Box 30
folder 14

RISCy VAX, 1988

Digital Equipment Corporation records
Engineers' papers: Mike Uhler papers: VAX development recor
X2675.2004, *Box 30*

<<< STAR::DOCD$:[NOTES$LIBRARY]EXTENDED-VAX.NOTE;1 >>>
-< Extended (Riscy) VAX Conference >-

Note 26.0 Riscy VAX Support for VAX Emulation No replies
STAR::KRO 594 lines 13-SEP-1988 17:21

+---+---+---+---+---+---+---+
| d | i | g | i | t | a | l |
+---+---+---+---+---+---+---+

INTEROFFICE MEMORANDUM

To: Distribution
Cc:

Date: Sept. 9, 1988
From: Joel Emer
Dept: Dist Sys Arch and Perf
Ext: 226-7537
Loc: LKG1-2/A19
Enet: ERLANG::EMER

Subj: RISCy VAX Support for VAX Emulation

This note is a discussion of some of the VAX architectural support required by RISCy VAX. The operational model is one in which there exists a native RISC architecture and a software environment that provides source level compatibility for non-privileged VAX/VMS user applications. This support would extend to all languages, with the possible exception of MACRO.

In addition to source level compatibility, object level compatibility is also provided. In this case, support is provided for translation of VAX executable images into the native RISC architecture. This translation may be either static or dynamic. Static translation involves taking an executable image and generating an equivalent RISC image. Static translation cannot cover all cases, as it may not be possible to find all the instructions in an images, nor does it work for self-modifying code. In such cases, one must resort to dynamic translation, which does on-the-fly translation of VAX instructions into RISC instructions. Even with no static translation, dynamic translation allows for object level compatibility for existing VAX/VMS program images on RISCy VAX.

To achieve adequate user level compatibility, the VMS environment must also be available on RISCy VAX. As it is unlikely that a compatible rewrite of VMS in a reasonable time frame is possible, it is anticipated that VMS will also be translated into native RISC instructions. Some portions will need to be rewritten, and some portions may simply be recompiled, while others will will require translation. Again the translation may be either be static or dynamic.

In order to provide support for VAX instruction translation a RISC processor needs to provide support a number of characteristics of the original VAX ISP. Such things as common data types and certain instruction and exception/interrupt semantics must be maintained. Furthermore to facilitate the migration of VMS, certain aspects of the privileged architecture must also be preserved. This notes attempts to enumerate important characteristics of the VAX, and discusses how compatibility is maintained or why it is not.

This memo assumes that RISCy VAX is intrinsically a 64 bit machine,

i.e., it provides 64bit registers and operations on quantities up to 64bits in size. Addressing is also assumed to have an address space of larger than 4 Gigabytes, however the ramifications of this address space extension are not discussed here.

Furthermore, this memo assumes that the compatibility constraints are limited to those characteristics of the VAX architecture that are preserved in a multiprocessor environment. In specific, certain instruction atomicity constraints that are maintained in currently existing VAX uniprocessor implementations will not be maintained even on a uniprocessor RISCy VAX.

Following is a discussion of a number of VAX characteristics that must be preserved in the transition to RISCy VAX, and how they can be preserved. Some topics that are not addressed in this memo are the memory management architecture and issues related to process synchronization.

1) Data Types

In order to make emulation of the VAX architecture feasible a number of characteristics of the basic VAX architecture must be preserved. In specific, it must be possible to perform byte addressing of data. There must also be compatibility among the data types supported.

1.1) Integer Data Types

RISCy VAX should support all the VAX integer data types:

- Byte (8bits),
- Word (16bits),
- Longword (32bits), and
- Quadword (64bits).

With a 64bit architecture, the the register representation of a native quadword is different than for a VAX quadword, i.e., it all fits in a single register. This doesn't correspond to the original VAX architecture, therefore translation of VAX quadword data will require that the VAX Quadword be treated as two separate longwords, each of which will reside in the lower 32bits of distinct registers.

All integer data will be represented "little-endian" style, i.e., least significant bits in lowest order addresses. (Bits are numbered right to left 0 on up). This representation is required to preserve the existing data aliasing semantics. For example, perserving the behavior seen when using FORTRAN EQUIVALENCE statements or PASCAL variant records.

Whether loads of words and bytes will do "inserts" or "zero-extend" will be determined by the performance ramifications of the alternatives.

1.2) Floating Point Data Types

All four VAX floating point formats should also be supported.

- F (32bits),
- D (64bits),
- G (64bits),

- H (128bits).

Similar to the case for quadword data, native mode D and G values can reside in a single register. H could reside in two. To get exact VAX semantics the translator would have to treat D and F as 2 longwords, and H as 4 longwords.

To retain compatibility with programs that do source level aliasing of floating point values, the memory representation of the floating point values on RISCy VAX must be identical to that for the VAX.

Subsetting of support for some of the floating point data types should be allowed, especialy H.

1.3) Unsupported Data Types

No explicit native support is provided for some of the VAX data types. In particular no support is provided for:

- Variable Length Bit Fields
- Character Strings
- Decimal
- Queue

No native support is provided for the VAX Variable Length Bit Field data type. This data type can be implemented by interpreting the byte address, bit position and size. Fortunately, since most uses of this data type use a fixed bit position and size this data type can probably be emulated more efficiently in most cases by using shifts and masks. The translator could generate the shift and mask sequence whenever possible.

No specific support is provided for the VAX Character String data type. This data type is really only a sequence of bytes of given length. Therefore, emulation, as already done for the VAX subset architecture, is straightforward.

None of the VAX the decimal formats are supported. Specifically no support is provided for Trailing Numeric String, Leading Separate Numeric String or Packed Decimal String. Again, as for the VAX subset architecture these data type may be emulated. (Is there any specific instruction that might improve the emulation?).

The VAX Queue data type is not supported, but is easily emulated by accessing the two adjacent longwords that make up the data type.

1.4) Alignment

RISCy VAX supports byte addressing. The hardware, however, will only support aligned references. Aligned implies that the datum does not cross specific boundaries in memory associated with the size of the data being referenced. Thus, word alignment implies low order address bit is 0; longword alignment implies 2 low order address bits are 0; and native quadword alignment implies 3 low order address bits are 0.

VAX architectural semantics allow unaligned references. To cover such cases, unaligned references result in traps to software. The trap software must then interpret the reference and make the required aligned references. Consequently, there can be expected to be a significant performance penalty for unaligned references. This penalty might be mitigated slightly by providing special instructions (like on MIPS) to reference the two halves of an unaligned reference.

2.0) Register Set

2.1) GPRs

The translated code must be able to emulate the user visible aspects of the VAX register set. Thus 16 of the RISC GPRs can be dedicated to correspond to the the 16 VAX GPRs. There are no special attributes that distinguish the first 14. The last two, the stack pointer, SP; and the program counter, PC, must be handled specially.

2.2) Program Counter

RISCy VAX will not mechanize the VAX program counter. This does not affect code compiled directly for RISCy VAX, but will affect translated code. In that case, the code generated by the translator is required to generate PC values when they are needed. Mostly the PC values are used for operand specifiers. For most specifiers the results of using the PC as the register are UNPREDICTABLE. The remaining cases are:

- Immediate and Absolute
- PC Relative and PC Relative Deferred

In the case of Immediate or Absolute the translated RISC code may simply generate its own corresponding Immediate or Absolute references. In the PC Relative cases, the translated code should maintain base registers outside of the 16 VAX-visible registers and use references relative to those registers in place of PC relative addressing. These base registers need only be updated on entry to specific code blocks.

In general, base registers could probably be updated exclusively on procedure or subroutine entry. Thus, the the translated code for the start of a procedure or subroutine could load them. This unfortunately doesn't cover the case of branches into the middle of a subroutine. One way of handling such cases is to explicitly capture the need to load base registers as part of the translation of the VAX PC of the target address into the corresponding RISC address of the target.

PC values are also pushed onto the stack for subroutine and procedure calls. In these cases, the translated code must generate the appropriate PC to push. These values are typically static and can be computed just once at the time of translation.

Finally, dynamic PCs must be pushed onto the stack for certain exceptions and interrupts. In general, the translated code should try to avoid generating VAX PCs, except in those cases where user code will be executed as a result of the exception or interrupt. Section 4 contains more discussion of the precision of exceptions and interrupts.

2.3) Stack Pointer

The other special register is the SP. There are no problems with the normal use of the SP as a register or its use for data references. The VAX architecture, however, specifies that there are actually 5 SPs, one for each processor mode, and one for the interrupt stack. RISCy VAX does not provide explicit support these multiple stacks, so this behavior must be emulated in software (see Section 5)

Note that to avoid distributing the emulated VAX stack on RISC exceptions and interrupts, it may be desirable to make the VAX SP register distinct from the register normally designated as the RISC SP.

2.4) Floating Point

The VAX architecture does not have distinct floating point registers. There is no reason that implementation considerations cannot dictate that RISCy VAX have a separate set of floating point registers. In such a circumstance, the software translator would have to move values between the GPRs and the Floating Point registers. Architectural support for these moves would be required.

2.5) Processor Status Longword

The final application user visible aspect of the VAX register set is the VAX Processor Status Longword, PSL. Some components of the PSL correspond directly to hardware state maintained by the RISCy VAX CPU, others are simply software state maintained by the translator. The aggregate state of the PSL must be generated by the translator when a user application needs to examine it. The important parts of the PSL are:

CM (Compatibility Mode)	- Software Emulated
TP (Trace-bit...)	- Software Emulated
FPD (First Part Done)	- May be used by translator
IS (Interrupt Stack)	- Software Emulated
CURMOD	- Hardware Supported?
PRVMOD	- Hardware Supported?
IPL	- Hardware Supported
T,DV,FU,IV (Trap Enables)	- Software Emulated
NZVC (Condition codes)	- Hardware Supported

PDP-11 compatibility mode would be supported entirely by software interpretation as done on all recent VAXes.

In the event of interrupts or exceptions that occur in the middle the execution of a translated VAX instruction, the translator may set First Part Done as allowed by the VAX SRM.

A discussion of processor modes is needed....

RISCy VAX will directly implement the 32 VAX IPL levels.

The T-bit trace control would have be to be handled in software. Since the individual VAX instructions are translated into RISC code, if TP is enabled the translator could simply translate individual instructions and include a trap at the end of the execution sequence.

The other traps (DV,FU,IV) would also be handled in software. In the normal case when no traps are enabled, the translated code does not pay attention to the trap events. If the traps are enabled, then alternative versions of the translated code are generated that explicitly check for the trap conditions. The version of the code to use can be determined by concatenating the trap bits as high-order VAX address bits in the process of

translating VAX PCs to RISC PCs. (It still needs to be tested whether this will perform adequately).

Finally, RISCy VAX must mechanize the Condition Code bits. Generation must be data type dependent, and will be optional on various RISC opcodes. Generating these values rapidly is a significant hardware burden. Therefore the examination of these values must be delayed one instruction following their generation (Similar to read delay slots).

Since CC generation is slow, the translator should avoid making decisions based on their values. For example, when translating consecutive VAX instructions that generate and use condition codes the translator should recognize the sequence of instructions and generate a combined RISCy VAX instruction sequence. This sequence could generate the CCs as a side effect, but use an alternative means of actually causing the same behavior (usually a branch) as would actual examination of the condition codes. Thus, only for those cases where CCs are not used for a number of instructions, need the actual hardware CCs be examined.

3) Exceptions

3.1) VAX Exceptions

To support the VAX architecture it must be possible to generate a number of events in support of VAX exceptions. In specific, the VAX exceptions can be divided into six categories:

- Arithmetic Traps and Faults
- Memory management exceptions
- Operand reference exceptions
- Exceptions occurring as a consequence of an instruction
- Trace exceptions
- Serious system failures

3.1.1) Arithmetic Traps and Faults

In order to achieve reasonable efficiency on arithmetic operations, RISCy VAX must allow arithmetic operations to generate exceptions on certain overflow and underflow conditions. For the VAX user controlled excpetions, individual decsions need to be make on how they are handled, e.g., with mode bits, with distinct instructions or with explicit conditional trap instructions.

3.1.2) Memory Management exceptions

RISCy VAX will generate a memory management exceptions on load/store and probe operations. In addition, any TB miss results in a Memory Management exception. These should be sufficient to generate the VAX standard set of memory management exceptions.

3.1.3) Operand Reference Exceptions

Most VAX operand reference exceptions will not be supported directly by RISCy VAX. Only reserved floating point operand traps may be supported directly. For VAX emulation, the others can be supported by the software translator.

VAX operand reference exceptions fall into two categories. First, are the illegal addressing mode faults. These can easily be detected when the VAX instruction is translated. Second, are the reserved operand exceptions. In many cases, i.e., when the operand specifiers are literals, the validity of an operand can be checked at translation time. When they can't be checked at translation time, they must be checked as part of instruction execution. Most of the cases should have little performance impact, but two might warrant special support. They are:

- Reserved floating point operand

- PSL/W bits

Floating point operands must be checked to see if they have the sign bit set, and a zero exponent. Hardware support for this checking should be provided. In an implementation with a separate floating point register set, this check (and associated exception) could be combined with the operation that transfers the value between the GPRs and the Floating Point register set.

An operation to check for legal combinations of PSL/W bits might also be beneficial. The magnitude of this benefit should be estimated.

3.1.4) Exceptions occurring as a consequence of an instruction

Certain RISCy VAX instruction will generate exceptions, e.g., illegal instructions and mode-change traps.

A exception occurring as a consequence of a VAX instruction can easily be detected when the VAX instruction is translated. These would include, Illegal VAX instruction opcodes, instructions to be emulated, Change-mode instructions and Breakpoints.

3.1.5) Trace exceptions

Handled by translator as described in Section 2.

3.1.6) Serious system failures

Kernel-stack-not-valid, Interrupt-Stack-not-valid and machine-check exceptions must be supported.

3.2) Additional Exceptions

The design of a RISC mitigates that certain other conditions must generate exceptions. In particular, RISCy VAX must provide an unaligned reference exception to indicated that a attempt to read or write a value at an alignment memory address was attempted. To do a VAX emulation, the software in the trap routine must then make the appropriate pair of references to read or write the data.

3.3) Exception Implementation

Exceptions will be handled via a subroutine linkage to a

trap routine. Pertinent information about the trap will be saved in processor registers. Details still need to be specified.

RISCy VAX must be able to emulate the VAX exception vector including the specification of stack on which to handle the exception. Such emulation will be done primarily by VAX emulation code in the RISCy VAX trap handler.

4) VAX Instruction Atomicity

On a uniprocessor, the set of actions associated with many individual VAX instructions appear to be atomic. In a translated version of these instructions, these actions will frequently correspond to multiple RISC instructions. Thus, achieving compatibility with the existing atomicity behavior could be difficult. On a multiprocessor, this atomicity is not preserved, except on designated interlocked instructions. (RISCy VAX will support VAX-style memory interlocks). Thus, by considering the behavior of RISCy VAX to only need correspond to the behavior that would be observed on a multiprocessor, this instruction atomicity need not be preserved.

The only case were this atomicity is important is when a user can regain control following a exception event. In such cases, the state should correspond to the appropriate VAX atomic instruction execution. These occur for the following reasons:

- User written exception handlers
- Delivery of Asynchronous System Traps (ASTs)

In each case, the translator must be able to generate translations of VAX instructions that will complete atomically (for traps) or allow the instruction to be restarted (for faults).

4.1) Atomicity on Exceptions

Exceptions are synchronous with respect to instruction execution. Thus, in a translated instruction sequence the specific RISC instructions that can generate exceptions are known. Given this information, making VAX instruction execution atomic with respect to exceptions, can, in general, be accomplished by deferring all side effects of VAX instruction execution (including register updates for autoincrement and autodecrement) until the very end of the translation sequence. Then in event of an exception, the exception can be taken on behalf of the offending RISC instruction itself. In the case of a fault, no side effects have occurred and the VAX PC can be backed up to reexecute the same translated sequence. In the case of a trap, the remaining work of the VAX instruction must be done as appropriate for the trap. Finally, if a call to a user written exception routine is needed, then the call can be made directly because the state of the machine corresponds properly to atomic VAX instruction execution.

As before the relevant set of VAX exceptions are:

- Arithmetic Traps and Faults
- Memory management exceptions
- Operand reference exceptions

- Exceptions occurring as a consequence of an instruction
- Trace exceptions
- Serious system failures

With the support described in the previous section, judicious placement of instructions that generate the exceptions should allow the translator to guarantee the proper behavior of a RISC instruction sequence corresponding to a single VAX instruction.

The RISCy VAX generates only RISCy VAX exceptions. Thus, the saved PC is a RISC PC. When exceptions are translated into VAX exceptions the RISC PC must be mapped into a VAX PC. This mapping to a VAX PC from a given RISC instruction address may be expensive. Thus, in those cases where the system handles the exception, one can avoid the mapping cost by having the system handle the exception as a RISC exception until it determines it must translate it into a VAX exception. Memory management faults are a good example of where this could be done. At best, this could limit the mapping cost to occur only when the application program is going to receive control as a result of the exception.

4.2) Atomicity on ASTs

ASTs can occur at any time. Furthermore, within a process, AST delivery is defined to be synchronous with respect to VAX instruction execution. Thus, a RISC instruction sequence corresponding to a VAX instruction must complete before service of the AST is allowed. Otherwise, it is possible for a partial set of the side effects of the VAX instruction to occur. Such a partial execution of translated instruction sequence violates the VAX instructions atomicity constraints.

Two basic mechanisms have been proposed to ensure VAX instruction atomicity on ASTs: a hardware solution and a software solution.

The hardware solution relies on hardware selectivity on when an AST can be delivered. Such selectivity can be achieved by designating certain instructions as allowing AST delivery. For example, certain load, branch and operate instructions might allow AST delivery. The translator could then use these instructions as the first instructions of a translated instruction sequence. AST delivery would then occur only on those instructions, thus assuring atomic VAX instruction execution. (Note that every VAX instruction need allow AST delivery, but branches should, so that one cannot create a loop that never looks for ASTs.)

The software solution allows ASTs to occur on any RISC instruction. The AST trap routine would then determine where the end of current VAX instruction is, and single step up to it.

5) Privileged Architecture

The RISCy VAX privileged architecture will be supported using an EPICODE strategy like that of PRISM. Under this strategy, some RISC instructions result in direct subroutine calls to routines at predetermined addresses. Arguments for these routines will be provided in the GPRs. These routines will execute in a privileged context.

Following are those aspects of the VAX privileged architecture that need to be supported.

- Stack pointers (*SP)
  Not supported by RISCy VAX, but can be handled by Epicode.

- Memory Management (*BR,*LR)
  The VAX memory management registers are irrelevant to RISCy VAX. VMS memory management must be rewritten.

- IPL/AST (IPR, ASTLVL, SIRR, SISR)
  The VAX IPL architecture is supported directly by RISCy VAX. Actually, the IPL state can be maintained outside of the CPU and managed using Epicode.

- Clock Stuff (ICCS, NICR, ICR, TODR)
  Functionality supported by processor dependent Epicode with control logic external to the CPU. (Is subsetability important?)

- Console (EX*,TX*)
  Functionality can be supported by processor dependent Epicode.

- Mem Mngment (MAPEN, TBI[AS], TBCHK)
  Functionality supported (except TBCHK) by processor dependent Epicode.

- PME
  Unsupported.

- SID
  Supported by Epicode, and external logic.

VAX Task Force
Status Update

October 14, 1988

Bob Supnik

digital

For Internal Use Only

Purpose

The purposes of today's presentation are:

- To acquaint STF with the work done to date by the VAX Task Force.
- To obtain STF's technical feedback on key issues.
- To obtain STF's approval for further work.

The VAX Task Force

- When Prism was cancelled, the Executive Committee asked for renewed focus on making proprietary (VAX/VMS) systems competitive.
- An interdisciplinary VAX Task Force was formed with the aim of extending VAX/VMS' competitive lifespan.
- The Task Force, consisting of representatives from HPS, MSB, SSG, NaC, CRA, and SCO, has met periodically since then to study alternatives.
- The Task Force has now reached a point where its work is affecting existing plans.
- The Task Force seeks feedback on, and approval of, the proposed courses of action.

Mission

- Extend the competitive lifetime of DEC's proprietary VAX/VMS systems.
  - Improve VAX/VMS' competitiveness against newer hardware/software systems.
- Maximize the lifetime revenue from VAX/VMS systems.
  - Maintain large, healthy (high margin) proprietary business segment.

Strategy

A **proprietary** VAX hardware architecture and VMS software system that will be **demonstrably better** than standard offerings for the next 20 years.

Goals

Proprietary VAX/VMS systems demonstrably better along all dimensions of the product spectrum:

- Function:
  - State of the art software functionality.
  - Robust hardware and software.
  - Compatibility with existing VAX/VMS systems.
- Cost/performance:
  - Improved cost/performance against RISC.
  - Improved absolute performance.
- Time to market:
  - Flexible, extensible, maintainable software.
  - Less complex hardware.

Issues

- Customers increasingly inclined towards standard solutions.
  - Proprietary systems must provide stability and unique value added.
- VAX/VMS hardware and software is becoming uncompetitive.
  - VAX and VMS designed with 1970's concepts to 1970's constraints.
  - Cost/performance uncompetitive vs RISC.
  - I/O uncompetitive vs IBM.
  - Functional limitations (address space, gpr's, page size).
  - Implementation schedule and cost penalties.

Task Force Methodology

- Short term focussed on evolutionary improvements without major architecture or software perturbations.
- Long term focussed on competitiveness from 1992 to 2000+.
  - Assumption: VAX hardware **must** be on the cost/performance curve.
  - Assumption: VMS and layered software must support state of the art computing structures, concepts, and functions.
  - Assumption: Extended addressing is **required** in this timeframe.
  - Assumption: Some non-negligible portion of VAX/VMS value is in binary compatibility.
  - Strong goal: Competitive system performance.
  - Strong goal: Extended functionality.
  - Strong goal: User-perceived VAX/VMS compatibility.

Short Term Recommendations

- VAX processors - more competitive processors in the next 3 years:
  - Better technology - ECL VLSI VAX.
  - Better microarchitecture - Centaurus, ultimate VLSI VAX.
  - Better delivery - NVAX schedule, performance.
- VMS software - more competitive software in the next 3 years:
  - VMS improvements - production systems, human interface, distributed systems, standards, etc.
  - Compiler improvements - interprocedural analysis, targetted code generators, vectors, parallel decomposition, etc.
- VAX/VMS I/O - see recommendations from I/O Task Force.

Long Term Alternatives

- VAX/VMS forever, as is or with minor evolutionary changes.
- New VAX/new VMS, with strong compatibility to existing VAX/VMS.
  - Many choices for implementing new ISA.
  - New ISA has significant "concessions" to VAX history.
  - VMS on VAX migrates to new VMS on new VAX, with VAX image compatibility.
  - I/O architecture must also be overhauled.
- RISC/ported VMS, with (at best) source level compatibility.
- First alternative is status quo (default choice).
- Third alternative was P.VMS (tried and rejected).
- Focus on second alternative - migration to new VAX/new VMS while preserving compatibility.

Key Questions

Key questions facing the Task Force are:

- Is a competitive "makeover" of VAX/VMS:
  - an essential corporate strategy?
  - an interesting technical issue?
  - a lost opportunity of only historical interest?
- Is such a transformation technically feasible?
  - Can VMS be moved off the VAX ISA in our lifetimes?
  - Does software translation work?
- How are tradeoffs on compatibility to be made?
  - What do customers think compatibility means?
  - What is the impact on customers of discontinuities?
  - What are the costs to DEC of indefinite continuity?

  Plans: surveys, focus groups, competitive analysis, etc.
- How does VAX/VMS coexist with new VAX/new VMS? What happens in clusters?
- What are the quantitative tradeoffs between alternatives?

Recommendations

The Task Force's recommendations to date:

- Approve short term focussed projects.
- Close on strategy, goals, assumptions for VAX longevity.
- Close on extended functionality for new VAX/VMS systems.
- Close on a "pre-migration" strategy aimed at larger page sizes, loose read/write ordering.
- Within SDT, assign highest priority to VAX-oriented compiler work:
  - Prototype software translator.
  - Interprocedural analysis, optimization.
- Begin intensive study of VMS evolution and porting.

Next Steps

The Task Force's next steps:

- Quantitatively analyze, choose an alternative.
  - The impact on VMS and customer software will be a principle evaluation factor.
- Investigate phaseover issues.
- Integrate with I/O TF recommendations.
- Create concensus, generate enthusiasm, get moving!

Appendix: Details On Alternatives

As we examined alternatives, we found that we were retracing old ground, and that the available alternatives form a spectrum of increasing breaks in compatibility.

- Alternatives:
  - VAX/VMS forever (as is, or with small evolutionary changes).
  - VAX mode on RISC, or VAX translation to RISC.
  - VAX and RISC ISP's in one processor.
  - VAX/RISC coprocessor.
  - VAX software translator to VAX-inated RISC.
  - VMS port to vanilla RISC.
- Or:
  - VAX.
  - Riscy VAX.
  - Dual instruction sets.
  - HR32.
  - SAFE.
  - Prism/P.VMS.

VAX/VMS Forever

- VAX architecture as is, with (perhaps) larger pages and other minor architectural tweaks.
- "Perfect" binary compatibility.
- Maintains present VMS.
- Maintains present compilers.
- Permanent >2X cost/performance penalty, because of:
  - Too few general registers.
  - Irregular, unaligned instruction formats.
  - Procedure call overhead.
  - Small page size.
  - Ordering requirements on reads, writes, exceptions.
- Longer time to market.
  - Complexity $->$ longer design cycles.
  - Complexity $->$ denser technology requirement.
- "Irregular" rather than linear extended addressing.

Riscy VAX

- Several variations:
  - (Sites proposal 1) RISC subset of VAX instructions.
  - (Lampson proposal) RISC translation from VAX instructions.
- "Perfect" binary compatibility.
- Maintains present VMS.
- Requires retargetted compilers.
- Does not realize full potential of RISC technology, due to:
  - VAX calling standard.
  - Insufficient GPRs.
  - Intra- and inter-instruction dependencies.

  (But would the improvement suffice?)
- "Irregular" rather than linear extended addressing.

Dual Instruction Sets

- Several variations:
  - (Sites proposal 2) Conventional VAX implementation, underlying RISC available as alternate ISP.
  - (Stewart proposal) RISC translation from VAX instructions, underlying RISC available as alternative ISP.
- "Perfect" binary compatibility.
- Could maintain present VMS:
  - Eventually, VMS would run native on RISC.
- Requires retargetted compilers, new RTL.
- Complicated to implement - VAX **plus** new ISA in one design.
- Extended addressing only in native RISC.

VAX/RISC Coprocessor

- VAX with RISC coprocessor, migrating to RISC with VAX hardware compatibility mode.
- Or: HR32.
- "Perfect" binary compatibility.
- Could maintain present VMS:
  - Start with user mode ASMP support.
  - Eventually, VMS would run native on new ISA.
- Requires retargetted compilers, new RTL.
- Extra CPU in all systems: affordable in ECL?
- Initial VMS changes to support ASMP model are extensive.
- ASMP performance is questionable.
- RISC must make concessions to current VAX/VMS architectural model.
- Extended addressing only in RISC coprocessor.

VAX Software Translator

- Software translation of VAX EXE's to a "VAX-inated" RISC.
- Or: SAFE.
- Key RISC concessions to VAX past:
  - VAX data types.
  - VAX condition code tracking.
  - 32b image (longword address/data) support.
  - Exceptions and AST support.
  - Small page size support (maybe).
  - VAX privileged architecture (maybe).
- User-mode only binary compatibility; good performance.
- VMS must be **extensively changed** for new architecture.
- Requires retargetted compilers, new RTL, translation tools.
- Is a strategy based on extensively reworking VMS achievable, in our lifetimes?
- What is the "cutover" strategy between old VAX's and new VAX's? between old VMS and new VMS? How do clusters work?

VMS on RISC

- Port VMS to vanilla RISC.
- Or: P.VMS/Prism.
- VAX support limited to source level compatability.
- VMS must be **ported** to new architecture, without significant hardware support.
- Requires retargetted compilers, new RTL.
- Can be prototyped with existing hardware.
- Will this keep VAX customers for DEC?
- Is a strategy based on porting VMS achievable, in our lifetimes?
- How are old VAX's and new RISC's positioned? transitioned?