CSL Notebook Entry

To:	DoradoUsers, CedarImplementors	Date:	January 19, 1981
From:	Ed Taft	Location:	PARC/CSL
Subject:	Mesa opcode execution times on the Dorado	File:	[Ivy]<CSL-Notebook>80CSLN-YYYY

XEROX

Attributes: technical, Cedar, Dorado, Mesa

Abstract: This memo tabulates the execution time of all common Mesa opcodes on the Dorado. Possible future changes likely to affect these timings are also summarized.

The execution time on the Dorado for all common Mesa opcodes is presented in the table below. This information is intended to be useful to compiler writers in deciding about code sequences to be generated by the Mesa compiler for Cedar. It may also be of interest to others, e.g., in assessing the relative costs of operations on short versus long data values.

The information here is for the PrincOps (Pilot, as opposed to Alto) implementation of the Mesa instruction set. At present, only Mesa 6.0 opcodes are described; this memo will be revised as timing information becomes available on new opcodes implemented for Cedar.

Following the opcode table are some general comments on the timings, along with brief descriptions of possible future changes that are likely to have important effects on performance.

Opcode timings

The timings are expressed in terms of microinstruction execution cycles; with current Dorado clock settings, 1 cycle = 64 nanoseconds. (So, to convert to microseconds, divide the timings by 16.)

The timings include all *unavoidable* delays due to memory timing and IFU restart latency, but do not include variable delays due to cache misses, IFU pipeline starvation, and the like; measurements have shown that these degrade performance only a small amount [Clark, *et al.*, 1981; Lampson, *et al.*, 1981]. Nor do the timings account for I/O overhead; but this is typically less than 10 percent (including Alto-compatible display), and in any event will affect all opcodes proportionally.

In the opcode names, *'' stands for a digit denoting an encoded constant; the name thereby refers to a family of related opcodes. The comments generally specify the conditions under which the timings apply.

Opcode(s)	*Timing*	*Comments*
ADD	2	
ALLOC	10	Ordinary case (no trap, AV[fsi] non-empty).
AND	2	
ASSOC	103	Minimum (more if many dirty cache entries)
BCAST	71	Typical minimum (assuming 1 process awakened, no process switch).
BITBLT	86+height*(35+5*(width/16)) (Typical case)	
BLT, BLTC, BLTL	27+3*count	Approximate.

BNDCK	4	Ordinary case (in bounds).
CATCH	1	
CKSUM	9+3.5*count	
DADD	4	
DBL	1	
DCOMP	7	Approximate.
DESCB, DESCBS	4	
DST	15	Minimum.
DUCOMP	6	Approximate.
DIV	42	Approximate; ordinary case (no overflow).
DSUB	4	
DUP	1	
DWDC	4	Ordinary case (no interrupts pending).
EFC*, EFCB	54	Ordinary case (no trap, external link is procedure)
EXCH	3	
FADD	56	Typical for normal'' operands.
FCOMP	27	Typical for normal'' operands.
FDIV	156	Typical for normal'' operands.
FIX	26	Typical for normal'' operands.
FIXC	20	Typical for normal'' operands.
FIXI	24	Typical for normal'' operands.
FLOAT	41	Typical for normal'' operands.
FMUL	104	Typical for normal'' operands.
FREE	8	
FSUB	57	Typical for normal'' operands.
GADRB	3	
GETF	101	Minimum (more if many dirty cache entries)
INC	1	
IWDC	3	
J*, JB	1	
JEQ*, JEQB	9	if jump occurs;
	4	if jump does not occur. See general comments on conditional jumps.
JGB, JGEB	10	if jump occurs;
	5	if jump does not occur.
JIB	17	if jump index in range;
	5	if jump index out of range.
JIW	16	if jump index in range;
	5	if jump index out of range.
JLB, JLEB	10	if jump occurs;
	5	if jump does not occur.
JNE*, JNEB	9	if jump occurs;
	4	if jump does not occur.
JUGB, JUGEB	9	if jump occurs;
	4	if jump does not occur.
JULB, JULEB	9	if jump occurs;
	4	if jump does not occur.
JW	10	
JZEQB, JZNEB	8	if jump occurs;
	3	if jump does not occur.
KFCB	49	Ordinary case (no trap, SD entry is procedure).
LADRB	3	
LDIV	42	Approximate; ordinary case (no overflow).

LFC*, LFCB	35	Ordinary case (no trap).
LG*, LGB	2	
LGDB	3	
LI*, LIB	1	
LIN1, LINB, LINI	2	
LINKB	4	
LIW	3	
LL*, LLB	2	
LLDB	3	
LLKB	7	
LP	3	
LST	37	Ordinary case (no trap, control link is frame).
LSTF	45	Ordinary case (no trap, control link is frame).
ME	12	Ordinary case (monitor not already locked).
MISC		See description of individual MISC opcodes (ASSOC, CKSUM, FADD, FCOMP, FDIV, FIX, FIXC, FIXI, FLOAT, FMUL, FSUB, GETF, ROUND, ROUNDC, ROUNDI, SETF).
MRE	32	Ordinary case (monitor not already locked).
MUL	24	
MXD	15	Ordinary case (no process waiting on monitor).
MXW	145	Minimum (includes one process switch).
NEG	1	
NILCK	3	Ordinary case (not NIL).
NILCKL	4	Ordinary case (not NIL).
NOOP	2	Reduced to 1 if post-XFER interrupt restriction is eliminated.
NOTIFY	71	Minimum (assuming no process switch).
OR	2	
PL*	2	
POP	1	
PORTI	6	
PORTO	34	Ordinary case (no trap, control link is frame).
PUSH	1	
R*, RB	2	
RBL	4	
RD*, RDB	4	
RDBL	6	
REQUEUE	63	Minimum (assuming no process switch).
RET	34	Ordinary case (no trap, return link is frame).
RF	3	
RFC	4	
RFL	5	
RFS	6	
RFSL	7	
RIGP	5	
RIGPL	9	
RIL*	4	
RILP	5	
RILPL	9	
ROUND	35	Typical for ‘‘normal’’ operands.
ROUNDC	28	Typical for ‘‘normal’’ operands.
ROUNDI	33	Typical for ‘‘normal’’ operands.
RR	6	Typical.

RSTR	5	
RSTRL	6	
RXGPL	9	
RXLP	6	
RXLPL	9	
SHIFT	6	Ordinary case (ABS[shiftCount] < 16)
SETF	122	Minimum (more if many dirty cache entries)
SFC	49	Ordinary case (no trap, control link is procedure).
SG*, SGB	2	
SGDB	4	
SL*, SLB	2	
SLDB	4	
SUB	2	
W*, WB	3	
WBL	4	
WD*, WDB	4	
WDBL	5	
WF	5	
WFL	7	
WFS	7	
WFSL	9	
WIGPL	9	
WILP	6	
WILPL	9	
WR	5	Typical.
WS*, WSB	4	
WSDB	5	
WSF	6	
WSTR	7	
WSTRL	8	
WXGPL	9	
WXLP	6	
WXLPL	9	
XOR	2	

General comments on timings

Simple instructions

For the most part, except for conditional jumps the simple opcodes (those with timings less than about 10) are presently implemented as efficiently as is possible. The microcode for these opcodes has remained relatively stable for some time.

The only potential improvement will come from use of the Dorado IFU's instruction forwarding feature. At the expense of control store (somewhat over 200 microinstructions), a number of opcodes can be made one cycle faster by deferring their final operations into the next opcode, where those operations can sometimes be overlapped with other work. In particular, all instructions whose final operation is to fetch from memory (LL*, R*, etc.) or store into memory (SL*, W*, etc.) can be speeded up by one cycle, though it is not always the case that the deferred operation can actually be overlapped with other work by the next instruction.

Until recently, the Alto-Mesa emulator actually did use the instruction forwarding feature, resulting in measured improvements of up to 8 percent. Unfortunately, instruction forwarding cannot be used in the Pilot microcode and still remain faithful to the PrincOps. This is because the PrincOps requires faults and traps to abort the instruction that causes them; with instruction forwarding, such

exceptions are frequently not detected until emulation of the next instruction has begun. (There are some more subtle technical problems as well.)

Most likely we will stretch the PrincOps to permit instructions to be suspended in mid-execution and later resumed. We believe it is possible to save and restore the necessary machine-dependent state; of course, Pilot modifications will be required to remember this state when suspending a process that has faulted.

Conditional jumps

For the conditional jumps, the jump case is typically at least twice as costly as the non-jump case; the costlier case is the one in which the IFU has to be restarted. The two cases can trivially be interchanged (on a per-opcode basis) to favor the non-jump case, if dynamic execution frequency seems to warrant this. Alternatively, two opcodes can be provided, one favoring each case; there are undoubtedly situations (e.g., the jump that closes a FOR loop) in which the compiler could easily decide which case is likely to be dynamically more frequent.

Additionally, the favored case of each conditional jump can be made one cycle faster, through use of the instruction forwarding and conditional exit features of the Dorado IFU. The cost of this is additional control store (somewhat over 100 microinstructions) and, in some cases, a restriction on whether the jump or non-jump case is to be favored.

Process/monitor opcodes

The ME and MXD opcodes are invoked upon every entry to and exit from a monitor. Approximately half their execution time is spent deciding whether their pointer arguments are short or long, by inspecting the stack depth. This would be eliminated if separate short and long versions of these opcodes were introduced (as has long been specified in the PrincOps but has never been implemented).

Control transfers

Substantial improvements are possible in the control transfer (XFER) primitives, which are presently rather expensive. These improvements will derive from various forms of caching of execution contexts, for which there is some Dorado hardware support that is not presently used.

To take advantage of this will require rewriting nearly all the microcode for the control transfer opcodes. Additionally, we will have to identify those places in Pilot that reference the cached information (e.g., frame overhead words) and add code to dump or invalidate the cache at appropriate times.

To gain an idea of what improvements are possible, it is worth examining the workings of the control transfer primitives in some detail. The two most important cases are transfers to procedures as typified by EFC*, and transfers to frames as typified by RET.

	Operation	*Timing*
	EFC (procedure transfer)*	
P1.	Save PC in local frame, and obtain local frame pointer.	5
P2.	Fetch external control link from code segment.	5
P3.	Dispatch on control link tag.	3
P4.	Decode procedure descriptor to obtain global frame and entry index.	8
P5.	Load global frame and code base registers.	5
P6.	Obtain PC for entry point and restart IFU.	5

P7.	Allocate new local frame.	11
P8.	Set up overhead words in new local frame.	5
P9.	Dispatch on exit actions, and load local frame base register.	3
P10.	Push source and destination control links onto stack.	3

RET (frame transfer)

F1.	Fetch return link from local frame.	2
F2.	Dispatch on control link tag.	3
F3.	Load global frame and code base registers.	5
F4.	Obtain PC from destination frame and restart IFU.	4
F5.	Dispatch on exit actions, and load local frame base register.	3
F6.	Push source and destination control links onto stack.	5
F7.	Free old local frame.	8

The caching strategy comes in two parts. First, the base registers and local frame overhead words are cached in the hardware, eliminating most of P1, P8, and F1 F4, for a total of 24 cycles per EFC*/RET pair. Second, up to four local frames are kept in the cache rather than being allocated and freed, eliminating most of P7 and F7, for a total of 19 cycles. Of course, there are some compensating costs arising from the need to check for cache over/underflow and other conditions that require the cache to be invalidated.

It is worth mentioning some architectural changes that would also speed things up. First, the compact encoding of procedure descriptors has a substantial cost (P4); a simpler encoding (probably requiring 32 bits) would speed up procedure transfers. Second, the control links pushed onto the stack (P10 and F6) are saved for the benefit of coroutines and nested local procedures, which are used relatively infrequently; some way of obtaining this information only when it is needed would be better.

Other observations

The operations performed by NILCK and BNDCK could be provided nearly free if performed as part of the dereferencing and array access opcodes rather than as distinct opcodes.

References

[Clark, *et al.*, 1981]
Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier, The Memory System of a High-Performance Personal Computer,'' *IEEE Transactions on Computers*, to appear; CSL report 81-1.

[Lampson, *et al.*, 1981]
Butler W. Lampson, Gene A. McDaniel, and Severo M. Ornstein, An Instruction Fetch Unit for a High-Performance Personal Computer,'' submitted for publication; CSL report 81-1.