TR-I-0297

Instruction Duration Timing Results for the Sequent Symmetry Parallel Computer

John K. Myers

1993. 1

Abstract

Command-duration benchmark data is mandatory for programmers who want to write code that runs as fast as possible. This report provides the results of benchmarking the duration of the execution of various commands on ATR's Sequent Symmetry parallel computer. This data will allow Sequent programmers to be able to write efficient code, and to understand why programs take as long to run as they do.

ATR自動翻訳電話研究所

ATR Interpreting Telephony Research Laboratories

Contents

1 Durations Ranked by Duration 1

2 Introduction 5

3 Configuration 5

4 Disclaimer 5

5 Testing Procedure and Considerations 5

6 Monitor Level Timings 8

7 Durations Ranked by Topic 9
- 7.1 Common Lisp 9
- 7.2 CLiP Parallel Commands 12
- 7.3 Beholder Toolbox Commands 12
- 7.4 List, Array, and Structure Reference Commands 13

8 Discussion and Comments 14
- 8.1 Conditionals 14
- 8.2 Structures, Arrays, and Lists 14

9 Results 15

10 Conclusion 15

11 References 16

A Listing of Test Program 17

B Sample Run of Program 39

C Durations Ranked by Command Name 41

1 Durations Ranked by Duration

The following results are for compiled code run in a subprocess.

Results are ordered from the fastest to the slowest.

A microsecond (abbreviated "musec") is 0.000001 second.

Timing results under 1 musec are probably not reliable.

Standard significant-digit conventions are followed in representing the timing information. All significant digits are shown. Partially significant digits are represented as a trailing 0 or 5 (i.e., +/- 0.25). A trailing decimal point means the estimate is accurate to about +/- 0.5; no trailing decimal indicates the estimate is accurate to +/- 2. or more.

```
'A                                          0.01 musec Estimated.
(PROGN 'A 'B 'C 'D 'E 'F 'G 'H 'I 'J):      0.05 musec (?!)
(PROGN 'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J):      0.05 musec (?!)
(PROGN NIL):                                0.1  musec
(PROGN NIL NIL):                            0.1  musec
(PROGN 'A):                                 0.1  musec (?)
(PROGN 'A 'B):                              0.1  musec (?)
(EQUAL 'A 'A):                              0.15 musec
(EQ 'A 'A):                                 0.22 musec
(EQ 'A 'NOT-A):                             0.22 musec
(EQUAL 'A 'NOT-A):                          0.25 musec
(DOTIMES (I 0)):                            1.46 musec
(LIST 'A):                                  1.6  musec
(CONS 'A 'B):                               1.8  musec
(DOTIMES (I iteration-count)):              1.925 musec
   [once through the iteration]
(LET ((A 'B) )):                            2.4  musec
(LET ((A 'B) (C 'D))):                      2.7  musec
(DOTIMES (I 1)):                            2.7  musec
(PROGN  A):                                 5.90 musec
(PROGN *THIS-LWP*):                         5.90 musec
 A                                          6    musec Estimated.
(PROGN (PROGN (PROGN (PROGN (PROGN (PROGN (PROGN (PROGN
  (PROGN (PROGN *THIS-LWP*))))))))):        6.02 musec
(SETQ A 1):                                 6.4  musec
(WHEN A 'B)      [A = T]:                   6.4  musec
(IF A 'B 'C)     [A = T]:                   6.6  musec
```

```
(FIRST A):                                  6.7  musec
(CAR A):                                    6.70 musec
(CDR A):                                    6.65 musec
(<= C 3):                                   6.8  musec       [C = 5]
(<= C 5):                                   6.8  musec       [C = 5]
(<= C 10):                                  6.8  musec       [C = 5]
(< C 0):                                    6.8  musec       [C = 5]
(< C 3):                                    6.8  musec       [C = 5]
(< C 10):                                   6.8  musec       [C = 5]
(> C 0):                                    6.8  musec       [C = 5]
(> C 5):                                    6.8  musec       [C = 5]
(> C 10):                                   6.8  musec       [C = 5]
(SECOND A)                                  7.   musec
(WHEN A 'B)   [A = NIL]:                    7.1  musec
(SETQ A 'B):                                7.2  musec
(NTH 1 A):                                  7.2  musec
(<= C 0):                                   7.5  musec       [C = 5]  ;Possibly = 0.
(RPLACD A 'B):                              7.5  musec
(THIRD A):                                  7.5  musec
(IF A 'B 'C)  [A = NIL]:                    8.0  musec
(FOURTH A):                                 8.   musec
(FIFTH A):                                  8.5  musec
(NTH 5 A):                                  8.8  musec
(SIXTH A):                                  9.   musec
(SEVENTH A):                                9.5  musec
(EIGHTH  A):                               10.   musec
(NINTH   A):                               10.5  musec
(TENTH   A):                               11.   musec
(STRUCTA-P STRUCTA-INSTANCE):              11.0  musec
(AREF ARRAY1  0):                          11.5 musec
(AREF ARRAY10 0):                          11.5 musec
(AREF ARRAY10 9):                          11.5 musec
(AREF ARRAY100 0):                         11.5 musec
(AREF ARRAY100 9):                         11.5 musec
(AREF ARRAY100 99):                        11.5 musec
(AREF ARRAY1000 0):                        11.5 musec
(AREF ARRAY1000 9):                        11.5 musec
(AREF ARRAY1000 99):                       11.5 musec
(AREF ARRAY1000 999):                      11.5 musec
(SETF (AREF ARRAY1 0) 1):                  11.7  musec
(SETF (AREF ARRAY10 9) 1):                 11.7  musec
(AREF ARRAY1000 99):                       12.1  musec
(MAKE-SPIN-LOCK):                          13    musec  [Depends on swapping.]
(SETQ A 0.111):                            13.4  musec
(MAKE-MAILBOX):                            14   musec    [Depends on swapping.]
(LIST 'NIL 'NIL 'NIL 'NIL 'NIL 'NIL 'NIL): 14 musec
(SETQ A B):                                14   musec       [B = 'A, or 5, or 0.1234567890]
```

```
(MAKE-STRUCTA):                          15   musec  [Depends on swapping.]
(++ E):                                  14.4 musec
(++ E):                                  14.4 musec
(LIST NIL NIL NIL NIL NIL NIL NIL):      15   musec
(STRUCTA-SLOT1 STRUCTB):                 16.5 musec    ;All slots the same.
(STRUCTA-SLOT2 STRUCTB):                 16.5 musec
(STRUCTA-SLOT3 STRUCTB):                 16.5 musec
(SETF (STRUCTA-SLOT1 STRUCTB) NIL):      16.5 musec
(SETF (STRUCTA-SLOT2 STRUCTB) NIL):      16.5 musec
(SETF (STRUCTA-SLOT3 STRUCTB) NIL):      16.5 musec
(DIV2 15):                               17   musec
(MAKE-SLEEPY-LOCK):                      17   musec  [Depends on swapping.]
(DOTIMES (I 10)):                        18   musec
(POP A):                                 20.  musec Estimate.
(PROGN (POP A)):  [A is NIL]             20.5 musec
(PUSH 'B A):                             21.  musec
(AND A A A):  [A = T]:                   22   musec
(WITH-SPIN-LOCK SP-LOCKA NIL):           23   musec
(OR A A A):   [A = NIL]                  24   musec
(MAKE-FORK):                             27   musec  [Depends on swapping.]
(PROGN (POP A)):  [A is full]            27.5 musec
(PROGN (PUSH 'B A) (SETQ A 'NIL)):       28.  musec
(<= C 2.38):                             28.0 musec       [C = 5]  ;Floating.
(PROGN (ACQUIRE-SPIN-LOCK SP-LOCKA)
       NIL
       (RELEASE-SPIN-LOCK SP-LOCKA)):    30   musec
(PROGN (PUSH 'B A)):                     30   musec         [A fills up]
(DOLIST (I A) ):                         29.1 musec [A = '(1 2 3 4 5 6 7 8 9 10)]
(PROGN (++ E) (-- E)):                   32.8 musec
(MAKE-FMAILBOX):                         35   musec    [Depends on swapping.]
(++L E SP-LOCKA):                        38   musec
(NTH 10 A):                              40.  musec
(my-with-spin-lock-2 SP-LOCKA NIL):      40   musec
(PROGN (PUSH 'B A) (POP A)):             43   musec
(EVAL B):                                44.0 musec
(EQUAL '(A B C) '(A B C)):               46.5 musec
(MAKE-ARRAY 1):                          64   musec
(MAKE-ARRAY 10):                         69   musec
(FMB-RECEIVE MAILBOXB):                  90   musec
(FMB-SEND NIL MAILBOXB):              100-190 musec  [Depends on swapping.]
(PROGN (FMB-SEND NIL MAILBOXB) (FMB-RECEIVE MAILBOXB)):      110 musec    (?!)
(MAKE-ARRAY 1 INITIAL-ELEMENT 0):        118  musec
(MAKE-ARRAY 10 INITIAL-ELEMENT 0):       130  musec
(MB-RECEIVE MAILBOXA):                   140  musec[Apparently varies with swapping.
(MAKE-ARRAY 100):                        136  musec
(GETHASH 'NOT-THERE HASH-EQL):           137  musec    ;Hash table empty.
(MB-SEND NIL MAILBOXA):               140-200 musec    [Depends on swapping.]
```

```
(SETF (GETHASH 'ONLY-ONE-THERE HASH-EQL) 'A):      150 musec    ;Table: One entry.
(GET-TIME):                              159.  musec
(STRING-APPEND A A)      [A  = 'i]:      190   musec
(MAKE-ARRAY 100 INITIAL-ELEMENT 0):      201   musec
(MAKE-HASH-TABLE):                       223   musec    [Depends on swapping.]
(MAKE-HASH-TABLE TEST 'EQUAL):           259   musec    [Depends on swapping.]
(PROGN (MB-SEND NIL MAILBOXA)
       (MB-RECEIVE MAILBOXA)):           300   musec [with 0, 5, or 10 in queue]
(NTH 100 A):                             299  musec
(MAKE-ARRAY '(10 10)):                   332 musec
(PFORMAT NIL "A short message."):        415   musec
(STRING-APPEND A A):                     415   musec     [A  = 'ten-charst]
(MAKE-ARRAY '(10 10) INITIAL-ELEMENT 0): 416 musec
(EVAL (EVAL (EVAL (EVAL (EVAL (EVAL (EVAL
  (EVAL (EVAL (EVAL A))))))))):          450.   musec
(RANDOM 10):                             938   musec
(RAND 10):                               985   musec
(BURN-CYCLES 500):                      1006.5 musec
(MY-STRING A)      [A  = 'i]:           1295   musec
(MY-STRING A) [A  = 'ten-charst]:       1560   musec
(PFORMAT NIL "A long message, ~A ~A ~A ~A ~A with five evaluations."
    B C B C B):                         1925   musec
(BURN-CYCLES 1000):                     2009.5 musec
(STRING-APPEND A A)  [A 100 chars]:     2670   musec
(MY-STRING A) [A's name 100 chars]:     4230   musec
```

2 Introduction

The main reason for using a parallel computer is the potential for significant increases in the speed of processing. When comparing the use of parallel computers against the use of sequential computers, the power of the processing and the power of the algorithms will remain the same as programs run on sequential computers; only the speed will increase. However, in order to obtain fast processing, it is not enough simply to "use a parallel computer". Programmers must write efficient programs that use the capabilities of the computer well. In order to write fast programs, it is mandatory that a programmer know how long different basic functions take to compute.

This report attempts to partially fill this need. The basic functions of Common Lisp, the extended parallel functions of CLiP, and the extended functions of the Beholder Toolbox[1] package are all documented here. A programmer can use this data to predict how long his or her routines should take to run. With this knowledge, a programmer can design programs that run as fast as possible.

3 Configuration

The ATR Sequent Symmetry S2000/700 parallel computer currently is using 15 processors, of type 1003-54504. It runs the DYNIX v.3.1.1 NFS #6 system, and the Allegro CLiP 3.0.3 [sequent] version of parallel Common Lisp. The machine has 56 Megabytes of main memory. It has a 16 Meg memory controller board, and a 24 Meg EXP memory board. The machine is configured for up to 16 users. All statistics are as of January 1992.

4 Disclaimer

The test results presented here are approximate, and are intended for advisory purposes only. These are *not* official benchmarks. In order to really test the machine, it would be necessary to test it under unloaded conditions, lightly loaded conditions, and fully loaded conditions; this was not done. There were some discrepancies in the timing based on the number of iterations performed for testing (e.g., 10,000 iterations vs. 100,000); these should be traced down. There were discrepancies on the order of 1%-5% in the timing of most instructions other than the tight loop; this is also rather strange for a deterministic processor. The major difference between code running on forked processes and code running under the main process should be tracked down and explained. In short, although these results are useful for ATR programmers, they should not be used as accurate "official" benchmarks.

5 Testing Procedure and Considerations

Timing an instruction execution duration appears to be an easy matter. A naive method of timing would be to get the time, execute the instruction, get the time again, and subtract

[1]The Beholder Toolbox package is a series of routines that extends the capabilities of CLiP, simplifies programming, and fixes many problems with the current version of the Sequent CLiP language. It is documented separately in [Mye92]. Programmers at installations that do not have the Beholder Toolbox package can safely ignore sections of this report that refer to such routines.

the two times to get the duration. Why is timing an instruction more difficult than this?

There are two main reasons. The first reason has to do with the instruction that gets the time on the Sequent. The second reason has to do with the Sequent operating system.

Problems in the timing routines. The instruction `get-interal-real-time` on the Sequent is calibrated to tenths of a millisecond. One would think from this that the function could therefore be used to time events accurately to a tenth of a millisecond. However, it turns out that the `get-internal-real-time` instruction on the Sequent is only accurate to one or two hundreths of a second. The timing result always ends in 0.0074 seconds, so the third and fourth decimal places are not significant and the second is questionable.

Thus, in order to obtain significant results, each experiment has been repeated 10,000,000 times using the routing `extra-time-routine`, or 100,000 times using the routine `time-routine`, in order to shift the timing results into the few-seconds range. This is done by surrounding the code to be tested by a `dotimes` loop, and then timing the duration of this loop plus the code. The result of this is the "gross time" for the loop plus the repetitions of the code. The "overhead time" for running the loop itself without any code in it has been previously determined, and has been found to be quite stable. The overhead time is subtracted from the gross time to yield the "net time" for the duration of the code repetitions by itself. This is then divided by the number of repetitions to obtain the average duration of a single execution of the test code.

The `get-internal-real-time` instruction itself takes 160 microseconds to execute. It is called twice: once at the beginning, and once at the end of the instruction to be timed, after all of the loop has been performed. However, due to signpost considerations, although it is called twice, it only contributes once to the timing error, and the gross timings will be consistently high by 160 microseconds. Since the repeat loop is *inside* the timing, this error is divided by the repetition count, and the actual error in measurement for 100,000 repetitions is 0.00160 microseconds [not seconds].[2] This is lost in the noise for most instructions; the few instructions that are down at microsecond levels get timed at higher repetition rates, so there is no problem. If the loop were to be *outside* of the timing start and stop, the 160 microsecond figure would contribute significantly to the timing result and would have to be subtracted out.

Significantly, since the routine `time-routine` only repeats its test code 100,000 times, the results are only accurate to about the nearest microsecond, and anything under 10 microseconds cannot be tested accurately using this routine. It is necessary to use the routine `extra-time-routine`, which uses 10,000,000 repetitions, in order to time such quick routines. This is accurate to about the nearest 0.01 microsecond, so it should not be used to test anything under 0.1 musec duration.

Timing differences in forked processes vs. the monitor process. The second problem comes with the Sequent operating system. There is a surprising, huge difference between timed results for running compiled routines at the `clip` monitor level (using the so-called "initial lwp") and running compiled routines in a subprocess that has been spawned using `start-lwps`. Routines that run at the top level are about *seven times as slow* as routines that run inside an activated process after `start-lwps` has been called. It is unclear why this is the case, but it is speculated that the operating system for the high-level lisp

[2]The latest thinking observes that this offset error is constant, and is present both in the gross time and in the overhead time measurements. Therefore, this should cancel out automatically, leaving no error.

somehow is including some kind of multi-tasking heart-beat-scheduled swapping, whereas the low-level lisps inside activated lwps run relatively clean, without this swapping.

It is possible to draw a conclusion from this observation. If an application's code is to run as fast as possible, then a programmer should make the program so that it only sets things up inside the code at the top level and then calls `start-lwps` as fast as possible. Use a running spawned lwp to initialize things and set up the other lwps, if possible.

In order to test the routines properly, it was thus necessary to perform the tests in subprocesses that were running at a lower level. This is the level that will most often be used for actual programs. This current report presents only a few results from running compiled routines at the monitor level, in Section 6.

Timing differences in compiled vs. interpreted code. Note that there is also obviously a hugh difference between running compiled and interpreted code. Interpreted code is significantly slower than compiled code. Most programmers will only run compiled code, and at present no statistics are available for interpreted code.

6 Monitor Level Timings

The following brief results are for compiled code run at the monitor level. Notice that most commands are significantly longer in duration than the same commands when run in a subprocess. Most programmers will run applications only in subprocesses, and should use the other tables which contain data for subprocesses. They will probably not need the data in this table. The table is included for comparison purposes. Only a few runs were made, so the data in this table is not very reliable.

```
(CAR A):                          47  musec
(CDR A):                          47  musec
(CONS 'A 'B):                      2  musec
(LIST 'A):                         2  musec
(RPLACD A 'B):                    47  musec
(FIRST A):                        47  musec
(SECOND A):                       47  musec
(THIRD A):                        48  musec
(FOURTH A):                       48  musec
(FIFTH A):                        49  musec
(SIXTH A):                        49  musec
(SEVENTH A):                      50  musec
(EIGHTH A):                       63  musec
(NINTH A):                        60  musec
(TENTH A):                        51  musec
(NTH 1 A):                        47  musec
(NTH 5 A):                        58  musec
(NTH 10 A):                       90  musec
(NTH 100 A):                     340  musec
(MAKE-STRUCTA):                   15  musec
(SETQ STRUCTB (MAKE-STRUCTA)):                68  musec
(SETF (STRUCTA-SLOT1 STRUCTB) NIL):           58  musec
(SETF (STRUCTA-SLOT2 STRUCTB) NIL):           63  musec
(SETF (STRUCTA-SLOT3 STRUCTB) NIL):           57  musec
(AREF ARRAY1 0):                  51  musec
(AREF ARRAY10 0):                 51  musec
(AREF ARRAY100 0):                56  musec
(AREF ARRAY1000 0):               51  musec
(AREF ARRAY10 9):                 51  musec
(AREF ARRAY100 9):                51  musec
(AREF ARRAY1000 9):               56  musec
(AREF ARRAY100 99):               51  musec
(AREF ARRAY1000 99):              51  musec
(AREF ARRAY1000 999):             51  musec
(DIV2 15):                        16  musec
(MAKE-MAILBOX):                   14  musec
(MB-SEND NIL MAILBOXA):          200  musec
(MB-RECEIVE MAILBOXA):           230  musec
(PROGN (MB-SEND NIL MAILBOXA) (MB-RECEIVE MAILBOXA)):      380  musec
```

7 Durations Ranked by Topic

The following results are for compiled code run in a subprocess.

Results are ordered in groups by the type of instruction.

A microsecond (abbreviated "musec") is 0.000001 second.

Timing results under 1 musec are probably not reliable.

Standard significant-digit conventions are followed in representing the timing information. All significant digits are shown. Partially significant digits are represented as a trailing 0 or 5 (i.e., +/- 0.25). A trailing decimal point means the estimate is accurate to about +/- 0.5; no trailing decimal indicates the estimate is accurate to +/- 2. or more.

7.1 Common Lisp

```
(DOTIMES (I iteration-count)):          1.925 musec     [once through the iteratio

'A                                      0.01 musec Estimated.
 A                                      6    musec Estimated.

(RANDOM 10):                          938    musec
(SETQ A 1):                             6.33 musec
(SETQ A 0.111):                        13.4  musec
(SETQ A 'B):                            7.2  musec
(SETQ A B):                            14    musec  [B = 'A, or 5, or 0.1234567890
(LET ((A 'B) (C 'D))):                  2.7  musec
(PROGN NIL):                            0.1  musec
(PROGN NIL NIL):                        0.1  musec
(PROGN 'A):                             0.1  musec (?)
(PROGN 'A 'B):                          0.1  musec (?)
(PROGN 'A 'B 'C 'D 'E 'F 'G 'H 'I 'J): 0.05 musec (?!)
(PROGN 'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J): 0.05 musec (?!)
(PROGN  A):                             5.90 musec
(PROGN A B C D E F G H I J):           63    musec
(PROGN *THIS-LWP*):                     5.90 musec
(PROGN (PROGN (PROGN (PROGN (PROGN (PROGN (PROGN
  (PROGN (PROGN *THIS-LWP*))))))))):    6.02 musec
(IF A 'B 'C)  [A = NIL]:                8.0  musec
(IF A 'B 'C)    [A = T]:                6.6  musec
```

```
(WHEN A 'B)    [A = NIL]:                7.1  musec
(WHEN A 'B)      [A = T]:                6.4  musec
(OR A A A):    [A = NIL]:               24    musec
(AND A A A):   [A = T]:                 22    musec
(CAR A):                                6.70 musec
(CDR A):                                6.65 musec
(CONS 'A 'B):                           1.8  musec
(LIST 'A):                              1.6  musec
(LIST NIL NIL NIL NIL NIL NIL NIL):     15  musec
(LIST 'NIL 'NIL 'NIL 'NIL 'NIL 'NIL 'NIL): 14 musec
(RPLACD A 'B):                          7.5  musec

(DOLIST (I A) ):                        29.1  musec [A = '(1 2 3 4 5 6 7 8 9 10)]
(DOTIMES (I 10)):                       18    musec
(EVAL B):                               44.0  musec
(EVAL (EVAL (EVAL (EVAL (EVAL (EVAL (EVAL
  (EVAL (EVAL (EVAL A))))))))):       450.   musec
(PUSH 'B A):                            21.   musec
(POP A):                                20.   musec Estimate.
(PROGN (PUSH 'B A) (SETQ A 'NIL)):      28.   musec
(PROGN (PUSH 'B A) (POP A)):            43    musec
(PROGN (PUSH 'B A)):                    30    musec         [A fills up]
(PROGN (POP A)):  [A is NIL]            20.5  musec
(PROGN (POP A)):  [A is full]           27.5  musec
(DIV2 15):                              17    musec

(<= C 0):                               7.5  musec       [C = 5]  ;Possibly = 0.
(<= C 2.38):                           28.0  musec       [C = 5]  ;Floating.
(<= C 3):                               6.8  musec       [C = 5]
(<= C 5):                               6.8  musec       [C = 5]
(<= C 10):                              6.8  musec       [C = 5]
(< C 0):                                6.8  musec       [C = 5]
(< C 3):                                6.8  musec       [C = 5]
(< C 10):                               6.8  musec       [C = 5]
(> C 0):                                6.8  musec       [C = 5]
(> C 5):                                6.8  musec       [C = 5]
(> C 10):                               6.8  musec       [C = 5]
(EQ 'A 'A):                             0.22 musec
(EQ 'A 'NOT-A):                         0.22 musec
(EQUAL 'A 'A):                          0.15 musec
(EQUAL 'A 'NOT-A):                      0.25 musec
(EQUAL '(A B C) '(A B C)):             46.5 musec

(MAKE-HASH-TABLE):                     223   musec    [Depends on swapping.]
(MAKE-HASH-TABLE TEST 'EQUAL):         259   musec    [Depends on swapping.]
(GETHASH 'NOT-THERE HASH-EQL):         137   musec    ;Hash table empty.
(SETF (GETHASH 'ONLY-ONE-THERE HASH-EQL) 'A):   150. musec ;Table:One entry
```

```
(GETHASH 'ONLY-ONE-THERE HASH-EQL):    145 musec
(GETHASH 'NOT-THERE HASH-EQUAL):       133 musec
(SETF (GETHASH 'ONLY-ONE-THERE HASH-EQUAL) 'A): 150. musec ;Table:One entry
(GETHASH 'ONLY-ONE-THERE HASH-EQUAL): 148 musec

(MAKE-ARRAY 1):                         60  musec    [Depends on swapping.]
(MAKE-ARRAY 1 INITIAL-ELEMENT 0):      120  musec    [Depends on swapping.]
(MAKE-ARRAY 10):                        65  musec    [Depends on swapping.]
(MAKE-ARRAY 10 INITIAL-ELEMENT 0):     130  musec    [Depends on swapping.]
(MAKE-ARRAY 100):                      125  musec    [Depends on swapping.]
(MAKE-ARRAY 100 INITIAL-ELEMENT 0):    190  musec    [Depends on swapping.]
(MAKE-ARRAY '(10 10)):                 335  musec    [Depends on swapping.]
(MAKE-ARRAY '(10 10) INITIAL-ELEMENT 0):  420 musec [Depends on swapping.]

(AREF ARRAY1 0):                       11.5 musec

(AREF ARRAY10 0):                      11.5 musec
(AREF ARRAY10 9):                      11.5 musec

(AREF ARRAY100 0):                     11.0 musec
(AREF ARRAY100 9):                     11.5 musec
(AREF ARRAY100 99):                    11.5 musec

(AREF ARRAY1000 0):                    11.5 musec
(AREF ARRAY1000 9):                    11.0 musec
(AREF ARRAY1000 99):                   12.1 musec
(AREF ARRAY1000 999):                  11.4 musec

(SETF (AREF ARRAY1 0) 1):              11.7 musec

(SETF (AREF ARRAY10 0) 1):             11.9, 13.3 musec
(SETF (AREF ARRAY10 9) 1):             11.7 musec

(SETF (AREF ARRAY100 0) 1):            11.7 musec
(SETF (AREF ARRAY100 9) 1):            11.9, 13.3 musec
(SETF (AREF ARRAY100 99) 1):           11.8 musec

(SETF (AREF ARRAY1000 0) 1):           11.8 musec
(SETF (AREF ARRAY1000 9) 1):           11.7 musec
(SETF (AREF ARRAY1000 99) 1):          11.5 musec
(SETF (AREF ARRAY1000 999) 1):         11.5, 13.3 musec


(MAKE-STRUCTA):                        15   musec  [Depends on swapping.]
(STRUCTA-SLOT1 STRUCTB):               16.5 musec
(STRUCTA-SLOT2 STRUCTB):               16.5 musec
(STRUCTA-SLOT3 STRUCTB):               16.5 musec
```

```
(SETF (STRUCTA-SLOT1 STRUCTB) NIL):   16.5 musec
(SETF (STRUCTA-SLOT2 STRUCTB) NIL):   16.5 musec
(SETF (STRUCTA-SLOT3 STRUCTB) NIL):   16.5 musec
```

7.2 CLiP Parallel Commands

```
(MAKE-SPIN-LOCK):                      13   musec  [Depends on swapping.]
(MAKE-SLEEPY-LOCK):                    17   musec  [Depends on swapping.]
(WITH-SPIN-LOCK SP-LOCKA NIL):         23   musec
(PROGN (ACQUIRE-SPIN-LOCK SP-LOCKA)
       NIL
       (RELEASE-SPIN-LOCK SP-LOCKA)): 30   musec
(MAKE-MAILBOX):                        14   musec    [Depends on swapping.]
(MB-SEND NIL MAILBOXA):           140-200   musec    [Depends on swapping.]
(MB-RECEIVE MAILBOXA):                140 musec  [Apparently varies with swapping.]
(PROGN (MB-SEND NIL MAILBOXA)
       (MB-RECEIVE MAILBOXA)):        300   musec [with 0, 5, or 10 in queue]
*THIS-LWP*:                            5.8 musec Estimated.
```

7.3 Beholder Toolbox Commands

```
(++ E):                               14.6 musec
(++L E SP-LOCKA):                       38   musec
(GET-TIME):                            159.  musec
(RAND 10):                             985   musec
(BURN-CYCLES 500):                    1006.5 musec
(BURN-CYCLES 1000):                   2009.5 musec
(MY-STRING A):                        1295   musec
    [A  = 'i]
(STRING-APPEND A A):                    190   musec
    [A  = 'i]
(MY-STRING A):                        1560   musec
    [A  = 'ten-charst]
(STRING-APPEND A A):                    415   musec
    [A  = 'ten-charst]
(MY-STRING A):                        4230   musec
(STRING-APPEND A A):                  2670   musec
(PFORMAT NIL "A short message."):       415   musec
(PFORMAT NIL "A long message, ~A ~A ~A ~A ~A with five evaluations."
    B C B C B):                       1925   musec
(MAKE-FORK):                            27   musec  [Depends on swapping.]
(MAKE-FMAILBOX):                        35   musec  [Depends on swapping.]
(FMB-RECEIVE MAILBOXB):                 90   musec
(FMB-SEND NIL MAILBOXB):           100-190   musec  [Depends on swapping.]
(PROGN (FMB-SEND NIL MAILBOXB) (FMB-RECEIVE MAILBOXB)):      110 musec    (?!)
```

7.4 List, Array, and Structure Reference Commands

```
(FIRST   A):                          6.7 musec
(SECOND  A):                          7.  musec
(THIRD   A):                          7.5 musec
(FOURTH  A):                          8.  musec
(FIFTH   A):                          8.5 musec
(SIXTH   A):                          9.  musec
(SEVENTH A):                          9.5 musec
(EIGHTH  A):                         10.  musec
(NINTH   A):                         10.5 musec
(TENTH   A):                         11.  musec

(NTH 1 A):                            7.2  musec
(NTH 2 A):                            7.5  musec
(NTH 3 A):                            8.   musec
(NTH 4 A):                            8.5  musec
(NTH 5 A):                            8.8  musec
(NTH 6 A):                            9.1  musec
(NTH 7 A):                            9.7  musec
(NTH 8 A):                           10.   musec
(NTH 9 A):                           10.5  musec
(NTH 10 A):                          40.  musec
(NTH 100 A):                        299. musec

(AREF ARRAY1  0):                     11.5 musec
(AREF ARRAY10 0):                     11.5 musec
(AREF ARRAY10 9):                     11.5 musec
(AREF ARRAY100 0):                    11.5 musec
(AREF ARRAY100 9):                    11.5 musec
(AREF ARRAY100 99):                   11.5 musec
(AREF ARRAY1000 0):                   11.5 musec
(AREF ARRAY1000 9):                   11.5 musec
(AREF ARRAY1000 99):                  11.5 musec
(AREF ARRAY1000 999):                 11.5 musec

(STRUCTA-SLOT1 STRUCTB):              16.5 musec    ;All slots the same.
(STRUCTA-SLOT2 STRUCTB):              16.5 musec
(STRUCTA-SLOT3 STRUCTB):              16.5 musec
(SETF (STRUCTA-SLOT1 STRUCTB) NIL):   16.5 musec
(SETF (STRUCTA-SLOT2 STRUCTB) NIL):   16.5 musec
(SETF (STRUCTA-SLOT3 STRUCTB) NIL):   16.5 musec
```

8 Discussion and Comments

8.1 Conditionals

One surprising result was that there is a significant difference in the time taken to execute a conditional branch depending upon whether the branch test is positive or negative. If the branch test is *positive*, the execution time is **shorter** than if the branch test is *negative*. For the (IF A 'B 'C) statement, when A is positive (T), the statement takes 6.6 microseconds to evaluate (to 'B), whereas when A is negative (NIL), it takes 8.0 microseconds to evaluate (to 'C). For the (WHEN A 'B) statement, when A is positive (T), the statement takes 6.4 microseconds to evaluate (to 'B), whereas when A is negative (NIL), it takes 7.1 microseconds to evaluate (to NIL). Note that it apparently takes about 0.05 microseconds or less to evaluate 'B. The evaluation is about 20% or 10% longer if a negative branch is used rather than a positive branch. This is apparently due to an extra branch needed in the microcode. The conclusion is that if a programmer is designing tight inner-loop code containing a conditional, then he or she should make sure that the positive test represents the situation that is expected to happen more often.

8.2 Structures, Arrays, and Lists

One of the most important design decisions in writing a computer program is how to represent the data. There are three methods that are quite similar for representing a set of data: lists, arrays, and structures. At a low level of coding, it would normally be acceptable to use any one of these. Times for storing and accessing these data structures were tested.

The time for accessing an array reference was constant at about 11.4 microseconds, no matter how large the array, or which item was required. This is because an array reference is computed by multiplying the index times the size of an array entry, adding the address of the start of the array, and pulling the contents of the resulting address out directly. The time for setting an array reference was also constant, at about 11.7 microseconds, indicating about 0.3 microseconds for the SETF.

There are two methods of accessing entries in a list. The first uses the FIRST, SECOND, THIRD series, which are more understandable aliases for the old CAR, CADR, CADDR series. The second method uses NTH and a numerical index. A list entry is normally accessed by counting down the list, i.e. sequentially taking the CAR of the list until the desired entry is reached. The first method normally does this in a compiled way by sequentially listing CAR's inside the routine, e.g. SEVENTH would have seven nested CAR's. The second, NTH method normally does this by using a DOTIMES loop around a CAR. However, apparently the Sequent's NTH operator first tests to see whether the argument is nine or under, and if so then goes ahead with a CASE dispatch and uses the faster FIRST, SECOND, etc. series. Thus, the time for both methods is expected to be order n, with the NTH access method taking slightly longer than the hard-compiled sequential method.

The results of the tests show that the FIRST, SECOND series takes roughly $6.0 + 0.5n$ microseconds to perform, where n is the order in the list. The NTH command takes an additional 4.5 microseconds or so to perform, if the argument is 1st through 9nth, for about 10.5 microseconds for (NTH 9 list). The timing then takes a big jump at 10nth to 40 microseconds.

Thus, the FIRST, SECOND methods are in fact faster than the NTH methods. As expected, data that is stored as the first entry in a list is the fastest to access, followed by

the second. This means that critical data, which is expected to be accessed the most often, should be stored at the front of the list.

The third method of representing information in a set is the use of structures. Structures are currently implemented as arrays in the present version of Clip. The 0'th index stores the structure type; the other indicies store the structure slots, in order. This means that the time to access a structure entry is also constant, as with the array. However, this time was measured at about 16 microseconds per access, the slowest of the three representational methods. Apparently the structure reference is performing a type-check to make sure that the structure is of the correct type before accessing the entry.

Note, because structures are actually implemented as arrays, that it is thus possible to access structure slots by using `aref`, if you can find out the index location of the particular slot that you want to get. If a constant is used for the entry index, this unsupported method of accessing structures is faster than using the normal structure reference method. However, if the entry index has to be computed, this must be taken into account, as the normal structure reference uses a constant for its array reference that depends on the name of the slot-access function.

One of the main reasons for using structures is that they are tagged with a type. A piece of data can be tested as to what kind of information it represents. When the structure is printed out, the operating system can use a special format to identify the type of structure, communicate this information and print the structure out beautifully. This must be given up if the information is represented in another format, such as a list or an array.

The results of this research show that lists under ten items long are significantly faster than both arrays and structures. If only the first item in a list is used, the array representational method takes about 75% longer than the list method, and structures take about an extra 140%, more than twice as much. In other words, the time taken to reference data can be cut in half or more, if structures are converted into properly-ordered lists and the corresponding structure reference calls are converted into `FIRST`, `SECOND` etc. calls. In time-critical code, such as that required to lock and unlock spinlocks, this apparently small savings can have a significant effect on the overall runtime of application programs.

9 Results

The information presented in this report, combined with the theory of how agendas and queueing processes (used by all parallel-process systems) work which was presented on pp. 44-47 of [Mye92], can be used to significantly speed up parallel-process application programs. Using the results of this report, the following improvements have been made: a fast spin-lock system, a fast mailbox system, a fast worker-agenda system, and a parallel do-loop system. These will be presented in a companion report.

10 Conclusion

If a programmer wants to make code that runs as fast as possible, it is important to know how fast each component instruction runs, and understand how fast his or her code should run. Then, and only then, can the programmer fine-tune the code so as to get the best performance. Although timing the durations of individual instructions seems like a task that has no end, this report has attempted to cover most of the main instructions

that have the highest importance. The timing macros that have been constructed and are presented in the code listing in the appendix are simple, reliable, and easy to use. If further instruction timings are required, the programmer can easily use these macros to perform timings himself or herself. The results of this report are already being used to speed up programs on the Sequent.

11 References

References

[Inc89] Franz Inc. *Allegro Common Lisp User Guide, Release 3.1.* Number D-U-00-000-01-91120-0-0. Franz Inc., Berkely, CA, November 1989.

[Mye92] John K. Myers. The beholder toolbox manual, part 1. Technical Report TR-1-0247, ATR Interpreting Telephony Research Laboratories, Kyoto, Japan, March 1992.

A Listing of Test Program

[3]

```
;;;  -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;; NAME: TEST-TIMING-BENCH-1
;;; PATH:           SQ: /usr2/myers/nl/test-timing-bench-1.lisp, .fasl
;;;
;;; VERSION: 1.1
;;;
;;; WHAT IT DOES: Benchmarks operations on the Sequent.
;;;
;;;
;;; PROBLEMS IT: How fast does the Sequent run different things?
;;; IS DESIGNED
;;; TO SOLVE
;;;
;;; WHAT YOU NEED: This package can now stand on its own,
;;; except for the Beholder Toolbox process commands SPAWN and PFORMAT
;;; and the commands that are getting timed.
;;; For instance, if you want to time Fast Mailboxes, you have to
;;; include beholder-3.
;;;
;;; Timing routines have been copied from the Beholder package.
;;;
;;;
;;; USR INTERFACES:   (benchtest1)     ;Runs all the timing programs.
;;;
;;; HOW IT WORKS: The program forks off a child subprocess and then
;;; executes things in the subprocess.
;;; First, an empty loop is timed.  Previous results have shown that
;;; empty loop durations are quite stable.
;;; Next, full loop durations are timed, and the results have the empty
;;; loop duration subtracted before dividing by the loop count.
;;; Results are printed out on the OS stream.
;;; The macro OPEN-FILE channels the OS stream to a file; this should be
;;; done separately.
;;;
;;; It is important to make sure that there are no print statements or
;;; other unnecessary instructions inside the timing loop.
;;;
```

[3]This program uses the Beholder Toolbox queued asynchronous parallel output facility `pformat` for reporting output. If `pformat` is not available, `format` can be used, with the normal caveat that characters can be dropped or rendered out of order. Apart from this, and the process spawn command (basically a cleaner "make-lwp"), care has been taken to make the benchmarks stand-alone, without other `include` files required.

```
;;;
The clock is only accurate to a few milliseconds.  It is thus necessary to
;;; get things up in the sub-second range in order to get precise timings.
;;;
;;;
;;; FUTURE EXTNS.:
;;;
;;; HISTORY
;;; -------
;;; Jan 8 '92  Created.
;;;
Jan 8 '92  DON'T USE FLOAT.  FLOAT IS SINGLE-PRECISION.  USE 6F FOR FORMATS.
;;;
June '92   Note that    (super-time-routine  'A )  does not work, because
;;;  the compiler interprets the 'A to be a *tag*, not a statement.
;;;   It is necessary to use (PROGN 'A).
;;;
;;;
;;; To Do:
;

;;;;CDR-CODING????????
```

```
;;;
;;;  VARIABLES
;;;


;This variable is used to control where the printed output gets sent to.
;If it is T, the output gets sent to the computer screen.
;If it is an open file (using Beholder command "use-file"),
; the output gets sent to the file.
(defvar OS T)    ;Output Stream

;
;These are internal variables used by the timing routines.
;
  ;Basic number of times through the loop
(defvar *iteration-count* 100000)
  ;Basic number of times through the loop per microsecond
(defvar *iteration-count-per-mu* (/ *iteration-count* 1000000))
  ;Total time taken to execute the entire loop
(defvar total-time)
  ;Time (in microseconds) each instruction takes to go once through
(defvar time-per-instruction-in-musec)
  ;Total time taken to run the LOOP ITSELF, and NOT the instruction
  ;inside the loop
(defvar *loop-duration*)
  ;Time (in seconds) that a single iteration of the LOOP ITSELF takes.
(defvar *secs-per-iteration* 0.000001925)



;
;These are register variables used by some of the instructions to be timed.
;They don't have fancy names, because they aren't important at all.
;
(defvar A)                              ;A generic variable.
(defvar B 'A)                           ;A variable storing a quoted variable.
(defvar c 5)                            ;A variable storing an integer.
(defvar D 85.1234567890)                ;A variable storing a floating-point num.
(defvar E 0)                            ;A variable storing zero.
(defvar sp-lockA (make-spin-lock))      ;A generic spin-lock.
(defvar array1 (make-array 1))          ;An array of only one element.
(defvar array10 (make-array 10))        ;A 1-D array of ten elements.
(defvar array100 (make-array 100))      ;A 1-D array 100 elements long.
(defvar array1000 (make-array 1000))    ;A 1-D array 1000 elements long.
(defvar MailBoxA (make-mailbox))        ;A generic mailbox.
```

```
(defvar f)
(defvar g)                              ;Generic variables.
(defvar h)
(defvar i)
(defvar j)

;(defvar MailBoxB (make-fmailbox))      ;This is a Beholder Fast Mailbox.

    ;Here is a generic structure, with three slots in it.
(defstruct (StructA
    (:print-function (lambda (node stream ignore)
       (format stream "#Struct<~A>"  node)))
    :named)
  (slot1)
  (slot2)
  (slot3)
)

     ;This is an instance of a generic structure.
(defvar structB (make-structA))
```

```
;;;
;;;  REQUIRED MACROS
;;;

;;;
;;; These are the macros that will be used to perform timing.
;;;
;;; The following macros are internal (not used by the user in this case).
;;;

;This routine gets the current CLOCK time.
;Although the time is reported in microseconds or so,
;the results are not accurate!!!!!!!
;The results are only accurate to a few HUNDRETHS of a second!!!!
(defmacro get-time ()
"Time in internal-real-time ticks.  Accurate to a few hundreths of a second."
  '(get-internal-real-time))


;;;This macro takes a bunch of code as an argument.
;;;Before it runs this routine, it uses get-time to get the absolute clock time.
;;;It then runs the routine, and then uses get-time to get the new absolute clock
;;;time.  These are subtracted to get the relative duration time.
;;; The subtraction is performed on the stack, so that it is not necessary
;;; to use the time to store the first clock-time into a variable and then
;;; pull it out again to subtract it.  This is the fastest way.
;;;The subtracted result is divided by internal-time-units to convert from
;;; internal ticks into seconds.
;;;
(defmacro time-it (routine)
  "Executes and times the given routine.
   Returns the resulting duration in SECONDS.
   Only accurate to a few hundreths of a second.
"
    '(/ (- (get-time) (progn ,routine (get-time)))
,(- internal-time-units-per-second)))

;;
;;; This routine is exactly the same as the previous routine,
;;; except that it returns its results in milliseconds (ticks)
;;; instead of in seconds.
;;;
(defmacro time-it-in-ticks (routine)
  "Executes and times the given routine.
   Returns the resulting duration in MILLISECONDS.
   Only accurate to a few hundreths of a second.
```

```
"
    '(- (- (get-time) (progn ,routine (get-time)))))



;;;
;;;   These are the user-interface routines for actually timing instructions.
;;;
;;;   The reasons why there are many different versions is that both the physical
;;;   duration of the timing routine varies, and also the precision of the
;;;   results varies accordingly.
;;;   It is important to get a timing run that lasts about 1-10 minutes for
;;;   one instruction in order to get good precision.  It is important not
;;;   to have a timing run that lasts two hours for one instruction, because
;;;   the operator will become bored, and if there is a mistake, the run has
;;;   to be performed again.  Most runs were performed several times and then
;;;   averaged anyway to get the results shown in the paper.
;;;   It is important not to get a run that lasts only a few tenths of seconds,
;;;   because then precision is lost due to the poor precision of the clock.
;;;   Because the duration depends both on the estimated duration of the instruction
;;;   itself, as well as on the number of iterations, many different versions
;;;   of the timing routine are offered.

;;;  This is the simple vanilla version.
;;;  It is only useful for routines that take about 100-1000 microseconds to run.
;;;
(defmacro time-routine (routine)
  '(progn
  (setq total-time (time-it                                    ;in seconds
    ;---------TIMED FUNCTION--------------------
      (dotimes (i *iteration-count*)
     ,routine
      )
    ;----------------------------------------
  ))

  (setq total-time (- total-time *loop-duration*))

  (setq time-per-instruction-in-musec
        (/ total-time *iteration-count-per-mu*))
  (pformat OS "~A: ~12F musec    (total ~12F secs for ~A iterations).~%"
   ',routine
   time-per-instruction-in-musec
   total-time
```

```
   *iteration-count*
  )
))


;;;  This is the enhanced version.
;;;  It is useful for routines that take about 10-100 microseconds to run.
;;;  THIS VERSION IS THE STANDARD VERSION USED FOR MOST INSTRUCTION TIMINGS.
;;;
(defmacro extra-time-routine (routine)
  `(let ((local-iteration-count (* *iteration-count* 100)))
  (setq total-time (time-it                                    ;in seconds
    ;---------TIMED FUNCTION--------------------
      (dotimes (i local-iteration-count)
     ,routine
      )
    ;-----------------------------------------
  ))

  (setq total-time (- total-time (* *loop-duration* 100)))

  (setq time-per-instruction-in-musec
         (/ total-time (* *iteration-count-per-mu* 100)))
  (pformat OS "~A: ~12F musec    EXTRA(total ~12F secs for ~A iterations).~%"
   ',routine
   time-per-instruction-in-musec
   total-time
   local-iteration-count
  )
))



;;;  This version goes in the other direction.
;;;  It is useful for routines that take 1000 or more microseconds to run.
;;;
(defmacro low-time-routine (routine)
  `(let ((local-iteration-count (* *iteration-count* .1)))
  (setq total-time (time-it                                    ;in seconds
    ;---------TIMED FUNCTION--------------------
      (dotimes (i local-iteration-count)
     ,routine
      )
    ;-----------------------------------------
  ))

  (setq total-time (- total-time (* *loop-duration* .1)))
```

```
  (setq time-per-instruction-in-musec
            (/ total-time (* *iteration-count-per-mu* .1)))
  (pformat OS "~A: ~12F musec    LOW(total ~12F secs for ~A iterations).~%"
   ',routine
   time-per-instruction-in-musec
   total-time
   local-iteration-count
  )
))


;;;  This is the ultra-enhanced version.
;;;  It is useful for routines that take .1-10 microseconds to run.
;;;
(defmacro super-time-routine (routine)
  `(let ((local-iteration-count (* *iteration-count* 1000)))
  (setq total-time (time-it                                    ;in seconds
    ;---------TIMED FUNCTION--------------------
      (dotimes (i local-iteration-count)
     ,routine
      )
    ;-----------------------------------------
  ))

  (setq total-time (- total-time (* *loop-duration* 1000)))

  (setq time-per-instruction-in-musec
              (/ total-time (* *iteration-count-per-mu* 1000)))
  (pformat OS "~A: ~12F musec    SUPER(total ~12F secs for ~A iterations).~%"
   ',routine
   time-per-instruction-in-musec
   total-time
   local-iteration-count
  )
))




;;;
;;;  UTILITY FUNCTIONS
;;;


;;; These internal functions are necessary to set up the variables
```

```
;;; *loop-duration* and time-per-instruction-in-musec
;;; so that the timing routines can work properly.

(defun time-loop-by-itself ()

  (let ((local-iteration-count  (* *iteration-count* 100)))
   (setq total-time (time-it                                    ;in seconds
    ;---------TIMED FUNCTION--------------------
      (dotimes (i local-iteration-count)
      )
    ;----------------------------------------
   ))

  (setq time-per-instruction-in-musec (/ total-time local-iteration-count .000001))
  (pformat OS
"LOOP DURATION time (dotimes (i *iteration-count*)):
~12F musec (total ~12F secs for ~A iterations).~%"
   time-per-instruction-in-musec
   total-time
   *iteration-count*
  )
  (setq *loop-duration* (/ total-time 100))
))



;;;  This routine is the same as the last,
;;;  it just runs through more iterations
;;;  to make sure.

(defun extra-time-loop-by-itself ()

  (let ((local-iteration-count  (* *iteration-count* 1000)))
   (setq total-time (time-it                                    ;in seconds
    ;---------TIMED FUNCTION--------------------
      (dotimes (i local-iteration-count)
      )
    ;----------------------------------------
   ))

  (setq time-per-instruction-in-musec (/ total-time local-iteration-count .000001))
  (pformat OS
"EXTRA LOOP DURATION time (dotimes (i *iteration-count*)):
~12F musec (total ~12F secs for ~A iterations).~%"
   time-per-instruction-in-musec
   total-time
   *iteration-count*
```

```
  )
  (setq *loop-duration* (/ total-time 1000))
))



;;;  This routine fills the arrays and variables with values.
;;;  It is useful in case these values get changed between runs.
;;;
(defun init-arrays ()
  (setf (aref array1 0) 1)
  (setf (aref array10 0) 1)
  (setf (aref array100 0) 1)
  (setf (aref array1000 0) 1)

  (setf (aref array10 9) 1)
  (setf (aref array100 9) 1)
  (setf (aref array1000 9) 1)

  (setf (aref array100 99) 1)
  (setf (aref array1000 99) 1)

  (setf (aref array1000 999) 1)


  (setq B 'A)
  (setq c 5)
  (setq D 85.1234567890)
  (setq E 0)
)
```

```
;;;
;;;   THE MAIN TEST ROUTINE:  TIMINGS
;;;


;;;
;; ****** Uncomment out any routines that you want tested.  *******
;;;


;This single routine is the important part of the main program.
; It sets up the arrays, initializes the important variables,
; and then starts timing routines.
;

(defun benchtest1-routine ()
"This routine times the execution durations of important instruction types.
"

   ;First of all, set up the internal variables so that the timing routines
   ;will work properly.

  ;Sets up the scaling variable.
  (setq *iteration-count-per-mu* (/ *iteration-count* 1000000))
  ;Not used; replaced by set in time-loop.
  (setq *loop-duration* (* *iteration-count* *secs-per-iteration*))

  (init-arrays)

  (time-loop-by-itself)
;  (extra-time-loop-by-itself)



;;;  Here are some routines to be timed that DON'T work!!!
;;;  Don't try them again!!!

;;;  (super-time-routine  NIL    )
;;;  Doesn't work for some reason...unclear why.
;;;  (super-time-routine  'A   )
;;;  error:  "Warning: Tag A is never referenced."
;;;  (super-time-routine  1     )
;;;  error:  "Warning: Tag 1 is never referenced."
;;;  (super-time-routine  1.1     )
;;;  error: "illegal Tag-body 1.1"
```

```
;;;!! (time-routine  (setq a  0.11111111111111111111111111111111111111) )
;;; error:  The SEQUENT does not support entering bignums!!!! Number too long!!!


  ;Note:  Extra-time is the default, for routines from 1 musec to 100 musec.


;;  Here we find out whether there is a difference in the SETQ time
;;  between assigning different things to a variable.
;;
;;  (extra-time-routine  (setq a 'b)    )   ;Assign a quoted constant.
;;  (extra-time-routine  (setq a b)     )   ;Assign a quoted constant in a variable.
;;  (extra-time-routine  (setq a c)     )   ;Assign an integer in a variable.
;;  (extra-time-routine  (setq a d)     )   ;Assign a float in a variable.




;;  Here we start to find out about how progn works.
;;
  (setq A NIL)
  (extra-time-routine  (progn 'a )   )
  (extra-time-routine  (progn  a )   )


;;   Here we find out about logic statements.
;;   Note that each logic statement is executed all the way through in both cases.
;;   This is different from, say, (or a a a) when A is T, or (and a a a) when A is NIL.
;;
  (setq A NIL)
  (extra-time-routine  (or  a a a)      )
  (setq A T)
  (extra-time-routine  (and  a a a)      )



;;  Here we try to find out how long different primitive commands take,
;;  such as 'A, an integer 1, a float 1.1, or NIL.
;;  Because it is impossible to time these instructions by themselves,
;;  it is necessary to encase them in a progn.
;;
;  (super-time-routine  (progn )   )
;  (super-time-routine  (progn () )   )
;   (super-time-routine  (progn *this-lwp*)     )
;  (super-time-routine  (progn 1 )   )
;  (super-time-routine  (progn 1.1 )   )
```

```
;  (super-time-routine  (progn (progn 'a) )   )
;  (super-time-routine  (progn () ())   )
;  (super-time-routine  (progn 'a 'b)   )
;  (extra-time-routine  (progn 'a )   )
;  (super-time-routine  (progn 'a )   )
;  (extra-time-routine  (progn NIL )   )
;  (extra-time-routine  (progn NIL NIL)   )

;;  Here we try to pin down how long it takes to execute a single
;;  quoted variable.  A quoted variable is repeated 2 times, 5 times,
;;  10 times, and 100 times.  The 100 times version should override
;;  the side-effect of having a progn at the beginning.
;;
;  (extra-time-routine  (progn 'a 'b)   )
;  (extra-time-routine  (progn 'a 'b 'c 'd 'e)   )
;  (extra-time-routine  (progn 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j)   )
;  (extra-time-routine  (progn 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;'a 'b 'c 'd 'e 'f 'g 'h 'i 'j
;       )   )


;;  How long does it take to execute a progn?
;;
;  (extra-time-routine
;   (progn(progn(progn(progn(progn(progn(progn(progn(progn(progn *this-lwp*))))))))))
```

```
;;  Here we start to work with lists.
;;  Let's try to find out the difference between CAR and CDR.
;;  Is there any difference in precision between timing the routines
;;  with time- and with extra-time- ?  [Yes, there is.]
;;
;  (setq a '(1 2 3 4 5 6 7 8 9 10))
;   (time-routine  (car A)    )
```

```
;   (extra-time-routine  (car A)    )
;   (time-routine  (cdr A)    )
;   (extra-time-routine  (cdr A)    )


;; Here we explore how long it takes to access various parts of a list.
;; It turns out that this number varies, depending upon the rank in the list.
;; This will be important when we compare it against structures and arrays later.
;;
;  (setq a '(1 2 3 4 5 6 7 8 9 10))
;  (extra-time-routine  (first A)    )
;  (extra-time-routine  (second A)    )
;  (extra-time-routine  (third A)    )
;  (extra-time-routine  (fourth A)    )
;  (extra-time-routine  (fifth A)    )
;  (extra-time-routine  (sixth A)    )
;  (extra-time-routine  (seventh A)    )
;  (extra-time-routine  (eighth A)    )
;  (extra-time-routine  (ninth  A)    )


;;  Here are some more list instructions.
;;
;   (super-time-routine  (cons 'a 'b)    )
;   (super-time-routine  (list 'a)    )
;   (extra-time-routine  (list nil nil nil nil nil nil nil)    )
;   (extra-time-routine  (list 'nil 'nil 'nil 'nil 'nil 'nil 'nil)    )
;   (setq a (list 'c))
;   (super-time-routine  (rplacd a 'b)    )

;;  How about DOTIMES?  This is extremely fast, so use a super-time-routine.
;;
;   (super-time-routine  (dotimes (i 0))    )
;   (super-time-routine  (dotimes (i 1))    )
;   (extra-time-routine  (dotimes (i 10))    )

;; Let's test a dolist also.
;;
;  (setq a '(1 2 3 4 5 6 7 8 9 10))
;
;  (time-routine  (dolist (i A))    )


;; Here we start to look at a LET.
;;
;  (extra-time-routine  (let  ((a 'b) )  )    )
```

```
;  (extra-time-routine  (let  ((a 'b) (c 'd))  )    )


;;  How about the Beholder routines ++ and --?
;;
;  (setq E 0)
;  (extra-time-routine  (++  E)    )    ;Note that this can get quite large.
;  (extra-time-routine  (--  E)    )    ;And this one can get quite small.
;  (setq E 0)
;       This stays around 0.  Does it make a difference? (No).
;  (extra-time-routine  (progn  (++  E)(-- E))    )
  (setq E 0)
;  (extra-time-routine  (++1 E sp-lockA)    )
;  How about incrementing with a spin-lock?


;;  Here we look at more Beholder routines.
;;
;  (time-routine  (get-time)    )
;  (time-routine  (random 10)    )
;  (time-routine  (rand   10)    )


;;  Here we look at how long it takes to use a spin-lock.
;;
;  (time-routine  (with-spin-lock sp-lockA () )    )
;  (extra-time-routine  (with-spin-lock sp-lockA () )    )


;; What happens if with-spin-lock is broken out into its important parts?
;;
;  (time-routine  (progn (pp::acquire-spin-lock sp-lockA)   ()
;                        (pp::release-spin-lock sp-lockA)   ) )
;  (extra-time-routine  (progn (pp::acquire-spin-lock sp-lockA)  ()
;                              (pp::release-spin-lock sp-lockA)  ) )

;;  Here we look at some conditionals.
;;  There are various different ways of switching control in a program.
;;
;   (setq A nil)
;   (extra-time-routine  (if a 'b 'c)    )
;   (extra-time-routine  (when a 'b)     )
;   (extra-time-routine  (and a)        )
;   (extra-time-routine  (and a a)      )
;   (extra-time-routine  (and a a a)    )
;   (extra-time-routine  (or  a )      )
;   (extra-time-routine  (or  a a)      )
```

```
;   (extra-time-routine  (or  a a a)      )

;;  These are more conditionals.
;;  Does it matter if the conditional argument A is true or false?
;;
;   (setq A T)
;   (extra-time-routine  (if a 'b 'c)    )
;   (extra-time-routine  (when a 'b)     )
;   (extra-time-routine  (and a)        )
;   (extra-time-routine  (and a a)      )
;   (extra-time-routine  (and a a a)    )
;   (extra-time-routine  (or  a )      )
;   (extra-time-routine  (or  a a)      )
;   (extra-time-routine  (or  a a a)      )
;;
;;   [Surprise!  It does matter--the times are quite different.]

;;  Let's test the burn-cycles routine from Beholder.
;;
;  (time-routine  (burn-cycles 500)    )
;  (time-routine  (burn-cycles 1000)    )
;




;;Here we set A to a list that is 100 elements long.
;;
   (setq a '(1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10
     1 2 3 4 5 6 7 8 9 10 ))
;; Let's look at the function NTH.  How is it different
;; from FIRST, SECOND, etc.?
;;
;   (extra-time-routine  (nth 1  A)    )
;   (extra-time-routine  (nth 2  A)    )
;   (extra-time-routine  (nth 3  A)    )
;   (extra-time-routine  (nth 4  A)    )
;   (extra-time-routine  (nth 5  A)    )
;   (extra-time-routine  (nth 6  A)    )
```

```
;   (extra-time-routine  (nth 7  A)    )
;   (extra-time-routine  (nth 8  A)    )
;   (extra-time-routine  (nth 9  A)    )
;   (extra-time-routine  (nth 10 A)    )
;   (extra-time-routine  (nth 100 A)    )
;;
;;  How is NTH 100 different from AREF 100? (See the text).

; Does the length of the list make any difference when referencing
; parts of the list?
;  (extra-time-routine  (first A)    )
;  (extra-time-routine  (second A)    )
;  (extra-time-routine  (third A)    )
;  (extra-time-routine  (fourth A)    )
;  (extra-time-routine  (fifth A)    )
;  (extra-time-routine  (sixth A)    )
;  (extra-time-routine  (seventh A)    )
;  (extra-time-routine  (eighth A)    )
;  (extra-time-routine  (ninth  A)    )
;  (extra-time-routine  (tenth  A)    )
;  [No.]



;; Here we look at the difference between EQUAL and EQ,
;; and also between true and false tests.
;;
;  (super-time-routine  (equal 'A 'A))
;  (super-time-routine  (equal 'A 'NOT-A))
;  (super-time-routine  (eq 'A 'A))
;  (super-time-routine  (eq 'A 'NOT-A))
;  (extra-time-routine  (equal '(A B C) '(A B C)))
;  What if a list of length 3 is used?


;;  Here we test variables being evaluated to their contents,
;;  when the contents are simple quoted atoms.
;;  Ten evaluations are done inside a PROGN.
;;
;  [remember this is the same as (setq a 'b)(setq b 'c)(setq c 'd) etc.]
;  (setq a 'b  b 'c  c 'd  d 'e  e 'f  f 'g  g 'h  h 'i  i 'j  j 'k)
;  (extra-time-routine  (progn A B C D E F G H I J))

;;  Here we test the EVAL routine on a nesting of atoms 10 deep.
;;  (The results of the routine are K ).
;;
```

```
;  (extra-time-routine (eval(eval(eval(eval(eval(eval(eval
;                                (eval(eval(eval A)))))))))))
;;  How about a single EVAL?
;
;  (setq b 'a)
;  (time-routine  (eval b)   )
;


;;  Here we check out normal hash-tables based on eql,
;;  and special hash-tables based on equal.
;;
;  (extra-time-routine  (make-hash-table))
;Default is eql, which doesn't do lists.
;  (extra-time-routine  (make-hash-table :test 'equal))
;Default is eql, which doesn't do lists.

;;  Let's do some more tests on these kinds of hash-tables.
;;  Does it make any difference if we are searching for something that is there,
;;  or something that is not there?  How about if we keep entering an entry in the
;;  table, that is already there?  (It would also make a good test
;;  to fill up the hashtable
;;  with 100 or 1000 entries, and then run the same tests.  Do this next time.)
;;
;  (defvar hash-eql)
;  (setq hash-eql (make-hash-table))
;  (extra-time-routine  (gethash 'NOT-THERE hash-eql))
;  (extra-time-routine  (setf (gethash 'ONLY-ONE-THERE hash-eql) 'A))
;  (extra-time-routine  (gethash 'ONLY-ONE-THERE hash-eql))
;;
;  (defvar hash-equal)
;  (setq hash-equal (make-hash-table))
;  (extra-time-routine  (gethash 'NOT-THERE hash-equal))
;  (extra-time-routine  (setf (gethash 'ONLY-ONE-THERE hash-equal) 'A))
;  (extra-time-routine  (gethash 'ONLY-ONE-THERE hash-equal))


;;  Here we start to test mailboxes.
;;
;  (time-routine  (make-mailbox)    )
;;
;;  How long does it take to use an empty mailbox?
;;
;  (setq MailboxA (make-mailbox))
;  (time-routine  (progn (mb-send NIL MailboxA)
;                  (mb-receive MailboxA))   )
```

```
;;  Here we test the Beholder fast mailboxes.
;
;  (time-routine  (make-fmailbox)    )
;
;  (setq MailboxB (make-fmailbox))
;  (time-routine  (progn (fmb-send NIL MailboxB)
;                  (fmb-receive MailboxB))   )


;;  Here we test how long it takes to send 10,000 messages to a fast mailbox
;;  that starts out empty.
;;
;  (low-time-routine  (fmb-send NIL MailboxB)    )
;
;  How about if it starts out with 10,000 messages in it?
;  (low-time-routine  (fmb-send NIL MailboxB)    )
;
;; Here we test how long it takes to receive messages
;;   from a 20,000-message mailbox.
;  (low-time-routine  (fmb-receive MailboxB) )
;;  Or one with 10,000 messages in it.
;  (low-time-routine  (fmb-receive MailboxB) )


;;  These are the same tests, except with the regular mailboxes, (not fast).
;;
;  (low-time-routine  (mb-send NIL MailboxA)    )
;  (low-time-routine  (mb-send NIL MailboxA)    )
;  (low-time-routine  (mb-receive MailboxA) )
;  (low-time-routine  (mb-receive MailboxA) )


;;  Here we test popping a null list a couple thousand times.
;;
;  (setq a '())
;  (extra-time-routine (progn (pop a) ))

;;  Here we test pushing 100,000 atoms onto a stack.
;;  (setq a '())
;;  (time-routine (push 'b a))
;;
;;  Here we test pushing an atom onto a stack and then clearing the stack.
;;  (setq a '())
;;  (time-routine (progn (push 'b a) (setq a '())))
;;  Pushing an atom onto the stack and then popping the stack.
```

```
;;  (time-routine (progn (push 'b a) (pop a)))
;;  Pushing 10,000 atoms onto the stack.
;;  (time-routine (progn (push 'b a) ))
;;  Popping 10,000 atoms off of the stack.
;;  (time-routine (progn (pop a) ))

;;  Get extra precision by repeating the previous tests more times.
;  (extra-time-routine (progn (push 'b a) (setq a '())))
;  (extra-time-routine (progn (push 'b a) (pop a)))
;  (extra-time-routine (progn (push 'b a) ))
;  (extra-time-routine (progn (pop a) ))

;
;;  Here we test basic math comparison routines.
;;  Does it matter whether <, <=, or > is used?
;;  Does it matter whether the test constant is zero,
;;  a non-zero integer, or a floating-point number?
;;
;  (setq c 5)
;  (extra-time-routine (<= c 0) )
;  (extra-time-routine (<= c 3) )
;  (extra-time-routine (<= c 5) )
;  (extra-time-routine (<= c 10) )
;  (extra-time-routine (< c 0) )
;  (extra-time-routine (< c 3) )
;  (extra-time-routine (< c 10) )
;  (extra-time-routine (> c 0) )
;  (extra-time-routine (> c 5) )
;  (extra-time-routine (> c 10) )
;  (extra-time-routine (<= c 2.38) )
;
;;  Here we test the Beholder routine my-string and the string-append routine,
;;  with a string name that is one character long.
;;
;  (setq a 'i)
;  (time-routine (my-string a) )
;  (setq a (my-string a))
;  (time-routine (string-append a a))
;
;;  Same test, ten characters long.
;;
;  (setq a 'ten-charst)
;  (time-routine (my-string a) )
;  (setq a (my-string a))
;  (time-routine (string-append a a))
;
;;  Same test, 100 characters long.
```

```
;;
;  (setq a
;'a-string-that-is-one-hundred-characters-long-and-takes-up-a-lot-of-space-abcdef
;  (time-routine (my-string a) )
;  (setq a (my-string a))
;  (time-routine (string-append a a))
;
;
;;  How about the Beholder PFORMAT command, with short and long messages?
;;
;  (time-routine (pformat NIL "A short message."))
;  (time-routine (pformat NIL
;    "A long message, ~A ~A ~A ~A ~A with five evaluations." b c b c b))
;

;;  Here we test some of the CLiP routines.
;  (extra-time-routine (progn *this-lwp*))
;  (extra-time-routine (make-spin-lock))
;  (extra-time-routine (make-sleepy-lock))
;  (extra-time-routine (make-fork))


;;  Here we investigate structures.
;;
;   Making a structure.
;  (extra-time-routine (make-structA) )
;   Making a structure and assigning it to something.
;  (extra-time-routine (setq structB (make-structA)) )
;   Testing for structure type.
;  (extra-time-routine (structA-p structB) )
;
;  Referencing the contents of the structure.
;  Does it matter which structure slot is used?  [No.]
;  (extra-time-routine  (structA-slot1 structB)  )
;  (extra-time-routine  (structA-slot2 structB)  )
;  (extra-time-routine  (structA-slot3 structB)  )
;
;  Setting the contents of the structure.
;  Does it matter which structure slot is used?  [No.]
;  (extra-time-routine (setf (structA-slot1 structB) NIL) )
;  (extra-time-routine (setf (structA-slot2 structB) NIL) )
;  (extra-time-routine (setf (structA-slot3 structB) NIL) )



)
```

```
;;;
;;;   THE ACTUAL PROGRAM CALLED BY THE USER
;;;


;; This routine forks a process to take care of the timing.
;; It then runs this process.

(defun benchtest1 ()
  (spawn-lwp  (benchtest1-routine) )
  (start-lwps)
)
```

B Sample Run of Program

```
% cd /usr2/myers/nl
% clip
Allegro CLiP 3.0.3 [sequent] (6/27/91 12:37)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA, USA
; Fast loading /usr/local/lib/clip/code/flavors.fasl.

========================================================================
This is a revised implementation of Franz Inc Allegro Common Lisp Flavors:
 - Eliminates problems with compile-time flavor definitions altering
   flavor definitions in the running lisp.
 - Provides the :ALTERNATE-COMPONENT-ORDERING option to DEFFLAVOR.
   If specified, that flavor (but not necessarily its descendents) will
   emulate the flavor component ordering of Symbolics Genera 7 flavors.
   This feature may be enabled globally by setting the variable
   FLA::*ALTERNATE-COMPONENT-ORDERING* to T.

This software is in beta release:
  Report all problems, bugs, and questions about this beta version to
  cl-bugs%franz.uucp@berkeley.edu, (415) 548-3600.
========================================================================
; Fast loading /usr/local/lib/clip/code/vanilla.fasl.
; Fast loading /usr2/myers/nl/system.fasl.
; Fast loading /usr2/myers/nl/beholder-2C.fasl.
; Fast loading /usr/local/lib/clip/code/spurlisp.fasl.
; Fast loading /usr2/myers/nl/heap.fasl.
; Fast loading /usr2/myers/nl/new-flavors.fasl.
; Fast loading /usr/local/lib/clip/code/trace.fasl.
; Fast loading /usr2/myers/nl/lf-gen-sq.fasl.
; Loading /usr2/myers/nl/agent-nl.lisp.
; Fast loading /usr2/myers/nl/okuma/tdmt/patches/patch-for-setq-default.fasl.
; Fast loading /usr2/myers/nl/okuma/tdmt/patches/patch-for-readtable.fasl.
<Initial lwp>

<Initial lwp> :cl test-timing-bench-1

; --- Compiling file /usr2/myers/nl/test-timing-bench-1.lisp ---
; Compiling MAKE-STRUCTA
; Compiling STRUCTA-P
; Compiling GET-TIME
; Compiling TIME-IT
; Compiling TIME-IT-IN-TICKS
; Compiling TIME-ROUTINE
; Compiling EXTRA-TIME-ROUTINE
; Compiling LOW-TIME-ROUTINE
; Compiling SUPER-TIME-ROUTINE
```

```
; Compiling TIME-LOOP-BY-ITSELF
; Compiling EXTRA-TIME-LOOP-BY-ITSELF
; Compiling INIT-ARRAYS
; Compiling BENCHTEST1-ROUTINE
; Compiling BENCHTEST1
; Writing fasl file "/usr2/myers/nl/test-timing-bench-1.fasl"
; Fasl write complete
; Fast loading /usr2/myers/nl/test-timing-bench-1.fasl.
Warning: redefining 'GET-TIME', originally defined in /usr2/myers/nl/beholder-2C.lisp
Warning: redefining 'TIME-IT', originally defined in /usr2/myers/nl/beholder-2C.lisp
Warning: redefining 'TIME-IT-IN-TICKS', originally defined in /usr2/myers/nl/beholder-2C
<Initial lwp>

<Initial lwp> (benchtest1)
Using 1 processor
LOOP DURATION time (dotimes (i *iteration-count*)):
                     1.925 musec    (total        19.25 secs for 100000 iterations).
(PROGN 'A):    0.161 musec    EXTRA(total         1.61 secs for 10000000 iterations).
(PROGN A):     8.302 musec    EXTRA(total        83.02 secs for 10000000 iterations).
(OR A A A):   23.742 musec    EXTRA(total       237.42 secs for 10000000 iterations).
(AND A A A):   21.937 musec    EXTRA(total       219.37 secs for 10000000 iterations).
Warning: All (1) live lwps are on wait queue

NIL
<Initial lwp>
```

C Durations Ranked by Command Name

The following results are for compiled code run in a subprocess.

Results are ordered alphabetically by the name of the main command. Some results, with (PROGN COMMAND), are entered twice, once under progn and once under command.

A microsecond (abbreviated "musec") is 0.000001 second.

Timing results under 1 musec are probably not reliable.

Standard significant-digit conventions are followed in representing the timing information. All significant digits are shown. Partially significant digits are represented as a trailing 0 or 5 (i.e., +/- 0.25). A trailing decimal point means the estimate is accurate to about +/- 0.5; no trailing decimal indicates the estimate is accurate to +/- 2. or more.

```
(<= C 0):                                  7.5  musec       [C = 5]  ;Possibly = 0.
(<= C 2.38):                              28.0  musec       [C = 5]  ;Floating.
(<= C 3):                                  6.8  musec       [C = 5]
(<= C 5):                                  6.8  musec       [C = 5]
(<= C 10):                                 6.8  musec       [C = 5]
(< C 0):                                   6.8  musec       [C = 5]
(< C 3):                                   6.8  musec       [C = 5]
(< C 10):                                  6.8  musec       [C = 5]
(> C 0):                                   6.8  musec       [C = 5]
(> C 5):                                   6.8  musec       [C = 5]
(> C 10):                                  6.8  musec       [C = 5]
(++ E):                                   14.6 musec
(++L E SP-LOCKA):                         38   musec
*THIS-LWP*:                                0.2 musec
'A                                         0.01 musec Estimated.
 A                                         6    musec Estimated.
(PROGN (ACQUIRE-SPIN-LOCK SP-LOCKA)
        NIL
       (RELEASE-SPIN-LOCK SP-LOCKA)):     30   musec
(AND A A A):  [A = T]:                    22   musec
(AREF ARRAY1  0):                         11.5 musec
(AREF ARRAY10 0):                         11.5 musec
(AREF ARRAY10 9):                         11.5 musec
(AREF ARRAY100 0):                        11.5 musec
(AREF ARRAY100 9):                        11.5 musec
(AREF ARRAY100 99):                       11.5 musec
(AREF ARRAY1000 0):                       11.5 musec
(AREF ARRAY1000 9):                       11.5 musec
(AREF ARRAY1000 99):                      11.5 musec
(AREF ARRAY1000 999):                     11.5 musec
(BURN-CYCLES 500):                      1006.5  musec
(BURN-CYCLES 1000):                     2009.5  musec
(CAR A):                                  6.70 musec
(CONS 'A 'B):                             1.8  musec
(CDR A):                                  6.65 musec
```

```
(DIV2 15):                                  17    musec
(DOLIST (I A) ):                            29.1  musec [A = '(1 2 3 4 5 6 7 8 9 10)]
(DOTIMES (I 0)):                             1.46 musec
(DOTIMES (I 1)):                             2.7  musec
(DOTIMES (I 10)):                           17.5  musec
(DOTIMES (I 10)):                           18.675 musec
(DOTIMES (I iteration-count)):               1.925 musec
   [once through the iteration]
(EIGHTH  A):                                10.   musec
(EQ 'A 'A):                                  0.22 musec
(EQ 'A 'NOT-A):                              0.22 musec
(EQUAL 'A 'A):                               0.15 musec
(EQUAL 'A 'NOT-A):                           0.25 musec
(EQUAL '(A B C) '(A B C)):                  46.5  musec
(EVAL B):                                   44.0  musec
(EVAL (EVAL (EVAL (EVAL (EVAL (EVAL (EVAL
  (EVAL (EVAL (EVAL A))))))))):            450.   musec
(FIFTH   A):                                 8.5  musec
(FIRST   A):                                 6.7  musec
(FMB-RECEIVE MAILBOXB):                     90    musec
(FMB-SEND NIL MAILBOXB):                 100-190  musec    [Depends on swapping.]
(PROGN (FMB-SEND NIL MAILBOXB) (FMB-RECEIVE MAILBOXB)):      110 musec    (?!)
(FOURTH  A):                                 8.   musec
(GETHASH 'NOT-THERE HASH-EQL):              137   musec    ;Hash table empty.
(SETF (GETHASH 'ONLY-ONE-THERE HASH-EQL) 'A):      150 musec    ;Table: One entry.
(GET-TIME):                                 159.  musec
(IF A 'B 'C)  [A = NIL]:                     8.0  musec
(IF A 'B 'C)    [A = T]:                     6.6  musec
(LET ((A 'B) )):                             2.4  musec
(LET ((A 'B) (C 'D))):                       2.7  musec
(LIST 'A):                                   1.6  musec
(LIST 'NIL 'NIL 'NIL 'NIL 'NIL 'NIL 'NIL):     14 musec
(LIST NIL NIL NIL NIL NIL NIL NIL):          15  musec
(MAKE-ARRAY 1):                              60  musec    [Depends on swapping.]
(MAKE-ARRAY 1 INITIAL-ELEMENT 0):           120  musec    [Depends on swapping.]
(MAKE-ARRAY 10):                             65  musec    [Depends on swapping.]
(MAKE-ARRAY 10 INITIAL-ELEMENT 0):          130  musec    [Depends on swapping.]
(MAKE-ARRAY 100):                           125  musec    [Depends on swapping.]
(MAKE-ARRAY 100 INITIAL-ELEMENT 0):         190  musec    [Depends on swapping.]
(MAKE-ARRAY '(10 10)):                      335  musec    [Depends on swapping.]
(MAKE-ARRAY '(10 10) INITIAL-ELEMENT 0):  420 musec [Depends on swapping.]
(MAKE-FMAILBOX):                            35   musec    [Depends on swapping.]
(MAKE-FORK):                                27   musec    [Depends on swapping.]
(MAKE-HASH-TABLE):                         223   musec    [Depends on swapping.]
(MAKE-HASH-TABLE TEST 'EQUAL):             259   musec    [Depends on swapping.]
(MAKE-MAILBOX):                             14   musec    [Depends on swapping.]
(MAKE-SLEEPY-LOCK):                         17   musec    [Depends on swapping.]
```

```
(MAKE-SPIN-LOCK):                          13   musec    [Depends on swapping.]
(MAKE-STRUCTA):                            15   musec    [Depends on swapping.]
(MB-RECEIVE MAILBOXA):                    140   musec  [Apparently varies with swapping
(MB-SEND NIL MAILBOXA):                140-200   musec    [Depends on swapping.]
(PROGN (MB-SEND NIL MAILBOXA)
       (MB-RECEIVE MAILBOXA)):             300   musec [with 0, 5, or 10 in queue]
(MY-STRING A):                            1295   musec    [A  = 'i]
(MY-STRING A):                            1560   musec    [A  = 'ten-charst]
(MY-STRING A):                            4230   musec    [A is 100 chars long]
(NINTH   A):                              10.5 musec
(NTH 1 A):          7.15 musec
(NTH 2 A):          7.40 musec
(NTH 3 A):          7.90 musec
(NTH 4 A):          8.9  musec
(NTH 5 A):          8.8  musec
(NTH 6 A):          9.2  musec
(NTH 7 A):         10.6  musec
(NTH 8 A):         10.0  musec
(NTH 9 A):         10.3  musec
(NTH 10 A):         40.877 musec
(NTH 100 A):       300.068 musec
(NTH 1 A):                                 7.2  musec
(NTH 5 A):                                 8.8  musec
(NTH 10 A):                               40.   musec
(NTH 100 A):                             299.   musec
(OR A A A):    [A = NIL]                   24    musec
(PFORMAT NIL "A short message."):         415    musec
(PFORMAT NIL "A long message, ~A ~A ~A ~A ~A with five evaluations."
   B C B C B):                           1925     musec
(PROGN *THIS-LWP*):                       5.90  musec
(PROGN (PROGN (PROGN (PROGN (PROGN (PROGN (PROGN (PROGN
  (PROGN (PROGN *THIS-LWP*))))))))):      6.02  musec
(POP A):                                  20.    musec Estimate.
(PROGN (POP A)):  [A is NIL]              20.5   musec
(PROGN (POP A)):  [A is full]             27.5   musec
(PROGN  A):                                5.90  musec
(PROGN 'A):                                0.1   musec (?)
(PROGN 'A 'B):                             0.1   musec (?)
(PROGN 'A 'B 'C 'D 'E 'F 'G 'H 'I 'J): 0.05  musec (?!)
(PROGN 'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
```

```
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J
       'A 'B 'C 'D 'E 'F 'G 'H 'I 'J): 0.05 musec (?!)
(PROGN (ACQUIRE-SPIN-LOCK SP-LOCKA)
        NIL
       (RELEASE-SPIN-LOCK SP-LOCKA)): 30   musec
(PROGN (MB-SEND NIL MAILBOXA)
       (MB-RECEIVE MAILBOXA)):       300   musec [with 0, 5, or 10 in queue]
(PROGN NIL):                          0.2 musec
(PROGN NIL NIL):                      0.12, 0.200 musec
(PROGN (POP A)):  [A is NIL]         20.5 musec
(PROGN (POP A)):  [A is full]        27.5 musec
(PROGN (PUSH 'B A) (POP A)):         43 musec
(PROGN (PUSH 'B A) (SETQ A 'NIL)):   28.. musec
(PROGN (PUSH 'B A)):                      30   musec          [A fills up]
(PUSH 'B A):                         21.  musec
(PROGN (PUSH 'B A) (POP A)):         43  musec
(PROGN (PUSH 'B A) (SETQ A 'NIL)):   28.  musec
(RAND 10):                          985   musec
(RANDOM 10):                        938   musec
(RPLACD A 'B):                        7.5 musec
(SECOND A)                           7.  musec
(SETF (AREF ARRAY1 0) 1):           11.7 musec
(SETF (AREF ARRAY10 0) 1):          11.9, 13.3 musec
(SETF (AREF ARRAY10 9) 1):          11.7 musec
(SETF (AREF ARRAY100 0) 1):         11.7 musec
(SETF (AREF ARRAY100 9) 1):         11.9, 13.3 musec
(SETF (AREF ARRAY100 99) 1):        11.8 musec
(SETF (AREF ARRAY1000 0) 1):        11.8 musec
(SETF (AREF ARRAY1000 9) 1):        11.7 musec
(SETF (AREF ARRAY1000 99) 1):       11.5 musec
(SETF (AREF ARRAY1000 999) 1):      11.5, 13.3 musec
(SETF (GETHASH 'ONLY-ONE-THERE HASH-EQL) 'A):      150 musec ;Table: One entry.
(SETF (STRUCTA-SLOT1 STRUCTB) NIL):  16.5 musec
(SETF (STRUCTA-SLOT2 STRUCTB) NIL):  16.5 musec
(SETF (STRUCTA-SLOT3 STRUCTB) NIL):  16.5 musec
(SETQ A 1):                           6.4 musec
(SETQ A 0.111):                      13.4 musec
(SETQ A 'B):                          7.2 musec
(SETQ A B):                          14   musec    [B = 'A, or 5, or 0.1234567890]
(SEVENTH A):                          9.5 musec
(SIXTH   A):                          9.  musec
(STRING-APPEND A A):                190   musec    ;A is 1 char long
(STRING-APPEND A A):                415   musec    ;A is 10 chars long
(STRING-APPEND A A):               2670   musec    ;A is 100 chars long
(STRUCTA-SLOT1 STRUCTB):             16.5 musec    ;All slots the same.
(STRUCTA-SLOT2 STRUCTB):             16.5 musec
(STRUCTA-SLOT3 STRUCTB):             16.5 musec
```

```
(TENTH   A):                          11.   musec
(THIRD   A):                           7.5  musec
*THIS-LWP*:                            0.2  musec
(WHEN A 'B)    [A = NIL]:              7.1  musec
(WHEN A 'B)      [A = T]:              6.4  musec
(WITH-SPIN-LOCK SP-LOCKA NIL):        23    musec
```