

# DOCUMENT CHANGE LOG

| REVISION LETTER | REVISION DATE | REVISION AUTHORITY | PAGE AFFECTED | REMARKS                          |
|-----------------|---------------|--------------------|---------------|----------------------------------|
| -               |               | Michael A. Morris  | All           | Initial development and release. |

# Reference Manual

## M65C02A

### An Enhanced Microprogrammed 6502/65C02-compatible Processor Core



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 2 OF 56

# Table of Contents

|   |           |
|---|-----------|
| <b>TABLE OF CONTENTS .....</b>  | <b>3</b>  |
| <b>LIST OF TABLES .....</b>   | <b>6</b>  |
| <b>TABLE OF FIGURES.....</b>  | <b>6</b>  |
| <b>1. GENERAL DESCRIPTION .....</b>   | <b>7</b>  |
| 1.1. GENERAL DESCRIPTION OF THE M65C02A MICROCOMPUTER.....                      | 7         |
| 1.2. PROGRAMMER'S MODEL .....   | 8         |
| 1.2.1. Compatibility View.....  | 9         |
| 1.2.2. Extended Capabilities View .....   | 10        |
| 1.3. RESTRICTIONS .....   | 12        |
| <b>2. M65C02A CORE.....</b>   | <b>12</b> |
| 2.1. PROGRAMMER'S MODEL .....   | 13        |
| 2.1.1. Accumulators (A, X, Y).....  | 17        |
| 2.1.2. Index Registers (X, Y, A) .....  | 18        |
| 2.1.3. Stack Pointers (S <sub>K</sub> , S <sub>U</sub> , S <sub>X</sub> ) ..... | 19        |
| 2.1.4. Program Counter (PC) .....   | 20        |
| 2.1.5. Processor Status Word (P) .....  | 21        |
| 2.1.5.1. ALU Status Flags .....   | 22        |
| 2.1.5.1.1. N Flag – Bit 7 .....   | 22        |
| 2.1.5.1.2. V Flag – Bit 6 .....   | 22        |
| 2.1.5.1.3. Z flag – Bit 1 .....   | 22        |
| 2.1.5.1.4. C flag – Bit 0 .....   | 23        |
| 2.1.5.2. Processor Mode Flags .....   | 23        |
| 2.1.5.2.1. I Flag – Bit 2 .....   | 23        |
| 2.1.5.2.2. D Flag – Bit 3 .....   | 23        |
| 2.1.5.2.3. M flag – Bit 5 .....   | 23        |
| 2.1.5.3. B Flag – Bit 4.....  | 24        |
| 2.1.6. Virtual Machine Support Registers .....                                  | 24        |
| 2.1.6.1. VM Interpreter Pointer (IP) .....                                      | 24        |
| 2.1.6.2. VM Working Register (W) .....  | 25        |
| 2.2. M65C02A CORE PORTS .....   | 25        |
| 2.2.1. System Interface .....   | 25        |
| 2.2.1.1. Rst : input .....  | 26        |
| 2.2.1.2. Clk : input .....  | 26        |
| 2.2.2. Interrupt Handler Interface .....  | 26        |
| 2.2.2.1. IRQ_Msk : output .....   | 26        |
| 2.2.2.2. LE_Int : output .....  | 27        |
| 2.2.2.3. INT : input.....   | 27        |
| 2.2.2.4. xIRQ : input .....   | 27        |
| 2.2.2.5. Vector : input .....   | 28        |
| 2.2.2.6. VP : output .....  | 28        |
| 2.2.3. Set oVerflow Flag Interface.....   | 28        |
| 2.2.3.1. SO : input .....   | 28        |
| 2.2.3.2. SO_Clr : output .....  | 28        |
| 2.2.4. Core Status Interface .....  | 29        |
| 2.2.4.1. Done : output.....   | 29        |
| 2.2.4.2. SC : output .....  | 29        |
| 2.2.4.3. Mode : output .....  | 29        |
| 2.2.4.4. RMW : output .....   | 30        |
| 2.2.5. Memory Cycle Length Control Interface .....                              | 30        |
| 2.2.5.1. Wait : input .....   | 30        |



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 3 OF 56

|           |   |           |
|-----------|---|-----------|
| 2.2.5.2.  | Rdy : output.....   | 30        |
| 2.2.6.    | Memory Interface .....                                    | 31        |
| 2.2.6.1.  | IO_Op[1:0] : output.....                                  | 31        |
| 2.2.6.2.  | AO[15:0] : output.....                                    | 31        |
| 2.2.6.3.  | DI[7:0] : output.....                                     | 31        |
| 2.2.6.4.  | DO[7:0] : output.....                                     | 31        |
| 2.2.7.    | Co-processor Interface .....                              | 31        |
| 2.2.8.    | Core Internal State Interface.....                        | 32        |
| 2.2.8.1.  | X[15:0] : output.....                                     | 32        |
| 2.2.8.2.  | Y[15:0] : output.....                                     | 32        |
| 2.2.8.3.  | A[15:0] : output.....                                     | 32        |
| 2.2.8.4.  | IP[15:0] : output.....                                    | 32        |
| 2.2.8.5.  | W[15:0] : output.....                                     | 32        |
| 2.2.8.6.  | S[15:0] : output.....                                     | 32        |
| 2.2.8.7.  | P[7:0] : output.....                                      | 33        |
| 2.2.8.8.  | M[15:0] : output.....                                     | 33        |
| 2.2.8.9.  | IR[7:0] : output.....                                     | 33        |
| 2.2.9.    | Prefix Instruction Flag Interface .....                   | 33        |
| 2.2.9.1.  | IND : output.....   | 34        |
| 2.2.9.2.  | SIZ : output.....   | 34        |
| 2.2.9.3.  | OAX : output.....   | 34        |
| 2.2.9.4.  | OAY : output.....   | 35        |
| 2.2.9.5.  | OSX : output.....   | 35        |
| 2.3.      | M65C02A CORE COMPONENTS .....                             | 36        |
| 2.3.1.    | M65C02A Core .....  | 36        |
| 2.3.2.    | Microprogram Controller (MPC).....                        | 37        |
| 2.3.3.    | Address Generator.....                                    | 39        |
| 2.3.4.    | FORTH Virtual Machine.....                                | 39        |
| 2.3.5.    | Arithmetic and Logic Unit (ALU).....                      | 39        |
| 2.3.5.1.  | Load/Store/Transfer Unit (LST).....                       | 40        |
| 2.3.5.2.  | Logic Unit (LU) .....                                     | 41        |
| 2.3.5.3.  | Shift/Rotate Unit (SU).....                               | 41        |
| 2.3.5.4.  | Arithmetic Unit (AU) .....                                | 41        |
| 2.3.5.5.  | Write Select Generator.....                               | 42        |
| 2.3.5.6.  | Register A .....  | 42        |
| 2.3.5.7.  | Register X .....  | 43        |
| 2.3.5.8.  | Register Y .....  | 45        |
| 2.3.5.9.  | Register P .....  | 45        |
| <b>3.</b> | <b>ADDRESSING MODES .....</b>                             | <b>45</b> |
| 3.1.      | IMPLICIT/ACCUMULATOR .....                                | 47        |
| 3.1.1.    | Effect of the <i>ind/siz/isz</i> Prefix Instructions..... | 48        |
| 3.1.2.    | Effect of the <i>osx/oax/oay</i> Prefix Instructions..... | 48        |
| 3.2.      | IMMEDIATE [#IMM] .....                                    | 49        |
| 3.2.1.    | Effect of the <i>ind/siz/isz</i> Prefix Instructions..... | 49        |
| 3.2.2.    | Effect of the <i>osx/oax/oay</i> Prefix Instructions..... | 49        |
| 3.3.      | ZERO PAGE DIRECT [ZP].....                                | 50        |
| 3.3.1.    | Effect of the <i>ind/siz/isz</i> Prefix Instructions..... | 50        |
| 3.3.2.    | Effect of the <i>osx/oax/oay</i> Prefix Instructions..... | 51        |
| 3.4.      | PRE-INDEXED ZERO PAGE DIRECT [ZP,X] .....                 | 52        |
| 3.5.      | POST-INDEXED ZERO PAGE DIRECT [ZP,Y] .....                | 52        |
| 3.6.      | ZERO PAGE INDIRECT [(ZP)].....                            | 53        |
| 3.7.      | PRE-INDEXED ZERO PAGE INDIRECT [(ZP,X)] .....             | 53        |
| 3.8.      | POST-INDEXED ZERO PAGE INDIRECT [(ZP),Y].....             | 54        |



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

**-**

SCALE: NONE

SHEET 4 OF 56

|           |  |           |
|-----------|--|-----------|
| 3.9.      | RELATIVE [REL8] .....  | 54        |
| 3.10.     | ABSOLUTE [ABS] .....   | 54        |
| 3.11.     | PRE-INDEXED ABSOLUTE [ABS,X] .....                           | 54        |
| 3.12.     | POST-INDEXED ABSOLUTE [ABS,Y] .....                          | 54        |
| 3.13.     | ABSOLUTE INDIRECT [(ABS)] .....                              | 54        |
| 3.14.     | PRE-INDEXED ABSOLUTE INDIRECT [(ABS,X)] .....                | 54        |
| 3.15.     | ZERO PAGE RELATIVE [ZP,REL8] .....                           | 54        |
| 3.16.     | RELATIVE [REL16] .....                                       | 54        |
| 3.17.     | BASE POINTER RELATIVE [BP,B] .....                           | 54        |
| 3.18.     | POST-INDEXED BASE POINTER RELATIVE INDIRECT [(BP,B),Y] ..... | 54        |
| 3.19.     | IP-RELATIVE WITH AUTO-INCREMENT [IP,I++] .....               | 55        |
| <b>4.</b> | <b>M65C02A INSTRUCTION SET .....</b>                         | <b>55</b> |
| 4.1.      | ACCUMULATOR AND MEMORY INSTRUCTIONS .....                    | 55        |
| 4.1.1.    | Loads, Stores, and Transfers .....                           | 55        |
| 4.1.2.    | Logical Operations .....                                     | 55        |
| 4.1.3.    | Shift and Rotates .....                                      | 55        |
| 4.1.4.    | Arithmetic Operations .....                                  | 55        |
| 4.2.      | STACK INSTRUCTIONS .....                                     | 55        |
| 4.3.      | PROGRAM CONTROL INSTRUCTIONS .....                           | 55        |
| 4.3.1.    | Branches .....   | 55        |
| 4.3.2.    | Jumps .....  | 56        |
| 4.3.3.    | Subroutine Calls and Returns .....                           | 56        |
| 4.3.4.    | Interrupt Handling .....                                     | 56        |
| 4.4.      | PREFIX INSTRUCTIONS .....                                    | 56        |
| 4.5.      | REGISTER STACK INSTRUCTIONS .....                            | 56        |
| 4.6.      | FORTH VM INSTRUCTIONS .....                                  | 56        |
| 4.7.      | OTHER INSTRUCTIONS .....                                     | 57        |
| <b>5.</b> | <b>BOOT LOADER LISTINGS .....</b>                            | <b>57</b> |
| <b>6.</b> | <b>FIG-FORTH 1.0 LISTINGS .....</b>                          | <b>57</b> |



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 5 OF 56

## List of Tables

|   |    |
|---|----|
| Table 1: M65C02A Core Instruction Mode Output Definition.....           | 30 |
| Table 2: M65C02A Core IO_Op[1:0] Output Encoding.....                   | 31 |
| Table 3: M65C02A Core 16-bit Default Operation Size Instructions.....   | 34 |
| Table 4: M65C02A Core OSX Prefix Instruction Effects.....               | 35 |
| Table 5: M65C02A Core Modules.....                                      | 36 |
| Table 6: Notable 6502/65C02 zp Instructions Enhanced Using IND/SIZ..... | 51 |
| Table 7: Register Stack Instructions. ....                              | 56 |

## Table of Figures

|  |    |
|--|----|
| Figure 1: Block Diagram of M65C02A Microcomputer Using the M65C02A Core..... | 9  |
| Figure 2: M65C02A Compatibility View Programmer's Model. ....                | 10 |
| Figure 3: M65C02A Extended Capabilities View Programmer's Model. ....        | 11 |
| Figure 4: M65C02A Core Block Diagram.....                                    | 13 |



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 6 OF 56

# 1. General Description

The M65C02A synthesizable, microprogrammed processor core is an enhanced implementation of the MOS6502 Instruction Set Architecture (ISA) using a microprogrammed micro-architecture. The base instruction set of the M65C02A core is essentially that of the WDC W65C02S. In addition to the base instruction set, the M65C02A provides additional instructions and addressing modes. The unique M65C02A enhancements provide the following features:

- Two operating modes: Kernel (default) and User modes;
- Multiple accumulators: A, X and Y;
- Three stacks:  $S_K$ ,  $S_U$ , and  $S_X$ ;
- Base Pointer relative Addressing: bp,B and (bp,B),Y;
- Stack Pointer relative Addressing: sp,S and (sp,S),Y
- Selectable Indirection: single and double indirection;
- Selectable Operation Size: 8-bit (default) or 16-bit;
- Multi-level (3 level) register stacks: A, X, and Y;
- FORTH VM functional unit supporting Indirect and Direct Threaded Code (ITC/DTC);
- Block moves with independent source and destination addressing modes: Hold, Increment, and Decrement;
- Coprocessor interface for expansion of the CPU core.

In addition to the features listed above, the M65C02A core implements several other features that provide additional performance on the basic instructions:

- Two cycle branches (whether taken or not taken);
- No dummy cycles to cross page boundaries;
- No dummy cycles prior to stack operations;
- No dummy cycles during Read-Modify-Write (RMW) instructions.

This document provides a reference manual for the M65C02A core. This document will describe the M65C02A core in detail, and describe how to apply the M65C02A core in applications that may require a high-performance, compact 8/16-bit soft-core microprocessor.

## 1.1. General Description of the M65C02A Microcomputer

In order to demonstrate the M65C02A processor core, an example application of the M65C02A core has been developed: the M65C02A soft-core microcomputer. The M65C02A soft-core microprocessor is fully synthesizable, and wraps a number of synthesizable peripheral functions



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 7 OF 56

around the M65C02A core to produce a soft-core microcomputer which can be used as a stand-alone processor or integrated with other Intellectual Property (IP).

A block diagram of the M65C02A soft-core microcomputer is provided in Figure 1. The M65C02A soft-core microcomputer consists of the following components and peripherals:

- M65C02A core (synthesizable, enhanced 6502/65C02-compatible core)
- a Memory Management Unit (with support for Kernel and User modes)
- a Multi-Source (16) Interrupt Handler
- 28kB of memory (built from synchronous Block RAM)
- 1 Synchronous Peripheral Interface (SPI)
- 2 Universal Asynchronous Receiver/Transmitter (UARTs)
- 1 Multi-function Timer (TMR)
- and an External Memory Interface

As shown, the M65C02A soft-core microcomputer has been synthesized and targeted to several Xilinx Field Programmable Gate Arrays (FPGAs). In addition, the M65C02A soft-core microcomputer has been tested in hardware using a Xilinx XC3S200A-4VQ100I FPGA on two different platforms: the M65C02/M16C5x Development Board and the Chameleon Arduino UNO-compatible Shield board

## 1.2. Programmer's Model

The overarching design and implementation goal for the M65C02A core was for compatibility with the base instruction set of the 6502/65C02 ISA. The programmer's model of the M65C02A can be viewed from two perspectives: (1) the compatibility view and (2) the extended capabilities view.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 8 OF 56

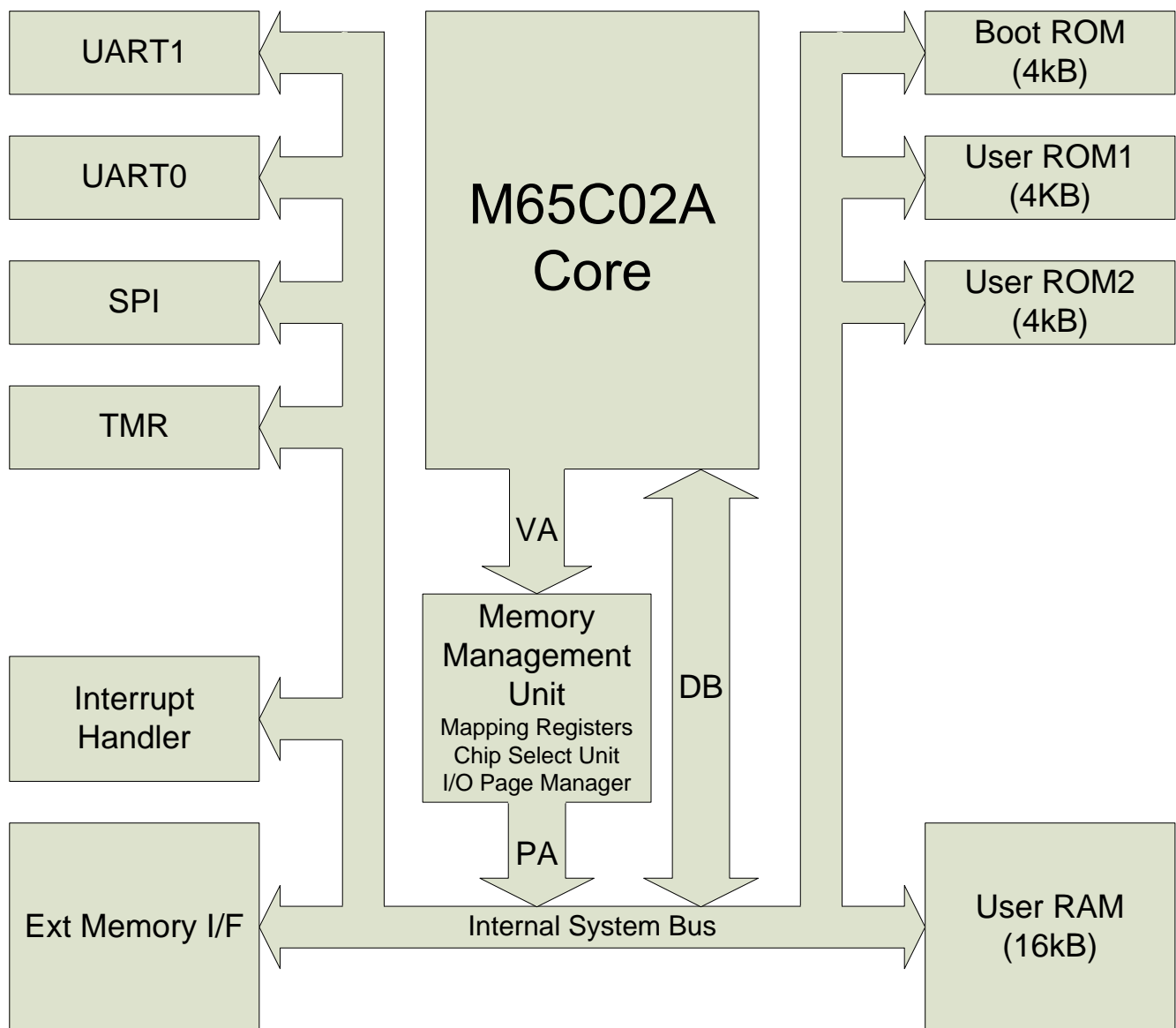


Figure 1: Block Diagram of M65C02A Microcomputer Using the M65C02A Core.

### 1.2.1. Compatibility View

The compatibility view of the programmer's model is simply that of the standard 6502/65C02 processors. That is, the programmer has access to a single 8-bit accumulator (A), an 8-bit pre-index register (X), an 8-bit post-index register (Y), an 8-bit system stack pointer register (S), an 8-bit Processor Status Word register (P), and a 16-bit program counter (PC). Figure 2 provides the compatibility view programmer's model for the M65C02A.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

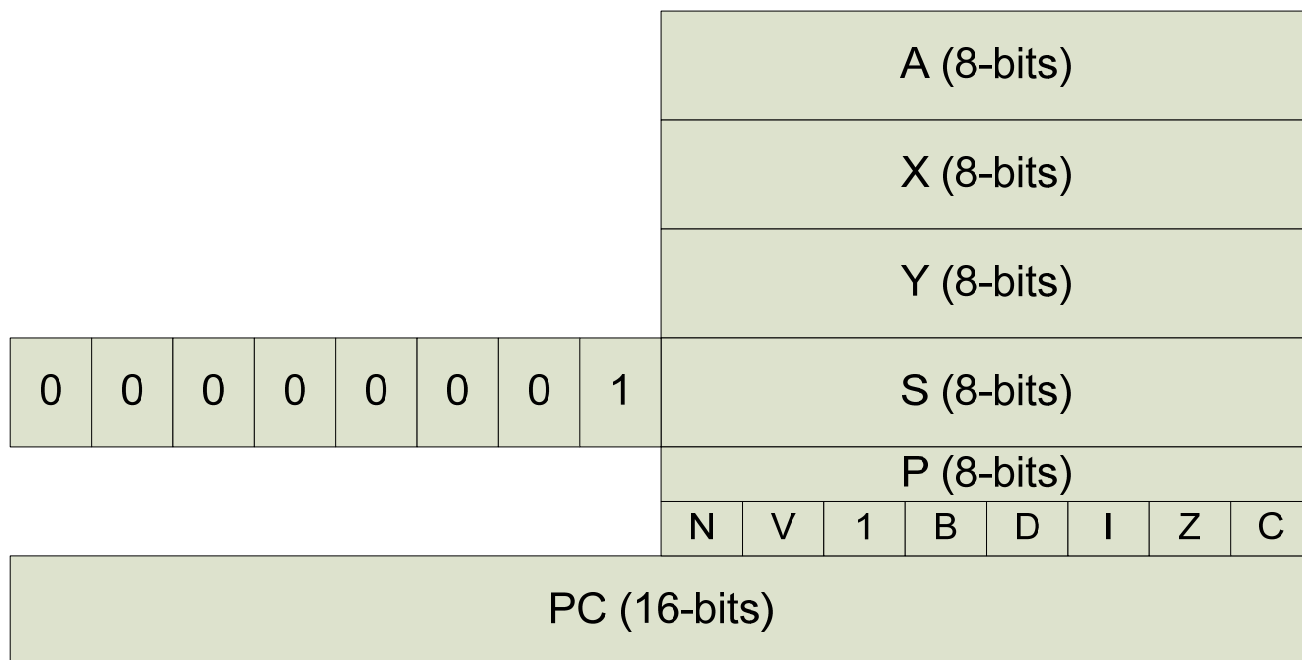
SCALE: NONE

SHEET 9 OF 56



## 1.2.2. Extended Capabilities View

The extended capabilities of the M65C02A core provide access to 16-bit arithmetic and logic operations, three 8/16-bit accumulators/index registers (3 deep register stacks), two 8/16-bit hardware stack pointers, and a 16-bit FORTH virtual machine (VM) core. Figure 3 shows the programmer's model for the extended capabilities view of the M65C02A core.



**Figure 2: M65C02A Compatibility View Programmer's Model.**

The three basic registers (A, X, and Y) have been extended from 8 bits to 16 bits. In addition, register override prefix instructions allow the functionality of the A and X registers and the A and Y registers to be swapped, OAX and OAY, respectively. Thus, when X is selected by the programmer to be an accumulator, the A register (the normal accumulator) becomes the pre-index register. Similarly, when the programmer selects the Y register as an accumulator, the A register becomes the post-index register.

In addition to the extended capabilities provided by the M65C02A's register override prefix instructions, each of the three basic registers has been converted into a three deep push down register stack. Normal load and stores of the registers affect only the Top-Of-Stack (TOS) register in each of these register stacks. New instructions have been added to the M65C02A instruction set to explicitly control the M65C02A register stacks:

- ***dup*** – duplicate the TOS register and push down the register stack;
- ***swp*** – swap/exchange the top two locations in the register stack;
- ***rot*** – rotate the registers in the register stack.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 10 OF 56

These three M65C02A instructions enable the use of the register stack and maintain compatibility with standard 6502/65C02 programs and subroutines. The register stack, when used appropriately, enables the M65C02A to perform extended precision 32-bit or 48-bit arithmetic and logic operations without resorting to multi-cycle load and store operations as normally required by a standard 6502/65C02 processor.

In addition to swapping the functionality of A with X or Y, the functionality of the system stack pointer S may be swapped with the X register using the *osx* prefix instruction. This capability provides the M65C02A with an auxiliary hardware stack pointer in the X register,  $S_x$ . When preceded by the *osx* prefix instruction, the instructions that normally affect the X register will operate on S instead. This allows S to be more easily manipulated than in a standard 6502/65C02 processor.

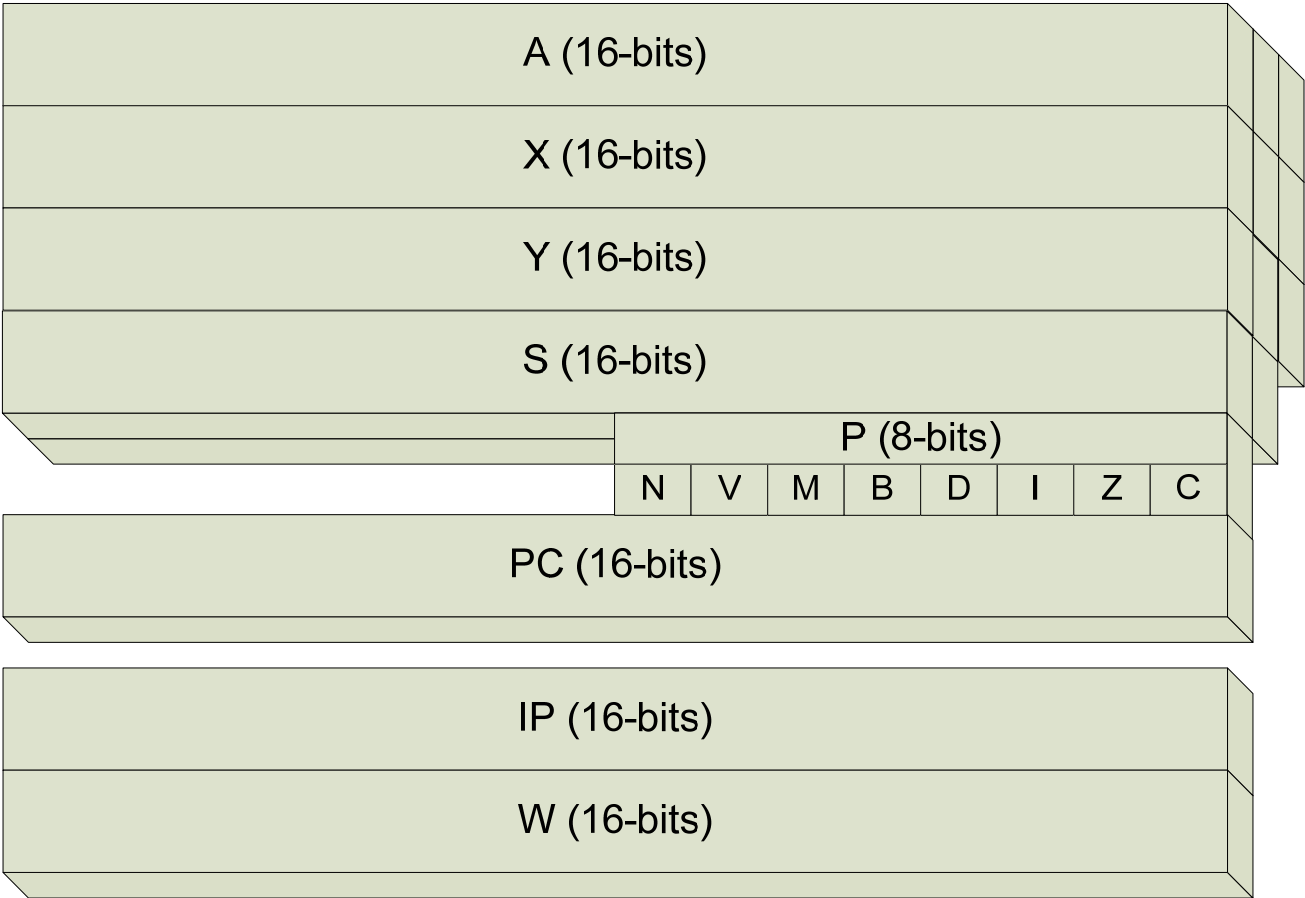


Figure 3: M65C02A Extended Capabilities View Programmer's Model.

Further, S has been extended to support two 16-bit stack pointers: (1) kernel mode stack pointer ( $S_K$ ), and (2) user mode stack pointer ( $S_U$ ).  $S_U$  is automatically selected whenever the Kernel/User mode bit (M) in the processor status word is set to logic 0, User mode. The M bit may only be changed by the *xti* instruction when the M65C02A is operating in the kernel mode. The M65C02A processor initializes in the Kernel mode on power up; this characteristic maintains compatibility with 6502/65C02 processors. The loading of auxiliary stack pointer  $S_x$  (*provided by*

X) is not restricted to the kernel mode, so user mode programs can readily use  $S_X$  as a stack pointer at any time.

Finally, the M65C02A incorporates several instructions and two 16-bit registers to support a FORTH Virtual Machine (VM), or other threaded-code interpreters. The IP and W registers provide the registers needed to implement a FORTH VM using either indirect threaded code (ITC) or direct threaded code (DTC). If these registers are not used as part of a FORTH VM implementation, they are available as general purpose 16-bit pointers and are supported by several M65C02A-specific instructions and extensions. (**Note:** *the other two standard FORTH threading models, Subroutine Threaded Code (STC) or Token Threaded Code (TTC), are supported by normal M65C02A instructions without the need for any additional special instructions.*)

## 1.3. Restrictions

The primary objective for the M65C02A core is to execute existing unmodified 6502/65C02 compatible programs. However, due to the pipelined nature of the M65C02A core, some behavioral differences were allowed that do not adversely impact most existing 6502/65C02 compatible programs. For example, a number of base and extended instructions are implemented as uninterruptable instructions: branches, jumps, subroutine calls, the *sei/cli* (Set Interrupt Mask/Clear Interrupt Mask) instructions, and the M65C02A-specific prefix instructions (*oax/oay/osx/ind/siz/isz*).

An M65C02A programmer must be aware of the uninterruptable nature of some of the core's instructions. If they are used in self-referencing loops, i.e. **here: bra here**, then interrupts, (*including non-maskable interrupts,*) will be effectively disabled/masked. Waiting for interrupts in a self-referencing loop is bad programming practice. If waiting for interrupts is necessary, then the programmer should use the standard **wai** (Wait for Interrupt) instruction instead of a self-referencing loop, or include an interruptable instruction, i.e. **nop**, inside the loop.

## 2. M65C02A Core

The M65C02A core provides an 8/16-bit synthesizable processor core intended to seamlessly emulate the instruction set of the MOS Technology MOS6502 and the California Micro Devices G65SC02-A microprocessors. In addition, the M65C02A core's instruction set includes the bit-oriented instructions found in the Rockwell R65C02 microprocessor, and the **wai** and **stp** (StoP) instructions found on the WDC W65C816 microprocessor. A block diagram of the M65C02A core is shown in Figure 4.

In many situations, 6502/65C02 processors require dead memory cycles. Given the simplicity of the 6502/65C02 memory interface, the dead memory cycles may interfere with interrupt generation and acknowledgement. This behavior is a particular issue with most 6502/65C02 microprocessors while they are executing Read-Modify-Write (RMW) instructions. The M65C02A executes its base instruction set in a manner similar to the 65CE02, which eliminates most dead memory cycles. The M65C02A has no dead memory cycles with the exception of three non-RMW enhanced/extended instructions: **phr rel16**; **pul dp**; **pul abs**.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 12 OF 56

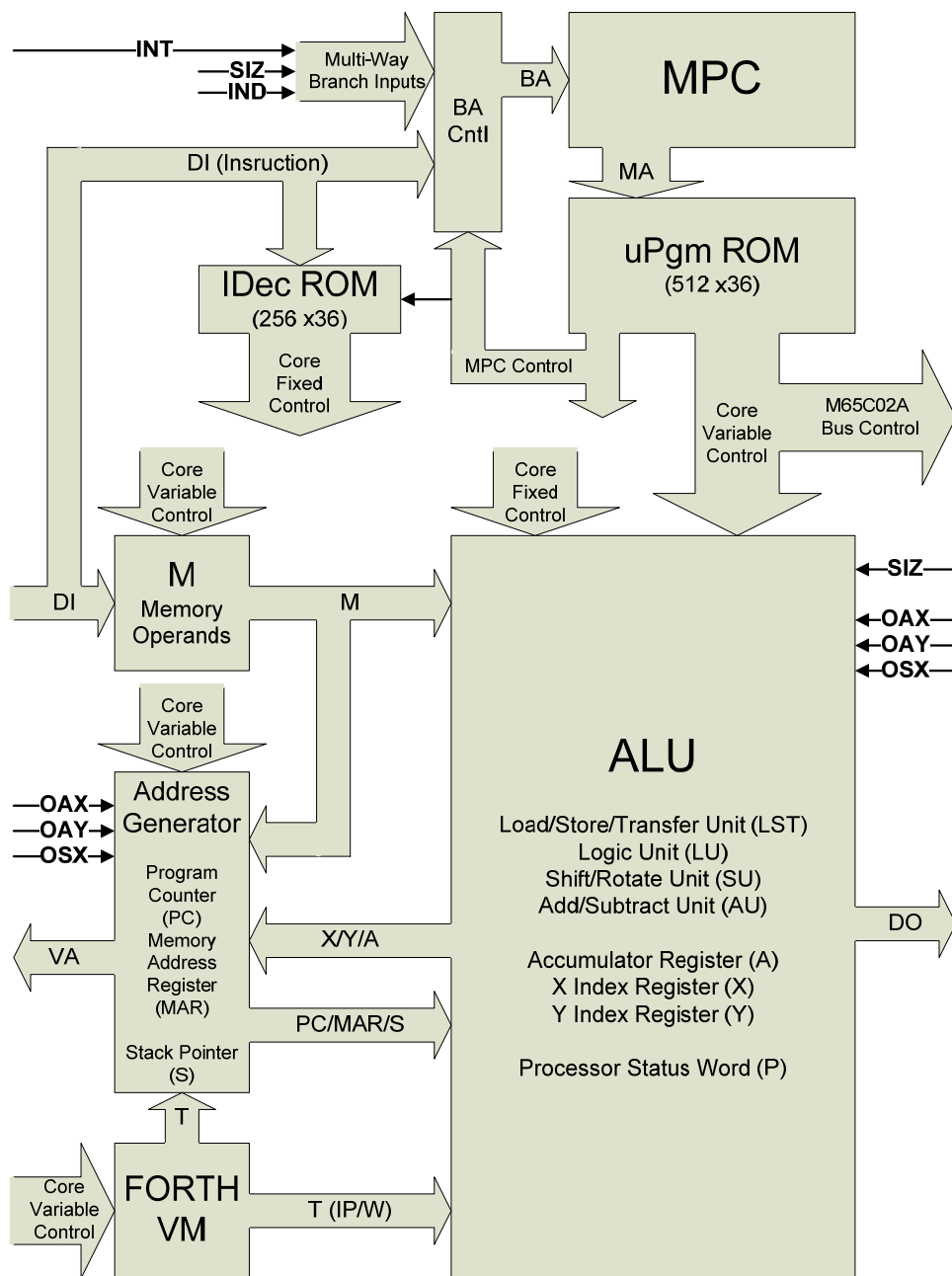


Figure 4: M65C02A Core Block Diagram.

## 2.1. Programmer's Model

The programmer's model for the M65C02A core was introduced in Figure 2 and Figure 3. The M65C02A core is intended to operate in a manner consistent with the instruction set of the 6502/65C02 microprocessors.

As previously discussed, the base instruction set of the M65C02A core is essentially that of the WDC W65C02S family of microprocessors. No mode changes are required to operate the M65C02A core compatibly with the base instruction set. If only the base instruction set is used,



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 13 OF 56

then the programmer is using the core in the compatibility view model as shown in Figure 2. To use the M65C02A core compatibly with the 6502/65C02 processors, the programmer only needs to avoid using any of the unused opcodes of those processors.

The M65C02A core expands on the base instruction set using the unused opcodes in the base instruction set. All except three of unused opcodes are used by the M65C02A core. No special mode switches are needed to take advantage of the additional features of the M65C02A core. The programmer simply uses any of the required instructions. Any 6502/65C02 assembler can be coerced into supporting the additional capabilities offered by the extended features of the M65C02A core. An assembler which provides macro support is more easily adapted to support the M65C02A core.

The programmer's model of the enhanced M65C02A core is that shown in Figure 3. The model shows the additional registers provided and that the additional registers are 16 bits in width. The following subsections will describe the features shown in Figure 3.

The 6502/65C02 microprocessors are a register-poor architecture. The limitations imposed by the lack of on-chip registers are mitigated by the richness of the addressing modes supported by the instruction set. Zero page memory, address range 0x0000...0x00FF, is supported by several addressing modes, and can be viewed as an extension of the on-chip registers. Thus, zero page memory provides programmers with as many as 256 8-bit or 128 off-chip 16-bit registers and/or pointers.

The 6502/65C02 microprocessors may be categorized as one address machines. This means that for instructions requiring two operands, one operand is found in an on-chip register, and the other is found in memory. The memory operand is located at the effective address defined by the addressing mode, and the register is implicitly addressed by the opcode of the instruction.

**(Note:** *unlike more conventional instruction encoding schemes, the 6502/65C02 microprocessors do not explicitly allocate a portion of their limited (8-bit) opcode for addressing one of the on-chip registers. Instead, the opcode itself is used. There was some rhyme and reason applied to the assignment of the opcodes, but as the instruction set became filled, the originally intended encoding rules became too restrictive. Rather than increasing the size of the opcode, the additional decoding complexity was transferred into the Programmable Logic Array (PLA) that functions as the instruction decoder and sequencer for 6502/65C02 processors. The M65C02A core does not use a PLA for instruction decoding and sequencing. Instead it uses microprogram Read-Only Memories (ROMs) for decoding and sequencing. Regardless of whether a PLA or ROM is used, no dedicated register select fields are necessary.***)**

In addition, the 6502/65C02 microprocessors are categorized as having an accumulator based Arithmetic and Logic Unit (ALU). An accumulator based ALU generally provides only a single register as an implicit source for one ALU operand and the same register serves as the destination for the ALU result. The 6502/65C02 processors deviate slightly from this model for single operand ALU operations because they can operate directly on memory. But for double operand ALU operations, the ALU result is always written to the A register. Thus, it may be appropriate to categorize the 6502/65C02 processors as having an accumulator-memory ALU. This characteristic of the 6502/65C02 ALU makes the accumulator a bottleneck, and frequently requires the



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 14 OF 56

accumulator to be loaded from and stored to memory. These additional memory cycles are generally required because the accumulator is only 8 bits in width, and many operations, e.g. address calculations, require more resolution (bits).

**(Note:** many microprocessors do not perform ALU operations directly on operands located in memory. The Intel and AMD x86 microprocessors are the most notable examples of processors with the capability of operating on operands in memory. Other notable examples are the Texas Instruments (TI) TMS9900-series of microprocessors (as used in the TI 99/A personal computer) and the Digital Equipment Corporation (DEC) PDP-10, PDP-11, and VAX-11 minicomputers.)

The M65C02A core alleviates some of the basic limitations of the 6502/65C02 microprocessors by adding the following features:

- (1) The M65C02A core allows the 6502/65C02 index registers, X and Y, to be used as accumulators. Although the one address, accumulator-based architecture of the 6502/65C02 microprocessors is preserved, three on-chip accumulators should make it easier for the programmer to keep extended width results in on-chip registers rather than loading and storing partial results from/to memory;
- (2) The M65C02A core allows the basic registers (A, X, Y, S) to be extended to 16 bits in width. To maintain compatibility with 6502/65C02 microprocessors, the default operation width of the registers and ALU operations is 8 bits. Internally, the upper byte of any register (A, X, Y, S) or the memory operand register (M) is forced to logic 0 (except for S which is forced to 0x01) unless the programmer explicitly extends the width of the operation with a prefix instruction;
- (3) The M65C02A core's ALU registers (A, X, and Y) are implemented using a modified, three level push-down register stack. This provides the programmer the ability to preserve intermediate results on-chip. The modification to the register stack is that load and store instructions only affect the TOS locations of the A, X, and Y register stacks. In other words, the TOS location of the register stacks is not automatically pushed on loads from memory, nor is it automatically popped on stores to memory. Explicit actions are required by the programmer to manage the contents of the register stacks associated with A, X, and Y;
- (4) The M65C02A core's X Top-Of-Stack register,  $X_{TOS}$ , serves as a base pointer for the base-relative addressing modes: **bp,B** and **(bp,B),Y**. This addressing mode provides the stack frame capability needed by programming languages like C and Pascal, and which must be emulated by 6502/65C02 microprocessors. **(Note:** base-relative addressing using  $X_{TOS}$  is generally associated with the system stack, but can be used in a more general way with any data structures in memory.)
- (5) The M65C02A core's  $X_{TOS}$  can function as a third (auxiliary) stack pointer,  $S_X$ , when instructions are prefixed with the **osx** instruction. **(Note:** when used as the auxiliary stack pointer, S becomes the source/target for all of the 6502/65C02 instructions specific to



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 15 OF 56



the *X* register: *ldx*, *stx*, *cpx*, *txa*, *tax*, *plx*, *phx*. This feature provides seven more ways to affect the system stack pointer: *lds*, *sts*, *cps*, *tss*, *tas*, *pls*, *phs*.)

- (6) The M65C02A core provides support for kernel and user modes. The previously unused and unimplemented bit of the processor status word (P), bit 5, is used to indicate the processor mode, M. The M65C02A core provides kernel mode and user mode stack pointers,  $S_K$  and  $S_U$ , respectively, for this purpose.  $S_U$  may be manipulated from kernel mode routines, but  $S_K$  is inaccessible to user mode routines. (**Note:** a 6502/65C02 program will stay in the kernel mode unless bit 5 (kernel mode) of the PSW on the system stack is cleared when a kernel mode *rti* instruction is performed. On reset, the M65C02A defaults to kernel mode for compatibility with 6502/65C02 microprocessors.)
- (7) The M65C02A core provides automatic support for stacks greater than 256 bytes. This feature is automatically activated whenever stacks are allocated in memory outside of memory page 0 (0x0000-0x00FF) or memory page 1 (0x0100-0x01FF). (**Note:** a limitation of this feature is that if the stack grows into page 1, then the mod 256 behavior of normal 6502/65C02 stacks will be automatically restored.)
- (8) The M65C02A core provides two prefix instructions, *ind* and *isz*, that add indirection to an addressing mode. When an indirection prefix instruction is applied, **indirection is performed before indexing**. (**Note:** a consequence of this rule is that the indexed zero page direct addressing modes, *zp,X* and *zp,Y*, are converted to post-indexed indirect addressing modes: *(zp),X* and *(zp),Y*. Similar behavior applies to the indexed absolute addressing modes. When *ind* or *isz* is applied to an indirect addressing mode, the result is a double indirection as expected. However, the rule that indirection is applied before indexing still applies. The result is that pre-indexed indirect addressing modes, *(zp,X)* and *(abs,X)*, translate into post-indexed double indirect addressing modes, *((zp)),X* and *((abs)),X*, instead of into pre-indexed double indirect addressing modes, *((zp,X))* and *((abs,X))*.)
- (9) The M65C02A core provides a prefix instruction, *isz*, which allows indirection to be added to the addressing mode while simultaneously increasing the width of the ALU operation from 8 to 16 bits.
- (10) The M65C02A core provides support for the implementation of virtual machines (VMs) for threaded interpreter's such FORTH. The M65C02A core's IP and W are 16 bit registers which support the implementation of DTC/ITC FORTH VMs using several dedicated M65C02A instructions.
- (11) The M65C02A core provides support for implementing application-specific coprocessors. Direct support for application-specific coprocessors allows an implementation based on the M65C02A core to be easily extended in a domain-specific manner.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 16 OF 56

## 2.1.1. Accumulators (A, X, Y)

The M65C02A core supports the standard register model of the 6502/65C02 microprocessors. When using the base instruction set, the programmer is restricted to using the A register as an 8-bit accumulator and the X and Y registers as 8-bit index registers.

However, the M65C02A core provides two prefix instructions, **oax** and **oay**, which swap the functionality of A with the functionality of X or Y for the following non-prefix instruction. Thus, if either of these prefix instructions precedes an instruction for which A is the normal accumulator, then index register, X or Y, becomes the accumulator, and A assumes the index function for that instruction.

*(Note: the lifetime of a prefix instruction is until the completion of the following non-prefix instruction; multiple prefix instructions may be applied. Furthermore, no attempt is made to declare as invalid **oax** or **oay** prefix instructions which are applied to accumulator instructions for an operation on X or Y which is already supported by a 6502/65C02 instruction. For example, applying **oax** to the instruction **inc a** will provide the same result as an **inx** instruction. The cycle count of the **oax inc a** instruction sequence is 1 cycle longer than the cycle count of the **inx** instruction, so such a construction, even if allowed by the M65C02A core, is not recommended because of the reduced efficiency.)*

The M65C02A core supports 16-bit operations for its standard registers. Two prefix instructions, **siz** and **isz**, allow the programmer to use the A, X, and Y registers as 16-bit registers. Although the default size is 8 bits in order to maintain compatibility with the 6502/65C02 microprocessors, the size of all ALU and stack operations related to these three registers can be promoted to 16 bits by using the **siz** and **isz** prefix instructions. *(Note: the **ind** and **siz** prefix instruction set the IND and SIZ flag registers as described in 2.2.9.1 and 2.2.9.2, respectively. The **isz** prefix instruction sets both flags simultaneously, and allows both addressing mode indirection and operation size control with a single prefix instruction. Some M65C02A instructions default to 16 bits, see Table 3.)*

When operated in the 8 bit mode, by default the upper half of a register is loaded with 0, but both halves are loaded with a 16-bit value when **siz/isz** (or if one of the 16-bit instructions listed in Table 3 is used). For example, instructions such as **lda #imm** may be promoted to from 8 bits to 16 bits, i.e. **siz lda #imm16**, and the instruction length increases by two bytes.

As described above, the M65C02A core implements the A, X, and Y registers as push-down register stacks. The load and store (and ALU) instructions only affect the top-of-stack registers of the push-down stacks for these registers: A<sub>TOS</sub>, X<sub>TOS</sub>, and Y<sub>TOS</sub>. This approach, which doesn't automatically push or pop the register stack when load or store instructions are used, requires the programmer to manage the register stack associated with each of these registers explicitly. This makes the register stack invisible to the programmer in compatibility mode. It also has the benefit that storing the top-of-stack register to memory does not automatically overwrite its value with the value of the next-on-stack (NOS) register due to an automatic pop of the register stack. Thus, the programmer can store the TOS value to multiple memory locations without requiring the repeated use of a duplicate TOS instruction, **dup**.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 17 OF 56



The M65C02A core provides three single byte instructions to affect the contents of the register stacks. These instructions assume that the target is the A register stack. The *oax* and *oay* prefix instructions will retarget these instructions to the X and Y register stacks.

The register stack management instructions provided are: *dup*, *swp*, and *rot*:

- (1) The *dup* instruction pushes the TOS down into the register stack.
- (2) The *swp* instruction swaps the TOS and NOS registers of the stack.
- (3) The *rot* instruction implements a circular shift/rotation of the register stack which puts the NOS into the TOS, BOS into NOS, and TOS into BOS.

Thus, these three stack management instructions, along with the *oax* and *oay* prefix instructions, allow the programmer complete access to nine on-chip 16-bit registers, three per register stack.

The Accumulator register stack has two operations not available to the X and Y register stacks. The bytes of the A<sub>TOS</sub> register can be swapped if the *swp* instruction is prefixed *ind*, i.e. *ind swp*: A<sub>TOS</sub>[15:0] <= {A<sub>TOS</sub>[7:0], A<sub>TOS</sub>[15:8]}. In addition, the bits of the A<sub>TOS</sub> register may be swapped if the *rot* instruction is prefixed by *ind*, i.e. *ind rot*: A<sub>TOS</sub>[15:0] <= A<sub>TOS</sub>[0:15]. These operations are not available for the X<sub>TOS</sub> and Y<sub>TOS</sub> registers because of the additional functions associated with those registers: auxiliary stack pointer and *mov* instruction source pointer (X<sub>TOS</sub>), and *mov* instruction destination pointer (Y<sub>TOS</sub>).


The N, V, Z, and C flags of the processor status word, P, are not affected by the register stack management instructions. Thus, pushing, swapping, or rotating the register stack has no effect on P unlike the load and transfer instructions. This register stack behavior allows the programmer to use the register stack to perform extended precision operations without having to spill registers to memory.

## 2.1.2. Index Registers (X, Y, A)

The M65C02A core provides the standard 8-bit index registers, X and Y, found in the 6502/65C02 microprocessors. In addition, the 6502/65C02 accumulator, A, can also be used as an index register. A is used as an index register whenever the register override prefix instructions, *oax* and *oay*, swaps A with the X or Y index registers; and instruction uses an indexed addressing mode.

The M65C02A core makes an additional subtle change in the behavior of the index registers that can have a significant impact on the operation of the indexed addressing modes. The effective address calculation for indexed addressing modes for the 6502/65C02 microprocessors can best be described by the following equation:

$$A_{effective} = M + index,$$

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 18 OF 56                     |                 |

where  $M$  is the memory operand address which is either 8 bits for zero page or 16 bits for absolute addressing modes. The subtlety in the preceding equation is that the *index* register (X, Y, or A) may have been previously loaded with a 16-bit value. Thus, the programmer can think of the address in an instruction as an offset from a base address which is found in one of the index registers. In other words, when any of the M65C02A core's ALU registers have been programmed as 16-bit values, the preceding effective address equation can be written as:

$$A_{effective} = offset + base,$$

where *base* is one of the three index registers (X, Y, or A) and *offset* is the address operand embedded in the instruction. This formulation provides the programmer much better addressing than the standard 6502/65C02 microprocessor indexed addressing modes.

The programmer has the discretion to treat either the memory address operand embedded in the instruction as an offset or a base address. A standard assembler or compiler will not use the base plus offset paradigm, but a macro assembler can be coerced to provide the programmer with this improved view of the M65C02A's effective address calculation. This view with X, Y, and A as base address registers is expected to provide better High Level Language (HLL) support.

### 2.1.3. Stack Pointers ( $S_K$ , $S_U$ , $S_X$ )

The M65C02A core provides the standard system stack pointer, S, of the 6502/65C02 microcomputers. Within the M65C02A core, S is implemented as a 16-bit register. Thus, it may be used in a 16-bit manner, but with normal 6502/65C02 instructions its operation will automatically ensure that it is locked to page 1 of memory just as expected in 6502/65C02 microcomputers.

The M65C02A core supports two stack pointers: (1) kernel mode stack pointer  $S_K$ , and (2) user mode stack pointer  $S_U$ . On power up, reset, interrupts, or *brk*, the M65C02A core's mode is automatically set to the kernel mode and the processor state is automatically saved in the kernel mode stack. The programmer must initialize the user stack pointer  $S_U$  and force a transition to the user mode in order to use  $S_U$ .

While in the kernel mode, the programmer must use the *ind txs* or the *isz txs* instruction sequences to transfer  $X_{TOS}$  to  $S_U$ . In kernel mode, it is the IND flag register that directs the  $X_{TOS}$  value into  $S_U$ , and that flag is set by the *ind* or *isz* prefix instructions. If the SIZ flag register is not set, then  $S_U$  page will be initialized to 1, i.e. the normal memory page for the system stack. Thus, the *isz* prefix instruction is used to set  $S_U$  in a page other than page 1. Also, while in the kernel mode, use either the *ind tsx* or the *isz tsx* instruction sequences to transfer  $S_U$  to  $X_{TOS}$ .

**(Note:** in the kernel mode, the *osx txa* and *osx tax* instructions combinations are valid, and allow the system stack pointer  $S_K$  (or  $S_U$ , if preceded by *ind* or *isz*) to be transferred to/from A, respectively. In the user mode, the *osx txa* and *osx tax* instructions combinations are also valid, but only allow the user stack pointer  $S_U$  to be transferred to/from A, respectively.)



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 19 OF 56

While in user mode, the programmer may load  $S_U$  using the *txs* and *osx ldx (lds)* instruction sequences. While in the user mode, the programmer may store  $S_U$  using the *tsx* and *osx stx (sts)* instruction sequences. Finally, while in user mode, the programmer may modify  $S_U$  using the *osx inx (ins)* and *osx dex (des)* instruction sequences.

**(Note:** while in user mode, use of the *ind* and *isz* prefix instructions does not allow the user mode  $X_{TOS}$  to be written to  $S_K$ , or vice-versa.)


The M65C02A core's  $X_{TOS}$  can function as a second, auxiliary stack pointer,  $S_X$ . Preceding any stack instruction by the *osx* prefix instruction will result in the  $X_{TOS}$  register being used as the stack pointer. As previously discussed, the programmer can then use any of the 6502/65C02 dedicated X register instructions to modify the system stack pointer ( $S_K$  or  $S_U$ ). Since the  $X_{TOS}$  can also be used as accumulator, the auxiliary stack pointer supported by  $X_{TOS}$  can be more easily manipulated using the full power of the ALU.

This capability is expected to make X very attractive as the either the parameter or return stack for a FORTH VM. The need for the *osx* prefix instruction to access the stack pointer capabilities of  $X_{TOS}$  will result in an additional memory cycle for any stack operations that use  $X_{TOS}$  compared to those the use S. However, the M65C02A core's implementation saves one memory cycle per stack access compared to a 6502/65C02 microprocessor. So the additional memory cycle required for the *osx* prefix instruction simply makes the M65C02A core's  $X_{TOS}$  stack operations cycle length compatible with 6502/65C02 system stack operations.

Finally, the M65C02A core allows the programmer to load the upper 8 bits of any of the stack pointers, which allows the system and auxiliary stacks to be placed anywhere in the address space of the 6502/65C02/M65C02A. Under standard usage, with the stacks locked to page 1 (or page 0), the normal 256 byte stack limit is automatically imposed, i.e. modulo 256 address calculations are automatically performed. Thus, like the 6502/65C02 system stack pointer, S, the M65C02A core's stack pointers behave in a manner that is transparent to the programmer. However, if a stack's page is not page 1 (or page 0), then the standard 6502/65C02 modulo 256 behavior for stacks is not imposed and the stack size becomes unlimited, i.e. modulo 65536.

## 2.1.4. Program Counter (PC)

The M65C02A core has a standard 16-bit program counter (PC). The PC points to the next instruction byte. The 6502/65C02 microprocessors use a variable length instruction which varies from one to three bytes in length. Instruction length for the M65C02A core may be longer because of the prefix instructions. **(Note:** for the M65C02A core, the number of prefix instructions which may be applied is determined by the programmer. Typically, only two or three non-interruptable prefix instructions are required to modify a base instruction into an extended instruction. So in typical situations, M65C02A core instruction lengths will vary from 1 to 6 bytes in length, with the typical length being 2 to 4 bytes. However, the M65C02A core does not impose a limit on the number of prefix instructions that may precede an instruction opcode, so an instruction can be as long as 65536 bytes, inconceivable as that may be.)

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 20 OF 56                     |                 |

In 6502/65C02 microprocessors, unlike most microprocessors, subroutine calls leave the PC pointing at the last byte of the current instruction rather than the first byte of the next instruction. Thus, to return from the subroutine correctly, the address popped from the stack by the *rts* instruction must be incremented by one before the opcode of the next instruction may be read from memory. The M65C02A core emulates this behavior.

Interrupts behave differently than subroutine calls in a 6502/65C02 microprocessor. In 6502/65C02 microprocessors, interrupts are evaluated before execution of an instruction is begun. As such, the PC is not advanced if an interrupt is to be taken. This action results in the address placed on the stack being the address of the current instruction. Therefore, a return from an interrupt, *rti*, on a 6502/65C02 microprocessor does not need to advance the PC by one before fetching the opcode of the interrupted instruction.


Traps like the *brk* instruction behave differently than interrupts in a 6502/65C02 microprocessor. The *brk* instruction in a 6502/65C02 microprocessor advances the PC by two rather than by one after reading the opcode of the *brk* instruction. In other words, to return to the instruction after the *brk* instruction, a 6502/65C02 microprocessor interrupt service routine for *brk* must decrement the address on the stack by 2 before returning from the trap. For traps, the M65C02A core leaves the PC pointing to the address after the trap instruction. This means the M65C02A core's interrupt service routine must decrement the PC on the stack by one rather than 2 before returning from the trap.

Unlike most 6502/65C02 microprocessors, the M65C02A core treats interrupts, traps, and subroutines calls uniformly. That is, both the *rts* and *rti* instructions increment the PC by one before fetching the opcode of the instruction to which the processor should return following the completion of a subroutine or interrupt/trap service routine. This means that interrupts are evaluated when an instruction completes rather than before it begins as in the 6502/65C02 microprocessors. This action pushes as the PC the address of the last byte of the instruction and makes the adjustment required to the PC on return from an interrupt service routine the same as that required after returning from a subroutine. This implementation is specific to the M65C02A core but does not incur any incompatibilities with software/firmware for 6502/65C02 microprocessors except as noted above for the *brk* instruction.

*(Note: given the nature of the M65C02A core's internal micro-architecture, it is possible to emulate the behavior of the 6502/65C02 microprocessors for subroutines, interrupts, and traps without changing any logic: simply change the contents of the microprogram memories. However, it was not considered essential for compatibility to emulate any "features" of the 6502/65C02 microprocessors that could be easily accounted for software/firmware that normally must be adapted for any particular application such as interrupt service routines.)*

## 2.1.5. Processor Status Word (P)

The M65C02A core's processor status word P contains the ALU status flags and processor control flags. With the exception of the User/Kernel mode flag which is unique to the M65C02A core, the remaining bits are the same as those found on a 6502/65C02 microprocessor. The register models provided in Figure 2 and Figure 3 define the P register.

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 21 OF 56                     |                 |

### 2.1.5.1. ALU Status Flags

The ALU status flags are N, V, Z, and C. The bit locations for these flags in P are 7, 6, 1, and 0, respectively. The N is the Negative flag, V is the arithmetic oVerflow flag, Z is the Zero flag, and C is the Carry flag.

#### 2.1.5.1.1. N Flag – Bit 7

The N flag is generally set when the last ALU result is negative. (**Note:** a 2's Complement number is assumed to represent a negative value when its most significant bit is set, or equal to logic 1. The ALU of the 6502/65C02 microprocessors uses the 2's Complement representation.) Since the registers of the M65C02A core are attached to the output of the ALU, any write to a register through the ALU will affect the N flag. (**Note:** the *tsx* instruction which transfers the system stack pointer S into the X index register does not affect the N flag. In a 6502/65C02 microprocessor, the transfer of S to X does not go through the ALU. Consequently, *tsx* instruction does not affect any of the ALU flags in the M65C02A core.)

In addition, the N flag is set to the value of bit 7 of the operand of *bit* instructions. (**Note:** the *bit #imm* instruction is an exception. The *bit #imm* instruction does not affect the N flag.)

#### 2.1.5.1.2. V Flag – Bit 6

The V flag indicates an overflow of the signed 2's complement arithmetic operations of the *adc* and *sbc* instructions. It indicates that a carry was sensed out of bit 6 into bit 7 of the ALU, but no carry was generated out of bit 7. The 6502/65C02 *cmp/cpx/cpy* instructions function strictly as an unsigned comparison, so it does not modify the V flag. The M65C02A core allows the 16-bit versions of the *cmp/cpx/cpy* instructions to modify the V flag. In this manner, the M65C02A enables 2's complement comparisons for 16-bit operands, which supports the enhanced conditional branch instructions supported by the core: *bgt/bge/blt/ble*, and *blo/bls/bhi/bhs*.

Like the 6502/65C02 microprocessors, the M65C02A core provides support for an external set oVerflow input pin/port. A falling edge on the set oVerflow external pin/port, nSO, sets the V flag. There is no set oVerflow instruction, but the *clv* instruction can be used by the programmer to clear the state of the V flag. A falling edge on the external nSO pin/port can then be used to set the V flag. The state of the V flag can be tested in a variety of ways, but it is most easily tested using the *bvs* and *bvc* instructions.

The V flag is used by the coprocessor interface to indicate the result of the test of the coprocessor's status flags: Done and Busy. (**Note:** the V flag is modified by the co-processor instruction, *cop #imm8*. When the co-processor instruction tests the selected processor's status, the V flag is set using a mechanism similar to that used with the nSO pin/port. Thus, the programmer must clear the V flag prior to testing the coprocessor status flags if proper synchronization with a co-processor is required by the application. More details on the use of the V flag by the co-processor instruction are provided in the instruction description.)

#### 2.1.5.1.3. Z flag – Bit 1

The Z flag indicates that the ALU result is zero.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 22 OF 56



#### 2.1.5.1.4. C flag – Bit 0

The C flag indicates that a carry has been generated by the ALU. The carry can be the result of an arithmetic carry out of bit 7: *adc*, *sbcb*, *cmp*, *inc*, or *dec*. The value of the C flag can also be affected by the value shifted out of bit 7 (or bit 15) or bit 0 when the programmer uses one of the shift/rotate instructions: *asl*, *lsl*, *rol*, or *ror*.

### 2.1.5.2. Processor Mode Flags

The 6502/65C02 microprocessors provide two processor mode flags, and the M65C02A core adds a third. The 6502/65C02 microprocessor mode flags are the D and I flags, and the M flag is specific to the M65C02A core.

#### 2.1.5.2.1. I Flag – Bit 2

The I flag, or interrupt mask flag, is set by the programmer to inhibit maskable interrupts, and cleared by the programmer to enable maskable interrupts. The I flag is automatically set on reset and when a trap or an interrupt service routine is entered.

#### 2.1.5.2.2. D Flag – Bit 3

The D flag, or decimal arithmetic flag, is set by the programmer whenever decimal addition and subtraction operations are needed. The D flag is automatically cleared by the 65C02 and the M65C02A core during reset, and when a trap or an interrupt service routine is entered. In the 6502, control of the D flag is strictly left to the programmer. The programmer may set the D flag using the *sed*, and clear the D flag using the *clb* instructions.

In the 6502/65C02 microprocessors, decimal mode arithmetic only applies to the *adc* and *sbcb* instructions. The comparison (*cmp/cpx/cpy*), increment (*inc/inx/inx*), and decrement (*dec/dex/dey*) instructions for the A, X, and Y registers are limited to binary arithmetic.

**(Note: the M65C02A core imposes another restriction to the use of decimal mode arithmetic: decimal mode arithmetic only applies to 8-bit addition and subtraction operations. Thus, the D flag is suppressed during 16-bit addition and subtraction operations.)**

#### 2.1.5.2.3. M flag – Bit 5

The M flag, or processor mode flag, determines whether the M65C02A core is operating in the Kernel (privileged) mode or in the User (non-privileged) mode. This flag is specific to the M65C02A core. It has no effect on the operation of the M65C02A core itself. However, the application in which the M65C02A core is being used can make use of the M flag to provide privileged and non-privileged instructions/operations, user and kernel mode address spaces, etc.

The M flag is set on reset by the M65C02A core. Furthermore, the M flag is set whenever a trap or an interrupt service routine is entered. Therefore, the kernel mode is the default processor mode after reset and during any interrupt service routine.

To enter the User mode, a return from interrupt (*rti*) instruction must load P from the kernel mode stack with the M flag (bit 5) cleared; the M flag is unchanged by a PLP instruction.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 23 OF 56

### 2.1.5.3. B Flag – Bit 4

Two bits in P of 6502/65C02 microprocessors are unimplemented: bit 5 and bit 4. Bit 5 has been used by the M65C02A core to implement the M flag. Bit 4 of the M65C02A core's P register behaves in the same manner as it does for 6502/65C02 microprocessors.

First, it is not implemented as a register. Second, it is set in two situations:

- (1) when the *php* instruction is used to push P onto the stack, bit 4 in P is set as P is being written to the selected stack;
- (2) when the *brk* instruction causes a trap, bit 4 is set in P as it is being pushed onto the kernel stack before the service routine is entered.

Thus, only when P is pushed onto the stack when entering an interrupt service routine for a maskable interrupt is the B flag cleared in the P on the stack. Therefore, if P (on the stack) is examined in the maskable interrupt's service routine, bit 4 will be set if the interrupt service routine was entered because the processor "took" a *brk* instruction otherwise bit 4 will be cleared. It is for this behavior that bit 4 of the P register of the 6502/65C02 microprocessors and the M65C02A core is generally known as the Break flag.

### 2.1.6. Virtual Machine Support Registers

The M65C02A core provides support for implementing Virtual Machines (VMs) such as those required to support the FORTH programming language. As such, the M65C02A core contains a module that provides two 16-bit registers needed to efficiently support Direct and Indirect Thread Code (DTC/ITC) implementations of a FORTH VM. Some FORTH VM implementations, such as Subroutine Threaded Code (STC) FORTHs, do not require any specialized registers or facilities, they can simply use the registers described in the following subsections as spare registers.

#### 2.1.6.1. VM Interpreter Pointer (IP)

The IP register provides the capabilities to operate as the interpretive pointer of a DTC/ITC FORTH VM. Support is provided in the expanded instruction set of the M65C02A core to use IP to move through a FORTH program. The M65C02A core's support for IP extends to providing the FORTH VM primitives *ent* to enter FORTH words, and *nxt* to execute the next FORTH word. Applying the *ind* prefix instruction to the *ent* and *nxt* instructions causes the M65C02A core to perform a second level of indirection using W. In performing these operations, the M65C02A core will automatically increment IP as needed.

In addition to these FORTH VM instructions, IP is supported by instructions to push IP (*phi*) and pull IP (*p1i*) from the stack, and to increment IP (*ini*) by 1.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 24 OF 56

### 2.1.6.2. VM Working Register (W)

The W register is a second 16-bit register expected to be used to support FORTH VMs. It generally points to the Code Field Address (CFA) of an ITC FORTH word. It is loaded automatically whenever the M65C02A core completes the first indirection through IP which is required by the *ent* and the *nxt* instructions. After *ent* pushes IP onto the FORTH VM's return stack, IP is loaded automatically from W. The FORTH VM then continues interpreting from that new address, with an *ind* prefix determining if single or double indirection is performed.

Using the *ind* to prefix the *phi*, *pli*, and *ini* instructions, the programmer has access to the W register: *phw*, *plw*, *inw*. The programmer's access to the W register is limited, but the programmer is still able to save, load, and increment the W register.

## 2.2. M65C02A Core Ports

The ports of the M65C02A core provide the interface to the application. The ports are organized by function:

- System Interface
- Interrupt Handler Interface
- Set Overflag Interface
- Status Interface
- Memory Cycle Length Control Interface
- Memory Interface
- Co-processor Interface
- Internal State Interface

The following subsections will define the ports for each of these interfaces. Appropriate timing diagrams of the signals are provided where appropriate.

**(Note:** *all of the ports of the M65C02A core are active high. In other words, a logic 1 is the asserted state of all signals into and out of the M65C02A core. Active low signals are not used within the M65C02A core in order to avoid naming conventions issues such as using leading lower case N or slashes, etc.)*

### 2.2.1. System Interface

The M65C02A core is designed to operate from a single clock. The core uses both edges of the clock. Thus, the duty cycle of the clock must be 50% in order for the M65C02A core to provide the best performance.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 25 OF 56



### 2.2.1.1. Rst : input

The core makes liberal use of the Rst reset signal. The primary use of the reset signal, beyond determining the initial state of the various registers of the core, is to support behavioral simulation of the core.

With the exception of the A, X, and Y registers, all registers in the core are reset. Synchronous resets are the predominant type, although a limited number of asynchronously reset registers are included. In addition, where necessary for correct behavior, the reset signal is stretched as necessary to ensure that pipelined components are fully reset.

*(Note: in the present implementation, only the Micro-Program Controller (MPC) stretches the Rst input signal. It stretches the Rst signal 1 clock cycle in to properly initialize the Block RAM microprogram memories and the MPC. The pipelined microprogram utilized by the M65C02A requires that its Rst be asserted for at least two cycles. This is due to the nature of the internal synchronous Block RAMs that are being used to store the M65C02A microprogram.)*

### 2.2.1.2. Clk : input

The M65C02A core's Clk port provides the single clock used throughout the core. Both edges of the clock are utilized, so the duty cycle and period jitter must be controlled. For best performance, the net supplying the clock should be connected to a low-skew clock net, and the duty cycle of the clock signal supplied to the Clk port must be 50%.

The jitter of the clock period must be tightly controlled, which implies that gated clocks should not be used for the M65C02A core clock. The design and implementation of the M65C02A core does not use gated clocks. The core's Clk net is directly connected to the clock ports core's registers.

*(Note: using both edges of the clock is an implementation decision. It enables single cycle behavior. However, the multi-threaded/multi-core implementation of the M65C02A core uses only the rising edge of the clock.)*

## 2.2.2. Interrupt Handler Interface

The M65C02A core expects that interrupts will be handled by an external interrupt handler. That handler is expected to prioritize interrupt requests as needed by the application, and to supply the interrupt vector when the core is ready to take the interrupt. The interrupt handler is expected to provide support for traps (BRK, ABRT, etc.), non-maskable interrupts, and maskable interrupts.

### 2.2.2.1. IRQ\_Msk : output

The interrupt request mask is a control signal from the M65C02A core to the external interrupt handler logic. IRQ\_Msk reflects the state of the I bit in the processor status word P. When set, IRQ\_Msk should inhibit the acceptance of maskable interrupts by the external interrupt handler.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 26 OF 56

### 2.2.2.2. LE\_Int : output

The M65C02A core signals the external interrupt handler using the LE\_Int to latch the maskable interrupts. This allows multiple maskable interrupts to be safely prioritized. Latching (registering) the maskable interrupts preserves the interrupt source until the interrupt handling microroutine of the M65C02A is ready to accept the interrupt vector from the interrupt handler.

The LE\_Int output signal is asserted by the M65C02A core after an interrupt has been signaled and recognized by the core's microprogram. Thus, the M65C02A core expects LE\_Int to cause the interrupt handler logic to hold the highest priority interrupt, at the time the interrupt is recognized by the core, until the vector is read by the core's interrupt handling microroutine.

The M65C02A core's interrupt handler pushes the address of the last byte of the current instruction, followed by the processor status word, and then reads the interrupt vector. After the interrupt vector is read by the core, the LE\_Int is deasserted. Only after LE\_Int is deasserted can the interrupt handling logic resolve the next interrupt request to the M65C02A core.

### 2.2.2.3. INT : input

The external interrupt handler signals the M65C02A core that interrupt request is asserted using the INT signal. While an interrupt is unserved by the core, Int will remain asserted for non-maskable interrupts and traps.

The state of the IRQ\_Msk output will regulate whether the maskable interrupt requests are accepted and passed to the core using the INT input signal. As defined above, if IRQ\_Msk is asserted, then the interrupt handler will not accept, or allow, any maskable interrupts. Thus, if only maskable interrupts are being requested while IRQ\_Msk is asserted, then INT will not be asserted unless a non-maskable interrupt is requested or a trap instruction is executed.

### 2.2.2.4. xIRQ : input

The xIRQ input is the logical OR of all of the maskable interrupt sources. The M65C02A core uses this signal in its implementation of the *wai*, wait for interrupt, instruction.

If the core executes a *wai* instruction with the IRQ\_Msk asserted, there exists the potential that *wai* would not exit unless a non-maskable interrupt was requested. Therefore, the xIRQ signal is used by the core to exit the *wai* instruction whenever a maskable interrupt is asserted while IRQ\_Msk is also asserted. This allows the WAI instruction to synchronize the core to the edge of external non-maskable interrupts or an asserted maskable interrupt.

In the case of a non-maskable interrupt, the core will take the interrupt and continue execution with the instruction following the *wai* instruction when the non-maskable interrupt service routine completes. In the case of maskable interrupts, an unmasked maskable interrupt will continue with the following instruction after the appropriate maskable interrupt service routine completes. In the case of a masked maskable interrupt, execution will continue with the instruction following the *wai* instruction, but no interrupt service routine is executed in this case.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

**-**

SCALE: NONE

SHEET 27 OF 56

### 2.2.2.5. Vector : input

The M65C02A core performs an indirect jump through the 16-bit address provided by the Vector input. The external interrupt handler generates the Vector address and the LE\_Int output from the core latches it in the external interrupt handler. In the case of multiple simultaneous interrupt requests, the Vector input may vary while the interrupt handler resolves which interrupt source will be serviced. When the M65C02A core determines it will service the INT input, it will assert the LE\_Int output and capture the Vector input on the following micro-cycle.

### 2.2.2.6. VP : output

The M65C02A core asserts the VP output during the two memory cycles that it requires to read the vector location for the address of the interrupt service routine. The external interrupt handler is expected to use the assertion of the VP output to unlatch and enable its interrupt processing logic that the M65C02A core previously latched with the core's assertion of its LE\_Int output.

## 2.2.3. Set oVerflow Flag Interface

Unlike most microprocessors, a 6502/65C02 microprocessor has an external input whose falling edge clears the V flag in its processor status word, P. This input, typically named SOB, can be used in a variety of ways. But it must be used carefully as the external input directly affects the status flags of the processor. Asynchronously modifying the V flag in P can have a detrimental effect on signed arithmetic operations because arithmetic overflow may be unexpectedly set by the asynchronous assertion of the external SOB signal.

The M65C02A core provides the same capability as a standard 6502/65C02 microprocessor to set the V flag in its processor status word. The implementation selected for the M65C02A core is synchronous, and expects logic external to the core to perform the necessary clock domain synchronization, edge detection, etc.

### 2.2.3.1. SO : input

The SO input sets the V flag in the processor status word of the M65C02A core. External logic is expected to provide whatever logic is necessary to synchronize the SO input to the clock domain of the M65C02A core. The M65C02A core will set the V flag in its P register at the completion of the instruction it is executing when the SO input is asserted.

### 2.2.3.2. SO\_Clr : output

The M65C02A core asserts the SO\_Clr output in response to the SO input. The M65C02A core expects the external logic driving the SO input to assert and hold the SO port until it is acknowledged by the SO\_Clr output. The M65C02A core will assert its SO\_Clr output during the micro-cycle in which it will be setting the V flag in its P register.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 28 OF 56

## 2.2.4. Core Status Interface

The M65C02A core's status is exported to external logic using its core status interface. The core status interface provides signals that reflect the instruction type and execution status. External logic, such as a memory interface, can use the core status interface signals to construct signals such as the common 6502/65C02 microprocessor SYNC and MLB signals.

### 2.2.4.1. Done : output

The M65C02A core asserts the Done output during the fetch of the next instruction. In essence, Done is asserted by the M65C02A core at the completion of the current instruction and the fetch of the next instruction. Given the pipelined nature of the M65C02A core's microprogram, the Done output is asserted during the memory cycle that completes any read-only instructions and which simultaneously reads the opcode of the next instruction, i.e. fetches the next instruction. For all other instruction types, Done is asserted during the fetch of the next instruction's opcode.

### 2.2.4.2. SC : output

The SC output is asserted by the M65C02A core for all instructions that are executed in a single memory cycle.

### 2.2.4.3. Mode : output

Each instruction supported by the M65C02A core is associated with a 3-bit Mode code. Mode is used to define which instruction opcodes are invalid or reserved in a particular implementation of the M65C02A core's instruction set. It is also used to define special instructions or special modes that the core's logic may use to implement specific behaviors. For example, certain instructions supported by a specific implementation may provide only support a 16-bit operation. These instructions can be marked with a specific Mode code and that specific characteristic identified for the core's logic, which can then configure the M65C02A core's functional units appropriately.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 29 OF 56

The following table defines Mode for the full implementation of the M65C02A core:

**Table 1: M65C02A Core Instruction Mode Output Definition.**

| Mode[2:0] | Mnemonic | Comment   |
|-----------|----------|---|
| 0         | VAL      | Signifies that the opcode fetched is a valid instruction.   |
| 1         | INV      | Signifies that the opcode fetched is an invalid instruction. The INV mode can be defined for any reserved or unused opcode, and if supported by an external interrupt handler, be used to generate an instruction trap.   |
| 2         | COP      | Signifies that the instruction fetched is the COProcessor instruction. The M65C02A core uses this mode code to enable any implemented coprocessors.   |
| 3         | BRK      | Signifies that the instruction fetched is the <i>brk</i> instruction.   |
| 4         | FTH      | Signifies that the instruction is a FORTH Virtual Machine instruction.  |
| 5         | SPC      | Signifies that the instruction requires special handling by the core's logic.   |
| 6         | PFX      | Signifies that the instruction is a prefix instruction. The M65C02A core logic specially handles instructions marked as prefix instructions in order to ensure that the programmer can apply multiple prefix instructions to a supported standard or extended M65C02A core instruction. |
| 7         | WAI      | Signifies that the <i>wai</i> instruction is being executed. The M65C02A core logic uses this mode to configure the test logic to sense the occurrence of non-maskable and maskable interrupts as described above in 2.2.2.4.   |

#### 2.2.4.4. RMW : output

The RMW output is asserted by the M65C02A core throughout the execution of a read-modify-write (RMW) instruction. It is asserted immediately after an RMW instruction is fetched and decoded, and remains asserted until a non-RMW instruction is fetched and decoded.

### 2.2.5. Memory Cycle Length Control Interface

The M65C02A core operates at the memory cycle rate. The memory cycle length control interface provides the mechanism by which external logic can extend any memory cycle of the M65C02A core.

#### 2.2.5.1. Wait : input

The Wait input is asserted by external logic to extend an M65C02A core's memory cycle.

#### 2.2.5.2. Rdy : output

The Rdy output is asserted by the M65C02A core to indicate that a memory cycle is complete. If Wait is not asserted, then Rdy is asserted on every clock cycle of the M65C02A core.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 30 OF 56

## 2.2.6. Memory Interface

The memory interface provides the control, address, and data ports by which the M65C02A core accesses memory. Unlike the memory cycles of 6502/65C02 processors, the M65C02A core's memory cycle may read, write, or perform no operation. For compatibility with 6502/65C02 processors, external logic can map the M65C02A core's no operation state to either a read or a write.

### 2.2.6.1. IO\_Op[1:0] : output

The IO\_Op[1:0] outputs provide the memory cycle control signals. The following table defines the memory interface actions encoded by the IO\_Op[1:0] outputs:

**Table 2: M65C02A Core IO\_Op[1:0] Output Encoding.**

| IO_Op[1:0]      | Mnemonic | Description  |
|-----------------|----------|--|
| 00 <sub>2</sub> | NOP      | M65C02A core does not read or write memory or I/O peripherals. |
| 01 <sub>2</sub> | WR       | M65C02A core writes to memory or I/O peripherals.              |
| 10 <sub>2</sub> | RD       | M65C02A core reads memory or I/O peripherals.                  |
| 11 <sub>2</sub> | Reserved | Reserved for future use, e.g. read from program memory.        |

### 2.2.6.2. AO[15:0] : output

The AO[15:0] outputs provide the 16-bit virtual address of the memory cycle. These outputs can be directly connected to memory and I/O peripherals in which case the M65C02A core is being used in an unmapped application. Alternatively, these outputs can be mapped by external logic to expand the physical address space accessible by applications using the M65C02A core.

### 2.2.6.3. DI[7:0] : output

The DI[7:0] inputs are the input ports for the data read from memory and/or I/O peripherals. These ports are applied directly to internal holding registers and simultaneously during instruction fetch cycles to the M65C02A core's microprogram ROMs.

### 2.2.6.4. DO[7:0] : output

The DO[7:0] outputs are the multiplexed data outputs of the M65C02A core. All register and ALU results are multiplexed internally by the core onto these ports during any write memory cycle.

## 2.2.7. Co-processor Interface

TBD



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 31 OF 56



## 2.2.8. Core Internal State Interface

The M65C02A's internal state is exposed using the Internal State Interface. The output ports defined for this function provide access to the current values of various registers within the M65C02A core.

### 2.2.8.1. X[15:0] : output

The X[15:0] outputs provide the current value of the Top-Of-Stack (TOS) of the X register stack, i.e.  $X_{TOS}$ . In addition to its function as the pre-index register, the M65C02A core's  $X_{TOS}$  register can also function as an accumulator, a post-index register, a stack pointer, and as a memory address pointer. When used in the 8-bit mode, the upper 8-bits of the  $X_{TOS}$  register are loaded automatically with 0.

### 2.2.8.2. Y[15:0] : output

The Y[15:0] outputs provide the current value of the TOS of the Y register stack, i.e.  $Y_{TOS}$ . In addition to its function as the post-index register, the M65C02A core's  $Y_{TOS}$  register can function as an accumulator and as a memory address pointer. When used in the 8-bit mode, the upper 8-bits of the  $Y_{TOS}$  register are loaded automatically with 0.

### 2.2.8.3. A[15:0] : output

The A[15:0] outputs provide the current value of the TOS of the A register stack, i.e.  $A_{TOS}$ . In addition to its function as the primary accumulator, the M65C02A core's  $A_{TOS}$  register can function as a pre-index and post-index register and as a counter. When used in the 8-bit mode, the upper 8-bits of the  $A_{TOS}$  register are loaded automatically with 0.

### 2.2.8.4. IP[15:0] : output

The IP[15:0] outputs provide the current value of the Interpretive Pointer (IP) of the FORTH Virtual Machine (VM) built into the M65C02A core. For FORTH, or any other interpreted VM, IP represents the program counter of the VM.

### 2.2.8.5. W[15:0] : output

The W[15:0] outputs provide the current value of the Working (W) register of the FORTH Virtual Machine (VM) built into the M65C02A core. For FORTH VMs, W is loaded with the value of the first indirect memory read. Thus, in both Direct Threaded Code (DTC) and Indirect Threaded Code (ITC) FORTH VMs, W points to the Code Field Address (CFA) of the FORTH word being executed.

### 2.2.8.6. S[15:0] : output

The S[15:0] outputs provide the current value of either the kernel mode or the user mode system stack pointer. In the 8-bit mode, the upper 8-bits are automatically loaded with a 0x01, which maintains compatibility with a standard 6502/65C02 processor. Either stack may be load-



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 32 OF 56

ed with a 16-bit value, which will be preserved until an 8-bit value is loaded. When loaded with a 16-bit value, the stack can be relocated to any page in the virtual address space of the M65C02A core.

#### 2.2.8.7. P[7:0] : output

The P[7:0] outputs provide the current value of the M65C02A core's processor status word register.

#### 2.2.8.8. M[15:0] : output

The M[15:0] outputs provide the current value of the M65C02A core's memory operand register. The memory operand register is the internal register that is loaded with all values read from memory during execution of an instruction. The upper 8 bits, M[15:8], behave in following special ways:

- (1) During the fetch of an 8-bit signed offset, such as that used for all relative branch instructions, M[15:8] are loaded with the sign bit of the value loaded into M[7:0], i.e. sign extension of the relative branch offset value.
- (2) During the fetch of the lower 8 bits of any other direct or indirect memory operand and value, i.e. M[7:0], M[15:8] are loaded with 0.
- (3) During the execution of the block move instruction, *mov*, or the co-processor instruction, *cop*, M[15:8] is loaded with the mode/operand byte of these instructions. The mode/operand byte controls the behavior of these instructions.

**(Note:** *within the M65C02A core, M[7:0] are mapped to the internal register OP1, and M[15:8] are mapped to the internal register OP2.*)

#### 2.2.8.9. IR[7:0] : output

The IR[7:0] outputs provide the opcode value of the current instruction being executed.

### 2.2.9. Prefix Instruction Flag Interface

The M65C02A core implements 6 prefix instructions. Three prefix instructions modify the addressing modes and the size of the operation, and three prefix instructions modify the function of the three base registers: A, X, and Y. Several enhancements to the instruction set architecture are realized when the prefix instructions, singly or in combination, are applied to the other M65C02A instructions.

Within the core, the prefix instructions set flag registers which are cleared when the immediately following, non-prefix instruction completes. This characteristic allows some prefix instructions to be used in combinations with others. Not all combinations are meaningful and may result in one or more of the prefix instructions used in combination having no effect, and will not cause an invalid instruction trap.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 33 OF 56



(**Note:** all of the M65C02A prefix instructions are implemented as single byte, non-interruptable instructions.)

### 2.2.9.1. IND : output

The IND prefix instruction flag is set by the *ind* (0x9B) and *isz* (0xBB) prefix instructions. It adds indirection to a direct, absolute, or zero page addressing mode. If *ind* or *isz* is applied to an indirect addressing mode, then another level of indirection is added.

If *ind* or *isz* is applied to an indexed addressing mode, the **indirection is performed before indexing**. Essentially this rule translates indexed zero page direct or absolute addressing modes into post-indexed indirect addressing modes. The rules also translates indexed zero page indirect or absolute indirect addressing modes into post-indexed double indirect addressing modes. (**Note:** *ind* or *isz* is ignored if asserted for immediate operands.)

### 2.2.9.2. SIZ : output

The SIZ prefix instruction flag is set by the *siz* (0xAB) and *isz* (0xBB). If applied to any ALU instructions, the size of the operation is changed from a default of 8 bits to 16 bits. There are some M65C02A-only instructions with a default size of 16 bits, and applying *siz/isz* to these instructions means that SIZ is ignored. The default size for the following M65C02A-only instructions is 16 bits:


**Table 3: M65C02A Core 16-bit Default Operation Size Instructions.**

| Mnemonic   |
|------------|
| PSH #imm16 |
| PSH abs    |
| PSH zp     |
| PHR rel16  |
| PUL abs    |
| PUL zp     |
| DUP        |
| SWP        |
| ROT        |

### 2.2.9.3. OAX : output

The OAX prefix instruction flag indicates that the roles of the A and X register will be swapped. For any ALU instructions, the X register becomes the left source operand and the destination register, and the A register assumes the role of the X register as an index register. For other instructions, if OAX is asserted, the indexing operation is performed using the contents of the A register instead of the X register. For example, *jmp (abs,X)* becomes *jmp (abs,A)*.

(**Note:** if either A is a 16-bit value, i.e. has a non-zero upper byte, the index operation is not truncated to 8 bits but is performed as a 16-bit operation. This allows indexed access to the full virtual address space of the core.)

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 34 OF 56                     |                 |

## 2.2.9.4. OAY : output

The OAY prefix instruction flag indicates that the roles of the A and Y register will be swapped. For any ALU instructions, the Y register becomes the left source operand and the destination register, and the A register assumes the index register role of the Y register. For example, `oay adc (zp),Y` becomes `adc.Y (zp),A`. This is an example where there is a base addressing mode, post-indexed zero page indirect, that does not need to be synthesized by adding the `ind/isz` prefix to the `oay` prefix to the base (zp),Y instructions. However, the same addressing mode can be obtained from the base indexed (by Y) zero page direct addressing mode if the `oay` prefix instruction is teamed with the `ind/isz` prefix, but at a penalty of an additional byte and instruction cycle.

**(Note:** if either A is a 16-bit value, i.e. has a non-zero upper byte, the index operation is not truncated to 8 bits but is performed as a 16-bit operation. This allows indexed access to the full virtual address space of the core.)

## 2.2.9.5. OSX : output

The OSX prefix instruction flag indicates that the roles of the S and X register will be swapped. For any stack operation instructions, the X register becomes the stack pointer. Furthermore, any instructions affecting the X register will modify S instead. This prefix is intended to allow the creation of an alternate and/or auxiliary stack pointer using X instead of S. It is also intended to allow direct manipulation of the system stack pointer using the instructions in the 6502/65C02 instruction set that directly manipulate the X register.

The following table shows the instructions intended to be affected by the `osx` prefix instruction:

**Table 4: M65C02A Core OSX Prefix Instruction Effects.**

| Base Instruction                             | Instruction prefixed by OSX                                |
|--|--|
| <code>LDX #imm/zp/zp,Y/abs/abs,Y</code>      | <code><u>LDS</u> #imm/zp/zp,Y/abs/abs,Y</code>             |
| <code>STX zp/zp,Y/abs</code>                 | <code><u>STS</u> zp/zp,Y/abs</code>                        |
| <code>CPX #imm/zp/abs</code>                 | <code><u>CPS</u> #imm/zp/abs</code>                        |
| <code>TAX/TXA</code>                         | <code><u>TAS/TSA</u></code>                                |
| Uses the System Stack ( $S_K$ , $S_U$ )      | Uses the Auxiliary Stack ( $S_X$ )                         |
| <code>PHA/PHP/PHX/PHY/PLA/PLP/PLX/PLY</code> | <code>PHA/PHP/<u>PHS</u>/PHY/PLA/PLP/<u>PLS</u>/PLY</code> |
| <code>PSH #imm16/zp/abs</code>               | <code>PSH #imm16/zp/abs</code>                             |
| <code>PHR rel16</code>                       | <code>PHR rel16</code>                                     |
| <code>PUL zp/abs</code>                      | <code>PUL zp/abs</code>                                    |
| <code>JSR abs</code>                         | <code>JSR abs</code>                                       |
| <code>BSR rel16</code>                       | <code>BSR rel16</code>                                     |
| <code>RTS/RTI</code>                         | <code>RTS/RTI</code>                                       |



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 35 OF 56

## Notes:

- (1) *ind*, *siz*, *isz* and *oay* may also be applied, with the expected effects.
- (2) *osx* applied to the base pointer (X) relative addressing modes has **NO** effect;
- (3) If X is being used as a system stack base pointer, using *osx* to push or pull values from an auxiliary stack will corrupt the base pointer unless it is saved before and restored after any auxiliary stack operation. The register stack for X can be used for this purpose, and the two cycle instruction sequence, *oax swp (swp X)*, can be used to maintain a system stack base pointer and an auxiliary stack pointer in on-chip registers.

## 2.3. M65C02A Core Components

The M65C02A core has been constructed using a number of modules. The modules comprising the M65C02A core are tabulated in Table 5 below:

**Table 5: M65C02A Core Modules.**

| Module              | Description  |
|---------------------|--|
| M65C02A_Core        | Top level module   |
| M65C02A_MPC         | Microprogram Controller  |
| M65C02A_AddrGen     | Address Generator (includes PC and S)  |
| M65C02A_SysStkPtrV2 | System Stack Pointer (includes User mode stack pointer)  |
| M65C02A_ForthVM     | Forth Virtual Machine  |
| M65C02A_ALUv2       | Wrapper module for the ALU module that implements the effects necessary for signed/unsigned extended branch instructions, the MOV instruction, and multi-precision compare instructions. |
| M65C02A_ALU         | Arithmetic and Logic Unit (includes A, X, Y, and P)  |
| M65C02A_LST         | ALU Load/Store/Transfer Multiplexer  |
| M65C02A_LU          | ALU Logic Unit   |
| M65C02A_SU          | ALU Shift/Rotate Unit  |
| M65C02A_AU          | ALU Adder Unit   |
| M65C02A_WrSel       | ALU Register Write Select Logic  |
| M65C02A_RegStk      | ALU Register Stack: A  |
| M65C02A_RegStkV2    | ALU Register Stack: X and Y  |
| M65C02A_StkPtr      | ALU Stack Pointer for X <sub>TOS</sub> (implements auxiliary stack logic) and Y <sub>TOS</sub>   |
| M65C02A_PSW         | ALU Processor Status Word (P) Register   |

The following sections will discuss the general characteristics of the modules defined in Table 5.

### 2.3.1. M65C02A Core

The M65C02A core module, M65C02A\_Core, ties together all of the components that comprise the M65C02A soft-core processor. Within this top level module, the modules are instantiated, but a working M65C02A soft-core processor consists of more than stringing the components together.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 36 OF 56

The core module provides the decoding of the various encoded control fields of the microprogram and module outputs. The module generates the internal ready signal, Rdy, that enables the M65C02A core to have its basic cycle extended. The internal ready signal can be delayed by the external wait state request signal (Wait), by the internal instruction complete signal (Done), and by the ALU operation complete signal (Valid).

The microprogrammed nature of the M65C02A core is enabled by two important multiplexers implemented in the top level core module. The first of these multiplexers controls the branch address field into the Microprogram Controller (MPC). The address provided by this multiplexer controls the behavior of the M65C02A microprogram with respect to instruction decoding, interrupt handling, and microprogram branching. The second multiplexer provides the multi-way branch offsets. These offsets are particularly important to the efficient implementation of the extended instruction set, and to the implementation of interrupts in the M65C02A core.


The two microprogram ROMs are inferred in the top level core module. The top level core provides the decoding of the 36-bit wide outputs of the two ROMs. The instruction sequencer ROM is decoded into named fields in order to make the implementation more understandable. The instruction decoder ROM is similarly decoded into named fields, but in addition, the Mode and Opcode subfields are further decoded to implement specialized instructions: break, coprocessor operations, Forth VM instructions, register stack instructions, and wait for interrupts.

The top level module also provides the instruction register (IR), and the two temporary registers ({OP2, OP1}), which provide storage for operands and data read from memory. These registers provide the M(emory) operand input into the M65C02A ALU. The registers also provide the zero page, absolute, and relative addresses for the M65C02A core's address generator. One particular implementation detail supported by the OP2 and OP1 registers is that OP2 is loaded with 0, the sign extension of OP1, or with dedicated control data for instructions like the M65C02A block move instruction. Finally, the {OP2, OP1} register pair also capture the interrupt vector provided by the external interrupt handler.

The logic to support the kernel/user operating mode of the M65C02A core is implemented in the top level module. During the processing of interrupts, the return address and P are stacked on the kernel mode stack. Therefore, the top level module provides the logic necessary to ensure that the transition back to the user, if required, does not occur until the return address and P register have been read from the kernel stack and the P register has been updated so that the instruction fetch occurs from the proper space.

## 2.3.2. Microprogram Controller (MPC)

The M65C02A core is microprogrammed using classical techniques. To control the flow of the microprogram the M65C02A core incorporates a microprogram controller, otherwise known as a microprogram sequencer. The purpose of the microprogram controller is to sequence through the microprogram sequences invoked by the instruction decode process. The M65C02A core's MPC provides several facilities/capabilities that directly support the development of the M65C02A core's base and extended instruction sets.

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 37 OF 56                     |                 |

The base instruction set utilizes no microprogram subroutines, i.e. micro-subroutines. In other words, the base instruction set uses only sequential microprogram sequences, conditional microprogram branches, or multi-way microprogram branches. The control microprogram is organized around the sequences necessary to implement the addressing mode, which do not generally require organizing commonly used sequences as microprogram subroutines.

*(Note: the control sequences needed to implement the various addressing modes could potentially share a number of individual states, but then these states would require multi-way branching in order to continue the control sequence needed to implement a particular addressing mode. This style of microprogrammed control introduces “spaghetti code” into a microprogram, making it difficult to read and understand much like a program which misuses **GOTOs**. The microprogram represents the low level control structure, and a well structured, easily read, and readily understood microprogram promotes maintainability and testability. Thus, microprogram sequences are only shared when the resultant does not negatively impact the readability and understandability of the overall microprogram.)*


The extended instruction set of the M65C02A required the implementation in the MPC of **relative multi-way branching**, but like the base instruction set, the extended instructions set does not require any micro-subroutines. Relative multi-way branching allows the microprogram to contain multi-way branch tables located at any microprogram address. Absolute multi-way branching requires that the branch tables are located at microprogram addresses which are multiples of the number of multi-way select bits. For example, a two bit wide multi-way select would require that the branch tables are located on multiples of four (4) microprogram words. This requirement increases the microprogram memory requirements, particularly if “spaghetti code” is to be minimized. The gaps created by the branch tables, and the difficulty in automatically controlling their placement, leads to increased use of **GOTOs** to increase the utilization factor of the microprogram memory.

The M65C02A MPC is based on the F9408 Microprogram Sequencer. A complete description and block diagram for this sequencer can be found on the bitsavers.org website in the Fairchild subdirectory of the PDF Documents Archive:

[http://bitsavers.trailing-edge.com/pdf/fairchild/\\_dataBooks/1975\\_Fairchild\\_Macrologic\\_Preliminary.pdf](http://bitsavers.trailing-edge.com/pdf/fairchild/_dataBooks/1975_Fairchild_Macrologic_Preliminary.pdf).

The M65C02A MPC is a reimplementaion of that microprogram sequencer with a few minor differences:

- (1) M65C02A MPC does not implement the input latches for the test inputs;
- (2) M65C02A MPC implements only the synchronous, or pipelined, mode of operation;
- (3) M65C02A MPC can be configured with a four level or a single level return stack;
- (4) M65C02A MPC reimplements the branch multi-way instruction of the F9408 as a **relative multi-way branch** instruction.
- (5) M65C02A MPC replaces the four conditional branch if test input low (BTLx) instructions of the F9408 with four **relative multi-way branch** instructions.

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 38 OF 56                     |                 |



The most significant difference between the M65C02A MPC and the F9408 sequencer is the operation of the multi-way branch instructions. The **relative multi-way branch** instruction of the M65C02A MPC provides a very flexible and powerful way to branch the microprogram. The M65C02A microprogram uses this facility to support the prefix instructions that provide the extended instruction set of the M65C02A core.

### 2.3.3. Address Generator

The M65C02A address generator provides the means by which all instruction and data memory addresses are generated. The address generator incorporates a dedicated adder and address operand multiplexers which readily allow the calculation of the effective address of instructions or data. The M65C02A core's program counter (PC) and system stack pointer registers ( $S_K$  and  $S_U$ ) are located in the address generator. In addition, the address generator incorporates a 16-bit temporary memory address register (MAR) that is used to hold sequential data memory addresses and for translating relative addresses into absolute addresses.

### 2.3.4. FORTH Virtual Machine


The FORTH Virtual Machine (VM) module provides the Interpretive Pointer (IP) and the Working (W) register necessary to the implementation of either an Indirect Threaded Code (ITC) or a Direct Threaded Code (DTC) FORTH VM. This module provides the 16-bit registers for the IP and the W register. In addition, the module includes an incrementer that allows IP and W to be incremented (by 1 or 2) to implement the operations of these registers in a FORTH VM. Finally, the module provides operations necessary to transfer W to IP, load IP/W, and store IP/W.

### 2.3.5. Arithmetic and Logic Unit (ALU)

The ALU provides the arithmetic, logic, and shift/rotate operations needed to implement the instruction set as either 8-bit or 16-bit operations. In addition, the ALU incorporates the registers for the accumulator (A), the two index registers (X and Y), and the processor status word (P).

The A, X, and Y registers are implemented as register stacks of three 16-bit registers each. This feature provides more on-chip register storage that relieve one of the more notable deficiencies in the 6502/65C02 instruction set architecture. Several register stack manipulation instructions provide single cycle (A), or two cycle (X/Y) operations on the register stacks. The TOS registers of these register stack also provide special operations that further enhance the M65C02A's performance.

The P register provides the ALU status flags, NVZC, and the processor status flags, MBDI. The P register supports the standard Set oVerflow (SO) operation provided in the 6502/65C02 architecture using an external, falling-edge sensitive pin, nSO (or SOB), which sets the V flag in P. In addition, the M65C02A P register supports the use of the V flag for testing the co-processor interface Busy and Done flags. To improve interrupt response, P is available to the core logic and to the LST multiplexer. The direct connection to the core's output data multiplexer supports stacking of the processor state during traps and interrupts.

|  |  |                  |                |                                    |                 |
|--|--|------------------|----------------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE      | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      | SHEET 39 OF 56 |                                    |                 |

The ALU is constructed from several multiplexers, functional units, and registers. The functional units drive a common result bus which, in turn, drives the input of the ALU output multiplexer. The ALU output multiplexer feeds the M65C02A core's output data multiplexer and loops around to the inputs of the following ALU registers: A<sub>TOS</sub>, X<sub>TOS</sub>, Y<sub>TOS</sub>, and P. The operand ports of the functional units which implement logical, shift/rotation, and arithmetic operations are driven by the functional unit operand multiplexer.

The ALU's registers (A, X, Y, P) and external operands (S, M, T) are connected to the common result bus by the Load/Store/Transfer (LST) functional unit. The LU, SU, and AU get their operands from the functional unit operand multiplexer, and each drives their results onto the common result bus. This common result bus is passed through the ALU output multiplexer onto the ALU output bus to the M65C02A core. For 8-bit ALU operations, the ALU output multiplexer drives only the low byte of the common result bus onto the ALU output bus, and zeroes the high byte of the ALU output bus. For 16-bit ALU operations, the ALU output multiplexer drive all 16 bits of the common result bus onto the ALU output bus.

All ALU registers, except P, are 16 bits wide. The functional unit operand multiplexer, the ALU output multiplexer, and the functional units all support 16-bit operands. All ALU operations are performed in a single cycle whether they are 8-bit or 16-bit operations. One limitation of the M65C02A ALU is that BCD arithmetic is only supported for 8-bit operations by suppressing the D flag to the AU a 16-bit operation is specified by the SIZ flag.


The ALU output bus is composed of the ALU output data, several ALU flags, a Valid output, and a condition code output. The Valid output indicates that the result of the requested ALU operation is present on the ALU output bus. Given the single-cycle operating mode of the ALU, Valid is effectively a pass through of the ALU module's Rdy input. The Rdy input indicates that the operands are available, whether in the memory operand holding register, M, or in an internal register of the ALU, address generator, or FORTH VM.

The Condition Code (CC) multiplexer is directly controlled by the core's microprogram. It provides an output that indicates the state of one of the bits in P. This enables the conditional and unconditional branch instructions. In addition, the CC multiplexer incorporates special tests to support *trb/tsb* and *bbxr/bbsx* instructions. Finally, the M65C02A core supports a number of 16-bit signed and unsigned tests, and the CC multiplexer implements the 8 additional tests supported: less than (<), less than or equal (≤), greater than (>), greater than or equal (≥), lower than (<),, lower than or same (≤), higher than (>), and higher than or same (≥). The first four of these tests are for signed tests, and the second four are for unsigned tests.

The register override prefix instructions are supported by having the functional unit operand multiplexer and the LST multiplexer perform the necessary input register overrides. The Write Select functional unit implements the destination register select modifications necessary to support the register override prefix instructions.

### 2.3.5.1. Load/Store/Transfer Unit (LST)

The LST module provides the means by which the input operands of the ALU are selected. In dual operand instructions, one operand is taken from one of the registers and the other is taken

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 40 OF 56                     |                 |

from memory, except in the case of register transfer instructions where one register is the source, and another register is the destination. In single operand instructions, the operand is either a register or data memory.

In most cases, the LST routes one of the core registers to the output of the ALU: A, X, Y, S, or P. The LST also routes the memory operand register M and the FORTH VM output T to the output of the ALU. In addition to providing normal register and/or operand routing for the M65C02A core, the LST module also implements the source operand multiplexing needed to support register override prefix instructions: *osx*, *oax*, *oay*.

The LST treats all registers as having a width of 16 bits. The LST output is truncated to 8 bits by the ALU output multiplexer before it is written into any ALU registers, the system stack pointer, or memory. The LST supports the following instructions: *lda/ldx/ldy*, *sta/stx/sty*, *tax/txa*, *tay/tya*, *tsx*, *pha/phx/phy*, *pla/plx/ply*, *php/plp*, *psh/phr/pul*, *phi*.

The input to the system stack pointer S and the input to the FORTH VM IP/W registers are directly connected to X and M, respectively, so the ALU output bus is not used for writing to the system stack pointer S or the FORTH VM IP/W registers. Therefore, the operands for the *txs* and the *pli* instructions are not routed through the ALU by the LST module.

### 2.3.5.2. Logic Unit (LU)

The LU module provides the means by which the ALU performs the bit-wise AND/OR/EOR of the accumulator and a memory operand. In addition, the LU module provides the operation needed to implement the bit-wise reset of memory and bits in P. The LU's OR function is also used to perform bit-wise set of memory and bits in P.

The LU is able to perform all of its functions with an 8-bit or a 16-bit operand. When an 8-bit operation is performed the upper byte is zeroed by default. The LU supports the following 6502/65C02 instructions: *and/ora/eor*, *bit*, *trb/tsb*, *rmbx/smbx*, *bbrx/bbsx*, *clc/sec/clv*, *cli/sei/cld/sed*.


### 2.3.5.3. Shift/Rotate Unit (SU)

The SU module provides the means by which the ALU performs shift and rotations of the accumulator or memory operands, i.e. read-modify-write operations. The SU supports 8-bit or 16-bit operations for all four 6502/65C02 shift/rotate instructions: *asl*, *rol*, *lsr*, *ror*.

### 2.3.5.4. Arithmetic Unit (AU)

The AU of the M65C02A core is a dual mode add/subtract unit. It is able to perform both decimal (BCD) and binary mode additions and subtractions. Binary mode is used for the increment instructions (*inc/inx/iny/ini*), decrement instructions (*dec/dex/dey*), and the comparison instructions (*cmp/cpx/cpy*).

The D flag in P selects whether decimal or binary additions or subtractions are performed. If D is set and an 8-bit operation is to be performed, then a decimal mode *adc/sbc* operation will be

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 41 OF 56                     |                 |



performed. If D is not set or a 16-bit operation is to be performed, then a binary mode *adc/sbc* operation will be performed.

The 6502/65C02 microprocessors do not perform 2's complement arithmetic for addition and subtraction unless the C flag in P is cleared before addition or set before subtraction. Essentially, addition is performed as the sum of the left and right operands plus C, and subtraction is performed as the sum of the left operand plus the complement of the right operand plus C. If the carry is set before addition, the sum would be plus 1. If the carry is not set before subtraction, then the difference would be the difference minus 1.

This configuration of the AU means that only one addition and subtraction instruction is required to perform multiple precision sums and differences. The penalty is that the programmer must clear or set C appropriately for the first addition/subtraction. The Intel/Zilog 8080/Z80 or the Intel x86 processors, use two addition and two subtraction opcodes in order to support single and multi-precision arithmetic.

### 2.3.5.5. Write Select Generator

The M65C02A Write Select Generator provides the select signals that enable the writing of the programmer visible registers: A, X, Y, P, and S. The M65C02A core utilizes a hybrid, split microprogram architecture. One portion of the microprogram is accessed once per instructions, and can be thought of as the instruction decoder. The other portion of the microprogram provides the micro-sequences necessary to implement the addressing modes, interrupt and sub-routine processing, etc.


Within this hybrid microprogram architecture, both the static microprogram portion and the variable microprogram have control fields which can exert control which registers are read and written. The M65C02A Write Select Generator generates the correct register write select signal based on these two microprogram control fields. Generally, priority is given to the variable microprogram register select field, but when the field code is the generic register write enable (RegWE), the register write select field of the fixed microprogram is used to generate the correct register write select signal.

The register override prefixes, *osx/oax/oay*, are also processed by the module. The control field in the microprogram is mapped as needed to override the default register write select in the microprogram.

*(Note: the FORTH VM registers, IP and W, are controlled by the microprogram sequences for the dedicated instructions which support the FORTH VM. Similarly, the microprogram directly controls the writing of the PC, the MAR, and the two 8-bit registers which comprise the memory operand register M.)*

### 2.3.5.6. Register A

The A register is the accumulator for the 6502/65C02 processors. The M65C02A core implements the A register as a modified three-level push-down register stack. The register stack is not automatically pushed or popped.

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 42 OF 56                     |                 |

The programmer must explicitly push the top element of the stack,  $A_{TOS}$ , into the next-on-stack (NOS) register,  $A_{NOS}$ . When the stack is pushed down by the programmer, the  $A_{NOS}$  register is automatically transferred to the bottom-of-stack (BOS) register,  $A_{BOS}$ ; the value of the  $A_{BOS}$  register is lost when this occurs. Similarly, the programmer must explicitly pop the top element of stack. When the stack is popped,  $A_{TOS}$  is transferred to  $A_{BOS}$ ,  $A_{BOS}$  is transferred to  $A_{NOS}$ , and  $A_{NOS}$  is transferred to  $A_{TOS}$ . In other words,  $A_{TOS}$  is not lost, but it is stored in the bottom element of the register stack.

The enhanced M65C02A instruction set does not include an instruction for pushing the register stacks. Instead, the TOS duplication instruction, *dup*, is used for this purpose. Similarly, the enhanced M65C02A instruction set does not include an instruction to destructively pop the register stack. Instead, the register stack rotation instruction, *rot*, is used to rearrange the register stack in the desired manner.

The register stack swap instruction, *swp*, allows the TOS and NOS registers to be swapped. With these three instructions, the register stacks can be used for a number of operations that would otherwise require storing and loading the accumulator to memory or to the system or auxiliary stacks. The accumulator register stack enables the programmer to maintain an evaluation stack on-chip, and greatly reduce the number of memory cycles needed to evaluate complex equations.


All register stack register transfers are performed as 16-bit operations. Unlike 6502/65C02 instructions, register stack transfers do not affect the ALU flags in P.

In addition, to the 16-bit register stack operations described above, the accumulator register stack includes support for byte swapping and right rotation of the nibbles in  $A_{TOS}$ . When prefixed by *ind* (or *isz*), the *swp* instruction swaps the bytes in the 16-bit  $A_{TOS}$  register. Similarly, when prefixed by *ind* (or *isz*), the *rot* instruction rotates the nibbles of  $A_{TOS}$  to the right four bit positions. The least significant nibble wraps around to the most significant nibble like the LSB to the MSB in the *ror* instruction. Unlike the shift/rotate instructions, these operations do not affect the ALU flags.

### 2.3.5.7. Register X

The X register is an index register in the 6502/65C02 processor architecture. In 6502/65C02 processors, the X register is generally described as the pre-index register. For indexed zero page direct and indexed absolute addressing modes, the effective address formed by pre-indexing by X is indistinguishable from post-indexing. However, for indirect addressing modes, pre-indexing by X is very different from post-indexing by X.

In the M65C02A core, the X register is much more flexible than in the 6502/65C02. Beyond its use as an index register, it can be used as an accumulator, as an auxiliary stack pointer, as a source data pointer, or as a base address register. In addition, like the A register, the X register is implemented as a modified three level push-down register stack. The register stack is not automatically pushed or popped.

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 43 OF 56                     |                 |

As an index register, the  $X_{TOS}$  can be used in the traditional pre-index role. However, when used as an index register in an indirect addressing mode formed using the *ind* (or *isz*) prefix instruction,  $X_{TOS}$  functions as a post-index register like the Y register. This behavior is due to the implementation of the effects of the *ind/isz* instructions in the microprogram of the M65C02A core.  $X_{TOS}$  retains its traditional pre-index role for all 6502/65C02 indirect addressing modes instructions which are not prefixed with *ind/isz*. When 6502/65C02 indirect addressing modes pre-indexed by X are prefixed by *ind/isz*, the double indirection is performed first without indexing and then indexing is applied. This results in  $X_{TOS}$  being used as a post-index register.

When prefixed by the *oax* prefix instruction, any ALU operation that is written to the accumulator,  $A_{TOS}$ , is redirected to  $X_{TOS}$ , and the address indexing function of  $X_{TOS}$  is provided by  $A_{TOS}$  instead. When the X register stack is properly used, the accumulator functionality of  $X_{TOS}$  is expected to improve the performance of the M65C02A with respect to addressing complex data structures. For example, all of the shift and rotate instructions can now be applied to  $X_{TOS}$  with only a single cycle penalty.

The  $X_{TOS}$  register also incorporates all of the increment/decrement logic required to implement a 6502/65C02-compatible auxiliary stack,  $S_X$ . FORTH VM instruction stack accesses default to  $S_X$ , which serves as the FORTH VM Return Stack (RS). With respect to FORTH VM stack access instructions, the *osx* prefix is used to change the default stack for instructions such as *phi/pli*, *phw/plw*, and *ent* from the auxiliary stack to the system stack, or FORTH VM Parameter Stack (PS).

The same increment/decrement logic used to support a 6502/65C02-compatible auxiliary stack is also used when  $X_{TOS}$  acts as the source address pointer for the M65C02A core's data move instruction, *mov*. The mode byte of the *mov* instruction, in addition to controlling if block or single moves are performed, also controls whether the source pointer is incremented, decremented, or held after each byte. (**Note:** *the mode byte of this instruction allows the instruction to function as a non-interruptable block move, *mvb*, or as interruptable single byte move instruction, *mvb*. Although two distinct mnemonics can be defined, only a single opcode, 0x42, is used.*)

Finally, the  $X_{TOS}$  is the base pointer register for the base pointer relative and post-indexed (by Y) base pointer relative indirect addressing mode. These base pointer relative addressing modes provide a mechanism by which High Level Languages (HLL) like C and Pascal can be implemented more readily. When coupled with the unlimited stack size capability when the system or auxiliary stacks are not implemented within page 1 or page 0, the base pointer relative addressing modes greatly enhance the operation of a processor based on the M65C02A core.

A marked difference between the 65816 stack pointer relative addressing mode and the M65C02A base pointer relative addressing mode is that the offset for the first element on the stack is at offset 0 for the M65C02A instead of offset 1 as it is for the 65816. In addition, the offset is signed. Positive offsets may be used to access parameters on the stack, and negative offsets may be used to access local variables on the stack. With the *swp x* instruction sequence (*oax swp*),  $X_{TOS}$  and  $X_{NOS}$  can be rapidly interchanged so that a base pointer and an auxiliary stack pointer, or FORTH VM RS pointer (RSP), can be maintained simultaneously and used as required.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 44 OF 56

### 2.3.5.8. Register Y

The Y register is an index register in the 6502/65C02 processor architecture. In 6502/65C02 processors, the Y register is generally described as the post-index register. For indexed zero page direct and indexed absolute addressing modes, the effective address formed by post-indexing by Y is indistinguishable from pre-indexing. However, for indirect addressing modes, post-indexing by Y is very different from pre-indexing by Y.

In the M65C02A core, the Y register is more flexible than in the 6502/65C02. Beyond its use as an index register, it can be used as an accumulator, and as a destination pointer. In addition, like the A register, the Y register is implemented as a modified three level push-down register stack. The register stack is not automatically pushed or popped.

As an index register, the  $Y_{TOS}$  is used in its traditional post-index role. When *ind* (or *isz*) add indirection, the rule that **indirection is performed before indexing** means that  $Y_{TOS}$  always maintains its traditional post-indexing role unlike  $X_{TOS}$  which changes roles.

When prefixed by the *oay* prefix instruction, any ALU operation that is written to the accumulator,  $A_{TOS}$ , is redirected to  $Y_{TOS}$ , and the address indexing function of  $Y_{TOS}$  is provided by  $A_{TOS}$  instead. When the Y register stack is properly used, the accumulator functionality of  $Y_{TOS}$  is expected to improve the performance of the M65C02A with respect to addressing complex data structures. For example, all of the shift and rotate instructions can now be applied to  $Y_{TOS}$  with only a single cycle penalty.


The  $Y_{TOS}$  register also incorporates all of the increment/decrement logic required to implement a 6502/65C02-compatible auxiliary stack,  $S_Y$ . However, the lack of an available opcode to implement an *osy* prefix instruction means the  $Y_{TOS}$  cannot be used as a stack pointer. However, that up-down counter logic is used implement the destination pointer function of the M65C02A core's data move instruction, *mov*. The mode byte of the *mov* instruction, in addition to controlling if block or single moves are performed, also controls whether the destination pointer is incremented, decremented, or held after each byte. (**Note:** *the mode byte of this instruction allows the instruction to function as a non-interruptable block move, *mvb*, or as interruptable single byte move instruction, *mvb*. Although two distinct mnemonics can be defined, only a single opcode, 0x42, is used.*)

### 2.3.5.9. Register P

The P register provides the ALU result flags and the processor mode flags. It is implemented as a 7-bit register. The B flag is implemented as a virtual register instead of a physical register. The P register is not initialized by system reset. This is done to allow P and the program counter to be written to the lowest three bytes of page 1 during reset. See 2.1.5 for more detail on P.

## 3. Addressing Modes

This section will describe the addressing modes supported by the M65C02A core. The M65C02A core supports all fifteen (15) addressing modes of the 6502/65C02 microprocessors:

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 45 OF 56                     |                 |

- (1) Implicit/Accumulator: ***-/A***
- (2) Immediate: ***#imm***
- (3) Zero Page: ***zp***
- (4) Pre-Indexed Zero Page: ***zp,X***
- (5) Post-Indexed Zero Page: ***zp,Y***
- (6) Zero Page Indirect: ***(zp)***
- (7) Pre-Indexed Zero Page Indirect: ***(zp,X)***
- (8) Post-Indexed Zero Page Indirect: ***(zp),Y***
- (9) Relative (8-bit): ***rel8***
- (10) Absolute: ***abs***
- (11) Pre-Indexed Absolute: ***abs,X***
- (12) Post-Indexed Absolute: ***abs,Y***
- (13) Absolute Indirect: ***(abs)***
- (14) Pre-Indexed Absolute Indirect: ***(abs,X)***
- (15) Zero Page Relative: ***zp,rel8***

Four new addressing modes have been included in the instruction set of the M65C02A core:

- (16) Relative (16-bit): ***rel16***
- (17) Base-relative: ***bp,B***
- (18) Base-relative Indirect Post-Indexed: ***(bp,B),Y***
- (19) IP-relative with Auto-increment: ***ip,l++***

The M65C02A core provides a means for modifying the behavior of the existing addressing modes using five prefix instructions: ***ind***, ***isz***, ***osx***, ***oax***, and ***oay***. Without considering the change in index register provided by the register override prefix instructions, ***oax*** and ***oay***, the M65C02A core provides large number of additional addressing modes not supported by the 6502 or the 65C02 processors:

- (20) Zero Page Indirect Post-Indexed (by X): ***(zp),X***
- (21) Zero Page Double Indirect: ***((zp))***
- (22) Zero Page Double Indirect Post-Indexed (by X): ***((zp)),X***
- (23) Zero Page Double Indirect Post-Indexed (by Y): ***((zp)),Y***
- (24) Absolute Indirect Post-Indexed (by X): ***(abs),X***
- (25) Absolute Indirect Post-Indexed (by Y): ***(abs),Y***
- (26) Absolute Double Indirect: ***((abs))***
- (27) Absolute Double Indirect Post-Indexed (by X): ***((abs)),X***
- (28) Base-relative Indirect: ***(bp,B)***
- (29) Base-relative Double Indirect Post-Indexed: ***((bp,B)),Y***
- (30) IP-relative with Auto-increment Indirect: ***(ip,l++)***



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 46 OF 56



If the effects of the *oax* and *oay* register override prefix instructions are considered, an additional number of unique addressing modes are possible:

- (31) Zero Page Post-Indexed (by A): ***zp,A***
- (32) Zero Page Indirect Post-Indexed (by A): ***(zp),A***
- (33) Zero Page Double Indirect Post-Indexed (by A): ***((zp)),A***
- (34) Absolute Post-Indexed (by A): ***abs,A***
- (35) Absolute Indirect Post-Indexed (by A): ***(abs),A***
- (36) Absolute Double Indirect Post-Indexed (by A): ***((abs)),A***

Many of the unique addressing modes of the M65C02A given in the preceding lists are unlikely to be used. The limited applicability of some of these addressing modes means that they are unlikely to be used except in special circumstances, and then only by a programmer writing in assembler.

The M65C02A core does not prohibit the programmer from applying the prefix instructions in inefficient or nonsensical ways. *As implemented by its current microprogram*, the M65C02A core applies the *ind*, or *isz* prefix instructions according to the following rule:

***Indirection by IND is applied before any indexing operation.***

This rule has some practical consequences. Applying *ind/isz* to a 6502/65C02 indexed indirect addressing mode increases the level of indirection, but indirection (single or double) is performed first, and then the indexing is applied. In other words, indexed addressing modes are converted to post-indexed indirect (single or double) addressing modes by *ind/isz*.

For example, if *ind/isz* is applied to the 6502/65C02 pre-indexed zero page indirect addressing mode, ***(zp,X)***, the resulting addressing mode is not pre-indexed zero page double indirect, ***((zp,X))***, but zero page double indirect post-indexed by X, ***((zp)),X***. Similarly, if *ind/isz* is applied to instructions which use the indexed zero page ***(zp,X*** or ***zp,Y)*** or indexed absolute ***(abs,X*** or ***abs,Y)*** addressing modes, the resulting addressing modes are post-indexed zero page indirect and post-indexed absolute indirect addressing modes: ***(zp),X***; ***(zp),Y***; ***(abs),X***; and ***(abs),Y***.

### 3.1. Implicit/Accumulator

The implicit/accumulator addressing mode is used whenever the ALU operation has a single destination register operand and does not require a memory-based operand. In other words, an internal register of the 6502/65C02 is the one operand, and the second operand, if required, is automatically supplied. This addressing mode applies specifically to the following instructions:

*inc A/inx/inx*  
*dec A/dex/dey*  
*asl A/rol A/lsl A/ror A*  
*tax/txa/tay/tya/txs/tsx*



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 47 OF 56



In all of these instructions, the destination register operand is implied by the instruction opcode, and any required second operands, i.e.  $\pm 1$ , are automatically generated internally to the ALU.

For the register transfer instructions, *tax/txa/tay/tya/txs/tsx*, the source register is also implicitly determined. Alternate mnemonics, *ina/dea*, could have been used for the *inc A/dec A* instructions to emphasize implicit addressing of the accumulator. The generally accepted syntax for the shift/rotate instructions, *asl A/rol A/lsl A/ror A*, (which were added to the instruction set by the 65C02,) is used to distinguish the implicit/accumulator addressing mode of these four instructions because they also support a large number of other addressing modes for operating on memory operands.

### 3.1.1. Effect of the *ind/siz/isz* Prefix Instructions

The IND prefix flag, if set using *ind* or *isz*, has no affect on the implicit/accumulator addressing mode. The SIZ prefix flag has an effect on the operation width for implicitly addressed operands. When set by the *siz* or *isz* prefix instructions, the operation performed is 16 bits in width.

(**Note:** unless the SIZ prefix is set by using the *siz* or *isz* prefix instructions, the register transfer instructions default to 8-bit operations. An 8-bit transfer from  $X_{TOS}$  to the system stack pointer,  $S_K$  or  $S_U$ , will always have the effect of setting the upper 8-bits of the system stack pointer to 0x01.)

### 3.1.2. Effect of the *osx/oax/oay* Prefix Instructions

The OSX, OAX, and OAY register override prefix flags have the expected effect on the destination register. Inefficient combinations are allowed rather than trapped as invalid instructions. For example, the instruction sequence *oax inc A* is allowed even though that two byte, two cycle sequence produces the same result as the single byte, single cycle instruction *inx*.

Even seemingly nonsensical combinations are allowed rather than trapped as invalid instructions. For example, the instruction sequence, *oax tax*, defines a transfer of the accumulator to the accumulator, *taa*. Since all stores into registers in the 6502/65C02 architecture result in the N and Z flags being set, that instruction sequence may be considered nonsensical in most cases, but may be useful for testing  $A_{TOS}$  after a register stack operation since register stack operations do not affect the ALU flags.

The *oax* or *oay* prefix instructions convert the accumulator shift/rotate instructions, as expected, into shift/rotate instructions that target the X, or Y registers, respectively:

|                                    |                                |
|------------------------------------|--------------------------------|
| <i>oax asl A/rol A/lsl A/ror A</i> | <i>asl X/rol X/lsl X/ror X</i> |
| <i>oay asl A/rol A/lsl A/ror A</i> | <i>asl Y/rol Y/lsl Y/ror Y</i> |

The *osx* prefix instruction does not convert the system stack pointer into a general purpose accumulator; *osx* only allows the instructions that specifically target X to be used to manipulate S. Thus, applying *osx* to the shift/rotate instructions has no effect: the accumulator remains the



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 48 OF 56

target of these instructions when they are preceded by *osx*. (**Note:** certain FORTH VM instructions (*phi/pli*, *phw/plw*, and *ent*) assume that by default X acts as the stack pointer. Therefore, a more accurate view of the effect of *osx* is to think of it as changing the default stack pointer for an instruction. In this example, *osx phi/pli/phw/plw/ent* causes the system stack pointer to be used instead of the auxiliary stack pointer.)

## 3.2. Immediate [#imm]

The immediate addressing is used to load a value, which follows the instruction, into a register. By convention, the octothorpe symbol, #, is used to indicate that the value or symbol provided are to be treated as an immediate value which follows the instruction opcode. The following are examples of instructions supporting the immediate addressing mode:

```
lda #$FF
ldx #$0180
ldy #47
```

These three examples demonstrate how the immediate addressing mode may be used to load the A<sub>TOS</sub>, X<sub>TOS</sub>, or Y<sub>TOS</sub> registers with 8-bit or 16-bit constants. The first and third examples load 8-bit constants into the A and Y registers. The second example loads a 16-bit constant into the X register. (**Note:** the *siz* or *isz* prefix instruction is expected to be inserted by the programmer/assembler/compiler as part of the *ldx #\$0180* instruction sequence.)

### 3.2.1. Effect of the *ind/siz/isz* Prefix Instructions

The IND prefix flag, if set using *ind* or *isz*, has no effect on the immediate addressing mode. The SIZ prefix flag has the expected effect on the operation width for immediate mode operands. When SIZ is set by the *siz* or *isz* prefix instructions, the immediate mode operand is treated as being 16 bits in width.

### 3.2.2. Effect of the *osx/oax/oay* Prefix Instructions

The OSX, OAX, and OAY register override prefix flags have the expected effect on the destination register. Inefficient combinations are allowed rather than trapped as invalid instructions. For example, the instruction sequence *oax lda #\$FF* is allowed even though that three byte, three cycle sequence produces the same result as the two byte, two cycle instruction *ldx #\$FF*.

(**Note:** the instruction sequence *osx ldx #\$F000* (or *lds #\$F000*) has the effect of directly loading the system stack pointer, S<sub>K</sub> or S<sub>U</sub>, with a 16-bit constant. The appropriate prefix instruction sequence (*osx siz*, *osx isz*, *siz osx*, or *isz osx*) is expected to be generated by the programmer/assembler/compiler for the *lds #\$F000* instruction sequence.)



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 49 OF 56

### 3.3. Zero Page Direct [zp]

The zero page direct addressing mode is common to the 6502, 65C02 and the M65C02A. It provides a way to refer to address page zero locations in a faster manner, using less address bytes.

The Effective Address (EA) of the zero page direct addressing mode is given as:

$$EA[15:0] = \{0x00, zp\}$$

where zp is the byte following the instruction opcode.

Reads from memory are deposited in the lower byte of the memory operand register:

$$M[7:0] \leftarrow \text{Mem}[EA]$$

The upper half of the memory operand register, M[15:8], is zeroed. The memory operand register is written to the destination register during the following memory read cycle, i.e. the fetch cycle for the next instruction.

The output bus of the M65C02A core provides the byte of data to be written to memory

$$\text{Mem}[EA] = DO$$

#### 3.3.1. Effect of the *ind/siz/isz* Prefix Instructions

If the IND flag is asserted, any instructions using the zero page direct addressing mode will automatically perform an indirection operation using the zero page address supplied. The low byte of the pointer in zero page will be read from the designated location. The high byte of the pointer will be read from the next location modulo 256. Thus, if the zero page address is 0xFF, the high byte of the pointer will be read from zero page address 0x00 rather than page 1 address 0x0100. The effective address of the data pointer is given as:

$$EA = \{\text{Mem}[\{0x00, (zp + 1)\}], \text{Mem}[\{0x00, zp\}]\}$$

If the SIZ flag is asserted, the operation of any instructions using the zero page direct addressing mode will be promoted from 8 bits to 16 bits. The least significant byte of the operand will be read from, or written to, the designated zero page location. The high byte will be read from or written to the next sequential location. The next sequential location address computation is performed modulo 256, so it wraps on the page boundary. The effective addresses of each byte of the 16-bit operand are given as:

$$\begin{aligned} EA[0] &= \{0x00, zp\} \\ EA[1] &= \{0x00, (zp + 1)\} \end{aligned}$$

If both IND and SIZ are both asserted, the indirection required is performed first and then the operand is read from or written to memory. Modulo 256 address arithmetic is used for fetching



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 50 OF 56

the pointer, but not for the operand read/write cycles. The effective address for the pointer to the 16-bit operand is given as:

$$EA = \{Mem[\{0x00, (zp + 1)\}], Mem[\{0x00, zp\}]\}$$

For reads, the low byte of the operand is read first and the high byte of the operand is read second from the next sequential address modulo 65536:

$$M[7:0] = Mem[EA]$$

$$M[15:8] = Mem[EA + 1]$$

The memory operand register is transferred to an internal register, if required, during the next memory cycle.

For writes, the low byte is written first, and then the high byte:

$$Mem[EA] = DO \text{ (low)}$$

$$Mem[EA + 1] = DO \text{ (high)}$$

The microprogram controls which byte of the result is output on the data bus of the core for each 8-bit write cycle.

A number of useful 6502/65C02 instructions support a limited number of addressing modes. Using these three prefix instructions, the M65C02A provides an enhancement that should improve the programmer's ability to better use the following instructions.

**Table 6: Notable 6502/65C02 zp Instructions Enhanced Using IND/SIZ.**

| Mnemonic | Description   | IND | SIZ |
|----------|---|-----|-----|
| BIT      | Test memory with mask in A <sub>TOS</sub>           | Y   | Y   |
| TRB      | Test and reset memory with mask in A <sub>TOS</sub> | Y   | Y   |
| TSB      | Test and set memory with mask in A <sub>TOS</sub>   | Y   | Y   |
| RMBx     | Reset (zero page) memory bit x                      | Y   | N   |
| SMBx     | Set (zero page) memory bit x                        | Y   | N   |
| CPX      | Compare X <sub>TOS</sub> to memory                  | Y   | Y   |
| CPY      | Compare Y <sub>TOS</sub> to memory                  | Y   | Y   |

### 3.3.2. Effect of the *osx/oax/oay* Prefix Instructions

The OSX, OAX, and OAY register override prefix flags have the expected effect on the destination register. Inefficient combinations are allowed rather than trapped as invalid instructions. These prefix instructions can be combined with the indirection and size prefix instructions.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 51 OF 56

### 3.4. Pre-Indexed Zero Page Direct [zp,X]

The pre-indexed zero page direct addressing mode is common to the 6502, 65C02 and the M65C02A. It provides a way to refer to address page zero locations in a faster manner, using less address bytes using the  $X_{TOS}$  index register to offset the effective address.

The Effective Address (EA) of the pre-indexed zero page direct addressing mode is given as:

$$EA[15:0] = (X_{TOS}[15:9] == 0) ? \{0x00, X_{TOS} + \{0x00, zp\}\} : X_{TOS} + \{0x00, zp\}$$

where zp is the byte following the instruction opcode. (**Note:** as discussed elsewhere, the upper byte of the  $X_{TOS}$  register will determine if modulo 256 address arithmetic is performed when determining the effective address. In a 6502/65C02 processor with an 8-bit X register, the pre-indexed zero page direct addressing mode is always performed using modulo 256 address arithmetic. In the M65C02A core, if  $X_{TOS}$  holds an address in page 0 or page 1, then the address arithmetic is performed modulo 256. If  $X_{TOS}$  holds an address **NOT** in page 0 or page 1, then the address arithmetic is not performed modulo 256. Thus, if the upper 7 bits of  $X_{TOS}$  are not all 0, the addressing mode is base plus off without modulo 256 arithmetic.)

Reads from memory are deposited in the lower byte of the memory operand register:

$$M[7:0] \leftarrow Mem[EA]$$

The upper half of the memory operand register,  $M[15:8]$ , is zeroed. The memory operand register is written to the destination register during the following memory read cycle, i.e. the fetch cycle for the next instruction.

The output bus of the M65C02A core provides the byte of data to be written to memory

$$Mem[EA] = DO$$

### 3.5. Post-Indexed Zero Page Direct [zp,Y]

The post-indexed zero page direct addressing mode is common to the 6502, 65C02 and the M65C02A. It provides a way to refer to address page zero locations in a faster manner, using less address bytes using the  $Y_{TOS}$  index register to offset the effective address.

The Effective Address (EA) of the post-indexed zero page direct addressing mode is given as:

$$EA[15:0] = (Y_{TOS}[15:9] == 0) ? \{0x00, Y_{TOS} + \{0x00, zp\}\} : Y_{TOS} + \{0x00, zp\}$$

where zp is the byte following the instruction opcode. (**Note:** as discussed elsewhere, the upper byte of the  $Y_{TOS}$  register will determine if modulo 256 address arithmetic is performed when determining the effective address. In a 6502/65C02 processor with an 8-bit Y register, the post-indexed zero page direct addressing mode is always performed using modulo 256 address arithmetic. In the M65C02A core, if  $Y_{TOS}$  holds an address in page 0 or page 1, then the address arithmetic is performed modulo 256. If  $Y_{TOS}$  holds an address **NOT** in page 0 or page 1, then the



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 52 OF 56

*address arithmetic is not performed modulo 256. Thus, if the upper 7 bits of  $Y_{TOS}$  are not all 0, the addressing mode is base plus off without modulo 256 arithmetic.)*

Reads from memory are deposited in the lower byte of the memory operand register:

$$M[7:0] \leq \text{Mem}[\text{EA}]$$

The upper half of the memory operand register,  $M[15:8]$ , is zeroed. The memory operand register is written to the destination register during the following memory read cycle, i.e. the fetch cycle for the next instruction.

The output bus of the M65C02A core provides the byte of data to be written to memory

$$\text{Mem}[\text{EA}] = \text{DO}$$

### 3.6. Zero Page Indirect [(zp)]

The zero page indirect addressing mode was introduced by the 65C02. It provides a way to refer to a pointer located in a page zero location to access a location anywhere in memory. Using a zero page saves one byte and one cycle when accessing memory.

The Effective Address (EA) of the zero page indirect addressing mode is given as:

$$\begin{aligned} \text{EA}[7:0] &= \text{Mem}[\{0x00, \text{zp}\}] \\ \text{EA}[15:8] &= \text{Mem}[\{0x00, (\text{zp} + 1)\}] \end{aligned}$$

where zp is the byte following the instruction opcode. The low byte of the pointer is read from the page zero location defined by zp. The high byte of the pointer is read from the next higher location in page zero modulo 256, i.e.  $(\text{zp} + 1) \% 256$ .

Reads from memory are deposited in the lower byte of the memory operand register:


$$M[7:0] \leq \text{Mem}[\text{EA}]$$

The upper half of the memory operand register,  $M[15:8]$ , is zeroed. The memory operand register is written to the destination register during the following memory read cycle, i.e. the fetch cycle for the next instruction.

The output bus of the M65C02A core provides the byte of data to be written to memory

$$\text{Mem}[\text{EA}] = \text{DO}$$

### 3.7. Pre-Indexed Zero Page Indirect [(zp,X)]

|  |  |                  |           |                                    |                 |
|--|--|------------------|-----------|------------------------------------|-----------------|
|  | THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS. | SIZE<br><b>A</b> | CAGE CODE | DRAWING NUMBER<br><b>1004-0900</b> | REV<br><b>-</b> |
|  |  | SCALE: NONE      |           | SHEET 53 OF 56                     |                 |



**3.8. Post-Indexed Zero Page Indirect [(zp),Y]**

**3.9. Relative [rel8]**

**3.10. Absolute [abs]**

**3.11. Pre-Indexed Absolute [abs,X]**

**3.12. Post-Indexed Absolute [abs,Y]**

**3.13. Absolute Indirect [(abs)]**

**3.14. Pre-Indexed Absolute Indirect [(abs,X)]**

**3.15. Zero Page Relative [zp,rel8]**

**3.16. Relative [rel16]**

**3.17. Base Pointer Relative [bp,B]**

**3.18. Post-Indexed Base Pointer Relative Indirect [(bp,B),Y]**



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

**-**

SCALE: NONE

SHEET 54 OF 56

### 3.19. IP-relative with Auto-increment [ip, l++]

## 4. M65C02A Instruction Set

This section describes all of the instructions in the instruction set of the M65C02A. The operation of the instructions is described. Each instruction description also includes a table that defines the associated opcodes, the addressing modes supported by each instruction, and the effects of prefix instructions.

### 4.1. Accumulator and Memory Instructions

#### 4.1.1. Loads, Stores, and Transfers

#### 4.1.2. Logical Operations

#### 4.1.3. Shift and Rotates

#### 4.1.4. Arithmetic Operations

### 4.2. Stack Instructions

### 4.3. Program Control Instructions

#### 4.3.1. Branches



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 55 OF 56

## 4.3.2. Jumps

## 4.3.3. Subroutine Calls and Returns

## 4.3.4. Interrupt Handling

## 4.4. Prefix Instructions

## 4.5. Register Stack Instructions

This section describes the instructions that manipulate the register stacks. The following table defines the instructions in this category:

Table 7: Register Stack Instructions.

| Mnemonic | Description      | Opcode | # Cycles | Operation       |                                  |
|----------|------------------|--------|----------|-----------------|----------------------------------|
|          |                  |        |          | Normal          | IND                              |
| DUP      | Duplicate TOS    | 0B     | 1        | {TOS, TOS, NOS} | {TOS, TOS, NOS}                  |
| SWP      | Swap TOS and NOS | 1B     | 1        | {NOS, TOS, BOS} | {{TOS[7:0],TOS[15:8]}, NOS, BOS} |
| ROT      | Rotate Stack     | 2B     | 1        | {NOS, BOS, TOS} | {{TOS[3:0],TOS[15:4]}, NOS, BOS} |

These instructions are all single byte instructions. When operating on the register stack, all of these instructions are single cycle instructions. Two of the instructions, *swp* and *rot*, support the *ind* prefix instruction. As indicated in Table 7, when prefixed by *ind*, the *swp* instruction swaps the high and low bytes of the TOS element, and the *rot* instruction performs a 4-bit rotation to the right of the TOS element.

The register stack manipulation instructions do not affect the ALU flags in P: N, V, Z, C. This feature allows the register stack to be used to support extended length ALU operations. (**Note:** *it is possible using the *oax* and *oay* prefix instructions to generate transfers of the TOS elements of the register stack to the same TOS element. Transfers such as *taa* (*oax tax*), *txx* (*oax txa*), and *tyy* (*oay tya*) will set the N and Z flags.*)

## 4.6. FORTH VM Instructions

The section describes the instructions for the FORTH VM.



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

SCALE: NONE

DRAWING NUMBER

**1004-0900**

REV

-

SHEET 56 OF 56

## 4.7. Other Instructions

This section describes instructions not previously described.

## 5. Boot Loader Listings

## 6. Fig-FORTH 1.0 Listings



THE INFORMATION DISCLOSED IN THIS DOCUMENT IS COPYRIGHTED. ALL RIGHTS RESERVED. FURTHER DISSEMINATION IS PROHIBITED WITHOUT THE INCLUSION OF THIS NOTICE. MICHAEL A. MORRIS.

SIZE  
**A**

CAGE CODE

DRAWING NUMBER

**1004-0900**

REV

-

SCALE: NONE

SHEET 57 OF 56