outstanding, and matches that of the IBM RS/6000.

12 A case study: DEC Alpha

Introduction

The test machine, a DEC Alpha AXP 3000/500 with a 150-MHz DECchip 21064 CPU, on which these results were obtained was kindly loaned to the author by Digital Equipment Corporation for a two-week trial period during January 1993. The system was running DEC OSF/1 T1.2-2 (Rev. 5), an almost-final release of OSF/1, and had 94MB of memory, 85MB of which was available to user processes.

The DEC Alpha AXP 3000 systems have 8KB of on-chip instruction cache, 8KB of on-chip data cache, and a 512KB external cache.

While some C and C++ compiler problems were encountered in larger programs, the Fortran compiler was found to be solid, and none of the matrix multiply benchmark programs were affected. Some of the compiler problems that I found have already been fixed at DEC development laboratories in newer, and as yet unreleased, versions of the compilers. I provided DEC engineers with the programs that precipitated problems to assist them in fixing the compilers.

The very high clock speeds of the Alpha have astonished the industry, and since this is a completely new architecture, it is of considerable interest to investigate the quality of compiler code generation. RISC machines are difficult systems to write good compilers for, because their performance is critically dependent on reduction of memory accesses, and clever scheduling of instructions to avoid delays and to keep the pipeline and all functional units busy.

Alpha architecture

The February 1993 issue of the journal *Communications of the ACM* has several articles about the Alpha architecture, and about the work done to host the OpenVMS, OSF/1, and NT operating systems on it. Two of these operating systems provide binary compatibility for executable programs from the VAX and MIPS architectures. The issue includes a very informative overview of Alpha by the chief architect [30]. There is also a large book describing the architecture in detail [29].

The Alpha architecture was designed for extremely high performance, and sacrifices in the instruction set design were made to avoid complexities that would slow down the system.

Cray Research, Inc., is building its first massively parallel processor system using Alpha chips. I believe that many industry analysts share the view that Alpha is critical to the continued existence of DEC as a major vendor in the computer business. Personnel cuts of 1/3 of the workforce at DEC and IBM in the last year have not gone unnoticed, and both companies have recently posted their first-ever operating losses.

Here are some key points taken from the Alpha Architecture Handbook [8]:

- All instructions are exactly 32 bits long, and only load and store instructions reference memory.
- The address space is a linear 64-bit byte-addressable one, although current implementations only support 43-bit addressing. This is more than adequate: with current memory costs of about US$30/MB, 1GB costs US$30K, and 2^{43} bytes would cost US$245M.
- To support efficient use of wide memory-bus technology, instructions and data should be aligned on 16-byte boundaries.

  The compiler code listings for the matrix multiply benchmark programs show that the array addresses end in 10_{16}, guaranteeing the recommended 16-byte alignment.
- There are 32 64-bit integer registers and 32 64-bit floating-point registers. In each set, the last register is dedicated to hold a zero value. There are no register windows like the SPARC architecture has.
- While there are integer add, subtract, and multiply instructions, there are no integer divide or remainder instructions, nor is there a reciprocal approximation instruction to facilitate division. The Architecture Handbook states [8, p. A-12] that integer division can be done with an average of 5 multiplies (best case: 1, worst case: 9) using a 1000-entry lookup table.
- Both IEEE 754 and VAX floating-point arithmetic formats are supported: 32-bit S (IEEE), 64-bit T (IEEE), 32-bit F (VAX), and 64-bit G (VAX). VAX 64-bit D format has 3 fewer exponent bits than G, and 3 more fraction bits; it is supported only through software conversion to G format, with concomitant loss of precision. No IEEE 80-bit temporary real format is provided.
- Floating-point instructions include load, store, add, subtract, multiply, divide, compare, conditional move, conditional branch, integer-to-floating-point, and floating-point-to-integer, plus two instructions to load and store the Floating-Point Control Register. The arithmetic instructions are available in both 32-bit and 64-bit versions. Absent are absolute value, remainder, multiply-add, square root, and transcendental functions.
- The conditional move instructions are interesting, and appear to be unique to the Alpha among RISC architectures. Assuming data values in registers, the computations `a = max(b,c)`, `a = abs(b)`, the C conditional expression in the assignment `a = b ? c : d`, and the Algol assignment

`a := if b then c else d` can all be done by three instructions each: a single compare, followed by two conditional moves. Conditional moves eliminate branches, which is extremely important for performance because branches are costly, generally requiring instruction pipelines to be flushed, and sometimes, instruction caches to be refilled.

- Rounding modes supported include round-to-nearest (default), round-to-minus-infinity, round-to-zero, and round-to-plus-infinity. The first three can be specified in the instruction; the last can be achieved by setting an appropriate bit in the Floating-point Control Register and then selecting dynamic rounding in the instruction.

  I could find no compiler options to select alternate rounding modes, so it is not clear whether this is a temporary software limitation, or an incomplete hardware implementation.

  The possible lack of dynamic rounding control on DEC Alpha systems has been discussed on the Numerical C Extensions Group mailing list, but I believe the question is still outstanding. The Architecture Handbook certainly describes it [8, p. 4-56].

- Floating-point instruction execution sequencing need not match instruction sequencing, so when sequencing is needed, such as to localize floating-point exceptions, trap barrier instructions must be inserted to cause pipelines to empty and floating-point status to be brought to a consistent state. Imprecise interrupts are not new with the Alpha architecture; the IBM 360/91 of late 1960s vintage was the first machine to have them.

- IEEE floating-point NaN, Infinity, and denormalized numbers all impose serious execution-time penalties, and are best avoided.

  With the pre-release compilers, I was unable to prevent code from terminating on the generation of a NaN or an Infinity, and denormalized numbers could not be generated at all. However, the Fortran compiler manual pages document `-fpe[0-4]` options that are supposed to allow such control, as they do under ULTRIX on DECstation (MIPS R2000 and R3000) systems.

- Branches are expensive, and loops should be unrolled to get at least 20 instructions in the loop body [8, p. A-3]. Infrequently-executed code should be moved out-of-line (and therefore, probably out of the instruction cache), so as to maximize instruction cache usage.

Analysis of benchmark results

The extensive tables at the end of this report summarize the complete benchmark results in Mflops ratings, and relative to the fastest system in each test.

In this subsection, we analyze the compiler-generated code to determine how good it is relative to the theoretical capabilities of the Alpha architecture, and to measure the average instruction speeds.

A summary of instruction counts for various Fortran compiler optimization levels is presented in Tables 10 through 12.

With the lowest optimization level (Table 10), 55% to 85% of the inner-loop instructions are of type ‘other’, mostly redundant memory accesses and subscript computation. This overhead is probably typical of unoptimized compilation on *any* RISC architecture.

Medium-level optimization (Table 11) reduces the number of loop instructions by a factor of 3 to 4.

High-level optimization (Table 12) with the Alpha compiler produces aggressive loop unrolling, except in the 5×5 algorithm, offering noticeable improvement over lower optimization levels. However, the loop unrolling that is done in all but the 5×5 algorithm becomes detrimental to performance in the 4×4 case, as evidenced by the large numbers of loads and stores as the compiler runs out of available registers. Without the unrolling, that case can be done with only 8 loads and no stores, whereas with unrolling, it takes an average of 11.8 loads and 3.8 stores.

The results of Tables 10–12 should be compared with corresponding numbers for the IBM RS/6000 in Tables 13–15.

To make the numbers in Tables 10–15 easier to understand, Tables 16 and 17 reduce them to the average number of instructions issued for every floating-point multiplication and addition.

In the ideal ‘peak Mflops’ case, *every* Alpha instruction issued would be a floating-point instruction, and the numbers in the columns of Table 16 would all be 2. For the IBM RS/6000, in the ideal case, every instruction is a floating-point multiply-add, and the averages in Table 17 would be 1.

The highest optimization level combined with the $p \times p$ unrolling techniques used in the benchmark programs push this number down to a very respectable 2.9, noticeably better than the 6.5 for the normal inner loop, and 4.5 for the 5-deep unrolled inner loop.

In order to compare with other architectures, particularly the IBM RS/6000 and the HP PA-RISC, we need to ask: *What is the average time per instruction?* Architecture manuals seldom carry this information, and it is best obtained by measurement. These results are collected in Tables 18–19; the numbers in columns 2–4 are computed from

$$\begin{aligned}
\text{cycles/instr} &= (\text{benchmark time in seconds}) \times (150 \text{ cycles} / \mu\text{sec}) \times \\
&\quad (1\ 000\ 000 \mu\text{sec} / \text{sec}) / \\
&\quad (1\ 000\ 000 \times (\text{loop instructions for 1 multiply} + 1 \text{ add})) \\
&= (\text{benchmark time in seconds}) \times 150 / \\
&\quad (\text{loop instructions for 1 multiply} + 1 \text{ add})
\end{aligned}$$

Table 10: DEC Alpha Fortran `matmr8` matrix multiply benchmark code generation: inner loop instruction counts at optimization level `-g` (lowest).

Test	total inst	unroll depth	FP add	FP mult	FP load	FP store	others
normal	38	1	1	1	3	1	32
unroll-5	134	1	5	5	11	1	112
2×2	102	1	4	4	16	8	70
3×3	178	1	9	9	33	15	112
4×4	274	1	16	16	56	24	162
5×5	390	1	25	25	85	35	220

Table 11: DEC Alpha Fortran `matmr8` matrix multiply benchmark code generation: inner loop instruction counts at optimization level `-O2` (medium).

Test	total inst	unroll depth	FP add	FP mult	FP load	FP store	others
normal	11	1	1	1	2	0	7
unroll-5	27	1	5	5	10	0	7
2×2	42	1	4	4	4	4	26
3×3	63	1	9	9	6	9	30
4×4	47	1	16	16	8	0	7
5×5	79	1	25	25	16	6	7

Table 12: DEC Alpha Fortran `matmr8` matrix multiply benchmark code generation: inner loop instruction counts at optimization level `-O4` (highest).

Test	total inst	unroll depth	FP add	FP mult	FP load	FP store	others
normal	26	4	4	4	10	0	8
unroll-5	90	4	20	20	42	0	8
2×2	56	4	16	16	16	0	8
3×3	104	4	36	36	24	0	8
4×4	198	4	64	64	47	15	8
5×5	80	1	25	25	16	6	8

Table 13: IBM RS/6000 Fortran `matmr8` matrix multiply benchmark code generation: inner loop instruction counts at optimization level `-g` (lowest). There are no separate multiply or add instructions generated.

Test	total inst	unroll depth	FP multadd	FP load	FP store	others
normal	31	1	1	3	1	26
unroll-5	91	1	5	11	1	74
2×2	76	1	4	16	8	48
3×3	128	1	9	33	15	71
4×4	192	1	16	56	24	96
5×5	268	1	25	85	35	123

Table 14: IBM RS/6000 Fortran `matmr8` matrix multiply benchmark code generation: inner loop instruction counts with optimization level `none` (medium). There are no separate multiply or add instructions generated.

Test	total inst	unroll depth	FP multadd	FP load	FP store	others
normal	30	1	1	3	1	25
unroll-5	90	1	5	11	1	73
2×2	75	1	4	16	8	47
3×3	127	1	9	33	15	70
4×4	191	1	16	56	24	95
5×5	267	1	25	85	35	122

Table 15: IBM RS/6000 Fortran `matmr8` matrix multiply benchmark code generation: inner loop instruction counts at optimization level `-O` (highest). There are no separate multiply or add instructions generated.

Test	total inst	unroll depth	FP multadd	FP load	FP store	others
normal	4	1	1	2	0	1
unroll-5	16	1	5	10	0	1
2×2	9	1	4	4	0	1
3×3	16	1	9	6	0	1
4×4	25	1	16	8	0	1
5×5	46	1	25	15	5	1

Table 16: DEC Alpha Fortran `matmr8` matrix multiply benchmark code generation: instructions per floating-point multiply/add pair.

Test	Optimization level		
	`-g` (low)	`-O2` (medium)	`-O4` (high)
normal	38.0	11.0	6.5
unroll-5	26.8	5.5	4.5
2×2	25.5	10.5	3.5
3×3	19.8	7.0	2.9
4×4	17.1	2.9	3.1
5×5	15.6	3.2	3.1

Table 17: IBM RS/6000 Fortran `matmr8` matrix multiply benchmark code generation: instructions per floating-point multiply-add instruction.

Test	Optimization level		
	`-g` (low)	`none` (medium)	`-O` (high)
normal	31.0	30.0	4.0
unroll-5	18.2	18.0	3.2
2×2	19.0	18.8	2.3
3×3	14.2	13.1	1.8
4×4	12.0	11.9	1.6
5×5	10.7	10.1	1.8

Table 18: DEC Alpha 3000/500 Fortran `matmr8` matrix multiply benchmark code generation: cycles per inner-loop instruction (cpi) and Mflops ratings.

Test	Optimization level					
	`-g` (low) cpi	`-O2` (medium) cpi	`-O4` (high) cpi	`-g` (low) Mflops	`-O2` (medium) Mflops	`-O4` (high) Mflops
normal	3.60	1.64	2.05	2.18	16.60	22.42
unroll-5	4.73	2.61	2.87	2.35	21.20	23.10
2×2	3.08	0.79	1.66	3.81	36.07	51.49
3×3	2.98	0.99	1.70	5.05	43.14	60.56
4×4	2.80	1.58	1.68	6.24	65.21	57.49
5×5	2.73	1.67	1.61	7.00	55.91	59.73

Table 19: IBM RS/6000-560 Fortran `matmr8` matrix multiply benchmark code generation: cycles per inner-loop instruction (cpi) and Mflops ratings.

Test	Optimization level					
	`-g` (low) cpi	`none` (medium) cpi	`-O` (high) cpi	`-g` (low) Mflops	`none` (medium) Mflops	`-O` (high) Mflops
normal	1.97	1.48	0.73	1.63	2.24	34.31
unroll-5	1.79	1.34	0.92	3.05	4.12	33.92
2×2	1.68	1.30	0.71	3.13	4.07	60.91
3×3	1.61	1.30	0.83	4.36	5.85	66.33
4×4	1.43	1.10	0.92	5.82	7.58	67.85
5×5	1.36	1.15	0.85	6.85	8.58	64.88

Table 20: Best measured `matmr8` performance for DEC, HP, and IBM systems in Mflops. Optimization levels are shown in parentheses.

Test	DEC Alpha 3000/500 150 MHz	HP PA-RISC 9000-720 50 MHz	IBM RS/6000-560 50 MHz
normal	24.215 (`-O3`)	37.315 (`-OP4 -O3`)	35.116 (`-O`)
unroll-5	25.129	26.184 (`-O`)	33.919 (`-O`)
2×2	51.488 (`-O4`)	42.045 (`-O`)	59.706 (`-O`)
3×3	64.527	45.227 (`-O`)	66.333 (`-O`)
4×4	71.634 (`-O2`)	40.888 (`-O`)	67.849 (`-O`)
5×5	61.534 (`-O3`)	30.461 (`-O`)	64.884 (`-O`)

Table 21: Clock speeds for several RISC and supercomputer systems.

Machine	clock rate (MHz)
Cray 3	500
Cray 2	244
DEC Alpha 10000/6x0	200
DEC Alpha 7000/6x0	182
Cray Y-MP	167
DEC Alpha 4000/6x0	160
DEC Alpha 3000/500	150
DEC Alpha 3000/400	133
Cray X-MP	105–117
HP 9000-7100	98
Cray 1	80
IBM 3090-600S/VF	54–66
HP 9000-730	66.7
IBM RS/6000-580	62.5
HP 9000-720	50
IBM RS/6000-560	50
IBM RS/6000-550	42
IBM RS/6000-530H	33
IBM RS/6000-220	33
IBM RS/6000-530	25
IBM RS/6000-320H	25
IBM RS/6000-320	20