To: Forest Baskett
Doug Clark
Dave Cutler
Sam Fuller
Alan Kotok
Jud Leonard
Dick Sites
Bob Stewart
Bill Strecker
Bob Supnik
Bob Taylor
Chuck Thacker
Jamie Woodward

I thought you might find this interesting. I don't currently plan to pursue it any further myself, but I would be glad to answer questions, hear comments, discuss other possible schemes, and help to promote it.

Butler Lampson

Jesse Thunder
Bob/Terry Vendi

Write down proposal

VOR: VAX on a RISC

Butler W. Lampson

Systems Research Center
Digital Equipment Corporation
CIRCUS::Lampson

Company Confidential

1. Introduction

One way to make a fast VAX is to build a fast machine with a simple instruction set, and translate VAX instructions into sequences of simple instructions (as opposed to interpreting VAX instructions with sequences of simple instructions, the way current machines do). We call the fast machine a VR (VAX RISC), and the resulting VAX a VOR (VAX on RISC). In order to make the translation as transparent as possible, we keep a cache of translated instructions, and do the translation only on cache misses. Thus the translated instructions are never visible to the program. (This idea was proposed for the C-machine at Bell Labs, and is used in a 68000 Smalltalk system done at Xerox Parc.)

VOR is attractive if the following proposition is true:

P1) A RISC can run programs significantly faster than a VAX built with the same technology for about the same cost.

More precisely, a practical compiler can translate a Fortran, Pascal, C or Bliss program into RISC code which runs faster (say, two times faster) than the VAX code produced by our current compilers. I won't try to argue for this proposition here; if you don't believe it, you may still wish to read on for amusement.

If it is true, however, then the reason for running VAX programs on the VR is compatibility with existing programs, both DEC's and its customers' (another possible reason is the greater compactness of the VAX code; I won't consider this any further here). Since most programs are compiled, a modest compiler effort will make it possible for most programs to run as VR programs. It is OK for VAX programs to run more slowly (since most programs will be converted, and converting any given program should not be too hard unless it is written in assembler). This is just as well, since they certainly will run more slowly. We capture this essence of this argument in:

P2) A VOR is successful if VAX programs run as fast as they would on a conventional implementation of the VAX in the same technology.

The advantage of the VOR is that programs recompiled for the native VR will run faster. The drawbacks are that

the extra hardware for VAX emulation increases the cost, and

the cache must be larger to hold the large translations of VAX into RISC instructions.

P2 asserts that the advantage outweighs the drawbacks.

You might ask: why is a RISC faster than a VAX? I don't hope to settle this question, but it's worthwhile to make a few observations. The native RISC has three advantages:

Simple instructions can be decoded quickly and executed at the rate of one per cycle. It is difficult or impossible to attain this peak rate for VAX instructions. Simple instructions are quite common.

Having lots of registers is valuable if procedures are large, or if register assignments are done across procedure boundaries. In the limit all the local variables for all procedures could be in registers. Access to registers is faster.

RISC branches are faster, and because they are deferred it is often possible to do useful work in the pipeline stutter after a branch. This is only occasionally possible in a VAX implementation.

The VOR doesn't have all these advantages when emulating the VAX, and indeed I think it will be hard to get better performance running VAX instructions than from a VAX done in the same technology. But there are two important gains that remain.

One is that the translator can generate VR instructions based on the entire VAX instruction, which is impractical when the instruction must be decoded at top speed. Thus a register to register or memory to register move can translate into one VR instruction, and even r1:=r2+r3 is only one VR instruction. It's hard to do this in less than four cycles with on-the-fly decoding: one for each operand specified, and one to do the add.

Probably more important is that the VR is simple and small. This means that its design can be optimized much better, since the designer can get it all in his head, and because of the small size signal propagation delays are not as important. This means that a faster cycle time should be possible.

The rest of this document sketches a VOR implementation. Section 2 gives the overall architecture. Section 3 describes the properties of a VR, and Section 4 outlines the translations for the operand specifiers, and for a few instructions. Section 5 discusses cache invalidation, and Section 6 gives possible ways of dealing with a few problems not directly related to the instruction-by-instruction translations.

Many details which require bit-fiddling are omitted, since this is only a sketch. Unfortunately, some important details are probably also omitted; comments on these will be gratefully received.

2. Architecture

Translated instructions are stored in the VR's instruction cache, along with ordinary VR instructions. Since the translated instructions don't exist in the ordinary address space, they must have pseudo-addresses, the VR must be able to generate these pseudo-address as PC values, and the cache must be able to store them. To be specific, suppose we have a 32-bit VR with 16-byte cache lines which uses byte addresses, even though it can only do 32-bit fetches and stores, and therefore traps if the low 2 bits of address are non-zero. Suppose also for simplicity that the VR also handles 32-bit real byte addresses. We convert a VAX PC to a VR PC by adding 2**32 and then shifting it left 6 bits (this works for either virtual or real addresses). The resulting value addresses the first of a group of sixteen 32-bit VR instructions. It is looked up in the cache in the usual way. Since it is 39 bits wide, the cache must handle 39-bit real addresses, or in general addresses with 7 more bits than a vanilla RISC would need.

There are two major questions to answer for ordinary instruction execution:

Q1) How do VAX instructions get translated and put into the cache?

Q2) How does the VR get from one VAX instruction to the next?

Translation is done by a separate element called a translator T which plays the role of a memory to the VR.

```
   VR processor------VR cache-------Translator------Memory bus
```

It watches the addresses the cache sends to the memory when there is a miss. An address <2**32 is an ordinary address; T passes it on to the memory unchanged, and passes back the resulting data unchanged to the cache. Thus T does no work in this case. An address >2**32 is a pseudo-address for a translated instruction. T does not pass this address on to the memory. Instead, it fetches the corresponding VAX instruction (VI), decodes it very much as the decoding section of an existing VAX processor does, and generates translated VR instructions (TIs) to satisfy the miss. These are delivered to the cache just as though they had come from the memory, and the VR proceeds to execute them.

We left room for 16 TIs in the address space occupied by one VAX byte PC. This means that no VI can take more than 16 TIs to emulate. Many more VR instructions might be executed as subroutines, of course. If 16 is not enough, we can increase it to 32, 64 or whatever, at the cost of storing a couple of extra address bits for each cache line and return link.

A more important issue is the breakage in the cache: each VI consumes cache space in multiples of 4 words (the cache line size). So if a VI translates into a single TI, as many MOVs will for example, there will be 3 wasted words. This is part of the cost of the emulation, though it is alleviated a bit by the instruction joining described below.

We can make a rough estimate that the translation will be 4 times the size of the translated instructions, including breakage, by looking at the sizes of a few translations done using the scheme of Section 4. This means that the instruction part of the cache will have to be about 5 times as large as that for a vanilla VAX to yield the same miss rate. E.g., if a 8 KB instruction cache is enough to yield a 5% miss rate, then VR will need a 40 KB instruction cache.

It is desirable to keep the miss rate moderately low, so that T does not have to be too fast. For instance, with a 5% miss rate for instructions, if T takes 5 times as long to generate each TI as VR does to execute it, the total execution time increases from 20 VRs (assuming no misses) to 19+5=24, or a 20% slowdown. We can probably do considerably better than this with a fairly simple translator.

Now for Q2. A straightforward solution would be to generate a branch TI at the end of each translated sequence. This is unattractive because it consumes an entire cycle on every VI, and also takes up a lot of space. So I propose to add a kludge to VR which allows this branch instruction to be encoded more compactly and executed in parallel. The idea is to leave room in each cache line for two small numbers called <u>where</u> and <u>when</u>. Where is added to the address of the first word in the line to yield an <u>implicit branch address</u>, and when is a count of the number of TIs to execute before taking the implicit branch. If the pipelining is like the Titan's, when must be at least 2. The implicit branch is executed in parallel with whatever is going on in instruction (when-1). For ordinary misses, when is loaded with a value such as 0 that disables the mechanism.

If desired, we could avoid doing an add for an implicit branch by making where replace the lsb of PC instead of adding to it. Occasionally the translator would have to generate an explicit branch, when VPC advances so as to change some bits not replaced by where.

We are still in trouble, since every translation must be at least of length 2, and we have many common cases in which the translation should be of length 1. To fix this, the translator will <u>join</u> a short translation of VI1 with the translation of the next VI, VI2. There are several cases. The translation of VI1 is of length 1 (if it were of length 2, no joining would be needed). But VI2's translation may be of any length L:

L=1 Put TI2.1 with TI1.1 of VI1 and set when=2.

L=2 Put both TI2.1 and TI2.2 with TI1,1, making 3 instructions in all, and set when=3.

L>2 Set when=2 and where=(the address of VI2)+4 (pointing to TI2.2). This means that TI2.1 will be joined, but there will be at least 2 more TI2s that are not joined, so there will be time for the when for VI2 to take effect.

VI2 may still have a complete translation in the proper place if it is branched to. For L>2 it will always have this translation, since the join sequence jumps into the middle of it. For L<=2 this translation will never be generated unless VI2 is the target of a successful branch.

We need to keep track of the current VPC. Without joining this is simple, since it is set exactly by implicit branches (and, of course, by explicit branches). With joining we need an extra joined bit in the cache entry which indicates that the first TI of the line is a complete translation for a VI. The use of this bit is described in C8 below.

The translator will presumable be implemented by a microcoded processor much like the I-Box of a standard VAX implementation. Its most important microinstructions are those which emit TIs, parameterized in a couple of ways as discussed in Section 4.

Note: One unfortunate consequence of the pseudo-address scheme is that when a TI PC is mapped from virtual to real it must be reconverted to the VPC form. This means some multiplexors in the TLB.

3. The VR

To keep things definite, I take the Titan as the basis for the VR. I am not sure that this is the best design, but I know that it is a reasonable RISC, and it is certainly a simple one. However, it is necessary to make some changes. I have kept these as few as possible just to reduce the number of variables in this proposal. I am not taking a position on whether it is a good idea to make bigger changes, or to have a quite different VR. It is vitally important to keep VR simple, since we don't want to compromise the performance of native RISC programs.

Titan

For the uninitiated, I will summarize the Titan here. It is a 32-bit machine with 64 registers. Register 0 is a pseudo-register with special properties: when read, it is the program counter; when written, there is no effect. Instructions are of four kinds:

```
Load or store--r1:=(r2+d)^
               (r2+d)^:=r1.
Here d is a 16-bit signed displacement.
```

Operate--r1:=r2 op r3. The op may be arithmetic or logical. Also, r2 may be a 6-bit signed literal instead of a register.

Shift and extract--r1:=(r2,r3)[i..j]. This puts bits i through j of the 64 bit quantity obtained by concatenating r2 and r3 into r1. There is a variation which puts j bits of r3 starting at the byte addressed by the two low-order bits of r2 into r1, written r1:=r3[8*r2+j..8*r2].

```
Branch--IF condition(r1) THEN GOTO (PC+d)
      --r1:=PC; GOTO (r2+d)
```
A branch doesn't take effect until after the next instruction is executed. Conditions are comparisons with 0 and tests of single bits in r1.

Titan has stalls only in the following cases:

Load or Store in the next instruction after a Store (1 cycle).

Load, Store or Branch indexed by ri in the next instruction after setting ri (1 cycle).

Attempt to read a coprocessor result that isn't ready.

Cache miss.

Exception (3 cycles).

Changes

C1) Titan numbers the bits and bytes backwards from the VAX. It also has the floating-point slightly different. VR has the same data types as VAX.

C2) Just as Titan has a pseudo-register for its PC, VR has a VPC pseudo-register (r15) for the VAX PC, for the use of PC-relative operand specifiers, JSB, CALL and exceptions. It holds the byte address of the first byte of the instruction (see T2 below). Normally this is (PC rsh 6)-2**32, unless it points to TI2.1 or TI2.2 of a joined translation, in which case VI1.L (the length of VI1) must be added. It would be unpleasant to store this in each cache line and add it in automatically. Instead I propose to make T take account of it during normal execution (see T2), and have a single bit in the cache line, copied into the VR machine state, which indicates that a joined instruction is in progress. When VR takes an exception and the emulator code needs to recover VPC, it tests this bit, and if it is set executes the instruction at (PC OR 12). This is the last word of the cache line, not used for a TI when joining has been done, and T will leave there a TI to increment r15 by VI1.L. If you don't like this encoding trick, there are plenty of other possibilities.

C3) Titan has word addresses. VR addresses bytes, although it will trap a Load or Store unless the two lsb of the address are zero (but see C5). This unaligned reference trap captures the address and the source or destination register for use by the trap routine, so that the reference can be completed correctly with reasonable efficiency. The trap is done with the same mechanism used for page faults. However, the register must also be captured, and there must be a way to OR the captured register number into the r1 field of some later instruction (or something equivalent in its effect).

C4) VR has a Load Immediate instruction, just like Load except that it loads the effective address instead. It is written r1:=r2+d. It differs from an operate instruction in having a 16-bit immediate operand. Perhaps there should also be one which does r1:=r2+(d*2**16), though I have not assumed this below.

C5) VR has versions of Load and Store which ignore the two lsb of address, rather than trapping if they are non-zero. These are for reading and writing byte operands. These versions save the two lsb in a pseudo-register BN; they are written (r+d)^B. Also, there are versions which trap only if the lsb are 11, for reading and writing 2-byte operands; written (r+d)^W. These set BN:=0 if they trap, so that the unaligned address trap can load or store the 4 bytes addressed, just as it does in the 32-bit case, and the extract and insert code can believe that it should always work on the bytes pointed to by BN.

Note that ^B and ^W only affect the traps taken and the loading of BN; they don't do any shifting or masking.

C6) VR probably has a way to modify only the low-order byte or halfword of a register. Doing this with vanilla Titan instructions adds two cycles, which is unpleasant. At the same time the CC can be computed correctly for a byte or 2-byte operation.

C7) VR has pseudo-registers for the condition code. I propose to have three CC registers, one for N and Z, one for C, and one for V. They will be set in the obvious way directly from the ALU output for VAX instructions which set them in the obvious way; there must be some way in the instruction to specify that the CC registers should be set. Fortunately Titan instruction are not very tightly encoded, so there are spare bits. For 32-bit Load and Store, it probably works to set CC whenever r1 is one of the 16 VAX registers. The NZ register will just be loaded with the result; Titan can do all the simple arithmetic branches on comparisons with 0.

For VIs that set the CC in a messier way, extra TIs will be generated to get the CC registers set correctly. Possibly there will be added versions of Operate to help this, but things will be kept simple by taking extra cycles when necessary. Likewise, extra TIs will be generated and extra cycles taken when a conditional branch tests a complex combination involving C and V.

The CC pseudo-registers must be set properly for byte and halfword operands. This should probably be done as a byproduct of C6.

C8) Obviously VR must have a VAX branch instruction, which converts a 32-bit number representing the new VPC into the correct 39-bit PC value (PC=(VPC+2**32) lsh 6).

More doubtful changes

C9) VR might have some provision for sign-extension. This is not absolutely required, since sign extension is not that common on the VAX and it costs only 2-3 cycles to do it with Titan instructions.

C10) It might be worthwhile to make a special case for a conditional branch as the last TI: if the branch succeeds, execution of the next TI (which would be the first TI of the next VI in sequence) is turned into the NOP demanded by Titan branch sequencing. Since about 3/4 of VAX branch instructions other than CALL and RETURN actually branch, this might be quite valuable.

Tricks

There are some tricks that can be used to avoid further modifications to the RISC.

T1) To ensure that auto-increment and auto-decrement don't take

permanent effect until after the danger of faults is past, the standard scheme is a log of register changes, which can be used to undo the change in case there is a fault. An alternative suggested by Dave Cutler requires no change to the VR. Instead, T keeps track of the changes caused by auto-increment etc. Whenever a changed register is used, T generated the right code. E.g., after r1 is autoincremented by 4, the address 25(r1) would be translated into 29(r1). Then the register change instructions are put out at the end of the TIs. Since executing the TIs of a VI can never cause a page fault (because the whole VI is translated at once), this is safe.
**** Is this really true?

T2) Similarly, T adjusts for changes in r15 as execution proceeds through the operand specifiers; r15 is not actually updated until when and where send control to the next VI. This is a much bigger win than avoiding the register change log using T1, since there is no convenient place to store the OS lengths. In the same way T will generate a TI to save the correct value of VPC after a subroutine call.

T3) The integer overflow trap can be handled by generating an explicit test as part of the TIs for every VI that can generate it when the trap is enabled. We add the enable bit to the PC pseudo-address to avoid any other testing of it.

T4) The trace trap can be handled in the same way, by generating an explicit TI to cause the trap at the end of every VI when the trap is enabled, and adding the enable bit to the PC pseudo-address.

4. Translations

The strategy is to generate zero or more TIs for each operand specifier (OS), ending up in T with each OS encoded as one of:

Rr	a register
Cc	a 6-bit signed constant
Md(r)	a memory address, contents of r + d

How an OS is encoded is governed by the opcode-dependent use (rwmav) of the operand, and the operand size s (which governs the translations of auto-increment, auto-decrement and indexing. Note that the registers don't have to be VAX registers; the translator is free to assign scratch registers, of which there is an ample supply.

For each OSi (i IN [1..6]) T keeps the following information

Ki: the kind, one of R, C or M. Set according to table 4.1, and not changed thereafter.

rci: the register, either a register number r or a short constant c, i.e., 6 bits plus a one-bit flag. Initialized to ti, a different temporary register for each OS. Set to a register if Ki=R, to a constant if Ki=C according to Table 4.1. Also changed by Read as indicated below.

mi: the memory address d(r), valid only if Ki=M; set up according to table 4.1.

For each VAX register r, T keeps adj[r], the adjustment arising from auto-increment, auto-decrement and advancing of the PC (see T1 and T2 above), according to Table 4.1. Each adj[r] is initialized to 0.

Note that different opcodes which specify differend operand sizes for the same operation may generate quite different sequences. For example, it may be desirable for an 8 or 16 byte ADD to generate the address of a memory operand and call extracode (see below) to get it copied into registers.

Notation:

<...>	generate these VR instructions
s	operand size, in [0..5]
x	index register, initialized to nil
c small	-2**15<=c<2**15
c.lsb	least significant 16 bits of c
c.msb	most significant 16 bits of c
mn.d	the d part of mn=d(r)
mn.r	the r part of mn=d(r)
ti, tti	two sets of 6 temporary registers, one per OS

```
Mode                        Translation

literal c                   IF use#r OR x#nil THEN error, Kn:=C, rcn:=c

immediate                   IF use not IN rav OR x#nil THEN error, kn:=R,
                            IF c small THEN <rcn:=c>
                            ELSE <rcn:=c.msb; rcn:=rcn lsh 16;
                                  rcn:=rcn+c.lsb>

register r                  IF use=a OR x#nil THEN error, Kn:=R,
                            IF adj[r]=0 then rcn:=r
                            ELSE rcn:=NewR(); <rcn:=r+adj[r]>

index [r]                   IF x#nil THEN error
                            ELSIF adj[r]=0 THEN x:=r
                            ELSE x:=NewR(); <x:=r+adj[r]>

register deferred (r)       let d=0 in base

base d(r)                   let dd=d+adj[r]
                            IF dd small THEN Kn:=M, mn:=dd(r), Index
                            ELSE <rcn:=dd.msb; rcn:=rcn lsh 16;
                                  rcn:=rcn+dd.lsb; rcn:=r+rcn>,
                                 Kn:=M, mn:=0(rcn), Index

base deferred @d(r)         let dd=d+adj[r], Kn:=M; mn:=0(rcn),
                            IF dd small THEN <rcn:=(r+dd)^>, Index
                            ELSE <rcn:=dd.msb; rcn:=rcn lsh 16;
                                  rcn:=r+rcn; rcn:=(rcn+dd.lsb)^>, Index

auto-decrement -(r)         adj[r]:=adj[r]-2**s, Kn:=M, mn:=adj[r](r), Index

auto-increment (r)+         Kn:=M, mn:=adj[r](r), Index, adj[r]:=adj[r]+2**s

auto-increment deferred <rcn:=(r+adj[r])^>, Kn:=M, mn:=0(rcn), Index,
@(r)+                       adj[r]:=adj[r]+2**s

Index:  IF x=nil THEN   nothing
        IF x#nil AND i#0 THEN
                            rx=NewR(), <rx:=x lsh i; rcn:=rx+mn.r>,
                            mn:=(mn.d+(adj[x]*2**s))(rx)
        IF x#nil AND i=0 THEN
                            <rcn:=x+mn.r>, mn:=mn.d(rcn)
**** What if the displacement overflows 16 bits?
```

Table 4.1: Operand specifier handling

Once T has each OS encoded (and has generated any instructions needed to make it possible to encode the OS in one of these forms), it interprets some opcode-specific commands to generate the instructions that do the work. These take one of the following forms. Here n refers to an OS and r is a register.

```
Read(n, r, s):  IF Kn#M THEN nothing.
                IF Kn=M THEN
                        rcn:=r,
                        IF s=2 THEN <r:=mn^>,
                        ELSIF s=1 THEN
                          <ttn:=mn^W; r:=ttn[8*BN+16..8*BN]>,
                        ELSIF s=0 THEN
                          <ttn:=mn^B; r:=ttn[8*BN+8..8*BN]>,

ReadR(n, r):    IF Kn=C THEN <r:=rcn> ELSE Read(n, r)
```

This is used for an operand which is going to be the second operand of Operate, and hence can't be a short constant (Titan allows only the first operand to be a constant).

```
Addr(n, r)      IF Kn=M THEN <r:=mn>, rcn:=r
                ELSE reserved operand fault

Do(op,n1,n2,n3) <n1:=n2 op n3>

Write(n, r)     IF Kn=R THEN <rcn:=r>
                ELSIF Kn=C THEN reserved operand fault
                ELSIF Kn=M THEN <mn^:=r>
```

Finally, T generates TIs to adjust any registers for which adj#0.

Here are some examples of simple opcodes.

MOVL E.g.,	MOV r5,r7	MOV m1,r7	MOV r5,m2	MOV m1,m
Read(1, rc2)	---	r7:=m1^	---	t1:=m1^
Write(2, rc1)	r7:=r5	---	m2^:=r5	m2^:=t1

ADDL E.g.,	ADD r5,r7,r9	ADD m1,r7,r9	ADD r5,r7,m3	ADD m1,m
Read(1, rc1)	---	t1:=m1^	---	t1:=m1^
Read(2, rc2)	---	---	---	t2:=m2^
Op(ADD, 1,2,3)	r9:=r5+r7	r9:=t1+r7	t3:=r5+r7	t3:=t1+t
Write(3, rc3)	---	---	m3^:=t3	m3^:=t3

Rather than generating TIs to do the work, T may generate calls on extracode, vanilla VR instructions that handle lengthy operations. These are just ordinary VR subroutine calls, to ordinary VR code which resides at fixed addresses in system space, and must be present for VAX emulation to work. Presumably all the VAX instructions which are interpreted by the MicroVAX would be done in extracode, together with page fault and perhaps other trap handling. The linkage to extracode is 2 cycles at most, and only 1 cycle if there is anything that can be put after the call instruction. Hence it may be desirable to put a lot of other things into extracode. For example, the byte and 2-byte write sequences are quite lengthy, and come at the end of the instruction; they could be put into extracode. There are two advantages to putting something into extracode:

> the size of T is reduced;
>
> less space is taken in the cache for the TIs (of course we must pay for the space taken by the extracode, so it isn't useful unless more than one cached instruction is using it).

The obvious drawback is the cost of the linkage. In addition, it is difficult if not impossible to parameterize extracode with register numbers, so all parameters must be moved into fixed registers, or there must be different versions of the extracode for each register combination allowed.

5. Invalidating TIs

This is the worst problem with the VOR. Whenever a store is done, we must make sure that any translation which could be affected by the store is invalidated. Stores can some from the processor itself, from another processor, or from an i/o device. The latter, unfortunately, is quite common when code page frames are reassigned and overwritten.

The basic difficulty is that a cache invalidation needs to be done on a fairly large unit, typically a cache line or 16 bytes. Unfortunately, the VOR cache may have as many as 4 lines occupied by TIs tagged with a single VAX PC byte address. Since the last three lines can only be reached by passing through the first, and since the VAX architecture allows the effect of a store on the instruction stream to be deferred until the next execution of REI or whatever, it is sufficient to invalidate only the first line. So a total of 16 lines would have to be examined. The straight-forward implementation of this scheme will be much too slow.

If the tags are implemented in VLSI, we can have an associative lookup which would check every line in a single cycle. Then there will be no problem with invalidation. If the tags are implemented in the usual way with of-the-shelf RAMs, however, this method won't work and we must be more clever. If you are only interested in a VLSI VOR, the only reason to read the rest of this section is to convince yourself that the cache tags should be built with associative lookup on the real address.

I have considered a number of invalidation schemes, including several that track down invalid TIs in the cache and zap them. I have concluded, however, that it's too hard to make this method have good performance, in the light of the fact that the reuse of code pages means there are a lot of stores that invalidate TIs. As a result, I am proposing a scheme based on sequence numbers.

The basic idea is to keep a sequence number SN for each page that contains VI bytes which have been translated into TIs that are in the cache, and also for each cache line containing TIs. Whenever a TI is fetched, the SNs are compared; if they don't match, there is a miss. Whenever a store is done into the page, its SN in incremented; because of the previous sentence, any TIs for the page that are in the cache are automatically invalidated. We keep these SNs in each TLB entry and each cache entry, and compare them on each cache lookup just as we compare the real addresses.

When a store is done, we may have only the real address (if it is done by another processor or an i/o device). Hence we must be able to find the TLB entry from the real address. This is done with another direct-mapped cache called RT (for Real address to TLB index). It is a table of suitable size, say 1024 entries; each entry holds a TLB entry address. Of course not all pages can have valid RT entries, i.e. ones that point to a TLB entry whose real page number is that of the page.

To account for this, we introduce the idea of hot pages. A hot page is one whose TLB entry

has a valid SN, and
can be reached from RT.

When a page becomes hot, it gets a suitable SN assigned, guaranteed not to match anything now in the cache. Any store makes a page cold; external stores find the TLB entry through RT, and this always works for a hot page. An attempt to fetch TIs that have bytes in a page will cause it to be made hot. When a page becomes hot, it must have a RT entry; this will displace the page currently using that RT entry, and it becomes cold (if it wasn't already).

If we keep assigning sequence numbers to pages when they become hot, we will run out eventually. Therefore we run a background Cleanup process (implemented with a small finite-state machine, most likely) which scans the cache and erases old SNs so they can be reused. To avoid losing valid TIs in this process, we assign SNs in cycles. At any time there is a current cycle, a next one and a last one. Cleanup removes each last cycle SN from the cache; if it agrees with the corresponding TLB entry (found through RT), then it is still valid and Cleanup updates it and the TLB's SN to the current cycle. Otherwise the SN is zapped. During the cycle new SNs are assigned from next cycle. At the end there are no cache entries with last cycle SNs, so these SNs can be reused.

It is still possible to run out of next cycle SNs if Cleanup gets too far behind. In this case TIs are executed directly and not cached; this will slow things down several times, but should be rare. The idea is to run Cleanup during cache misses, when the cache is idle for several cycles. There should be time to clean about 4 lines per miss. Since we can't assign more than one SN per miss, if SN (not counting the 2-bit cycle part) has 2 less bits than the cache line number we will never run out. In practice 5 or 6 bits should be plenty.

The details of the SN scheme and the invariant that it maintains are given by the program in the Appendix. It is written in Modula 2; if you know Pascal, you should have no trouble with it.

Page crossings

We also have to worry about the fact that a VI can span two pages, and in general the translation can be invalidated by stores into either page. This is especially tricky since the cache does business in real address, where there is no relationship between the pages. The solution, however, is simple: when T sees a page crossing during instruction decode, it immediately generates a VR branch to the first pseudo-address in the new page. Stores into the first page will invalidate the initial TIs of the VI; stores into the second page will invalidate the TIs branched to. As long as no irreversible changes are done before the branch, all will be well.

If the VAX ever branches to the first byte of the new page, it will execute the second half of the translation; this would be bad, and something must be done about it.

Performance hits

In addition to being ugly and complicated, the SN scheme has some performance costs:

> RT misses make TLB entries cold, losing any TIs on those pages.
>
> TLB misses displace TLB entries, losing any TIs on those pages.
>
> If we run out of SNs during a cycle, no more TIs can be cached until the cycle is over; direct execution is slower.

Handwaving arguments suggest that these costs are small with a TLB and RT of adequate size. It is probably desirable to make the TLB remember entries for processes other than the one currently running, as the Titan does, since losing the TLB on a context switch not only means that it must be reloaded, but also that all the TIs must be regenerated.

In addition the main path from the VR processor to the cache is somewhat complicated. For reference we summarize all the complications introduced by VOR:

> A cache line has about 20 extra bits, for the pseudo-address and the SN, and where and when. The cache tag comparison is about 15 bits wider, about 36 instead of 21 bits. The two-bit cycle number must be compared with both the current and the previous cycle number.
>
> A TLB entry has about 10 extra bits, for hot and the SN.
>
> There is another input to the cache line addressing, for the pseudo-address. We can't just send the logical pseudo-address as though it were a real address, because too many cache lines would correspond to TIs beyond four per VI, which are seldom used.
>
> Cleanup must be able to address the cache and read the SN and tag.

Details of an implementation are needed to determined whether any of these factors will actually increase the cycle time.

6. Other issues

This section comments on a number of other issues connected with the VOR.

6.1. Page size

The Titan pipeline depends on the fact that TLB lookup and cache lookup can be done in parallel. This requires that the low-order address bits used to address a cache set agree in the virtual and the real address. The simple way to achieve this is to make the page size big enough. Since this is not compatible with the VAX architecture, we need another way.

The trick is to optimize for the case when enough bits do agree, by addressing the cache immediately with the low-order virtual address. When the cache tags and the real page number have been read out, we compare the dubious bits of VA and RA in parallel with comparing the tags and msb of real address. If both agree, there is a hit. If the dubious VA and RA bits don't agree, we take another cache cycle (causing a one-cycle stall), this time using the real address just obtained from the TLB.

To take advantage of this, the operating system needs to ensure that usually enough bits of VA and RA agree. How many is "enough?" With a 64KB direct-mapped cache (as on Nautilus), we need 16 bits to address the cache. The current page size is 9 bits, so 7 more bits must agree (the easy way would require a 64 KB page size!). With a 1 MB main memory, this means there are 16 places (2**20/2**16) where a page with given VA can go. This is a little marginal, but so is 1 MB. With a more realistic 10 MB, there are 160 places where a page can go. So the need for an extra cache cycle should be infrequent. Of course, if the same page appears in different places in two maps, one must lose out. But the penalty is not too severe.

6.2. Why not static translation?

I considered a scheme in which VAX code is translated in larger chunks, say a page at a time, and cached in main memory. This has the advantage that no hardware T is required, and the VR and its cache don't have to deal with pseudo-addresses. Also, much more optimization of the translated code is possible. However, I can't see any way to do this without losing a lot of transparency. The main reason is that large amounts of main memory are required, since the translation is quite expensive and hence can't be done very often. It seems best to think of the translation as being done during page-fault processing. Also, large auxiliary tables are needed. There isn't much hope of concealing a six-fold increase in the space needed for code from the operating system.

On the other hand, the proposal made by Dick Sites seems very sound: to provide a tool which does automatic or semi-automatic translation of most VAX programs to RISC instructions. This translation would be done only once, and the translated program stored in the file system in place of the original. If the automatic translator can't handle a program, it can still be run under emulation. In this way many more programs can be quickly converted to run in RISC native mode, and hence faster.

6.3. Compatibility

The goal of the VOR is VAX compatibility as good as the MicroVAX; i.e., there needs to be code to handle instructions and traps not implemented in hardware, but this is done transparently to normal execution.

The only area I know of where this may not be feasible has to do with detailed rules for the order in which page faults from instruction fetches and from data references happen. VOR wants to translate the entire instruction before beginning execution. I understand that the current VAX spec demands that a fault from a data reference early in the instruction must happen before a fault from a reference to the rest of the instruction. Hopefully no one will regard this as important.

I would be interested to know of other areas in which it may be difficult or impractical to achieve complete compatibility.

6.4. Exceptions

VAX has a large assortment of exceptions which must be emulated. The general strategy is similar to that used at the microcode level in existing implementations. Interrupts are turned into VR interrupts (perhaps fewer of them), and VR code turns them back into VAX interrupts if appropriate. Traps like decimal overflow and subscript range are detected explicitly by the TIs or extracode and the appropriate state changes generated. This is also done for integer overflow, and for the trace trap, using TI sequences conditioned by whether these traps are enabled (see T3 and T4 above). Faults from malformed instructions, such as reserved operand, are detected by T and give rise to TIs which explicitly call extracode to report the fault. ASTs are checked for explicitly on ring crossings by extracode.

6.5. TI misses after the first line

An attempt to fetch a TI can get a miss on any cache line, not just on the first one. The translator probably can't start up a translation in the middle. So it starts at the beginning (remember that the VPC of the VI is easily computed from the pseudo-address of any TI), generating the entire translation, and then tells VR to continue. Some extra work may be done, but there are no complications.

7. What next

What should be done to pursue the idea of a VOR? I think the next steps are these.

1) Write the VR code for all the VAX instructions (except perhaps the really infrequent ones). This will give essential information for calculating performance and cache size.

2) Using this information, simulate the operation of the cache and T for some VAX instruction traces, and calculate the miss rate as a function of cache size, as well as the average number of VR cycles per VI.

3) Examine VR implementations in MCA2 and CMOS. This should be only a minor extension of vanilla RISC implementations, except for the implementation of the translator, and the need for a somewhat larger cache.

Appendix. Implementation details

```
(* Cache and translation buffer implementation for VOR *)
(* A * marks lines which implement the sequence number scheme
   for invalidation. *)

CONST
  NTLBIs=1024;
  NLines=4096;
  BytesPerWord=4;
  Words ~~Bytes~~PerLine=4;
  BytesPerPage=512;
  NRealPages=2**21;
  TIsPerVI=16;
  RTSize:=1024;
  MaxN:=64;

TYPE
  TLBI=[0..NTLBIs-1];                     (* TLB index *)
  LI=[0..NLines-1];                       (* Cache line index *)

  Address=[0..2**32-1];
  TLBAddress=RECORD (* one word, lsb first, to match Address *)
    byte: BITS 9 FOR [0..BytesPerPage-1];
    i: BITS 10 FOR TLBI;
    residue: BITS 13 FOR TLBResidue;
    END;
  TLBResidue=[0..((LAST(Address)+1) DIV BytesPerPage) DIV NTLBIs-1];

  RealAddress=RECORD (* one word, lsb first *)
    byte: BITS 9 FOR [0..BytesPerPage-1];
    page: BITS 21 FOR RealPage;
    fill: BITS 2 FOR NULL;
    END;
  RealPage=[0..NRealPages-1];

  CacheAddress=RECORD (* one word, lsb first, to match RealAddress *)
    byte: BITS 2 FOR [0..BytesPerWord-1];
    word: BITS 2 FOR [0..WordsPerLine-1];
    line: BITS 12 FOR LI;
    residue: BITS 14 FOR CacheResidue;
    fill: BITS 2 FOR NULL;
    END;
  CacheResidue=[0..((LAST(RealPage)+1)*BytesPerPage) DIV
                     (BytesPerWord*WordsPerLine*NLines)];

  PAExtension=RECORD flag: BOOLEAN; TIAddress: [0..TIsPerVI-1] END;

* SN=RECORD y: Y; n: N END;               (* sequence number *)
* Y=[0..3];                               (* cycle of cleanup *)
* N=[0..MaxN];
```

```
VAR
* li: LI;                                 (* the li being cleaned up *)
* cy: Y;                                  (* current cycle of cleanup *)
* nMax: N;                                (* largest N used in cy+1 *)

  c: ARRAY LI OF RECORD                   (* the cache)
*   sn: SN;
    dirty: BOOLEAN;
    residue: CacheResidue;                (* the vanilla tag *)
    pa: PAExtension;                      (* extra tag for pseudo-addrs *)
    data: ARRAY [0..WordsPerLine-1] OF WORD;
    END;

  tlb: ARRAY TLBI OF RECORD               (* the TLB *)
*   sn: SN;
*   hot: BOOLEAN;                         (* must = TRUE to execute TIs *)
    realPage: RealPage;
    residue: TLBResidue;
    protection: Protection;
    modified: BOOLEAN;
    END;

* rt: ARRAY [0..RTSize] OF TLBI;          (* map hot RealPages to TLBIs *)


 (*
 Invariant:
 (* Main part, which says that invalidation works *)
   FOR ALL i IN TLBI: tlb[i].hot =>
     FOR ALL li IN LI:     c[li].pa.flag (* line holds TIs *)
                       AND RealAddressFromLI(li).page=tlb[i].realPage
                       AND Matches(c[li].sn, tlb[i].sn)
                       => c[li].data was translated from the current
                          contents of tlb[i].realPage
 (* Auxiliary part, internal to the implementation of the SN scheme *)
     AND RTLookup(tlb[i].realPage)=i  (* hot page is reachable via rt *)
  AND (* SNs in the cache are getting cleaned up properly *)
   FOR ALL li' IN LI: c[li].pa.flag (* i.e., the line holds TIs *) =>
        c[li'].sn.y=cy-1 AND li'>=li     (* not yet cleaned up *)
     OR c[li'].sn.y=cy                   (* not translated this cycle *)
     OR c[li'].sn.y=cy+1 AND ( c[li'].sn.n<nMax OR EXISTS i IN TLBI:
                      (    RealAddressFromLI(li').page=tlb[i].realPage
                       AND tlb[i].hot
                       AND c[li'].sn.n<=tlb[i].sn.n ) )
 *)
```

```
(* InternalStore, ExternalStore and FetchTI are the interface procedures
   Cleanup runs concurrently *)

PROCEDURE InternalStore(a: Address; d: WORD);
VAR
  ta: TLBAddress; i: TLBI;
  ra: RealAddress; ca: CacheAddress; li: LI;
BEGIN
  ta:=TLBAddress(a); i:=ta.i;
  IF tlb[i].residue#ta.residue THEN (* TLB fault *) ... END;
* IF tlb[i].modified=FALSE THEN (* Set modified *) ... END;
  IF tlb[i].hot THEN Chill(i);
  ra.page:=tlb[i].realPage; ra.byte:=ta.byte;
  ca:=CacheAddress(ra); li:=ca.line;
  IF c[li].pa.flag OR c[li].residue#ca.residue THEN (* Miss *) ... END;
  c[li].data[ca.word]:=d;
END InternalStore;

PROCEDURE ExternalStore(ra: RealAddress; d: WORD);
VAR ca: CacheAddress; li: LI;
BEGIN
(* Invalidate any TIs in the page being stored into *)
  i:=RTLookup(ra.page);
  IF tlb[i].realPage=ra.page AND tlb[i].hot THEN Chill(i) END;
(* Update the cache with the data being stored; it could also be
   invalidated, in which case we wouldn't need the d parameter *)
  ca:=CacheAddress(ra); li:=ca.line;
  IF c[li].pa.flag=FALSE AND c[li].residue:=ca.residue THEN
    c[li].data[ca.word]:=d;
  END;
END ExternalStore;

PROCEDURE FetchTI(a: Address; x: PAExtension): WORD;
VAR
  ta: TLBAddress; i: TLBI;
  ra: RealAddress; ca: CacheAddress; li: LI;
BEGIN
  ta:=TLBAddress(a); i:=ta.i;
  IF tlb[i].residue#ta.residue THEN (* TLB fault *) ... END;
* IF Warm(i)=FALSE THEN (* Can't cache TIs *) ... END;
  ra.page:=tlb[i].realPage; ra.byte:=ta.byte;
  ca:=CacheAddress(ra); li:=ca.line;
  IF c[li].residue#ca.residue OR c[li].pa#x
*    OR NOT Matches(li, i) THEN (* Miss *)
    IF c[li].dirty THEN FlushLine(li) END;
    c[li].pa:=x; c[li].residue:=residue; c[li].sn:=tlb[i].sn;
    ... ; (* Fill in c[li].data with translated instructions *)
  END;
  RETURN c[li].data[ca.word];
END FetchTI;
```

```
PROCEDURE Warm(i: TLBI): BOOLEAN;
(* Call before generating TIs from VI bytes in page tlb[i].realPage *)
(* POST result=tlb[i].hot; i.e., returns FALSE if it couldn't *)
VAR j: [0..RTSize-1];
BEGIN
  IF tlb[i].hot THEN RETURN TRUE;
(* Now we are sure to have a miss, since we are bumping the SN *)
  j:=tlb[i].realPage MOD RTSize;
  IF tlb[i].hot THEN Chill(rt[j]) END; rt[j]:=i;
  tlb[i].sn.y:=cy+1;
  IF tlb[i].sn.y#cy+1 THEN tlb[i].sn.n:=1;
  ELSIF nMax<LAST(N) THEN tlb[i].sn.n:=nMax;
  ELSE (* ran out of SNs in this cycle *) RETURN FALSE;
  END;
  tlb[i].hot:=TRUE; RETURN TRUE;
END Warm;

PROCEDURE Chill(i: TLBI);
BEGIN
  IF tlb[i].sn.y=cy+1 THEN nMax:=MAX(nMax, tlb[i].sn.n+1) END;
  tlb[i].hot:=FALSE;
END Chill;

PROCEDURE Cleanup();
VAR ra: RealAddress; i: TLBI;
BEGIN
 <cy:=cy+1 MOD (LAST(Y)+1); li:=0; nMax:=1; (* 0 never used in TLB *) >
  REPEAT
    IF c[li].sn.y-cy-1 THEN
      c[li].sn.y:=cy;
      ra:=RealAddressFromLI(li); i:=RTLookup(ra.page);
      IF tlb[i].realPage=ra.page AND Matches(li, i) AND tlb[i].hot THEN
        (* the cache entry is still valid *)
        IF  c[li].sn.y=cy-1 THEN  c[li].sn.y:=cy;
        IF tlb[i].sn.y=cy-1 THEN tlb[i].sn.y:=cy;
      ELSE c[li].sn.n:=0; (* zap the entry *)
      END;
    END;
    li:=li+1;
  UNTIL li>LAST(LI);
END Cleanup;

PROCEDURE Matches(li: LI; i: TLBI): BOOLEAN;
BEGIN RETURN
       c[li].sn.n=tlb[i].sn.n
  AND (c[li].sn.y=tlb[i].sn.y OR c[li].sn.y=cy AND tlb[i].sn.y=cy-1)
END Matches;

PROCEDURE RealAddressFromLI(li: LI): RealAddress;
VAR ca: CacheAddress;
BEGIN
  ca.byte:=0; ca.word:=0; ca.line:=li; ca.residue:=c[li].residue;
  RETURN RealAddress(ca);
END RealAddressFromLI;

PROCEDURE RTLookup(rp: RealPage): TLBI;
BEGIN RETURN rt[rp MOD RTSize] END RTLookup;
```