Instructions. The integer arithmetic instructions are add, subtract, multiply, and compare. Add, subtract, and multiply have variants that enable arithmetic overflow traps. They also have longword variants that check for 32-bit overflow (instead of 64) and force the high 33 bits of the result to all equal bit<31>. Add and subtract also have scaled variants that shift the first operand left by 2 or 3 bits (with no overflow checking) to speed up simple subscripted address arithmetic. The UMULH instruction (from PRISM) gives the high 64 bits of an unsigned 128-bit product and may be used for dividing by a constant. There is no integer divide instruction; a software subroutine is used to divide by a nonconstant. The compare instructions are signed or unsigned and write a Boolean result (0 or 1) to the target register.

Logical Instructions. The logical instructions are AND, OR, and XOR, with the second operand optionally complemented (ANDNOT, ORNOT, XORNOT). The shifts are shift left logical, shift right logical, and shift right arithmetic. The 6-bit shift count is given by RB or a literal. The conditional move instructions test RA (same tests as the branching instructions) and conditionally move RB to RC. These can be used to eliminate branches in short sequences such as MIN(a,b).

Byte-manipulation Instructions. The byte-manipulation instructions are used with the load and store unaligned instructions to manipulate short unaligned strings of bytes. Long strings should be manipulated in groups of eight (aligned quadwords) whenever possible. The byte-manipulation instructions are fundamentally masked shifts. They differ from normal shifts by having a byte count (0..7) instead of a bit count (0..63), and by zeroing some bytes of the result, based on the data size given in the function field.

The extract (EXTxx) instructions extract part of a 1-, 2-, 4-, or 8-byte field from a quadword and place the resulting bytes in a field of zeros. A single EXTxL instruction can perform byte or word loads, pulling the datum out of a quadword and placing it in the low end of a register with high-order zeros. A pair of EXTxL/EXTxH instructions can perform unaligned loads, pulling the two parts of an unaligned datum out of two quadwords and placing the parts in result registers. A simple OR operation can then combine the two parts into the full datum.

The insert (INSxx) and mask (MSKxx) instructions position new data and zero out old data in registers for storing bytes, words, and unaligned data. If the Alpha AXP architecture were a four-operand one, inserting and masking could have been combined into a single instruction.

The compare-byte instruction allows character-string search and compare to be done eight bytes at a time. The ZAP instructions allow zeroing of arbitrary patterns of bytes in a register. These instructions allow very fast implementations of the C language string routines, among other uses.

Floating-point Arithmetic Instructions. The floating-point arithmetic instructions are add, subtract, multiply, divide, compare, and convert. The first four have variants for IEEE and VAX floating-point, and single- and double-precision data types. They also have variants that enable combinations of arithmetic traps and that specify the rounding mode. The single-precision instructions write canonical 64-bit results, but do exponent checking and rounding to single-precision ranges. The compare instructions write a Boolean result (0 or nonzero) to the target register. The convert instructions transfer between single and double, floating-point and integer, and two forms of VAX double (D-float and G-float). A combination of hardware and software provides full IEEE arithmetic. Operations on VAX reserved operands,* dirty zeros,* IEEE denormals, infinities,* and not-a-numbers* are done in software.

There are also a few floating-point instructions that move data without applying any interpretation to it. These include a complete set of conditional move instructions similar to the integer conditional moves.

Miscellaneous Instructions. The miscellaneous instructions include: memory prefetching instructions to help decrease memory latency, a read cycle counter instruction for performance measurement, a trap barrier instruction for forcing precise arithmetic traps, and memory barrier instructions for forcing multiprocessor read/write ordering.

Load/Store Instructions

The load and store instructions only move data. They never apply an interpretation to the data and therefore never take any data-dependent traps. This design allows moving completely arbitrary bit patterns in and out of registers and allows completely transparent saving/restoring of registers.

The integer load and store quadword unaligned (LDQ_U, STQ_U) instructions ignore the low three bits of the byte address and always transfer an aligned quadword. These instructions are used with the in-register byte manipulation instructions to operate on byte, word, and unaligned data by short sequences of RISC instructions.

Example 1 in Figure 4 shows a two-instruction sequence for loading a byte into the low end of a register, using little-endian byte numbering. Example 2 shows a similar sequence for loading a byte into the high end of a register, using big-endian byte numbering. Example 3 shows a sequence for storing a byte (the first two and last two instructions might issue simultaneously on the first Alpha AXP implementation). Example 4 shows a sequence for an explicit unaligned load quadword (no data alignment trap).

The integer load-locked and store-conditional (LDQ_L, LDL_L, STQ_C, STL_C) instructions are included in the architecture to facilitate atomic updates of multiprocessor-shared data. As described above, they can be used in short sequences of RISC instructions to do atomic read-modify-writes. Example 5 shows a sequence for doing a multiprocessor test-and-set. Note that changing the LDQ_U/STQ_U in Example 3 to AND/LDQ_L/STQ_C/BEQ gives a byte-store sequence that is safe to use with multiprocessor-shared data.

There are two related load address instructions. LDA calculates the effective address and writes it into RC. LDAH first shifts the displacement left 16 bits, then calculates the effective address and writes it into RC. LDAH is included to give a simple way of creating most 32-bit constants in a pair of instructions. (Because LDA sign-extends the displacement, some values in the range 000000007FFF8000 .. 000000007FFFFFFF require three instructions.) Constants of 64 bits are loaded with LDQ instructions.

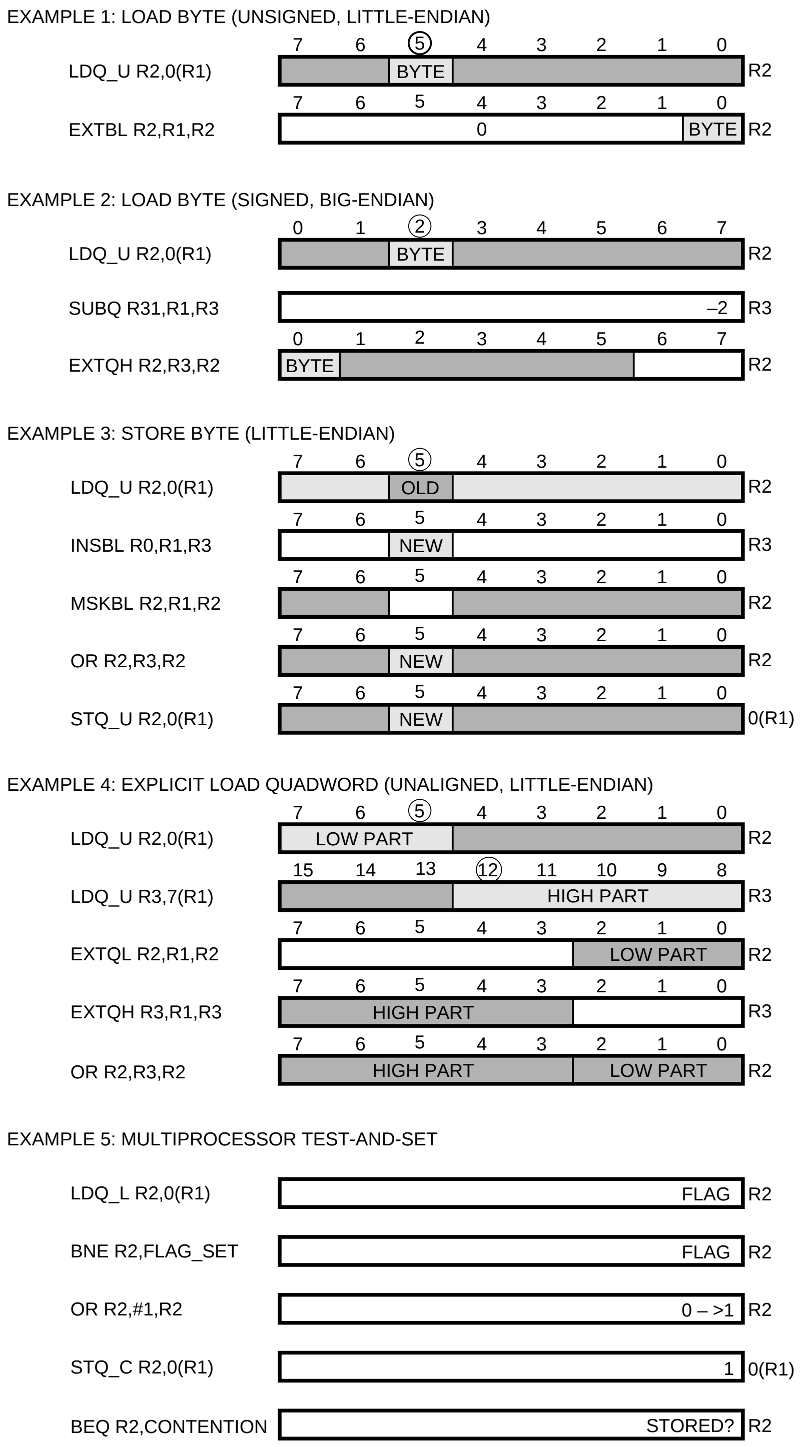

Figure 4   Load/Store Instructions

Branching Instructions

The branch instructions include conditional branches, unconditional branches, and calculated jumps. In addition to the previously described conditional moves, the architecture contains hints to improve branching performance.

The integer conditional branches test register RA for an opcode-specified condition (>0 >=0 =0 !=0 <=0 <0 even odd) and either branch to the target address or fall through to the updated PC address. The floating-point conditional branches are the same, except they do not include even/odd tests. Arbitrary testing (and faulting on VAX or IEEE nonfinite values) can be done by sequences of compare instructions and branch instructions. Logical or arithmetic instructions can combine compare results without using branches.

Unconditional branches write the updated PC to RA for subroutine linkage and branch to the target address. RA = R31 may be used if no linkage is needed.

Calculated jumps write the updated PC to RA and jump to the target address in RB. Calculated jumps are used for subroutine call, return, CASE (or SWITCH) statements, and coroutine linkage.

The architecture specifies three kinds of branching hints in instructions. The hints need not be correct, but to the extent that they are, implementations may perform faster.

The first form of hint is an architected static branch prediction rule: forward conditional branches are predicted not-taken, and backward ones taken. To the extent that compilers and hardware implementers follow this rule, programs can run more quickly with little hardware cost. This hint does not eliminate the use of dynamic branch prediction in an implementation, but it may reduce the need to use it.

The second form describes computed jump targets. Unused instruction bits are defined to give the low bits of the most likely target, using the same target calculation as unconditional branches. The 14 bits provided are enough to specify the instruction offset within a page, which is often enough to start a fastest-level instruction-cache read many cycles before the actual target value is known.

The third form describes subroutine and coroutine returns. By marking each branch and jump as call, return, or neither, the architecture provides enough information to maintain a small stack of likely subroutine return addresses within an implementation. This implementation stack can be used to prefetch subroutine returns quickly.

The conditional move instructions (discussed previously in the Logical Instructions section and the Floating-point Arithmetic Instructions section) and the branching hints eliminate some branches and speed up the remaining ones without compromising multiple instruction issue.

Supervision

The actions underpinning an operating system are performed in PALcode subroutines and are a flexible part of the architecture. All asynchronous events, such as interrupts, exceptions, and machine errors, are mediated by PALcode routines. PALcode establishes the initial state of the machine before execution of the first software instruction. PALcode routines mediate all accesses to physical hardware resources, including physical main memory and memory-mapped I/O device registers.

This design allows implementers to craft a set of PALcode routines that closely match an operating system design, not only for traditional operating systems, but also for specialized environments such as real-time or highly secure computing. As new computing paradigms are adopted and new operating systems are created, the Alpha AXP architecture may well prove flexible enough to accommodate them efficiently.

Future Changes

The Alpha AXP architecture will surely change during its lifetime. In addition to the PALcode flexibility discussed above, explicit performance flexibility and instruction-set flexibility exist in the architecture.

Architectural fields that are too small can limit performance. The Alpha AXP architecture therefore has many fields deliberately sized for later expansion.

Although initial implementations use only 43 bits of virtual address, they check the remaining 21 bits, so that software can run unmodified on later implementations that use (up to) all 64 bits. Furthermore, although initial implementations use only 34 bits of physical address, the architected page table entry (PTE) formats and page-size choices allow growth to 48 bits. By expanding into a 16-bit PTE field that is not currently used by mapping hardware, another 16 bits of physical address growth can be achieved, if ever needed.

Initial implementations also use only 8KB pages, but the design accommodates limited growth to

64KB pages. Beyond that, page table granularity hints allow groups of 8, 64, or 512 pages to be treated as a single large page, thus effectively extending the page-size range by a factor of over 1,000. Each architected PTE format also has one bit reserved for future expansion.

Several other soft PALcode registers, such as the PS or ASN, that need only a few bits today are allocated a full 64 bits for future expansion.

Exception processing can limit performance. PALcode routines deliver exceptions to an operating system, so the design can be gradually improved. In fact, PALcode routines for the data alignment have been improved in the OpenVMS AXP and DEC OSF/1 AXP operating systems. Some currently specified software exceptions (such as IEEE denormal arithmetic) could be moved into PALcode or hardware.

There are a number of areas of instruction-set flexibility designed into the architecture. Four of the 6-bit opcodes are nominally reserved for adding integer and floating-point aligned octaword* (128-bit) load/store instructions.[14] Nine more 6-bit opcodes remain for other expansion. Within each opcode, the function field contains room for further expansion. For example, the scaled add/subtract functions were added between prototype chip and product chip. The fact that the function fields are not fully policed is a mistake.

Within the IEEE floating-point function field, code points are nominally reserved for double-extended* precision (128-bit) arithmetic. Within the memory barrier instruction group, three code points were reserved for subset barriers. One of these has already been redefined as a write-write barrier.

Not all changes involve growth. There are subsetting rules defined for removing either one or both (IEEE and VAX) floating-point data types. If both are removed, the floating-point registers can also be removed. The AMOVxx PALcode routines and RS/RC instructions are defined as optional and can be deleted when the transition of translated VAX code is completed. Other unneeded PALcode routines can also be removed eventually.

Summary

The goals that shaped the Alpha AXP architecture design have largely been realized. For high performance, the first implementation (the DECchip 21064 microprocessor) is listed in the October 1992 *Guinness Book of Records* as the world's fastest single-chip microprocessor. It is too early to measure longevity, but the fact that we had designed-in flexibility in places that changed during development is at least encouraging. OpenVMS AXP, DEC OSF/1 AXP, and Windows NT operating systems all run on Alpha AXP implementations today. Programs from the VAX and MIPS architectures transport easily to Alpha AXP implementations and run quickly. Many of the ideas in the Alpha AXP design are now being adopted by other architectures in the industry.

Appendix

Binary translation—A software technique to change an executable program written for one architecture /operating-system pair into an equivalent program for a different architecture/operating-system pair.

Big-endian memory addressing— A view of memory in which byte 0 of an operand contains the most significant (sign) bit of an integer. Compare little-endian memory addressing.

Byte—An 8-bit datum.

Byte granularity—The appearance that two processors can update adjacent bytes in memory without interfering with each other.

CISC—Complex instruction set computer, characterized by variable-length instructions, a wide variety of memory addressing modes, and instructions that combine one or more memory accesses with arithmetic. CISC designs express computation as a few complex steps.

IEEE denormalized number (denormal)—A floating-point number with magnitude between zero and the smallest representable normalized number. Numbers in this range are typically not representable in other floating-point arithmetic systems; such systems might signal an underflow exception or force a result to zero instead.

IEEE double-extended format—A loosely specifed floating-point format with at least 64 significant bits of precision and at least 15 bits of exponent width; typically implemented using a total of 80 or 128 bits.

IEEE dynamic rounding mode—One of four different rounding rules.

IEEE floating-point—A form of computer arithmetic specified by IEEE standard 754.[12] IEEE arithmetic includes rules for denormalized numbers, infinities, and not-a-numbers. It also specifies four different modes for rounding results.

IEEE infinity—An operand with an arbitrarily large magnitude.

IEEE not-a-number (NaN)—A symbolic entity encoded in a floating-point format. The IEEE standard specifies some exceptional results (e.g., 0/0) to be NaNs.

Linear addressing—A memory addressing technique in which all addresses form a single range, from 0 to the largest possible address. Subscript calculations can create any address in the entire range.

Little-endian memory addressing—A view of memory in which byte 0 of an operand contains the least significant bit of an integer. The terms little-endian and big-endian are borrowed from *Gulliver's Travels* in which religious wars were waged over which end of an egg to break.

Longword—A 32-bit datum.

Multiple instruction issue—A high-performance computer implementation technique of starting more than one instruction at once. An implementation that starts (up to) two instructions at once is called dual-issue; four instructions, quad-issue or four-way issue; etc.

Octaword—A 128-bit datum.

Quadword—A 64-bit datum.

RISC—Reduced instruction set computer, characterized by fixed-length instructions, simple memory addressing modes, and a strict decoupling of load /store memory access instructions from register-to-register arithmetic instructions. RISC designs express computation as many simple steps.

Segmented addressing—A memory addressing technique in which addresses are broken into two or more parts (segments). Subscript calculations can only be done within a single segment, and elaborate software techniques are needed to extend addressing beyond a single segment.

VAX dirty zero—A zero value represented with a non-zero faction; must be converted to a true zero result.

VAX floating-point—A form of computer arithmetic specified by the VAX architecture manual.[4] VAX arithmetic includes rules for reserved operands and dirty zeros.

VAX reserved operand—A non-number that signals an exception when used as an operand in VAX floating-point arithmetic.

VAX word swapping—The rearrangement needed for the 16-bit pieces of a VAX floating-point number to put the fields in a more usual order; this is an artifact of the PDP–11 16-bit architecture.

Word—A 16-bit datum.

Acknowledgments

Hundreds of people have worked on the Alpha AXP architecture, hardware, and software. Many Alpha AXP architectural ideas came from the PRISM design, most notably the PALcode idea.[3] The architecture work was done in the rich environment of dozens and later hundreds of bright, thoughtful, and outspoken professional peers. Ellen Batbouta, Dileep Bhandarkar, Richard Brunner, Wayne Cardoza, Dave Cutler, Daniel Dobberpuhl, Robert Giggi, Henry Grieb, Richard Grove, Robert Halstead, Jr., Michael Harvey, Nancy Kronenberg, Raymond Lanza, Stephen Morris, William Noyce, Charles Nylander, Dave Orbits, Mary Payne, Audrey Reith, Robert Supnik, Benjamin Thomas, Catharine van Ingen, and Rich Witek all contributed directly to the written specification. Rich Witek is co-architect and is the other half of the term "we" used in this paper.

References and Notes

1. G. Amdahl, G. Blaauw, and F. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development,* vol. 8, no. 2 (April 1967): 87-101.

2. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, MA: Digital Press, 1992).

3. R. Conrad et al., "A 50 MIPS (Peak) 32/64b Microprocessor," *ISSCC Digest of Technical Papers* (February 1989): 76-77.

4. R. Brunner, ed., *VAX Architecture Reference Manual* Second Edition (Bedford, MA: Digital Press, 1991).

5. G. Kane and J. Heinrich, *MIPS RISC Architecture* (Englewood Cliffs, NJ: Prentice-Hall, 1992).

6. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation," *Digital Technical Journal*, vol. 4, no. 4 (1992, this issue).

7. The little-endian bias is very slight; both big- and little-endian Alpha AXP systems and software are in fact being built.

8. There are two special-resource anomalies in the architecture that we were unable to avoid: the dedicated state for the load-locked instruction and the dynamic rounding-mode register required for full IEEE conformance.

9. This is borne out in a large customer's recent C string manipulation benchmark result, running 3 to 6 times faster than the customer's expectation (which was based solely on clock rate ratios).

10. *Cray-1 Computer System Reference Manual*, Form 2240004 (Minneapolis: Cray Research, Inc., 1977).

11. *IBM System/370 Principles of Operation*, Form GA22-7000-4 (Armonk, NY: IBM Corporation, 1974): 28.

12. Institute of Electrical and Electronics Engineers, "Binary Floating-point Arithmetic for Microprocessor Systems," Standard Number IEEE-754 (New York, 1985).

13. The careful reader will notice that Alpha AXP implementations require a longword shifter in the load/store path for 32-bit operands. Although we briefly considered a design with no 32-bit operands, we decided to keep 32-bit load/store support for good business reasons. Similarly, Alpha AXP implementations require a word swapper in the load/store path for VAX floating-point operands. We decided to keep VAX floating-point support for good business reasons. Depending on market needs, VAX floating-point support can be removed in the future.

14. Many commercially successful architectures have grown to double-width memory implementations in mid-life: the IBM 709 series from 36 to 72 bits; the IBM System/360 series from 32 to 64 bits; the Digital PDP–11 series from 16 to 32 bits; and the Digital VAX series from 32 to 64 bits. This trend is likely to continue.

Trademarks

The following are trademarks of Digital Equipment Corporation: Alpha AXP, AXP, DEC OSF/1 AXP, OpenVMS AXP, PDP–11, VAX, VAX 8700, and VMS.

CRAY-1 is a registered trademark of Cray Research, Inc.

IBM is a registered trademark of International Business Machines, Inc.

MIPS is a trademark of MIPS Computer Systems, Inc.

OSF/1 is a registered trademark of Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Windows and Windows NT are trademarks of Microsoft Corporation.

Biography

Richard L. Sites Dick Sites is a senior consultant engineer in the Semiconductor Engineering Group, where he is working on binary translators and the Alpha AXP architecture. He joined Digital in 1980 and has contributed to various VAX implementations. Previously, he was employed by IBM, Hewlett-Packard, and Burroughs, and taught at the University of California. Dick received a B.S. in mathematics from MIT and a Ph.D. in computer science from Stanford University. He also studied computer architecture at the University of North Carolina. He holds a number of patents on computer hardware and software.