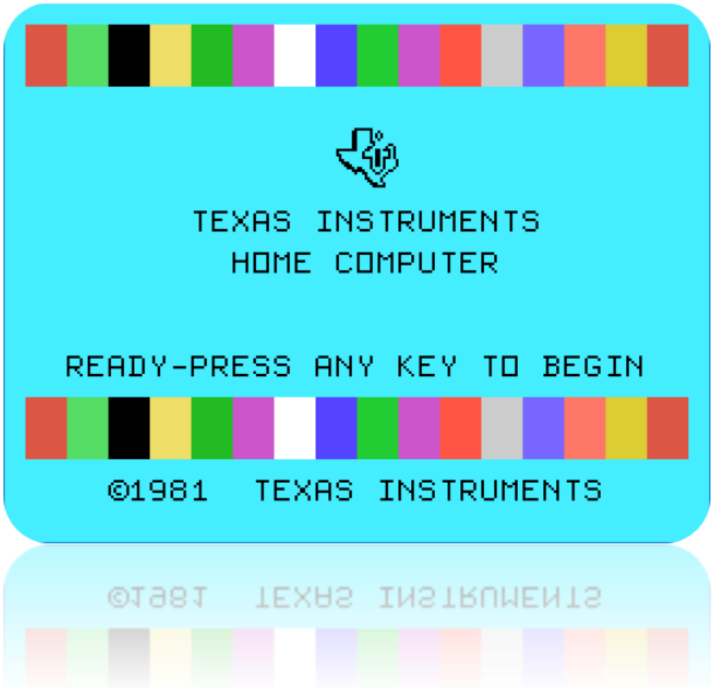

ASMForth II

For the TI-99 and Camel99 Forth

08.Mar.2023

Version 0.73

Brian Fox
Kilworth Ontario
brian.fox@brianfox.ca

Overview

ASMFORTH II is an experimental Machine Forth for the TI-99. It uses Forth-like syntax but is really an Assembler. Many of the words are aliases of the Forth assembler in Camel99 Forth. Although the syntax looks like Forth the significant difference is that registers are referenced explicitly for maximum performance. The Forth data stack and return stack are still available to the programmer but also must be referenced explicitly in your ASMForth code.

Justification

The author has spent a considerable amount of time adapting Chuck Moore's Machine Forth concept to the TMS9900 and the results are good but not great. The hypothesis is that the lack-lustre performance of Machine Forth on the 9900 is due to the hardware mis-match between the 9900 and the F21 Forth CPU. Machine Forth is actually the Assembly language for Chuck's F21. The conclusion is that like the original machine Forth, any machine Forth must leverage underlying hardware to be efficient.

Motivation

ASMForth was originally a curiosity project to see if a 9900 Assembler could be written that used Forth type syntax. It worked and was put on the shelf.
ASMForth II was created when comparing the performance of the Byte Magazine Sieve of Eratosthenes benchmark using conventional Forth, a GCC compiled 'C' version and a TMS 9900 Assembly language version. Here are the timings:

- Camel99 Forth (Indirect threaded) 120 seconds
- C compiled with GCC -02 optimizing 13 seconds
- Hand coded Assembly language 10 seconds

A version of the sieve using ASMForth II for the inner loop and Forth for the outer loop did the benchmark in 10 seconds.

Not the Whole Story

ASMForth II is built on top of Camel99 Forth. This document only explains the ASMForth vocabulary. For a fuller understanding of the underlying Forth system you will need the Camel99 Forth Manual and Glossary found at:

[CAMEL99-ITC/DOCS at master · bfox9900/CAMEL99-ITC · GitHub]

ASMForth Fundamentals

AsmForth II gives the programmer access to the full instruction set of the TMS9900. In many cases there is a one-to-one relationship between a Forth word and an instruction. In many other cases the instruction is completely unique to the 9900 CPU and is provided as a TI mnemonic that operates with reverse-polish-notation syntax. (RPN) TI mnemonics and ASMForth words can be interleaved in a program freely.

Registers are Required

In Forth the data stack is used to provide a simple way to move parameters to and from different blocks of code. The 9900 CPU has no native stack and so the 'PUSH" operation requires two instructions. The POP operation is one instruction. Both however require slower indirect addressing modes and so are expensive on the 9900.
Registers are where performance is gained on this CPU so ASMForth programming uses them. This makes it a bit like using ANS Forth local variables except that the variables are CPU registers. In many cases the simplicity of using registers is easier to understand than "stackrobatics'' using SWAP DUP OVER and the rest.
Where the data stack and return stack are useful is for passing parameters from Forth to ASMForth and getting results back to the Forth system. The return stack also lets us create nestable subroutines and loops with ease.

Registers Allocations

The data stack can be used to save/restore registers for nesting calls. The return stack can also be used to save registers BUT they must be restored before exiting a routine.
The following six registers are free for ASMForth programs to use. R4 doubles as the cache for the top of the data stack. Simply PUSH it to free it up. You can also PUSH any other registers that you want to preserve and POP them back later.

R0   free to use
R1   free to use
R2   free to use
R3   free to use
R4   TOS Accumulator
R5   free to use (used by UM* with operations on the DATA stack)

These registers have special purposes in the Forth virtual machine. R8 is only a temp register in the interpreter so you can use it as well.   Registers marked with '**' are critical if you plan to return to the Forth environment.

SP   data stack pointer **
RP   return stack pointer **

R8   free to use (Forth working register)
R9   Forth interpreter pointer **
R10 holds NEXT address (Forth interpreter) **
R11 subroutine linkage ** use it but be careful
R12 CRU I/O   ** use it but be careful

These registers are completely free if you don't use multitasking with MTASK99 .

R13 Multi-tasking -or- temp
R14 Multi-tasking -or- temp
R15 Multi-tasking -or- temp

ASMForth Register Names

Registers are re-named to mimic a Forth CPU with TOS (top of stack) and NOS (next on stack) being the Registers used for DATA stack operations. Their Forth definition is shown.

Primary Registers Used with DATA Stack

```
: TOS    R4   ;           \ cache for the top of stack item
: NOS   *SP   ;      \ Next on Stack
: NOS^  *SP+ ;      \ Use NOS^ and pop itself
```

Deeper Into the DATA Stack

```
: 3RD   2 (SP) ;
: 4TH   4 (SP) ;
: 5TH   6 (SP) ;
: 6TH   8 (SP) ;
```

You CANNOT use '@' or @+ with these deep DATA stack items.
The deeper registers would not typically be needed because we can use hardware registers like local variables, but they are easily accessed by the 9900 architecture. You might want to grab one and put it into a register without destroying it like this:

```
5TH R0 !
3RD R1 !
```

Forth to 9900 Instruction Mapping

The simple secret of making ASMForth work is converting the MOV and MOVB instructions to the Forth machine equivalents ! (store) and C! (char store).

```
Name    ASMForth    9900
-----   --------    -----
Store      !         MOV
CStore    C!         MOVB
```

C! Has been given some “smarts”. If the destination argument is a register, MOVB will be followed by an 8 bit SRL instruction on that register to get the byte on the other side of the register. This allows you to use Math operations on it.

Then we translate Forth’s “fetch” to be indirect addressing:

```
Fetch    @         do indirect addressing on register argument
```

The 9900 gives us a powerful instruction that is used by Forth programmers to fetch and auto-increment the address.

```
Fetch++  @+        indirect addressing with auto-increment
```

RPUSH and RPOP are Your Friends

If you really need to use Forth critical Registers you can code in the INTEL CPU style by using RPUSH and RPOP pseudo-instructions. These instructions let you save registers on the return stack. This means you can use the data stack as normal. At the end of the code you restore the registers with RPOP in reverse order to how they were RPUSHed. It costs cycles but it’s there if needed.

Example:

```
: MYWORD
 \ Save these specific registers
      R10 RPUSH
      R11 RPUSH
      R12 RPUSH

   ( Your code goes here )

 \ Restore registers in reverse order
      R12 RPOP
      R11 RPOP
      R10 RPOP
```

;

High-Level Additions

Since ASMForth is a compiler, albeit a simple one, it provides the programmer with a few goodies that would need to be hand coded in conventional Assembly Language.

Structured Looping and Branching

Conventional Forth structures are used for loops

BEGIN UNTIL	Jump to BEGIN until condition is true.
BEGIN WHILE REPEAT	Perform code after WHILE while condition is true. Jump to BEGIN
BEGIN AGAIN	Jump unconditionally to BEGIN

FOR NEXT

Execute code between FOR and NEXT for the number of iterations given in the TOS register. Used in Moore's machine Forth it is a simple down-counting loop. Count is maintained on the return stack so FOR NEXT can be nested. If you want to access the loop counter you can "fetch" the return stack pointer register, RP. (Example: RP @ TOS !)

IF ELSE THEN

These are very similar to standard Forth with one important difference. In Forth the top of the data stack is used to determine true or false status. In ASMForth the CPU status register is used to determine how the jumps proceed. The 'IF' word is actually a "compiler" that compiles the correct jump instruction depending on the following comparison tokens.

Comparison Tokens

Signed comparisons: = <> < >

Unsigned Comparisons: U> U< U<= U>=

Each machine instruction will cause the status register in the CPU to record the results of that instruction. For maximum performance ASMForth uses that native hardware to jump. In some cases where you need an explicit comparison the CPU provides three comparison operations. The ASMForth names are below;

- CMP compare two 16 bit arguments
- CMPB compare 8 bit arguments
- #CMP compare a 16 bit argument to a literal number

All these words are limited to the constraints of the jump instructions of the 9900 CPU. +128/-127 cells are the maximum distances they can branch to. This is seldom a problem.

See code examples at the end of the document.

Branching Farther

If you require the branch instruction (B) you can create a label and branch to it.

```
CREATE MYLABEL
       <code here>
       MYLABEL @@ B
```

Constants

Forth variables and constants are used in ASMForth with only a few things to remember. Constants can used for instructions that require a LITERAL number listed below:

- #!
- #AND
- #OR
- #CMP

Load a constant into a register with #! ("number store")

```
X R3 #!
```

Mask a register with constant

```
HEX
007F CONSTANT MASK
TOS MASK #AND
```

Compare a register to a constant

```
 R2 MASK #CMP
 = IF  <true  code>  ELSE  < false  code>  THEN
```

Push a CONSTANT onto the DATA stack with the # macro

```
1234 CONSTANT  X
5678 CONSTANT Y

X #  Y # ( - x y )       \ X and Y are on the data stack
NOS^ TOS +               \ add them together with + and pop NOS
```

Variables

ASMForth uses normal Forth variables.
Variables in Forth return their address so that fits nicely with an Assembly Language.
We can "fetch" the contents of a Forth variable with ASMForth's '@@' operator.
This invokes the memory features of the 9900 CPU called symbolic addressing.

The following are legal statements using VARIABLES in ASMForth:

```
\ declare the variables in the HOST Forth
HOST
VARIABLE Q
VARIABLE T

ASMForth           \ write the code in ASMForth
CODE TESTMEM
      TOS Q @@ !   \ store TOS into memory at Q location
      T @@ TOS !   \ fetch data from memory location T into TOS
      Q @@ OFF     \ reset the memory at location Q to zero
      T @@ ON      \ set the memory at location T to -1 (Forth true)
;CODE
```

Arrays

Arrays are created in the HOST Forth with CREATE ALLOT

```
HOST
CREATE Q      1001 ALLOT
```

We then can choose to reference the array Q with either @, @+ or indexed via a register.

```
ASMFORTH
Q R1 #!            \ store the base address in R1
DUP                \ make room in TOS register
R1 @ TOS !         \ read a cell
R1 @+ TOS !        \ read a cell, increment R1 by 2

R1 @ TOS C!        \ read a byte into TOS
R1 @+ TOS C!       \ read a byte, automatically increment address in R1

\ Indexed addressing
R1 OFF             \ set R1 to index 0
Q (R1) TOS !       \ read a cell from the array into TOS
TOS 2+             \ add 2 to TOS
TOS Q (R1) !       \ put the new value back into the array
DROP               \ restore old value to the TOS register
```

"Colon" Definitions (nestable sub-routines)

Forth uses the colon/semi-colon structure to define new words and ASMForth borrows that syntax to create subroutines. These subroutines can be called from a CODE word by simply invoking their name just like normal Forth. The important difference is that a subroutine can call another subroutine because they use the Forth return stack.
Consider the following ASMForth code:
(This is NOT an example of how to efficiently add 2+2.) :-)

```
ASMFORTH              \ change the namespace to use ASMFORTH keywords
\ sub-routines must be declared before they can be used
: SUB1   NOS^ TOS + ; \ add NOS to TOS and POP NOS (remove it)
: SUB2   SUB1 ;       \ call sub1
: SUB3   SUB2 ;       \ call sub2


\ CODE words give Forth an interface to ASMForth

CODE MYADDITION
     2 #        \ push 2 on the DATA stack
     2 #        \ push another 2 onto DATA stack
     SUB3       \ call SUB3 (compiles: BL @SUB3)
;CODE           \ TOS register holds 4  (2+2)
```

How it works

Behind the curtain ASMForth ':' compiles code to save the CPU linkage register R11 on the return stack. The ';' pops R11 from the Return stack and does a B @R11 instruction.

Tail-Call Optimization with -;

Something that Charles Moore added to machine Forth was tail-call Optimization.
This can be used if a subroutine calls another subroutine **as the last function** in the code. Tail-call optimization replaces the branch and link instruction with a simple branch and enters that subroutine without pushing R11 onto the return stack. This speeds up the call greatly and the return stack does not grow larger. Less return stack usage can be very important if you write a recursive subroutine in ASMForth.

Invoking Tail-Call Optimization

In the previous example we could have code the subroutines this way:

```
: SUB1   NOS^ TOS + ; \ add NOS to TOS and POP NOS (remove it)
: SUB2   SUB1 -;     \ call sub1
: SUB3   SUB2 -;     \ call sub2
```

Notice the use of -; at the end of SUB2 and SUB3. This is the command that invokes tail-call optimization. We cannot tail-call optimize SUB1 because it ends with an instruction not a subroutine call. You have been warned. 🙂

How much difference does Tail-Call Optimization Make?

The difference between using semicolon and minus-semicolon is about 30% speed improvement on the call/return time.

See: demo/nesting.fth for a timing on a one million nest/unnest benchmark

Using the DATA Stack

You are free to use both of Forth's stacks but it is less automatic than standard Forth. You must manage literal numbers with the # word. TOS is actually R4 so must keep that in mind as well. This is done typically by using the NOS^ word pops the data stack from memory.

The advantage of embracing the native machine is that we can use operations on registers directly which is more efficient on the 9900 CPU.

The advantage of having the stacks is nested calls and passing parameters to subroutines does not take up valuable registers.

Addition Example:

```
   45 #   7 #     \ push 2 numbers onto the DATA stack. NOS=45 TOS=7
    NOS^ TOS +    \ Add NOS to TOS. NOS^ pops the stack automatically
                  \ TOS=52
```

This compiles the following TI Assembler code

```
    DECT SP
    MOV  R4,*SP
    LI   R4,45
    DECT SP
    MOV  R4,*SP
    LI   R4,7
    A    *SP+,R4
```

Subroutine Parameter Passing in/out

Compute two numbers squared.

```
ASMFORTH
:  SQUARE ( n – n^2'   )
        DUP               \ push TOS onto memory stack
        NOS^ TOS   *      \ Add NOS to TOS and pop NOS
;

CODE PROGRAM              \ this is our entry to ASMForth from HOST Forth
        5 # SQUARE        \ we can call SQUARE without using any registers
except TOS
;CODE

HOST ( Switch back to Forth dictionary)
```

ASMForth Glossary

Data Stack Words

TOS top of data stack cache register
PUSH (reg –) push any register onto the DATA stack in memory
POP (reg –) pop the NOS value of the DATA into any register

DUP (x -- x x) Push TOS onto data stack in memory
DROP (x --) Refill TOS from the data stack in memory
NIP (x1 x2 -- x2) Remove the NOS value from the DATA stack in memory
SWAP (x1 x2 -- x2 x1) Swap the value of TOS and NOS
OVER (x1 x2 -- x1 x2 x1) Copy NOS to TOS. (other values move down)

Double-Jugglers

2DROP (x1 x2 --)
2DUP (x1 x2 -- x1 x2 x1 x2)

Return Stack Control

RPUSH (reg –) push a register onto the Return stack
RPOP (reg –) pop top of the return stack into a register
DUP>R (x -– x) (R: -- x) Push copy of x onto return stack
>R (x --) (R: -- x) push x onto return stack (remove x from data stack)
R> (-- x) (R: x --) pop x from return stack onto data stack
R@ (-- x) (R: x -- x) Get a copy of x from return stack to TOS

Stack pointers

SP (-- addr) DATA stack pointer register
RP (addr --) Return stack pointer register

Logic

RSHIFT (x1 u -- x2) Arithmetic right-shift of u bit-places
LSHIFT (x1 u -- x2) Arithmetic left-shift of u bit-places
INVERT (x1 -- x2) Invert all bits
XOR (x1 x2 -- x3) Bitwise Exclusive-OR
OR (x1 x2 -- x3) Bitwise OR
AND (x1 x2 -- x3) Bitwise AND

Machine Code Support

CODE name (--)
Create a header in the dictionary for a machine code word. Store current stack position in variable CSP

;CODE (--)
Used with CREATE to make a defining word. Children of the new word will automatically run the machine code that follows ;CODE .

Integer Math (ANS with some 9900 extensions)

U/ (n1 n2 -- n3) n1 / n2 = n3 *Unsigned division
* (u1|n1 u2|n2 -- u3|n3) Single Multiplication

2- (u1|n1 -- u2|n2) Subtracts two, optimized as one instruction
1- (u1|n1 -- u2|n2) Subtracts one, optimized as one instruction
2+ (u1|n1 -- u2|n2) Adds two, optimized as one instruction
1+ (u1|n1 -- u2|n2) Adds one, optimized as one instruction
2* (n1 -- n2) Arithmetic left-shift by 1
4* (n1 -- n2) Arithmetic left-shift by 2
8* (n1 -- n2) Arithmetic left-shift by 3

2/ (n1 -- n2) Arithmetic right-shift by 1
ABS (n -- u) Return Absolute value of n
NEGATE (n1 -- n2) Negate
- (u1|n1 u2|n2 -- u3|n3) Subtraction
+ (u1|n1 u2|n2 -- u3|n3) Addition

Mixed Math Operators

UM* (u1 u2 -- ud) u1 * u2 = ud
UM/MOD (ud u1 -- u2 u3) ud / u1 = u3 remainder u2

Comparisons Tokens

=
<>
> signed
< signed
U> unsigned
U< unsigned
U>= unsigned
U<= unsigned

Number Base

DECIMAL (--) Sets base to 10
HEX (--) Sets base to 16
BASE (-- addr) Base variable address

Memory Access

@ (Reg -- u|n) Fetches the value from the address in a register.
! (u|n addr --) Stores single number from data stack to memory address
C@ (caddr -- char) Fetch byte from memory to data stack
C! (char caddr) Stores byte on data stack to memory address
+! (reg addr --) Add reg to contents of addr (use with variables)

2@ (addr -- ud|d) Fetches double number from memory
2! (ud|d addr --) Stores double number in memory

Video Memory Library (VDPLIBII)

(Not available as of Mar7)

VC@	(Vaddr --)	fetch a byte from Video RAM
VC!	(char Vaddr --)	store a byte to Video RAM
V@	(Vaddr --)	fetch an integer from Video RAM (2 bytes)
V!	(n Vaddr --)	store an integer into Video RAM

VREAD	(Vaddr addr n --)	read n bytes from Video RAM to CPU RAM (VMBR)
VWRITE	(addr Vaddr n --)	write n bytes from CPU RAM to Video RAM (VMBW)
VFILL	(Vaddr n char--)	fill Video RAM at Vaddr with n bytes of char
VWTR	(c n --) "VDP Write to Register". Write c to video chip register n	

String Routines

/STRING (caddr len n -- caddr' len')

Cut n characters off the front of stack string caddr,len.
Return the new stack string.

COUNT (caddr -- caddr len) Convert counted string to stack string

Dictionary Management

ALIGN (--) Aligns dictionary pointer
ALIGNED (caddr -- addr) Advances to next aligned address
CELL+ (x -- x+4) Add size of one cell
CELLS (n -- 4*n) Calculate size of n cells
ALLOT (n --) Move Dictionary Pointer by n bytes (can be negative)
HERE (-- addr|caddr) Return current Dictionary position
, (u|n --) Append a single number to dictionary
C, (c --) Append a byte or character to dictionary
CREATE name (--) Compile time: Create a definition with default action DOVAR
Runtime: Returns the data address of the created word.

Subroutine Creation

: name (--) Start a new subroutine definition
; (--) Finishes subroutine definition
-; (- -) Finishes subroutine with tail-call optimization

Utility constants

FALSE (-- #0) ANS/ISO Forth requirement
TRUE (-- #-1) ANS/ISO Forth requirement
BL (-- #32) Ascii value of a space character

Utility Words

COLD Restart ASMForth

DUMP (addr size –) Dump the contents of memory at addr for size bytes

VDUMP (Vaddr size –) Dump the contents of VDP memory at Vaddr for size bytes

.S (–) Show the contents of the data stack without destroying it.

Control structures

Conventional Forth words IF, THEN, BEGIN, UNTIL etc. are used in ASMFORTH with an important difference. The CPU status register is used by default on conditional branches like IF WHILE or UNTIL. This means that if you decrement a register with 1-, or perform any other operation, the status register will contain bits that signal the result of the operation. Words like IF WHILE and UNTIL respond to the status bits with the following comparison tokens preceding them.

- = equal
- <> not equal
- < signed
- > signed
- U> unsigned
- U>= unsigned
- U< unsigned
- U<= unsigned

Behind the scenes the branching and looping words are actually "compilers" that compile the correct "JMP" instruction into your code based on the token you gave it.

NOTE:
Operators like '=' '<>' are CPU specific here and test the EQ flag in the status register.

Register Comparisons

To explicitly compare two registers we must use the compare instructions.
Two instructions allow us to do a comparison between registers, and/or memory locations. CMP for 16 bits and CMPB for bytes.

We can also compare a register or memory location to a literal value with #CMP.

* It is an ASMForth convention that words that begin with '#' literal argument.
They are typically "immediate" instructions in 9900 Assembler. (see: #+ #OR #AND)

Branching Words

IF (token --) Jump to THEN if flag is true
ELSE (--) { flag IF ... [ELSE ...] THEN } Path when flag is false
THEN (--) End of an IF block. (you think of it as endif)

BEGIN (-- addr) Mark the start of a loop structure. Leaves address on data stack
REPEAT (--) Finish of a BEGIN WHILE loop.
WHILE (token --) Check a flag in the middle of a loop
UNTIL (token --) begin ... flag until loops as long flag is true
AGAIN (--) BEGIN ... AGAIN is an endless loop

FOR (n --) Push n from TOS register onto the return stack
NEXT (- -) Decrement top of return stack. Jump back to FOR until it is -1
Pops the return stack value when loop ends
NEXT2 (- -) Same as NEXT but loop index decrements by 2

Loop Words

BEGIN ... AGAIN Loop forever
BEGIN ... flag UNTIL Loop until flag is true
BEGIN ... flag WHILE ... REPEAT Loop while flag is true

FOR ... NEXT loop for the number of iterations given in TOS register

CODE EXAMPLES

ASMFORTH LOOPS

```
HEX
CODE DOUNTIL  \ smallest loop
      FFFF #   \ DUP R4 and put a number into R4
      BEGIN
        TOS 1- \ decrement a register
      = UNTIL  \ loop until TOS hits zero
      DROP     \ clean the stack
;CODE

HEX
CODE DOWHILE
      FFFF #
      BEGIN
        TOS 1-
      <> WHILE
        R5 2+
      REPEAT
      DROP
;CODE

\ for/next gives you a nestable loop structure
\ put the number of iterations into the TOS register (or use #)
\ It will run unto loop index <0 (uses the JNC instruction)
\ It does not consume a register while running because
\ the counter is on the return stack.
CODE FORNEXT
     100 # FOR
        100 # FOR
           RP @  R0 !   \ fetch loop index store it in R0
        NEXT
     NEXT
;CODE

CODE FOREVER  \ reset the machine to get out
     BEGIN
        TOS 1+
     AGAIN
;CODE
```

Accessing Memory

```
COLD              \ restart the compiler

VARIABLE X        \ Forth variables are used seamlessly
VARIABLE Y

CODE FETCHVAR      \ get a variable into register
     Y @@ R0 !     ( @@ specifies "symbolic" addressing to the CPU)
;CODE

CODE REGISTER2VAR
     R1 X @@ !
;CODE

CODE MEM2MEM        \ move memory to memory in one instruction
     X @@ Y @@ !
;CODE


\ Arrays in memory

CREATE Q    400 CELLS ALLOT  \ Q returns base address
                             \ 400 CELLS = 800 bytes allocated

CODE FETCH-ARRAY
     6 R1 #!           \ store 6 in R1. R1 is our "index register"
      Q (R1) R4 !       \ fetch contents of Q+R1 into R4
;CODE
```

Byte Magazine Sieve of Eratosthenes

Changes were made to the original code for ANS/ISO Forth. The leading zero was removed in the VARIABLE statement. We removed the text I/O from the computation loop and put it in the main loop. This fits better with ASMForth which does not have a console I/O library and also suits the author's sense of Forth coding style.

```
DECIMAL
8190 CONSTANT SIZE
VARIABLE FLAGS   SIZE ALLOT  0 FLAGS !

: DO-PRIME
   FLAGS SIZE  1 FILL  ( set array )
   0                   ( counter )
   SIZE 0
   DO FLAGS I + C@
     IF I DUP +  3 +  DUP I +
        BEGIN
          DUP SIZE <
        WHILE
           0 OVER FLAGS +  C!
           OVER +
        REPEAT
        DROP DROP
        1+
     THEN
   LOOP
;

: PRIMES ( -- )
   PAGE ."  10 Iterations"
   10 0 DO  DO-PRIME  CR SPACE . ." Primes"  LOOP
   CR ." Done!"
;
```

ASMForth II Sieve Program

This code is a translation of the Assembly language program that follows this code.

```
HOST
DECIMAL
    8190 CONSTANT SIZE
HEX 2000 CONSTANT FLAGS   \ array in Low RAM

ASMFORTH
: FILLW ( addr size char --)
    R0 POP            \ size
    R1 POP            \ base of array
    BEGIN
        TOS R1 @+ !   \ write ones to FLAGS
        R0 2-
    NC UNTIL
    DROP ;

HEX
CODE DO-PRIME ( -- n)  \ CODE words can be called from Forth
  FLAGS # SIZE # 0101 # FILLW
\ inits
   R0 OFF                  \ clear loop index
   R3 OFF                  \ 0 constant
   FLAGS R5 #!             \ array base address
   0 #                     \ counter on top of Forth stack
   SIZE # FOR             \ ASMForth II has Chuck's FOR NEXT loop
     R5 @+ R3 CMPB         \ FLAGS C@+ byte-compared to R3 (ie: 0)
     <> IF                 \ not equal to zero ?
          R0 R1 !          \ I -> R1
          R1 2*  R1 3 #+   \ R1 3+
          R0 R2 !          \ I -> R2 ( R2 is K index)
          R1 R2 +          \ PRIME K +!
          BEGIN
            R2 SIZE #CMP    \ K SIZE compare
          < WHILE
            R3 FLAGS (R2) C! \ reset byte, FLAGS(R2)
            R1 R2 +         \ PRIME K +!
          REPEAT
          TOS 1+            \ increment no. of primes
     THEN
```

(continues on next page)

```
   R0 1+                    \ bump index register
  NEXT
;CODE


\ 10 iteration loop is written in interpreted Forth

: PRIMES ( -- )
   PAGE ."  10 Iterations"
   10 0 DO  DO-PRIME  CR SPACE . ." Primes"  LOOP
   CR ." Done!"
;
```

Assembly Language Version

For comparison, here is an assembler version written by "Reciprocating Bill" on Atariage.com.
It is clearly the work of an experienced programmer. It includes code to write text and numbers to the TI-99 screen.

```
* TMS9900 Assembly Language Version
* SIEVE OF ERATOSTHENES ------------------------------------------
* WSM 4/2022
* TMS9900 assembly adapted from BYTE magazine 9/81 and 1/83 issues
* 10 iterations 6.4 seconds on 16-bit console
* ~10 seconds on stock console

        DEF   START
        REF   VMBW

M       DATA  >0000             for PRINT routine
M2      BSS   6
M3      DATA  10000
        DATA  1000
        DATA  100
        DATA  10
ROW     DATA  >0004

MSG1    TEXT '  Sieve of Eratosthenes '
MSG2    TEXT '      10 iterations '
MSG3    TEXT '      done!  '

SIZE    EQU  8190
FLAGS   BSS  8191
PRWS    BSS  >20

* main *
        EVEN
START   LWPI >8300              workspace in PAD for stock console
        LI   R5,>0101           to initialize array
        LI   R8,SIZE            index
        LI   R12,10             # iterations
        LI   R15,11             constant (for calculating row of display)

        LI   R0,36              display title
        LI   R1,MSG1
        LI   R2,24
        BLWP @VMBW
        AI   R0,32
```

```
       LI   R1,MSG2
       LI   R2,20
       BLWP @VMBW


AGAIN  CLR  R7                  0 constant
       CLR  R10                 counts primes
       LI   R6,FLAGS            base of FLAGS array
       LI   R0,SIZE             index FLAGS array

LOOP   MOV  R5,*R6+             write ones to FLAGS
       DECT R0
       JNE  LOOP

       LI   R6,FLAGS            reset index
FOR    CB   *R6+,R7             FLAGS(I)=0?
       JEQ  SIEVE3              yes, skip strikeout loop

       MOV  R0,R1               PRIME=I
       SLA  R1,1                *2
       AI   R1,3                +3
       MOV  R0,R2               K=I
       A    R1,R2               add PRIME to K

SIEVE1 C    R2,R8               K>SIZE?
       JGT  SIEVE2              yes, goto SIEVE2
       MOVB R7,@FLAGS(R2)       no,reset @FLAGS(R2)
       A    R1,R2               K=K+PRIME
       JMP  SIEVE1
SIEVE2 INC  R10                 increment count of primes


SIEVE3 INC  R0                  next I
       C    R0,R8               I>SIZE?
       JLT  FOR                 no, next

       MOV  R10,@M              print #primes found
       BL   @PRINT
       DEC  R12
       JNE  AGAIN               rinse and repeat

       MOV  @ROW,R15            print "Done!"
       INCT R15
       SLA  R15,5
       MOV  R15,R0
       LI   R1,MSG3
       LI   R2,12
```

```
       BLWP @VMBW


OUT    LIMI 2                   wait for quit
       JMP  OUT
```

(continues on next page)

```
* Print binary value as decimal (from Morley, p.149)
* value to print @M

PRINT  LWPI PRWS
       INC  @ROW                increment row counter
       LI   R0,4
       LI   R1,M3
       LI   R2,M2               R2 points to address containing result
       MOV  @M,R4               mov integer to be converted to R4
J1     CLR  R3                  see Morley for comments
       DIV  *R1+,R3
       SLA  R3,8
       AI   R3,>3000
       MOVB R3,*R2+
       DEC  R0
       JNE  J1
       SLA  R4,8
       AI   R4,>3000
       MOVB R4,*R2+
       LI   R1,>2000
       MOVB R1,*R2
       LI   R0,>3000
       LI   R3,M2
J2     CB   *R3,R0
       JNE  J3
       MOVB R1,*R3+
       JMP  J2
J3     CB   *R3,R1
       JEQ  J4
       JMP  J5
J4     DEC  R3
       MOVB R0,*R3
J5     MOV  @ROW,R15            use row counter
       SLA  R15,5               row *32
       MOV  R15,R0
```

```
INCT R0                   write to this address in VDP
LI   R1,M2
LI   R2,6
BLWP @VMBW
LWPI >8300
RT

END
```