ORNL/TM-10656

ornl

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

Sieve of Eratosthenes Benchmarks for the Z8 FORTH Microcontroller

Robert Edwards

OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

ORNL/TM-10656

Energy Division

SIEVE OF ERATOSTHENES BENCHMARKS FOR THE Z8 FORTH MICROCONTROLLER

Robert Edwards

Date Published - February 1989

Prepared for the
Smart House Project
National Association of Home Builders
Research Foundation

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
operated by
MARTIN MARIETTA ENERGY SYSTEMS, INC.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

ABSTRACT

This report presents benchmarks for the Z8 FORTH microcontroller system that ORNL uses extensively in proving concepts and developing prototype test equipment for the Smart House Project. The results are based on the sieve of Eratosthenes algorithm, a calculation used extensively to rate computer systems and programming languages. Three benchmark refinements are presented, each showing how the execution speed of a FORTH program can be improved by use of a particular optimization technique. The last version of the FORTH benchmark shows that optimization is worth the effort: It executes 20 times faster than the Gilbreaths' widely-published FORTH benchmark program.

The National Association of Home Builders Smart House Project is a cooperative research and development effort being undertaken by American home builders and a number of major corporations serving the home building industry. The major goal of the project is to help the participating organizations incorporate advanced technology in communications, energy distribution, and appliance control products for American homes.

This information is provided to help project participants use the Z8 FORTH prototyping microcontroller in developing Smart House concepts and equipment. The discussion is technical in nature and assumes some experience with microcontroller devices and the techniques used to develop software for them.

TABLE OF CONTENTS

LIST OF TABLES

1. INTRODUCTION

The Oak Ridge National Laboratory (ORNL) is a participant in the Smart House Project, a cooperative effort between the National Association of Home Builders Research Foundation and a number of manufacturers of home-construction products. ORNL is providing technical evaluations of proposed Smart House designs and advice on methods for design evaluation and system integration.

This report presents benchmarks for the Z8 FORTH system (Edwards 1987a, 1987b) that ORNL uses extensively in proving concepts and developing prototype test equipment for the Smart House Project. The results are based on the sieve of Eratosthenes algorithm (Knuth, 1969), a calculation used extensively to rate computer systems and programming languages. The first benchmark presented is taken from one developed by Jim and Gary Gilbreath and presented in *BYTE* magazine (Gilbreath 1981, 1983). Three refinements of the Gilbreaths' program are presented, each showing how the execution speed of a FORTH program can be improved by use of a particular FORTH optimization technique.[1] The last version of the FORTH benchmark shows that optimization is worth the effort: It executes 20 times faster than the widely-published sieve program usually used to test FORTH system performance.

The information in this report is intended to help Smart House participants make effective use of Z8 FORTH for developing and testing Smart House equipment and concepts. The discussion is technical in nature and assumes some experience with microcontroller devices and the techniques used to develop microcontroller software.

---

[1]The Gilbreaths would probably object to presenting the refinements as "benchmark" results. In Jim Gilbreath's September 1981 article, he states, "The program for each language was coded conventionally, taking advantage of features that are defined in the language, but not exploiting the clever or obscure innovations that can make it run faster. In most cases, some improvement in running time could be achieved by knowledgeable trickery. [*In fact, that is the point of a benchmark program: to compare language performance by running the same algorithm encoded in different languages. (BYTE editor)*]." Contrary to the Gilbreaths' and *BYTE* editor's opinions, the objective of computer benchmarking is to permit comparison of expert implementations of an algorithm on particular combinations of computer language and equipment. The algorithm, in this context, is the statement of the process as described by Eratosthenes, not by Gilbreath in a Pascal program. What the Gilbreaths and the *BYTE* editor measured in their benchmarking was a program porting process, in this case from Pascal (or C) to FORTRAN, BASIC, FORTH, and other languages without regard to use of advantageous features of the target language.

2. THE SMART HOUSE PROJECT

The goal of the Smart House project is to help the participants incorporate advanced technology into new products for communications, energy distribution, and appliance control. One of the project's most important goals is to bring Smart House cabling, electrical control devices, consumer appliances, and gas products into the market for the majority of homes constructed in the next century.

The impetus for the project comes from two areas. One is the pressure of foreign consumer products, which dominate the after-sale consumer appliance market and threaten to make inroads into sales to home builders. The other is the increasing sophistication of modern electric and gas appliances and the inability of present energy distribution and control systems to accommodate the potential these appliances offer.

The Smart House will provide intelligent control and coordination between appliances and their controlling devices. Appliance functions that are presently accomplished with difficulty - distributed control of multizone heating, ventilating, and air conditioning and multiunit remote control of entertainment components - are simple to support with the Smart House design.

This same intelligent control and coordination also makes possible a new standard of safety in the home, primarily by providing closed-loop control of electrical appliances. With closed-loop control, branch circuits are de-energized except when power is necessary to operate appliances. This feature is made possible through redesigning the function of an appliance switch, which in the Smart House is a signaling device rather than a power controller. When an appliance is turned on, a signal is sent from the appliance switch to a circuit controller, which is part of the Smart House wiring system. This device then powers the circuit that supplies energy to the appliance. Once the flow of power has been initiated, the appliance must send a recurring "nominal-operation" signal to the circuit controller to continue the power feed. If the circuit controller does not receive this signal, it assumes that a fault has occurred (e.g., the plug was pulled out of the outlet) and de-energizes the circuit. Closed-loop protection of circuits in the

Smart House can be thought of as the electrical equivalent of the "proof-of-flame" protection in gas appliances.

The Smart House design must meet rigid requirements for reliability and ease of installation and maintenance. As an example of the attention being paid to reliability, the Smart House design provides home control through use of several distributed controllers all wired together, each with the ability to back up one another in case of failure. This redundancy avoids the sudden loss of all home control functions that could occur if only one central controller were used.

3. THE FORTH LANGUAGE

FORTH is a high-level language that is widely used in real-time applications. Its most distinctive feature is the ease with which a FORTH subroutine (called a word in FORTH) can be changed. In fact, a FORTH programmer can completely redefine the FORTH system words if the need arises.

Charles Moore developed FORTH in the 1970s for use in astronomical applications. During the early 1980s, FORTH became popular with amateur programmers because it allowed the use of a high-level language on microcomputers with limited memory and processing ability. Since then, FORTH has had a checkered history. Its recent decline in popularity for general-purpose programming is the result of the ever-increasing availability of high performance microcomputers (such as the ubiquitous IBM clones) and the tendency of some FORTH programmers to develop unstructured code that is difficult to follow and impossible to change. Guidelines for averting this latter problem are discussed in the excellent tutorial, *Starting FORTH* (Brodie 1981).

Both experienced and inexperienced application programmers often have trouble using FORTH effectively, but for different reasons. Almost all inexperienced application programmers are uncomfortable with FORTH's use of "reverse-Polish" notation and extensive use of stack operators. To appreciate this concern, contrast two equivalent BASIC and FORTH programs for printing a table of squares from 1 to 10:

BASIC	FORTH
`FOR I = 1 to 10`	`10 1 DO`
`PRINT I,I*I`	`I . I I * .`
`NEXT I`	`LOOP`

In the FORTH example, the first line puts the final limit on the stack, puts the initial value of the loop index on the stack, then calls the FORTH word *DO* to start the loop. The six FORTH words in the second line, which form the body part of the loop, produce the same result as does the corresponding line of the BASIC program. By writing out

detailed remarks for each of the FORTH words on the second line, the operations performed in the body part of the loop become clear:

FORTH WORD	Comments
I	Put a copy of the loop index on the stack
.	Remove the last value on the stack and print it out (which is accomplished by the cryptic FORTH word appropriately pronounced "dot")
I	Put a copy of the loop index on the stack
I	Put a copy of the loop index on the stack (on top of the copy already there)
*	Multiply the two values of the index together to produce the square, which is left on the stack
.	Remove the square from the stack and print it out.

Someone familiar with Hewlett/Packard hand calculators, which are based on reverse-Polish operation, will immediately recognize FORTH notation. Unfortunately, Hewlett/Packard calculators are becoming less and less common, so that the majority of application programmers are uncomfortable with FORTH when first introduced to it.

Experts have another, bona-fide concern about the use of FORTH in time-critical applications. With FORTH words, and especially the elementary words such as *DUP*, *DROP*, +, *AND*, etc., the time necessary to execute the operation is usually dwarfed by the time needed to fetch the word from the stack, push the result back onto the stack, and perform the operations necessary to transfer control to the succeeding word. Typically, elementary FORTH operations are only about a tenth as efficient as the same application written in native code (Appendix A shows an example of the overhead associated with the FORTH word that adds two 16-bit integers).

4. THE SIEVE OF ERATOSTHENES

In 1981, Jim and Gary Gilbreath proposed a simple method of testing microcomputer system performance through use of a small program based on the sieve of Eratosthenes,[2] an algorithm developed in the third century B.C. The process is based on crossing out multiples of two, then repeatedly advancing to the next highest integer not marked out and marking out multiples of it. Thus, the first set of markings would cross out 2, 4, 6, etc. Three would then be selected (the next integer not marked after 2) and 6, 9, 12, etc, would be marked. The next integer after 3 not marked is 5, which is used to mark out 10, 15, etc. The numbers not crossed out are prime numbers.

The Gilbreaths' implementation of the algorithm is a structured program that is easily translated into Pascal or C, and has neither multiplication nor division.[3] As in the Knuth algorithm, the Gilbreaths' program omits processing multiples of two; only odd integers are considered for primality. An array of flags is maintained, one for each odd integer. The flag with index 0 is for the integer 3, index 1 is for 5, index 2 is for 7, etc. In general, the index for the integer *I* is *(I-3)/2*.

The standard Gilbreath benchmark program, which appears at the top of the next page, determines the primes between 3 and 16,384 (the 14th power of 2). When the benchmarks were written (1980-1981), 8K bytes (the space necessary to flag primality of the odd integers between 3 and 16,384) was a significant amount of memory:

---

[2]Knuth's statement of the sieve of Eratosthenes algorithm is "Start with all the odd numbers less than N; then successively strike out the multiples p_k^2, $p_k(p_k + 2)$, $p_k(p_k + 4)$, ... of the kth prime p_k, for $k=2,3,4,....$, until reaching a prime p_k with p_k^2 greater than N."

[3]Gilbreath's statement of the sieve of Eratosthenes algorithm, in Knuth terms, is: "Start with all the odd numbers less than N; then successively strike out the multiples $3p_k$, $5p_k$, ... of the kth prime p_k, for $k=2,3,4,....$, until reaching a prime p_k greater than N."

```
 1.  constant size = 8190;
 2.  var
 3.       flags : array [0..size] of integer;
 4.       i,prime,k,count,iter : integer;
 5.  begin
 6.       count := 0;
 7.       for i := 0 to size do
 8.            flags[i] := 1;
 9.       for i:= 0 to size do
10.            if flags[i] then
11.                 begin
12.                 prime := i + i + 3;
13.                 k :=i + prime;
14.                 while k <= size do
15.                      begin
16.                      flags[k] := 0;
17.                      k :=k + prime
18.                      end;
19.                 count := count + 1
20.                 end;
21.  end;
```

Lines 1 through 4 define the constants, variables, and the array for keeping track of whether an odd integer is a prime or not. A 1 in this array indicates a prime; a 0 indicates a nonprime. With the array dimension 8,190, the last odd integer examined for primality is 16,383.

The body part of the program examines each of the flags sequentially. When a flag is found not marked to a 0, all the remaining flags corresponding to multiples of the prime are set to 0. This is done by computing the prime (line 12), computing the first index of the first multiple (line 13), then marking members of the flag array for the computed index and all succeeding multiples of the prime for which the index is less than 8,190 (lines 14-18). Finally, the count of primes is updated, and the process repeats for the next member of the flag array not marked to 0.

In 1983, the Gilbreaths' translated their Pascal/C program into a number of languages (BASIC, FORTRAN, Ada, COBOL, etc.) and published an extensive list of timings for

computers ranging in size from IBM mainframes (3033) to home microcomputers (Apple II). The following table shows a selection from the Gilbreaths' compilation:[4]

Table 1. Benchmark Results from BYTE Magazine

Computer	Language	Time for 10 Iterations (Seconds)
IBM 3033	Assembly	.0078
VAX 11/780	C Compiler	1.4
IBM AT (6 MHz)	C Compiler	.7
IBM PC (5 MHz)	Assembly	4.
IBM PC (5 MHz)	C Compiler	30.
IBM PC (5 MHz)	FORTH	70.
Apple II	Assembly	13.9
Apple II	FORTH	190.

When evaluating this list of timings, remember that they reflect timings for a translation of the original Pascal algorithm to the equipment/language combination being benchmarked.

[4]By convention, sieve algorithm timings are reported for performing the sieve calculation ten times because the time for a single pass on the faster computers is only a fraction of a second.

5. THE GILBREATHS' FORTH PROGRAM FOR THE SIEVE ALGORITHM

The FORTH version of the Gilbreaths' benchmark, as presented in their *BYTE* articles, is shown below:

```
 1.  8190 CONSTANT SIZE
 2.  0 VARIABLE FLAGS SIZE ALLOT
 3.  : DO-PRIME
 4.    FLAGS SIZE 1 FILL
 5.    0 SIZE 0
 6.    DO FLAGS I + C@
 7.       IF I DUP + 3 + DUP I +
 8.            BEGIN DUP SIZE <
 9.            WHILE 0 OVER FLAGS + C! OVER + REPEAT
10.            DROP DROP 1+
11.       THEN
12.    LOOP
13.    . ." PRIMES" ;
```

The Gilbreaths admit they are not skilled FORTH programmers; unfortunately, their FORTH program reflects their lack of skill. There are two errors in the programs that must be corrected to obtain correct results.[5] First, the condition for testing the completion of the *BEGIN...WHILE...REPEAT* (lines 8 and 9) loop is erroneously written as *BEGIN DUP SIZE <*. To be consistent with the original version of the algorithm, the line should read *BEGIN DUP SIZE <=*.

The second error occurs on line 4 of the program. The FORTH word *FILL* initializes the number of bytes specified in its second argument. Thus, the result of executing the words on line 2 is to initialize only bytes *0, 1, ..., 8189*. To also initialize *FLAG[8190]*, the statement on line 2 should read:

```
FLAGS SIZE 1+ 1 FILL
```

A third error, which occurs on line 2, does not affect the correctness of the results. As written, 8,192 bytes are allocated for the *FLAGS* array. The sequence *0 VARIABLE*

---

[5]The errors result in incorrectly determining that 16,383 (which has factors 3, 43 and 127) is a prime.

FLAGS allocates 2 bytes for a variable called *FLAGS*. The sequence *SIZE ALLOT* allocates *SIZE* more bytes for the variable, thus defining *SIZE* + 2 bytes (8,192) bytes total. A correct definition of the *FLAGS* array is:

```
0 VARIABLE FLAGS SIZE 1 - ALLOT
```

A better way of defining the *FLAGS* array is to use the standard FORTH word for array definition, *ARRAY-VARIABLE*, to define an array of length *SIZE+1*:

```
SIZE 1+ ARRAY-VARIABLE FLAGS
```

A corrected version of the Gilbreaths' version of the sieve algorithm is:

```
 1.  8190 CONSTANT SIZE
 2.  SIZE 1+ ARRAY-VARIABLE
 3.  : DO-PRIME
 4.    FLAGS SIZE 1+ 1 FILL
 5.    0 SIZE 0
 6.    DO FLAGS I + C@
 7.       IF I DUP + 3 + DUP I +
 8.            BEGIN DUP SIZE <=
 9.            WHILE 0 OVER FLAGS + C! OVER + REPEAT
10.            DROP DROP 1+
11.       THEN
12.    LOOP
13.    . ." PRIMES" ;
```

By adding comments and grouping the words as arguments to the higher-level FORTH words, the Gilbreaths' FORTH benchmark program is more easily understood:

```
1.    8190 CONSTANT SIZE                  Define constant SIZE with value 8190
2.    SIZE 1+ ARRAY-VARIABLE FLAGS        Define the array FLAGS of length 8191
3.    : DO-PRIME                          Define a FORTH word called DO_PRIME
4.      FLAGS SIZE 1+ 1 FILL              Initialize 8191 bytes of FLAGS to 1
5.      0                                 Initialize prime count to 0
6.      SIZE 0 DO                         Loop to examine 8191 FLAG bytes
7.         FLAGS I + C@ IF                Fetch FLAGS[I]; do multiples if not 0
8.              I DUP + 3 + DUP I +       Calculate 2*I+3 (prime) and 3*I+3 (K)
9.              BEGIN                     Begin marking of multiples of prime
10.             DUP SIZE <=                 while the value of K is <= SIZE
11.             WHILE
12.                  0 OVER FLAGS + C!    Store zero in FLAGS[K]
13.                  OVER + REPEAT        Increment K by PRIMES; repeat WHILE
14.             DROP DROP 1+ THEN         Drop PRIME, K; increment prime count
15.     LOOP                              End of outer loop to examine FLAGS[I]
16.     . ." PRIMES" ;                    Print out prime count
```

The execution time for the corrected version of the Gilbreaths' FORTH program on a Z8 microcontroller (12 MHz, no wait-states) is 102 seconds. As expected, the Z8 result puts it in the same league as the 6502, the microprocessor used in the Apple II computer. The following table shows this result together with the benchmarks previously reported:

Table 2. Z8 FORTH Benchmark Results Using the Gilbreath Version of the FORTH Benchmark

Computer	Language	Time for 10 Iterations (Seconds)	
IBM 3033	Assembly	.0078	
VAX 11/780	C Compiler	1.42	
IBM AT (6 MHz)	C Compiler	.7	
IBM PC (5 MHz)	Assembly	4.	
IBM PC (5 MHz)	C Compiler	30.	
IBM PC (5 MHz)	FORTH	70.	
Apple II	Assembly	13.9	
Apple II	FORTH	190.	
Z8	FORTH	102.	(Gilbreath program)

The major contributor to the poor performance of the Gilbreaths' FORTH program is the inefficient code that results when PASCAL is ported literally without regard to using features of the FORTH language that can result in a more efficient program. The next section presents a way of improving the efficiency of the inner loop of the program, which is used to mark multiples of primes.

6. IMPROVING THE EFFICIENCY OF THE INNER LOOP

To take advantage of an alternative method of marking the multiples, notice that the *BEGIN..WHILE..LOOP* consists of marking the last row of an *N* by *SIZE/N* matrix with zeros except for the cell in the first column. In the case of *N=3*, the result of the inner calculation is:

A_1	A_4	A_7	A_{10}	...	$A_{SIZE/3-2}$
A_2	A_5	A_8	A_{11}	...	$A_{SIZE/3-1}$
1	0	0	0	...	0

Once the relation of the multiple-marking process to matrix operations is recognized, a more efficient method of initializing the *FLAGS* array is possible using a FORTH word *(FILL-ROW)* to initialize a row of a *FLAGS* array in matrix format.[6] After using *FILL-ROW*, the program must reset the initial element of the row back to 1. The four arguments for the *FILL-ROW* word are similar to those for *FILL*, except that extra parameters for the row index and column length are inserted after the array-name argument. The stack situation when calling *FILL-ROW* must be programmed to be:

```
Array name, Row index, Column length, Array size, Initialization value
```

Modifying the FORTH program to use the word *FILL-ROW* results in the program shown at the top of the next page:

---

[6]An implementation of the word FILL-ROW for Z8 FORTH is provided in Appendix B.

```
1.  8190 CONSTANT SIZE                  Define constant SIZE with value 8190
2.  SIZE 1+ ARRAY-VARIABLE FLAGS        Define an array of length 8191
3.  : DO-PRIME                          Define a FORTH word called DO_PRIME
4.    FLAGS SIZE 1+ 1 FILL              Initialize 8191 bytes of FLAGS to 1
5.    0                                 Initialize prime count to 0
6.       SIZE 0 DO                      Loop to examine 8191 FLAG bytes
7.       FLAGS I + C@ IF                Fetch FLAGS[I]; do multiples if not 0
8.                 FLAGS                Put address of FLAGS on the stack
9.                 I I DUP + 3 +        Put row index and PRIME (2*I+3) on stack
10.                SIZE 0               Matrix size, initialization constant to stack
11.                FILL-ROW             Invoke word to zero last row of matrix
12.                1 I FLAGS + C!       Restore the value one at FLAGS[PRIME]
13.                1+ THEN              Increment the prime count
14.           LOOP                      End of outer loop to examine FLAGS[I]
15.    . ." PRIMES" ;                   Print out prime count
```

The time to execute ten iterations of this improved version of the sieve algorithm in FORTH is shown in the following table together with the previously presented benchmarks:

Table 3. Z8 FORTH Benchmark Results Using the ROW-FILL Version of the FORTH Benchmark

Computer	Language	Time for 10 Iterations (Seconds)	
IBM 3033	Assembly	.0078	
VAX 11/780	C Compiler	1.42	
IBM AT (6 MHz)	C Compiler	.7	
IBM PC (5 MHz)	Assembly	4.	
IBM PC (5 MHz)	C Compiler	30.	
IBM PC (5 MHz)	FORTH	70.	
Apple II	Assembly	13.9	
Apple II	FORTH	190.	
Z8	FORTH	102.	(Gilbreath program)
Z8	FORTH	36.	(Version using ROW-FILL)

Thus, by using the FORTH words applicable to the sieve algorithm, rather than simply developing FORTH as a direct translation from Pascal, execution speed can be improved by a factor of three, which is comparable to IPB PC/C Compiler results.

7. IMPROVING THE EFFICIENCY OF THE OUTER LOOP

To further improve execution performance of the FORTH benchmark program, the outer loop can be made more efficient by converting it to its assembly language equivalent.[7] The outer loop is that part of the FORTH program that examines each integer's flag in turn and initiates the marking of multiples for each prime encountered. The strategy for optimizing the outer loop of the program shown on the previous page is to redefine the FORTH words that accomplish the looping (lines 6 and 14), and the flag test (line 7). If a prime is encountered as a result of the flag test, the previously developed FORTH code (lines 8 through 13) is to be used to mark multiples. The program to accomplish this is as follows:

```
1.   8190 CONSTANT SIZE                 Define constant SIZE with value 8190
2.   SIZE 1+ ARRAY-VARIABLE FLAGS       Define an array of length 8191 (0 .. 8190)

3.   : SIEVESUB                         Define a FORTH word to mark primes
4.        FLAGS                         Put the address of FLAGS on the stack
5.        I I DUP + 3 +                 Put row index and PRIME (2*I+3) on stack
6.        SIZE 0                        Matrix size, initialization constant to stack
7.        FILL-ROW  ;                   Invoke word to zero last row of matrix
8.        1 I FLAGS + C1                Restore the value of the prime's FLAG
9.   : OUTER-LOOP
10.       do-code,                      Push the limit, starting index to RStack
11.       LTBAI,                        Load the byte at I, test if on-zero
12.       nz if,                        If the byte at I is nonzero; mark its multiples
13.       GOFORTH SIEVESUB              Use a FORTH word to mark multiples
14.       inc-prime-count,              Increment prime count
15.       then,                         End of multiple-marking section
16.       loop-code,                    End of outer loop
17.       EXIT,                         Exit back to DO-PRIME word

18.  : DO-PRIME                         Define a FORTH word called DO_PRIME
19.    FLAGS SIZE 1+ 1 FILL             Initialize 8191 bytes of FLAGS to 1
20.    0                                Initialize prime count to 0
21.       SIZE 0 OUTER-LOOP             Loop to examine 8191 FLAG bytes
22.    . ." PRIMES" ;                   Print out prime count
```

[7]This conversion borders on the "knowledgeable trickery" that Gilbreath disallows.

The FORTH words needed to generate this program are:

do-code,	Push do-limit and starting-index from Stack to RStack
LTBAI,	Load the byte in the FLAG array at index i, test if nonzero
if,	Conditionally execute block of statements depending on the condition code obtained by LTBAI,
GOFORTH	During execution, transfer control to the following FORTH word
loop-code,	Increment the loop index, test for end-of-loop condition

Definitions of these assembler constructs for optimizing the outer loop appear in Appendix C.

The execution rate of the FORTH program using assembly language constructs for the outer loop is shown in the following table along with the timings for the benchmarks previously presented:

Table 4. Z8 FORTH Benchmark Results Using an Optimized Outer Loop Version of the FORTH Benchmark

Computer	Language	Time for 10 Iterations (Seconds)	
IBM 3033	Assembly	.0078	
VAX 11/780	C Compiler	1.42	
IBM AT (6 MHz)	C Compiler	.7	
IBM PC (5 MHz)	Assembly	4.	
IBM PC (5 MHz)	C Compiler	30.	
IBM PC (5 MHz)	FORTH	70.	
Apple II	Assembly	13.9	
Apple II	FORTH	190.	
Z8	FORTH	102.	(Gilbreath program)
Z8	FORTH	36.	(Version using ROW-FILL)
Z8	FORTH	21.	(Assembly language for outer loop)

8. ELIMINATING DUPLICATE MARKING OF MULTIPLES

To squeeze out another increment of improvement, notice that the Gilbreaths' version of the sieve algorithm marks many nonprimes more than once. To see this, consider the determination of the odd nonprimes less than 50:

Prime	Multiples marked
3	9 15 21 27 33 39 45
5	15 25 35 45
7	21 35 49
11	33
13	39

Notice the multiple markings for 33, 35, 39, and 45. In this example, 33, 35, 39, and 45 are each marked twice because these multiples can be obtained from either of two products, e.g., 39 can be obtained from 13 threes or 3 thirteens. As can be seen from Knuth's statement of the algorithm (footnote 2), multiple markings may be omitted for any prime greater than the square root of the limit. Use of this observation for the example above will eliminate multiple markings for primes greater than seven.

To implement the rule for the FORTH program, a test must be inserted in the *OUTER-LOOP* word to eliminate multiple markings whenever the corresponding prime is greater than the square root of the limit. For *SIZE = 8190*, the last integer tested for primality is 16,383. Because the relationship between the integer examined for primality *(P)* and the index in the outer loop *(I)* is $P = (I - 3)/2$ multiple marking for value of the outer loop index greater than $(SQRT[LIMIT] - 3)/2$ can be omitted. In the case of *LIMIT = 16,384*, multiple markings can be omitted for values of the outer loop index greater than 62.

To generalize the program, a square root calculation should be added to permit considering SIZE as a program parameter. A FORTH program incorporating calculation of

the limit to eliminate many multiples is shown below. The additional FORTH words used in this benchmark (*limit>i:if* and *word-then*) are defined in Appendix C:

```
1.  8190 CONSTANT SIZE                  Define constant SIZE with value 8190
2.  SIZE 1+ ARRAY-VARIABLE FLAGS        Define an array of length 8191 (0 .. 8190)

3.  : SIEVESUB                          Define a FORTH word to mark primes
4.      FLAGS                           Put the address of FLAGS on the stack
5.      I I DUP + 3 +                   Put row index and PRIME (2*I+3) on stack
6.      SIZE 0                          Matrix size, initialization constant to stack
7.      FILL-ROW  ;                     Invoke word to zero last row of matrix
8.      1 I FLAGS + C! ;                    Restore the value of the prime's FLAG

9.  : OUTER-LOOP
10.     do-code,                        Push the limit, starting index to RStack
11.     LTBAI,                          Load the byte at I, test if on-zero
12.     nz if,                          If the byte at I is nonzero; mark its multiples
13.     limit>i:if,                     Test if LIMIT > outer-loop index
14.     GOFORTH SIEVESUB                Use a FORTH word to mark multiples
15.     word-then,                      Pairs with limit>i:if conditional test
16.     inc-prime-count,                Increment prime count
17.     then,                           End of multiple-marking section
18.     loop-code,                      End of outer loop
19.     EXIT, ;                         Exit back to DO-PRIME word

20. : DO-PRIME                          Define a FORTH word called DO_PRIME
21.   FLAGS SIZE 1+ 1 FILL              Initialize 8191 bytes of FLAGS to 1
22.   0                                 Initialize prime count to 0
23.     SIZE 0 OUTER-LOOP               Loop to examine 8191 FLAG bytes
24.   . ." PRIMES" ;                    Print out prime count1.
```

The time to perform ten iterations of the prime calculation with the program just presented is only five seconds. This result is shown with the benchmarks previously presented in the table below:

Table 5. Z8 FORTH Benchmark Results - Duplicate Marking Avoidance Version of the FORTH Benchmark

Computer	Language	Time for 10 Iterations (Seconds)	
IBM 3033	Assembly	.0078	
VAX 11/780	C Compiler	1.42	
IBM AT (6 MHz)	C Compiler	.7	
IBM PC (5 MHz)	Assembly	4.	
IBM PC (5 MHz)	C Compiler	30.	
IBM PC (5 MHz)	FORTH	70.	
Apple II	Assembly	13.9	
Apple II	FORTH	190.	
Z8	FORTH	102.	(Gilbreath program)
Z8	FORTH	36.	(Version using ROW-FILL)
Z8	FORTH	21.	(Assembly language for outer loop)
Z8	FORTH	5.	(Avoid duplicate marking of multiples)

9. CONCLUSION

This memorandum presents four FORTH programs that implement the sieve of Eratosthenes algorithm. The baseline version of the benchmark, which was published in *BYTE* articles on computer benchmarks, results in execution that is three times longer than those obtained by writing the program so that it uses a FORTH matrix operation word *ROW-FILL*. The alternative shows that the Z8 microcontroller programmed in FORTH can perform at speeds comparable to an IBM PC programmed using a C compiler.

By replacing the benchmark's outer loop with an assembly-language equivalent that more accurately implements the Knuth algorithm (in this case, eliminating some of the duplicate markings of nonprimes), speeds can be reduced by another factor of five. However, in fairness to the other sieve benchmarks, their timings would also show a significant reduction if a similar duplicate marking of primes was avoided in those programs.

10. REFERENCES

Brodie, L., *Starting FORTH*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Edwards R., *Evaluation of a Single Board Microcomputer Suitable for Rapid Prototyping*, ORNL TM/10361, Oak Ridge National Laboratory, February 1987.

Edwards, R., *Optimizing the Zilog Z8 FORTH Microcomputer for Rapid Prototyping*, ORNL TM/10463, Oak Ridge National Laboratory, September 1987.

Gilbreath, Jim, "A High-Level Language Benchmark," *BYTE*, September 1981.

Gilbreath, Jim and Gilbreath, Gary, "Eratosthenes Revisited," *BYTE*, January 1983.

Knuth, Donald E., *The Art of Computer Programming Vol 2: Semi-Numerical Algorithms.* Reading MA: Addison-Wesley, 1969.

Z8 Microcomputer Technical Reference Manual, Zilog Inc., 1984.

APPENDIX A

EXAMPLE OF SYSTEM OVERHEAD FOR AN ELEMENTARY FORTH WORD

Consider the word "+" that adds the two sixteen-bit words at the top of the stack and replaces them with their sum. In Z8 FORTH, the sequence of instructions to accomplish this operation consist of:

```
    Z8
Instruction                Comments
-----------                --------

POP   WR9          Pop low byte at top of stack into Reg 9
POP   WR8          Pop high byte at top of stack into Reg 8
POP   WR11         Pop low byte at 2nd from top into Reg 11
POP   WR10         POP high byte at 2nd from top into Reg 10
ADD   R9,R11       Add least significant bytes of the two 16-bit arguments
ADC   R8,R10       Add most significant bytes, propagate the carry (if any)
PUSH  WR9          Push result low byte to stack
PUSH  WR8          Push result high byte to stack
LD    R15,#EVR     Load address of execution vector register
LDCI  @R15,@RR2    Load high byte of next execution vector
LDCI  @r15,@RR2    Load low byte of next execution vector
LDCI  @R15,@RR4    Load high byte of next execution address
LDCI  @R15,@RR4    Load low byte of next execution address
JP    @WR6         Jump to code for next FORTH word
```

Calculating the execution times for these operations shows that the *ADD and ADC* instructions amount to just 15% of the total time required to execute the FORTH word.

APPENDIX B

IMPLEMENTATION OF ROW-FILL FOR Z8 FORTH

```
      Z8 Instruction      Comments
      ------------------  ------------------------------------------------

      POP  WR6            POP INITIALIZATION VALUE TO WR7
      POP  WR7
      CALL POPCA8         POP VALUE OF MATRIX SIZE TO WR12/13
                          POP VALUE OF COLUMN LENGTH TO WR10/11
                          POP VALUE OF ROW INDEX TO WR8/9
      POP  WR14           POP MATRIX ADDRESS TO WR14/15
      POP  WR15
      ADD  WR13,WR15      ADD MATRIX ADDRESS TO MATRIX SIZE
      ADC  WR12,WR14
      ADD  R9,R15         ADD MATRIX ADDRESS TO ROW INDEX
      ADC  R8,R14
LOOP:LDC  @R8,R7          STORE INITIALIZATION VALUE VIA R14/15
      ADD  R9,R11         INCREMENT POINTER USING COLUMN LENGTH
      ADC  R8,R10
      CP   R12,R8         TEST HIGH BYTE OF NEW VALUE OF POINTER AGAINST LIMIT
      JR   ULT,EXIT
      JR   UGT,LOOP
      CP   R13,R9         TEST LOW BYTE IF HIGH BYTES ARE EQUAL
      JR   UGT,LOOP
EXIT:JP@  EXIT-VECTOR
```

APPENDIX C

This section defines the FORTH words required in the optimization of the outer loop of the FORTH benchmark for the sieve algorithm discussed in Section 6. The words used in the outer loop optimization make use of locations 60 through 69 of the Z8 register file.

Register File Address	Description of Definition
60/61	Prime count
62/63	Index
64/65	Array size (SIZE)
66/67	Address of the FLAGS array
68/69	LIMIT (Used to limit duplicate markings of nonprimes)

```
: do-code    62 R POP, 63 R POP,       Initialize starting value of index
             64 SIZE-ADDR LOAD-WORD,   Copy of array size to register file
             66 FLAGS LOAD-WORD,       Copy of FLAGS addr to register file
             68 R INCW, do, ;          Increment SIZE to do SIZE+1 iterations

: LTBAI      R9 63 RR LD,              Copy of INDEX to working register 8/9
             R8 62 RR LD,
             R9 67 RR ADD,             Add FLAGS address to Index
             R8 66 RR ADC,
             R10 R8 LDC,               Load R10 via pointer at R8/9
             R10 R10 rr OR, ;          Set condition codes in Z8 flag register

: limit>i:if,     68 62 RR CP,         Test LIMIT > INDEX
                       le if,         If LIMIT < INDEX skip out
                       eq if,         IF LIMIT(H)=INDEX(H) test low byte
                  69 63 RR CP,        Test low bytes
                       ult if,        if LIMIT(L) <= INDEX(L) skip out
                  SWAP then,
             62 R PUSH, 63 R PUSH, ;  Push current value of INDEX to stack

: word-then,   then, then,            Targets for the i<limit tests

: loop-code, 62 R INCW,               Increment INDEX
             64 R DECW,               Decrement the loop counter
             loop,                    Test loop counter for end-of-loop
```

```
: LOAD-WORD OVER 1+ OVER 0 100 U/ RR LD, DROP FF AND RR LK, ;

: ZERO-WORD DUP 1+ R CLR, R CLR , ;

: GOFORTH    FIND                    Address of target FORTH word to stack
             R0 R DECW,              Save R2/R3 on RStack
             R0 R3 STC,              R3 to RStack (via R0)
             R0 R DECW,
             R0 R2 STC,              R2 to RStack (via R0)
       HERE 12 + and 3C00 OR,        Low byte of HERE+12 to R3
       HERE 10 + 0 100 U/
         SWAP 2C00 OR , DROP         High byte of HERE+10 to R2
       DUP FF and 5C00 OR ,          Low byte of target addr to R5
       0 100 U/ 4C00 OR , DROP       High byte of target addr to R4
            R6 R4 LDC,               Load jump addr to R6/7
            R4 R INCW,
            R7 R4 LDC,
            R4 R INCW,
            R6 EXIT,                 Jump to target addr via R6/7
       HERE @ + , HERE @ + ,         IP and EV to return to prior routine
            R2 R0 LDC,                    Restore IP
            R0 R INCW,
            R3 R0 LDC,
            R0 R INCW,

: / 0 0 SWAP U/ SWAP DROP ;

: SQRT DUP 8000 AND IF NEGATE THEN 60 5 0 DO OVER OVER / + 2 / LOOP SWAP DROP ;
```

INTERNAL DISTRIBUTION LIST

1. F. Paul Baxter
2-11. R. Edwards
12. W. Fulkerson
13. R. Gryder
14. R. B. Honea
15. J. O. Kolb
16. Russell Lee
17. G. T. Privon
18. D. E. Reichle
19. P. M. Spears
20. R. B. Shelton
21. D. P. Vogt
22-24. Steve Wallace
25. Central Research Library
26. Document Reference Section
27-29. Laboratory Records
30. Laboratory Records -RC
31. Patent Office
32. C. B. Grillot
33. Eleanor T. Rogers

EXTERNAL DISTRIBUTION LIST

34. Office of Assistant Manager for Energy Research & Development, Department of Energy, Oak Ridge Operations Office, Oak Ridge, TN 37831.

35-44. Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831.

45. J. J. Cuttica, Vice President of Research and Development, Gas Research Institute, 8600 W. Bryn Mawr Avenue, Chicago, IL 60631.

46. J. P. Kolt, Professor of Economics, Kennedy School of Government, Harvard University, 79 John F. Kennedy Street, Cambridge, MA 02138.

47. D. E. Morrison, Professor of Sociology, Michigan State University, 201 Berkey Hall, East Lansing, MI 48824-1111.

48. R. L. Perrine, Professor, Engineering and Applied Sciences, Civil Engineering Department, Engineering I, Room 2066, University of California, Los Angeles, CA 90024.