This document provides a design description of the FORTH VM support being included in the M65C02A soft-core processor. The M65C02A soft-core processor is an extended version of the 6502/65C02 microprocessor originally developed by Western Design Center (WDC).

The purpose of this document is to document the design decisions made with respect to custom instructions added to the basic M65C02A instruction set in order to provide better support for a FORTH VM than a standard 65C02-compatible processor. It is an objective of this effort to add the least number of instructions (and dedicated hardware) to the M65C02A instruction set as necessary to provide an efficient FORTH VM.

The instruction set of the M65C02A is already extended beyond that of a standard 6502/65C02 microprocessor. The M65C02A implements all of the standard 6502/65C02 instructions and addressing modes, but it also includes the four bit-oriented instructions added by Rockwell and the WAI and STP instructions included by WDC in its extended 6502/65C02 implementation: the W65C02S.

The M65C02A adds 8 base pointer-relative and 8 post-indexed base pointer-relative indirect instructions matching the W65C816 processor's 16 stack-relative/stack-relative indirect instructions. The M65C02A also includes the W65C816's three 16-bit push instructions and its 16-bit relative branch instruction. But the M65C02A adds an additional 16- bit push instruction, several 16-bit pop/pull instructions, post-indexed base pointer-relative indirect jumps and subroutine calls, and a 16-bit relative subroutine call instruction. The M65C02A also includes an instruction for supporting co-processors, and six prefix instructions that change the operand size, add indirection, and override the destination registers of various instructions.

The IND prefix instruction provides the programmer with the ability to add indirection to many instructions which do not include an indirect addressing mode. The SIZ and the OAX/OAX prefix instructions provide the programmer the means to increase the operand/ALU operation size, and to override the default destination register of an instruction. The OSX prefix instruction allows the X register to be used as an auxiliary hardware stack pointer in memory. The effects of IND and SIZ are combined in the ISZ prefix instruction.

The FORTH VM can generally be thought of as being constructed from a minimum of two stacks: (1) a parameter/data stack (PS), and (2) a return stack (RS). The PS is intended to hold all parameters/data used within a program/function, and the RS generally holds the return addresses of FORTH program words. In addition to holding FORTH program word return addresses, the RS is also used to hold loop addresses, and may be used to hold parameter/data addresses.

These stacks are generally implemented within the memory of whatever microprocessor is implementing the FORTH VM. To support an efficient implementation of the FORTH VM, any specific FORTH implementation should provide hardware assisted stack pointers whenever possible.

The M65C02A sof-core processor is an implementation that provides a faithful reimplementation of the 6502/65C02 processor in a synthesizable manner. Although it provides all of the registers and executes all of the standard instructions, its implementation attempts to provide the best performance possible. As a result, the M65C02A reimplementation is not instruction cycle length compatible with 6502/65C02 or W65C02S processors.

In all other respects it is a 6502/65C02 microprocessor. That means that, in its 6502/65C02 compatibility mode, it appears to the programmer has having an 8-bit arithmetic accumulator (A), an 8-bit processor flags registers (P), two 8-bit index registers (X, Y), and a 8-bit system stack pointer (S). Except for S, the general purpose accumulator (A) and the index registers (X, Y) do not provide any hardware support for the FORTH VM data or return stacks. In its extended mode, the M65C02A, has three modified 16-bit push-down stacks, one for each base register: A, X, Y. Each register stack is composed of three 16-bit registers arranged such that the top-of-stack is the register directly affected by the load/store and ALU instructions. Automatic pushing/poping of each register stack is not performed. Instead, explicit management of each register stack is left to the programmer. Three specific instructions are available for managing the register stacks: DUP (duplicate/push TOS), SWP (swap TOS and NOS), ROT (rotate the register stack). Thus, to load the TOS register and to push down the values currently in the stack, a DUP instruction must precede a load instruction. To simply duplicate the TOS and push down the stack, only a DUP instruction is needed. To store the TOS value to memory and pop the stack, a store instruction followed by a ROT instruction is needed.

Although the extensions to the 6502/65C02 instruction set provided by the M65C02A are uniformly applicable to most instructions and registers, each register stack have some unique capabilities. For example, the A register stack supports byte swapping and wrap-around nibble rotation (4 bits at a time) in the stack's TOS register. Similarly, the TOS of the X register stack can function as a third hardware-assisted stack pointer and as a automatically incremented or decremented memory pointer. Finally, the TOS of the Y register stack may function as an automatically incremented or decremented memory pointer.

Performance degradations in a 6502/65C02 FORTH VM are generally due to the fact that all of the registers are 8-bits in length and 16 bits is the assumed operand and pointer size of the VM. Support is provided in the 6502/65C02 instruction set architecture for multi-precision addition and subtraction, but any operations greater than 8 bits in length will entail several loads and stores. All of the additional steps necessary to implement a 16-bit or 32-bit FORTH VM operations reduces the performance a native 6502/65C02 FORTH VM can deliver.

Brad Rodriguez wrote a series of articles on the development of FORTH VMs. The lead article of the series, "MOVING FORTH Part 1: Design Decisions in the Forth Kernel" (http://www.bradrodriguez.com/papers/moving1.htm.), provides a good discussion of the design trades needed when implementing a FORTH VM on a microprocessor.

Brad Rodriguez identifies the following registers as being the "classic" FORTH VM registers:

```
    W   - Work Register             : general work register
    IP  - Interpreter Pointer       : address of the next FORTH word to execute
    PSP - Parameter Stack Pointer   : points to the top of parameter/data stack
    RSP - Return Stack Pointer      : points to the return address
    UP  - User Pointer              : points to User space of a multi-task FORTH
    X   - eXtra register            : temporary register for next address
```

There are several generally accepted methods for implementing the FORTH VM. The classic FORTH VM is implemented using a technique known as Indirect Threaded Code (ITC). The next most common approach is a technique known as Direct Threaded Code (DTC). A more recent approach is a technique known as Subroutine Threaded Code (STC). A final, less commonly used approach is a technique known as Token Threaded Code (TTC).

A TTC FORTH VM uses tokens that are smaller than the basic address pointer size to refer to FORTH words. The smaller size of the tokens allows a FORTH program to be compressed into a smaller image. The additional indirection required to locate the memory address of the FORTH word referenced by a particular token makes TTC FORTH VM implementations the slowest of the FORTH VM techniques. The pre-indexed indirect addressing modes of the 6502/65C02 can be used to implement a token threaded VM using the M65C02A. Thus, the basic indexed addressing modes of the M65C02A provide the necessary support to implement a TTC FORTH VM.

For an STC FORTH VM, each word is called using a native processor call instruction. Thus, there is no need for an IP register, and there is no inner interpreter. The FORTH program simply chains together the various FORTH words using native processor calls. There are two penalties for this simplicity: (1) subroutine calls are generally larger than simple address pointers; and (2) requires pushing and popping the return stack on entry to and exit from each FORTH word. Thus, STC FORTH programs may be larger, and the additional push/pop operations performed by the native subroutine calling instructions may not deliver the performance improvements expected. Since an STC FORTH VM relies on the basic instructions of the processor, and the M65C02A provides those instructions, no additional instructions are needed in the M65C02A's instruction set to support an STC FORTH.

ITC FORTH and DTC FORTH VMs both require an inner interpreter. Thus, if the instruction set of a processor allows the implementation of the inner interpreter with a minimum number of instructions, then a FORTH VM on that processor would be "faster" than a FORTH VM on a processor without that support. This is the prime motivating factor for adding custom instructions to the M65C02A to support FORTH VMs.

A TTC/DTC/ITC FORTH VM is composed of two interpreters: (1) an outer interpreter, and (2) an inner intepreter. The outer interpreter is written in FORTH, i.e. it is composed of FORTH words. The inner interpreter, on the other hand, "executes" FORTH words. Therefore, a TTC/DTC/ITC FORTH VM spends the majority of its processing time in its inner interpreter. Thus, any decrease in the number of clock cycles required to "execute" a FORTH word will result in a clear increase in the performance of a FORTH program all other things being equal.

The basic structure of a 16-bit FORTH word is provided by the following C-like structure:

```
    struct {
        uint8_t     Len;
        uint8_t     [Max_Name_Len] Name;
        uint16_t    *Link;
        uint16_t    *Code_Fld;
        uint8_t     [Code_Len] Param_Fld;
    } FORTH_word_t;
```

There are other forms, but the preceding structure defines all of the necessary components of the FORTH word, and succintly conveys the required elements of a FORTH word.

The first three fields provide the "dictionary" header of FORTH words in a FORTH program. The first field defines the length of the name of the FORTH word. This is used to distinguish two or more FORTH words in a FORTH program whose names share the same initial letters but which differ in length. The second field defines the significant elements of the name of the FORTH word. The total lengths of these two fields will determine the amount of memory that is required just for the dictionary of a FORTH program. These two fields are generally limited in size in order to conserve memory. If the immediate mode of the FORTH compiler is not supported or included in the distribution of a FORTH program, then the fields supporting the "dictionary" can be removed to recover their memory for use by the application.

The fields, Code_Fld and Param_Fld, represent the "executable" portion of a FORTH word. There are two types of FORTH words: (1) secondaries, and (2) primitives. Secondaries are the predominant type of words in a FORTH program. Their Param_Fld doesn't contain any native machine code. The Param_Fld of FORTH secondaries is simply a list of the Code_Fld of other FORTH words. Primitives are the FORTH words that perform the actual work of any FORTH program. The Code_Fld of a primitive is a link to their Param_Fld, which contains the machine code that performs the work the primitive FORTH word is expected to provide.

In FORTH, the outer interpreter is used to perform immediate operations, and construct, define, or compile other FORTH words. As already stated above, the FORTH VM's outer interpreter is generally composed of secondary FORTH words. After all of the FORTH words associated with a FORTH program have been compiled, the outer interpreter simply transfers control to the inner interpreter to "execute" the top-most FORTH word of the program.

The FORTH VM's inner interpreter "executes" FORTH words. Since the outer interpreter is mostly composed of secondary FORTH words, the inner interpreter must move through each word until a FORTH primitive is found, and then transfer temporary control to the machine code. The machine code of the primitive FORTH word must return control to the inner interpreter once it completes its task.

The inner interpreter "executes" the FORTH word that its IP points to. It must advance the IP through the Param_Fld of a secondary FORTH word, and jump to the machine code pointed to by the Code_Fld of a FORTH primitive. The Code_Fld of a secondary does not point to the Param_Fld of the word. Instead it points to an inner interpreter function that "enters" the Param_Fld. The Code_Fld of a primitive does point to the Param_Fld, and the inner interpreter simply jumps to the machine code stored in the Param_Fld of the word.

Thus, there are three fundamental operations that the inner interpreter of a DTC/ITC FORTH VM must perform:

```
    (1) NEXT    : fetch the FORTH word addressed by IP; advance IP.
    (2) ENTER   : save IP; load IP with the Code_Fld value; perform NEXT.
```

Each primitive FORTH word must transfer control back to the inner interpreter so that the next FORTH word can be "executed". This action may be described as:

```
    (3) EXIT    : restore IP; perform NEXT.
```

These three fundamental operations are very similar to the operations that the target processor performs in executing its machine code. NEXT corresponds directly to the normal instruction fetch/execute cycle of any processor. ENTER corresponds directly to the subroutine call of any processor. Similarly, EXIT corresponds directly to the subroutine return of any processor.

As previously discussed, FORTH uses two stacks: parameter/data stack and return stack. Thus, ENTER and EXIT save and restore the IP from the return stack. Most processors implement a single hardware stack into which return addresses and data are placed. FORTH maintains strict separation between the parameter/data stack and the return stack because it uses a stack-based arithmetic architecture. Mixing return addresses and parameters/data on a single stack would complicate the passing of parameters and their processing.

Table 6.3.1 in Koopman's "Stack Computers", provides a summary the relative frequency of the most frequently used FORTH words for several FORTH applications:

Name	FRAC	LIFE	MATH	COMPILE	AVE
CALL	11.16%	12.73%	12.59%	12.36%	12.21%
EXIT	11.07%	12.72%	12.55%	10.60%	11.74%
VARIABLE	7.63%	10.30%	2.26%	1.65%	5.46%
@	7.49%	2.05%	0.96%	11.09%	5.40%
0BRANCH	3.39%	6.38%	3.23%	6.11%	4.78%
LIT	3.94%	5.22%	4.92%	4.09%	4.54%
+	3.41%	10.45%	0.60%	2.26%	4.18%
SWAP	4.43%	2.99%	7.00%	1.17%	3.90%
R>	2.05%	0.00%	11.28%	2.23%	3.89%
>R	2.05%	0.00%	11.28%	2.16%	3.87%
CONSTANT	3.92%	3.50%	2.78%	4.50%	3.68%
DUP	4.08%	0.45%	1.88%	5.78%	3.05%
ROT	4.05%	0.00%	4.61%	0.48%	2.29%
USER	0.07%	0.00%	0.06%	8.59%	2.18%
C@	0.00%	7.52%	0.01%	0.36%	1.97%
I	0.58%	6.66%	0.01%	0.23%	1.87%
=	0.33%	4.48%	0.01%	1.87%	1.67%
AND	0.17%	3.12%	3.14%	0.04%	1.61%
BRANCH	1.61%	1.57%	0.72%	2.26%	1.54%
EXECUTE	0.14%	0.00%	0.02%	2.45%	0.65%

In table above, CALL corresponds to ENTER. As can be seen, the remaining common FORTH words are a combination of parameter stack operations (DUP, ROT, SWAP), parameter stack loads (VARIABLE, LIT, CONSTANT, @, C@), parameter stack arithmetic and logic operations (+, =, AND), parameter stack branching and looping (0BRANCH, BRANCH, I), and parameter and return stack operations (R>, >R), and special operations (USER, EXECUTE).

With the previous discussion and the FORTH word frequency data in the preceding table it is easy to assert that the M65C02A instruction set should contain custom instructions for at least NEXT, ENTER, and EXIT. The question then becomes to what extent should these operations be supported? In other words, should they be supported by a single instruction each and should they be supported for both ITC and DTC FORTH VMs?

The following pseudo code defines the operations for these three operations in terms of the ITC and the DTC models:

```
               ITC                                    DTC
================================================================================
NEXT:   W      <= (IP++) -- Ld *Code_Fld    ; W      <= (IP++) -- Ld *Code_Fld
        PC     <= (W)    -- Jump Indirect   ; PC     <= W      -- Jump Direct
================================================================================
ENTER: (RSP--) <= IP     -- Push IP on RS   ;(RSP--) <= IP     -- Push IP on RS
        IP     <= W + 2  -- => Param_Fld    ; IP     <= W + 2  -- => Param_Fld
;NEXT
        W      <= (IP++) -- Ld *Code_Fld    ; W      <= (IP++) -- Ld *Code_Fld
        PC     <= (W)    -- Jump Indirect   ; PC     <= W      -- Jump Direct
================================================================================
EXIT:
        IP     <= (++RSP) -- Pop IP frm RS  ; IP     <= (++RSP)-- Pop IP frm RS
;NEXT
        W      <= (IP++) -- Ld *Code_Fld    ; W      <= (IP++) -- Ld *Code_Fld
        PC     <= (W)    -- Jump Indirect   ; PC     <= W      -- Jump Direct
================================================================================
```

Except for the indirection needed for ITC, there are several key takeaways from the side-by-side comparison provided above of the NEXT, ENTER, and EXIT operations. First, ENTER and EXIT are essentially the same for ITC and DTC FORTH VMs. Second, both ENTER and EXIT terminate with the operations implemented by NEXT. Also note that EXIT is simply an RS pop operation followed by a DTC/ITC NEXT operation.

Finally, there are two important observations made by Rodriguez:

(1) if W is left pointing to the Code_Fld of the word being executed, the Param_Fld of a FORTH word being ENTered can be found using the value in W;

(2) providing a second stack pointer for the RS is important and will greatly improve the performance of the inner interpreter.

The preceding analysis and discussions set the stage for the critical design decisions for an M65C02A FORTH VM:

(1) mapping the FORTH VM registers onto the M65C02A registers;

(2) determining how to modify the M65C02A to support the FORTH VM inner interpreter operations.

The IP register is strictly used as the instruction pointer of the inner interpreter. It cannot be assigned to the target processor's program counter, but it does operate as such for the inner interpreter. An easy means for including IP is to place it in zero page memory, but this means that several memory cycles will be needed to increment, push, pop, or otherwise manipulate its value. A better solution is to add a 16-bit register within the core. Alternatively, IP may be accessed with whatever custom instructions are added to the M65C02A instruction set to support NEXT, ENTER, and EXIT. In addition to load and store operations, support should be provided to increment the IP by 2.

Similarly, the W register is used strictly as a pointer for indirect access to a FORTH word by the inner interpreter. It is only loaded indirectly from IP. Like the IP register, it can easily be implemented in zero page memory, but this means that several memory cycles will be needed to increment, push, pop, or otherwise manipulate its value. Like the IP, the best way for the M65C02A to support W is to include it in the processor core itself. Also, for it to be effectively utilized, support should be provided to increment the W register by 2.

The PS is used more often than the RS. Thus, it makes more sense, from a speed perspective, to use the native M65C02A stack for the PS. Using a pre-indexed zero page location for the RSP will slow the push and pop operations significantly. Therefore, a better solution would be to use one of the index registers as the RSP, and place the RS anywhere in memory, including page 0.

As the 6502/65C02 index register for the less frequently used pre-indexed (direct and indirect) addressing modes, the X index register is the natural choice to provide a hardware-assisted RSP. The auxiliary stack pointer capabilities of the TOS of the X register stack easily allow X to be used as the RSP. In addition, two instructions will be added to support pushing and pulling the IP from either stack, and a single cycle instruction will be added to increment the IP by 1. By overloading the IND prefix instruction, it is possible to use the dedicated IP push, pop, and increment instructions to perform the same operations with the W register.

(Note: instead incurring a byte/cycle penalty by using the OSX prefix instruction be used before any FORTH VM instructions that use the RS, the OSX prefix instruction's effects have been redefined to override of the default stack pointer of any instruction that utilizes the stack. The default stack for the FORTH VM ENT, PHI, PLI, PHW, and PLW has been set to use the auxiliary stack provided by X. The change in the behavior of OSX means that only when the PS is needed will these five FORTH VM instruction require OSX. It also saves 1 byte/cycle for every access to the RS.)

Thus, the FORTH VM supported by the M65C02A will provide the following mapping of the various FORTH VM registers:

```
    IP  - Internal dedicated 16-bit register
    W   - Internal dedicated 16-bit register
    PSP - System Stack Pointer (S), allocated in memory (page 1 an option)
    RSP - Auxiliary Stack Pointer (X), allocated in memory (page 0 an option)
    UP  - Memory (page 0 an option)
    X   - Not needed, {OP2, OP1} or MAR can provide temporary storage required
```

The inner interpreter of the FORTH VM will be implemented directly in the M65C02A using five dedicated instructions: NXT (NEXT), ENT (ENTER), PLI (Pull IP), PHI (Push IP), and INI (Increment IP by 1). All five of these instructions can be prefixed by IND. The NXT and ENT instructions directly support a DTC FORTH VM, and when prefixed by IND, they support an ITC FORTH VM. The DTC EXIT will be implemented using the PLI NXT instruction sequence, and the ITC EXIT will be implemented using the PLI IND NXT instruction sequence. Access and control of the IP is provided by the PHI, PLI, and INI instructions. (When these instructions are prefixed by IND, access and control of W is provided: PHW, PLW, and INW, respectively.)

Loading constants/literals is a frequent operation in FORTH programs. Thus, support for efficient loading of in-line constants/literals relative to the IP is included in the M65C02A. The LDA ip,I++ instruction will load the byte which follows the current IP into the accumulator and advances the IP by 1. If this instruction is prefixed by SIZ, then the word following the current IP is loaded into the accumulator and the IP is advanced by 2. If prefixed by IND, the instruction becomes LDA (ip,I++), which uses the 16-bit word following the current IP as a byte pointer. The IP is advanced by 2, and the byte pointed to by the pointer is loaded into the accumulator. If prefixed by ISZ, the word following the current IP is used as a word pointer, while the IP is advanced by 2, to load a word into the accumulator.

The LDA ip,I++ instruction is matched by the STA ip,I++. Without indirection, the STA ip,I++ instruction will write directly into the FORTH VM instruction stream. With indirection, the STA ip,I++ instruction can be used for directly updating byte/word variables whose pointers are stored directly in the FORTH VM instruction stream. (Although, the ability to create may be useful when compiling FORTH programs and for creating self-modifying FORTH programs, the STA ip,I++ instruction is expected to be prefixed with IND or ISZ under normal usage.)

Finally, the ADD ip,I++ instruction allows constants (or relative offsets) located at the current IP to be added to the accumulator. Like LDA ip,I++ and STA ip,I++, the ADD ip,I++ supports the IND, SIZ, and ISZ prefix instructions.

Some consideration was given to directly supporting IP-relative conditional branches for the FORTH VM with the relative branch instructions of the M65C02A. Given that the ADD ip,I++ and LDA ip,I++ instructions can use the same microsequence, it was decided that directly supporting FORTH branches or jumps was too expensive in area and speed. IP-relative conditional FORTH branches can be implemented using the following instruction sequence:

```
            [SIZ] Bxx $1    ; [2[3]] test xx condition and branch if not true
            ISZ DUP         ; [2] exchange A and IP (XAI)
            SIZ ADD ip,I++  ; [5] add IP-relative offset to A
            ISZ DUP         ; [2] exchange A and IP (XAI)
    $1:
```

The IP-relative conditional branch instruction sequence only requires 11[12] clock cycles, and IP-relative jumps require only 9 clock cycles. Conditional branches and unconditional jumps to absolute addresses rather than relative addresses can also be easily implemented. A conditional branch to an absolute address can be implemented as follows:

```
            [SIZ] Bxx $1    ; [2[3]] test xx condition and branch if not true
            SIZ LDA ip,I++  ; [5] load relative offset and autoincrement IP
            IND DUP         ; [2] transfer A to IP (TAI)
    $1:
```

Thus, a conditional branch to an absolute address requires 9[10] cycles, and the unconditional absolute jump only requires 7 clock cycles. Clearly, if the position independence of IP-relative branches and jumps is not required, then the absolute address branches and jumps provide greater performance.

(Note: the M65C02A supports the eight 6502/65C02 branch instructions which perform true/false tests of the four ALU flags. When prefixed by the SIZ, the eight branch instructions support additional tests of the ALU flags which support both signed and unsiged comparisons. The four signed conditional branches supported are: less than, less than or equal, greater than, and greater than or equal. The four unsigned conditional branches supported are: lower than, lower than or same, higher than, and higher than or same. These conditional branches are enabled by letting the 16-bit comparison instructions set the V flag.)

The following table provides the instruction lengths (cycles) for the M65C02A-specific instructions which support the implementation of FORTH VMs:

```
                DTC     ITC

NXT             1(3)                ; NEXT
ENT             1(5)                ; ENTER/CALL/DOCOLON
PLI NXT         2(6)                ; EXIT
--
IND NXT                 2(6)
IND ENT                 2(8)
PLI IND NXT             3(9)
--
PLI                 1(3)            ; Pop IP
PHI                 1(3)            ; Push IP
INI                 1(1)            ; Increment IP
--
IND PLI             2(4)            ; PLW - Pop W
IND PHI             2(4)            ; PHW - Push W
IND INI             2(2)            ; INW - Increment W
```

```
--
LDA ip,I++          2(4)            ; Load byte from IP++ into A
SIZ LDA ip,I++      3(5)            ; Load word from IP++ into A
IND LDA ip,I++      3(7)            ; Load byte from IP++ indirect into A
ISZ LDA ip,I++      3(8)            ; Load word from IP++ indirect into A
--
STA ip,I++          2(4)            ; Store byte in A at IP++
SIZ STA ip,I++      3(5)            ; Store word in A at IP++
IND STA ip,I++      3(7)            ; Store byte in A at IP++ indirect
ISZ STA ip,I++      3(8)            ; Store word in A at IP++ indirect
--
ADD ip,I++          2(4)            ; Add byte from IP++ into A
SIZ ADD ip,I++      3(5)            ; Add word from IP++ into A
IND ADD ip,I++      3(7)            ; Add byte from IP++ indirect into A
ISZ ADD ip,I++      3(8)            ; Add word from IP++ indirect into A
--
IND DUP             2(2)            ; TAI - Transfer A to IP
SIZ DUP             2(2)            ; TIA - Transfer IP to A
ISZ DUP             2(2)            ; XAI - Exchange A and IP
```

The M65C02A implements a base-pointer relative addressing mode. This mode can be used to directly access variables on the PS or the RS. Thus, the address calculations required to access the PS and RS required when using a 6502/65C02 processor are not required with the M65C02A core. This addressing mode, although also applicable to other HLLs, is expected to significantly improve the performance of FORTH programs on the M65C02A core relative to the same programs on 6502/65C02 FORTH VM implementations.

This concludes the FORTH VM trade study for the M65C02A soft-core processor.