MEMORANDUM

To: ENGINEERING NOTEBOOK

From: MICHAEL A. MORRIS,

SUBJECT: M65C02 DESIGN DESCRIPTION

Date: 4/22/2012

CC:

INTRODUCTION

This memorandum describes the implementation of a synthesizable implementation of the Western Design Center (WDC) W65C02 microprocessor instruction set architecture. The objective is to develop a fast synthesizable implementation of the W65C02 instruction set architecture using a microprogrammed implementation rather than a hardwired implementation. The M65C02 is intended to emulate the instruction set of the W65C02, but it is not intended as a cycle accurate emulation of the W65C02. That is, the M65C02 will not emulate the number of instruction cycles of the W65C02. The M65C02 will instead use additional resources, such as additional program counter and stack pointer adders/incrementers, and overlapped instruction fetch and execution to reduce the number of clock cycles required to complete the execution of an instruction. The implementation of the M65C02 described in the memorandum will not use multi-port memory, or wider internal and external busses to improve operand fetch performance.

BACKGROUND

In the late 1970s and early 1980s, several Apple Corporation Apple][-series computers utilized the MOS Technology MOS6502 microprocessor, which was fabricated in an NMOS process. WDC and Rockwell produced CMOS variants of the original MOS6502. If employed in an Apple][computer, the enhancements of the WDC W65C02 or the Rockwell R65C02 microprocessors were unused by the software in order to maintain backward compatibility with its computers employing the MOS6502.

The basic architecture of the MOS6502 (and W65C02) microprocessor is similar to the register poor architecture of the Motorola MC6800. (**Note:** *many of MC6800 design team left Motorola to join MOS Technology to implement the MOS6502, and one, William D. Mensch, Jr., left MOS Technology after its purchase by Commodore Semiconductor Group to found WDC to design and implement the W65C02.*). The 6502's 8-bit architecture is built around just five 8-bit programmer-visible registers: an 8-bit accumulator (A), two 8-bit index registers (X, and Y), an 8-bit stack pointer (S), and an 8-bit Processor

Status Word (P) (with only six registered status and control flags). The processor also provides a 16-bit Program Counter (PC) which allows direct addressing of a maximum of 65,536 of 8-bit memory or memory-mapped I/O locations. (*The small number of registers is generally touted as providing the 6502 microprocessor lower interrupt response latency compared to the interrupt response latency of register-rich contemporary microprocessors such as the Intel 8080/8085.*)

The base architecture supported a large number of addressing modes with the base set consisting of 14 (or 15) addressing modes. The WDC W65C02 differs from the MOS6502 in that two (2) new addressing modes and eight (8) new instructions were added. These additions make the instruction set more regular and address the MOS6502's deficiencies with respect to bit manipulation. Rockwell provided four additional bit-oriented instructions (beyond those made by WDC) in its enhanced version of the MOS6502, the R65C02. Several other vendors also provided enhanced versions of the W65C02 as well, but given the limits in the number of unused opcodes, these enhanced processors were not fully compatible with the W65C02.

Compared to its competitors, the MOS6502/W65C02/R65C02 processors provided a small and simple processor core (*approximately 3500 transistors were used in the NMOS MOS6502's implementation*) which allowed the use of lower clock rates to deliver the same performance. Compared to a 4 MHz Intel 8080/8085 or Zilog Z80A, a 1 MHz 6502 delivered the same or better performance. Thus, for the same level of performance, the lower clock speed meant that slower memories could be used. This feature gave the 6502 a noteworthy cost and power advantage over many of its competitors. In addition, the introductory chip price of the MOS6502 was about a tenth of that of its primary competitors: Intel 8080/8085, Zilog Z80A, and the Motorola MC6800. These advantages led to this architecture's initial dominance in personal computers in the late 70s and early 80s. (*Amazingly, long after the 6502 faded from the PC market, the WDC W65C02 continues to be a major player in the embedded market. It is licensed by WDC to several OEMs, and the architecture continues to be included in many kinds of embedded systems such that several billion W65C02s, or variants thereof, have been shipped to date. Reportedly, several hundred million 6502 derivatives are shipped each year.*)

W65C02 MICROPROCESSOR

INSTRUCTION SET

The basic set of 64 instructions is shown in the following table. The enhanced instructions are shown in **bold** font, and the new instructions are shown in a red font. Included in the table is a column which indicates which PSW flags are affected by each instruction. (**Note:** *bit 5 of the PSW is permanently set to logic 1, and bit 4 of the PSW read as logic 0. The PSW's bit 4 is known as the Break flag (B), and it is set only in the PSW pushed onto the stack during a* **BRK** *instruction trap, and by the Push PSW (***PHP***) instruction.*)

Table 1: W65C02 Instruction Set.

Mnemonic	Description	Operation Performed	PSW Flags
NOP	No Operation	PC <= PC + 1	--10----
ADC	Add Memory to Accumulator with Carry	A <= A + M + C	NV10--ZC
SBC	Subtract Memory from Accumulator with Carry	A <= A + ~M + C	NV10--ZC
ORA	OR Accumulator with Memory	A <= A | M	N-10--Z-
AND	AND Accumulator with Memory	A <= A & M	N-10--Z-
EOR	Exclusive OR Accumulator	A <= A ^ M	N-10--Z-
CMP	Compare Accumulator with Memory	A – M	N-10--ZC
STA	Store Accumulator in Memory	M <= A	--10----
LDA	Load Accumulator from Memory	A <= M	N-10--Z-
INC	Increment (Memory/Accumulator)	M/A <= {M | A} + 1	N-10--Z-
DEC	Decrement (Memory/Accumulator)	M/A <= {M | A} – 1	N-10--Z-
ASL	Arithmetic Shift Left (Memory/Accumulator)	{M | A} <= {{M[6:0] | A[6:0]}, 0} C <= {M[7] | A[7]}	N-10--ZC
LSR	Logical Shift Right (Memory/Accumulator)	{M | A} <= {0, {M[7:1] | A[7:1]}) C <= {M[0] | A[0]}	N-10--ZC
ROL	Rotate Left (Memory/Accumulator)	{M | A} <= {{M[6:0] | A[6:0]}, C} C <= {M[7] | A[7]}	N-10--ZC
ROR	Rotate Right (Memory/Accumulator)	{M | A} <= {C, {M[7:1] | A[7:1]}) C <= {M[0] | A[0]}	N-10--ZC
CPX	Compare X with Memory	X – M	N-10--ZC
CPY	Compare Y with Memory	Y – M	N-10--ZC
INX	Increment X	X <= X + 1	N-10--Z-
INY	Increment Y	Y <= Y + 1	N-10--Z-
DEX	Decrement X	X <= X – 1	N-10--Z-
DEY	Decrement Y	Y <= Y – 1	N-10--Z-
STX	Store X to Memory	M <= X	--10----
LDX	Load X from Memory	X <= M	N-10--Z-
STY	Store Y to Memory	M <= Y	--10----
LDY	Load Y from Memory	Y <= M	N-10--Z-
STZ	Store Zero to Memory (Clear Memory)	M <= 0	--10----
BIT	Test Memory Bits with Accumulator Mask	Z <= ~|(M & A); {N,V} <= M[7:6] **Z <= ~|(**imm & A**);**	NV10--Z- **--10--Z-**
TRB	Test/Reset Memory Bits with Accumulator Mask	M <= M & ~A Z <= ~|(M & A)	--10--Z-
TSB	Test/Set Memory Bits with Accumulator Mask	M <= M | A Z <= ~|(M & A)	--10--Z-
TAX	Transfer Accumulator to X	X <= A	N-10--Z-
TAY	Transfer Accumulator to Y	Y <= A	N-10--Z-
TXA	Transfer X to Accumulator	A <= X	N-10--Z-
TYA	Transfer Y to Accumulator	A <= Y	N-10--Z-
TXS	Transfer X to Stack Pointer	S <= X	--10----
TSX	Transfer Stack Pointer to X	X <= S	N-10--Z-
PHP	Push PSW	*(S--)<= P	--11----
PLP	Pull (Pop) PSW	P <= *(++S)	NV10DIZC
PHA	Push Accumulator	*(S--) <= A	--10----
PLA	Pull Accumulator	A <= *(++S)	N-10--Z-
PHX	Push X to Stack	*(S--)<= X	--10----
PLX	Pull X from Stack	X <= *(++S)	N-10--Z-

Mnemonic	Description	Operation Performed	PSW Flags
PHY	Push Y to Stack	*(S--) <= Y	--10----
PLY	Pull Y from Stack	Y <= *(++S)	N-10--Z-
CLC	Clear Carry	C <= 0	--10---**0**
SEC	Set Carry	C <= 1	--10---**1**
CLI	Clear Interrupt Mask (Disable maskable interrupts)	I <= 0	--10-**0**--
SEI	Set Interrupt Mask (Enable maskable interrupts)	I <= 1	--10-**1**--
CLD	Clear Decimal Mode (ADC/SBC in binary mode)	D <= 0	--10**0**---
SED	Set Decimal Mode (ADC/SBC in BCD mode)	D <= 1	--10**1**---
CLV	Clear Overflow flag	V <= 0	-**0**10----
BRK	Break (Software Interrupt)	*(S--) <= PCH *(S--) <= PCL *(S--) <= P (P[4] <= 1) (B flag) PC <= (0xFFFC) (P[3] = 0, P[2] <= 1] (D, I flags)	--**1**1---- --10**01**--
RTI	Return from Interrupt	P <= *(++S) PCL <= *(++S) PCH <= *(++S)	NV10DIZC
JSR	Jump to Subroutine	*(S--) <= PCH *(S--) <= PCL PC <= EA	--10----
RTS	Return from Subroutine	PCL <= *(++S) PCH <= *(++S)	--10----
JMP	Jump	PC <= EA	--10----
BRA	Branch Always (unconditional branch)	PC <= PC + rel	--10----
BPL	Branch if Plus	PC <= (N **?** PC + 1 **:** PC + rel)	--10----
BMI	Branch if Minus	PC <= (N **?** PC + rel **:** PC + 1)	--10----
BVC	Branch if Overflow Clear	PC <= (V **?** PC + 1 **:** PC + rel)	--10----
BVS	Branch if Overflow Set	PC <= (V **?** PC + rel **:** PC + 1)	--10----
BCC	Branch if Carry Clear	PC <= (C **?** PC + 1 **:** PC + rel)	--10----
BCS	Branch if Carry Set	PC <= (C **?** PC + rel **:** PC + 1)	--10----
BNE	Branch if Not Equal to Zero	PC <= (Z **?** PC + 1 **:** PC + rel)	--10----
BEQ	Branch if Equal to Zero	PC <= (Z **?** PC + rel **:** PC + 1)	--10----

These 64 instructions, along with their address modes, are represent 178 distinct opcodes. (*If instructions using the accumulator and implicit addressing modes are considered as separate instructions, then there are 69 distinct instructions using 178 distinct opcodes.*)

Inspection of the table shows a minimalist approach with an accumulator-based processor and a sparse instruction set. Prior to the addition of the STZ instruction by the W65C02, there was no way to clear memory without explicitly loading the accumulator (or an index register) with a 0, and then writing that register to memory or transferring it to the other registers. In addition, the BRA branch instruction added by the W65C02 allowed unconditional branches. Before this instruction was added by the W65C02, unconditional branches were implemented by explicitly setting one of the four testable PSW condition code flags, and then executing a branch on that condition. Another W65C02 enhancement allows the X and Y indexed registers to be directly pushed to or pulled from the stack. Prior to the W65C02 enhancements, saving and restoring the X and Y registers required transferring the registers to the accumulator before pushing the accumulator to the stack

and vice-versa. The W65C02 also enhanced the BIT instruction with an immediate operand. Finally, the W65C02 added two bit manipulation instructions, TRB (Test and Reset Memory Bit) and TSB (Test and Set Memory Bit) to test and set/reset bits in memory based on a mask value in the accumulator.

The Rockwell enhancements added an additional four bit manipulation instructions and a new multiple operand addressing mode to the enhancements provided by the W65C02. Two instructions were added to set/reset a zero page bit, and two instructions were added to test a zero page bit and branch if the bit was set/reset. Each of these instructions explicitly coded the bit number into the instruction itself. Thus, although only four instructions were added by Rockwell, 32 op codes were used for these instructions. If the Rockwell enhancements are included, then the number of instructions increases to 68, and the number of opcodes used increases to 210.

Other enhancements to the W65C02 were made by Commodore Semiconductor Group (CSG). (*CSG purchased MOS Technology, and marketed a number of 6502 based products: Commodore 64 and 128, and the PET line of computers.*) Although never released, the CSG65CE02 included several new instructions and addressing modes. In particular, stack pointer relative addressing was added, several new instructions were added for direct handling of 16-bit quantities, and a register was added to allow the relocation of the direct page. All in all, the 65CE02 enhancements look to be well considered, even if they had to be shoe-horned into the existing opcodes maps. However, since the processor was never formally released commercially, there are some questions regarding reliability of the available documentation regarding the operation of some of the CSG instructions.

WDC founder William D. Mensch, Jr extended the instruction set and processor into the 16-bit domain, while maintaining backward compatibility with the W65C02. These WDC enhancements extended the 8-bit instruction set in a manner similar to that of the CSG65CE02, and provided an emulation mode for the 16-bit core, so that the resulting device could be operated in an 8-bit environment with full compatibility with the W65C02. The WDC W65C802 is a pin-compatible upgrade for the W65C02 that provides seamless emulation of the W65C02, and can be easily switched between its 16-bit and 8-bit operating modes to provide additional computational capabilities. The WDC W65C816, although not a pin-compatible 16-bit enhancement, is capable of fully emulating the W65C02 and dynamically switching between 16-bit and 8-bit modes. In the full 16-bit mode, the W65C816 provides access to as much as 16MB of memory (24-bit addresses) using 256 64kB pages.

INTERNAL REGISTERS

The W65C02 processor is an accumulator-based machine with few registers. All of the registers generally perform some dedicated functions. There are five (5) 8-bit user addressable registers. The register model for the W65C02 processor is given in Table 2.

Table 2. W65C02 Programmer-Accessible Registers.

Name	Function	Comments
A	Accumulator	Used for ALU operations except Read-Modify-Write Memory operations
X	Pre-Index Register	Provides pre-indexing of the effective address of memory operands
Y	Post-Index Register	Provides post-indexing of the effective address of memory operands
S	Stack Pointer	Points to the next free location of a LIFO memory stack in page 1
P	Processor Status Word	Holds the ALU result status bits and various processor state control bits
PC	Program Counter	Provides the address of the instruction and operands

The processor status word, P (see Table 2), holds the status bit results of ALU operations and several control bits affecting the state of the processor. The bits in the processor status word are defined in Table 3.

Table 3. W65C02 Processor Status Word.

Bit	ID	Name	Comments
0	C	Carry	Holds the carry result for arithmetic and shift operations. **SEC**/**CLC** instructions explicitly Set/Clear C. Tested by the **BCC**/**BCS** branch instructions.
1	Z	Zero	Indicates that the ALU result value is 0. Since all register transfers (except X to S transfers) operate through the ALU, Z also reflects the state of a destination register. Tested by the **BEQ**/**BNE** branch instructions.
2	I	Interrupt Mask	When set, external IRQ interrupts are masked. Automatically set when IRQ/NMI exception is taken. **SEI**/**CLI** instructions explicitly Set/Clear I.
3	D	Decimal Mode	When set, the adder (except for **INC**/**DEC**/**CMP**) functions in BCD mode. **SED**/**CLD** instructions explicitly Set/Clear D.
4	B	Break	Only set by the **BRK**/**PHP** instructions when the PSW is pushed onto stack.
5	1	Reserved	Reserved. Always reads as logic 1.
6	V	Overflow	Set by **ADC**/**SBC** on arithmetic overflow, and bit 6 of the result of a **BIT** operation. May also be set by the processor's external SO input pin. **CLV** instruction clears V. Tested using the **BVS**/**BVC** branch instructions.
7	N	Negative	Set when MSB of ALU operation is set. Since all register transfers (except X to S transfers) operate through the ALU, N also reflects the state of the MSB of a destination register. Tested by the **BMI**/**BPL** branch instructions.

The M65C02 processor also includes several other registers which are not directly accessible to the programmer:

1). IR – 8-bit register which holds the current instruction opcode;
2). OP1 – 8-bit register which holds the first operand of the instruction (dp, #imm, rel, or abs[0]), or the LSB of the indirect address of the operand;
3). OP2 – 8-bit register which holds the second operand (abs[1]), or the MSB of the indirect address of the operand.

EXCEPTION VECTORS

The W65C02 processor, or any of the other variants, has three exceptions vectors in high memory. When Non-Maskable Interrupts (NMI), Reset (RST), or unmasked maskable

interrupts (IRQ) requests are present, the processor will take an exception at the completion of the current instruction. For the RST exception, the processor simply loads the program counter with the address stored in the reset vector, and begins executing instructions at that point. IRQ and NMI exceptions cause the processor to push the processor state onto the stack, and then load the program counter from the appropriate vector and begin execution at that point. In addition, the I bit, interrupt mask, in the program status word (after the program status word has been pushed to the stack) is set and the D bit, Decimal ALU mode, is cleared. These actions disable (mask) maskable interrupt exceptions in either the NMI or the IRQ exception handlers, and ensure that the ALU operates in a known state, i.e. binary arithmetic mode, during all exceptions.

The **BRK** instruction utilizes the maskable interrupt exception's vector for its handler's address. The processor sets the B bit, Break, in the processor word to indicate that a BRK instruction is the source of the interrupt. In this way, the IRQ handler can process software interrupts (**BRK**) or external hardware interrupts. When a **BRK** instruction is executed, the I bit is set and the D bit is cleared before the first instruction of the service routine is fetched.

The exception vectors are stored in the following addresses:

1). RST: {0xFFFD, 0xFFFC}
2). IRQ/BRK: {0xFFFF, 0xFFFE}
3). NMI: {0xFFFB, 0xFFFA}

These three 16-bit vector locations are standard across all implementations of the 6502 microprocessor. (*The WDC W65C802 and W65C816 processors allow the BRK vector to be separated from the IRQ vector*.)

ADDRESSING MODES

The W65C02 has fifteen (15) addressing modes, and sixteen (16) if the Rockwell enhancements are included. Two modes of the W65C02 can generally be combined into a single mode: (1) implicit operand mode, and (2) accumulator mode. The following table, Table 4, summarizes the addressing modes of the W65C02 processor.

With respect to notation used in Table 4, the effective address is given using a mixture of C and Verilog notation. The notation *(·) is used to indicate that the contents of the location whose address is the argument is the address. (*That is, the classic C dereference of a pointer*.) Another notational usage is the use of PC in the effective address equations. In the table, **$** refers to the address of the instruction. Thus, **$**+1 points to the first operand of a multi-byte instruction, or to the next instruction in the case of single byte instructions.

The notation {8{·}} indicates an 8-bit vector. The argument, either 1'b0 or rel[7], indicates a logic 0 bit, or the most significant bit of the PC relative offset, respectively. In first case, the 8-bit index register is being extended to 16 bits by setting the most significant byte to logic 0, and in the second, the PC relative offset is sign extended to 16 bits.

Finally, the modulus operator is represented as using the % symbol. This operator is explicitly included in the effective address equations to represent the fact that the arithmetic carries between the least significant byte and the most significant byte are dropped, i.e. the operation is mod 256.

Table 4. W65C02 Addressing Modes.

ID	Name	Syntax	Effective Address
0	Implicit		Operands are implicit in the instruction
1	Accumulator	A	Operand is the Accumulator
2	Immediate	#imm	EA <= $+1
3	PC Relative	rel	EA <= {$+2 + {CC **?** {{8{OP1[7]}}, OP1}} **:** 0}}
4	Zero Page	dp	EA <= {{8{1'b0}}, OP1}
5	Zero Page Indirect	(zp)	EA <= *({{8{1'b0}}, OP1})
6	Zero Page Pre-Indexed	zp,X	EA <= {{8{1'b0}}, (OP1 + X) % 256}
7	Zero Page Pre-Indexed Indirect	(zp,X)	EA <= *({{8{1'b0}}, (OP1 + X) % 256})
8	Zero Page Post-Indexed	zp,Y	EA <= {{8{1'b0}}, (OP1 + Y) % 256}
9	Zero Page Post-Indexed Indirect	(zp),Y	EA <= *({{8{1'b0}}, OP1}) + {{8{1'b0}}, Y}
A	Absolute	abs	EA <= {OP2, OP1}
B	Absolute Indirect	(abs)	EA <= *({OP2, OP1})
C	Absolute Pre-Indexed	abs,X	EA <= {OP2, OP1}+{{8{1'b0}}, X}
D	Absolute Pre-Indexed Indirect	(abs,X)	EA <= *({OP2, OP1} +{{8{1'b0}}, X})
E	Absolute Post-Indexed	abs,Y	EA <= {OP2, OP1} +{{8{1'b0}}, Y}

QUIRKS OF THE W65C02

The W65C02 corrected most of the deficiencies of the MOS6502 instruction set, but it had to retain some significant quirks in order to remain compatible. The W65C02's quirks influence how memory and software is implemented. The following subsections discuss the quirks of the W65C02.

STACKED SUBROUTINE ADDRESS

Unlike most processors, the W65C02 does not push onto the stack the address of the following instruction. Instead, the processor pushes the address of the most significant address of the subroutine address, i.e. the address of the second instruction operand. Thus, during a ReTurn from Subroutine (**`RTS`**) instruction, the processor pops both bytes of the return address from the stack, LSB first, and then uses one cycle to increment the value read from the stack by one before fetching the instruction from the adjusted address.

STACKED INSTRUCTION ADDRESS AFTER INTERRUPT

Like the issue with the stacked address of subroutines, the W65C02 does not push the address of the instruction following the instruction being interrupted. Instead, it pushes the address of the last byte of the current instruction. Thus, the ReTurn from Interrupt (**`RTI`**) must also increment the return address retrieved from the stack by one before fetching the next instruction.

*STACKED ADDRESS AFTER BREAK (**BRK**) INSTRUCTION*

After the BRK instruction is encountered, the W65C02 pushes a return address followed by the PSW. The addressed pushed onto the stack is not the address of the **BRK** instruction. Instead it is the address of the byte following the **BRK** instruction. This means that following a **BRK**, the W65C02 continues execution with second instruction following the BRK. If the byte following the **BRK** instruction is not an immediate operand to the **BRK** instruction trap handler, then the handler needs to subtract 1 from the return address before executing the **RTI** instruction, otherwise the instruction following the **BRK** instruction will be skipped.

*ARITHMETIC OVERFLOW FLAG NOT SET BY **CMP**/**CPX**/**CPY***

The comparison instructions of the 6502 are unsigned comparisons between the designated register and an immediate or memory operand. Thus, signed comparisons using the N, V, and C flags are not possible with the 6502.

*ARITHMETIC OVERFLOW FLAG SET BY **BIT***

The BIt Test (**BIT**) instruction sets/clears the V bit of the PSW if bit 6 of a memory operand is set/clear. If the **BIT** instruction is used with an immediate operand, then the V flag is not modified. (*The V flag may also be set by external hardware using a falling edge signal applied to the SV input pin of the 6502 processor. This feature could be used to construct very tight polling loops.*)

ZERO PAGE ADDRESS WRAP-AROUND

If an indexed zero page addressing mode is used, then the resulting effective address is restricted to lie in zero page memory. In other words, indexing a zero page address, whether using a direct or indirect addressing mode, will not propagate carries from the LSB of the effective address into the MSB of the effective address. Without the propagation of the carry, the effective address will wrap around modulo 256 in the zero page. For example, if the page zero base address is 0xF0, and the X index register contains a value of 0x20, then an instruction using the **0xF0,X** zero page indexed by X addressing mode will read/modify/write page 0 location 0x0010 instead of page 1 location 0x0110. (**Note:** *this address wrap around also occurs for page 1 stack operations.*)

B FLAG (P[4]) SET ONLY IN STACKED PSW

The B flag is only set in the value of the P register (PSW) pushed onto the stack as part of the **BRK** instruction or the **PHP** instruction. In the 6502, the interrupt/trap service routine is shared between the maskable external interrupt IRQ and the **BRK** instruction. To determine whether the common service routine is handling an interrupt or a **BRK** instruc-

tion, the service routine must check bit 4 (B flag) of the P register stored on the stack. The B flag of the P register in the processor is indeterminate, and therefore can't be used for this purpose.

Memory Cycle During Modify Cycle of RMW Instructions

Both the MOS6502 and the W65C02 perform a dummy memory access during the internal operand modification cycle of Read/Modify/Write instructions. The MOS6502 performed a dummy read with an invalid address during the modify cycle. The W65C02 converted the dummy read operation of the MOS6502 into a dummy write cycle, but the W65C02 also ensured that the address of the dummy write during the modify cycle is the same address as that used for the read and write cycles.

This quirk of the MOS6502 could cause erratic behavior of memory-mapped I/O devices. Instead of being "quite" and performing no external bus cycles during the modify cycle of a RMW instruction, the W65C02 performs a dummy write of the same memory location as used for the read cycle. Since the address is defined, the W65C02 should not cause an issue with memory-mapped I/O devices unless the RMW instruction is acting on an I/O register of a memory-mapped device. Thus, RMW instructions should be avoided when memory-mapped I/O is present in a system with a MOS6502, but may be used in a system based on the W65C02 if the potential side-effects of these W65C02 instructions are carefully considered.

Fixes to MOS6502 Implemented by W65C02

The W65C02 fixed all known bugs of the MOS6502. The following subsections discuss the fixes implemented by the W65C02 with respect to the MOS6502.

*D Flag (P[3]) Cleared on Reset, IRQ, NMI, and **BRK***

Unlike the original MOS6502, the W65C02 ensures that the D flag in the PSW is cleared after reset. TheW65C02 also ensures that the D flag is also cleared when an interrupt/trap service routine is started. This means that the NMI, IRQ, and **BRK** service routines all start operation in the binary mode.

Operation of Arithmetic Flags in Decimal Mode

In the Decimal arithmetic mode, the arithmetic flag logic of the MOS6502 did not operate correctly. That is, during decimal mode arithmetic operations on the MOS6502, only the zero, Z (P[1]), flag operated correctly, and the negative, N (P[7]), flag, the arithmetic overflow, V (P[6]), flag, and the carry, C (P[0]), flag operated incorrectly. The W65C02 corrected the issues with these three flags during decimal mode operations.

JUMP ABSOLUTE INDIRECT INSTRUCTIONS

The MOS6502's implementation of these instructions did not operate correctly when the address operands straddled a page boundary. The W65C02 corrected this issue so that the Jump Absolute Indirect, **`JMP (abs)`**, and the Jump Pre-Indexed Absolute Indirect, **`JMP (abs,X)`**, instructions correctly cross page boundaries. (**Note:** *the Jump Pre-Indexed Absolute Indirect instruction was added by the W65C02.*)

UNIMPLEMENTED INSTRUCTIONS

Execution of an unimplemented instruction opcode on the MOS6502 resulted in unspecified behavior which included unspecified modifications of the registers. The W65C02 rectifies this behavior by ensuring that unimplemented instructions do not modify any of the processor's registers, i.e. behave as NOPs. However, a variable number of clock cycles are required for these unimplemented instructions to complete.

IMPLEMENTATION OF THE M65C02

The implementation of the M65C02 has been performed in two stages: (1) the implementation of the core logic; and (2) the integration of the processor core with a memory interface/controller and interrupt controller. The implementation of the processor core will be described first. That discussion will be followed by a description of the processor core integrated with a memory interface/controller. As part of these descriptions, the verification plan and verification test bench of each of these two implementations will be discussed.

IMPLEMENTATION OF THE M65C02 CORE LOGIC

The objective of the M65C02 core logic is to provide the basis for the implementation with FPGAs of M65C02-based soft-core microprocessors of various configurations. To achieve this goal, the core logic is organized into three main modules:

(1) M65C02_ALU
(2) M65C02_MPC
(3) M65C02_Core

The M65C02_ALU module implements the core logic for the programmer visible registers except the program counter (PC), the arithmetic, shift, bit, and logic functions, and the stack pointer logic. The binary and decimal arithmetic functions are implemented separately in two sub-modules: M65C02_BIN, and M65C02_BCD.

The M65C02_MPC module implements the Micro-Program Controller (MPC) function. The M65C02 is implemented using a microprogrammed control unit. In the case of the M65C02, a microprogrammed control unit provides a structured approach to the development of the control unit which is very well matched to the capabilities of the target

hardware, RAM-based FPGAs. A balanced approach is used to determine the level of encoding used for the control fields in the microprogram word. That is, encoded control fields are used when performance of the resulting M65C02 core is greater than 100 MHz, and un-encoded fields are used otherwise. Encoded control fields naturally require decoders to control the logic of the core, and these decoders will generally require additional logic levels in a RAM-based FPGA. Therefore, periodic FPGA synthesis, placement and routing cycles were used to ensure that the M65C02 core with its encoded microprogram fields would still meet that performance objective of 100 MHz operation.

The core module, M65C02_Core, instantiates the ALU and MPC modules, and implements the control unit microprogram ROM, the instruction Decoder ROM, the Instruction Register (IR), the operand registers (OP1 and OP2), the Program Counter (PC), the Memory Address Register (MAR), and various other decoders, multiplexers, incrementers and adders using behavioral Register-Transfer-Level (RTL) descriptions using the Verilog Hardware Description Language (HDL).

The core assumes that it is connected to two external components: a memory interface controller, and an interrupt controller. Thus, for instruction fetches and data memory reads, the core provides a memory address under the assumption that the output data from the memory is available in the same cycle, i.e. an asynchronous, zero wait state memory. Similarly, the core assumes that data memory writes are completed in the same cycle. The core does provide for a transfer acknowledge (`Ack`) input signal to allow the external memory interface to insert wait states when required. The following sub-sections will provide additional details of the implementation of the M65C02 core.

The interrupt controller is expected to sample the external interrupt request signals, IRQ and NMI, and to assert an interrupt request to the core. The interrupt controller is also expected to provide the vector to the core. However, as currently implemented, the M65C02 core's microprogram does not execute a **`JMP (vec)`** instruction to initiate the service routine. Instead, if 6502-compatible registers are provided for the three defined vectors, the interrupt controller provides the contents of these registers to the core so that the two indirect address fetches of the indirect jump instruction are not performed. The reason for this optimization is that the vector locations are generally in ROM at the upper end of the M65C02's address space. Since the M65C02 is intended for operation as a soft-core within an FPGA, it does not make much sense to implement read-only registers for these vectors. Thus, the intention is for these vectors to be provided to the M65C02's interrupt controller module by the user's logic in whatever manner, input ports or parameters/generic constants, is deemed most appropriate for the application.

IMPLEMENTATION OF THE M65C02 ALU MODULE

As stated above, the M65C02 ALU module implements the programmer visible registers (with the exception of the PC), the functional units, and the stack pointer. The interface of M65C02 ALU module, provided below, succinctly defines the basic functionality implemented in the ALU module.

```
module M65C02_ALU #(
    parameter pStkPtr_Rst = 0 // Stack Pointer Value after Reset
)(
    input   Rst,              // System Reset - synchronous reset
    input   Clk,              // System Clock

    input   Rdy,              // Ready

    input   En,               // Enable - ALU functions
    input   [2:0] Reg_WE,     // Register Write Enable
    input   ISR,              // Asserted on entry to ISR

    //  ALU Inteface

    input   [3:0] Op,         // ALU Operation Select
    input   [1:0] QSel,       // ALU Q Bus Multiplexer Select
    input   RSel,             // ALU R Bus Multiplexer Select
    input   Sub,              // ALU Adder Operation Select
    input   CSel,             // ALU Carry In Multiplexer Select
    input   [2:0] WSel,       // ALU Register WE Select
    input   [2:0] OSel,       // ALU Output Multiplexer Select
    input   [4:0] CCSel,      // ALU Condition Code Operation Select

    input   [7:0] M,          // ALU Memory Operand Input
    output  reg [7:0] Out,    // ALU Output(asynchronous)
    output  Valid,            // ALU Output Valid

    output  reg CC_Out,       // Condition Code Test Output

    //  Stack Pointer Interface

    input   [1:0] StkOp,      // Stack Pointer Operation Select

    output  [7:0] StkPtr,     // Stack Pointer Output: {S, S+1}

    //  Internal Registers Interface

    output  reg [8:0] ALU,    // Internal ALU Result Bus
    output  reg [7:0] A,      // Accumulator
    output  reg [7:0] X,      // X Index Register
    output  reg [7:0] Y,      // Y Index Register
    output  reg [7:0] S       // Stack Pointer

    output  [7:0] P           // Processor Status Word
);
```

System Interface

In the implementation of the M65C02, all modules operate from a single clock, `Clk`, and all modules have a reset input, `Rst`. The remainder of the system interface ports of the M65C02_ALU modules are discussed in the following sub-sections.

Rdy

The Rdy input is provided from the core logic to prevent the updating of processors registers (A, X, Y), the stack pointer (S), or the processor status word (P) until an internal or external wait state is complete. Thus, the ready input, Rdy, provides a way for the M65C02 core logic to stretch ALU cycles synchronously.

En

The En input enables the ALU to execute the required function as defined by the instruction decoder. En is asserted by the core logic whenever the Reg_WE field is not equal to 0. When asserted, En initiates the ALU operation defined by the Op input. For all ALU functions, other than decimal mode (BCD) additions and subtractions, asserting En also results in the ALU Valid output being asserted on the same cycle. In other words, all ALU operations are single cycle combinatorial functions except for decimal mode addition and subtraction. For decimal mode additions and subtractions, En initiates the two cycle decimal arithmetic module, and the ALU Valid output is asserted with a one clock delay, i.e. on the second clock cycle.

Reg_WE

The Reg_WE input is provided directly by one of the M65C02 microprogram's control fields. It functions as a write enable field for the ALU's various registers. During development of the M65C02 it was found that a single control input, En, was not sufficient to control the writing to the ALU's registers or external memory, and simultaneously updating of the P register flags. This was particularly an issue for Read/Modify/Write (RMW) instructions. The following table, Table 5, provides the encoding of the Reg_WE input:

Table 5: Register Write Enable Mapping.

Reg_WE[2:0]	Register
000	-
001	A
010	X
011	Y
100	WSel
101	S
110	P
111	M

ISR

The ISR input is asserted by the microprogram after the return address and processor status word have been pushed onto the stack, and before the execution of the first instruction in the interrupt/trap service routine. Asserting ISR causes the decimal mode flag, D, to

be cleared and the interrupt mask flag, I, to be set. In this manner, the M65C02 ensures that the interrupt/trap service routine will start with the binary arithmetic mode selected and external interrupts disabled.

ALU Interface

The signals described in this section provide the control inputs, ALU data input and output, and condition code test outputs. The control inputs discussed in this section are provided by the instruction decoder. In the implementation of the M65C02, the instruction decoder ROM provides the fixed control and status elements for each implemented or unimplemented instruction. The following fields from the instruction decoder ROM control the operation of the M65C02 ALU: `Op`, `QSel`, `RSel`, `Sub`, `CSel`, `WSel`, `OSel`, and `CCSel`. The `M`, `Out`, and `Valid` signals provide the ports for data input and output from the M65C02 ALU, and `CC_Out` provides the condition code test output.

`Op`

The `Op` input selects one of 16 functions/operations that are implemented by the M65C02 ALU. The following table, Table 6, defines the ALU functions selected by the `Op` input. In Table 6, the mnemonics M, Q, and R refer to memory, Q multiplexer output, and the R multiplexer output, respectively. The mnemonics N, V, Z, and C refer to the ALU flags in the processor status word, `P`, which generally indicate a 2's complement negative value (N), 2's complement arithmetic overflow (V), logic zero (Z), and arithmetic carry (C), respectively. The V flag is modified by the addition, subtraction, and bit test operations. Modification of the V and N flags by the **`BIT`** operation is dependent on whether an immediate operand is used or not. If the **`BIT`** operation's memory operand is an immediate constant, the N and V flags are not set to the values of the two most significant bits of the operand, respectively. If the operand of **`BIT`** is not an immediate constant, then the N and V flags are set to the values of the two most significant bits of the memory operand, respectively. (*Not indicated in Table 6 is the fact that the V flag may also be modified by an external signal on the SOV, Set Overflow, pin of the W65C02 processor. Thus, using the V flag for arithmetic overflow can be problematic on all 6502 variants, including the M65C02.*) Also note that in most instances, the zero (Z) flag is set by evaluating the output of the ALU. However, in the case of the **`BIT`**, **`TRB`**, and **`TSB`** operations, the Z flag is set from the logical AND of the memory operand (M) and the accumulator (A). Also note the definition for the **`ADC`**, **`SBC`**, **`INC`**, **`DEC`**, and **`CMP`** operations. Both addition and subtraction is defined in terms of addition only. Subtraction is performed by addition of the complement of the memory operand. Thus if the carry is not set, an automatic 2's complement borrow will result. This is the reason that this microprocessor requires the carry to be cleared before addition, and set before subtraction. It also means that carries/borrows are automatically incorporated into multi-precision operations without resorting to two different instructions as is the case for most other processors.

Table 6: M65C02 ALU Operation Codes.

Op[3:0]	Mnemonic	Operation	Condition Codes
0000	XFR	ALU <= {0 | A | X | Y | S | P | M}	None
0001	OR	ALU <= A | M	N = ALU[7] Z = ~|ALU
0010	AND	ALU <= A & M	N = ALU[7] Z = ~|ALU
0011	EOR	ALU <= A ^ M	N = ALU[7] Z = ~|ALU
0100	ADC	ALU <= A + M + C	N = ALU[7] V = OVF Z = ~|ALU C = C_{out}
0101	SBC	ALU <= A + ~M + C	N = ALU[7] V = OVF Z = ~|ALU C = C_{out}
0110	INC	Q = {A | X | Y | M} ALU <= Q + 1 + 0	N = ALU[7] Z = ~|ALU
0111	DEC	Q = {A | X | Y | M} ALU <= {A | M} + ~1 + 1	N = ALU[7] Z = ~|ALU
1000	ASL	W = {A | M} ALU <= {W[6:0], 0}	N = ALU[7] Z = ~|ALU C = W[7]
1001	LSR	W = {A | M} ALU <= {0, W[7:1]}	N = ALU[7] Z = ~|ALU C = W[0]
1010	ROL	W = {A | M} ALU <= {W[6:0], C}	N = ALU[7] Z = ~|ALU C = W[7]
1011	ROR	W = {A | M} ALU <= {C, W[7:1]}	N = ALU[7] Z = ~|ALU C = W[0]
1100	BIT	ALU <= M & A	N = imm **?** N **:** M[7] V = imm **?** V **:** M[6] Z = ~|(M & A)
1101	TRB	ALU <= M & ~A	N = ALU[7] Z = ~|(M & A)
1110	TSB	ALU <= M | A	N = ALU[7] Z = ~|(M & A)
1111	CMP	Q = {A | X | Y} ALU <= Q + ~M + 1	N = ALU[7] Z = ~|ALU C = C_{out}

`QSel`

The `QSel` input refers to the select control of the Q multiplexer of the M65C02 ALU. The Q multiplexer selects among the various processor registers. The `QSel` input is provided by the instruction decoder output and remains fixed until the next instruction is loaded into the IR. The following table, Table 7, defines the values of the QSel input and the register which is multiplexed.

Table 7: Q Multiplexer Select Mapping.

QSel[1:0]	Q Multiplexer Output
00	A
01	M
10	X
11	Y

`RSel`

The `RSel` input refers to the select control of the R multiplexer of the M65C02 ALU. The R multiplexer selects between two sources. The following table, Table 8, defines the values of the RSel input and the sources multiplexed. The normal state of `RSel` is logic 0, so that the output of the R multiplexer is the memory operand. For the decrement and increment instructions, **`INC`**, **`INX`**, **`INY`**, **`DEC`**, **`DEX`**, and **`DEY`**, `RSel` is set to a logic 1 which places a value of 1 on the second input of the binary adder.

Table 8: R Multiplexer Select Mapping.

RSel	R Multiplexer Output
0	M
1	1

`Sub`

The `Sub` input refers to the subtraction control of the M65C02 ALU. When asserted, the R multiplexer output is complemented. This is how subtraction is implemented in the M65C02 ALU. The following table, Table 9, defines how Sub affects the output of the R multiplexer.

Table 9: R Multiplexer Output Control.

Sub	R Multiplexer Output
0	R
1	~R

`CSel`

The `CSel` input refers to the carry multiplexer select of the M65C02 ALU. When asserted, the carry input to the M65C02 ALU adders, the binary mode and decimal mode ad-

ders, is the Sub input, otherwise it is the C flag from P. This architecture ensures the correct implementation of **SBC** and the **DEC**, **DEX**, and **DEY** instructions. CSel is not asserted for **SBC**, but it is asserted for the decrement instructions. This means that the C flag is used for the **SBC** subtraction operation, and forced to a logic 1 for the decrement instructions. **(Note:** *the increment and decrement instructions are binary mode only instructions. Only the add and subtract instructions,* **ADC** *and* **SBC**, *are dual mode operations.*)

Table 10: Carry Input Multiplexer Mapping.

CSel	Carry In
0	C
1	Sub

WSel

The WSel input acts as a write select for the ALU registers. Generally, WSel selects an ALU register to write, but it also indicates a write to memory, M. Writes to A, X, and Y also update the flags in P. RMW instructions to M also modify the flags in P as well. Thus, all arithmetic and logic operations update P. In addition, all load operations from memory or stack, and transfers between the A, X, and Y registers update P. Transfer of the stack pointer register, S, to X updates P, but not transfers from X to S. Stores from A, X, and Y do not update P, and P is also not updated by the Store Zero (**STZ**) instruction.

Reg_WE and WSel operate in concert to provide the M65C02 microprogrammed control unit the ability to directly or indirectly update registers. Reg_WE explicitly updates registers when the microprogram sequence is explicitly tied to a particular register, and Reg_WE indirectly writes to a register selected by WSel whenever the microprogram sequence is not explicitly aware of the destination register. An example of the first case are instructions using implied or accumulator addressing modes such as Pull P (**PLP**), Pull A (**PLA**), and Arithmetic Shift Left Accumulator (**ASL A**), and an example of the second case are instructions such as the zero page direct instructions: Add with Carry zero page (**ADC zp**), Load Accumulator from zero page (**LDA zp**), Compare X with immediate value (**CPX #imm**), Increment zero page (**INC zp**), etc.

In the first case, direct register writes, the control unit “knows” which register is being written, and in the second case, indirect register writes, the control unit “knows” when all of the operands are available but not which register is being written, or even what operation is to be executed. In the second case, the Op and WSel inputs set up the ALU to perform the correct operation and update the correct ALU register when signaled/enabled by the general Reg_WE signal as shown in Table 5, i.e. (Reg_WE == 4). The encoding of the WSel input is defined in Table 11.

Table 11: Write Select Encoding.

WSel[2:0]	Register
000	-
001	A
010	X
011	Y
100	-
101	S
110	P
111	M

`OSel`

The `OSel` input selects which of the registers or ALU outputs are placed on the output bus, `Out`. The encoding of the `OSel` input follows that of the `WSel` input as defined in Table 11. However, `OSel` also selects a zero output to facilitate the implementation of the Store Zero (**`STZ`**) instructions. The following table, Table 12, defines the encoding of `OSel`.

Table 12: Output Select Encoding.

OSel[2:0]	Register
000	M
001	A
010	X
011	Y
100	**0**
101	S
110	P
111	M

`CC_Sel`

The `CC_Sel` input provides the control for the various instructions that affect the flags in the processor status word, `P`. `CC_Sel` provides direct control of `P` and of the `CC_Out` output signal. The following table defines the encoding for the `CC_Sel` input, and defines the action performed by the M65C02 ALU for each defined value of `CC_Sel`.

Table 13: M65C02 ALU Condition Code Select Codes.

CC_Sel[4:0]	Mnemonic	CC_Out	Comment
00_xxx	NOP	1	Always true, used for Branch Always (**BRA**)
01_000	CC	~C	True if Carry flag not set
01_001	CS	C	True if Carry flag set
01_010	NE	~Z	True if Zero flag not set
01_011	EQ	Z	True if Zero flag set

CC_Sel[4:0]	Mnemonic	CC_Out	Comment
01_100	VC	~V	True if oVerflow flag not set
01_101	VS	V	True if oVerflow flag set
01_110	PL	~N	True if Negative flag not set
01_111	MI	N	True if Negative flag set
10_000	CLC	1	C <= 0; Clear Carry flag
10_001	SEC	1	C <= 1; Set Carry Flag
10_010	CLI	1	I <= 0; Enable Interrupts
10_011	SEI	1	I <= 1; Disable Interrupts
10_100	CLD	1	D <= 0; Set Binary Arithmetic Mode
10_101	SED	1	D <= 1; Set Decimal Arithmetic Mode
10_110	CLV	1	V <= 0; Clear oVerflow flag
10_111	BRK	1	B <= 1; Set Break flag
11_000	Z	1	Set selected flags in P
11_001	NZ	1	Set selected flags in P
11_010	NZC	1	Set selected flags in P
11_011	NVZ	1	Set selected flags in P
11_100	NVZC	1	Set selected flags in P
11_101	-	1	
11_110	-	1	
11_111	PSW	1	P <= M, used to load P from stack

Note that in Table 13 the `CC_Sel` signal is divided into four divisions of eight codes each. The first division is used to set `CC_Out` to a logic one for the purpose of a **BRA** instruction. The second division outputs on `CC_Out` the selected flag bit which controls one of the eight conditional branch instructions. The mnemonic shown in the table corresponds to the condition code tested by these eight instructions. As an example, the mnemonic CC corresponds to the Branch if Carry Clear (**BCC**) instruction. The third division represents one of the eight instructions that set or clear a flag bit in `P`. The mnemonics shown in the table correspond to the actual M65C02 instruction. Finally, the fourth division controls the writing of `P` flags. For example, the mnemonic NVZC indicates that all four arithmetic flags will be updated by whatever instruction will be executed during the execute cycle.

Another interpretation of the `CC_Sel` input is that `CC_Sel[4]` functions as a write enable for `P`, and that `CC_Sel[3]` is selects between two blocks of eight functions/operations on `P`. For the read mode, `CC_Sel[3]` selects between logic 1 and one of the eight branch instruction conditions. For the write mode, `CC_Sel[3]` selects between individual flag set/clear operations (all part of the instruction set, see Table 1), or writes to one or more flags on the basis of ALU operations.

In summary, `Op` controls the ALU operation, and `CC_Sel` controls the updating of `P`.

M

M is the input data for the ALU. For all two operand instructions, M is used as the second operand and the accumulator, A, is the first operand. M is also used as the only operand for all Read-Modify-Write (RMW) instructions. ALU operands from memory, M, are registered under control of the M65C02 microprogram into the OP1 register.

Out

Out is the output of the ALU module's output multiplexer. It represents the 8-bit data selected by the OSel input. Refer to Table 12 for the registers and/or results that the output multiplexer will connect to Out.

Valid

Valid is asserted by the ALU logic when the data on Out is valid. In the present construction, the core's MPC only uses Valid to stretch the execute cycle of the decimal mode **ADC** and **SBC** instructions. For all other instructions, the core's MPC does not use the Valid signal, although it is tied to one of the MPC's test inputs.

CC_Out

The CC_Out output is used to select whether an M65C02 branch instruction will branch or not. In the M65C02, even unconditional branches use CC_Out.

Stack Pointer Interface

In the MOS6502 and W65C02, the stack pointer is incremented and decremented using the binary adder of the ALU. In the M65C02, there are two dedicated adders/subtractors for the stack pointer. These dedicated adders/subtractors are used to speed the generation of the address for all stack operations. Therefore, the stack pointer interface provides for independent control of the stack pointer so that the stack pointer adjustments occur in parallel with stack memory operations.

In general, the 6502 stack pointer points to the next open location in the stack. Thus, stack writes, i.e. push operations, simply write to the memory location pointed to by the current stack pointer. After the write is complete, the stack pointer is automatically decremented to point to the next free location in the stack. The situation is reversed for stack read operations, i.e. pull (pop) operations. That is, before a value can be pulled (popped) from the 6502 stack, the stack pointer must be incremented, then the value stored in that location is read. This action leaves the stack pointer ready for a subsequent write operation.

StkOp

The StkOp input controls the stack pointer. It provides control of an auxiliary adder/subtractor that drives the external address bus, and it controls the operations of the stack pointer register itself. The following table, Table 14, defines the operations that the stack pointer performs under control of the StkOp input.

Table 14: Stack Operation Control Input Mapping.

StkOp[1:0]	StkPtr	S	Comment
00	S	S	Hold
01	S	S	Reserved
10	S	S--	Post-decrement (Push)
11	++SS	S++	Pre-increment (Pull)

Note the distinction between the StkPtr output and the stack pointer when the value of StkOp is 3. In this case, an auxiliary adder which increments the value of the stack pointer, S, is multiplexed onto the StkPtr output. The stack pointer, on the other hand, increments or decrements on the next rising edge of the system clock as defined by StkOp. The auxiliary adder in the StkPtr output is used to remove the pipeline delay that would be incurred during stack pull operations.

StkPtr

The StkPtr output provides a direct combinatorial output based on the value of StkOp and the current value of the stack pointer register, S.

Internal Processor Registers Interface

In general, the approach taken in the development of the ALU and core modules is to expose all internal signals on the module port list that may necessary for simulation and debugging. For this reason, all of the M65C02 ALU's programmer-visible registers are exposed.

A - Accumulator

The M65C02, and all variants of the 6502, are known as accumulator based processors. In all accumulator-based processors, the accumulator, A, is used as an implicit source and/or destination operand. Any ALU functions requiring a second operand will use another register (implicitly or explicitly addressed) or data from an explicitly addressed memory location to access the second operand. The 6502 and its variants (including the M65C02) are single address machines. The destination of all two operand instructions is the accumulator. (**Note:** *the 6502 provides single operand RMW instructions which can generally be thought of as violating the concept of an accumulator-based architecture.*)

X and Y – Index Registers

X and Y are the two index registers of the M65C02. They are output directly from the ALU so that they may be used for index calculations by the address generation logic of the core. These two registers can also be accessed through Out, but the M65C02 implements next address generation with a dedicated functional unit to reduce the cycle count of all indexed addressing modes, both direct and indirect. Thus, access through Out is reserved for saving these registers to the stack or to memory.

S – Stack Pointer

The M65C02 stack pointer, S, is brought out for debugging and simulation purposes. Unlike the A, X, and Y registers, S can only be written from and read through X using the Transfer X to S (**TXS**) and the Transfer S to X (**TSX**) instructions, respectively. All other operations on the stack pointer are controlled by the StkOp input as defined in Table 14.

P – Processor Status Word

Within the M65C02 ALU, the processor status word is implemented in a 6-bit register, PSW. The relationship between the internal PSW and P, the programmer-visible processor status word, is described in Table 15.

Table 15: Internal and Programmer-Visible Processor Status Word Mapping.

Bit	PSW[5:0]	Bit	P[7:0]	Comment
0	C	0	C	Updated by Arithmetic and Shift units
1	Z	1	Z	Updated by Arithmetic, Shift, Logic, Bit units, and register loads
2	I	2	I	Updated by **SEI**/**CLI** instructions, and set by ISR
3	D	3	D	Updated by **SED**/**CLD** instructions, and cleared by ISR
		4	{1 | 0}	Set by **BRK** and **PHP** (only on stack), otherwise reads as 0
		5	1	Unused bit in the M65C02 and W65C02; reads as a 1
4	V	6	V	Updated by Arithmetic and Bit (non-immediate operand) units, and cleared by **CLV** instruction
5	N	7	N	Updated by Arithmetic, Shift, and Bit (non-immediate operand) units

Note that reading and writing of the various status bits in PSW occurs for virtually all instructions. Only the store, branch, jump and subroutine return instructions don't modify the PSW, as shown in Table 1.

Description of M65C02 ALU Implementation Details

A high level block diagram is provided in **Error! Reference source not found.**. The intent of the block diagram is to show the general organization of the M65C02 ALU rather a details of the implementation. The block diagram of does provide a road map for the evaluation of the verilog RTL code for the M65C02 ALU module.

The key to understanding the implementation is how the data flows through the module. In the top left corner of the diagram are two multiplexers: the Q bus multiplexer, and the

R bus multiplexer. The Q multiplexer implements a 4:1 multiplexer controlled by the `QSel` input. The Q multiplexer provides the ALU functional units the primary operand: `A`, `M`, `X`, or `Y`. The 2:1 R multiplexer is controlled by the `RSel` and `Sub` inputs, and provides the ALU functional units the secondary operand: `M` or 1. The `Sub` input controls the complementing of the output of the R multiplexer.

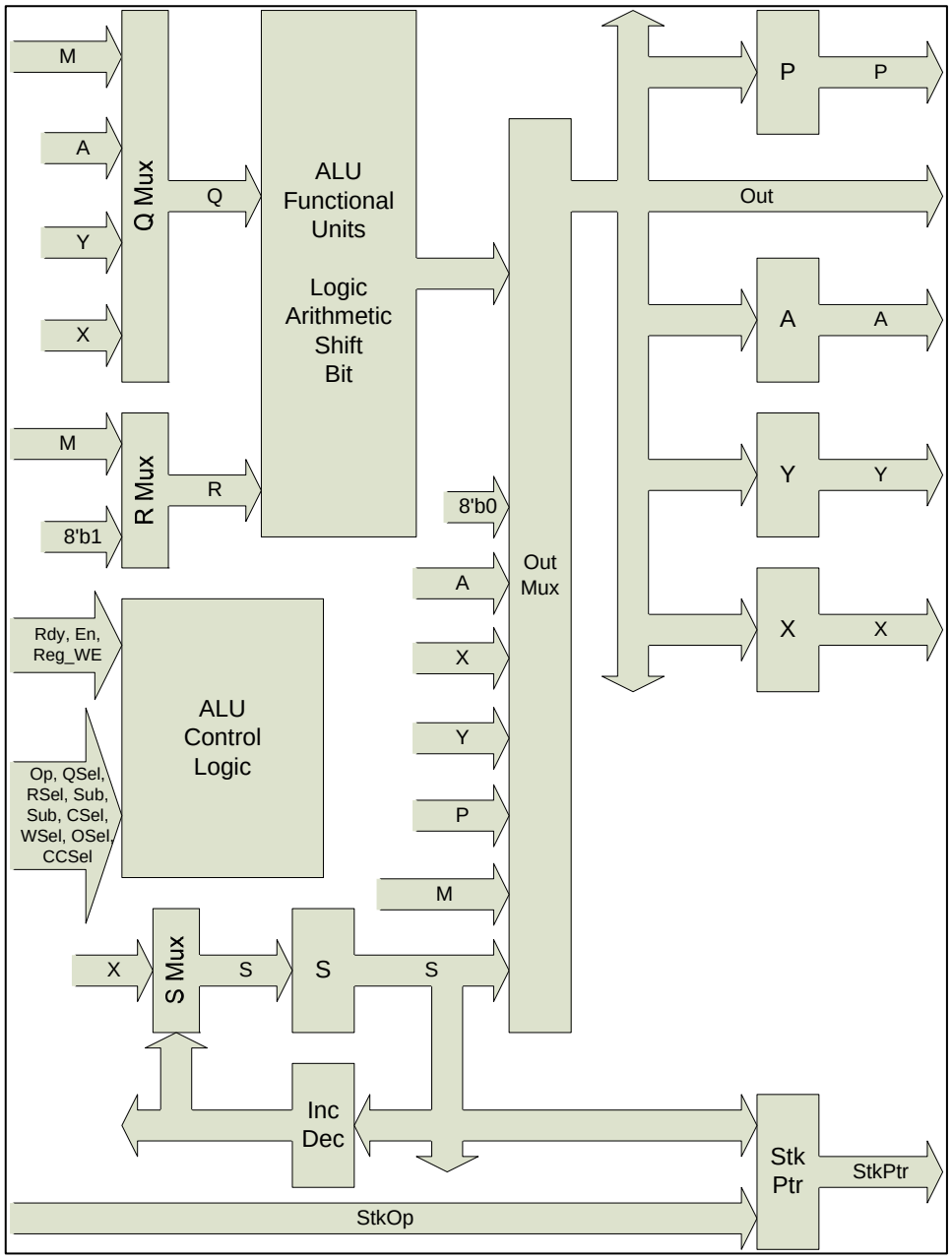

Figure 1: M65C02 ALU High-Level Block Diagram.

The two operands of the ALU, Q and R, are passed to the ALU functional units along with the `Op`, `Sub`, `CSel`, and `En` inputs. The output of the functional units is multiplexed

onto Out. To avoid combinatorial logic loops through these cascaded (mostly) combinatorial logic elements of the ALU, M must be supplied from a register like the other inputs to the Q and R multiplexers.

The output multiplexer is controlled by the OSel input. When RMW, load, or store operations are required, OSel must be set such that either the output of the ALU functional units (RMW instructions), or one of the other inputs to the output multiplexer is selected: A, X, Y, 0, S, P, or M. The output multiplexer is actually implemented as two cascaded multiplexers. The first multiplexer, referred to as the Load/Store/Transfer multiplexer, multiplexes all of the registers and/or M in order to connect the necessary value to Out, and as a result, to any of the remaining ALU registers since all ALU registers share a common input bus, Out. The second multiplexer, referred to as the ALU Output multiplexer, multiplexes one of the outputs from the ALU functional units or the output of the Load/Store/Transfer multiplexer onto Out.

Therefore, the general data flow is for A, X, Y, or M to be passed through the ALU functional units or the Load/Store/Transfer multiplexer and onto the output bus, Out. When the proper selections have been made with Op and OSel, then Rdy, En, Reg_WE, and WSel inputs and the Valid output control which ALU register, if any, is to be written. To develop the necessary register write enable signals, Reg_WE and WSel are combined to form a 6-bit control field which is explicitly decoded into one-hot register write enable signals. (*The intention was to fully define the 64 conditions which the combined vector {*Reg_WE, WSel*} represents. However, the full expansion necessary to have the synthesis pass generate a LUT-based ROM was not deemed necessary since the implementation was meeting its design goal on 100 MHz operation without it.*) The output of the {Reg_WE, WSel} is the register select, which is then ANDed with Valid and Rdy for the write enable of each programmer-visible register.

Writing to the PSW is a bit more complicated than writes into A, X, Y, or S registers. PSW is updated when ISR is asserted independent of Rdy, Reg_WE, WSel, or Valid. This allows the decimal mode flag and the interrupt mask flags to be set correctly when and interrupt/trap service routine is entered. Under normal register update conditions, PSW is written to with various combinations of flags, flag set/clear commands, and/or from the M input. The implementation uses a case statement to update the various bits in the PSW when the write enable PSW signal, WE_P, is asserted. This approach was used because the original mechanism for implementation of the PSW as individual registers with multi-level multiplexers proved difficult to modify in order to add the capabilities to set the interrupt mask, clear the decimal mode flag, etc. when special circumstances occurred.

The last component of the M65C02 ALU implemented is the stack pointer register. It is implemented in a straightforward manner using a register updated by the StkOp input or by Reg_WE, WSel, and Rdy. The StkOp input controls whether the register increments

(pulls/pops) or decrements (pushes). `Reg_WE`, `WSel`, and `Rdy` are used to control the updating of the register from the `X` register.

Implementation of the Binary and Decimal Arithmetic Modes

In most cases, the M65C02 is expected to operate in the binary arithmetic mode. The adjustments that must be applied to each nibble of a binary sum require too much logic to implement in a single cycle, dual-mode arithmetic module while attempting to maintain the objective of 100 MHz operation. Furthermore, the additional logic stage required for decimal mode operation of the ADd memory to accumulator plus Carry (**ADC**) and the SuBtract memory from accumulator plus Carry (**SBC**) instructions would impose a performance penalty on all binary arithmetic operations of the M65C02 core: **ADC**, **SBC**, **INC**, **DEC**, **INX**, **DEX**, **INY**, **DEY**, **CMP**, **CPX**, and **CPY**.

Within the M65C02 ALU module binary and decimal mode arithmetic is performed by two sub-modules: M65C02_BIN and M65C02_BCD. Two modules are used for consistency. The binary adder module is so simple that it could have been implemented directly within the M65C02 ALU module. However, since the more complex decimal mode adder module is better implemented as a sub-module and instantiated in the ALU module, the binary mode adder module was implemented in a like manner.

For both modules the inputs are the Q and R outputs of the first multiplexer stages and a carry input. The binary mode adder splits the sum into two sections in order to develop the arithmetic overflow output as the exclusive OR of the carry out of bit 6 and the carry out of bit 7. This implementation of the binary mode adder may impose a performance penalty because the carry chain is broken in order to break into it to determine the carry out of bit 6 of the adder. However, the performance of this adder is not a contributing factor to the performance limit of the ALU, or the core, in general.

The implementation of the decimal mode adder is simply a binary first stage followed by a BCD adjustment stage. The sum of the first stage binary adder is registered, as are several auxiliary signals. In the second stage, the Valid output is asserted, and the binary sum is adjusted using a case statement to add one of six adjustment factors which are computed using the registered nibble carries from the first stage, and three magnitude comparators. The second stage BCD adjustment is performed using a full width binary adder instead of the two 4-bit nibble adders used in the first stage. The BCD adjustments performed correctly implement the decimal mode overflow and carry flags.

IMPLEMENTATION OF THE M65C02 MICROPROGRAM CONTROLLER