Ideas for a simple fast VAX

B.W. Lampson
20 October 1985

Six ideas are put together here to make a design for a 4 tpi VAX with a very short cycle time, whose basic execution engine is simpler than Nautilus. Other parts of the design will be at least as complex, so that the whole machine will not be any simpler; in fact, if it has high-performance floating-point, there will be substantial added complexity for that. The hope is that the other parts are not critical to the performance, and hence can be designed without having to squeeze out every nanosecond.

THE IDEAS

1) Cache the microinstructions for each VAX instruction as it is decoded. Next time the VAX instruction is executed, these microinstructions (called TIs, for translated instructions) can be executed without any decoding. The first TI for a VI has a cache address computed from the PC, so a branch can be translated into the proper value for the NEXT field of the TI. Thus unconditional branches take no time. Conditional branches are delayed one cycle; normally this cycle is wasted.

I proposed this idea earlier, using a RISC as the translation target, rather than microcode; this was called VOR. That design proposes how to do the translation, and how to invalidate it if a store is done to the VI, using sequence numbers. Thos ideas apply to this proposal too. The VOR design also deals with how to get by using a RISC very much like the Titan; that is not necessary in the new proposal. However, it is necessary to use fairly short micro-instructions (say 50 bits), so there can't be too much more in a microinstruction that what the Titan or Prism has, plus a NEXT field.

Since there is only a 2.5% miss rate for instruction fetch with a Nautilus-sized cache, translation doesn't have to be blindingly fast. I assumed 12 cycles in the performance calculation, 4 for the cache miss and 8 for the translation, but slowing translation down 4 cycles only adds 0.1 tpi.

There's a question about how to allocate TIs beyond the first for a VI. Looking at some Nautilus cache simulations that show <20% of the cache filled at the 5000 VI context-switch point suggests a FIFO buffer. If the Nautilus cache is 50% instructions, and one VI is 2.5 TIs (see the performance section below), we get 16k x 20% x .5 x (2.5-1) = 2.5 k of excess TIs. So with a 4k buffer, we could afford to flush the whole TI cache when it filled up. This is attractively simple. But there are many other possibilities; one is given in the VOR design, but it has a lot of fragmentation.

2) Use a two-level cache in the processor, backed up by the very large, 4 cycle cache already proposed for Argonaut. The two-level cache has a 1k x 32 fast part, and a 16k x 32 slow part. The fast part goes in 1 cycle, the

slow part takes 2. The idea is that every address is presented to both parts; if the fast part misses, the slow part sees the address for 2 cycles and delivers the data, which is written into the fast part at the same time. Thus it costs 1 cycle for every fast miss. A slow miss costs 4 cycles at least. Writes are done to both parts, so back-to-back writes cause a 1-cycle stall, as does a fast miss in the cycle after a write.

The TI cache is arranged the same way, but 50 bits wide. The slow part also stores fixed microcode, which is hand programmed rather than generated by the translator to do complicated things like TB misses, character instructions, etc. If the translation isn't there, I assume that the VAX instructions are fetched from the 4-cycle cache and retranslated. The slow part might want to be 64k x 50, and 64k x 1 RAMs might be fast enough.

The advantage of this scheme is that it allows the fast cache to be fairly small, 4 KB, hence both faster and more compact that otherwise. Made out of 1k x 4 RAMs, the RAM delay is 5 ns (7 ns ?), rather than 10 ns for a 16k RAM, and it's a quarter the number of chips needed for a 64 KB cache. Note that a miss rate of R in the fast cache means that every cache reference is slowed down by R cycles. If the faster small cache allows the cycle time to be faster by R, we break even. Thus a 20% miss rate in the fast cache leads to a faster machine if the cycle time can be less that .8 of what it would be with a bigger single-level cache. And this argument is very conservative: it assumes a cache reference, with associated chance to miss, in every cycle. Of course there will be a TI cache reference in every cycle, but that isn't true for the data cache, and its timing is probably more critical.

3) Use a virtual cache. Keep a sequence number (SN) in the TB and in the cache entry, and compare these as part of the cache tag comparison. Whenever the TB entry is reassigned, the SN is incremented. This ensures that only cache entries with a TB entry are valid. In addition, keep a reverse TB (RTB) that maps from real to virtual addresses. On a TB miss, the page must be put into RTB also. This may displace a TB entry already in RTB; in that case the SN of that TB entry is incremented to invalidate any cache entries that depend on it, and it is removed from the TB. The RTB needs to be big enough that it has little effect on TB miss rate. However, it doesn't need to be fast, since there is at most one access per write on the NMI, plus 1 per TB miss.

Perhaps there is another way to get a virtual cache. This one seems simple.

The remaining three ideas are mainly relevant for fast floating-point.

4) Have a separate F-box with its own copy of the registers, organized for convenience in doing floating point. This might mean separate register sets for F and G formats, for example. Keep track for each register of which register set (E, F or G) has a valid copy. After a clear, all will; after any other operation only one will. Have explicit TIs to move data from one register set to another. The TIs refer to a specific register in a specific register set, e.g. E3 or G7. If a TI references a register that isn't valid, force retranslation. The translator knows which registers are valid, and generates the necessary explicit MOV TIs to get data where it's needed.

Obviously it is necessary to validate my assumption that forced retranslation for this reason would be rare; note that it only happens when the same VAX instruction reads a register that is sometimes loaded by an integer operation, and other times by a F or G operation.

This scheme also allows F and G results to be maintained separately, again with explicit MOVs generated in the (hopefully non-existent) cases where a register pair set by a G instruction is referenced by an F instruction. Thus there can be 16 64-bit G registers; when R6/R7 is loaded from a G instruction, the G6 register would be loaded. G7, F6 and F7 would become invalid, as well as E6 and E7.

The F-box can have its own data cache, 64 bits wide. Thus all its operations can proceed at full speed, without the need to serialize data transfers on a 32-bit bus, or to copy data back and forth to the E registers in the normal case that it isn't referenced by any non-floating instruction. The implementation proposed below has only data RAMs in the F-cache -- all the tags and control are in the E-box data cache.

The stores done by CALL and context-switching will require some special treatment. It is possible that other loads and stores need more careful handling also if the compilers move floating data with L or Q instructions rather than F or G instructions; hopefully they don't.

With this F-box we could do pretty good vector instructions too.

5) Implement condition code bookkeeping that allows the processor to go ahead even though an instruction that sets the CC hasn't completed, and forgets about the deferred CC setting if a later instruction writes the same CC. I have worked this out, and it doesn't look too messy. Basically, there is propagate logic on each CC bit which keeps track of whether the bit has been set since the instruction i cycles back; if not, then when that instruction reports a value, that becomes the CC value. The resulting structure looks like the carry propagation of an adder. Of course if someone tests the CC there must be a stall.

6) Take advantage of the fact that with a write buffer and a changed-register log longer than one instruction it is possible to undo the effects of execution as far back as the capacity of the write buffer and the log, even in a multiprocessor. The only hard constraint is the execution of synchronization instructions, or other operations that require the write buffer to be flushed. These are rare. Thus execution of floating point need not be delayed for the precise exceptions. If an exception does occur, the instruction can be retried.

For whatever it's worth, this observation also allows aother instruction sequences to be retried even if their errors are detected long after execution was complete. Some people seem to think that this is valuable.

IMPLEMENTATION

The goal is to get the shortest possible cycle time consistent with doing a Titan-like instruction in each cycle, with a pipe of length 3:

        instruction fetch
        register read, address add, start ALU
        cache read/write, finish ALU, register write

This is the same timing as the Titan. To achieve it, the execution engine has to be compact. I suggest the following arrangement:

```
 _______ <-ctl-- _______ <--BR-- _______ <-----------CC-----------  _______
|       |       |   X   |       |       |                          |       |
|   T   |-TIs-->|_______|--TIs->|   E   |<----------FE----------->|   F   |
|       |                       |       |                          |       |
|       |_______     ___________|       |________    ______________|       |
|_______|       |   |           |_______|        |  |              |_______|
    ^          PA   VA              ^            D  D                  ^
    |           |   |               |            |  |                  |
    |           V___V_              |            V__V_                 |
    |          |      |             |           |     |                |
    |          |  A   |             |           |  W  |                |
    |          |______|             |           |_____|                |
    |              ^                |              |                   |
    |              |                |              |                   |
    D              A                D              AD                  D
    |              |                |              |                   |
____|______________V________________V______________V___________________V
------------------------------------------------------------------------
                          NMI
```

Here the translator T plays the role of the I-box, A holds tags, most of the TB, and other non-data memory-related stuff, and W is the write buffer.

We try to put in E everything required for execution of non-floating instructions in the normal case of no misses. This is the following:

the data part of the fast TI cache	50 bits	13 1k x 4 RAMs
the data cache, fast and slow	33 data bits	9 1k X 4 RAMs
		9 16k x 4 RAMs
	32 tag bits	8 1k X 4 RAMs
	(addr + SN)	8 16k X 4 RAMs
		47
the register file	32 x 32 regs	4 MCAs ??
the ALU and shifter		
the TI branch logic		

The intention is for this to fit on one module. The tags for the fast TI, and the slow TI, are on X. The idea is that E plunges ahead fearlessly fetching TIs from its fast cache. If X sees that the tag doesn't match, it signals E to stall before the end of the execution pipe, when the registers are written. Then it supplies a new TI, after 1 cycle if it comes from the slow TI cache, much later if retranslation is needed. So E provides do-branch information to X, and gets told what to do on any kind of miss. Since X has to keep its idea of the TI PC up to date, it has to contain a copy of part of the fast TI data, namely the NEXT field.

There might be a hiccup on a taken branch if the fast TI misses (i.e. a 2 cycle rather than 1 cycle stall), but this would be a fairly good tradeoff: an extra cycle when a miss (20%) coincides with a taken branch (1/tpi x 25% < 10%), or <2%.

F gets instructions from E, and therefore runs either 1/2 cycle or 1 cycle behind. This hurts only on branches that test CC set by F. X keeps track of CC validity, since it knows all the timing and can stall E when needed. A memory write done by F must invalidate E's cache, and vice versa; since E decodes all the instructions and contains all the cache tags, this should be easy. The idea is to just have the data in the F cache, keeping F simple and avoiding complications about multiple copies. So normally the instruction and the 14 or so bits of cache address are sent from E to F on the FE bus. This bus is also used to transfer data back and forth, using explicit MOV TIs.

PERFORMANCE

The following attempts to calculate the performance of this design from the data in Clark and Emer. The numbers that seem questionable to me are indicated. What has been overlooked?

From C&E table 2, percent of all instructions that branch unconditionally, or can be assumed to branch with high probability (i.e. loop branches), and for which the branch address is known at translation time:

Simple	2	
loop	3.7	
lowbit	0	
subr	2.3	calls only
JMP	0.3	
CASE	0	
bit	1.9	
proc	1.2	calls only
system	0.2	CHM only
	----	
total	9.7	

Taken conditional branches and computed branches is 25.7-9.7=16.0
This number is used in table 8 below.

From C&E table 4, execution time for specifiers:

specifier	# TIs	time per avg specifier	
literal	0	0	
immediate	1/2	0.03	? >16 bits will take 2
displ	1/2	0.31	25% are >16 bits and take 2 TIs
reg deferred	1	0.09	
auto-inc	2	0.04	
disp deferred	2	0.05	
absolute	1/2	0.01	? >16 bits will take 2
auto-inc def	3	0.01	

auto-dec	2	0.02
indexed	1	0.06
		----
total		0.62

.62 x 1.5 = .93 cycles/VI for specifiers
10 x .016 = .16 cycles/VI for unaligned ops

These numbers are used in table 8 below.

From C&E table 8

Ins class	frequency, %	TI/occurrence	cycles/VI (not counting stalls)	
Decode	100	0	0	.2 2-cycle Decode
specifiers	150 [1.5/VI]	0.62 [table 4]	0.93	
unaligned refs	1.6	10 [made up]	0.16 [table 4]	
B-disp	31	0	0	.08 ← Rotation on B+W Fetches 7% x 1* = .07

Total for specifiers			1.09	
simple	84	0.6 [1]	0.5	
field	7	2 [2]	0.14 ???	
float	4	2.5 [3]	0.1	.2 Nautilus
call/ret	3	15 [4]	0.45	.45
system	2	12 [5]	0.25	.25
character	0.4	60 [6]	0.25	
decimal	0.03	100 [7]	0.03	

Total for direct execution (roughly equal parts simple, call/ret, sys/char)			1.68	
branch latency	16 [8]	1	0.16	
fast TI miss	20 [9]	1 Ta	0.2	.2
retranslation	2.5 [9]	12	0.3	
fast data miss	20 [10]	1	0.2	.3 From 10 cycle cache
slow data miss	2.5 [10]	4 [11]	0.1	
TB miss	1 [12]	20 [13]	0.2	1.7
error in times for system and char ins			0 ??? [5, 6]	
exceptions/aborts			0.1 [14]	

Total for overhead			1.26	
			====	
Total cycles per VI			3.97	

Notes to table

[1] MOV is 40% of simple instructions and takes 0 TIs (except for RR MOVs). Assume other simple VIs take 1 TI.

[2] 1 for extract, 3 for insert, others unclear. More if the field crosses a LW boundary. Much more if the field position and size are not constant. Need more data to get this right.

Write traffic .05 [illegible]/cycle x 4 = .2 [illegible]/cycle

[3] With shadowed registers and accelerated CC handling.

[4] 8 reads or writes on average. At 2 cycles each, the writes in CALL take 16 cycles, or about 18 altogether. At 1 cycle each, the reads in RET take 8, plus 3 to check that the mask hasn't changed, plus 2 for RET from CALLs (half?), or 12.

[5] Half the number of cycles used in the 780. I just made this up.

[6] Half the number of cycles used in the 780, but also 3 cycles per memory operation done. I just made this up.

[7] Same number of cycles used in the 780.

[8] From extension to table 2; these are taken cond branches (other than loops), plus all branches with computed addresses. UC branches with known addresses have 0 latency.

[9] From Steely's Nautilus write buffer memo, giving 20% misses for a 1 KB CVAX cache (assuming that 4 KB of TI cache is worth about as much as 1 KB of VI cache; since the CVAX cache stores data also, this is probably conservative) and 2.5% for Nautilus cache. His 780 numbers agree pretty well with Clark and Grady. The slow miss is 4 cycles to get to the huge cache, and I arbitrarily added 8 cycles for the translation, which is roughly 3 per TI generated.

[10] Clark and Grady for fast miss; they show 83% read hit in the 8 KB cache, 76% in the 4 KB cache on the 780. In both cases the cache is shared with instructions. Same source as [9], Nautilus number for slow miss. A VI makes one data reference.

[11] Assuming the multi-megabyte slow cache hits all the time and takes 4 cycles in addition to the 1 cycle paid for the fast miss that precedes the slow miss. If it misses .5% of the time, and main memory takes 25 cycles, this adds .1 tpi.

[12] Clark and Grady show 1% process TB misses on the 780. It should be possible to get system TB misses very close to 0.

[13] Same number of cycles as 780.

[14] 780 has 0.2, but 2.5 x the tpi. Assuming these go per cycle; not clear how to defend this, however.

General notes on performance

Nearly 25% is in various kinds of miss, comparable to 780.
In more detail, 780 has 1.4 tpi in R/W stall, .7 in IB stall. This has .3 in R-stall and .66 in IB stall.

About .6, or 15%, is in misses that would be drastically reduced with a longer context-switch interval (slow and TB miss). This means that the best we could hope do to with an infinite CS interval is 3.2 tpi. This isn't as important as I thought.

About .4 or 10% is in fast misses. This measures the extent to which the fast part of the two-level cache is imperfect. It means that the fast cache pays off if it allows the cycle time to be reduced by more than 10%.

At 20 ns cycle, we get 27.5 x 780. At 16 ns, it's 34 x.

About .45 of the overhead is in single cycles, i.e. definitely costs more if we have a longer cycle that does more. The rest is in multi-cycle stuff that might take fewer cycles if they were more powerful. Since nearly all of the direct execution is in single cycles, it's clearly best to have the cycle time short, as usual.

How much better could a pure RISC do? We give no credit for a shorter cycle time, or faster execution of RR type instructions, since we have optimized for this.

60% utilization of branch latency cycle (.6 x .16)	.1
4 vs 12 cycle slow miss, since no translation (8 x .025)	.2
90% saving in call/ret, assuming register windows that work 90 % of the time (.9 x .45)	.4
60% reduction in memory references by specifier because of larger number of registers, and register windows for passing arguments (.6 x .9)	.5

Total	1.2 or 30%

I tried to make this favor the RISC as much as possible, to get an upper bound on the performance gain. The 60% reduction in memory operands is consistent with 2 RISC instructions/VAX instruction and a memory reference in about 20% of RISC instructions, reported by a couple of people. But this is very flaky. The guesses that say a RISC is 2x a VAX don't assume this much reduction in memory traffic, I believe.

In addition, the RISC doesn't need the virtual cache complications, or the translator complications, so it is a significantly simpler machine to design. Of course, the VAX doesn't need the virtual cache complications either if the OS would align the pages properly.