

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 92

Table of Contents

1. GENERAL DESCRIPTION	9
1.1. DESCRIPTION OF A M65C02A-BASED MICROCOMPUTER	9
2. M65C02A CORE	11
2.1. PROGRAMMER'S MODEL	12
2.1.1. COMPATIBILITY VIEW	12
2.1.2. EXTENDED CAPABILITIES VIEW	12
2.1.3. PREFIX INSTRUCTIONS	15
2.1.3.1. IMPLEMENTATION OF PREFIX INSTRUCTIONS IN M65C02A CORE	16
2.1.3.2. REGISTER OVERRIDE PREFIX INSTRUCTIONS	16
2.1.3.3. OPERATION/OPERAND OVERRIDE PREFIX INSTRUCTIONS	16
2.1.4. ACCUMULATORS (A, X, Y)	16
2.1.5. INDEX REGISTERS (X, Y, A)	18
2.1.6. STACK POINTERS (S _K , S _U , S _X)	19
2.1.7. PROGRAM COUNTER (PC)	20
2.1.8. PROCESSOR STATUS WORD (P)	21
2.1.8.1. ALU STATUS FLAGS	21
2.1.8.1.1. C FLAG – BIT 0	22
2.1.8.1.2. Z FLAG – BIT 1	22
2.1.8.1.3. V FLAG – BIT 6	22
2.1.8.1.4. N FLAG – BIT 7	22
2.1.8.2. PROCESSOR MODE FLAGS	23
2.1.8.2.1. I FLAG – BIT 2	23
2.1.8.2.2. D FLAG – BIT 3	23
2.1.8.2.3. B FLAG – BIT 4	23
2.1.8.2.4. M FLAG – BIT 5	24
2.1.9. VIRTUAL MACHINE SUPPORT REGISTERS	24
2.1.9.1. VM INTERPRETER POINTER (IP)	24
2.1.9.2. VM WORKING REGISTER (W)	25
2.1.10. RESTRICTIONS	25
2.1.11. SUMMARY OF THE M65C02A CORE'S FEATURES/CAPABILITIES	26
2.2. M65C02A CORE PORTS	27
2.2.1. SYSTEM INTERFACE	28
2.2.1.1. RST : INPUT	28
2.2.1.2. CLK : INPUT	28
2.2.2. INTERRUPT HANDLER INTERFACE	29
2.2.2.1. IRQ_MSK : OUTPUT	29
2.2.2.2. INT : INPUT	29
2.2.2.3. LE_INT : OUTPUT	29
2.2.2.4. VECTOR : INPUT	30
2.2.2.5. VP : OUTPUT	30
2.2.2.6. XIRQ : INPUT	30
2.2.3. SET OVERFLOW FLAG INTERFACE	30
2.2.3.1. SO : INPUT	31
2.2.3.2. SO_CLR : OUTPUT	31
2.2.4. CORE STATUS INTERFACE	31
2.2.4.1. DONE : OUTPUT	31
2.2.4.2. SC : OUTPUT	31
2.2.4.3. MODE : OUTPUT	31
2.2.4.4. RMW : OUTPUT	32



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 92

2.2.5. MEMORY CYCLE LENGTH CONTROL INTERFACE	32
2.2.5.1. WAIT : INPUT.....	32
2.2.5.2. RDY : OUTPUT.....	32
2.2.6. MEMORY INTERFACE	33
2.2.6.1. IO_OP[1:0] : OUTPUT	33
2.2.6.2. AO[15:0] : OUTPUT	33
2.2.6.3. DI[7:0] : OUTPUT.....	33
2.2.6.4. DO[7:0] : OUTPUT	33
2.2.7. CO-PROCESSOR INTERFACE	33
2.2.8. CORE INTERNAL STATE INTERFACE	34
2.2.8.1. X[15:0] : OUTPUT	34
2.2.8.2. Y[15:0] : OUTPUT	34
2.2.8.3. A[15:0] : OUTPUT	34
2.2.8.4. IP[15:0] : OUTPUT	34
2.2.8.5. W[15:0] : OUTPUT	34
2.2.8.6. S[15:0] : OUTPUT	34
2.2.8.7. P[7:0] : OUTPUT	35
2.2.8.8. M[15:0] : OUTPUT	35
2.2.8.9. IR[7:0] : OUTPUT.....	35
2.2.9. PREFIX INSTRUCTION FLAG INTERFACE.....	35
2.2.9.1. IND : OUTPUT	36
2.2.9.2. SIZ : OUTPUT.....	36
2.2.9.3. OAX : OUTPUT	36
2.2.9.4. OAY : OUTPUT.....	37
2.2.9.5. OSX : OUTPUT	37
2.3. M65C02A CORE COMPONENTS.....	38
2.3.1. M65C02A_CORE MODULE – CORE TOP LEVEL MODULE	38
2.3.2. M65C02A_MPC MODULE – MICROPROGRAM CONTROLLER (MPC)	40
2.3.3. M65C02A_ADDRGEN MODULE – ADDRESS GENERATOR.....	41
2.3.4. M65C02A_FORTHVM MODULE – FORTH VIRTUAL MACHINE	42
2.3.5. M65C02A_ALUV2 MODULE – ARITHMETIC AND LOGIC UNIT (ALU).....	42
2.3.5.1. M65C02A_LST MODULE – LOAD/STORE/TRANSFER UNIT (LST).....	43
2.3.5.2. M65C02A_LU MODULE – LOGIC UNIT (LU)	44
2.3.5.3. M65C02A_SU MODULE – SHIFT/ROTATE UNIT (SU).....	44
2.3.5.4. M65C02A_AU MODULE – ARITHMETIC UNIT (AU).....	44
2.3.5.5. M65C02A_WRSSEL MODULE – WRITE SELECT GENERATOR	45
2.3.5.6. REGISTER A	45
2.3.5.7. REGISTER X	46
2.3.5.8. REGISTER Y	47
2.3.5.9. REGISTER P	48
3. ADDRESSING MODES	49
3.1. IMPLICIT/ACCUMULATOR	50
3.1.1. EFFECT OF THE <i>IND/SIZ/ISZ</i> PREFIX INSTRUCTIONS	51
3.1.2. EFFECT OF THE <i>OSX/OAX/OAY</i> PREFIX INSTRUCTIONS	51
3.2. IMMEDIATE [#IMM]	52
3.2.1. EFFECT OF THE <i>IND/SIZ/ISZ</i> PREFIX INSTRUCTIONS	52
3.2.2. EFFECT OF THE <i>OSX/OAX/OAY</i> PREFIX INSTRUCTIONS	52
3.3. ZERO PAGE DIRECT [ZP].....	53
3.3.1. EFFECT OF THE <i>IND/SIZ/ISZ</i> PREFIX INSTRUCTIONS	53
3.3.2. EFFECT OF THE <i>OSX/OAX/OAY</i> PREFIX INSTRUCTIONS	55



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 92

3.4.	PRE-INDEXED ZERO PAGE DIRECT [ZP,X]	56
3.4.1.	EFFECT OF THE <i>IND/SIZ/ISZ</i> PREFIX INSTRUCTIONS	57
3.4.2.	EFFECT OF THE <i>OSX/OAX/OAY</i> PREFIX INSTRUCTIONS	59
3.5.	POST-INDEXED ZERO PAGE DIRECT [ZP,Y]	60
3.5.1.	EFFECT OF THE <i>IND/SIZ/ISZ</i> PREFIX INSTRUCTIONS	61
3.5.2.	EFFECT OF THE <i>OSX/OAX/OAY</i> PREFIX INSTRUCTIONS	62
3.6.	ZERO PAGE INDIRECT [(ZP)]	63
3.6.1.	EFFECT OF THE <i>IND/SIZ/ISZ</i> PREFIX INSTRUCTIONS	63
3.6.2.	EFFECT OF THE <i>OSX/OAX/OAY</i> PREFIX INSTRUCTIONS	65
3.7.	PRE-INDEXED ZERO PAGE INDIRECT [(ZP,X)]	66
3.8.	POST-INDEXED ZERO PAGE INDIRECT [(ZP),Y]	66
3.9.	RELATIVE [REL8]	66
3.10.	ABSOLUTE [ABS]	66
3.11.	PRE-INDEXED ABSOLUTE [ABS,X]	66
3.12.	POST-INDEXED ABSOLUTE [ABS,Y]	66
3.13.	ABSOLUTE INDIRECT [(ABS)]	66
3.14.	PRE-INDEXED ABSOLUTE INDIRECT [(ABS,X)]	66
3.15.	ZERO PAGE RELATIVE [ZP,REL8]	66
3.16.	RELATIVE [REL16]	66
3.17.	STACK POINTER RELATIVE [SP,S]	67
3.18.	POST-INDEXED STACK POINTER RELATIVE INDIRECT [(SP,S),Y]	67
3.19.	BASE POINTER RELATIVE [BP,B]	67
3.20.	IP-RELATIVE WITH AUTO-INCREMENT [IP,I++]	67
4.	M65C02A INSTRUCTION SET	68
4.1.	M65C02A OPCODE TABLES	68
4.2.	PREFIX INSTRUCTIONS	72
4.3.	REGISTER STACK INSTRUCTIONS	76
4.4.	FORTH VM INSTRUCTIONS	77
4.4.1.	OPERATION OF <i>NXT</i> INSTRUCTION	77
4.4.2.	OPERATION OF <i>ENT</i> INSTRUCTION	78
4.4.3.	OTHER COMMON FORTH PRIMITIVES	78
4.5.	STACK INSTRUCTIONS	80
4.6.	OTHER M65C02A-UNIQUE INSTRUCTIONS	80
4.7.	ACCUMULATOR AND MEMORY INSTRUCTIONS	80
4.7.1.	LOADS, STORES, AND TRANSFERS	80
4.7.2.	LOGICAL OPERATIONS	80
4.7.3.	SHIFT AND ROTATES	80
4.7.4.	ARITHMETIC OPERATIONS	80
4.8.	PROGRAM CONTROL INSTRUCTIONS	80
4.8.1.	BRANCHES	80
4.8.2.	JUMPS	80
4.8.3.	SUBROUTINE CALLS AND RETURNS	81
4.8.4.	TRAPS AND INTERRUPT HANDLING	81
5.	BOOT LOADER LISTINGS	82
6.	FORTH VM STUDY	83
6.1.	"CLASSIC" FORTH VM REGISTERS	83
6.2.	GENERAL IMPLEMENTATION APPROACHES FOR FORTH VMS	83



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 92

6.3. BASIC STRUCTURE OF A FORTH WORD	85
6.4. USAGE FREQUENCY OF FORTH WORDS	86
6.5. OPERATIONS OF THE FORTH VM INNER INTERPRETER	87
6.6. MAPPING FORTH VM TO THE M65C02A CORE	88
6.6.1. ADDITIONAL INSTRUCTIONS FOR SUPPORTING FORTH	89
6.7. SUMMARY.....	92
7. FIG-FORTH 1.0 LISTINGS	93



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 92

Table of Figures

Figure 1: Block Diagram of Microcomputer Using the M65C02A Core.....	10
Figure 2: M65C02A Core Block Diagram.....	11
Figure 3: M65C02A Compatibility View Programmer's Model.	12
Figure 4: M65C02A Extended Capabilities View Programmer's Model.	13

List of Tables

Table 1: M65C02A Core Instruction Mode Output Definition.....	32
Table 2: M65C02A Core IO_Op[1:0] Output Encoding.....	33
Table 3: M65C02A Core 16-bit Default Operation Size Instructions.....	36
Table 4: M65C02A Core <i>osx</i> Prefix Instruction Effects.....	37
Table 5: M65C02A Core Modules.....	38
Table 6: Effect of <i>ind</i> 6502/65C02/M65C02A zp direct Instructions.....	54
Table 7: Effect of <i>siz</i> 6502/65C02/M65C02A zp direct Instructions.....	54
Table 8: Effect of <i>isz</i> 6502/65C02/M65C02A zp direct Instructions.....	55
Table 9: Effect of <i>osx</i> on 6502/65C02 zp direct instructions.....	55
Table 10: Effect of <i>oax</i> on 6502/65C02 zp direct instructions.....	56
Table 11: Effect of <i>oay</i> on 6502/65C02 zp direct instructions.....	56
Table 12: Effect of <i>ind</i> on 6502/65C02 pre-indexed (by X) zp direct instructions.....	58
Table 13: Effect of <i>siz</i> on 6502/65C02 pre-indexed (by X) zp direct instructions.....	58
Table 14: Effect of <i>isz</i> on 6502/65C02 pre-indexed (by X) zp direct instructions.....	58
Table 15: Effect of <i>osx</i> on 6502/65C02 pre-indexed (by X) zp direct instructions.....	59
Table 16: Effect of <i>oax</i> on 6502/65C02 pre-indexed (by X) zp direct instructions.....	60
Table 17: Effect of <i>oay</i> on 6502/65C02 pre-indexed (by X) zp direct instructions.....	60
Table 18: Effect of <i>ind</i> on 6502/65C02 post-indexed (by Y) zp direct instructions.....	62
Table 19: Effect of <i>siz</i> on 6502/65C02 post-indexed (by Y) zp direct instructions.....	62
Table 20: Effect of <i>isz</i> on 6502/65C02 post-indexed (by Y) zp direct instructions.....	62
Table 21: Effect of <i>osx</i> on 6502/65C02 post-indexed (by Y) zp direct instructions.....	62
Table 22: Effect of <i>oax</i> on 6502/65C02 post-indexed (by Y) zp direct instructions.....	63
Table 23: Effect of <i>oay</i> on 6502/65C02 post-indexed (by Y) zp direct instructions.....	63
Table 24: Effect of <i>ind</i> on 6502/65C02/M65C02A zp indirect Instructions.....	64
Table 25: Effect of <i>siz</i> on 6502/65C02/M65C02A zp indirect Instructions.....	65
Table 26: Effect of <i>isz</i> on 6502/65C02/M65C02A zp indirect Instructions.....	65
Table 27: Effect of <i>oax</i> on 6502/65C02 zp indirect instructions.....	65
Table 28: Effect of <i>oay</i> on 6502/65C02 zp indirect instructions.....	65
Table 29: Legend for M65C02A Instruction Set Tables.....	69
Table 30: Columns 0 – 7 HLL-Optimized M65C02A Opcode Table.....	70
Table 31: Columns 8 – 15 HLL-Optimized M65C02A Opcode Table.....	70
Table 32: Columns 0 – 7 W65C02S-Compatible M65C02A Opcode Table.....	71
Table 33: Columns 8 – 15 W65C02S-Compatible M65C02A Opcode Table.....	71
Table 34: M65C02A Prefix Instructions.....	72
Table 35: Effects of <i>ind/siz/isz</i> Prefix Instructions on Implicit/Accumulator Instructions.....	75
Table 36: Effects of <i>ind/siz/isz</i> on Compare and Branch Instructions.....	76
Table 37: Register Stack Instructions.....	76
Table 38: Extended Register Stack Instructions – A Register Stack Only.....	77
Table 39: FORTH VM Instructions.....	77



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 92

Table 40: Extended FORTH VM Instructions.....	77
Table 41: Operation of <i>nxt</i> in ITC and DTC Threaded Code FORTH VM.	78
Table 42: Operation of <i>ent</i> in ITC and DTC Threaded Code FORTH VM.	78
Table 43: Examples of Common FORTH Primitives Using M65C02A Instruction Set.....	78
Table 44: Usage Frequency of Common FORTH Words.	86
Table 45: Consolidated List of M65C02A-specific Instructions Supporting FORTH VM.....	91

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 that of the WDC W65C02S. Extensions have been added in such a manner that the instruction set can be easily configured by changing the microprogram without re-synthesizing the core. The M65C02A can provide additional instructions and addressing modes to support the following features:

- Two operating modes: Kernel (default) and User modes;
- Selectable Operation Size: 8-bit (default) or 16-bit;
- Selectable Indirection: single and double indirection;
- Three accumulators: A, X, and Y;
- Three index registers: X, Y, and A;
- Three stacks: S_K , S_U , and S_X ;
- Multi-level (3 level) register stacks: A, X, and Y;
- Signed/unsigned 16-bit comparison and branch instructions with 8-bit and/16-bit relative displacements;
- FORTH VM functional supporting Indirect and Direct Threaded Code;
- Base Pointer relative Addressing: bp,B and (bp,B)
- Stack Pointer relative Addressing: sp,S, (sp,S), and (sp,S),Y
- IP-relative (with auto-increment) addressing mode instructions;
- 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:

- Branches with fixed cycle count whether branch condition is true or not;
- 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 require a high-performance, compact 8/16-bit soft-core microprocessor.

1.1. Description of a M65C02A-based 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 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).



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 92

A block diagram of a microcomputer based on the M65C02A core 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

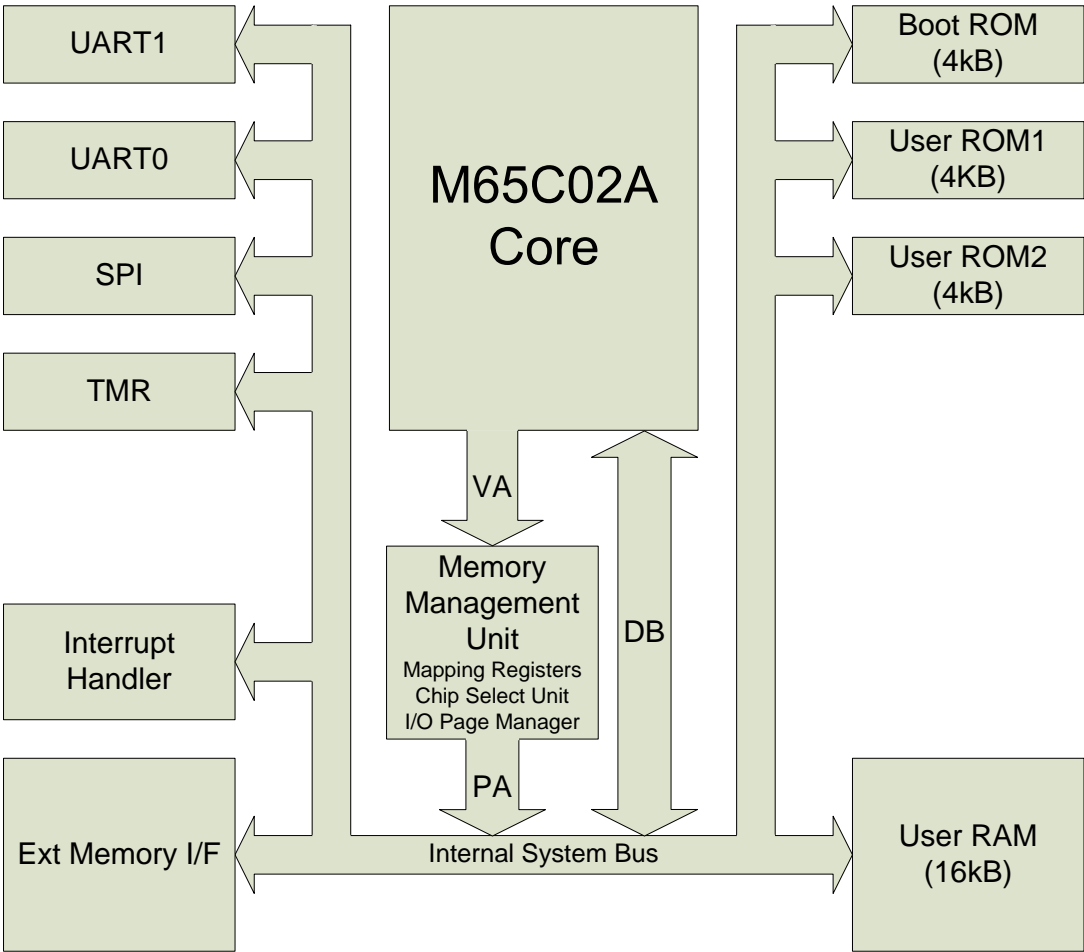



Figure 1: Block Diagram of Microcomputer Using the M65C02A Core.

	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	CAGE CODE	DRAWING NUMBER	REV
		A		1004-0900	-
		SCALE: NONE		SHEET 10 OF 92	

2. M65C02A Core

The M65C02A core is 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 instruction set includes the WDC W65C816's *wai* (WAlt) and *stp* (StoP) instructions. The 32 Rockwell bit-oriented instructions or another set of HLL-optimized instructions optimized may be included via a microprogram change. A block diagram of the M65C02A core is shown in Figure 2.

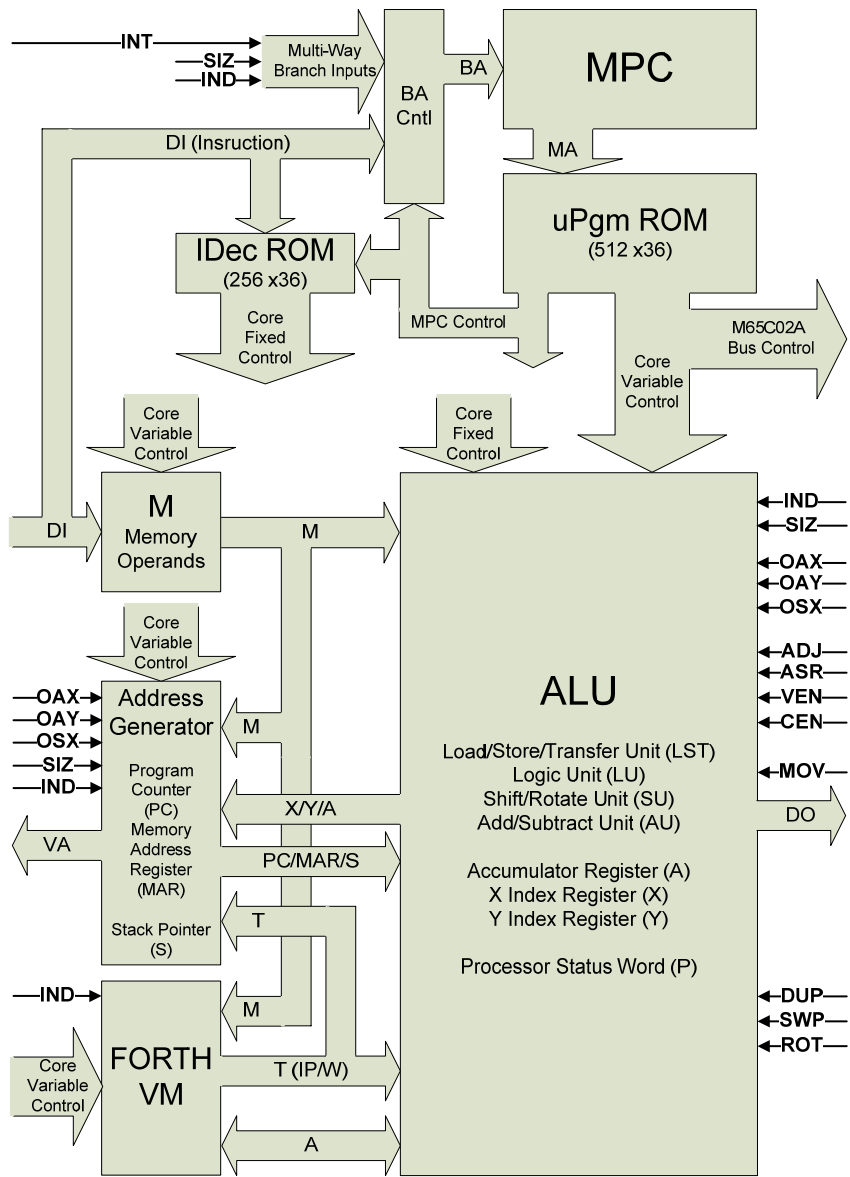


Figure 2: M65C02A Core Block Diagram.

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 genera-



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 11 OF 92

tion 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 all dead memory cycles for base instruction set. With the exception of three non-RMW enhanced/extended instructions (*phr rel16*; *pul dp*; *pul abs*), the M65C02A has no dead memory cycles

2.1. 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.

2.1.1. Compatibility View

The compatibility view 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 3 provides the compatibility view programmer’s model for the M65C02A.

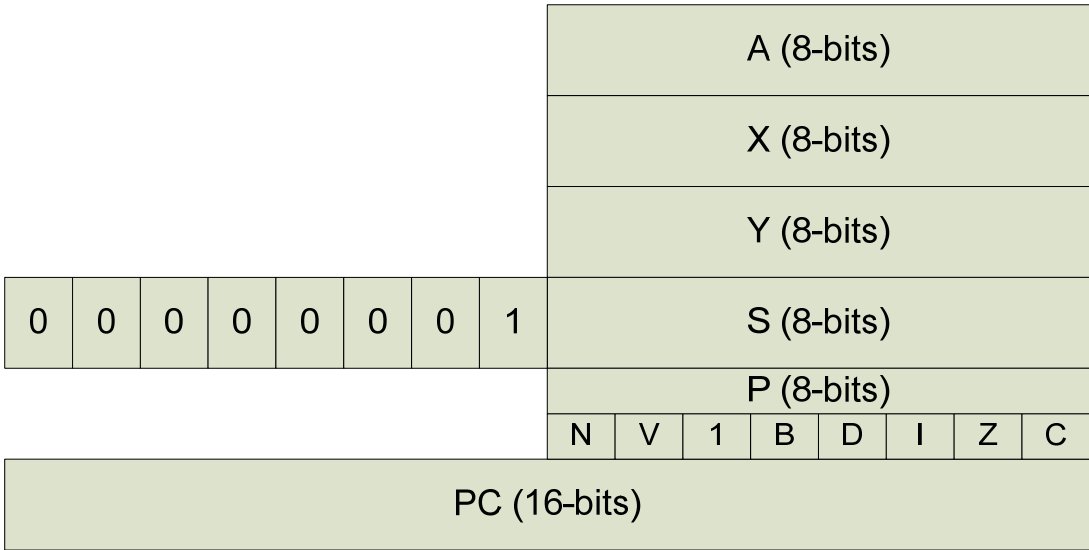


Figure 3: M65C02A Compatibility View Programmer’s Model.

2.1.2. Extended Capabilities View

The extended capabilities programmer's model provides access to 16-bit ALU operations, three 8/16-bit accumulators/index registers (A, X, and Y are implemented as 16-bit, three deep, push down register stacks), three 8/16-bit hardware stack pointers, and a 16-bit FORTH Virtual Machine (VM) core. Figure 4 shows the programmer's model for the extended capabilities view of the M65C02A core.

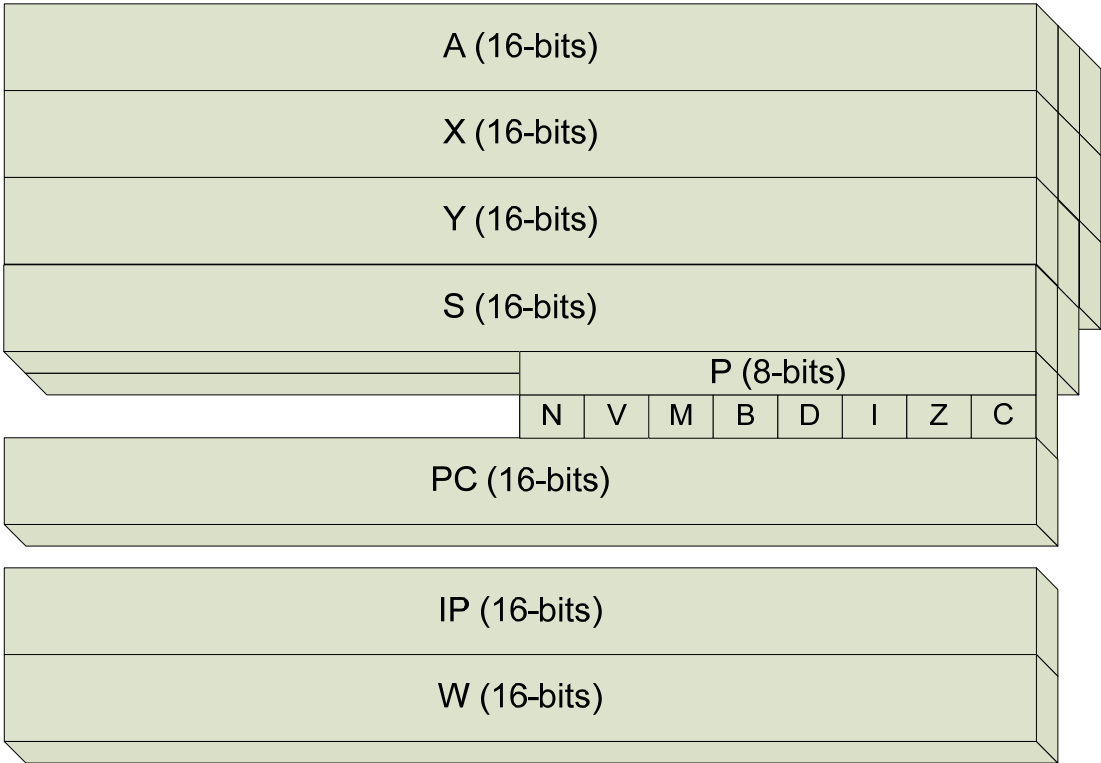


Figure 4: M65C02A Extended Capabilities View Programmer's Model.

The three basic registers (A, X, and Y) have been extended from 8 bits to 16 bits, and converted into three deep push down register stacks.

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 *rti* 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 M65C02A supports threaded-code interpreters like the FORTH Virtual Machine (VM). 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.)

As previously discussed, the base instruction set of the M65C02A core is essentially that of the WDC W65C02S microprocessor. No mode changes are required to operate the M65C02A core compatibly with the base instruction set. If only the base instruction set is used, then the programmer is using the compatibility view model as shown in Figure 3. 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.

(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.)

The M65C02A core expands on the base instruction set using the unused opcodes in the base instruction set. The complete extended instruction of the M65C02A has four (4) free opcodes. The M65C02A core uses only prefix instructions to implement the enhanced instruction set. 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. **(Note:** If the complete M65C02A instruction set is utilized, the W65C02S *wai* and *stp* instructions are replaced by two prefix instructions that provide improved support for stack-relative addressing mode of the M65C02A core.)

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 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 16-bit registers and/or pointers. Further, 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.

In addition, the 6502/65C02 microprocessors may be categorized as having an accumulator based Arithmetic and Logic Unit (ALU). An accumulator based ALU generally provides only a single register as the implicit source for one ALU operand and as the destination for the ALU result. This characteristic of the 6502/65C02 ALU makes the accumulator a bottleneck, and frequently requires the accumulator to be loaded from and/or stored to memory; additional memory cycles are required for extended precision operations, e.g. address calculations, because 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 14 OF 92

accumulator is only 8 bits in width. 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, the accumulator. Thus, it may be appropriate to categorize the 6502/65C02 processors as having an accumulator-memory ALU.

(**Note:** *Many modern microprocessors do not perform ALU operations directly on operands located in memory. The Intel and AMD x86 microprocessors are the most notable examples of modern processors with the capability of directly 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.*)

2.1.3. Prefix Instructions


To maintain compatibility with the 6502/65C02 processors, the M65C02A core uses prefix instructions to access the advanced features of the core. Some new, enhanced instructions make direct use of the advanced features of the M65C02A core, but the majority of the instruction set requires the use of prefix instructions to take advantage of features such as 16-bit ALU operations, three accumulators, three index registers, added addressing mode indirection, etc.

Prefix instructions must be placed before each base instruction for which an advanced/enhanced feature of the M65C02A is desired. Mode switching instructions enable advanced/enhanced features in a more efficient manner, but modes require the programmer to explicitly enable an advanced/enhanced feature and subsequently, explicitly disable that feature before returning to the 6502/65C02 compatible mode.

For the M65C02A core, the decision was made to support prefix instructions, rather than mode switching instructions, so that the programmer could enable an advanced/enhanced feature and the core would automatically return the 6502/65C02 compatibility on the next instruction. This behavior was considered more important than the code and cycle savings that mode switching offered.

The behavioral control of the prefix instructions only apply to the following non-prefix instruction. This allows the advanced/enhanced capabilities of the M65C02A to be used whenever desired without any setup or tear down required. Furthermore, **it is possible to make the advanced/enhanced features the default condition, and instead use the prefix instructions to access the capabilities of the base instruction set architecture.** In other words, it is possible that in a particular application, the 16-bit capabilities of the registers and ALU are used more often than the base 8-bit capabilities of the ALU. The M65C02A core's default behavior can easily be defaulted to 16-bit operations to obviate the need to include the operation size prefix instruction. However, to access the original 8-bit features, like the Binary Coded Decimal (BCD) arithmetic capabilities of the 6502/65C02 ALU, the operation size prefix instruction would need to be applied.

There are two classes of prefix instructions: (1) register override, and (2) operation/operand override. There are a total of 6 prefix instructions if the W65C02S *wai* and *stp* instructions are

	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 92	

retained, and 8 prefix instructions if the W65C02S *wai* and *stp* instructions are not retained. The following subsection 2.1.3.1 will describe how the M65C02A core implements the prefix instructions, and the last two subsections, 2.1.3.2 and 2.1.3.3, will describe the prefix instructions in the register override and operation/operand override categories, respectively.

2.1.3.1. Implementation of Prefix Instructions in M65C02A Core

2.1.3.2. Register Override Prefix Instructions

There are three register override prefix instructions: *oax*, *oay*, and *osx*. The register override prefix instructions control the accumulator and index registers used during instruction execution.

The *oax* and *oay* prefix instructions exchange the functions of the A_{TOS} and the X_{TOS} or Y_{TOS} registers, respectively. These two prefix instructions are mutually exclusive, and essentially enable the three accumulators and the three index registers of the M65C02A core. When an instruction, which uses an indexed addressing mode, is prefixed by *oax* or *oay*, the accumulator A_{TOS} is used as the index register. If the accumulator is the source/target of one of the operands of the instruction, then either X_{TOS} or Y_{TOS} replaces the accumulator as the target/source of the instruction.

2.1.3.3. Operation/Operand Override Prefix Instructions

2.1.4. 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 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 92

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** (or **lda.w #imm16**), and the instruction length increases by two bytes.

As previously described, the M65C02A core implements the A, X, and Y registers as three deep 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_{TOS}**, **X_{TOS}**, and **Y_{TOS}**. 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 M65C02A core provides three single byte, single cycle instructions that 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) **dup** pushes the TOS down into the register stack: **BOS <= NOS <= TOS**;
- (2) **swp** swaps the TOS and NOS registers of the stack: **TOS <= NOS <= TOS**;
- (3) **rot** implements a circular shift of the register stack: **TOS <= NOS <= BOS <= TOS**.

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

The Accumulator register stack has five operations not available to the X and Y register stacks. (**Note:** these operations are not available for the **X_{TOS}** and **Y_{TOS}** registers because of the additional functions associated with those registers: auxiliary stack pointer and **mov** instruction source pointer (**X_{TOS}**), and **mov** instruction destination pointer (**Y_{TOS}**).):

- (1) Transfer **A_{TOS}** to IP using **ind dup**: **IP <= A_{TOS}**;

	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 17 OF 92	

- (2) Transfer IP to A_{TOS} using *siz dup*: A_{TOS} <= IP;
- (3) Swap A_{TOS} and IP using *isz dup*: {IP, A_{TOS}} <= {A_{TOS}, IP};
- (4) Swap bytes of A_{TOS} using *ind swp*: A_{TOS}[15:0] <= {A_{TOS}[7:0], A_{TOS}[15:8]};
- (5) Bit reverse A_{TOS} using *ind rot*: A_{TOS}[15:0] <= A_{TOS}[0:15];

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

2.1.5. 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 be used an index register when a register override prefix instructions, *oax* and *oay*, is applied 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 an impact on how the indexed addressing modes are viewed. The effective address calculation for indexed addressing modes for the 6502/65C02 microprocessors can best be described by the following equation:

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

where the memory operand address *M* is either 8 bits in length for zero page addresses or 16 bits in length for absolute addresses; *index* is 8-bits in length for 6502/65C02 processors. (**Note:** when *M* is an 8-bit page 0 (or page 1) address, the effective address is calculated mod 256.)

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 now view the effective address as a 16-bit base address, which is found in one of the index registers, plus an offset found in the instruction. 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:

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

where *base* is a 16-bit address found in one of the three index registers (X, Y, or A) and *offset* is the 8/16-bit address operand embedded in the instruction. This formulation provides the programmer much better addressing than the standard 6502/65C02 microprocessor indexed addressing modes. (**Note:** for compatibility with the 6502/65C02, the effective address is calculated mod 256 if the base address is in either page 0 or page 1.)

Thus, 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.



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 18 OF 92

For the M65C02A, all pre-indexed (by X) addressing modes represent base plus offset addressing. What this means is that both zero-page (8-bit) and absolute (16-bit) pre-indexed (by X) addressing modes easily support the stack frame addressing needed to access local variables in C/C++ and Pascal whenever X holds the stack frame base pointer, the value of S on entry to a function/procedure.

2.1.6. 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 its operation will automatically ensure that it is locked to page 1 of memory as expected for 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 on the kernel mode stack S_K . 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.)

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 third stack pointer, S_X . Preceding any stack instruction by the *osx* prefix instruction will override the default stack pointer for the instruction. In most cases, the result will be that S_X is 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 S_X can be more easily manipulated using the full power of the ALU.



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 92

This stack pointer function of X_{TOS} should make it very attractive as a pointer for the parameter stack (PS) or the pointer for the return stack (RS) of 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, for the stacks initialized/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 not locked to page 1 (or page 0), then the standard 6502/65C02 mod 256 behavior for stacks is not imposed and the stack size becomes unlimited, i.e. modulo 65536. (**Note:** *This behavior has a restriction: if the stack grows into page 1, the stack will become locked to page 1. In other words, the mod 256 behavior of the stack address calculations is determined by the upper address bits of the stack pointer. If the upper address bits of the stack pointer are 0x01 (or 0x00), then the stack is locked to page 1 (or page 0).*)

2.1.7. 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.*)

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



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 20 OF 92

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 (1) rather than two (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 (1) 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 completed before the interrupt is accepted, 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.)

2.1.8. 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 3 and Figure 4 define the P register.

2.1.8.1. 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.



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 21 OF 92

2.1.8.1.1. 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*, *sbc*, *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*.

2.1.8.1.2. Z flag – Bit 1

The Z flag indicates that the ALU result is zero.

2.1.8.1.3. 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* instruction is strictly an unsigned comparison, so the V flag is not modified. The M65C02A core allows the 16-bit version of the *cmp/cpx/cpy* instruction 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:

- (1) 8-bit signed offsets: *bgt/bge/blt/ble*, and *blo/bls/bhi/bhs*;
- (2) 16-bit signed offsets: *jgt/jge/jlt/jle*, and *jlo/jls/jhi/jhs*.

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.)

2.1.8.1.4. 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 mi-



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 92

coprocessor, 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.)

2.1.8.2. Processor Mode Flags

The 6502/65C02 microprocessors provide three processor mode flags, and the M65C02A core adds a fourth. The mode flags are I, D, B, and M flags. The M flag is specific to the M65C02A core, while the other three flags are common to 6502/65C02 processors.

2.1.8.2.1. 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.

2.1.8.2.2. 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 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/inb*), 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.)

2.1.8.2.3. B Flag – Bit 4

The B flag, or Break flag, indicates that a *brk* instruction has been executed. Like bit 5 of the 6502/65C02 processor status word, bit 4 is unimplemented and only “exists” in the processor status word pushed onto the stack at the beginning of the *brk* instruction trap process.

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.



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 23 OF 92

Thus, only when P is pushed onto the stack when entering an interrupt service routine for a [maskable/non-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.

2.1.8.2.4. 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. Its only affect on the operation of the M65C02A core itself is to select the stack pointer: S_K or S_U . However, the application in which the M65C02A core is being used can make use of the **M** flag to provide privileged/non-privileged instructions, user/kernel mode address spaces, etc. For example, the M65C02A soft-core microcomputer uses the M flag to control the Memory Management Unit (MMU).

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.

2.1.9. 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.

2.1.9.1. 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 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 one (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

DRAWING NUMBER

1004-0900

REV

-

SCALE: NONE

SHEET 24 OF 92

In the FORTH VM, pushing and pulling IP is generally done using the return stack, RS. To reduce the number of cycles required for these operations, *ent*, *phi* and *pli*, the default return stack pointer (RSP) of the FORTH VM, RSP, is the auxiliary stack pointer, S_X. Thus, applying *osx* prefix instruction to these instructions, *ent*, *phi* and *pli*, will use the system stack pointer, S_K/S_U, i.e. the FORTH VM parameter stack pointer (PSP).

The VM IP register can also be loaded from, loaded into, or exchanged with the A_{TOS} register. These operations are enabled by the *ind*, *siz*, and *isz* prefix instructions in combination with the *dup* register stack instruction. These operations provide access to the programmer to the VM IP, but they also enable the programmer's direct use of the VM IP register and the IP-relative with auto-increment addressing mode instructions. These instructions are referred to as:

- *tai* (IP <= A_{TOS}),
- *tia* (A_{TOS} <= IP),
- *xai* (A_{TOS} <= IP; IP <= A_{TOS}).

2.1.9.2. 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.

By prefixing *ind* to 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. The default stack for the *phw* and *plw* instructions is RS, so the *osx* prefix is only required if these instructions need to target the parameter stack PS.

2.1.10. 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, *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 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 92	

the programmer should use the W65C02S *wai* (Wait for Interrupt) instruction instead of a self-referencing loop, or include an interruptable instruction such as *nop* inside the loop.

2.1.11. Summary of the M65C02A Core's Features/Capabilities

The following advanced features/capabilities define the M65C02A core:

- (1) The M65C02A core allows the 6502/65C02 index registers, X and Y, to be used as accumulators;
- (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 operation of the push down register stacks is modified such 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 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. S_U may be manipulated from kernel mode routines, but S_K is inaccessible to user mode routines. (**Note:** *On reset, the M65C02A defaults to kernel mode for compatibility with 6502/65C02 microprocessors. The M65C02A core 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.*)
- (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 *osx* is prefixed to instruction specific to the X register, S becomes the source/target for these instructions: *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*. When using the *pls/phs* instructions, S (S_K or S_U) is pulled/pushed using the auxiliary stack, S_X.*)
- (6) 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 1 (default) or page 0. (**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.*)
- (7) The M65C02A core's system stack registers, S_K and S_U, serve as base registers for the stack-relative addressing mode: **sp,S**, or **(sp,S)**. These addressing modes, which must be emulated by 6502/65C02 microprocessors, provide the capability needed to access



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 92


temporary stack variables used by programming languages like C and Pascal. (**Note:** the stack relative addressing modes are created by applying the *osx* prefix instruction to any instruction using a pre-indexed (by X) addressing mode. The *osx* instruction can be combined with the *siz* and *ind* prefix instructions. The W650C02S instructions *wai* and *stp* are replaced in the complete M65C02A instruction set by *osz* and *ois* prefix instructions. These two prefix instructions combine *osx* with *siz*, and *osx* with *siz* and *ind*. These prefix instructions provide better support for the stack-relative addressing mode.)

- (8) The M65C02A core's X Top-Of-Stack register, X_{TOS} , serves as a base register for the base-relative addressing modes: **bp,B**, and (**bp,B**). These addressing modes provide the stack frame capability needed by programming languages like C and Pascal, and which must be emulated by 6502/65C02 microprocessors.
- (9) The M65C02A core provides an instruction, *adj #imm*, to clean up the stack frame;
- (10) The M65C02A core provides two prefix instructions, *ind* and *isz*, that add indirection to an addressing mode.
- (11) 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.
- (12) The M65C02A core provides two prefix instructions, *siz* and *isz*, that increase the width of the ALU operation from 8 to 16 bits.
- (13) The M65C02A core provides support for the implementation of threaded code interpreters like the FORTH VM (virtual machine). 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.
- (14) The M65C02A core provides an IP-relative (with auto-increment) addressing mode which may be used to support more efficient implementation of common operations in a VM, or to provide register indirect access anywhere in memory;
- (15) 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.

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

	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 92	

- 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.

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.



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 92

(**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 inhibits the acceptance of maskable interrupts by the external interrupt handler.

2.2.2.2. INT : input

The external interrupt handler signals the M65C02A core that interrupt request is asserted using the INT signal. While an interrupt is un-serviced 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.

2.2.2.3. 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.



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 92

2.2.2.4. 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.

2.2.2.5. 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.

2.2.2.6. 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 the non-maskable interrupt 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.

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 the 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



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 92

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 microcycle in which it will be setting the V flag in the P register.

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 opcode for the next instruction.

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



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 31 OF 92

identified for the core's logic, which can then configure the M65C02A core's functional units appropriately.

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 CO-Processor 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 32 OF 92

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 ₂	NOP	M65C02A core does not read or write memory or I/O peripherals.
01 ₂	WR	M65C02A core writes to memory or I/O peripherals.
10 ₂	RD	M65C02A core reads memory or I/O peripherals.
11 ₂	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

DRAWING NUMBER

1004-0900

REV

-

SCALE: NONE

SHEET 33 OF 92

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 34 OF 92

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 being 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 35 OF 92

(**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
phi
pli

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)*.

	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 92	

(**Note:** if 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.)

2.2.9.4. 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 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
LDX #imm/zp/zp,Y/abs/abs,Y	<u>LDS</u> #imm/zp/zp,Y/abs/abs,Y
STX zp/zp,Y/abs	<u>STS</u> zp/zp,Y/abs
CPX #imm/zp/abs	<u>CPS</u> #imm/zp/abs
TAX/TXA	<u>TAS</u> / <u>TSA</u>
Uses the System Stack (S _K , S _U)	Uses the Auxiliary Stack (S _X)
PHA/PHP/PHX/PHY/PLA/PLP/PLX/PLY	PHA/PHP/ <u>PHS</u> /PHY/PLA/PLP/ <u>PLS</u> /PLY
PSH #imm16/zp/abs	PSH #imm16/zp/abs
PHR rel16	PHR rel16
PUL zp/abs	PUL zp/abs
JSR abs	JSR abs
BSR rel16	BSR rel16
RTS/RTI	RTS/RTI



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 37 OF 92

Notes:

- (1) *ind, siz, isz* and *oax* 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 base pointer into the system stack 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 _{TOS} (implements auxiliary stack logic) and Y _{TOS}
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 Module – Core Top Level Module

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 38 OF 92

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.

As described previously, the key to the enhanced instructions of the M65C02A core are the six prefix instructions. The registers which implement the flags for the six prefix instructions are implemented in the core's top level module. The rules regarding which prefix instructions may be applied simultaneously are implemented as part of the six prefix instruction flag registers.

The top level module also implements the output data multiplexer. This multiplexer is used to drive the output data from core with data originating in the address generator module or the ALU module. This logic provides the logic to output 8-bit or 16-bit data from the Memory Address Register (MAR) (which holds the 16-bit absolute address computed by the PHR rel16 instruction), the PC, the processor status word, and the ALU.



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 39 OF 92

2.3.2. M65C02A_MPC Module – 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 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.

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;

	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 40 OF 92	

- (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 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. M65C02A_AddrGen Module – 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.

The selection of the left and right operands of the adder is controlled directly by the microprogram. The operand select logic uses one-hot encoding which allows the microprogram great control of the operands, and facilitates in the implementation of the base and extended addressing modes supported by the M65C02A core.

The carry input to the adder is also microprogram controlled, which allows the microprogram to control the auto-increment of the PC, the MAR, and the stack pointer address. Control of the carry input is one way that the M65C02A is able to eliminate the dead cycles found on 6502/65C02 processors during stack pull operations: the address generator adder is used to generate the address of the operand in parallel with the increment of the stack pointer register. Control of the carry input also allows the address generator to decrement the MAR when performing relative addressing calculations to support the *phr rel16* instruction.

Several of the one-hot operand selects are modified by the register override prefix instructions: Sel_X, Sel_Y, Sel_A, Sel_S, and Ci. The *osx* prefix instruction exchanges Sel_X and Sel_S. The *oax* prefix instruction exchanges Sel_A and Sel_X. The *oay* prefix instruction exchanges Sel_A and Sel_Y. These exchanges of the register selects swap the left operand on the address generator. (Note: *osx* is mutually exclusive with *oax*, and *oax* is mutually exclusive with *oay*.) The *osx* prefix instruction affects the carry input to the adder by adding the Sel_X to the Ci input. This modification of Ci enables the use of 0-based stack-relative addressing modes with auxiliary stack S_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 A	CAGE CODE	DRAWING NUMBER 1004-0900	REV -
		SCALE: NONE		SHEET 41 OF 92	

2.3.4. M65C02A_ForthVM Module – 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. M65C02A_ALUv2 Module – 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 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_{TOS}, X_{TOS}, Y_{TOS}, 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.



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 42 OF 92

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 if 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 *bbrx/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 (\leq), greater than (>), greater than or equal (\geq), lower than (<), lower than or same (\leq), higher than (>), and higher than or same (\geq). 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. M65C02A_LST Module – 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 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 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 43 OF 92

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 *p1i* instructions are not routed through the ALU by the LST module.

2.3.5.2. M65C02A_LU Module – 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. M65C02A_SU Module – 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. M65C02A_AU Module – 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 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 perform 2's complement arithmetic for addition and subtraction but require the C flag in P to be 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 before the first addition/subtraction. The Intel/Zilog 8080/Z80 or the Intel x86 processors, provide two addition and two subtraction opcodes in order to support single and multi-precision arithmetic.

	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 92	

2.3.5.5. M65C02A_WrSel Module – 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 instruction, 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 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 FORTH VM instructions. 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 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 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 45 OF 92

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 swapping of the bits 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 reverses the bits of A_{TOS} .

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.

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 instead provided by A_{TOS} . 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 stack accesses default to S_X , which



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 92

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, 0x82, 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.

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} maintains its traditional post-indexing role unlike X_{TOS} which changes roles.

	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 92		

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 transfer. (**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, 0x82, 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 2.1.8 for more detail on P.



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 92

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:

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

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

- (16) Relative (16-bit): ***rel16***
- (17) Stack-relative: ***sp,S***
- (18) Post-Indexed (by Y) Stack-relative Indirect: ***(sp,S),Y***
- (19) Base-relative: ***bp,B***
- (20) 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 6502/65C02 processor:

- (21) Post-Indexed Zero Page Indirect (by X): ***(zp),X***
- (22) Post-Indexed Zero Page Indirect (by Y): ***(zp),Y***
- (23) Zero Page Double Indirect: ***((zp))***
- (24) Post-Indexed (by X) Zero Page Double Indirect: ***((zp)),X***
- (25) Post-Indexed (by Y) Zero Page Double Indirect: ***((zp)),Y***
- (26) Post-Indexed (by X) Absolute Indirect: ***(abs),X***
- (27) Post-Indexed (by Y) Absolute Indirect: ***(abs),Y***
- (28) Absolute Double Indirect: ***((abs))***
- (29) Post-Indexed (by X) Absolute Double Indirect: ***((abs)),X***
- (30) Stack-relative Indirect: ***(sp,S)***
- (31) Post-Indexed (by Y) Stack-relative Double Indirect: ***((sp,S)),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 49 OF 92

- (32) Base-relative Indirect: **(bp,B)**
- (33) IP-relative with Auto-increment Indirect: **(ip,l++)**

If the effects of the **osx**, **oax** and **oay** register override prefix instructions are considered, addressing (and M65C02A-unique) addressing modes are possible:

- (34) Indexed (by A) Zero Page Direct: **zp,A**
- (35) Post-Indexed (by A) Zero Page Indirect: **(zp),A**
- (36) Post-Indexed (by A) Zero Page Double Indirect: **((zp)),A**
- (37) Indexed (by A) Absolute: **abs,A**
- (38) Post-Indexed (by A) Absolute Indirect: **(abs),A**
- (39) Post-Indexed (by A) Absolute Double Indirect: **((abs)),A**
- (40) Post-Indexed (by Y) Base-relative Indirect: **(bp,B),Y**

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:

	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 50 OF 92	

```

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

```

In all of these instructions, the destination register operand is implied by the instruction opcode, and any required second operands, i.e. ± 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*, generally has no effect on the implicit/accumulator addressing mode instructions. The exception is that applying *ind* or *isz* to the *lsl* instruction converts the operations from a logical right shift, i.e. unsigned divide by 2, to an arithmetic right shift, i.e. signed divide by 2: *ind lsl* \Rightarrow *asr*, and *isz lsl* \Rightarrow *asr.w*. The SIZ prefix flag has an effect on the operation width for implicitly/accumulator addressing mode instructions. When SIZ is set by the *siz* or *isz* prefix instructions, the 16-bit operations are performed.

(Note: unless the SIZ internal flag 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*.

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:

```

oax asl/rol/lsl/ror A  $\Leftrightarrow$  asl/rol/lsl/ror X
oay asl/rol/lsl/ror A  $\Leftrightarrow$  asl/rol/lsl/ror Y

```

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.



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 92

Thus, applying **osx** to the shift/rotate instructions has no effect: the accumulator remains the 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
[siz/isz] ldx #$0180
ldy #47
```

These three examples demonstrate how the immediate addressing mode may be used to load the A_{TOS}, X_{TOS}, or Y_{TOS} 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, and the ALU performs a 16-bit operation.

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_K or S_U, 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 52 OF 92	

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 fewer 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:

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

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

$$Mem[EA] = DO$$

3.3.1. 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 addressed 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:

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

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 cation 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:

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

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

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

DRAWING NUMBER

1004-0900

REV

-

SCALE: NONE

SHEET 53 OF 92

the data pointer, but not for the operand read/write cycles. The effective address for the data pointer to the 8-bit/16-bit operand is given as:

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

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

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

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

The core's operand register M is transferred to an internal register, A, X, Y, S, P, during the next memory cycle while the next instruction is being fetched.

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

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

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

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 the basic 6502/65C02 addressing modes. Using these three prefix instructions, the enhanced M65C02A provides operations and addressing modes that should improve the programmer's ability to better use the following instructions.

Table 6: Effect of *ind* 6502/65C02/M65C02A zp direct Instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>ind bit zp</i>	<i>bit (zp)</i>	Test zero page indirect memory with mask in A _{TOS}	N	Y	Y
<i>ind trb zp</i>	<i>trb (zp)</i>	Test and reset zero page indirect memory with mask in A _{TOS}	N	Y	Y
<i>ind tsb zp</i>	<i>tsb (zp)</i>	Test and set zero page indirect memory with mask in A _{TOS}	N	Y	Y
<i>ind stx zp</i>	<i>stx (zp)</i>	Store X _{TOS} to zero page indirect memory	Y	Y	N
<i>ind ldx zp</i>	<i>ldx (zp)</i>	Load X _{TOS} from zero page indirect memory	Y	Y	N
<i>ind cpx zp</i>	<i>cpx (zp)</i>	Compare X _{TOS} to zero page indirect memory	Y	Y	N
<i>ind sty zp</i>	<i>sty (zp)</i>	Store Y _{TOS} to zero page indirect memory	N	N	Y
<i>ind ldy zp</i>	<i>ldy (zp)</i>	Load Y _{TOS} from zero page indirect memory	N	N	Y
<i>ind cpy zp</i>	<i>cpy (zp)</i>	Compare Y _{TOS} to zero page indirect memory	N	N	Y
<i>ind inc zp</i>	<i>inc (zp)</i>	Increment zero page indirect memory	N	N	N
<i>ind dec zp</i>	<i>dec (zp)</i>	Decrement zero page indirect memory	N	N	N
<i>ind asl zp</i>	<i>asl (zp)</i>	Arithmetic shift zero page indirect memory left (through carry)	N	N	N
<i>ind rol zp</i>	<i>rol (zp)</i>	Rotate zero page indirect memory left (through carry)	N	N	N
<i>ind lsr zp</i>	<i>lsr (zp)</i>	Logical shift zero page indirect memory right (through carry)	N	N	N
<i>ind ror zp</i>	<i>ror (zp)</i>	Rotate zero page indirect memory right (through carry)	N	N	N
<i>ind psh zp</i>	<i>psh (zp)</i>	Push 16-bit value from zero page indirect memory to system stack	Y	N	N
<i>ind pul zp</i>	<i>pul (zp)</i>	Pull 16-bit value from system stack to zero page indirect memory	Y	N	N
<i>ind rmbx zp</i>	<i>rmbX (zp)</i>	Reset (zero page) zero page indirect memory bit [X:0...7]	N	N	N
<i>ind smbx zp</i>	<i>smbX (zp)</i>	Set (zero page) zero page indirect memory bit [X:0...7]	N	N	N
<i>ind bbrx] zp,rel8</i>	<i>bbRX (zp),rel8</i>	Branch if (zero page) zero page indirect memory bit [X:0...7] reset	N	N	N
<i>ind bbsx zp,rel8</i>	<i>bbsX (zp),rel8</i>	Branch if (zero page) zero page indirect memory bit [X:0...7] set	N	N	N

Table 7: Effect of *siz* 6502/65C02/M65C02A zp direct Instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>siz bit zp</i>	<i>bit.w zp</i>	Test zero page memory with mask in A _{TOS}	N	Y	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

SCALE: NONE

DRAWING NUMBER

1004-0900

REV

-

SHEET 54 OF 92

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>siz trb zp</i>	<i>trb.w zp</i>	Test and reset zero page memory with mask in A _{TOS}	N	Y	Y
<i>siz tsb zp</i>	<i>tsb.w zp</i>	Test and set zero page memory with mask in A _{TOS}	N	Y	Y
<i>siz stx zp</i>	<i>stx.w zp</i>	Store X _{TOS} to zero page memory	Y	Y	N
<i>siz ldx zp</i>	<i>ldx.w zp</i>	Load X _{TOS} from zero page memory	Y	Y	N
<i>siz cpx zp</i>	<i>cpx.w zp</i>	Compare X _{TOS} to zero page memory	Y	Y	N
<i>siz sty zp</i>	<i>sty.w zp</i>	Store Y _{TOS} to zero page memory	N	N	Y
<i>siz ldy zp</i>	<i>ldy.w zp</i>	Load Y _{TOS} from zero page memory	N	N	Y
<i>siz cpy zp</i>	<i>cpy.w zp</i>	Compare Y _{TOS} to zero page memory	N	N	Y
<i>siz inc zp</i>	<i>inc.w zp</i>	Increment zero page memory	N	N	N
<i>siz dec zp</i>	<i>dec.w zp</i>	Decrement zero page memory	N	N	N
<i>siz asl zp</i>	<i>asl.w zp</i>	Arithmetic shift zero page memory left (through carry)	N	N	N
<i>siz rol zp</i>	<i>rol.w zp</i>	Rotate zero page memory left (through carry)	N	N	N
<i>siz lsr zp</i>	<i>lsr.w zp</i>	Logical shift zero page memory right (through carry)	N	N	N
<i>siz ror zp</i>	<i>ror.w zp</i>	Rotate zero page memory right (through carry)	N	N	N

Table 8: Effect of *isz* 6502/65C02/M65C02A zp direct Instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>isz bit zp</i>	<i>bit.w (zp)</i>	Test zero page indirect memory with mask in A _{TOS}	N	Y	Y
<i>isz trb zp</i>	<i>trb.w (zp)</i>	Test and reset zero page indirect memory with mask in A _{TOS}	N	Y	Y
<i>isz tsb zp</i>	<i>tsb.w (zp)</i>	Test and set zero page indirect memory with mask in A _{TOS}	N	Y	Y
<i>isz stx zp</i>	<i>stx.w (zp)</i>	Store X _{TOS} to zero page indirect memory	Y	Y	N
<i>isz ldx zp</i>	<i>ldx.w (zp)</i>	Load X _{TOS} from zero page indirect memory	Y	Y	N
<i>isz cpx zp</i>	<i>cpx.w (zp)</i>	Compare X _{TOS} to zero page indirect memory	Y	Y	N
<i>isz sty zp</i>	<i>sty.w (zp)</i>	Store Y _{TOS} to zero page indirect memory	N	N	Y
<i>isz ldy zp</i>	<i>ldy.w (zp)</i>	Load Y _{TOS} from zero page indirect memory	N	N	Y
<i>isz cpy zp</i>	<i>cpy.w (zp)</i>	Compare Y _{TOS} to zero page indirect memory	N	N	Y
<i>isz inc zp</i>	<i>inc.w (zp)</i>	Increment zero page indirect memory	N	N	N
<i>isz dec zp</i>	<i>dec.w (zp)</i>	Decrement zero page indirect memory	N	N	N
<i>isz asl zp</i>	<i>asl.w (zp)</i>	Arithmetic shift zero page indirect memory left (through carry)	N	N	N
<i>isz rol zp</i>	<i>rol.w (zp)</i>	Rotate zero page indirect memory left (through carry)	N	N	N
<i>isz lsr zp</i>	<i>lsr.w (zp)</i>	Logical shift zero page indirect memory right (through carry)	N	N	N
<i>isz ror zp</i>	<i>ror.w (zp)</i>	Rotate zero page indirect memory right (through carry)	N	N	N

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. Applying the *osx* prefix instruction to the instructions specific to the X register provides a number of new instructions that affect the system stack pointer:

Table 9: Effect of *osx* on 6502/65C02 zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>osx stx zp</i>	<i>sta.s zp</i>	Store S (S _K /S _U) to zero page location	Y	Y
<i>osx ldx zp</i>	<i>lda.s zp</i>	Load S (S _K /S _U) from zero page location	Y	Y
<i>osx cpx zp</i>	<i>cmp.s zp</i>	Compare S (S _K /S _U) to zero page location	Y	Y
<i>osx psh zp</i>	<i>psh.s zp</i>	Push 16-bit value from zero page to auxiliary stack	Y	N
<i>osx pul zp</i>	<i>pul.s zp</i>	Pull 16-bit value from auxiliary stack to zero page	Y	N

The following tables illustrate the effect of the *oax* and *oay* prefix instructions when applied to zp direct instructions that operate on the accumulator A:

	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 92	

Table 10: Effect of *oax* on 6502/65C02 zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oax anl zp</i>	<i>anl.x zp</i>	Bitwise AND X_{TOS} with zero page location	Y	Y
<i>oax ora zp</i>	<i>ora.x zp</i>	Bitwise OR X_{TOS} with zero page location	Y	Y
<i>oax eor zp</i>	<i>eor.x zp</i>	Bitwise XOR X_{TOS} with zero page location	Y	Y
<i>oax adc zp</i>	<i>adc.x zp</i>	Add with Carry X_{TOS} with zero page location	Y	Y
<i>oax sbc zp</i>	<i>adc.x zp</i>	Subtract with Borrow X_{TOS} with zero page location	Y	Y
<i>oax bit zp</i>	<i>bit.x zp</i>	Test bit from zero page location with X_{TOS}	Y	Y
<i>oax trb zp</i>	<i>trb.x zp</i>	Test and reset bit in zero page location with X_{TOS}	Y	Y
<i>oax tsb zp</i>	<i>tsb.x zp</i>	Test and reset bit in zero page location with X_{TOS}	Y	Y

Table 11: Effect of *oay* on 6502/65C02 zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oay anl zp</i>	<i>anl.y zp</i>	Bitwise AND Y_{TOS} with zero page location	Y	Y
<i>oay ora zp</i>	<i>ora.y zp</i>	Bitwise OR Y_{TOS} with zero page location	Y	Y
<i>oay eor zp</i>	<i>eor.y zp</i>	Bitwise XOR Y_{TOS} with zero page location	Y	Y
<i>oay adc zp</i>	<i>adc.y zp</i>	Add with Carry Y_{TOS} with zero page location	Y	Y
<i>oay sbc zp</i>	<i>adc.y zp</i>	Subtract with Borrow Y_{TOS} with zero page location	Y	Y
<i>oay bit zp</i>	<i>bit.y zp</i>	Test bit from zero page location with Y_{TOS}	Y	Y
<i>oay trb zp</i>	<i>trb.y zp</i>	Test and reset bit in zero page location with Y_{TOS}	Y	Y
<i>oay tsb zp</i>	<i>tsb.y zp</i>	Test and reset bit in zero page location with Y_{TOS}	Y	Y

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

The pre-indexed (by X) zero page direct addressing mode is common to the 6502, 65C02 and the M65C02A. It provides a way to address page zero locations indexed by contents of the X_{TOS} index register.

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

$$EA = (X_{TOS}[15:9] == 0) ? \{(X_{TOS} + \{0x00, zp\} \% 256) : \{X_{TOS} + \{0x00, zp\}\}$$

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, i.e. $X_{TOS}[15:9] == 0$, the addressing mode is base plus offset 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 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 56 OF 92

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

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

3.4.1. Effect of the *ind/siz/isz* Prefix Instructions

If the IND flag is asserted, any instructions using the pre-indexed 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 addressed 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:

$$\text{EA} = \{\{\text{Mem}[\{0x00, \text{zp}\} + 1] \% 256], \text{Mem}[\{0x00, \text{zp}\}]\} + \text{X}_{\text{TOS}}$$

As discussed elsewhere, indirection is applied before indexing. Thus, the index operation is performed after the pointer to the data has been loaded from memory. The indexing calculation is not performed using modulo arithmetic, and the full 16-bit X register value will be used.

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 location 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:

$$\begin{aligned} \text{EA}[0] &= (\text{X}_{\text{TOS}}[15:9] == 0) ? \{(\text{X}_{\text{TOS}} + \{0x00, \text{zp}\} \% 256) : \text{X}_{\text{TOS}} + \{0x00, \text{zp}\} \\ \text{EA}[1] &= (\text{X}_{\text{TOS}}[15:9] == 0) ? \{\text{EA}[0] + 1\} \% 256 : \text{EA}[0] + 1 \end{aligned}$$

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 data pointer, but not for the operand read/write cycles. The effective address for the data pointer to the 8-bit/16-bit operand is given as:

$$\text{EA} = \{\{\text{Mem}[\{0x00, \text{zp}\} + 1] \% 256], \text{Mem}[\{0x00, \text{zp}\}]\} + \text{X}_{\text{TOS}}$$

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

$$\begin{aligned} \text{M}[7:0] &= \text{Mem}[\text{EA}] \\ \text{M}[15:8] &= \text{Mem}[\text{EA} + 1] \end{aligned}$$



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 92

The core's operand register M is transferred to an internal register, A, X, Y, and S during the next memory cycle while the next instruction is being fetched.

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

$$\text{Mem}[\text{EA}] = \text{DO}[7:0]$$

$$\text{Mem}[\text{EA} + 1] = \text{DO}[15:8]$$

The microprogram controls which byte of the result is output on the data bus of the core for each 8-bit write cycle. The following table illustrates the effects of the ind/siz/isz prefix instructions on the 6502/65C02 instructions supporting the pre-indexed (by X) zero page direct addressing mode:

Table 12: Effect of *ind* on 6502/65C02 pre-indexed (by X) zp direct instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>ind anl zp,X</i>	<i>anl (zp),X</i>	Bitwise AND A _{TOS} with zero page location post-indexed by X _{TOS}	N	Y	Y
<i>ind ora zp,X</i>	<i>ora (zp),X</i>	Bitwise OR A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind eor zp,X</i>	<i>eor (zp),X</i>	Bitwise XOR A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind adc zp,X</i>	<i>adc (zp),X</i>	Add with Carry A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind sta zp,X</i>	<i>sta (zp),X</i>	Store A _{TOS} to zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind lda zp,X</i>	<i>lda (zp),X</i>	Load A _{TOS} from zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind cmp zp,X</i>	<i>cmp (zp),X</i>	Compare A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind sbc zp,X</i>	<i>sbc (zp),X</i>	Subtract zero page location indirect post-indexed by X _{TOS} from A _{TOS}	N	Y	Y
<i>ind asl zp,X</i>	<i>asl (zp),X</i>	Arithmetic shift zero page location indirect post-indexed by X _{TOS} left	N	Y	Y
<i>ind rol zp,X</i>	<i>rol (zp),X</i>	Rotate zero page location indirect post-indexed by X _{TOS} left	N	Y	Y
<i>ind lsr zp,X</i>	<i>lsr (zp),X</i>	Logical shift zero page location indirect post-indexed by X _{TOS} right	N	Y	Y
<i>ind ror zp,X</i>	<i>ror (zp),X</i>	Rotate zero page location indirect post-indexed by X _{TOS} right	N	Y	Y
<i>ind dec zp,X</i>	<i>dec (zp),X</i>	Decrement zero page indirect location post-indexed by X _{TOS}	N	Y	Y
<i>ind inc zp,X</i>	<i>inc (zp),X</i>	Increment zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind sty zp,X</i>	<i>sty (zp),X</i>	Store Y _{TOS} in s zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>ind sty zp,X</i>	<i>sty (zp),X</i>	Store Y _{TOS} in zero page location indirect post-indexed by X _{TOS}	N	Y	Y

Table 13: Effect of *siz* on 6502/65C02 pre-indexed (by X) zp direct instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>siz anl zp,X</i>	<i>anl.w zp,X</i>	Bitwise AND A _{TOS} with zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz ora zp,X</i>	<i>ora.w zp,X</i>	Bitwise OR A _{TOS} with zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz eor zp,X</i>	<i>eor.w zp,X</i>	Bitwise XOR A _{TOS} with zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz adc zp,X</i>	<i>adc.w zp,X</i>	Add with Carry A _{TOS} with zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz sta zp,X</i>	<i>sta.w zp,X</i>	Store A _{TOS} to zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz lda zp,X</i>	<i>lda.w zp,X</i>	Load A _{TOS} from zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz cmp zp,X</i>	<i>cmp.w zp,X</i>	Compare A _{TOS} with zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz sbc zp,X</i>	<i>sbc.w zp,X</i>	Subtract zero page location post-indexed by X _{TOS} from A _{TOS}	Y	Y	Y
<i>siz asl zp,X</i>	<i>asl.w zp,X</i>	Arithmetic shift zero page location post-indexed by X _{TOS} left	Y	Y	Y
<i>siz rol zp,X</i>	<i>rol.w zp,X</i>	Rotate zero page location post-indexed by X _{TOS} left	Y	Y	Y
<i>siz lsr zp,X</i>	<i>lsr.w zp,X</i>	Logical shift zero page location post-indexed by X _{TOS} right	Y	Y	Y
<i>siz ror zp,X</i>	<i>ror.w zp,X</i>	Rotate zero page location post-indexed by X _{TOS} right	Y	Y	Y
<i>siz dec zp,X</i>	<i>dec.w zp,X</i>	Decrement zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz inc zp,X</i>	<i>inc.w zp,X</i>	Increment zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz sty zp,X</i>	<i>sty.w zp,X</i>	Store Y _{TOS} in s zero page location post-indexed by X _{TOS}	Y	Y	Y
<i>siz sty zp,X</i>	<i>sty.w zp,X</i>	Store Y _{TOS} in zero page location post-indexed by X _{TOS}	Y	Y	Y

Table 14: Effect of *isz* on 6502/65C02 pre-indexed (by X) zp direct instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>isz anl zp,X</i>	<i>anl.w (zp),X</i>	Bitwise AND A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz ora zp,X</i>	<i>ora.w (zp),X</i>	Bitwise OR A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz eor zp,X</i>	<i>eor.w (zp),X</i>	Bitwise XOR A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz adc zp,X</i>	<i>adc.w (zp),X</i>	Add with Carry A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz sta zp,X</i>	<i>sta.w (zp),X</i>	Store A _{TOS} to zero page location indirect post-indexed by X _{TOS}	N	Y	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 58 OF 92

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>isz lda zp,X</i>	<i>lda.w (zp),X</i>	Load A _{TOS} from zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz cmp zp,X</i>	<i>cmp.w (zp),X</i>	Compare A _{TOS} with zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz sbc zp,X</i>	<i>sbc.w (zp),X</i>	Subtract zero page location indirect post-indexed by X _{TOS} from A _{TOS}	N	Y	Y
<i>isz asl zp,X</i>	<i>asl.w (zp),X</i>	Arithmetic shift zero page location indirect post-indexed by X _{TOS} left	N	Y	Y
<i>isz rol zp,X</i>	<i>rol.w (zp),X</i>	Rotate zero page location indirect post-indexed by X _{TOS} left	N	Y	Y
<i>isz lsr zp,X</i>	<i>lsr.w (zp),X</i>	Logical shift zero page location indirect post-indexed by X _{TOS} right	N	Y	Y
<i>isz ror zp,X</i>	<i>ror.w (zp),X</i>	Rotate zero page location indirect post-indexed by X _{TOS} right	N	Y	Y
<i>isz dec zp,X</i>	<i>dec.w (zp),X</i>	Decrement zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz inc zp,X</i>	<i>inc.w (zp),X</i>	Increment zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz sty zp,X</i>	<i>sty.w (zp),X</i>	Store Y _{TOS} in s zero page location indirect post-indexed by X _{TOS}	N	Y	Y
<i>isz sty zp,X</i>	<i>sty.w (zp),X</i>	Store Y _{TOS} in zero page location indirect post-indexed by X _{TOS}	N	Y	Y

3.4.2. 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. Applying the *osx* prefix instruction to the instructions using the X_{TOS} register as an index register essentially convert the pre-indexed zero page direct addressing mode to the stack relative addressing mode:

Table 15: Effect of *osx* on 6502/65C02 pre-indexed (by X) zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ	OAY
<i>osx anl zp,X</i>	<i>anl.s zp,S</i>	Bitwise AND A _{TOS} with stack relative location	N	Y	Y
<i>osx ora zp,X</i>	<i>ora.s zp,S</i>	Bitwise OR A _{TOS} with stack relative location	N	Y	Y
<i>osx eor zp,X</i>	<i>eor.s zp,S</i>	Bitwise XOR A _{TOS} with stack relative location	N	Y	Y
<i>osx adc zp,X</i>	<i>adc.s zp,S</i>	Add with Carry A _{TOS} with stack relative location	N	Y	Y
<i>osx sta zp,X</i>	<i>sta.s zp,S</i>	Store A _{TOS} to stack relative location	N	Y	Y
<i>osx lda zp,X</i>	<i>lda.s zp,S</i>	Load A _{TOS} from stack relative location	N	Y	Y
<i>osx cmp zp,X</i>	<i>cmp.s zp,S</i>	Compare A _{TOS} with stack relative location	N	Y	Y
<i>osx sbc zp,X</i>	<i>sbc.s zp,S</i>	Subtract stack relative location from A _{TOS}	N	Y	Y
<i>osx asl zp,X</i>	<i>asl.s zp,S</i>	Arithmetic shift stack relative location left	N	Y	N
<i>osx rol zp,X</i>	<i>rol.s zp,S</i>	Rotate stack relative location left	N	Y	N
<i>osx lsr zp,X</i>	<i>lsr.s zp,S</i>	Logical shift stack relative location right	N	Y	N
<i>osx ror zp,X</i>	<i>ror.s zp,S</i>	Rotate stack relative location right	N	Y	N
<i>osx dec zp,X</i>	<i>dec.s zp,S</i>	Decrement stack relative location	N	Y	N
<i>osx inc zp,X</i>	<i>inc.s zp,S</i>	Increment stack relative location	N	Y	N
<i>osx sty zp,X</i>	<i>sty.s zp,S</i>	Store Y _{TOS} in stack relative location	N	Y	N
<i>osx sty zp,X</i>	<i>sty.s zp,S</i>	Store Y _{TOS} in stack relative location	N	Y	N

The stack relative addressing mode created by applying the *osx* prefix to the pre-indexed zero page direct addressing mode has a number of limitations relative the M65C02A-specific instructions using the stack relative addressing mode:

- (1) Only the system stack pointer may be used for the base pointer;
- (2) Applying indirection using *ind/isz* (in addition to *osx*) does not yield a stack-relative indirect mode.



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 59 OF 92

Apply **osx** to a M65C02A-specific instruction that uses the stack relative addressing mode and the auxiliary stack pointer is used as the base for the effective address calculation instead of the system stack pointer. Apply **osx** and **ind/isz** to an instruction using the pre-indexed (by X) zero page direct addressing mode and the result is a post-indexed (by S) zero page indirect addressing mode rather than a stack relative indirect addressing mode.

The following tables illustrate the effect of the **oax** and **oay** prefix instructions when applied to pre-indexed (by X) zp direct instructions:

Table 16: Effect of *oax* on 6502/65C02 pre-indexed (by X) zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oax anl zp,X</i>	<i>anl.x zp,A</i>	Bitwise AND X_{TOS} with zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax ora zp,X</i>	<i>ora.x zp,A</i>	Bitwise OR X_{TOS} with zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax eor zp,X</i>	<i>eor.x zp,A</i>	Bitwise XOR X_{TOS} with zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax adc zp,X</i>	<i>adc.x zp,A</i>	Add with Carry X_{TOS} with zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax sta zp,X</i>	<i>sta.x zp,A</i>	Store X_{TOS} to zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax lda zp,X</i>	<i>lda.x zp,A</i>	Load X_{TOS} from zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax cmp zp,X</i>	<i>cmp.x zp,A</i>	Compare X_{TOS} with zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax sbc zp,X</i>	<i>sbc.x zp,A</i>	Subtract zero page location pre-indexed by A_{TOS} from X_{TOS}	Y	Y
<i>oax sty zp,X</i>	<i>sty.x zp,A</i>	Store Y_{TOS} to zero page location pre-indexed by A_{TOS}	Y	Y
<i>oax ldy zp,X</i>	<i>ldy.x zp,A</i>	Load Y_{TOS} from zero page location pre-indexed by A_{TOS}	Y	Y

Table 17: Effect of *oay* on 6502/65C02 pre-indexed (by X) zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oay anl zp,X</i>	<i>anl.y zp,X</i>	Bitwise AND Y_{TOS} with zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay ora zp,X</i>	<i>ora.y zp,X</i>	Bitwise OR Y_{TOS} with zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay eor zp,X</i>	<i>eor.y zp,X</i>	Bitwise XOR Y_{TOS} with zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay adc zp,X</i>	<i>adc.y zp,X</i>	Add with Carry Y_{TOS} with zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay sta zp,X</i>	<i>sta.y zp,X</i>	Store Y_{TOS} to zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay lda zp,X</i>	<i>lda.y zp,X</i>	Load Y_{TOS} from zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay cmp zp,X</i>	<i>cmp.y zp,X</i>	Compare Y_{TOS} with zero page location pre-indexed by X_{TOS}	Y	Y
<i>oay sbc zp,X</i>	<i>sbc.y zp,X</i>	Subtract zero page location pre-indexed by X_{TOS} from Y_{TOS}	Y	Y

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

The post-indexed (by Y) zero page direct addressing mode is common to the 6502, 65C02 and the M65C02A. It provides a way load and store X_{TOS} from/to page zero locations post-indexed by Y_{TOS} .

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

$$EA = (Y_{TOS}[15:9] == 0) ? \{(Y_{TOS} + \{0x00, zp\}) \% 256\} : \{Y_{TOS} + \{0x00, zp\}\}$$

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

DRAWING NUMBER

1004-0900

REV

-

SCALE: NONE

SHEET 60 OF 92

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 offset without modulo 256 arithmetic.)

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

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

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

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

3.5.1. Effect of the *ind/siz/isz* Prefix Instructions

If the IND flag is asserted, any instructions using the post-indexed (by Y) 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 addressed 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:

$$EA = \{\{\text{Mem}[\{0x00, zp\} + 1] \% 256], \text{Mem}[\{0x00, zp\}]\} + Y_{TOS}$$

As discussed elsewhere, indirection is applied before indexing. Thus, the index operation is performed after the pointer to the data has been loaded from memory. The indexing calculation is not performed using modulo arithmetic, and the full 16-bit Y_{TOS} register value will be used.

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 location 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:

$$\begin{aligned} EA[0] &= (Y_{TOS}[15:9] == 0) ? \{(Y_{TOS} + \{0x00, zp\}) \% 256\} : Y_{TOS} + \{0x00, zp\} \\ EA[1] &= (Y_{TOS}[15:9] == 0) ? \{EA[0] + 1\} \% 256 : EA[0] + 1 \end{aligned}$$

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 data pointer, but not for the operand read/write cycles. The effective address for the data pointer to the 8-bit/16-bit operand is given as:

$$EA = \{\{\text{Mem}[\{0x00, zp\} + 1] \% 256], \text{Mem}[\{0x00, zp\}]\} + Y_{TOS}$$



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 61 OF 92

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

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

$$M[15:8] = \text{Mem}[EA + 1]$$

The core's operand register M is transferred to an internal register, X_{TOS} or S, during the next memory cycle while the next instruction is being fetched.

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

$$\text{Mem}[EA] = \text{DO}[7:0]$$

$$\text{Mem}[EA + 1] = \text{DO}[15:8]$$

The microprogram controls which byte of the result is output on the data bus of the core for each 8-bit write cycle. The following tables illustrate the effect of *ind/siz/isz* on 6502/65C02 instructions using the post-indexed (by Y) zero page direct addressing mode:

Table 18: Effect of *ind* on 6502/65C02 post-indexed (by Y) zp direct instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>ind stx zp,y</i>	<i>stx (zp),y</i>	Store X _{TOS} to zero page indirect location post-indexed by Y _{TOS}	Y	N	N
<i>ind ldx zp,y</i>	<i>ldx (zp),y</i>	Load X _{TOS} with zero page indirect location post-indexed by Y _{TOS}	Y	N	N

Table 19: Effect of *siz* on 6502/65C02 post-indexed (by Y) zp direct instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>siz stx zp,y</i>	<i>stx.w (zp),y</i>	Store X _{TOS} to zero page location post-indexed by Y _{TOS}	Y	N	N
<i>siz ldx zp,y</i>	<i>ldx.w (zp),y</i>	Load X _{TOS} with zero page location post-indexed by Y _{TOS}	Y	N	N

Table 20: Effect of *isz* on 6502/65C02 post-indexed (by Y) zp direct instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>isz stx zp,y</i>	<i>stx.w (zp),y</i>	Store X _{TOS} to zero page indirect location post-indexed by Y _{TOS}	Y	N	N
<i>isz ldx zp,y</i>	<i>ldx.w (zp),y</i>	Load X _{TOS} with zero page indirect location post-indexed by Y _{TOS}	Y	N	N

3.5.2. 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 following table illustrates the effect of the *osx* prefix instruction when applied to post-indexed (by Y) zp direct instructions:

Table 21: Effect of *osx* on 6502/65C02 post-indexed (by Y) zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ	OAY
<i>osx stx zp,y</i>	<i>stx.s zp,y</i>	Store S to zero page location post-indexed by Y _{TOS}	N	Y	N
<i>osx ldx zp,y</i>	<i>ldx.s zp,y</i>	Load S from zero page location post-indexed by Y _{TOS}	N	Y	N

The following tables illustrate the effect of the *oax/oay* prefix instruction when applied to post-indexed (by Y) zp direct 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 62 OF 92	

Table 22: Effect of *oax* on 6502/65C02 post-indexed (by Y) zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oax stx zp,Y</i>	<i>stx.x zp,Y</i>	Store A _{TOS} to zero page location post-indexed by Y _{TOS}	Y	Y
<i>oax ldx zp,Y</i>	<i>ldx.x zp,Y</i>	Load A _{TOS} from zero page location post-indexed by Y _{TOS}	Y	Y

Table 23: Effect of *oay* on 6502/65C02 post-indexed (by Y) zp direct instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oay stx zp,Y</i>	<i>stx.y zp,A</i>	Store X _{TOS} to zero page location post-indexed by A _{TOS}	Y	Y
<i>oay ldx zp,Y</i>	<i>stx.y zp,A</i>	Load X _{TOS} from zero page location post-indexed by A _{TOS}	Y	Y

3.6. Zero Page Indirect [(zp)]

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

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

$$EA = \{ \text{Mem}[(zp + 1) \% 256], \text{Mem}[zp] \}$$

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. (zp + 1) % 256.

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:

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

3.6.1. Effect of the *ind/siz/isz* Prefix Instructions

If the IND flag is asserted, any instructions using the zero page indirect addressing mode will automatically perform an additional indirection operation after indirection is performed using the zero page address supplied. The low byte of the pointer in zero page will be read from the addressed 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:

	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 63 OF 92	

$$\text{tmp} = \{\text{Mem}[(\text{zp} + 1) \% 256], \text{Mem}[\text{zp}]\}$$

$$\text{EA} = \{\text{Mem}[(\text{tmp} + 1)], \text{Mem}[\text{tmp}]\}$$

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 effective address location. The high byte will be read from or written to the next sequential location. The each byte of the 16-bit operand are given as:

$$\text{EA} = \{\text{Mem}[(\text{zp} + 1) \% 256], \text{Mem}[\text{zp}]\}$$

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

$$\text{M}[15:8] = \text{Mem}[\text{EA}+1]$$

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 data pointer, but not for the operand read/write cycles. The effective address for the data pointer to the 8-bit/16-bit operand is given as:

$$\text{tmp} = \{\text{Mem}[(\text{zp} + 1) \% 256], \text{Mem}[\text{zp}]\}$$

$$\text{EA} = \{\text{Mem}[(\text{tmp} + 1)], \text{Mem}[\text{tmp}]\}$$

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

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

$$\text{M}[15:8] = \text{Mem}[\text{EA} + 1]$$

The core's operand register M is transferred to an internal register, A, X, and Y, during the next memory cycle while the next instruction is being fetched.

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

$$\text{Mem}[\text{EA}] = \text{DO}[7:0]$$

$$\text{Mem}[\text{EA} + 1] = \text{DO}[15:8]$$

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

The effect of the *ind/siz/isz* prefix instruction on the zero page indirect instructions is tabulated in the following three tables:

Table 24: Effect of *ind* on 6502/65C02/M65C02A zp indirect Instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>ind anl (zp)</i>	<i>anl ((zp))</i>	Bitwise AND A _{TOS} with zero page double indirect location	N	Y	Y
<i>ind ora (zp)</i>	<i>ora ((zp))</i>	Bitwise OR A _{TOS} with zero page double indirect location	N	Y	Y
<i>ind eor (zp)</i>	<i>eor ((zp))</i>	Bitwise XOR A _{TOS} with zero page double indirect location	N	Y	Y
<i>ind adc (zp)</i>	<i>adc ((zp))</i>	Add with Carry A _{TOS} with zero page double indirect location	N	Y	Y
<i>ind sta (zp)</i>	<i>sta ((zp))</i>	Store A _{TOS} to with zero page double indirect location	N	Y	Y
<i>ind lda (zp)</i>	<i>lda ((zp))</i>	Load A _{TOS} from with zero page double indirect location	N	Y	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 64 OF 92

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>ind cmp (zp)</i>	<i>cmp ((zp))</i>	Compare A _{TOS} with zero page double indirect location	N	Y	Y
<i>ind sbc (zp)</i>	<i>sbc ((zp))</i>	Subtract zero page double indirect location from A _{TOS}	N	Y	Y

Table 25: Effect of *siz* on 6502/65C02/M65C02A zp indirect Instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>siz anl (zp)</i>	<i>anl.w (zp)</i>	Bitwise AND A _{TOS} with zero page indirect location	N	Y	Y
<i>siz ora (zp)</i>	<i>ora.w (zp)</i>	Bitwise OR A _{TOS} with zero page indirect location	N	Y	Y
<i>siz eor (zp)</i>	<i>eor.w (zp)</i>	Bitwise XOR A _{TOS} with zero page indirect location	N	Y	Y
<i>siz adc (zp)</i>	<i>adc.w (zp)</i>	Add with Carry A _{TOS} with zero page indirect location	N	Y	Y
<i>siz sta (zp)</i>	<i>sta.w (zp)</i>	Store A _{TOS} to with zero page indirect location	N	Y	Y
<i>siz lda (zp)</i>	<i>lda.w (zp)</i>	Load A _{TOS} from with zero page indirect location	N	Y	Y
<i>siz cmp (zp)</i>	<i>cmp.w (zp)</i>	Compare A _{TOS} with zero page indirect location	N	Y	Y
<i>siz sbc (zp)</i>	<i>sbc.w (zp)</i>	Subtract zero page indirect location from A _{TOS}	N	Y	Y

Table 26: Effect of *isz* on 6502/65C02/M65C02A zp indirect Instructions.

Sequence	Alt. Mnemonic	Description	OSX	OAX	OAY
<i>isz anl (zp)</i>	<i>anl.w ((zp))</i>	Bitwise AND A _{TOS} with zero page double indirect location	N	Y	Y
<i>isz ora (zp)</i>	<i>ora.w ((zp))</i>	Bitwise OR A _{TOS} with zero page double indirect location	N	Y	Y
<i>isz eor (zp)</i>	<i>eor.w ((zp))</i>	Bitwise XOR A _{TOS} with zero page double indirect location	N	Y	Y
<i>isz adc (zp)</i>	<i>adc.w ((zp))</i>	Add with Carry A _{TOS} with zero page double indirect location	N	Y	Y
<i>isz sta (zp)</i>	<i>sta.w ((zp))</i>	Store A _{TOS} to with zero page double indirect location	N	Y	Y
<i>isz lda (zp)</i>	<i>lda.w ((zp))</i>	Load A _{TOS} from with zero page double indirect location	N	Y	Y
<i>isz cmp (zp)</i>	<i>cmp.w ((zp))</i>	Compare A _{TOS} with zero page double indirect location	N	Y	Y
<i>isz sbc (zp)</i>	<i>sbc.w ((zp))</i>	Subtract zero page double indirect location from A _{TOS}	N	Y	Y

3.6.2. Effect of the *osx/oax/oay* Prefix Instructions

The OSX register override prefix flag has no effect on 6502/65C02 instructions using zero page indirect addressing mode. The OAX, and OAY register override prefix flags have the expected effect on the destination register. These prefix instructions can be combined with the *ind/siz/isz* prefix instructions. The effect of the *oax/oay* prefix instructions on 6502/65C02 instructions using zero page indirect addressing mode is shown in the following two tables:

Table 27: Effect of *oax* on 6502/65C02 zp indirect instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oax anl (zp)</i>	<i>anl.x (zp)</i>	Bitwise AND X _{TOS} with zero page indirect location	Y	Y
<i>oax ora (zp)</i>	<i>ora.x (zp)</i>	Bitwise OR X _{TOS} with zero page indirect location	Y	Y
<i>oax eor (zp)</i>	<i>eor.x (zp)</i>	Bitwise XOR X _{TOS} with zero page indirect location	Y	Y
<i>oax adc (zp)</i>	<i>adc.x (zp)</i>	Add with Carry X _{TOS} with zero page indirect location	Y	Y
<i>oax sta (zp)</i>	<i>sta.x (zp)</i>	Store X _{TOS} to with zero page indirect location	Y	Y
<i>oax lda (zp)</i>	<i>lda.x (zp)</i>	Load X _{TOS} from with zero page indirect location	Y	Y
<i>oax cmp (zp)</i>	<i>cmp.x (zp)</i>	Compare X _{TOS} with zero page indirect location	Y	Y
<i>oax sbc (zp)</i>	<i>sbc.x (zp)</i>	Subtract zero page indirect location from X _{TOS}	Y	Y

Table 28: Effect of *oay* on 6502/65C02 zp indirect instructions.

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oay anl (zp)</i>	<i>anl.y (zp)</i>	Bitwise AND Y _{TOS} with zero page indirect location	Y	Y
<i>oay ora (zp)</i>	<i>ora.y (zp)</i>	Bitwise OR Y _{TOS} with zero page indirect location	Y	Y
<i>oay eor (zp)</i>	<i>eor.y (zp)</i>	Bitwise XOR Y _{TOS} with zero page indirect location	Y	Y
<i>oay adc (zp)</i>	<i>adc.y (zp)</i>	Add with Carry Y _{TOS} with zero page indirect location	Y	Y
<i>oay sta (zp)</i>	<i>sta.y (zp)</i>	Store Y _{TOS} to with zero page indirect location	Y	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 65 OF 92

Sequence	Alt. Mnemonic	Description	IND	SIZ
<i>oay lda (zp)</i>	<i>lda.y (zp)</i>	Load Y _{TOS} from with zero page indirect location	Y	Y
<i>oay cmp (zp)</i>	<i>cmp.y (zp)</i>	Compare Y _{TOS} with zero page indirect location	Y	Y
<i>oay sbc (zp)</i>	<i>sbc.y (zp)</i>	Subtract zero page indirect location from Y _{TOS}	Y	Y

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

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]



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 66 OF 92

3.17. Stack Pointer Relative [sp,S]

3.18. Post-Indexed Stack Pointer Relative Indirect [(sp,S),Y]

3.19. Base Pointer Relative [bp,B]

3.20. IP-relative with Auto-increment [ip,I++]



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 67 OF 92

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. M65C02A Opcode Tables

The following tables define the opcodes for the instruction set of the M65C02A core. Table 30 and

	0	1	2	3	4	5	6	7
0	BRK #imm	ORA (zp,X)	COP #imm	ORA sp,S	TSB zp	ORA zp	ASL zp	ORA bp,B
1	BPL rel	ORA (zp),Y	ORA (zp)	ORA (sp,S),Y	TRB zp	ORA zp,X	ASL zp,X	ASL bp,B
2	JSR abs	AND (zp,X)	ADJ #imm	AND sp,S	BIT zp	AND zp	ROL zp	AND bp,B
3	BMI rel	AND (zp),Y	AND (zp)	AND (sp,S),Y	BIT zp,X	AND zp,X	ROL zp,X	ROL bp,B
4	RTI	EOR (zp,X)	XMA sp,S	EOR sp,S	MOV #imm	EOR zp	LSR zp	EOR bp,B
5	BVC rel	EOR (zp),Y	EOR (zp)	EOR (sp,S),Y	PSH #imm	EOR zp,X	LSR zp,X	LSR sp,B
6	RTS	ADC (zp,X)	ADD ip,I++	ADC sp,S	STZ zp	ADC zp	ROR zp	ADC bp,B
7	BVS rel	ADC (zp),Y	ADC (zp)	ADC (sp,S),Y	STZ zp,X	ADC zp,X	ROR zp,X	ROR bp,B
8	BRA rel	STA (zp,X)		STA sp,S	STY zp	STA zp	STX zp	STA bp,B
9	BCC rel	STA (zp),Y	STA (zp)	STA (sp,S),Y	STY zp,X	STA zp,X	STX zp,Y	TSB bp,B
A	LDY #imm	LDA (zp,X)	LDX #imm	LDA sp,S	LDY zp	LDA zp	LDX zp	LDA bp,B
B	BCS rel	LDA (zp),Y	LDA (zp)	LDA (bp,S),Y	LDY zp,X	LDA zp,X	LDX zp,Y	TRB bp,B
C	CPY #imm	CMP (zp,X)	STA ip,I++	CMP sp,S	CPY zp	CMP zp	DEC zp	CMP bp,S
D	BNE rel	CMP (zp),Y	CMP (zp)	CMP (sp,S),Y	PSH zp	CMP zp,X	DEC zp,X	DEC bp,B
E	CPX #imm	SBC (zp,X)	LDA ip,I++	SBC sp,S	CPX zp	SBC zp	INC zp	SBC bp,B
F	BEQ rel	SBC (zp),Y	SBC (zp)	SBC (sp,S),Y	PUL zp	SBC zp,X	INC zp,X	INC bp,B

Table 31 provide all of the instruction opcodes for an HLL-enhanced instruction set for the M65C02A core. The 32 Rockwell bit-oriented instructions have been replaced by stack pointer, base pointer, and FORTH VM IP relative instructions. These instructions enhance the way that HLL are able to access working variables, i.e. temporary values and pointers, and local variables on the stack frame. The IP-relative instructions may also be used by HLL as auto-incrementing pointers into the full address space of the M65C02A core. Table 32 and Table 33 provide all of the instruction opcodes for a instruction set for the M65C02A core compatible with the instruction set of the WDC W65C02S microprocessor. It includes the 32 Rockwell bit-oriented instructions that the HLL-enhanced version of the instruction set replaces.

The following table provides a legend by which the M65C02A instruction set given in these four tables can be evaluated quickly.


	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	CAGE CODE	DRAWING NUMBER	REV
		A		1004-0900	-
		SCALE: NONE		SHEET 68 OF 92	

Table 29: Legend for M65C02A Instruction Set Tables.

Legend	Meaning
	Indirection and size prefix instructions have no affect
	Only supports indirection prefix instructions
	Only supports size prefix instructions
	Supports indirection and size prefix instructions
	Unimplemented/Untested
	Reserved for future use
TEXT	Instructions in original 6502
TEXT	Instructions added by 65C02
TEXT	Instructions added by WDC W65C02S (Rockwell 65C02 bit-oriented instructions plus W65C816 WAI and STP)
TEXT	Instructions added by 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 69 OF 92

Table 30: Columns 0 – 7 HLL-Optimized M65C02A Opcode Table.

	0	1	2	3	4	5	6	7
0	BRK #imm	ORA (zp,X)	COP #imm	ORA sp,S	TSB zp	ORA zp	ASL zp	ORA bp,B
1	BPL rel	ORA (zp),Y	ORA (zp)	ORA (sp,S),Y	TRB zp	ORA zp,X	ASL zp,X	ASL bp,B
2	JSR abs	AND (zp,X)	ADJ #imm	AND sp,S	BIT zp	AND zp	ROL zp	AND bp,B
3	BMI rel	AND (zp),Y	AND (zp)	AND (sp,S),Y	BIT zp,X	AND zp,X	ROL zp,X	ROL bp,B
4	RTI	EOR (zp,X)	XMA sp,S	EOR sp,S	MOV #imm	EOR zp	LSR zp	EOR bp,B
5	BVC rel	EOR (zp),Y	EOR (zp)	EOR (sp,S),Y	PSH #imm	EOR zp,X	LSR zp,X	LSR sp,B
6	RTS	ADC (zp,X)	ADD ip,I++	ADC sp,S	STZ zp	ADC zp	ROR zp	ADC bp,B
7	BVS rel	ADC (zp),Y	ADC (zp)	ADC (sp,S),Y	STZ zp,X	ADC zp,X	ROR zp,X	ROR bp,B
8	BRA rel	STA (zp,X)		STA sp,S	STY zp	STA zp	STX zp	STA bp,B
9	BCC rel	STA (zp),Y	STA (zp)	STA (sp,S),Y	STY zp,X	STA zp,X	STX zp,Y	TSB bp,B
A	LDY #imm	LDA (zp,X)	LDX #imm	LDA sp,S	LDY zp	LDA zp	LDX zp	LDA bp,B
B	BCS rel	LDA (zp),Y	LDA (zp)	LDA (bp,S),Y	LDY zp,X	LDA zp,X	LDX zp,Y	TRB bp,B
C	CPY #imm	CMP (zp,X)	STA ip,I++	CMP sp,S	CPY zp	CMP zp	DEC zp	CMP bp,S
D	BNE rel	CMP (zp),Y	CMP (zp)	CMP (sp,S),Y	PSH zp	CMP zp,X	DEC zp,X	DEC bp,B
E	CPX #imm	SBC (zp,X)	LDA ip,I++	SBC sp,S	CPX zp	SBC zp	INC zp	SBC bp,B
F	BEQ rel	SBC (zp),Y	SBC (zp)	SBC (sp,S),Y	PUL zp	SBC zp,X	INC zp,X	INC bp,B

Table 31: Columns 8 – 15 HLL-Optimized M65C02A Opcode Table.

	8	9	A	B	C	D	E	F
0	PHP	ORA #imm	ASL A	DUP	TSB abs	ORA abs	ASL abs	ORA ip,I++
1	CLC	ORA abs,Y	INC A	SWP	TRB abs	ORA abs,X	ASL abs,X	ASL ip,I++
2	PLP	AND #imm	ROL A	ROT	BIT abs	AND abs	ROL abs	AND ip,I++
3	SEC	AND abs,Y	DEC A	NXT	BIT abs,X	AND abs,X	ROL abs,X	ROL ip,I++
4	PHA	EOR #imm	LSR A	PHI	JMP abs	EOR abs	LSR abs	EOR ip,I++
5	CLI	EOR abs,Y	PHY	INI	PHR rel16	EOR abs,X	LSR abs,X	LSR ip,I++
6	PLA	ADC #imm	ROR A	PLI	JMP (abs)	ADC abs	ROR abs	
7	SEI	ADC abs,Y	PLY	ENT	JMP (abs,X)	ADC abs,X	ROR abs,X	ROR ip,I++
8	DEY	BIT #imm	TXA	OSX	STY abs	STA abs	STX abs	
9	TYA	STA abs,Y	TXS	IND	STZ abs	STA abs,X	STZ abs,X	TSB ip,I++
A	TAY	LDA #imm	TAX	SIZ	LDY abs	LDA abs	LDX abs	
B	CLV	LDA abs,Y	TSX	ISZ	LDY abs,X	LDA abs,X	LDX abs,Y	TRB ip,I++
C	INY	CMP #imm	DEX	WAI	CPY abs	CMP abs	DEC abs	CMP ip,I++
D	CLD	CMP abs,Y	PHX	STP	PSH abs	CMP abs,X	DEC abs,X	DEC ip,I++
E	INX	SBC #imm	NOP	OAX	CPX abs	SBC abs	INC abs	SUB ip,I++
F	SED	SBC abs,Y	PLX	OAY	PUL abs	SBC abs,X	INC abs,X	INC ip,I++



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 70 OF 92

Table 32: Columns 0 – 7 W65C02S-Compatible M65C02A Opcode Table.

	0	1	2	3	4	5	6	7
0	BRK #imm	ORA (zp,X)	COP #imm	ORA sp,S	TSB zp	ORA zp	ASL zp	RMB0 zp
1	BPL rel	ORA (zp),Y	ORA (zp)	ORA (sp,S),Y	TRB zp	ORA zp,X	ASL zp,X	RMB1 zp
2	JSR abs	AND (zp,X)	ADJ #imm	AND sp,S	BIT zp	AND zp	ROL zp	RMB2 zp
3	BMI rel	AND (zp),Y	AND (zp)	AND (sp,S),Y	BIT zp,X	AND zp,X	ROL zp,X	RMB3 zp
4	RTI	EOR (zp,X)	XMA sp,S	EOR sp,S	MOV #imm	EOR zp	LSR zp	RMB4 zp
5	BVC rel	EOR (zp),Y	EOR (zp)	EOR (sp,S),Y	PSH #imm	EOR zp,X	LSR zp,X	RMB5 zp
6	RTS	ADC (zp,X)	ADD ip,I++	ADC sp,S	STZ zp	ADC zp	ROR zp	RMB6 zp
7	BVS rel	ADC (zp),Y	ADC (zp)	ADC (sp,S),Y	STZ zp,X	ADC zp,X	ROR zp,X	RMB7 zp
8	BRA rel	STA (zp,X)		STA sp,S	STY zp	STA zp	STX zp	SMB0 zp
9	BCC rel	STA (zp),Y	STA (zp)	STA (sp,S),Y	STY zp,X	STA zp,X	STX zp,Y	SMB1 zp
A	LDY #imm	LDA (zp,X)	LDX #imm	LDA sp,S	LDY zp	LDA zp	LDX zp	SMB2 zp
B	BCS rel	LDA (zp),Y	LDA (zp)	LDA (bp,S),Y	LDY zp,X	LDA zp,X	LDX zp,Y	SMB3 zp
C	CPY #imm	CMP (zp,X)	STA ip,I++	CMP sp,S	CPY zp	CMP zp	DEC zp	SMB4 zp
D	BNE rel	CMP (zp),Y	CMP (zp)	CMP (sp,S),Y	PSH zp	CMP zp,X	DEC zp,X	SMB5 zp
E	CPX #imm	SBC (zp,X)	LDA ip,I++	SBC sp,S	CPX zp	SBC zp	INC zp	SMB6 zp
F	BEQ rel	SBC (zp),Y	SBC (zp)	SBC (sp,S),Y	PUL zp	SBC zp,X	INC zp,X	SMB7 zp

Table 33: Columns 8 – 15 W65C02S-Compatible M65C02A Opcode Table.

	8	9	A	B	C	D	E	F
0	PHP	ORA #imm	ASL A	DUP	TSB abs	ORA abs	ASL abs	BBR0 zp,rel
1	CLC	ORA abs,Y	INC A	SWP	TRB abs	ORA abs,X	ASL abs,X	BBR1 zp,rel
2	PLP	AND #imm	ROL A	ROT	BIT abs	AND abs	ROL abs	BBR2 zp,rel
3	SEC	AND abs,Y	DEC A	NXT	BIT abs,X	AND abs,X	ROL abs,X	BBR3 zp,rel
4	PHA	EOR #imm	LSR A	PHI	JMP abs	EOR abs	LSR abs	BBR4 zp,rel
5	CLI	EOR abs,Y	PHY	INI	PHR rel16	EOR abs,X	LSR abs,X	BBR5 zp,rel
6	PLA	ADC #imm	ROR A	PLI	JMP (abs)	ADC abs	ROR abs	BBR6 zp,rel
7	SEI	ADC abs,Y	PLY	ENT	JMP (abs,X)	ADC abs,X	ROR abs,X	BBR7 zp,rel
8	DEY	BIT #imm	TXA	OSX	STY abs	STA abs	STX abs	BBS0 zp,rel
9	TYA	STA abs,Y	TXS	IND	STZ abs	STA abs,X	STZ abs,X	BBS1 zp,rel
A	TAY	LDA #imm	TAX	SIZ	LDY abs	LDA abs	LDX abs	BBS2 zp,rel
B	CLV	LDA abs,Y	TSX	ISZ	LDY abs,X	LDA abs,X	LDX abs,Y	BBS3 zp,rel
C	INY	CMP #imm	DEX	WAI	CPY abs	CMP abs	DEC abs	BBS4 zp,rel
D	CLD	CMP abs,Y	PHX	STP	PSH abs	CMP abs,X	DEC abs,X	BBS5 zp,rel
E	INX	SBC #imm	NOP	OAX	CPX abs	SBC abs	INC abs	BBS6 zp,rel
F	SED	SBC abs,Y	PLX	OAY	PUL abs	SBC abs,X	INC abs,X	BBS7 zp,rel



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 71 OF 92

Applying the prefix instruction characteristics to these four tables provides the best way to interpret the instruction set of the M65C02A core. For example, the address indirection prefix instructions, *ind* and *isz*, convert direct addressing modes into indirect addressing modes. For instructions highlighted in lavender or orange, the effect of the *ind/isz* prefix instructions is to convert the zero page or the direct addressing mode (indexed or not) to an indirect addressing mode. For instructions highlighted in green or orange, the operand size prefix instructions, *siz* and *isz*, promote 8-bit operations to 16-bit operations.

However, there are a number of instructions highlighted in lavender or orange that use an implicit or an accumulator addressing mode. For these instructions, the *ind/isz* prefix instructions do not modifying a direct addressing mode into an indirect addressing mode. Instead, the effect of *ind/isz* is to modify the instruction in a manner unrelated to the addressing mode. Since the effect of *ind/isz* on such instructions cannot be conveyed readily in the context of the tables, the effect will be described in the sub-section dedicated to the base instruction.

An example of an instruction affected in this manner is the *dup* register stack instruction. Its cell in the tables is highlighted orange which implies that *ind*, *siz*, and/or *isz* prefix instructions may be meaningfully applied. The effect of these prefix instructions on the *dup* instruction cannot be determined from the table since the instruction utilizes the implicit addressing mode, and it is a 16-bit only instruction (see Table 3). Therefore, to understand the effect of the *ind/siz/isz* prefix instructions on the *dup* instruction, the subsection on register stack instructions should be consulted, Subsection 4.3.

4.2. Prefix Instructions

This section describes the behavior/effects of the six prefix instructions. Two prefix instructions modify the operand size or add indirection to the addressing mode of an instruction. One prefix instruction simultaneously adds indirection to the addressing mode and changes the operand size of an instruction. Two prefix instructions override the accumulator used by an instruction, and the last prefix instruction overrides the default stack register used by an instruction. Table 34 defines the six prefix instructions, and shows whether it can be combined with other prefix instructions.

Table 34: M65C02A Prefix Instructions.

Prefix Instruction	<i>ind</i>	<i>siz</i>	<i>isz</i>	<i>oax</i>	<i>oay</i>	<i>osx</i>
<i>ind</i>	-	Y	Y	Y	Y	Y
<i>siz</i>	Y	-	Y	Y	Y	Y
<i>isz</i>	Y	Y	-	Y	N	N
<i>oax</i>	Y	Y	Y	-	N	N
<i>oay</i>	Y	Y	Y	N	-	Y
<i>osx</i>	Y	Y	Y	N	Y	-

Prefix instructions can be applied to any instruction at any time. Whether the prefix instruction has an effect is a matter of the implementation of the instruction. Thus, it is possible that a prefix instruction applied to an instruction will not have an effect on the instruction. Instructions unaf-



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 72 OF 92

ected by the *ind*, *siz*, and *isz* prefix instructions are shown in un-highlighted cells in the M65C02A opcode tables: Table 30,

	0	1	2	3	4	5	6	7
0	BRK #imm	ORA (zp,X)	COP #imm	ORA sp,S	TSB zp	ORA zp	ASL zp	ORA bp,B
1	BPL rel	ORA (zp),Y	ORA (zp)	ORA (sp,S),Y	TRB zp	ORA zp,X	ASL zp,X	ASL bp,B
2	JSR abs	AND (zp,X)	ADJ #imm	AND sp,S	BIT zp	AND zp	ROL zp	AND bp,B
3	BMI rel	AND (zp),Y	AND (zp)	AND (sp,S),Y	BIT zp,X	AND zp,X	ROL zp,X	ROL bp,B
4	RTI	EOR (zp,X)	XMA sp,S	EOR sp,S	MOV #imm	EOR zp	LSR zp	EOR bp,B
5	BVC rel	EOR (zp),Y	EOR (zp)	EOR (sp,S),Y	PSH #imm	EOR zp,X	LSR zp,X	LSR sp,B
6	RTS	ADC (zp,X)	ADD ip,I++	ADC sp,S	STZ zp	ADC zp	ROR zp	ADC bp,B
7	BVS rel	ADC (zp),Y	ADC (zp)	ADC (sp,S),Y	STZ zp,X	ADC zp,X	ROR zp,X	ROR bp,B
8	BRA rel	STA (zp,X)		STA sp,S	STY zp	STA zp	STX zp	STA bp,B
9	BCC rel	STA (zp),Y	STA (zp)	STA (sp,S),Y	STY zp,X	STA zp,X	STX zp,Y	TSB bp,B
A	LDY #imm	LDA (zp,X)	LDX #imm	LDA sp,S	LDY zp	LDA zp	LDX zp	LDA bp,B
B	BCS rel	LDA (zp),Y	LDA (zp)	LDA (bp,S),Y	LDY zp,X	LDA zp,X	LDX zp,Y	TRB bp,B
C	CPY #imm	CMP (zp,X)	STA ip,I++	CMP sp,S	CPY zp	CMP zp	DEC zp	CMP bp,S
D	BNE rel	CMP (zp),Y	CMP (zp)	CMP (sp,S),Y	PSH zp	CMP zp,X	DEC zp,X	DEC bp,B
E	CPX #imm	SBC (zp,X)	LDA ip,I++	SBC sp,S	CPX zp	SBC zp	INC zp	SBC bp,B
F	BEQ rel	SBC (zp),Y	SBC (zp)	SBC (sp,S),Y	PUL zp	SBC zp,X	INC zp,X	INC bp,B

Table 31, Table 32, and Table 33.

In the M65C02A core, five (5) internal flag registers are associated with the six (6) prefix instructions. These five flag registers affect the M65C02A core's logic and microprogram, and control whether and how the behavior of M65C02A instructions is modified. These five internal flag registers are: IND, SIZ, OAX, OAY, and OSX. IND is set by the *ind* and *isz* prefix instructions. SIZ is set by the *siz* and *isz* prefix instructions. OAX and OAY are set by the *oax* and *oay* prefix instructions, respectively. OSX is set by the *osx* prefix instruction.

The primary use for the *ind* prefix instruction is to force indirection to be added to any zero page, absolute addressing modes, BP-relative, SP-relative, or IP-relative addressing modes, indexed or non-indexed. The address indirection added is before any indexing is performed. Thus, pre-indexed addressing modes, direct or indirect, are converted to post-indexed single or double indirect addressing modes by the *ind* (or *isz*) prefix instruction.

The secondary use for the *ind* prefix instruction is to generate an extended M65C02A instruction. That is, when *ind* is prefixed to an implicit or accumulator addressing mode instruction, or a branch instruction, address indirection is not added. Instead, the basic instruction is changed into another or another feature/capability of the base instruction is enabled. Examples of this feature of the *ind* prefix instruction will be provided later in this section.



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 73 OF 92

The primary use for the *siz* prefix instruction is to force the ALU operation and operand size to be promoted from 8 bits to 16 bits. A secondary use of the *siz* prefix instruction is to generate an extended M65C02A instruction. The *ind* prefix instruction is used more often for this purpose, but the *siz* prefix instruction is used for this purpose with the conditional branch instructions and with the register stack management instructions.

The primary use for the *isz* prefix instruction is to simultaneously add indirection to the addressing mode of an instruction, and to change the width of ALU and register operations. Like the *ind* and the *siz* prefix instructions, the *isz* prefix instruction is occasionally used to generate extended M65C02A instructions. The *isz* prefix instruction is used for this purpose with the conditional branch instructions and with the register stack management instructions.


When set, OAX or OAY override the role of the accumulator; In other words, the role of the accumulator and the selected index register, X or Y, are exchanged. The index register serves as the accumulator for the instruction, and the accumulator serves as the index register, if required, for the instruction. OAX and OAY are mutually exclusive. Thus, only the last *oax/oay* applied before an instruction will determine the register override.

When set, OSX overrides the default stack pointer register associated with an instruction. For most instructions which use the stacks, the system stack pointer (S_K or S_U) is the default stack pointer. When applied to these instructions, OSX cause the auxiliary stack pointer S_X to be used for the requested stack operation rather than the system stack pointer. Some instructions, i.e. FORTH VM instructions, use the auxiliary pointer S_X as the default stack pointer. For these instructions, OSX causes S_K or S_U to be used instead of S_X . When OSX is set, the stack-relative addressing mode is converted to the base-relative addressing mode. Similarly, when OSX is set, the base-relative addressing mode is converted to the stack-relative addressing mode.

In addition, when OSX is set, the destination register of those instructions dedicated to operating on the X register is replaced by the system stack pointer: the function of X and S are exchanged. This feature of OSX makes it easier to manipulate the system stack pointers. The normal 6502/65C02 instruction that affects the stack pointer directly, *txs*, is still available, and remains the only way to set the user mode system stack pointer S_U while operating the M65C02A core in the kernel mode. However, OSX allows the following instructions to affect the system stack pointer directly: *phx/plx*, and *stx/ldx/inx/dex/cpx*. (**Note:** these instructions, targeting the system stack pointer, when prefixed by *osx*, can be represented by alternate mnemonics: *phs/pls*, *sts/lts/ins/des/cps*.)

When set, OSX **DOES NOT** substitute S for X in indexed addressing modes in the manner that the accumulator takes on the index register role when OAX or OAY are set. Finally, OSX and OAX are mutually exclusive. Thus, only the last *osx/oax* applied before an instruction will control the setting of the OAX and OSX flag registers.

When applied to an instruction which uses the implicit or the accumulator addressing mode, the *ind*, *siz*, and *isz* prefix instructions generate new extended M65C02A instructions. Table 35 lists the implicit/accumulator addressing mode instructions modified by these three prefix in-

	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 74 OF 92	

structions into extended M65C02A instructions, or into instructions with special effects on the ALU status flags.

Table 35: Effects of *ind/siz/isz* Prefix Instructions on Implicit/Accumulator Instructions.

Instruction	<i>ind</i>	<i>siz</i>	<i>isz</i>
<i>dup</i>	<i>tai</i>	<i>tia</i>	<i>xai</i>
<i>swp</i>	<i>bsw</i>		
<i>rot</i>	<i>rev</i>		
<i>phi</i>	<i>phw</i>		
<i>ini</i>	<i>inw</i>		
<i>pli</i>	<i>plw</i>		
<i>asl a</i>	8-bit, NVZC		16-bit, NVZC
<i>inc a</i>	8-bit, NZC		16-bit, NZC
<i>rol a</i>	8-bit, NVZC		16-bit, NVZC
<i>dec a</i>	8-bit, NZC		16-bit, NZC
<i>lsr a</i>	8-bit, NZC, shift MSB(7) (<i>asr a</i>)		16-bit, NZC, shift MSB(15) (<i>asr a</i>)

As can be seen in Table 35, several new instructions are generated by the application of *ind*, *siz*, or *isz*, but they also enable several important changes to the ALU flags of standard instructions. Several notable examples is the enabling the setting of the C (carry) flag for the *inc a* and *dec a* instructions, and the enabling *asl a* and *rol a* to set the V flag. Another notable change is the change of the *lsr a* instruction into an arithmetic right shift instruction, *asr a*. In most instances where the function of an implicit/accumulator addressing mode instruction is changed based on one these prefix instructions, it is the IND flag which generates the changes noted in Table 35.

One of the major failings of the 6502/65C02 instruction set is the unsigned comparison that the compare instructions perform. Coupled with the unsigned comparisons, the 6502/65C02 instruction set does not support signed branches. These two features of the 6502/65C02 instruction set makes implementing the signed arithmetic, as used by most HLLs, more difficult. To alleviate this failing, the M65C02A allows the 16-bit version of the compare instructions (*cmp/cpx/cpy*) to set the V flag, which provides the support needed to implement signed branch instructions. Furthermore, the M65C02A relative branch instructions have been extended to support multi-flag tests which allow signed and unsigned operations. In addition, the range of the relative displacement of the M65C02A relative branch instructions can be increased from the standard 8-bit -128...+127 range to a full 16-bit -32768...+32767 range. Enabling the branch instructions to use a 16-bit relative displacement improves the capability of the M65C02A to support position independent programs. Table 36 tabulates the special effects of the *ind/siz/isz* prefix instructions on the compare and branch instructions:

Table 36: Effects of *ind/siz/isz* on Compare and Branch Instructions.

Instruction	<i>ind</i>	<i>siz</i>	<i>isz</i>
<i>cmp</i>		16-bit, NVZC	16-bit, NVZC
<i>cpx</i>		16-bit, NVZC	16-bit, NVZC
<i>cpy</i>		16-bit, NVZC	16-bit, NVZC
<i>bra rel8</i>	<i>jrl rel16</i>		<i>jrl rel16</i>
<i>bmi rel8</i>	<i>jmi rel16</i>	<i>ble rel8</i> (signed \leq)	<i>jle rel16</i> (signed \leq)
<i>bpl rel8</i>	<i>jpl rel16</i>	<i>bgt rel8</i> (signed $>$)	<i>jgt rel16</i> (signed $>$)
<i>bvs rel8</i>	<i>jvs rel16</i>	<i>blt rel8</i> (signed $<$)	<i>jlt rel16</i> (signed $<$)
<i>bvc rel8</i>	<i>jvc rel16</i>	<i>bge rel8</i> (signed \geq)	<i>jge rel16</i> (signed \geq)
<i>beq rel8</i>	<i>jeq rel16</i>	<i>bhs rel8</i> (unsigned \geq)	<i>jhs rel16</i> (unsigned \geq)
<i>bne rel8</i>	<i>jne rel16</i>	<i>blo rel8</i> (unsigned $<$)	<i>jlo rel16</i> (unsigned $<$)
<i>bcs rel8</i>	<i>jcs rel16</i>	<i>bhi rel8</i> (unsigned $>$)	<i>jhi rel16</i> (unsigned $>$)
<i>bcc rel8</i>	<i>jcc rel16</i>	<i>bls rel8</i> (unsigned \leq)	<i>jls rel16</i> (unsigned \leq)

As can be seen in Table 36, the effect of the IND flag is to extend the range of the relative branch from 8 bits to 16 bits; with IND flag set, the branch instructions' relative addressing mode is **NOT** converted to a relative indirect addressing mode. The SIZ flag modifies the eight conditional branch instructions to support multi-flag signed and unsigned conditional branches: 4 signed multi-flag and 4 unsigned multi-flag conditional branches.

4.3. Register Stack Instructions

This section describes the instructions that manipulate the register stacks. Table 37 defines the instructions in this category. These register stack instructions are all single byte, single cycle, 16-bit instructions. 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.

The register stack instructions listed in Table 37 support the register override prefix instructions, *oax* and *oay*, but only when not additionally prefixed by *ind*, *siz*, or *isz*. That is, *dup*, *swp*, and *rot* operations are supported for the X and Y register stacks, but the extended operations enabled by *ind*, *siz*, and *isz* are only available for the A register stack.

Table 37: Register Stack Instructions.

Mnemonic	Description	Opcode	# Cycles	Operation					
				Normal	IND	SIZ	OSX	OAX	OAY
<i>dup</i>	Duplicate TOS	0B	1	{TOS, TOS, NOS}	Y	Y	N	Y	Y
<i>swp</i>	Swap TOS and NOS	1B	1	{NOS, TOS, BOS}	Y	N	N	Y	Y
<i>rot</i>	Rotate Stack	2B	1	{NOS, BOS, TOS}	Y	N	N	Y	Y

Table 38 defines the resulting extended register stack instructions, which are only available for the A Register Stack.

Table 38: Extended Register Stack Instructions – A Register Stack Only.

Mnemonic	Prefix Seq	Description	Opcode	# Cycles	Operation
<i>tai</i>	<i>ind dup</i>	Transfer A _{TOS} to IP	9B0B	2	IP <= A _{TOS}
<i>tia</i>	<i>siz dup</i>	Transfer IP to A _{TOS}	AB1B	2	A _{TOS} <= IP
<i>xai</i>	<i>isz dup</i>	Exchange A _{TOS} and IP	BB2B	2	IP <= A _{TOS} ; A _{TOS} <= IP
<i>bsw</i>	<i>ind swp</i>	Byte Swap A	9B1B	2	A _{TOS} [15:0] <= {A _{TOS} [7:0], A _{TOS} [15:8]}
<i>rev</i>	<i>ind rot</i>	Reverse A	9B2B	2	A _{TOS} [15:0] <= A _{TOS} [0:15]

4.4. FORTH VM Instructions

The section describes the instructions for the FORTH VM. Table 39 defines the instructions in this category. These FORTH VM instructions are all single byte instructions. The FORTH VM instructions do not affect the ALU flags in P: N, V, Z, C. Push/pull of the IP/W register defaults to the RSP, which is assigned to the auxiliary stack pointer, S_X or X_{TOS}. The default stack pointer for these operations can be overridden using the *osx* prefix instruction. The other two register override prefix instructions, *oax* and *oay*, have no effect on these instructions and are ignored if prefixed to these instructions.

The *siz* prefix instruction is ignored as is the SIZ flag if set by the *isz* prefix instruction. The *ind* prefix instruction, and the IND flag if set by the *isz* prefix instruction, have an effect on all of the instructions listed in Table 39. The resulting extended FORTH VM instructions are described in Table 40.

Table 39: FORTH VM Instructions.

Mnemonic	Description	Opcode	# Cycles	Operation					
				Normal	IND	SIZ	OSX	OAX	OAY
<i>nxt</i>	Next word	3B	3	See 4.4.1	Y	N	Y	N	N
<i>phi</i>	Push IP	4B	3	(RSP--) <= IP	Y	N	Y	N	N
<i>ini</i>	Increment IP	5B	1	IP <= IP + 1	Y	N	Y	N	N
<i>pli</i>	Pull/Pop IP	6B	3	IP <= (++RSP)	Y	N	Y	N	N
<i>ent</i>	Enter word	7B	1	See 4.4.2	Y	N	Y	N	N

Table 40 defines the extended FORTH VM instructions:

Table 40: Extended FORTH VM Instructions.

Mnemonic	Prefix Seq	Description	Opcode	# Cycles	Operation
<i>phw</i>	<i>ind phi</i>	Push W	9B4B	4	(RSP--) <= W
<i>inw</i>	<i>ind ini</i>	Increment W	9B5B	2	W <= W + 1
<i>plw</i>	<i>ind dup</i>	Pull/Pop W	9B6B	4	W <= (++RSP)

4.4.1. Operation of *nxt* Instruction

A threaded code FORTH VM requires an inner interpreter to advance IP through the threaded code in a manner similar to the PC-indirect instruction fetch, PC auto-increment cycle of a normal processor. The *nxt* instruction provides this operation for a threaded code implementation of the FORTH VM, i.e. *nxt* is the inner interpreter.



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 77 OF 92

When used as is, *nxt* implements the inner interpreter of a Direct Threaded Code (DTC) FORTH VM. When prefixed by *ind* (or *isz*), *nxt* implements the inner interpreter of an Indirect Threaded Code (ITC) FORTH VM. Table 41 describes the actions performed by *nxt* for these two FORTH VM implementations.

Table 41: Operation of *nxt* in ITC and DTC Threaded Code FORTH VM.

ITC (<i>ind nxt</i>)		DTC (<i>nxt</i>)	
W <= (IP++)	Load *Code_Fld	W <= (IP++)	Load *Code_Fld
PC <= (W)	Jump Indirect	PC <= W	Jump Direct

4.4.2. Operation of *ent* Instruction

The FORTH VM follows linked chain of FORTH words, i.e. a threaded list of routines. For a DTC FORTH VM, the code field to which IP points is composed of a two byte branch into the parameter field if the word is a primary FORTH word, or it is an *ent* instruction if the word is a secondary FORTH word. For an ITC FORTH VM, the code field points into the parameter field if the word is a primary FORTH word, or to a common *ent* instruction if the word is a secondary FORTH word; this common *ent* instruction allows the ITC FORTH VM to walk the linked list through a second level of indirection. Below describes the operation of the *ent* FORTH VM instruction.

Table 42: Operation of *ent* in ITC and DTC Threaded Code FORTH VM.

ITC (<i>ind ent</i>)		DTC (<i>ent</i>)	
(RSP++) <= IP	Save current IP	(RSP++) <= IP	Save current IP
IP <= W	Load IP	IP <= W	Load IP
W <= (IP++)	Load *Code_Fld	W <= (IP++)	Load *Code_Fld
PC <= (W)	Jump Indirect	PC <= W	Jump Direct

As can be seen from examining the pseudo code definition in Table 42 and comparing it to Table 41, the *ent* instruction is essentially a push of the IP onto the RS followed by *nxt*. In the microprogram, the microroutine implementing the *ent* instruction falls through to the microroutine implementing the *nxt* instruction.

4.4.3. Other Common FORTH Primitives

Many common FORTH primitives are easily implemented using the M65C02A instruction set:

Table 43: Examples of Common FORTH Primitives Using M65C02A Instruction Set.

FORTH Primitive	M65C02A Sequence	Alt Mnemonics	Comments
EXIT	<i>pli</i> <i>nxt</i>	<i>pli</i> <i>nxt</i>	IP <= (++RSP) See Table 41
DUP	<i>siz lda 0,s</i> <i>siz pha</i>	<i>lda.w 0,s</i> <i>pha.w</i>	A _{TOS} <= (PSP+0) (PSP--) <= A _{TOS}



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 78 OF 92

FORTH Primitive	M65C02A Sequence	Alt Mnemonics	Comments
SWAP	<i>siz lda 2,s</i> <i>siz xma 0,s</i> <i>siz sta 2,s</i>	<i>lda.w 2,s</i> <i>xma.w 0,s</i> <i>sta.w 2,s</i>	A _{TOS} <= (PSP+1) A _{TOS} <= (PSP+0) (PSP+1) <= A _{TOS}
ROT	<i>siz lda 4,s</i> <i>siz xma 2,s</i> <i>siz xma 0,s</i> <i>siz sta 4,s</i>	<i>lda.w 4,s</i> <i>xma.w 2,s</i> <i>xma.w 0,s</i> <i>sta.w 4,s</i>	A _{TOS} <= (PSP+2) A _{TOS} <= (PSP+1) <= A _{TOS} A _{TOS} <= (PSP+0) <= A _{TOS} (PSP+2) <= A _{TOS}
R>	<i>osx siz pla</i> <i>siz pha</i>	<i>pla.sw</i> <i>pha.w</i>	A _{TOS} <= (++RSP) (PSP--) <= A _{TOS}
>R	<i>siz pla</i> <i>osx siz pha</i>	<i>pla.w</i> <i>pha.sw</i>	A _{TOS} <= (++PSP) (RSP--) <= A _{TOS}
+	<i>siz pla</i> <i>clc</i> <i>siz adc 0,s</i> <i>siz sta 0,s</i>	<i>pla.w</i> <i>clc</i> <i>adc.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= (++PSP) Clear Carry A _{TOS} <= A _{TOS} + (PSP) (PSP) <= A _{TOS}
-	<i>siz pla</i> <i>sec</i> <i>siz sbc 0,s</i> <i>siz sta 0,s</i>	<i>pla.w</i> <i>sec</i> <i>sbc.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= (++PSP) Set Carry A _{TOS} <= A _{TOS} - (PSP) (PSP) <= A _{TOS}
NEG	<i>lda #\$00</i> <i>sec</i> <i>siz sbc 0,s</i> <i>siz sta 0,s</i>	<i>lda #\$00</i> <i>sec</i> <i>sbc.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= #\$0000 Set Carry A _{TOS} <= A _{TOS} - (PSP) (PSP) <= A _{TOS}
AND	<i>siz pla</i> <i>siz anl 0,s</i> <i>siz sta 0,s</i>	<i>pla.w</i> <i>anl.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= (++PSP) A _{TOS} <= A _{TOS} & (PSP) (PSP) <= A _{TOS}
OR	<i>siz pla</i> <i>siz ora 0,s</i> <i>siz sta 0,s</i>	<i>pla.w</i> <i>ora.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= (++PSP) A _{TOS} <= A _{TOS} (PSP) (PSP) <= A _{TOS}
XOR	<i>siz pla</i> <i>siz eor 0,s</i> <i>siz sta 0,s</i>	<i>pla.w</i> <i>eor.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= (++PSP) A _{TOS} <= A _{TOS} ^ (PSP) (PSP) <= A _{TOS}
NOT	<i>siz lda #\$FFFF</i> <i>siz eor 0,s</i> <i>siz sta 0,s</i>	<i>lda.w #\$FFFF</i> <i>eor.w 0,s</i> <i>sta.w 0,s</i>	A _{TOS} <= #\$FFFF A _{TOS} <= A _{TOS} ^ (PSP) (PSP) <= A _{TOS}
LIT	<i>siz lda 0,i++</i> <i>siz pha</i>	<i>lda.w 0,i++</i> <i>pha.w</i>	A _{TOS} <= (IP++) (16-bit) (PSP++) <= A _{TOS} (16-bit)
CLIT	<i>lda 0,i++</i> <i>siz pha</i>	<i>lda 0,i++</i> <i>pha.w</i>	A _{TOS} <= (IP++) (unsigned 8-bit) (PSP++) <= A _{TOS} (16-bit)
EXECUTE	<i>osx pli</i> <i>[ind] nxt</i>	<i>pli.s</i> <i>nxt.i</i>	IP <= (++PSP) Next [w/ or w/o indirection]
BRANCH	<i>isz dup</i> <i>siz add 0,i++</i> <i>isz dup</i>	<i>xai</i> <i>add.w 0,i++</i> <i>xai</i>	A _{TOS} ⇔ IP A _{TOS} += (IP++) A _{TOS} ⇔ IP



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 79 OF 92

FORTH Primitive	M65C02A Sequence	Alt Mnemonics	Comments
OBRANCH	<i>siz pla</i> <i>beq BRANCH+2</i> <i>siz lda 0,i++</i>	<i>pla.w</i> <i>beq BRANCH+2</i> <i>lda.w 0,i++</i>	ATOS <= (PSP++) Do unconditional branch if A _{TOS} <> 0, skip branch offset

4.5. Stack Instructions

4.6. Other M65C02A-Unique Instructions

This section describes instructions not previously described.

4.7. Accumulator and Memory Instructions

4.7.1. Loads, Stores, and Transfers

4.7.2. Logical Operations

4.7.3. Shift and Rotates

4.7.4. Arithmetic Operations

4.8. Program Control Instructions

4.8.1. Branches

4.8.2. Jumps



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 80 OF 92

4.8.3. Subroutine Calls and Returns

4.8.4. Traps and Interrupt Handling



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 81 OF 92	

5. Boot Loader 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 82 OF 92

6. FORTH VM Study

This section provides a description of the study used to determine the support to include in the M65C02A core for a FORTH VM. Specifically, the objective is to document the design decisions made with respect to custom instructions added to the basic M65C02A instruction set in order to provide better support for a FORTH VM than a standard 6502/65C02-compatible processor. An objective of the study is to identify the least number of instructions (and dedicated hardware) to be added to the M65C02A instruction set to provide an efficient FORTH VM.

The FORTH VM can generally be thought of as being constructed from a minimum of two stacks: (1) a parameter/data stack (PS), and (2) a return stack (RS). The PS is intended to hold all parameters/data used within a program/function, and the RS generally holds the return addresses of FORTH program words. In addition to holding FORTH program word addresses, the RS is also used to hold loop addresses, and may be used to hold parameter/data addresses. These stacks are generally implemented within the memory of whatever microprocessor is hosting the FORTH VM. To support an efficient implementation of the FORTH VM, any specific FORTH implementation should provide hardware assisted stack pointers whenever possible.

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

6.1. "Classic" FORTH VM Registers

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

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

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

6.2. General Implementation Approaches for FORTH VMs

There are several generally accepted methods for implementing the FORTH VM. The classic FORTH VM is implemented using a technique known as Indirect Threaded Code (ITC). The next most common approach is a technique known as Direct Threaded Code (DTC). A more re-



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 83 OF 92

cent approach is a technique known as Subroutine Threaded Code (STC). A final, less commonly used approach is a technique known as Token Threaded Code (TTC).

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

For an STC FORTH VM, each FORTH word is called using a native processor call instruction. Thus, there is no need for an IP register, and there is no inner interpreter. A FORTH program simply chains together the various FORTH words using native processor calls. There are two penalties for this simplicity:


- (1) subroutine calls are generally larger than simple address pointers;
- (2) an STC FORTH requires pushing and popping the return stack on entry to and exit from each FORTH word.

Thus, STC FORTH programs may be larger, and the additional push/pop operations performed by the native subroutine calling instructions may not deliver the performance improvements expected. Since an STC FORTH VM relies on the basic instructions of the processor, and the M65C02A provides those instructions, no additional instructions are needed in the M65C02A's instruction set to support an STC FORTH.

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

A TTC/DTC/ITC FORTH VM is composed of two interpreters: (1) an outer interpreter, and (2) an inner interpreter. The outer interpreter is written in FORTH, i.e. it is composed of FORTH words. The outer interpreter provides the means by which FORTH words created. The outer interpreter also provides the immediate execution mode for which many FORTH implementations are known. The outer interpreter's word creation and immediate execution functionality provides the mechanism for self-hosting a complete FORTH system: editor, compiler, and assembler.

The inner interpreter, on the other hand, "executes" FORTH words. The inner interpreter "executes" FORTH words by moving through the threaded code representing the FORTH program. Therefore, a TTC/DTC/ITC FORTH VM spends the majority of its processing time in its inner interpreter. Thus, any decrease in the number of clock cycles required to "execute" a FORTH word will result in a clear increase in the performance of a FORTH program all other things being equal.

	THE 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 84 OF 92	

6.3. Basic Structure of a FORTH Word

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

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

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

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

The fields, Code_Fld and Param_Fld, represent the "executable" portion of a FORTH word. There are two types of FORTH words: (1) secondaries, and (2) primitives. Secondaries are the predominant type of words in a FORTH program. Their Param_Fld doesn't contain any native machine code. Instead, the Param_Fld of FORTH secondaries is simply a list of pointers to the Code_Fld of other FORTH words. Primitives are the FORTH words that perform the actual work of any FORTH program. The Code_Fld of a primitive is a pointer/link to their Param_Fld, which contains the machine code that performs the work the primitive FORTH word is expected to provide. In FORTH, the outer interpreter is used to perform immediate operations, and to construct, define, or compile other FORTH words. As already stated above, the FORTH VM's outer interpreter is generally composed of secondary FORTH words. After all of the FORTH words associated with a FORTH program have been compiled, the outer interpreter simply transfers control to the inner interpreter to "execute" the top-most FORTH word of the program.

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



THE 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 85 OF 92

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

6.4. Usage Frequency of FORTH Words

Table 6.3.1 in Phillip Koopman's "Stack Computers, The Next Wave" (reproduced in Table 44 below) provides a summary of the relative frequency of the most frequently used FORTH words for several FORTH applications:

Table 44: Usage Frequency of Common FORTH Words.

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

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

With the FORTH word frequency data in the preceding table it is easy to assert that the M65C02A instruction set should contain custom instructions for at least NEXT, ENTER, and EX-



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 86 OF 92

IT. The question then becomes to what extent should these operations be supported? In other words, should they be supported by a single instruction each and should they be supported for both ITC and DTC FORTH VMs?

6.5. Operations of the FORTH VM Inner Interpreter

Thus, the DTC/ITC FORTH VM requires three fundamental operations:

- (1) NEXT : fetch the FORTH word addressed by IP, and advance IP;
- (2) ENTER : push IP, load IP with the Code_Fld value, and perform NEXT;
- (3) EXIT : restore IP, and perform NEXT.

Each FORTH word must “fetched” and “executed”; this is the operation performed by NEXT. Each secondary FORTH word must be “entered” or called; this is the operation performed by ENTER. Each FORTH word must transfer control back to the inner interpreter so that the next FORTH word can be “fetched” and “executed”; this operation is performed by EXIT.

These three fundamental operations are very similar to the operations that the host processor performs in executing its machine code:

- (1) NEXT corresponds directly to the normal processor instruction fetch/execute cycle;
- (2) ENTER corresponds directly to a processor subroutine call;
- (3) EXIT corresponds directly to a processor subroutine return.

As previously discussed, FORTH uses two stacks: parameter/data stack and return stack. Thus, ENTER and EXIT save and restore the IP to/from the return stack, respectively. Most processors implement a single hardware stack into which both return addresses and data are written. FORTH maintains strict separation between the parameter/data stack and the return stack because it uses a stack-based arithmetic architecture. Mixing return addresses and parameters/data on a single stack would complicate the passing and processing of parameters. The following pseudo code defines these three operations in terms of the ITC and the DTC models:

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



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 87 OF 92

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

Finally, there are two important observations made by Rodriguez:

- (1) if W is left pointing to the Code_Fld of the word being executed, the Param_Fld of a FORTH word being ENTERed can be found using the value in W;
- (2) providing a second stack pointer for the RS will greatly improve the performance of the inner interpreter.

6.6. Mapping FORTH VM to the M65C02A Core

The preceding analysis and discussions set the stage for the critical design decisions for efficiently support a FORTH VM with the instruction set of the M65C02A:

- (1) Mapping the FORTH VM registers onto the M65C02A registers;
- (2) Modifying the M65C02A to implement the FORTH VM inner interpreter efficiently.

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

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

The PS is used more often than the RS. Thus, it makes more sense, from a speed perspective, to use the native M65C02A stack for the PS. Using a pre-indexed zero page location for the RSP will slow the push and pop operations significantly. Therefore, a better solution would be to use one of the index registers as the RSP, and place the RS anywhere in memory, including page 0 and page 1.

The 6502/65C02 X index register is often used to implement FORTH VM stacks using pre-indexed (direct and indirect) addressing modes. Thus, the X index register is the natural choice to provide the RSP. The auxiliary stack pointer capabilities of the TOS of the X register stack easily allow X to be used as a hardware-assisted RSP.



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 88 OF 92

In addition, two instructions will be added to support pushing and pulling the IP from both stacks, and a single cycle instruction will be added to increment the IP by 1. By overloading the IND prefix instruction, it is possible to use the dedicated IP push, pop, and increment instructions to perform the same operations with the W register.

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

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

VM Register	M65C02A Register
IP	Internal dedicated 16-bit register: IP
W	Internal dedicated 16-bit register: W
PSP	System Stack Pointer (S), allocated in memory (page 1 an option)
RSP	Auxiliary Stack Pointer (X), allocated in memory (page 0 an option)
UP	None – Memory (page 0 an option)
X	None – Not needed, M or MAR can provide any temporary storage required

The FORTH VM will be supported by the M65C02A using five single byte dedicated instructions: **next** (NEXT), **ent** (ENTER), **pli** (Pull IP), **phi** (Push IP), and **ini** (Increment IP by 1). The inner interpreter is implemented by the **next** instruction.

These five instructions can be prefixed by **ind**. The **next** and **ent** instructions directly support a DTC FORTH VM. When prefixed by **ind**, they support an ITC FORTH VM: **ind next**, **ind ent**. The DTC EXIT will be implemented using the **pli next** instruction sequence, and the ITC EXIT will be implemented using the **pli ind next** instruction sequence. Access and control of the IP is provided by the **phi**, **pli**, and **ini** instructions. (When these three instructions are prefixed by **ind**, access and control of W is provided: **phw** (**ind phi**), **plw** (**ind pli**), and **inw** (**ind ini**), respectively.)

6.6.1. Additional Instructions For Supporting FORTH

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



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 89 OF 92

accumulator. If prefixed by *isz*, the word following the current IP is used as a word pointer to load a 16-bit value into the accumulator, while the IP is advanced by 2.

The *lda ip,i++* instruction is matched by the *sta ip,i++* instruction. Without indirection, the *sta ip,i++* instruction will write directly into the FORTH VM instruction stream. With *ind/isz* applied, the resulting *sta (ip,i++)* instruction can be used for directly updating byte/word variables whose pointers are stored directly in the FORTH VM instruction stream. (Although it may be useful when compiling FORTH programs and for creating self-modifying FORTH programs, the *sta ip,i++* instruction is expected to be prefixed with *ind/isz* under normal usage.)

Finally, the *add ip,i++* instruction allows constants (or relative offsets) located at the current IP to be added to the accumulator. Like *lda ip,i++* and *sta ip,i++*, the *add ip,i++* supports the *ind*, *siz*, and *isz* prefix instructions. (Note: the *add ip,i++* instruction does not add the carry bit, so *clc* is not required before this instruction.)

Some consideration was given to directly supporting IP-relative conditional branches for the FORTH VM with the relative branch instructions of the M65C02A. Given that the *add ip,i++* and *lda ip,i++* instructions can use the same micro-sequence, it was decided that separately supporting FORTH branches or jumps was too expensive in microprogram space. IP-relative conditional FORTH branches can be implemented using the following instruction sequence:

<i>[siz] bxx \$1</i>	[2[3]] test xx condition and branch if not true
<i>isz dup</i>	[2] exchange A and IP (<i>xai</i>)
<i>siz add ip,i++</i>	[5] add IP-relative offset to A
<i>isz dup</i>	[2] exchange A and IP (<i>xai</i>)
<i>\$1:</i>	

The IP-relative conditional branch instruction sequence only requires 11[12] clock cycles, and IP-relative jumps require only 9 clock cycles. (**Note:** *the register stack manipulation instructions discussed above have been extended to support exchanging the A top-of-stack register and IP. isz dup exchanges A_{TOS} and IP (xai), siz dup transfers IP into A (tia), and ind dup transfers A into IP (tai).*)

Conditional branches and unconditional jumps to absolute addresses rather than relative addresses can also be easily implemented. A conditional branch to an absolute address can be implemented as follows:

<i>[siz] bxx \$1</i>	[2[3]] test xx condition and branch if not true
<i>siz lda ip,i++</i>	[5] load relative offset and autoincrement IP
<i>ind dup</i>	[2] transfer A to IP (<i>tai</i>)
<i>\$1:</i>	

Thus, a conditional branch to an absolute address requires 9[10] cycles, and the unconditional absolute jump only requires 7 clock cycles. Clearly, if the position independence of IP-relative



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 90 OF 92

branches and jumps is not required, then the absolute address branches and jumps provide greater performance.

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

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

Table 45: Consolidated List of M65C02A-specific Instructions Supporting FORTH VM.

	DTC		ITC	
<i>nxt</i>	1(3)			; NEXT
<i>ent</i>	1(5)			; ENTER/CALL/DOCOLON
<i>pli nxt</i>	2(6)			; EXIT
<i>pli</i>	1(3)			; Pop IP
<i>phi</i>	1(3)			; Push IP
<i>ini</i>	1(1)			; Increment IP
<i>ind nxt</i>			2(6)	; NEXT
<i>ind ent</i>			2(8)	; ENTER/CALL/DOCOLON
<i>pli ind nxt</i>			3(9)	; EXIT
<i>ind pli</i>			2(4)	; <i>plw</i> - Pop W
<i>ind phi</i>			2(4)	; <i>phw</i> - Push W
<i>ind ini</i>			2(2)	; <i>inw</i> - Increment W
<i>lda ip,i++</i>		2(3)		; Load byte from IP++ into A
<i>siz lda ip,i++</i>		3(5)		; Load word from IP++ into A
<i>ind lda ip,i++</i>		3(6)		; Load byte from IP++ indirect into A
<i>isz lda ip,i++</i>		3(7)		; Load word from IP++ indirect into A
<i>sta ip,i++</i>		2(3)		; Store byte in A at IP++
<i>siz sta ip,i++</i>		3(5)		; Store word in A at IP++
<i>ind sta ip,i++</i>		3(6)		; Store byte in A at IP++ indirect
<i>isz sta ip,i++</i>		3(7)		; Store word in A at IP++ indirect
<i>add ip,i++</i>		2(3)		; Add byte from IP++ into A
<i>siz add ip,i++</i>		3(5)		; Add word from IP++ into A
<i>ind add ip,i++</i>		3(6)		; Add byte from IP++ indirect into A
<i>isz add ip,i++</i>		3(7)		; Add word from IP++ indirect into A
<i>ind dup</i>		2(2)		; <i>tai</i> - Transfer A to IP
<i>siz dup</i>		2(2)		; <i>tia</i> - Transfer IP to A
<i>isz dup</i>		2(2)		; <i>xai</i> - Exchange A and IP



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 91 OF 92

6.7. Summary

The M65C02A can be modified to support an efficient implementation of a FORTH VM. Two 16-bit internal registers, IP and W, requiring a minimal amount of logic enable the implementation of efficient FORTH VMs with the M65C02A core. A single byte opcode is all that is needed to implement the inner interpreter of a DTC FORTH VM. Another single byte opcode implements the CALL/ENTER function required to efficiently implement a DTC FORTH. Three additional single byte opcodes allow IP to be pushed to and popped from the RS and to be incremented. The final modification is the inclusion of a several IP-relative with auto-increment addressing mode instructions for loading and storing constants, literals, and variables efficiently, and computing branch offsets: *add/sta/lda ip,i++*.

Many of the enhancements of the M65C02A can also be applied to the implementation of the FORTH VM and FORTH programs, in general. The *siz* prefix instruction's effects greatly simplify the number of instructions needed to implement 16-bit arithmetic and logical operations. The extended branch instructions also simplify the implementation of branch words. The stack-relative instructions and the memory exchange instruction, *xma sp16,s*, provide a mechanism for greatly simplifying the implementation of PS stack operators and stack-based arithmetic and logic operations.

Finally, since the bit-oriented conditional branch instructions are not often used, the M65C02A can be configured to replace the BBRx/BBSx Rockwell instructions with a full complement of instructions using the IP-relative with auto-increment addressing mode:

and/ora/eor/---/---/---/cmp/sub ip,i++
asl/rol/lsl/ror/tsb/trb/dec/inc ip,i++.

The specific instructions needed to support the FORTH VM in the M65C02A core and the additional, more general purpose instructions and addressing modes all combine to make the M65C02A core an ideal host for FORTH.



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 92 OF 92

7. 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 93 OF 92