Star Forth Test Suite

Current version: 1.0

Benchmarking

Benchmark	Words/Sec.	Notes
SimpleBench	1.3 mil/sec	loop overhead
Stackbench	725,000	Stack operations
FizzBench	460,000	Branch-heavy code
MathBench	800,000	Math-heavy code and bitwise ops
MemBench	330,000	Memory-access heavy code

The SD-8516 defaults to 4 MHz. If your program has a lot of general purpose computation it should operate at about 500,000 words per second. However for branch heavy, memory heavy (ex. sorting, line drawing algorithms, text processing) you might drop down to 400,000 words/sec or so. This mirrors normal language performance where branching and memory access are usually much slower than math and stack ops.

It's hard to say exactly how this compares in speed to, say, a C64 running COMMODORE BASIC V2.0. But considering lines per second, comparable 8-bit Forths for the C64 and other 80s microcomputers, and accounting for register pressure and differences in MHz, I'd say this is a reasonably fast Forth.

FORTH-79 TESTS

If you see all PASS and then ABC and then A on the next line underneath the space beside the C, it works.

```
: TEST= = IF ." PASS " ELSE ." FAIL " THEN ;

( Arithmetic )
5 1+ 6 TEST=
5 1- 4 TEST=
5 2+ 7 TEST=
5 2- 3 TEST=
5 NEGATE -5 TEST=
-7 NEGATE 7 TEST=
-5 ABS 5 TEST=
5 ABS 5 TEST=

( Comparisons )
0 0= 1 TEST=
5 0= 0 TEST=
-1 0< 1 TEST=
5 0< 0 TEST=
```

```
5 0> 1 TEST=
-1 0> 0 TEST=

( Stack ops )
1 2 NIP 2 TEST=
1 2 TUCK 2 TEST= DROP DROP
1 2 TUCK DROP 1 TEST= DROP
1 2 TUCK DROP DROP 2 TEST=
5 ?DUP + 10 TEST=
0 ?DUP 0 TEST=
1 2 3 ROT 1 TEST= DROP DROP
3 4 2DUP + >R + R> + 14 TEST=

( Math )
7 3 MAX 7 TEST=
3 7 MAX 7 TEST=
7 3 MIN 3 TEST=
3 7 MIN 3 TEST=
7 3 /MOD SWAP 1 TEST=
7 3 /MOD 2 TEST=
DROP

( Variable and +! )
VARIABLE TV
10 TV !
3 TV +!
TV @ 13 TEST=

( SPACE and SPACES )
65 EMIT 66 EMIT 67 EMIT CR
3 SPACES 65 EMIT CR

( Bit Shifting Ops )
1 0 LSHIFT 1 TEST=
1 1 LSHIFT 2 TEST=
1 4 LSHIFT 16 TEST=
1 8 LSHIFT 256 TEST=
$FF 4 LSHIFT $FF0 TEST=
16 1 RSHIFT 8 TEST=
256 8 RSHIFT 1 TEST=
$FF0 4 RSHIFT $FF TEST=
1 1 RSHIFT 0 TEST=


: TEST= = IF ." PASS " ELSE ." FAIL " THEN ;

( DO LOOP basic )
0 10 0 DO I + LOOP 45 TEST=

( DO LOOP with step )
0 10 0 DO I + 2 +LOOP 20 TEST=
```

```
( Nested DO LOOP )
0 3 0 DO 3 0 DO 1 + LOOP LOOP 9 TEST=

( I and J in nested loops )
0 2 0 DO 2 0 DO I J + + LOOP LOOP 4 TEST=

( BEGIN UNTIL )
0 BEGIN 1 + DUP 10 = UNTIL 10 TEST=

( BEGIN WHILE REPEAT )
0 BEGIN DUP 5 < WHILE 1 + REPEAT 5 TEST=

( BEGIN AGAIN with EXIT )
: TEST-EXIT 0 BEGIN 1 + DUP 7 = IF EXIT THEN AGAIN ;
TEST-EXIT 7 TEST=

( IF ELSE THEN )
1 IF 42 ELSE 99 THEN 42 TEST=
0 IF 42 ELSE 99 THEN 99 TEST=

( Nested IF )
1 IF 1 IF 77 ELSE 88 THEN ELSE 99 THEN 77 TEST=
1 IF 0 IF 77 ELSE 88 THEN ELSE 99 THEN 88 TEST=
0 IF 0 IF 77 ELSE 88 THEN ELSE 99 THEN 99 TEST=

( LEAVE )
0 10 0 DO I 5 = IF LEAVE THEN I + LOOP 10 TEST=

( >R R> R@ )
5 >R R@ 5 TEST=
R> 5 TEST=

( Loop accumulator stress )
0 100 0 DO I + LOOP 4950 TEST=
0 1000 0 DO I + LOOP 499500 TEST=

( FizzBuzz count — count how many fizzbuzz hits in 1-100 )
0 101 1 DO I 15 MOD 0= IF 1 + THEN LOOP 6 TEST=

CR ." All loop tests complete" CR
```

SIMPLEBENCH

A simple loop.

```
: SIMPLE
  PERF
  1000000 0 DO LOOP
  PERF SWAP -
  DUP . ." ms" CR
```

```
  2000000000 SWAP / . ." words/sec" CR ;
SIMPLE
```

STACKBENCH

```
: STACKBENCH
  ." Stack benchmark: 300000 x DUP DROP SWAP OVER DROP" CR
  PERF
  1 2 3
  300000 0 DO
    DUP DROP SWAP OVER DROP
  LOOP
  DROP DROP DROP
  PERF SWAP -
  DUP . ." ms" CR
  1800000000 SWAP / . ." words/sec" CR ;
STACKBENCH
```

FIZZBENCH

```
: FIZZBENCH
    PERF >R
    0 12800 0 DO
        I 15 MOD 0 = IF 1 + ELSE
        I 3 MOD 0 = IF 1 + ELSE
        I 5 MOD 0 = IF 1 + THEN
        THEN THEN
    LOOP
    PERF R> -
    SWAP . ." hits in "
    DUP . ." ms" CR
    12800 19 * 1000 * SWAP / . ." words/sec" CR ;

: FB10 10 0 DO FIZZBENCH LOOP ;
: FB1 1 0 DO FIZZBENCH LOOP ;
FB1
```

or 12800000 SWAP / . .“ iterations/sec” CR

MATHBENCH

```
: MATHBENCH
  ." Math: 100000 x 2add 2sub mul div AND XOR" CR
```

```
  PERF >R
  0 100000 0 DO
    I 42 + 13 + 7 - 3 - 6 * 2 / $FF AND $AA XOR +
  LOOP
  DROP
  PERF R> -
  DUP . ." ms" CR
  2000000000 SWAP / . ." words/sec" CR
  ." (9 math ops + 8 literals + I LOOP per iteration)" CR ;
MATHBENCH
```

MEMBENCH

```
VARIABLE M1
VARIABLE M2
VARIABLE M3
VARIABLE M4

: MEMBENCH
  ." Mem: 50000 x 23 memory-heavy words" CR
  PERF >R
  42 M1 ! 43 M2 ! 44 M3 ! 45 M4 !
  50000 0 DO
    M1 @ M2 @ + M3 @ + M4 @ +
    DUP M1 ! DUP M2 ! DUP M3 ! M4 !
  LOOP
  PERF R> -
  DUP . ." ms" CR
  1150000000 SWAP / . ." words/sec" CR
  ." (4@ 4! 3+ 3DUP I LOOP per iteration)" CR ;
MEMBENCH
```

PRIMEBENCH

```
: PRIME? ( n -- flag )
  DUP 2 < IF DROP 0 EXIT THEN
  DUP 2 MOD 0= IF 2 = EXIT THEN
  3
  BEGIN
    2DUP MOD 0= IF DROP DROP 0 EXIT THEN
    2 +
    2DUP DUP * < UNTIL
  DROP DROP 1 ;

: PRIMEBENCH ( n -- )
  DUP . ." primes..." CR
```

```
  PERF >R
  >R 0 2
  BEGIN
    DUP PRIME? IF SWAP 1 + SWAP THEN
    1 +
    OVER R@ >=
  UNTIL
  DROP R> DROP
  PERF R> -
  SWAP . ." primes found in "
  . ." ms" CR ;
100 PRIMEBENCH
```

How I optimized Star Forth

I came up with some tests (see above) then ruthlessly tried to optimize their hot-path.

For starters, I cheated. I made a DPUSH and DPOP command to help Forth push and pop on the data stack. I can do that because I wrote the CPU.

Switching Horses in Mid-Stream

The problem is, that Forth needs to peek at the top of the stack. This undermined the DPUSH and DPOP idea. I needed something more. I then took inspiration from the 68000's LOAD and STEP operation and created LD_STEP which is like

```
  LDA [BLX,+]    ; Load into A from address BLX and increase BLX by +2 (to
match A's length).
```

Simple, this gave a 30% speedup.

Out of my Depth on this one

I then took a look at what else I could optimize about DPUSH and DPOP. I was calculating depth on every dpush and dpop and depth was only really used by .S (user facing), more or less. So I moved it out of there and into .S. This caused an issue where underflows (overflows?) can occur if you don't manage your own stack. Kind of like memory leaks in C. But I added a few clamps here and there. It shouldn't outright crash, but you are responsible for managing the stack.

The ball of yarn unravels further...

But this led to a thought, wait a sec. DPUSH and DPOP are just one opcode now. Why are they being wrapped with a CALL? For that matter, what *else* can I inline? *Any stack only operation.* This led me down a fascinating journey of how to change the compiler to inline code.

> If the compiler recognized DUP, DROP and SWAP (etc) and emitted their bodies inline instead of CALL, it would eliminate two instructions per word. That's a compiler change; check if the word being compiled is “simple” (1-2 instructions), and if so, emit the body directly instead of STC'ing the code.

The fact is, this is too big for me, which is why I am writing this down. If I ever set this down, I will *never* remember how I got here.

- The compilation path for DUP
- : creates header, sets STATE=1
- DUP is found by forth_find, W2 = code address, W3 = entry start
- forth_compile_mode checks IMMEDIATE.
- DUP is not immediate so calls `forth_compile_call`
- `forth_compile_call` writes `CALL @code_dup` (4 bytes) at HERE.
- At runtime, DOUBLE does:
  - CALL @code_dup
  - DPUSH
  - RET
  - CALL @code_plus
- Three instructions – cal, body, ret, for one instruction of work. And its in the hot path. A theoretical X% gain! Oodelally!
- So each instruction gets a new flag. F_INLINE, alongside F_IMMEDIATE, etc.

Plan of Attack

Right before the call to compile CALL, we check for the inline flag in the dictionary then scan for its RET and copy it in (minus the RET)

```
    MOV TL, KL
    AND TL, @F_IMMEDIATE
    ;; CMP TL, #0 ;; previous sets ZF
    JNZ @forth_execute_immediate

        ; Check INLINE flag
        MOV TL, KL
        AND TL, @F_INLINE
        JNZ @forth_compile_inline

    ; Not immediate: compile CALL to the word
    CALL @forth_compile_call
    JMP @forth_interpret
```

Then at the end we can emit the body without the call:

```
forth_compile_inline:
    ; Copy body bytes from code address to HERE, stop at RET
    LDGLZ [@FORTH_HERE]
    LDTLK [@FORTH_W2]          ; TLK = code address (source)

inline_copy:
    LDKL [TLK,+]               ; read byte from source and advance
    CMP KL, @OP_RET            ; hit RET? done
    JZ @inline_done
    STKL [GLZ,+]               ; write byte to HERE and advance
    JMP @inline_copy

inline_done:
    STGLZ [@FORTH_HERE]        ; update HERE
    JMP @forth_interpret
```

inline_copy simply copies the code_(name) function body till it hits a RET. We don't need to write the RET (the + skips it).

The same pattern in EVAL, although I don't think it's as important, but for consistency – right before the CALL to compile CALL, check for F_IMMEDIATE.

So the six words I chose are:

- DUP
- DROP
- SWAP
- OVER
- NOT
- I

All of these are small. Notably, I calls @forth_dpush. This was a problem. In other words, all of this crap I just came up with didn't work.

But then I realized I could inline DPUSH itself. I removed the call to DPUSH and just inlined it – in I. All in all, there were almost 100 calls to DPOP or DPUSH I was able to optimize out, on top of the inlined DPUSH and DPOP itself. A surprising turn of events! Now suddenly I had been led to the idea of inlining dpusp and dpop. After all, it was the only way. The bytes in the CALL could interfere with the copy.

The Results

It was a hard road, but the results were worth it. After several optimizing cycles, performance kept climbing From 100,000 to 140,000, then to 180,000, then to 220,000. Then to 270,000. Then 310,000. Then 340,000. I couldn't believe it. It was ridiculous, but it worked. Suddenly, programming in Forth was fun again! I started optimizing around MATHBENCH next. The results?

Before: 248,725 words/sec. After: 302,984 words/sec. Another 20% performance boost!

I began looking everywhere for places to optimize. Rat. Cbang. Fploop. Places you never even thought

existed. Over time, I realized what I had done. Star Forth had finally come into it's own, and the galaxy would finally have .... peace.