65C816 Opcodes by Bruce Clark

Thanks to everyone who read over the early draft. Thanks to BigEd for the helpful feedback.

Table of contents:

1 INTRODUCTION

The 65C816 is a member of the 6502 microprocessor family, capable of addressing 16 megabytes of memory (i.e. 24-bit addressing), with instructions that operate on 8-bit and 16-bit data. It also includes an emulation mode for executing 65C02 and NMOS 6502 code.

2 MODES AND MEMORY ORGANIZATION

Both the 65C02 and the NMOS 6502 have two modes: binary mode and decimal mode; ADC and SBC instructions use binary arithmetic in binary mode and BCD (Binary Coded Decimal) arithmetic in decimal mode. The 65C816 has 10 combinations of modes of which binary mode and decimal mode are a part.

Also a part of these combinations are the width (8 or 16 bits) of the accumulator (and read-modify-write memory instructions, such as DEC), the width of the X and Y index registers (likewise, 8 or 16 bits), and native or emulation mode. The 10 combinations are:

- binary mode, emulation mode
- binary mode, native mode, 8-bit accumulator mode, 8-bit index register mode
- binary mode, native mode, 8-bit accumulator mode, 16-bit index register mode
- binary mode, native mode, 16-bit accumulator mode, 8-bit index register mode
- binary mode, native mode, 16-bit accumulator mode, 16-bit index register mode
- decimal mode, emulation mode
- decimal mode, native mode, 8-bit accumulator mode, 8-bit index register mode
- decimal mode, native mode, 8-bit accumulator mode, 16-bit index register mode
- decimal mode, native mode, 16-bit accumulator mode, 8-bit index register mode
- decimal mode, native mode, 16-bit accumulator mode, 16-bit index register mode

In emulation mode, both the accumulator (and memory) width and the index register width are always 8 bits, which is why there are 10 combinations (rather than 16 combinations). As on the 65C02 and NMOS 6502, ADC and SBC use binary arithmetic in binary mode, and BCD arithmetic in decimal mode. Whether instructions such as ASL, CMP, and CPX operate on 8-bit or 16-bit data depends on the accumulator (and memory) width (for ASL and CMP) or the index register width (for CPX). In emulation mode, as its name implies, the 65C816 emulates the operation of a 65C02 (albeit with the cycle counts of NMOS 6502). In native mode, the 65C816 can operate on 16-bit data, and the stack can be up to 64k (as compared to the 256 byte stack of the 65C02 and NMOS 6502).

The 65C816 is little endian, meaning the low byte is the lower address and the high(er) byte is the next address. For example, if:

- $123456 contains $AB
- $123457 contains $CD

then a LDA $123456 (when in 16-bit accumulator mode) puts $CDAB into the accumulator. This byte order applies everywhere, e.g. a LDA $123456 instruction assembles as $AF $56 $34 $12, JSR pushes the return address onto the stack in such a way that the lower 8 bits will be at at the lower address and the upper 8 bits will be at the higher address, etc.

The 65C02 and NMOS 6502 have a 64k address space, which is subdivided into 256 pages; each page, therefore is 256 bytes. In general, cycle counts are affected when a page boundary is crossed (e.g. LDA $1234,X takes 4 cycles when X is 0, but takes 5 cycles when X is $FF). Also, the zero page and the stack wrap at page boundaries (e.g. when X is 1, LDA $FF,X is equivalent to LDA $0000 rather than LDA $0100).

On the 65C816, a page is still defined as 256 bytes, and a bank is defined as 64k bytes. In emulation mode, the behavior of the 65C02 and NMOS 6502 is (generally) preserved, but in native mode, there is no wrapping at page boundaries, and page boundary crossings don't affect cycle counts in as many cases as the 65C02 and NMOS 6502. On the 65C816, bank boundary crossings don't affect cycle counts, but there are a number of things that wrap at bank boundaries. Specifics are described below (including exceptions to the general behavior mentioned above).

3 A NOTE ON 65C816 ASSEMBLERS AND IMMEDIATE DATA

Most microprocessors whose instructions can operate on two or more data widths (e.g. 8-bit data and 16-bit data) use different opcodes for different data widths (of the same operation, such as addition). By contrast, on the 65C816, the same opcode is used for both 8-bit and 16-bit data, and the data width is determined by a processor mode, rather than the opcode. For example, the instruction XOR $123456 (which assembles to the bytes $4F $56 $34 $12) is an 8-bit exclusive-or operation in 8-bit accumulator mode, and a 16-bit exclusive-or operation in 16-bit accumulator mode.

One ramification is that 8-bit immediate data and 16-bit immediate data also use the same opcode. For example, the opcode A9 is LDA immediate and the next one or two bytes will be the immediate data depending on the accumulator mode. So, in 16-bit accumulator mode, A9 01 0A is the 16-bit instruction LDA #$0A01, and in 8-bit accumulator mode A9 01 0A is a pair of 8-bit instructions, namely:

```
LDA #$01
ASL
```

Consequently, all 65C816 assemblers have the ability to explicitly specify the width of immediate data, since an instruction such as LDA #0 must assemble as two bytes if it is to be executed in 8-bit mode, and must assemble as three bytes if it is to be executed in 16-bit mode. However, different assemblers have different approaches for specifying immediate data width. Consider the following code:

```
           CLC            ; set native mode
           XCE            ; "
           REP #$30       ; 16-bit accumulator and index registers
           LDA #0         ; this needs to be assembled as three bytes
           JSR SUBROUTINE
           SEP #$30       ; 8-bit accumulator and index registers
           LDX #0         ; this needs to be assembled as two bytes
           RTS
; Note: this subroutine is CALLED from 16-bit index register mode
SUBROUTINE LDY #0         ; this needs to be assembled as three bytes
           RTS
```

One of the most common approaches is to have one or more assembler directives for specifying the width. For example:

- .m8 ; 8-bit accumulator immediate data
- .m16 ; 16-bit accumulator immediate data

- .x8 ; 8-bit index register immediate data
- .x16 ; 16-bit index register immediate data

or:

- .mx 16 ; 16-bit accumulator immediate data
- .mx ,8 ; 8-bit index register immediate data
- .mx 8,16 ; 8-bit accumulator, 16-bit index register immediate data

It is important to remember that these directives do not actually generate any code that will change the data width. They only tell the assembler whether subsequent immediate data should be assembled as two or three bytes.

You may find it useful to define a set of macros (with readable names) that generate a 65C816 instruction to switch data widths and issue an assembler directive. For example:

```
.macro acc8
       SEP #$20 ; put the 65C816 in 8-bit accumulator mode
      .m8       ; tell the assembler about it
.endmacro

.macro accindex8
       SEP #$30 ; put the 65C816 in 8-bit accumulator and index register mode
      .m8       ; tell the assembler about it
      .x8       ; "
.endmacro
```

and so on.

A variation of this approach is for the assembler to attempt to "follow along" and switch automatically when a REP or SEP instruction is assembled (it might also look for a SEC XCE sequence that puts the 65C816 in emulation mode which uses 8-bit data exclusively). The previous directive(s) are also provided to allow the user to override the assembler. For example in the earlier example code, an assembler can easily assemble the LDA and LDX instructions correctly, automatically switching to the correct width after encountering the REP and SEP instructions. However, correctly assembling the LDY instruction and cases like it can get far more complicated. For cases like that, it is usually up to the programmer to supply the correct assembler directive before the LDY to get the assembler into the correct mode; the assembler might simply switch to 8-bit mode at the SEP, then simply remain in 8-bit mode when it reached the LDY instruction. A "following along" feature can be convenient, but it can also lead to errors that are difficult to find.

A different approach is for the assembler to use different operand syntaxes for 8-bit immediate data and 16-bit immediate data. For example:

- LDA #0 ; the # means assemble a two byte instruction
- LDA ##0 ; the ## means assemble a three byte instruction

A common source of errors is assembling the wrong immediate data width. Be sure to consult the documentation for your assembler to learn how it handles immediate data. You may find it worthwhile to spend some time searching and/or experimenting to find an assembler that fits your programming style.

4 REGISTERS AND FLAGS

On the 65C02 and NMOS 6502 all registers and flags have unique one-letter names (with the exception of the Program Counter (PC)). On the 65C816, unfortunately, this is not the case. For example, there is a D register and a d flag. The D in the name of the CLD instruction refers to the D flag, but the D in the name of the TCD instruction refers to the D register. More unfortunate is that the term B register is ambiguous; it could refer to either of two registers: the B accumulator or the Data Bank Register (DBR). (As a further source of confusion, there is also a b flag.) For example, the B in the name of the XBA instruction refers to the B accumulator, but the B in the name of the PHB instruction refers to the DBR. The convention used in this document is to use lower case names for flags and upper case names for registers. Also, at the risk of being excessively verbose, terms such as "d flag" and "D register" will be used, in the belief that clarity is more helpful than succinctness. Likewise, in the interests of clarity, the terms "B accumulator" and "DBR" will be used to avoid ambiguity. Furthermore, there are times when it will be useful to refer specifically to the upper 8 bits and lower 8 bits of a 16-bit register, so terms will be defined for this purpose.

There are 9 registers. They are:

- The Accumulator (16 bits wide)
- The Data Bank Register (8 bits wide)
- The Direct register (16 bits wide)
- The program banK register (8 bits wide)
- The Program Counter (16 bits wide)
- The Processor status register (8 bits wide)
- The Stack pointer (16 bits wide)
- The X index register (16 bits wide)
- The Y index register (16 bits wide)

There are 10 flags. Each is one bit wide. They are:

- The Break flag
- The Carry flag
- The Decimal mode flag
- The Emulation mode flag
- The Interrupt disable flag
- The accumulator and Memory width flag
- The Negative flag
- The oVerflow flag
- The indeX register width flag
- The Zero flag

The terms used for these registers and flags will be:

- A accumulator: the lower 8 bits of the accumulator
- B accumulator: the upper 8 bits of the accumulator
- C accumulator: the 16-bit accumulator
- DBR: the data bank register
- D register: the 16-bit direct register
- DL register: the lower 8 bits of the direct register
- DH register: the upper 8 bits of the direct register
- K register: the program bank register
- PC: the 16-bit program counter
- PCL: the lower 8 bits of the program counter
- PCH: the upper 8 bits of the program counter
- P register: the processor status register
- S register: the 16-bit stack pointer
- SL register: the lower 8 bits of the stack pointer
- SH register: the upper 8 bits of the stack pointer
- X register: the 16-bit X index register
- XL register: the lower 8 bits of the X index register
- XH register: the upper 8 bits of the X index register
- Y register: the 16-bit Y index register
- YL register: the lower 8 bits of the Y index register
- YH register: The upper 8 bits of the Y index register

- b flag: the break flag
- c flag: the carry flag

- d flag: the decimal mode flag
- e flag: the emulation mode flag
- i flag: the interrupt disable flag
- m flag: the accumulator and memory width flag
- n flag: the negative flag
- v flag: the overflow flag
- x flag: the index register width flag
- z flag: the zero flag

The term "accumulator" without any additional qualifiers means the 16-bit accumulator (i.e. the C accumulator) when the m flag is 0, and the 8-bit accumulator (i.e. the A accumulator) when the m flag is 1. Furthermore, although "X register" and "Y register" refer to the 16-bit registers, when the x flag is 1, they are equivalent to the XL and YL registers, since the XH and YH register are forced to $00 when the x flag is 1. (See below.) In effect, "X register" means the 16-bit register when the x flag is 0, and the 8-bit register (i.e. the XL register) when the x flag is 1. Likewise for "Y register".

The P register contains 9 (yes, 9) of the flags.

- P register bit 7: n flag
- P register bit 6: v flag
- P register bit 5: m flag (native mode)
- P register bit 4: x flag (native mode), b flag (emulation mode)
- P register bit 3: d flag
- P register bit 2: i flag
- P register bit 1: z flag
- P register bit 0: c flag

The e flag is separate from the P register but is accessed via the c flag using the XCE instruction (details below).

How a particular instruction specifically affect the flags will be given in the description of that instruction below. Here is a general overview of the meaning of each flag. Note that for some instructions, the meaning of a flag (or flags) will be different than described in this brief overview. See the instruction description for the full details.

The n flag is 1 when the result is negative; in other words, when the high bit (bit 7 for an 8-bit result, bit 15 for a 16-bit result) is 1. The n flag is 0 when the result is non-negative (zero or positive), i.e. when the high bit of the result is 0.

The v flag is 1 when an arithmetic overflow occurs. The v flag is 0 if there was no arithmetic overflow.

When the m flag is 0, the accumulator (and memory) width is 16 bits. When the m flag is 1, the accumulator (and memory) width is 8 bits. One way to remember what's what is that the even flag value (i.e. 0) is the width that is an even number of bytes (i.e. 2), and the odd flag value is an odd number of bytes.

Likewise, when the x flag is 0, the index register width is 16 bits. When the x flag is 1, the index register width is 8 bits. Again, the even flag value is an even number of bytes and the odd flag value is an odd number of bytes. One important difference from the m flag is that when the x flag is 1 (8-bit index registers), the XH register and the YH register are both forced to $00. This means that after:

```
CLC
XCE        ; native mode
REP #$10   ; 16-bit index registers
LDX #$1234
LDY #$5678
SEP #$10   ; 8-bit index registers
REP #$10   ; 16-bit index registers
```

the X register will be $0034 and the Y register will be $0078. Thus, a subroutine that saves the X register and/or the Y register should do so BEFORE setting the x flag to 1. (As noted above, this is not the case for the m flag. The B accumulator is not forced to zero when the m flag is 1.) Attempting to change the value of the XH register or the YH register when the x flag is 1 will have no effect on XH or YH.

On the 65C02 and NMOS 6502, the b flag is used for distinguishing an IRQ (hardware) interrupt from a BRK (software) interrupt. On the 65C816, in emulation mode, it is still used for this purpose. In native mode, there are separate vectors for BRK and IRQ, so the b flag is not necessary.

When the d flag is 0, the ADC and SBC instructions perform binary arithmetic. When the d flag is 1, the ADC and SBC instructions perform BCD arithmetic.

When the i flag is 0, (IRQ) interrupts are enabled. When the i flag is 1, (IRQ) interrupts are disabled. Note that IRQ is in fact the only interrupt that can be disabled by the i flag.

- The z flag is 1 when the result is zero. The z flag is 0 when the result is non-zero.

- The c flag is 1 when there is an arithmetic carry (carry has the same meaning that "carrying the one" meant when you learned addition in school). The c flag is 0 when where was not an arithmetic carry.

When the e flag is 0, the 65C816 is in native mode. When the e flag is 1, the 65C816 is in emulation mode. When the e flag is 1, the SH register is forced to $01, the m flag is forced to 1, and the x flag is forced to 1. As a consequence of the x flag being forced to 1, the XH register and the YH register are forced to $00 (see above). This means that after:

```
CLC
XCE        ; native mode
REP #$30   ; 16-bit accumulator/memory and 16-bit index registers
LDX #$1234
LDY #$5678
LDA #$ABCD
TCS        ; set S register to $ABCD
SEC
XCE        ; emulation mode
CLC
XCE        ; native mode
```

the S register will be $01CD, the X register will be $0034, the Y register will be $0078, and m flag will be 1, and the x flag will be 1. The fact that the SH register gets forced to $01 is especially important if native mode code is calling emulation mode code. Attempting to change the value of the SH register, the XH register, the YH register, the m flag, or the x flag when the e flag is 1 will have no effect.

Register usage will be given when the addressing modes and the instructions are discussed. However, the Program Counter (PC) deserves mention here.

Note that although the 65C816 has a 24-bit address space, the Program Counter is only a 16-bit register and the Program Bank Register is a separate (8-bit) register. This means that instruction execution wraps at bank boundaries. This is true even if the bank boundary occurs in the middle of the instruction.

For example, if an INX instruction at $12FFFF is executed, the next instruction to be executed will be at instruction at $120000 rather than $130000.

Likewise, if the memory contents are as follows:

- $120000 contains $20
- $12FFFF contains $C2 (a REP instruction)
- $130000 contains $30

then the second byte of the REP instruction at $12FFFF is $20 (at $120000) rather than $30 (at $130000); the next instruction to be executed will be at $120001, of course.

It's also worth noting that branches (both forward and backward) wrap at bank boundaries as well. A BCC $FFE0 instruction at $130020 will branch to $13FFC0 rather than $12FFC0. Likewise, a BRL $2000 at

$13E000 will branch to $132000 rather than $142000.

5 ADDRESSING MODES

The official names of the addressing modes of the 65C816 can be confusing and include the seemingly paradoxical Direct Indirect, Direct Indexed Indirect, and Direct Indirect Indexed. (In fact, the latter two addressing modes use different index registers, even though their names do not suggest this.) For this reason, in the interests of clarity, this document will use addressing mode names that more closely correspond with the typical assembler operand syntax. Terms such as direct, absolute, long, and relative will be use to distinguish operand width and whether the (object) code generated is the address itself or whether it is relative to the value of the Program Counter.

5.1 PAGE AND BANK BOUNDARY WRAPPING

Although each addressing mode will be described in detail below, there are a few rules of thumb that you can use to derive whether page and/or bank boundaries are crossed or not.

5.1.1 PAGE BOUNDARY WRAPPING

Page boundary wrapping only occurs in emulation mode, and only for "old" instructions and addressing modes, i.e. instructions and addressing modes that are available on the 65C02. Page boundary wrapping only occurs in the following situations:

- A. When the DL register is $00 (and in emulation mode -- both conditions must be met), the direct page wraps at a page boundary
- B. In emulation mode, the stack wraps at the page 1 boundary

An example of case A is when the D register is $0000 (and thus DL is $00), the X register is $0001, and the e flag is 1 (i.e. emulation mode); in that case the memory location read by LDA $FF,X is $000000 rather than $000100.

An example of case B is when the S register is $01FF and the e flag is 1; in that case, a PLA reads (i.e. pulls) from memory location $000100 rather than $000200.

Note that because stack,S addressing is a "new" addressing mode (i.e. this addressing mode was not available on the 65C02), it does not wrap at a bank boundary under any circumstances. Likewise, since PEI is a "new" instruction, PEI $FF does not wrap at a page boundary (either the direct page part, or the (pushing onto the) stack part).

5.1.2 BANK BOUNDARY WRAPPING

Bank boundary wrapping occurs in both native and emulation mode (and does not depend on which mode the 65C816 is in). The following are confined to bank 0 ("confined to" means they address bank 0 and wrap at the bank 0 boundary):

- A. The direct page
- B. The stack
- C. [absolute] and (absolute) addressing modes (JMP is the only instruction available for either addressing mode)

The following are confined to bank K:

- A. (absolute,X) addressing mode (JMP and JSR are the only instructions available for this addressing mode)
- B. The Program Counter (i.e. the PC register); again, this means branches wrap at the bank K boundary

source,destination addressing (i.e. the MVN and MVP instructions) wraps at both the source and destination bank boundaries.

Otherwise, wrapping does not occur at bank boundaries.

5.1.3 NOTATION

To help illustrate when and if page and bank boundary wrapping occurs, a special notation is used in the diagrams below to indicate when, for example, adding an index register to an address should be considered a 24-bit calculation, when it should be truncated at 16 bits, and when it should be truncated at 8 bits.

A 24-bit calculation is represented by a "triple wide" box:

```
+----------+----------+----------+
!            $HHMMLL+X           !
+----------+----------+----------+
```

If $HHMMLL is $00FFEE and X is $44, then the address is $01 (high byte), $00 (middle byte), $32 (low byte).

A calculation that is truncated at 16 bits is represented by a "double wide" box:

```
+----------+----------+----------+
!    0     !        $LL+S        !
+----------+----------+----------+
```

If S is $FFEE and $LL is $44, then the address is $00 (high byte), $00 (middle byte), $32 (low byte).

A calculation that is truncated at 8 bits is represented by a "single wide" box:

```
+----------+----------+----------+
!    0     !    DH    !  $LL+X   !
+----------+----------+----------+
```

If DH is $FF, $LL is $EE, and X is $44 then the address is $00 (high byte), $FF (middle byte), $32 (low byte).

Note that all addressing mode formulas in this document use unsigned arithmetic. (Yes, relative displacements are actually signed, but that will be covered later in the section dealing with relative addressing.)

Each calculation is labelled with the address being represented. For example:

```
+----------+----------+----------+
!            $HHMMLL+X           ! data lo
+----------+----------+----------+
```

shows the formula for the address of the low byte of the data (in this case, for long,X addressing).

- 24-bit quantities have high, middle, and low bytes.
- 16-bit quantities have only high and low bytes.
- 8-bit quantities have only low bytes.

For example, when the x flag is 1, then LDX loads only 8 bits of data, and the formula for the high byte would not apply in that instance.

There are two addresses for indirect addressing modes: the address of the pointer, and the address of the data. For example, if the D register is $0000, then the address of the pointer of LDA ($12) is $000012. If the DBR is $AB, and:

- $000012 contains $EF
- $000013 contains $CD

then the address of the (low byte of the) data is $ABCDEF. So, for example:

```
+-----------+-----------+-----------+
!     0     !         D+$LL         ! pointer lo
+-----------+-----------+-----------+
```

shows the formula for the address of the low byte of the pointer. In addition, the following quantities are used in the formulas:

- $ll = the low byte of the pointer (i.e. the byte at address "pointer lo")
- $mm = the middle byte of the pointer (i.e. the byte at address "pointer mid")
- $hh = the high byte of the pointer (i.e. the byte at address "pointer hi")
- $rr = the (value of the) DBR

In the previous LDA ($12) example, $ll would be $EF (the value of the low byte of the pointer). D, DH, K, X, and Y are, as you might expect, the value of the D, DH, K, X, and Y registers respectively. $OP is not used in the formulas, but simply indicates the opcode byte. For example, the bytes of an instruction with absolute addressing are $OP $LL $HH. This simply means that the instruction has three bytes: an opcode, a operand low byte, and an operand high byte.

Note that $rrHHLL is a 24-bit value whose high byte is $rr, whose middle byte is $HH, and whose low byte is $LL. This is analogous to $223344 whose high, middle, and low bytes are $22, $33, and $44. This allows 16-bit and 24-bit quantities to be easily composed from 8-bit quantities and makes the formulas much more succinct than they would be if ($HH << 8) | $LL or 256 * $HH + $LL were used.

Although a lowercase l can be confused with a 1, you should not confuse $ll (double lowercase L) in the formulas with $11 (a pair of ones, i.e. 17 in decimal), since $11 is not used anywhere in any of the formulas. Likewise, $rr was chosen for the value of the DBR rather than the seemingly more natural $bb, to avoid any confusion with the hexadecimal value $BB (i.e. 187 in decimal).

However, case does matter; for example, $HH and $hh are different quantities. (Author's note: this notation is an imperfect solution; it seemed like composing $hhll from $hh and $ll made the formulas more readable than composing $hilo from $hi and $lo or composing $phpl from $ph (pointer hi) and $pl.)

5.2 ABSOLUTE

Length: 3 bytes, $OP $LL $HH

For JMP and JSR, the jump destination address is:

```
+-----------+-----------+-----------+
!     K     !         $HHLL         ! destination
+-----------+-----------+-----------+
```

For all other instructions, the address of the data is:

```
+-----------+-----------+-----------+
!              $rrHHLL              ! data lo
+-----------+-----------+-----------+
+----------+----------+----------+
!            $rrHHLL+1           ! data hi
+----------+----------+----------+
```

Example 1: If the K register is $12, then JMP $FFFF jumps to $12FFFF

Example 2: If the DBR is $12 and the m flag is 0, then LDA $FFFF loads the low byte of the data from address $12FFFF, and the high byte from address $130000

5.3 ABSOLUTE,X AND ABSOLUTE,Y

Length: 3 bytes, $OP $LL $HH

For absolute,X addressing the address of the data is:

```
+----------+----------+----------+
!            $rrHHLL+X           ! data lo
+----------+----------+----------+

+----------+----------+----------+
!           $rrHHLL+X+1          ! data hi
+----------+----------+----------+
```

absolute,Y addressing is the same, with Y substituted for X; the address of the data is:

```
+----------+----------+----------+
!            $rrHHLL+Y           ! data lo
+----------+----------+----------+

+----------+----------+----------+
!           $rrHHLL+Y+1          ! data hi
+----------+----------+----------+
```

Example: If the DBR is $12, the X register is $000A, and the m flag is 0, then LDA $FFFE,X loads the low byte of the data from address $130008, and the high byte from address $130009

Note that this is one of the rare instances where emulation mode has different behavior than the 65C02 or NMOS 6502. Since the 65C02 and NMOS 6502 have a 16-bit address space, when the X register is $80, an LDA $FFC0,X instruction (for example) loads from address $0040; however on the 65C816, it loads from address $010040 (rather than address $000040). In practice, this is not a problem since 65C02 and NMOS 6502 code would almost certainly use an LDA $C0,X instruction (rather than LDA $FFC0,X) because zero page addressing always wraps at the page boundary on a 65C02 and NMOS 6502 (i.e. when the X register is $80, LDA $C0,X loads from address $40).

5.4 (ABSOLUTE) AND [ABSOLUTE]

Length: 3 bytes, $OP $LL $HH

(absolute) addressing uses a 16-bit pointer. The address of the pointer is:

```
+----------+----------+----------+
!     0    !         $HHLL       ! pointer lo
+----------+----------+----------+

+----------+----------+----------+
!     0    !        $HHLL+1      ! pointer hi
+----------+----------+----------+
```

JMP is the only instruction that uses this addressing mode; the jump destination address is:

```
+-----------+----------+----------+
!     K     !         $hhll         ! destination
+-----------+----------+----------+
```

[absolute] addressing is similar, but it uses a 24-bit pointer instead. The address of the pointer is:

```
+-----------+----------+----------+
!     0     !         $HHLL         ! pointer lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !        $HHLL+1        ! pointer mid
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !        $HHLL+2        ! pointer hi
+-----------+----------+----------+
```

Likewise, JMP is the only instruction that uses this addressing mode; the jump destination address is:

```
+-----------+----------+----------+
!    $hh    !         $mmll         ! destination
+-----------+----------+----------+
```

Example: If the K register is $12 and

- $000000 contains $34
- $00FFFF contains $56

then JMP ($FFFF) jumps to $123456

(Some assemblers allow the syntax JML (absolute) which is equivalent to JMP [absolute]; however, the use of the latter syntax is more consistent with syntax of the [direct] and (direct) addressing modes, where parentheses indicate a 16-bit pointer and square brackets indicate a 24-bit pointer.)

Note that on the 65C816, as on the 65C02, (absolute) addressing does not wrap at a page boundary, i.e. for a JMP ($12FF) the low byte of the destination address is taken from $12FF and the high byte of the destination address is taken from $1300. On the NMOS 6502, (absolute) addressing did wrap on a page boundary, which was unintentional (i.e. a bug); there, a JMP ($12FF) took the low byte of the destination address from $12FF but took the high byte of the destination address from $1200 (rather than $1300).

5.5 (ABSOLUTE,X)

Length: 3 bytes, $OP $LL $HH

(absolute,X) addressing uses a 16-bit pointer. The address of the pointer is:

```
+-----------+----------+----------+
!     K     !        $HHLL+X        ! pointer lo
+-----------+----------+----------+

+-----------+----------+----------+
!     K     !       $HHLL+X+1       ! pointer hi
+-----------+----------+----------+
```

JMP and JSR are the only instructions that use this addressing mode; the jump destination address is:

```
+-----------+----------+----------+
!     K     !         $hhll         ! destination
+-----------+----------+----------+
```

Example: If the K register is $12, the X register is $000A, and

- $120008 contains $56
- $120009 contains $34

then JMP ($FFFE,X) jumps to $123456

5.6 ACCUMULATOR

Length: 1 byte, $OP

The accumulator addressing mode is used by the instructions ASL, DEC, INC, LSR, ROL, and ROR when the operand is the accumulator rather than a memory location. (The distinction between this addressing mode and implied addressing is that the six instructions with this addressing mode have multiple addressing modes, but the instructions with implied addressing have only one addressing mode.) Different assemblers allow or require slightly different syntaxes. For example:

```
ASL
ASLA
ASL A
```

are all equivalent, but an assembler may require or permit any of these syntaxes. (For this reason, many 6502 family assemblers reserve the one letter label A, and do not allow it to be used in source code.)

5.7 DIRECT

Length: 2 bytes, $OP $LL

For "old" instructions (PEI is the only "new" instruction for this addressing mode), when the e flag is 1 and the DL register is $00 (all three conditions -- the instruction, the e flag, and the DL register -- must be met), the address of the data is:

```
+-----------+-----------+-----------+
!     0     !    DH     !    $LL    ! data lo
+-----------+-----------+-----------+
```

Otherwise, the address of the data is:

```
+-----------+-----------+-----------+
!     0     !         D+$LL         ! data lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !        D+$LL+1        ! data hi
+-----------+-----------+-----------+
```

Example 1: If the D register is $FF00 and the e flag is 1 (note that this means the m flag must be 1), then LDA $FF loads the low byte of the data from address $00FFFF

Example 2: If the D register is $FF00 and the m flag is 0 (note that this means the e flag must be 0), then LDA $FF loads the low byte of the data from address $00FFFF, and the high byte from address $000000

5.8 DIRECT,X AND DIRECT,Y

Length: 2 bytes, $OP $LL

For direct,X addressing, when the e flag is 1 and the DL register is $00 (both conditions must be met), the address of the data is:

```
+-----------+----------+----------+
!     0     !    DH    !   $LL+X  ! data lo
+-----------+----------+----------+
```

Otherwise, for direct,X addressing, the address of the data is:

```
+-----------+----------+----------+
!     0     !        D+$LL+X      ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !       D+$LL+X+1     ! data hi
+-----------+----------+----------+
```

direct,Y addressing is the same, with Y substituted for X; when the e flag is 1 and the DL register is $00 (both conditions must be met), the address of the data is:

```
+-----------+----------+----------+
!     0     !    DH    !   $LL+Y  ! data lo
+-----------+----------+----------+
```

Otherwise, for direct,Y addressing, the address of the data is:

```
+-----------+----------+----------+
!     0     !        D+$LL+Y      ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !       D+$LL+Y+1     ! data hi
+-----------+----------+----------+
```

Example 1: If the D register is $FF00, the X register is $000A, and the e flag is 1 (note that this means the m flag must be 1), then LDA $FE,X loads the low byte of the data from address $00FF08

Example 2: If the D register is $FF00, the X register is $000A, and the m flag is 0 (note that this means the e flag must be 0), then LDA $FE,X loads the low byte of the data from address $000008, and the high byte from address $000009

5.9 (DIRECT)

Length: 2 bytes, $OP $LL

(direct) addressing uses a 16-bit pointer. When the e flag is 1 and the DL register is $00 (both conditions must be met), the address of the pointer is:

```
+-----------+----------+----------+
!     0     !    DH    !    $LL   ! pointer lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !    DH    !   $LL+1  ! pointer hi
+-----------+----------+----------+
```

Otherwise, the address of the pointer is:

```
+-----------+----------+----------+
!     0     !         D+$LL       ! pointer lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !        D+$LL+1      ! pointer hi
```

```
+-----------+----------+----------+
```

Unlike the address of the pointer, the address of the data does not depend on the value of the e flag, or the value of the DL register (or any other conditions, for that matter). The address of the data is always:

```
+-----------+----------+----------+
!              $rrhhll            ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!             $rrhhll+1           ! data hi
+-----------+----------+----------+
```

Note that this means that the address of the pointer wraps at a bank boundary, but the address of the data does not.

Example 1: if the D register is $FF00 and the e flag is 1 (note this means the m flag must be 1), then for LDA ($FF), the address of the low byte of the pointer is $00FFFF and the address of the high byte is $00FF00. Furthermore, if the DBR is $12 and

- $00FF00 contains $FF
- $00FFFF contains $FF

then LDA ($FF) loads the low byte of the data from address $12FFFF.

Example 2: if the D register is $FF00 and the m flag is 0 (note this means the e flag must be 0), then for LDA ($FF), the address of the low byte of the pointer is $00FFFF and the address of the high byte is $000000. Furthermore, if the DBR is $12 and

- $000000 contains $FF
- $00FFFF contains $FF

then LDA ($FF) loads the low byte of the data from address $12FFFF, and the high byte from $130000.

5.10 [DIRECT]

Length: 2 bytes, $OP $LL

[direct] addressing uses a 24-bit pointer. The address of the pointer is:

```
+-----------+----------+----------+
!     0     !         D+$LL       ! pointer lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !        D+$LL+1      ! pointer mid
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !        D+$LL+2      ! pointer hi
+-----------+----------+----------+
```

The address of the data is:

```
+-----------+----------+----------+
!              $hhmmll            ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!             $hhmmll+1           ! data hi
+-----------+----------+----------+
```

Again, this means that the address of the pointer wraps at a bank boundary, but the address of the data does not.

Example: if the D register is $FF00 and the m flag is 0, then for LDA [$FE], the address of the low byte of the pointer is $00FFFE, the address of the middle byte is $00FFFF, and the address of the high byte is $000000. Furthermore, if

- $000000 contains $12
- $00FFFE contains $FF
- $00FFFF contains $FF

then LDA [$FE] loads the low byte of the data from address $12FFFF, and the high byte from $130000.

5.11 (DIRECT,X)

Length: 2 bytes, $OP $LL

(direct,X) addressing uses a 16-bit pointer. When the e flag is 1 and the DL register is $00 (both conditions must be met), the address of the pointer is:

```
+-----------+-----------+-----------+
!     0     !    DH     !   $LL+X   ! pointer lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !    DH     !  $LL+X+1  ! pointer hi
+-----------+-----------+-----------+
```

Otherwise, the address of the pointer is:

```
+-----------+-----------+-----------+
!     0     !        D+$LL+X        ! pointer lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !       D+$LL+X+1       ! pointer hi
+-----------+-----------+-----------+
```

Again, unlike the address of the pointer, the address of the data does not depend on the value of the e flag, or the value of the DL register (or any other conditions, for that matter). The address of the data is always:

```
+-----------+-----------+-----------+
!             $rrhhll               ! data lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!            $rrhhll+1              ! data hi
+-----------+-----------+-----------+
```

Again, this means that the address of the pointer wraps at a bank boundary, but the address of the data does not.

Example 1: if the D register is $FF00, the X register is $000A, and the e flag is 1 (note that this means the m flag must be 1), then for LDA ($FE,X), the address of the low byte of the pointer is $00FF08 and the address of the high byte is $00FF09. Furthermore, if the DBR is $12 and

- $00FF08 contains $FF
- $00FF09 contains $FF

then LDA ($FE,X) loads the low byte of the data from address $12FFFF.

Example 2: if the D register is $FF00, the X register is $000A, and the m flag is 0 (note that this means the e flag must be 0), then for LDA ($FE,X), the address of the low byte of the pointer is $000008 and the address of the high byte is $000009. Furthermore, if the DBR is $12 and

- $000008 contains $FF
- $000009 contains $FF

then LDA ($FE,X) loads the low byte of the data from address $12FFFF, and the high byte from $130000.

5.12 (DIRECT),Y

Length: 2 bytes, $OP $LL

(direct),Y addressing uses a 16-bit pointer. When the e flag is 1 and the DL register is $00 (both conditions must be met), the address of the pointer is:

```
+-----------+-----------+-----------+
!     0     !    DH     !    $LL    ! pointer lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !    DH     !   $LL+1   ! pointer hi
+-----------+-----------+-----------+
```

Otherwise, the address of the pointer is:

```
+-----------+-----------+-----------+
!     0     !         D+$LL         ! pointer lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !        D+$LL+1        ! pointer hi
+-----------+-----------+-----------+
```

Again, unlike the address of the pointer, the address of the data does not depend on the value of the e flag, or the value of the DL register (or any other conditions, for that matter). The address of the data is always:

```
+-----------+-----------+-----------+
!             $rrhhll+Y             ! data lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!            $rrhhll+Y+1            ! data hi
+-----------+-----------+-----------+
```

Again, this means that the address of the pointer wraps at a bank boundary, but the address of the data does not.

Again, note that this is one of the rare instances where emulation mode has different behavior than the 65C02 or NMOS 6502. Since the 65C02 and NMOS 6502 have a 16-bit address space, if the address of the data were $FFFE+Y and the Y register were $0A, the address of the data would be $0008. On the 65C816, the address of the data would be $010008 (assuming the DBR was $00). In practice, this is typically not a problem, since code written for the 65C02 or NMOS 6502 would almost never use a pointer that would wrap at the 16-bit address space boundary like that.

Example 1: if the D register is $FF00 and the e flag is 1 (note that this means the m flag must be 1), then for LDA ($FF),Y, the address of the low byte of the pointer is $00FFFF and the address of the high byte is $00FF00. Furthermore, if the DBR is $12, the Y register is $000A, and

- $00FF00 contains $FF
- $00FFFF contains $FE

then LDA ($FF),Y loads the low byte of the data from address $130008.

Example 2: if the D register is $FF00 and the m flag is 0 (note that this means the e flag must be 0), then for LDA ($FF),Y, the address of the low byte of the pointer is $00FFFF and the address of the high byte is $000000. Furthermore, if the DBR is $12, the Y register is $000A, and

- $000000 contains $FF
- $00FFFF contains $FE

then LDA ($FF),Y loads the low byte of the data from address $130008, and the high byte from $130009.

5.13 [DIRECT],Y

Length: 2 bytes, $OP $LL

[direct],Y addressing uses a 24-bit pointer. The address of the pointer is:

```
+-----------+-----------+-----------+
!     0     !         D+$LL         ! pointer lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !        D+$LL+1        ! pointer mid
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !        D+$LL+2        ! pointer hi
+-----------+-----------+-----------+
```

The address of the data is:

```
+-----------+-----------+-----------+
!             $hhmmll+Y             ! data lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!            $hhmmll+Y+1            ! data hi
+-----------+-----------+-----------+
```

Again, this means that the address of the pointer wraps at a bank boundary, but the address of the data does not.

Example: if the D register is $FF00 and the m flag is 0, then for LDA [$FE],Y, the address of the low byte of the pointer is $00FFFE, the address of the middle byte is $00FFFF, and the address of the high byte is $000000. Furthermore, if the Y register is $000A, and

- $000000 contains $12
- $00FFFE contains $FC
- $00FFFF contains $FF

then LDA [$FE],Y loads the low byte of the data from address $130006, and the high byte from $130007.

5.14 IMMEDIATE

Length: 2 bytes, $OP $LL (for 8-bit data)

Length: 3 bytes, $OP $LL $HH (for 16-bit data)

$LL (or $HHLL for 16-bit data) is the data; thus, the address of the data is:

```
+-----------+----------+----------+
!     K     !         PC+1         ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!     K     !         PC+2         ! data hi
+-----------+----------+----------+
```

The width of the immediate data depends on the instruction. There are four cases.

- A. The data width depends on the value of the m flag (e.g. LDA)
- B. The data width depends on the value of the x flag (e.g. LDX)
- C. The data is 8 bits wide (e.g. REP)
- D. The data is 16 bits wide (e.g. PEA)

Incidentally, PER is an unusual case. It can be considered 16-bit immediate data, like PEA. Unlike PEA (which pushes the immediate data onto the stack), PER adds the immediate data to the address of the next instruction. This is the same formula that relative16 addressing uses for the destination address, and thus PER is often documented as relative16 addressing rather than immediate addressing. See the section on the PER instruction for further details.

5.15 IMPLIED

Length: 1 byte, $OP

Instructions with implied addressing (e.g. CLC) have no operand. The name "implied" comes from the fact that the instruction itself implies which register, flag, or memory location will be used by the instruction.

Note that even though all instructions with implied addressing have only one addressing mode, the converse is not true; not all instructions with only one addressing mode are implied addressing. For example. BCC and JSL each have only one addressing mode, yet are not implied addressing.

5.16 LONG

Length: 4 bytes, $OP $LL $MM $HH

For JMP, the jump destination address is:

```
+-----------+----------+----------+
!              $HHMMLL             ! destination
+-----------+----------+----------+
```

For all other instructions, the address of the data is:

```
+-----------+----------+----------+
!              $HHMMLL             ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!             $HHMMLL+1            ! data hi
+-----------+----------+----------+
```

Example: If the m flag is 0, then LDA $12FFFF loads the low byte of the data from address $12FFFF, and the high byte from address $130000.

5.17 LONG,X

Length: 4 bytes, $OP $LL $MM $HH

The address of the data is:

```
+-----------+-----------+-----------+
!             $HHMMLL+X             ! data lo
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!            $HHMMLL+X+1            ! data hi
+-----------+-----------+-----------+
```

Example: If the X register is $000A and the m flag is 0, then LDA $12FFFE,X loads the low byte of the data from address $130008, and the high byte from address $130009.

5.18 RELATIVE8 AND RELATIVE16

Length: 2 bytes, $OP $LL (for relative8 addressing)

Length: 3 bytes, $OP $LL $HH (for relative16 addressing)

For relative8 addressing, $LL is a signed 8-bit branch displacement, where $00 to $7F is a branch distance of 0 to 127, and $80 to $FF is a branch distance of -128 to -1. Note that PC is the address of $OP (i.e. the address of the relative8 instruction), whereas the branch distance is actually the distance from the next instruction; hence the formula uses PC+2 rather than PC. Treating the branch displacement as a signed number (this formula is also valid when treating the branch displacement as an unsigned number and $00 <= $LL <= $7F, i.e. $LL ranges from 0 to 127), the branch destination address is:

```
+-----------+-----------+-----------+
!     K     !       PC+2+$LL        !
+-----------+-----------+-----------+
```

When treating the branch displacement as unsigned number and $80 <= $LL <= $FF, (i.e. $LL ranges from 128 to 255), the branch destination address is:

```
+-----------+-----------+-----------+
!     K     !      PC-254+$LL       !
+-----------+-----------+-----------+
```

relative16 addressing is similar, but with a 16-bit branch displacement instead of an 8-bit displacement. Likewise, for relative16 addressing, the displacement is signed, but as it happens, you get the same result whether treat the displacement as signed or unsigned, so there is a single formula. BRL is the only instruction with this addressing mode (see the discussion of PER in the section on immediate addressing). The branch destination address is:

```
+-----------+-----------+-----------+
!     K     !      PC+3+$HHLL       !
+-----------+-----------+-----------+
```

5.19 SOURCE,DESTINATION

Length: 3 bytes, $OP $TT $SS

This is the addressing mode of the MVN and MVP instructions. The move source address is:

```
+----------+----------+----------+
!   $SS    !          X          !
+----------+----------+----------+
```

The move destination address is:

```
+----------+----------+----------+
!   $TT    !          Y          !
+----------+----------+----------+
```

Example: If the accumulator is $0002, the X register is $FFFE, and the Y register is $FFFF, then MVN #$12,#$34 moves

- the data at $12FFFE to $34FFFF
- the data at $12FFFF to $340000
- the data at $120000 to $340001

5.20 STACK,S

Length: 2 bytes, $OP $LL

The address of the data is:

```
+----------+----------+----------+
!    0     !        $LL+S        ! data lo
+----------+----------+----------+

+----------+----------+----------+
!    0     !       $LL+S+1       ! data hi
+----------+----------+----------+
```

Example: If the S register is $FF10 and the m flag is 0, then LDA $FA,S loads the low byte of the data from address $00000A, and the high byte from address $00000B.

5.21 (STACK,S),Y

Length: 2 bytes, $OP $LL

(stack,S),Y addressing uses a 16-bit pointer. The address of the pointer is:

```
+----------+----------+----------+
!    0     !        $LL+S        ! pointer lo
+----------+----------+----------+

+----------+----------+----------+
!    0     !       $LL+S+1       ! pointer hi
+----------+----------+----------+
```

The address of the data is:

```
+----------+----------+----------+
!            $rrhhll+Y           ! data lo
+----------+----------+----------+

+----------+----------+----------+
!           $rrhhll+Y+1          ! data hi
+----------+----------+----------+
```

Again, this means that the address of the pointer wraps at a bank boundary, but the address of the data does not.

Example: if the S register is $FF10 and the m flag is 0, then for LDA ($FA,S),Y, the address of the low byte of the pointer is $00000A and the address of the high byte is $00000B. Furthermore, if the DBR is $12, the Y register is $0050, and

- $00000A contains $F0
- $00000B contains $FF

then LDA ($FA,S),Y loads the low byte of the data from address $130040, and the high byte from $130041.

5.22 PAGE AND BANK BOUNDARY WRAPPING OF THE STACK

The previous sections covered the addressing of the instruction operands. However, some instructions push onto, and pull from, the stack. This section covers how the stack itself is addressed. It is not really an addressing mode, but it uses the same notation as the addressing modes did.

Instructions can push or pull 8-bit (e.g. PHK and PLP), 16-bit (e.g. JSR and PEA), or 24-bit (e.g. JSL) quantities. It is important to keep in mind that in this section, software (e.g. BRK) and hardware (e.g. NMI) interrupts are treated as two separate pushes: the address (16 bits in emulation mode, 24 bits in native mode), and the P register (8 bits), rather than a single (24-bit or 32-bit) push. Likewise, RTI is treated as two separate pulls: the P register and the address.

Note that in these formulas, S and SL refer to values of the S and SL registers before the instruction was executed. After pushing N bytes:

- In emulation mode, SL will be decremented N times
- In native mode, S will be decremented N times

Likewise, after pulling N bytes:

- In emulation mode, SL will be incremented N times
- In native mode, S will be incremented N times

For all interrupts and "old" instructions, when the e flag is 1, the address of the data for an 8-bit push is:

```
+-----------+-----------+-----------+
!     0     !     1     !    SL     ! data lo
+-----------+-----------+-----------+
```

Otherwise, the address of the data for an 8-bit push is:

```
+-----------+-----------+-----------+
!     0     !           S           ! data lo
+-----------+-----------+-----------+
```

For all interrupts and "old" instructions, when the e flag is 1, the address of the data for a 16-bit push is:

```
+-----------+-----------+-----------+
!     0     !     1     !    SL     ! data hi
+-----------+-----------+-----------+

+-----------+-----------+-----------+
!     0     !     1     !   SL-1    ! data lo
+-----------+-----------+-----------+
```

Otherwise, the address of the data for a 16-bit push is:

```
+-----------+-----------+-----------+
!     0     !           S           ! data hi
+-----------+-----------+-----------+
+-----------+----------+----------+
!     0     !          S-1         ! data lo
+-----------+----------+----------+
```

The address of the data for a 24-bit push is:

```
+-----------+----------+----------+
!     0     !           S          ! data hi
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !          S-1         ! data mid
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !          S-2         ! data lo
+-----------+----------+----------+
```

For all interrupts and "old" instructions, when the e flag is 1, the address of the data for a pull is:

```
+-----------+----------+----------+
!     0     !     1    !   SL+1    ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !     1    !   SL+2    ! data hi
+-----------+----------+----------+
```

Otherwise, the address of the data for a pull is:

```
+-----------+----------+----------+
!     0     !          S+1         ! data lo
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !          S+2         ! data mid
+-----------+----------+----------+

+-----------+----------+----------+
!     0     !          S+3         ! data hi
+-----------+----------+----------+
```

5.23 A NOTE ON OPERAND ADDRESSES

On the 65C02 and NMOS 6502, LDA $12 (direct addressing) and LDA $0012 (absolute addressing) load data from the same address. On the 65C816, this is not always the case. For example, if the DBR is $12, the D register is $3400, and the m flag is 1, then

```
LDA $56 (direct addressing) loads data from $003456
LDA $0056 (absolute addressing) loads data from $120056
LDA $000056 (long addressing) loads data from $000056
```

When encountering an operand address whose high byte is $00, almost all 65C02 and NMOS 6502 assemblers will use zero page addressing rather than absolute addressing as an optimization, since this usually saves a byte and a cycle. It is rarely necessary to override this optimization (i.e. to tell the assembler to use absolute addressing).

However, on the 65C816, it is necessary to pay much closer attention to the addressing mode being assembled, as the example above illustrates. An assembler that assembles code for the 65C816, as well as one (or both) of its 8-bit predecessors may perform the aforementioned optimization even for the 65C816, which may be not what you wanted or intended. It may be worthwhile to review the documentation of your assembler to see how to explicitly specify whether direct, absolute, or long addressing will be used.

6 INSTRUCTIONS

The columns of the tables that describe the instructions are:

```
OP - the opcode in hex
LEN - the length of the instruction in bytes
CYCLES - the number of cycles the instruction takes
MODE - the addressing mode of the instruction
nvmxdizc e - the flags affected by the instruction
SYNTAX - an example of the assembler syntax of the instruction
```

In the LEN column:

- m = m flag
- x = x flag

Thus, for example, 3-m is a succinct way of saying 3 bytes when the m flag is 0, and 2 bytes when the m flag is 1.

In the CYCLES column:

- e = e flag
- m = m flag
- p = 1 if a page boundary is crossed, 0 otherwise
- t = 1 if branch taken, 0 otherwise
- w = 0 if the DL register is $00, 1 otherwise
- x = x flag

Again, for example, 3-m is a succinct way of saying 3 cycles when the m flag is 0, and 2 cycles when the m flag is 1.

In the "nvmxdizc e" column:

- 0 = flag is cleared
- 1 = flag is set
- * = flag is affected
- m = flag is affected by the (16-8*m)-bit result
- x = flag is affected by the (16-8*x)-bit result

For example, the m in the z column of ADC means that the z flag is affected by the 16-bit result when the m flag is 0, and the 8-bit result when the m flag is 1.

In general, in emulation mode (and for 8-bit results in native mode), the 65C816 has the same behavior as 65C02 but the same cycle counts as the NMOS 6502. For example, when the d flag is 1 and the m flag is 1, ADC #$00 will have valid n, z, and c flag results (like the 65C02, but unlike the NMOS 6502), but will take 2 cycles (like the NMOS 6502, but unlike the 65C02).

Note that the cycle counts may look a little different than what you may be used to with the 65C02 or NMOS 6502, even when the cycle count is the same for 8-bit results. For example, LDA absolute is 4 cycles on the 65C02 and NMOS 6502, but the formula here is 5-m cycles, which is 4 cycles for the 8-bit case (i.e. when the m flag is 1).

6.1.1.1 ADC SBC

```
ADd with Carry
SuBtract with Carry

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
```

```
61 2   7-m+w       (dir,X)   mm....mm . ADC ($10,X)
63 2   5-m         stk,S     mm....mm . ADC $32,S
65 2   4-m+w       dir       mm....mm . ADC $10
67 2   7-m+w       [dir]     mm....mm . ADC [$10]
69 3-m 3-m         imm       mm....mm . ADC #$54
6D 3   5-m         abs       mm....mm . ADC $9876
6F 4   6-m         long      mm....mm . ADC $FEDBCA
71 2   7-m+w-x+x*p (dir),Y   mm....mm . ADC ($10),Y
72 2   6-m+w       (dir)     mm....mm . ADC ($10)
73 2   8-m         (stk,S),Y mm....mm . ADC ($32,S),Y
75 2   5-m+w       dir,X     mm....mm . ADC $10,X
77 2   7-m+w       [dir],Y   mm....mm . ADC [$10],Y
79 3   6-m-x+x*p   abs,Y     mm....mm . ADC $9876,Y
7D 3   6-m-x+x*p   abs,X     mm....mm . ADC $9876,X
7F 4   6-m         long,X    mm....mm . ADC $FEDCBA,X
E1 2   7-m+w       (dir,X)   mm....mm . SBC ($10,X)
E3 2   5-m         stk,S     mm....mm . SBC $32,S
E5 2   4-m+w       dir       mm....mm . SBC $10
E7 2   7-m+w       [dir]     mm....mm . SBC [$10]
E9 3-m 3-m         imm       mm....mm . SBC #$54
ED 3   5-m         abs       mm....mm . SBC $9876
EF 4   6-m         long      mm....mm . SBC $FEDBCA
F1 2   7-m+w-x+x*p (dir),Y   mm....mm . SBC ($10),Y
F2 2   6-m+w       (dir)     mm....mm . SBC ($10)
F3 2   8-m         (stk,S),Y mm....mm . SBC ($32,S),Y
F5 2   5-m+w       dir,X     mm....mm . SBC $10,X
F7 2   7-m+w       [dir],Y   mm....mm . SBC [$10],Y
F9 3   6-m-x+x*p   abs,Y     mm....mm . SBC $9876,Y
FD 3   6-m-x+x*p   abs,X     mm....mm . SBC $9876,X
FF 4   6-m         long,X    mm....mm . SBC $FEDCBA,X
```

ADC and SBC add to, and subtract from, the accumulator. When the m flag is 0, it is a 16-bit operation, and when the m flag is 1, it is an 8-bit operation. When the d flag is 0, binary arithmetic is used, and when the d flag is 1, BCD arithmetic is used. Note that like the NMOS 6502, but unlike the 65C02, decimal mode (i.e. when the d flag is 1) takes no additional cycles.

The formula for ADC is:

- accumulator = accumulator + data + carry

The formula for SBC can be written several ways; one way is:

- accumulator = accumulator - data - 1 + carry

In other words, the formula is accumulator = accumulator - data - 1 when the carry (i.e. the c flag) is 0, and accumulator = accumulator - data when the carry is 1.

When the d flag is 0:

- The n flag is 0 when the high bit of the result (bit 15 when the m flag is 0, bit 7 when the m flag is 1) is 0, and the n flag is 1 when the high bit of the result is 1.
- The v flag is 0 when there is not a signed arithmetic overflow, and the v flag is 1 when there is a signed arithmetic overflow. For 8-bit signed numbers, $00 to $7F represents 0 to 127, and $80 to $FF represents -128 to -1; an 8-bit arithmetic overflow occurs when the result is outside the range -128 to 127. For 16-bit signed numbers, $0000 to $7FFF represents 0 to 32767, and $8000 to $FFFF represents -32768 to -1; a 16-bit arithmetic overflow occurs when the result is outside the range -32768 to 32767.
- The z flag is 0 when the 16-bit (when m flag is 0) or 8-bit (when the m flag is 1) result is nonzero, and the z flag is 1 when the result is zero.
- The c flag is 0 when there is not an unsigned carry, and the c flag is 1 when there is an unsigned carry. For 8-bit unsigned numbers, $00 to $FF represents 0 to 255; for addition, an 8-bit carry occurs when the result is greater than 255. For 16-bit unsigned numbers, $0000 to $FFFF represents 0 to 65535; for addition, an 16-bit carry occurs when the result is greater than 65535. For subtraction (8-bit or 16-bit), there is a carry when the accumulator is greater than or equal to the data.

When the d flag is 1, the n, z, and c flags have the same meaning (i.e. the n flag reflects the high bit of the result, the z flag indicates when the result is zero, and the carry indicates when the result is outside the range 0 to 9999). The v flag is overwritten, but BCD is really an unsigned representation, so the v flag can be considered invalid, since it does not represent a signed arithmetic overflow.

Example 1: If the accumulator is $0001, the m flag is 0, the d flag is 0, and the c flag is 1, then after SBC #$2003

- the accumulator will be $DFFE
- the n flag will be 1
- the v flag will be 0
- the z flag will be 0
- the c flag will be 0

Example 2: If the accumulator is $0001, the m flag is 0, the d flag is 1, and the c flag is 1, then after SBC #$2003

- the accumulator will be $7998
- the n flag will be 0
- the z flag will be 0
- the c flag will be 0

6.1.1.2 CMP CPX CPY

```
CoMPare (to accumulator)
ComPare to X register
ComPare to Y register

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
C1 2   7-m+w       (dir,X)   m.....mm . CMP ($10,X)
C3 2   5-m         stk,S     m.....mm . CMP $32,S
C5 2   4-m+w       dir       m.....mm . CMP $10
C7 2   7-m+w       [dir]     m.....mm . CMP [$10]
C9 3-m 3-m         imm       m.....mm . CMP #$54
CD 3   5-m         abs       m.....mm . CMP $9876
CF 4   6-m         long      m.....mm . CMP $FEDBCA
D1 2   7-m+w-x+x*p (dir),Y   m.....mm . CMP ($10),Y
D2 2   6-m+w       (dir)     m.....mm . CMP ($10)
D3 2   8-m         (stk,S),Y m.....mm . CMP ($32,S),Y
D5 2   5-m+w       dir,X     m.....mm . CMP $10,X
D7 2   7-m+w       [dir],Y   m.....mm . CMP [$10],Y
D9 3   6-m-x+x*p   abs,Y     m.....mm . CMP $9876,Y
DD 3   6-m-x+x*p   abs,X     m.....mm . CMP $9876,X
DF 4   6-m         long,X    m.....mm . CMP $FEDCBA,X
E0 3-x 3-x         imm       x.....xx . CPX #$54
E4 2   4-x+w       dir       x.....xx . CPX $10
EC 3   5-x         abs       x.....xx . CPX $9876
C0 3-x 3-x         imm       x.....xx . CPY #$54
C4 2   4-x+w       dir       x.....xx . CPY $10
CC 3   5-x         abs       x.....xx . CPY $9876
```

CMP, CPX, and CPY compare the accumulator, X register, and Y register (respectively) to the data. CMP is a 16-bit comparison when the m flag is 0, and an 8-bit comparison when the m flag is 1. CPX and CPY are 16-bit comparisons when the x flag is 0, and 8-bit comparisons when the x flag is 1.

CMP is similar to SBC, except:

- A. It is always a binary subtraction (i.e. SBC as though the d flag was 0)
- B. It does not include the carry in the formula (i.e. register - data; in other words, SBC as though the carry was set before the SBC)

- C. The v flag is not affected
- D. The result is only reflected in the n, z, and c flags; the accumulator, X register, and Y register are left unchanged.

Likewise for CPX and CPY, except the X and Y registers, respectively, are used instead of the accumulator.

- The n flag reflects the high bit of the result.
- The z flag reflects whether the result is zero.
- The c flag is 0 when the register (accumulator, X register, or Y register) is less than the data, and the c flag is 1 when the register is greater than or equal to the data.

Example: If the accumulator is $1234 and the m flag is 0, then after CMP #$1234

- The n flag will be 0
- The z flag will be 1
- The c flag will be 1

6.1.1.3 DEC DEX DEY INC INX INY

```
DECrement
DEcrement X register
DEcrement Y register
INCrement
INcrement X register
INcrement Y register
```

```
OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
3A 1   2           acc       m.....m. . DEC
C6 2   7-2*m+w     dir       m.....m. . DEC $10
CE 3   8-2*m       abs       m.....m. . DEC $9876
D6 2   8-2*m+w     dir,X     m.....m. . DEC $10,X
DE 3   9-2*m       abs,X     m.....m. . DEC $9876,X
CA 1   2           imp       x.....x. . DEX
88 1   2           imp       x.....x. . DEY
1A 1   2           acc       m.....m. . INC
E6 2   7-2*m+w     dir       m.....m. . INC $10
EE 3   8-2*m       abs       m.....m. . INC $9876
F6 2   8-2*m+w     dir,X     m.....m. . INC $10,X
FE 3   9-2*m       abs,X     m.....m. . INC $9876,X
E8 1   2           imp       x.....x. . INX
C8 1   2           imp       x.....x. . INY
```

DEC, DEX, and DEY decrement (i.e. subtract 1 from) and INC, INX, and INY increment (i.e. add 1 to) a register or the data.

DEX, DEY decrement the X and Y registers respectively, and INX and INY increment the X and Y registers respectively. DEX, DEY, INX, and INY are 16-bit operations when the x flag is 0 and 8-bit operations when the x flag is 1.

For accumulator addressing, DEC and INC decrement and increment the accumulator; for all other addressing modes, DEC and INC decrement and increment the data (at the memory location specified by the addressing mode) and the accumulator is not affected. DEC and INC are 16-bit operations when the m flag is 0 and 8-bit operations when the m flag is 1.

Assemblers may require or permit the following syntaxes for accumulator addressing for DEC and INC

```
DEA
DEC
DECA
DEC A
```

```
INA
INC
INCA
INC A
```

For this reason, in some assemblers, the label A is reserved and cannot be used in source code.

For all 6 instructions:

- The n flag reflects the high bit of the result.
- The z flag reflects whether the result is zero.

Example: If the X register is $7FFF and the x flag is 0, then after INX

- the X register will be $8000
- the n flag will be 1
- the z flag will be 0

6.1.2.1 AND EOR ORA

```
bitwise AND
bitwise Exclusive OR
bitwise OR Accumulator

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
21 2   7-m+w       (dir,X)   m.....m. . AND ($10,X)
23 2   5-m         stk,S     m.....m. . AND $32,S
25 2   4-m+w       dir       m.....m. . AND $10
27 2   7-m+w       [dir]     m.....m. . AND [$10]
29 3-m 3-m         imm       m.....m. . AND #$54
2D 3   5-m         abs       m.....m. . AND $9876
2F 4   6-m         long      m.....m. . AND $FEDBCA
31 2   7-m+w-x+x*p (dir),Y   m.....m. . AND ($10),Y
32 2   6-m+w       (dir)     m.....m. . AND ($10)
33 2   8-m         (stk,S),Y m.....m. . AND ($32,S),Y
35 2   5-m+w       dir,X     m.....m. . AND $10,X
37 2   7-m+w       [dir],Y   m.....m. . AND [$10],Y
39 3   6-m-x+x*p   abs,Y     m.....m. . AND $9876,Y
3D 3   6-m-x+x*p   abs,X     m.....m. . AND $9876,X
3F 4   6-m         long,X    m.....m. . AND $FEDCBA,X
41 2   7-m+w       (dir,X)   m.....m. . EOR ($10,X)
43 2   5-m         stk,S     m.....m. . EOR $32,S
45 2   4-m+w       dir       m.....m. . EOR $10
47 2   7-m+w       [dir]     m.....m. . EOR [$10]
49 3-m 3-m         imm       m.....m. . EOR #$54
4D 3   5-m         abs       m.....m. . EOR $9876
4F 4   6-m         long      m.....m. . EOR $FEDBCA
51 2   7-m+w-x+x*p (dir),Y   m.....m. . EOR ($10),Y
52 2   6-m+w       (dir)     m.....m. . EOR ($10)
53 2   8-m         (stk,S),Y m.....m. . EOR ($32,S),Y
55 2   5-m+w       dir,X     m.....m. . EOR $10,X
57 2   7-m+w       [dir],Y   m.....m. . EOR [$10],Y
59 3   6-m-x+x*p   abs,Y     m.....m. . EOR $9876,Y
5D 3   6-m-x+x*p   abs,X     m.....m. . EOR $9876,X
5F 4   6-m         long,X    m.....m. . EOR $FEDCBA,X
01 2   7-m+w       (dir,X)   m.....m. . ORA ($10,X)
03 2   5-m         stk,S     m.....m. . ORA $32,S
05 2   4-m+w       dir       m.....m. . ORA $10
07 2   7-m+w       [dir]     m.....m. . ORA [$10]
09 3-m 3-m         imm       m.....m. . ORA #$54
0D 3   5-m         abs       m.....m. . ORA $9876
0F 4   6-m         long      m.....m. . ORA $FEDBCA
11 2   7-m+w-x+x*p (dir),Y   m.....m. . ORA ($10),Y
```

```
12 2   6-m+w       (dir)     m.....m. . ORA ($10)
13 2   8-m         (stk,S),Y m.....m. . ORA ($32,S),Y
15 2   5-m+w       dir,X     m.....m. . ORA $10,X
17 2   7-m+w       [dir],Y   m.....m. . ORA [$10],Y
19 3   6-m-x+x*p   abs,Y     m.....m. . ORA $9876,Y
1D 3   6-m-x+x*p   abs,X     m.....m. . ORA $9876,X
1F 4   6-m         long,X    m.....m. . ORA $FEDCBA,X
```

AND, EOR, and ORA bitwise And, bitwise Exclusive-Or, and bitwise (inclusive) Or the accumulator with the data, and store the result in the accumulator. These instructions are 16-bit operations when the m flag is 0, and 8-bit operations when the m flag is 1.

For AND, if bit 0 of the accumulator (input) and bit 0 of the data are both 1, then bit 0 of the accumulator (result) will be 1, otherwise, bit 0 of the accumulator result will be 0. And so on for bits 1, 2, 3, etc.

For EOR, if bit 0 of the accumulator (input) and bit 0 of the data have the same value (e.g. both are zero), then bit 0 of the accumulator (result) will be 0, otherwise, bit 0 of the accumulator result will be 1. And so on for bits 1, 2, 3, etc.

For ORA, if bit 0 of the accumulator (input) and bit 0 of the data are both 0, then bit 0 of the accumulator (result) will be 0, otherwise, bit 0 of the accumulator result will be 1. And so on for bits 1, 2, 3, etc.

For all 3 instructions:

- The n flag reflects the high bit of the result.
- The z flag reflects whether the result is zero.

Example: If the accumulator is $0F06 and the m flag is 0, then after EOR #$F103

- the accumulator will be $FE05
- the n flag will be 1
- the z flag will be 0

6.1.2.2 BIT

```
test BITs

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
24 2   4-m+w       dir       mm....m. . BIT $10
2C 3   5-m         abs       mm....m. . BIT $9876
34 2   5-m+w       dir,X     mm....m. . BIT $10,X
3C 3   6-m-x+x*p   abs,X     mm....m. . BIT $9876,X
89 3-m 3-m         imm       ......m. . BIT #$54
```

BIT tests the bits of the data with the bits of the accumulator. It is a 16-bit operation when the m flag is 0, and an 8-bit operation when the m flag is 1.

Just as CMP is similar to SBC without overwriting the accumulator, BIT performs the same function as AND, but BIT only uses the result to set flags and does not overwrite the accumulator.

Immediate addressing only affects the z flag (with the result of the bitwise And), but does not affect the n and v flags. All other addressing modes of BIT affect the n, v, and z flags. This is the only instruction in the 6502 family where the flags affected depends on the addressing mode.

- The n flag reflects the high bit of the data (note: just the data, not the bitwise And of the accumulator and the data).
- The v flag reflects the second highest bit of the data (i.e. bit 14 of the data when the m flag is 0, and bit 6 of the data when the m flag is 1, and again, just the data, not the bitwise And).
- The z flag reflects whether the result (of the bitwise And) is zero.

Example: If the accumulator is $43, the DBR is $12, the m flag is 1, and

- $12ABCD contains $9C

then after BIT $ABCD

- the n flag will be 1
- the v flag will be 0
- the z flag will be 1

6.1.2.3 TRB TSB

```
Test and Reset Bits
Test and Set Bits

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ---------- --------- ---------- ------
14 2   7-2*m+w     dir       ......m. . TRB $10
1C 3   8-2*m       abs       ......m. . TRB $9876
04 2   7-2*m+w     dir       ......m. . TSB $10
0C 3   8-2*m       abs       ......m. . TSB $9876
```

TRB and TSB test the bits of the data with the bits of the accumulator (using a bitwise And, like BIT), then reset (i.e. clear) or set (respectively) the bits of the data that are ones in the accumulator. The accumulator is unchanged. These are 16-bit operations when the m flag is 0, and 8-bit operations when the m flag is 1.

For example, if the accumulator is $43 and the m flag is 1, then TRB resets (i.e. clears) bits 0, 1, and 6 of the data, and does not affect the other bits (bits 2, 3, 4, 5, and 7). Under the same condition, TSB sets bits 0, 1, and 6 of the data and does not affect the other bits.

- The z flag reflects whether the result (of the bitwise And) is zero.

Example: If the accumulator is $43, the DBR is $12, the m flag is 1, and

- $12ABCD contains $9C

then after TSB $ABCD

- the z flag will be 1
- $12ABCD will contain $DF

6.1.3 ASL LSR ROL ROR

```
Arithmetic Shift Left
Logical Shift Right
ROtate Left
ROtate Right

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ---------- --------- ---------- ------
06 2   7-2*m+w     dir       m.....mm . ASL $10
0A 1   2           acc       m.....mm . ASL
0E 3   8-2*m       abs       m.....mm . ASL $9876
16 2   8-2*m+w     dir,X     m.....mm . ASL $10,X
1E 3   9-2*m       abs,X     m.....mm . ASL $9876,X
46 2   7-2*m+w     dir       0.....m* . LSR $10
4A 1   2           acc       0.....m* . LSR
4E 3   8-2*m       abs       0.....m* . LSR $9876
56 2   8-2*m+w     dir,X     0.....m* . LSR $10,X
```

```
5E 3   9-2*m       abs,X     0.....m* . LSR $9876,X
26 2   7-2*m+w     dir       m.....mm . ROL $10
2A 1   2           acc       m.....mm . ROL
2E 3   8-2*m       abs       m.....mm . ROL $9876
36 2   8-2*m+w     dir,X     m.....mm . ROL $10,X
3E 3   9-2*m       abs,X     m.....mm . ROL $9876,X
66 2   7-2*m+w     dir       m.....m* . ROR $10
6A 1   2           acc       m.....m* . ROR
6E 3   8-2*m       abs       m.....m* . ROR $9876
76 2   8-2*m+w     dir,X     m.....m* . ROR $10,X
7E 3   9-2*m       abs,X     m.....m* . ROR $9876,X
```

ASL, LSR, ROL, and ROR shift the accumulator or the data left (ASL and ROL) and right (LSR and ROR). They are 16-bit operations when the m flag is 0, and 8-bit operations when the m flag is 1.

For accumulator addressing, the accumulator is shifted; for all other addressing modes, the data (at the memory location specified by the addressing mode) is shifted and the accumulator is not affected.

ASL shifts left; a zero is shifted into the low bit (bit 0); the high bit (bit 15 when the m flag is one, bit 7 when the m flag is 0) is shifted into the c flag.

LSR shifts right; a zero is shifted into the high bit; the low bit is shifted into the c flag.

ROL shifts left; the (input) c flag is shifted into the low bit; the high bit is shifted into the c flag (result).

ROR shifts right; the (input) c flag is shifted into the high bit; the low bit is shifted into the c flag (result).

For all 4 instructions:

- The n flag reflects the high bit of the result.
- The z flag reflects whether the result is zero.

Assemblers may require or permit the following syntaxes for accumulator addressing

```
ASL
ASLA
ASL A
LSR
LSRA
LSR A
ROL
ROLA
ROL A
ROR
RORA
ROR A
```

Again, for this reason, in some assemblers, the label A is reserved and cannot be used in source code.

Note that for absolute,X addressing, when the m flag is 1, the cycle count matches the NMOS 6502 timing (7 cycles always) rather than the 65C02 (6 cycles if a page boundary was not crossed).

Example: If the DBR is $12, the m flag is 1, and

- $12ABCD contains $8F

then after ASL $ABCD

- $12ABCD will contain $1E
- the n flag will be 0
- the z flag will be 0
- the c flag will be 1

6.2.1.1 BCC BCS BEQ BMI BNE BPL BRA BVC BVS

```
Branch if Carry Clear
Branch if Carry Set
Branch if EQual
Branch if MInus
Branch if Not Equal
Branch if PLus
BRanch Always
Branch if oVerflow Clear
Branch if oVerflow Set
```

```
OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
90 2   2+t+t*e*p   rel8      ........ . BCC LABEL
B0 2   2+t+t*e*p   rel8      ........ . BCS LABEL
F0 2   2+t+t*e*p   rel8      ........ . BEQ LABEL
30 2   2+t+t*e*p   rel8      ........ . BMI LABEL
D0 2   2+t+t*e*p   rel8      ........ . BNE LABEL
10 2   2+t+t*e*p   rel8      ........ . BPL LABEL
80 2   3+e*p       rel8      ........ . BRA LABEL
50 2   2+t+t*e*p   rel8      ........ . BVC LABEL
70 2   2+t+t*e*p   rel8      ........ . BVS LABEL
```

BRA unconditionally branches; the other 8 instructions branch based on the value of the n, v, z, or c flag. These instructions can branch -128 to 127 bytes from the address of the next instruction (i.e. the address of the instruction after the branch instruction); since all of these branch instructions are 2 byte instructions, that means the branch can be -126 to 129 bytes from the address of the branch instruction.

Note, however, a page boundary is crossed when the branch destination is on a different page than the next instruction (again, the instruction after the branch instruction). This means that

```
LABEL BRA LABEL+2 ; 3 cycles
```

always takes 3 cycles, no matter where the BRA instruction is located in memory, since the branch destination is the next instruction, i.e. they are the same address, and thus on the same page.

```
BCC branches if the c flag is 0
BCS branches if the c flag is 1
BEQ branches if the z flag is 1
BMI branches if the n flag is 1
BNE branches if the z flag is 0
BPL branches if the n flag is 0
BVC branches if the v flag is 0
BVS branches if the v flag is 1
```

The names BEQ and BNE come from the fact that these two instructions often follow a compare (CMP, CPX, or CPY) and thus will branch if the register and the data were equal (BEQ) or not equal (BNE).

In some assemblers, BGE (Branch if Greater than or Equal) and BLT (Branch if Less Than) are synonyms for BCS and BCC, respectively. The former names come from the fact the CMP, CPX, and CPY clear the c flag if the register was less than the data, and set the c flag if the register was greater than or equal to the data.

Note that BRA actually has the same timing as the other instructions; the cycle count formula is different because there is no branch not taken case.

Example: If the z flag is 0, then a BNE $C023 at $ABC000 will branch to $ABC023. (The displacement is $21 since the instruction after the BNE is at $ABC002.)

6.2.1.2 BRL

```
BRanch Long

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
82 3   4           rel16     ........ . BRL LABEL
```

BRL is like BRA, except that the branch displacement is 16 bits, instead of 8 bits, which means it can branch -32768 to 32767 bytes from the address of the next instruction (or -32765 to 32770 bytes from the address of the BRL instruction). However, since BRL wraps at bank boundaries, this means it can branch anywhere within the current (program) bank.

BRL is one cycle longer than JMP absolute (which also can jump anywhere within the current bank). The reason BRL is used is for code (e.g. a section of code, or even an entire program) that can run no matter what address it is located at (in other words, its address isn't known until run time).

Example: A BRL $C045 at $ABC000 will branch to $ABC045. (The displacement is $0042 since the instruction after the BRL is at $ABC003.)

6.2.2.1 JMP JSL JSR

```
JuMP
Jump to Subroutine Long
Jump to SubRoutine

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
4C 3   3           abs       ........ . JMP $1234
5C 4   4           long      ........ . JMP $FEDCBA
6C 3   5           (abs)     ........ . JMP ($1234)
7C 3   6           (abs,X)   ........ . JMP ($1234,X)
DC 3   6           [abs]     ........ . JMP [$1234]
22 4   8           long      ........ . JSL $123456
20 3   6           abs       ........ . JSR $1234
FC 3   8           (abs,X)   ........ . JSR ($1234,X)
```

JMP jumps to the address specified by the addressing mode. absolute, (absolute) and (absolute,X) addressing jump to an address within the current program bank; [absolute] and long addressing use a 24-bit address and can jump to any bank.

Note that JMP (absolute) is 5 cycles, same as the NMOS 6502, but different from the 65C02 (6 cycles).

Again, in some assemblers, for [absolute] and/or long addressing, JML is a synonym for JMP. Also, some assemblers allow the syntax JML (absolute) for opcode $DC (i.e. JMP [absolute]), which is a 24-bit pointer, rather than a 16-bit pointer. Use of [absolute] is more consistent with the syntax (direct) and [direct], where parentheses indicate a 16-bit pointer, and square brackets indicate a 24-bit pointer.

JSL pushes the K register (i.e. program bank register), then pushes the 16-bit address (high byte first, then low byte) of the JSL instruction plus 3 (one less than the address of the next instruction), then jumps to the address specified by the operand. Thus, if the JSL instruction (i.e. the $22 opcode) is at $12FFFD, then the bytes pushed are (in order): $12, $00, and $00, rather than $13, $00, and $00.

JSR pushes the 16-bit address (i.e. the program counter) of the JSR instruction plus 2 onto the stack, and jumps to an address within the current program bank. In other words, the address pushed is one less than the address of the next instruction. The high byte is pushed first, then the low byte is pushed.

Some assemblers allow JSR to be a synonym for JSL when a 24-bit address is used with JSR. In general, you would be well advised not to utilize this; i.e. not use JSR in place of JSL. A JSR pushes two bytes and is matched with an RTS; a JSL pushes three bytes and is matched with an RTL. A JSR LABEL that sometimes pushes two bytes (i.e. for absolute addressing) and sometimes pushes three bytes (i.e. for long addressing) can result in code that is quite difficult to follow and/or debug.

Example: If the S register is $01FF, then a JSR $ABCD at $123456

- stores $34 at $0001FF
- stores $58 at $0001FE

then jumps to $12ABCD, and

- the S register will be $01FD

6.2.2.2 RTL RTS

```
ReTurn from subroutine Long
ReTurn from Subroutine

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
6B 1   6           imp       ........ . RTL
60 1   6           imp       ........ . RTS
```

RTL returns from a subroutine called with JSL (from any bank). It pulls the low byte, then the high byte of the program counter from the stack, then increments the program counter, then pulls the K register (i.e. the program bank register) from the stack. This means that if $FF, $FF, and $12 are pulled from the stack, the instruction at $120000 (rather than $130000) will be executed next.

RTS returns from a subroutine called with JSR (within the current program bank). It pulls the low byte, then the high byte of the program counter from the stack, then increments the program counter.

Example: if the K register is $12, the S register is $01FD, and

- $0001FE contains $56
- $0001FF contains $34

then an RTS will return to $123457, and

- the S register will be $01FF

6.3.1 BRK COP

```
BReaKpoint
COProcessor

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
00 1   8-e         imp       ....01.. . BRK
02 2   8-e         imm       ....01.. . COP #$12

BRK and COP are software interrupts.
```

In native mode, BRK and COP push the K register (i.e. program bank register), then push the 16-bit address (again high byte first, then low byte) of the BRK or COP instruction plus 2, then push the P register, then jump to the appropriate (16-bit) native mode interrupt vector. The native mode BRK vector is at $00FFE6 and the native mode COP vector is at $00FFE4.

In emulation mode, BRK and COP push the 16-bit address (again high byte first, then low byte) of the BRK or COP instruction plus 2, then push the P register, then jump to the appropriate (16-bit) emulation mode interrupt vector. The emulation mode BRK vector is at $00FFFE and the emulation mode COP vector is at $00FFF4. When BRK pushes the P register, the b flag (i.e. bit 5) will be set; because, in emulation mode, as on the NMOS 6502 and 65C02, BRK and IRQ share an interrupt vector, this allows the BRK/IRQ handler to distinguish a BRK from an IRQ. COP in emulation mode may seem somewhat paradoxical, since it was not available on the NMOS 6502 or 65C02, but COP can be used in emulation mode, and when pushing onto the stack it will wrap at the page 1 boundary (in other words, it is treated as an "old" instruction, rather than a "new" instruction).

- The i flag is set after pushing the P register; furthermore, like the 65C02 (but unlike the NMOS 6502), the d flag is cleared after pushing the P register, thus ensuring that the d flag has a known value (i.e. zero) for BRK and COP handlers.

An important note: as on the NMOS 6502 and 65C02, the correct way for an emulation mode BRK/IRQ handler to distinguish a BRK from an IRQ is to use the stacked value of the b flag (i.e. bit 5 of the stacked value of the P register). Thus, this is correct (assuming that 2,S is within page 1; i.e. that S < $01FE, which is usually the case):

```
EMULATION_BRKIRQ_HANDLER
   PHA
   LDA 2,S
   BIT #$10
   BNE BRK
   BEQ IRQ
```

but this is incorrect (since it uses the current, rather than the stacked value of the P register):

```
EMULATION_BRKIRQ_HANDLER
   PHA
   PHP
   PLA
   BIT #$10
   BNE BRK
   BEQ IRQ
```

Also, there are several ramifications of BRK and COP to keep in mind. First, the fact that both BRK and COP push the address of the BRK or COP instruction plus 2 means that RTI will return to the address of the BRK or COP instruction plus 2; thus, both instructions have a signature byte (even though neither instruction makes any use of this signature byte) which is skipped. Since most NMOS 6502 and 65C02 assemblers will assemble a BRK instruction as a one byte instruction (i.e. the opcode $00) rather than a two byte instruction (i.e. the opcode $00 and the signature byte), most 65C816 assemblers will also assemble a BRK instruction as a one byte instruction rather than a two byte instruction; despite this, on all members of the 6502 family, the BRK instruction is really a two byte instruction, consisting of an opcode ($00) and a signature byte.

Second, the fact that all interrupt vectors are 16-bit addresses (rather than 24-bit addresses) means that the interrupt handlers must be located in bank 0 (or at least begin in bank 0, since it's possible to use JMP or JSL to execute code in another bank).

Third, because emulation mode pushes a 16-bit address, a BRK or COP must be executed from bank 0 in emulation mode; the interrupt handler has no way of determining which bank to return to, since the program bank is not pushed. Furthermore, the handler must return from bank 0 (i.e. the RTI instruction of the handler must be located in bank 0) since RTI (in emulation mode) will pull a 16-bit address, rather than a 24-bit address. (Actually, there is a way to use BRK and COP in emulation mode from a nonzero bank: if the BRK or COP is preceded by a PHK, then PLP RTL can be used to instead of RTI to return from the interrupt handler. However, because RTL increments the program counter after pulling it from the stack, this will return to the address of the BRK or COP instruction plus 3; in effect, this behaves as though BRK and COP had 2 signature bytes rather than 1 signature byte. Note that most BRK and COP handlers do not make use of this technique.)

Fourth, because RTI pulls the K register in native mode, but not in emulation mode (essentially, it pulls a 24-bit address in the native mode, but a 16-bit address in emulation mode), it is vitally important that a native mode handler returns in native mode and an emulation mode handler returns in emulation mode. Generally, the handler itself would not need to switch modes, but if it calls a subroutine that executes in the other mode, it is important to ensure that the 65C816 is switched back into the original mode before returning from the handler.

Example: If the S register is $01FF, the e flag is 0, the P register is $08, and

- $00FFE6 contains $AB
- $00FFE7 contains $CD

then a BRK at $123456

- stores $12 at $0001FF
- stores $34 at $0001FE
- stores $58 at $0001FD
- stores $08 at $0001FC

then jumps to $00CDAB, and

- the S register will be $01FB
- the d flag will be 0
- the i flag will be 1

6.3.1.1 HARDWARE INTERRUPTS

In addition to the two software interrupts (BRK and COP), the 65C816 has four hardware interrupts: ABORT, IRQ, NMI, and RESET. While the hardware interrupts are not instructions, they function in much the same way as the software interrupt instructions.

In native mode, the K register (i.e. program bank register) is pushed, then the 16-bit program counter is pushed (again high byte first, then low byte), then the P register is pushed, and finally, the 65C816 jumps to the appropriate (16-bit) native mode interrupt vector.

In emulation mode, the 16-bit program counter is pushed (again high byte first, then low byte), then the P register is pushed, and finally, the 65C816 jumps to the appropriate (16-bit) emulation mode interrupt vector. ABORT (like COP) in emulation mode may seem somewhat paradoxical, since there is no ABORTB pin on the NMOS 6502 or 65C02, but when an ABORT interrupt occurs in emulation mode, (again, like COP) when pushing onto the stack it will wrap at the page 1 boundary (in other words, it is treated as an "old" hardware interrupt, rather than a "new" one).

The interrupt vectors for native (e flag = 0) and emulation mode (e flag = 1) are:

```
e = 0  e = 1
------ ------
00FFE4 00FFF4 COP
00FFE6 00FFFE BRK
00FFE8 00FFF8 ABORT
00FFEA 00FFFA NMI
       00FFFC RESET
00FFEE 00FFFE IRQ
```

The address decoding circuitry can use the VPB signal to tell the 65C816 to fetch any or all of these interrupt vectors from any address desired. A RESET interrupt puts the 65C816 into emulation mode, thus there is no native mode RESET vector. Again, note that, as on the NMOS 6502 and 65C02, BRK and IRQ share an interrupt vector, and the correct way for the interrupt handler to distinguish them is to use the stacked (rather than the current) value of the b flag.

As with the software interrupts, the i flag is set after pushing the P register; furthermore, like the 65C02 (but unlike the NMOS 6502), the d flag is cleared after pushing the P register, thus ensuring that the d flag has a known value (i.e. zero) for interrupt handlers.

If an IRQ or NMI occurs in the middle of an instruction (e.g. after the first cycle of a BCC instruction), then the instruction is completed before pushing anything and jumping to the interrupt vector. For ABORT and

RESET, the instruction is not completed (in the former case, the instruction is aborted, and then restarted when the interrupt handler returns).

Also, the ramifications of the software interrupts (described in the previous section) also apply to hardware interrupts; namely, that (a) the interrupt handler must be located in bank 0 (because the interrupt vectors are 16-bit addresses), (b) in emulation mode, if an interrupt were to occur while executing code in a nonzero bank, (because the program bank register is not pushed) an RTI instruction has no way of knowing what bank to return to (this is far more important for hardware interrupts, since in general, there is no way know when they'll occur, and furthermore, an NMI cannot be masked by the i flag, thus the technique of preceding a BRK or COP by a PHK does not apply to hardware interrupts), and (c) an interrupt must be in the same mode (native or emulation) it started in when returning via an RTI instruction.

Example: If the S register is $01FF, the e flag is 0, the P register is $08, and

- $00FFE6 contains $AB
- $00FFE7 contains $CD

and an NMI occurs after the first cycle of a LDA #$1122 instruction at $345678, then the LDA instruction will be completed, then

- $34 will be stored at $0001FF
- $56 will be stored at $0001FE
- $7B will be stored at $0001FD
- $08 will be stored at $0001FC

then the 65C816 will jump to $00CDAB; afterwards

- the S register will be $01FB
- the d flag will be 0
- the i flag will be 1

6.3.2 RTI

```
ReTurn from Interrupt

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
40 1   7-e         imp       ******** . RTI
```

In native mode, the P register is pulled, then the 16-bit program counter is pulled (low byte first, then high byte), then the K register (i.e. program bank register) is pulled.

In emulation mode, the P register is pulled, then the 16-bit program counter is pulled (again low byte first, then high byte).

Note that unlike RTS (and RTL), the program counter is not incremented after it is pulled from the stack.

Again, as noted in the previous two sections, before returning from an interrupt via an RTI instruction, the mode (native or emulation) must be the same as it was when the interrupt occurred, since a different number of bytes are pulled from the stack.

Example: If the S register is $01FB, the e flag is 0, and

- $0001FC contains $08
- $0001FD contains $12
- $0001FE contains $34
- $0001FF contains $56

then an RTI will jump to $563412, and

- the S register will be $01FF
- the P register will be $08

6.4.1 CLC CLD CLI CLV SEC SED SEI

```
CLear Carry
CLear Decimal mode
CLear Interrupt disable
CLear oVerflow
SEt Carry
SEt Decimal mode
SEt Interrupt disable

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
18 1   2           imp       .......0 . CLC
D8 1   2           imp       ....0... . CLD
58 1   2           imp       .....0.. . CLI
B8 1   2           imp       .0...... . CLV
38 1   2           imp       .......1 . SEC
F8 1   2           imp       ....1... . SED
78 1   2           imp       .....1.. . SEI
```

CLC, CLD, CLI, and CLV clear the c, d, i, and v flags respectively.

SEC, SED, and SEI set the c, d, and i flags respectively.

Incidentally, you may wish to define SEI and CLI as macros called (e.g.) DI and EI (i.e. Disable Interrupt and Enable Interrupt); that way you don't have to remember whether i flag 0 means disable interrupts (it doesn't) or enable interrupts (it does).

Example: after a CLC

- the c flag is 0

6.4.2 REP SEP

```
REset Processor status bits
SEt Processor status bits

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
C2 2   3           imm       ******** . REP #$12
E2 2   3           imm       ******** . SEP #$12
```

REP and SEP reset (i.e. clear) and set (respectively) the bits of the P register that are ones in the operand; bits of the P register that are zero in the operand are not affected. (This is analogous to how TRB and TSB reset and set the bits of the memory location that are ones in the accumulator, and do not affect the bits in the memory location that are zero in the accumulator.)

Note that when the e flag is 1, the m and x flag are forced to 1, so after the REP or SEP, both flags will still be 1 no matter what the operand is.

Example: If the e flag is 0, then after SEP #$21

- the m and c flags will be 1
- the n, v, x, d, i, and z flags will not be affected

6.5 LDA LDX LDY STA STX STY STZ

```
LoaD Accumulator
LoaD X register
LoaD Y register
STore Accumulator
STore X register
STore Y register
STore Zero

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
A1 2   7-m+w       (dir,X)   m.....m. . LDA ($10,X)
A3 2   5-m         stk,S     m.....m. . LDA $32,S
A5 2   4-m+w       dir       m.....m. . LDA $10
A7 2   7-m+w       [dir]     m.....m. . LDA [$10]
A9 3-m 3-m         imm       m.....m. . LDA #$54
AD 3   5-m         abs       m.....m. . LDA $9876
AF 4   6-m         long      m.....m. . LDA $FEDBCA
B1 2   7-m+w-x+x*p (dir),Y   m.....m. . LDA ($10),Y
B2 2   6-m+w       (dir)     m.....m. . LDA ($10)
B3 2   8-m         (stk,S),Y m.....m. . LDA ($32,S),Y
B5 2   5-m+w       dir,X     m.....m. . LDA $10,X
B7 2   7-m+w       [dir],Y   m.....m. . LDA [$10],Y
B9 3   6-m-x+x*p   abs,Y     m.....m. . LDA $9876,Y
BD 3   6-m-x+x*p   abs,X     m.....m. . LDA $9876,X
BF 4   6-m         long,X    m.....m. . LDA $FEDCBA,X
A2 3-x 3-x         imm       x.....x. . LDX #$54
A6 2   4-x+w       dir       x.....x. . LDX $10
AE 3   5-x         abs       x.....x. . LDX $9876
B6 2   5-x+w       dir,Y     x.....x. . LDX $10,Y
BE 3   6-2*x+x*p   abs,Y     x.....x. . LDX $9876,Y
A0 3-x 3-x         imm       x.....x. . LDY #$54
A4 2   4-x+w       dir       x.....x. . LDY $10
AC 3   5-x         abs       x.....x. . LDY $9876
B4 2   5-x+w       dir,X     x.....x. . LDY $10,X
BC 3   6-2*x+x*p   abs,X     x.....x. . LDY $9876,X
81 2   7-m+w       (dir,X)   ........ . STA ($10,X)
83 2   5-m         stk,S     ........ . STA $32,S
85 2   4-m+w       dir       ........ . STA $10
87 2   7-m+w       [dir]     ........ . STA [$10]
8D 3   5-m         abs       ........ . STA $9876
8F 4   6-m         long      ........ . STA $FEDBCA
91 2   7-m+w       (dir),Y   ........ . STA ($10),Y
92 2   6-m+w       (dir)     ........ . STA ($10)
93 2   8-m         (stk,S),Y ........ . STA ($32,S),Y
95 2   5-m+w       dir,X     ........ . STA $10,X
97 2   7-m+w       [dir],Y   ........ . STA [$10],Y
99 3   6-m         abs,Y     ........ . STA $9876,Y
9D 3   6-m         abs,X     ........ . STA $9876,X
9F 4   6-m         long,X    ........ . STA $FEDCBA,X
86 2   4-x+w       dir       ........ . STX $10
8E 3   5-x         abs       ........ . STX $9876
96 2   5-x+w       dir,Y     ........ . STX $10,Y
84 2   4-x+w       dir       ........ . STY $10
8C 3   5-x         abs       ........ . STY $9876
94 2   5-x+w       dir,X     ........ . STY $10,X
64 2   4-m+w       dir       ........ . STZ $10
74 2   5-m+w       dir,X     ........ . STZ $10,X
9C 3   5-m         abs       ........ . STZ $9876
9E 3   6-m         abs,X     ........ . STZ $9876,X
```

LDA, LDX, and LDY load data into the accumulator, X register, and Y register, respectively. STA, STX, STY, STZ store the accumulator, the X register, the Y register, and zero into the memory location specified by the operand. Note that no registers are affected by STZ.

When the m flag is 0, LDA, STA, and STZ are 16-bit operations, and when the m flag is 1, LDA, STA, and STZ are 8-bit operations. When the x flag is 0, LDX, LDY, STX, and STY are 16-bit operations, and when the x flag is 1, LDX, LDY, STX, and STY are 8-bit operations.

For LDA, LDX, LDY, the n flag is 0 when the high bit of the result is 0, and 1 when the high bit of the result is 1; the z flag is 1 when the result is zero, and 0 when the result is nonzero.

Example: If the m flag is 0, and

- $123456 contains $AB
- $123457 contains $CD

then after a LDA $123456

- the accumulator will be $CDAB
- the n flag will be 1
- the z flag will be 0

6.6 MVN MVP

```
MoVe memory Negative
MoVe memory Positive

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
54 3   7           src,dest  ........ . MVN #$12,#$34
44 3   7           src,dest  ........ . MVP #$12,#$34
```

MVN and MVP move blocks of memory downward (i.e. from a higher (source) address to a lower (destination) address) and upward (i.e. from a lower (source) address to a higher (destination) address). The (16-bit) accumulator contains the number of bytes to move minus 1, the X register contains the 16-bit source address, and the Y register contains the 16-bit destination address. The source bank and the destination bank are specified in the operand. Both the source and the destination address wrap at the bank boundary. For MVN, X and Y are the start (i.e. lowest address) of the block, but for MVP, X and Y are the end (i.e. highest address) of the block.

MVN and MVP decrement the (16-bit) accumulator and increment (for MVN) or decrement (for MVP) both X and Y each time a byte is moved; this means that the accumulator will be $FFFF after an MVN or MVP (since it contained the number of bytes to move minus 1) and the X and Y registers will be larger (for MVN) or smaller (for MVP) by the number of bytes moved.

Also, the DBR is overwritten by MVN and MVP; after an MVN or MVP, the destination bank is stored in the DBR. Thus, it may be necessary in some instances to put a PHB instruction before the MVN or MVP, and a PLB instruction after the MVN or MVP.

Each byte moved takes 7 cycles. MVN and MVP can be interrupted by IRQ and NMI before the move is complete (unlike every other instruction, which must finish before an IRQ or NMI is serviced); however, they can only be interrupted every seventh cycle, i.e. they cannot be interrupted in the middle of moving a byte, but it can be interrupted between moving one byte and moving the next.

Another way to describe the MVN and MVP instruction is to say that they are 7 cycle instructions (sharp-eyed readers may have noticed that the cycle count in the table above is 7, rather than 7 times the number of bytes moved); after moving the byte whose (16-bit) source address is in the X register and whose (16-bit) destination address is in the Y register (and whose source and destination bank are specified by the operand), the destination bank is stored in the DBR, the (16-bit) accumulator is decremented, and both the X and Y registers are incremented (for MVN) or decremented (for MVP); then the program counter will be the address of the next instruction (i.e. the instruction after the MVN or MVP) if the accumulator is $FFFF, and the program counter will be the address of the the MVN or MVP if the accumulator is not $FFFF (i.e. the instruction jumps to itself if the accumulator is not $FFFF). Thus, the instruction is executed until A is $FFFF, both the X and Y register are incremented or decremented until the entire block is moved, the source and destination addresses wrap at bank boundaries, the instruction must complete all 7 cycles before an IRQ or NMI can only be serviced, and it will take 7 cycles per byte moved total.

An interesting question is what happens when MVN is used move upward and MVP is used to move downward. If the source and destination banks are different, or the source and destination blocks do not overlap, then the (final) result is no different than if the upward move had used MVP (and X and Y were updated to begin at the end of the block) or the downward move had used MVN (and, likewise, X and Y were updated to begin at the start of the block). However, if the source and destination banks are the same and the blocks do overlap, MVN and MVP have an interesting (and useful) property; they make copies of a pattern. To see how this works, if the accumulator is $0003, the X register is $1000, the Y register is $1002, and

- $001000 contains $AB
- $001001 contains $CD

then after an MVN #0,#0

- $001002 will contain $AB
- $001003 will contain $CD
- $001004 will contain $AB
- $001005 will contain $CD

This happens because

- the byte at $001000 ($AB) is stored at $001002
- the byte at $001001 ($CD) is stored at $001003
- the byte at $001002 (which now contains $AB) is stored at $001004
- the byte at $001003 (which now contains $CD) is stored at $001005

Incidentally, some assemblers allow an alternate syntax (in addition to the traditional syntax of a pair of 8-bit banks) where the operand is a pair of 24-bit addresses; thus, for such assemblers, MVN #$123456,#$789ABC is equivalent to MVN #$12,#$78.

Finally, note that while MVN and MVP can be used when the x flag is 1 (and this includes the case when the e flag is 1), neither instruction is particularly helpful in that situation, because the high bytes of both the X and Y are $00, so both the source are destination blocks are confined to the first page (i.e. addresses $0000 to $00FF) of the block.

Example: If the accumulator is $0001, the X register is $1000, the Y register is $2000, and

- $000FFF contains $AB
- $001000 contains $CD

then after an MVP #0,#0

- the accumulator will be $FFFF
- the X register will be $0FFE
- the Y register will be $1FFE
- the DBR will be $00
- $001FFF will contain $AB
- $002000 will contain $CD

6.7 NOP WDM

```
No OPeration
William D. Mensch, Jr.

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- --------- ------
EA 1   2           imp       ........ . NOP
42 2   2           imm       ........ . WDM
```

NOP, as its name implies, performs no operation (affecting no flags or registers); it is primarily useful for (a) delaying 2 cycles when a routine requires exact, or special, timing, (b) disabling code via patch, or (c) reserving space for self-modifying code.

WDM are the initials of William D. Mensch, Jr, the designer of the 65C816 (and the 65C02). This opcode is reserved for future expansion. On the 65C816, it is acts like a 2-byte, 2-cycle NOP (note that the actual NOP instruction is only 1 byte). The second byte is read, but ignored. This can be used to skip 1-byte instructions.

Example: when entering

```
LABEL1 INX
       DB  $42 ;WDM opcode
LABEL2 DEX
```

at LABEL1, the X register will be incremented and the DEX will not be executed, because the WDM opcode ($42) will treat it as immediate data and ignore it. (The X register will be decremented when entering at LABEL2.)

6.8.1 PEA PEI PER

```
Push Effective Address
Push Effective Indirect address
Push Effective Relative address

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- --------- ------
F4 3   5           imm       ........ . PEA #$1234
D4 2   6+w         dir       ........ . PEI $12
62 3   6           imm       ........ . PER LABEL
```

PEA, PEI, and PER all push a 16-bit value onto the stack. In fact, they could be thought of as the different addressing modes of the same instruction, since the operation is otherwise the same. This is analogous to there being different instruction names for JMP absolute and BRL, which are really the same operation, it's just that there is no assembler syntax to differentiate absolute from relative addressing, hence the different names.

Note, however, that the names of these instructions are a bit misleading. The 16-bit value that is pushed onto the stack need not be an address; it could be a constant. Because of the names, the operand syntax permitted (or required) can vary from assembler to assembler.

PEA #$1234 is the syntax that is most consistent with the operation of the instruction, since it simply pushes the value $1234, but does not access memory location $1234 (in any bank). This is analogous to LDA #$1234 vs. LDA $1234; the former does not access memory location $1234 (in any bank), but the latter does (in some bank). Nonetheless, some assemblers permit (or require) the syntax PEA $1234.

Likewise, PEI $12 is the syntax that is most consistent with the operation of the instruction. It pushes the same 16-bit value that (assuming the m flag is 0) LDA $12 loads into the accumulator, rather that the value that LDA ($12) loads into the accumulator. Nonetheless, some assemblers permit (or require) the syntax PEI ($12). Note, however, that PEI always pushes a 16-bit value no matter what the value of the m flag (or, for that matter the x flag) is.

Most assemblers use the syntax PER LABEL, rather than PER #LABEL; in fact, assemblers that even permit the latter syntax (much less require it) are rare. Even though PER $1234 itself does not access memory location $1234, subsequent instructions in any program that uses PER are likely to use the value pushed to access that memory location shortly thereafter. For example

```
       PER LABEL2-1
       BRL LABEL1
LABEL2
```

is like a JSR LABEL1 but with relative (instead of absolute) addressing. PER doesn't access address LABEL2, but address LABEL2 gets accessed when the subroutine returns and the opcode at LABEL2 is fetched. A (somewhat) comparable situation is JMP $1234; the JMP itself does not access $1234, but $1234 is accessed when the opcode at $1234 (i.e. the next instruction) is fetched.

Example: If the S register is $01FF, then after PEA #$1234,

- $0001FF will contain $12
- $0001FE will contain $34
- the S register will be $01FD

6.8.2 PHA PHX PHY PLA PLX PLY

```
PusH Accumulator
PusH X register
PusH Y register
PulL Accumulator
PulL X register
PulL Y register
```

```
OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
48 1   4-m         imp       ........ . PHA
DA 1   4-x         imp       ........ . PHX
5A 1   4-x         imp       ........ . PHY
68 1   5-m         imp       m.....m. . PLA
FA 1   5-x         imp       x.....x. . PLX
7A 1   5-x         imp       x.....x. . PLY
```

PHA, PHX, and PHY push the accumulator, X register, and Y register, respectively, onto the stack. PLA, PLX, and PLY pull the accumulator, X register, and Y register, respectively, from the stack.

When the m flag is 0, PHA and PLA push and pull a 16-bit value, and when the m flag is 1, PHA and PLA push and pull an 8-bit value.

When the x flag is 0, PHX, PHY, PLX, and PLY push and pull a 16-bit value, and when the x flag is 1, PHX, PHY, PLX, and PLY push and pull an 8-bit value.

For PLA, PLX, and PLY, the n flag is 1 if the high bit of the result (pulled into the register) is 1, and the n flag is 0 when the high bit of the result is 0; the z flag is 1 if the result is zero, and the z flag is 0 if the result is nonzero.

Example: If the S register is $01FD, the m flag is 0, and

- $0001FE contains $AB
- $0001FF contains $CD

then after a PLA

- the accumulator will be $CDAB
- the n flag will be 1

- the z flag will be 0
- the S register will be $01FF

6.8.3 PHB PHD PHK PHP PLB PLD PLP

```
PusH data Bank register
PusH Direct register
PusH K register
PusH Processor status register
PulL data Bank register
PulL Direct register
PulL Processor status register
```

```
OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
8B 1   3           imp       ........ . PHB
0B 1   4           imp       ........ . PHD
4B 1   3           imp       ........ . PHK
08 1   3           imp       ........ . PHP
AB 1   4           imp       *.....*. . PLB
2B 1   5           imp       *.....*. . PLD
28 1   4           imp       ******** . PLP
```

PHB, PHD, PHK, and PHP push the DBR, the D register, the K register, and the P register, respectively, onto the stack. PLB, PLD, and PLP pull the DBR, the D register, and the P register, respectively, from the stack.

PHP, PHK, PHP, PLB, and PLP push and pull one byte from the stack; PHD and PLD push and pull two bytes from the stack.

For PLB and PLD, the n flag is 1 when the high bit of the result (i.e. the value pulled into the register) is 1, and the n flag is 0 when high bit of the result is 0; the z flag is 1 when the result is zero, and the z flag is 0 when the result is nonzero.

For PLP, (all of) the flags are pulled from the stack. Note that when the e flag is 1, the m and x flag are forced to 1, so after the PLP, both flags will still be 1 no matter what value is pulled from the stack.

Incidentally, note that the only way to change the value of the DBR without moving memory (with MVN or MVP) is the PLB instruction; there are no load or transfer instructions that change the DBR.

Example: If the S register is $01FD and

- $0001FE contains $AB
- $0001FF contains $CD

then after a PLD

- the D register will be $CDAB
- the n flag will be 1
- the z flag will be 0
- the S register will be $01FF

6.9 STP WAI

```
SToP the clock
WAit for Interrupt
```

```
OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
```

```
DB 1   3           imp       ........ . STP
CB 1   3           imp       ........ . WAI
```

STP stops the clock input of the 65C816, effectively shutting down the 65C816 until a hardware reset (interrupt) occurs. This puts the 65C816 into a low power state. This is useful for applications (circuits) that require low power consumption, but STP is rarely seen otherwise.

WAI puts the 65C816 into a low power sleep state until a hardware interrupt occurs. In addition to reducing power consumption, using WAI also ensures that the interrupt will be recognized immediately. In other words, if an interrupt (e.g. an NMI) occurs in the middle of an instruction (e.g. ADC), the instruction must finish before the interrupt will be recognized (i.e. before jumping to the interrupt vector). When WAI is used, once its third cycle is complete, the 65C816 will wait for the interrupt and can respond to it without any additional delay whenever it occurs.

When the i flag is 1, interrupts are disabled, and normally an IRQ would be ignored. However, WAI when the i flag is 1 is a special case; specifically, when an IRQ occurs (after the WAI instruction), the 65C816 will continue with the next instruction rather than jumping to the interrupt vector. This means an IRQ can be responded to within one cycle. The interrupt handler is effectively inline code, rather than a separate routine, and thus it does not end with an RTI, resulting in fewer cycles needed to handle the interrupt.

6.10.1 TAX TAY TSX TXA TXS TXY TYA TYX

```
Transfer Accumulator to X register
Transfer Accumulator to Y register
Transfer Stack pointer to X register
Transfer X register to Accumulator
Transfer X register to Stack pointer
Transfer X register to Y register
Transfer Y register to Accumulator
Transfer Y register to X register

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
AA 1   2           imp       x.....x. . TAX
A8 1   2           imp       x.....x. . TAY
BA 1   2           imp       x.....x. . TSX
8A 1   2           imp       m.....m. . TXA
9A 1   2           imp       ........ . TXS
9B 1   2           imp       x.....x. . TXY
98 1   2           imp       m.....m. . TYA
BB 1   2           imp       x.....x. . TYX
```

TAX, TAY, TSX, TXA, TXS, TXY, TYA, and TYX transfer (i.e. copy) the contents from one register to another. The second letter of the instruction is the register transferred from, and the third letter of the instruction is the register transferred to; so (e.g.) TXY transfers the contents of the X register to the Y register.

The size of the destination register (i.e. the register transferred to) determines whether these instructions are 8-bit operations or 16-bit operations. When the destination register is 8 bits wide, 8 bits are transferred, and when the destination register is 16 bits wide, 16 bits are transferred.

The width of the accumulator is based on the m flag, and the width of the X and Y registers is based on the x flag, but the S register is always considered 16 bits wide. However, when the e flag is 1, SH is forced to $01, so in effect, TXS is an 8-bit transfer in this case since XL is transferred to SL and SH remains $01. Note that when the e flag is 0 and the x flag is 1 (i.e. 8-bit native mode), that XH is forced to zero, so after a TXS, SH will be $00, rather than $01. This is an important difference that must be accounted for if you want to run emulation mode code in (8-bit) native mode.

- The n flag is 1 when the high bit of the result (i.e. the value transferred from one register to the other) is 1, and the n flag is 0 when the high bit of the result is 0.

- The z flag is 1 when the result is zero, and the z flag is 0 when the result is nonzero.

Example: If the accumulator is $1234, the X register is $ABCD, and the m flag is 1, then after a TXA

- the accumulator will be $12CD
- the n flag will be 1 (since only $CD was actually transferred)
- the z flag will be 0

6.10.2 TCD TCS TDC TSC

```
Transfer C accumulator to Direct register
Transfer C accumulator to Stack pointer
Transfer Direct register to C accumulator
Transfer Stack pointer to C accumulator

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
5B 1   2           imp       *.....*. . TCD
1B 1   2           imp       ........ . TCS
7B 1   2           imp       *.....*. . TDC
3B 1   2           imp       *.....*. . TSC
```

TCD, TCS, TDC, and TSC transfer the C accumulator (the full 16-bit accumulator) to and from the D and S registers. These instructions always transfer 16 bits, no matter what the value of the m flag is. However, when the e flag is 1, SH is forced to $01, so in that case, TCS acts like an 8-bit transfer, by transferring the A accumulator (i.e. the low byte of the accumulator) to the SL register.

- The n flag is 1 when the high bit of the result (i.e. the value transferred from one register to the other) is 1, and the n flag is 0 when the high bit of the result is 0.
- The z flag is 1 when the result is zero, and the z flag is 0 when the result is nonzero.

TAD, TAS, TDA, and TSA are alternate names for these instructions, and are allowed by many 65C816 assemblers. Note that the "C" names are somewhat more accurate names than the "A" names, since these instructions are always 16-bit operations, while other instruction names where A refers to the accumulator (e.g. PHA) are 8-bit operations or 16-bit operations, depending on the value of the m flag.

Example: If the D register is $1234, then after a TDC

- the accumulator will be $1234
- the n flag will be 0
- the z flag will be 0

6.10.3 XBA

```
eXchange B and A accumulator

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
EB 1   3           imp       *.....*. . XBA
```

XBA exchanges the B accumulator and the A accumulator, i.e. it swaps the high byte and the low byte of the accumulator. Note that this is a swap rather than a copy (as is the case for the transfer instructions).

The n and z flags are always based on an 8-bit result, no matter what the value of the m flag is. Specifically, they are based on the A accumulator (i.e. the low byte of the accumulator) result; in other words, the final value of the A accumulator, which is the same as the initial value of the B accumulator.

- The n flag is 1 when the high bit of the result is 1, and the n flag is 0 when the high bit of the result is 0.
- The z flag is 1 when the result is zero, and the z flag is 0 when the result is nonzero.

SWA is an alternate name for this instruction, and is allowed by many 65C816 assemblers.

Example: If the accumulator is $6789, then after an XBA

- the accumulator will be $8967
- the n flag will be 0
- the z flag will be 0

6.10.4 XCE

```
eXchange Carry and Emulation flags

OP LEN CYCLES      MODE      nvmxdizc e SYNTAX
-- --- ----------- --------- ---------- ------
FB 1   2           imp       .......* * XCE
```

XCE exchanges (i.e. swaps) the c and e flags. This is the only instruction that changes the value of the e flag. Note that when the e flag is 1, the m and x flags are forced to 1 (and consequently the XH and YH registers are forced to $00), and the SH register is forced to $01.

Example: If the c flag is 0 and the e flag is 1, then after an XCE

- the c flag will be 1
- the e flag will be 0

APPENDIX: EMULATION MODE

The sections above which described the addressing modes and instructions also covered emulation mode. However, there are several points about emulation mode worth emphasizing.

To begin with, emulation mode has slightly different behavior than 8-bit (i.e. when the m and x flags are both 1) native mode. For instance, when updating an NMOS 6502 or 65C02 routine (or program) for 16-bit data, you may find it helpful to get it working in 8-bit native mode first, then modify it for 16-bit mode. The two biggest differences are (a) emulation mode has direct page wrap around (when DL is $00, which is typically the case), but native mode does not, and (b) after a TXS, in emulation mode, the stack is on page 1 (since SH is forced to $01), but in 8-bit native mode, TXS transfers all 16 bits of the X register (since the S register is 16 bits wide) thus the stack is then on page 0 (since XH is forced to $00 when the x flag is 1).

There are also minor differences. In native mode, RTI pulls 4 bytes (versus 3 in emulation mode). This is not a problem for interrupts, since 4 bytes are pushed in native mode, but some monitor/debugger programs use RTI to prepare the flags and then jump to a user-specified address (on the stack). Another difference is that in emulation mode, a branch taken to a different page takes 4 cycles, but in native mode it takes 3 cycles (except BRL) regardless of whether a page boundary is crossed or not. This is almost never an issue since routines which depend on exact timing are generally located at an address specifically chosen so that page boundary crossings will not occur (and thus a branch taken is 3 cycles in both emulation and native mode).

Something else that was not explicitly stated above is that "new" (again, "new" meaning "not present on the 65C02") addressing modes and instructions are functional (i.e. not NOPs) in emulation mode, and in most cases can be useful. (One exception is the block move instructions, MVN and MVP, which are of limited use in emulation mode, since the high byte of the X and Y register is forced to $00.) Why use "new" addressing modes and/or instructions in emulation mode? While the 65C816 has 24-bit addresses and 16-bit data, there are some useful subsets. Using 16-bit addresses and 16-bit data simplifies address decoding circuitry. Using 24-bit addresses and 8-bit data can also be useful (as long as there isn't much 16-bit calculation), since most instructions are faster when operating on 8-bit data, and 8-bit immediate data takes less space; thus it's as though the system is a 65C02 with a larger address space. With emulation mode, you can have 24-bit addresses, 8-bit data, and direct page wrap around.

However, because "new" addressing modes and/or instructions were intended for native mode, they do not incorporate behavior that is specific to emulation mode such as page wrap around. For example, if the D register is $0000 (and the e flag is 1), then LDA ($FF) uses a pointer whose low byte is at $0000FF and whose high byte is at $000000 (like the 65C02), but PEI $FF pushes a 16-bit value whose low byte is at $0000FF and whose high byte is at $000100.

Page 1 wrap around of the stack is similar. For example, when the S register is $0100 (and the e flag is 1), then PHP pushes the P register to $000100, leaving the S register $0001FF; a subsequent PLP pulls the P register from $000100, leaving the S register at $000100. By contrast, replacing PHP and PLP with PHB and PLB in the preceding example, PHB pushes to $000100 (like PHP), but PLB pulls from $000200. PEA, when the S register is $0100 (and the e flag is 1), has similar behavior; it pushes the high byte to $000100, the low byte to $0000FF, and leaves the stack pointer at $0001FE. Likewise for other "new" instructions that push to, and pull from, the stack. JSR (absolute,X) is "new", even though JSR was present on the 65C02 (with absolute addressing) and (absolute,X) addressing is available on the 65C02 (with JMP), since that combination of instruction and addressing mode was not available on the 65C02. Generally, page 1 wrap around is not an issue, since the stack pointer is frequently initialized (despite the fact that this is often unnecessary) to $01FF on the NMOS 6502 and 65C02, so the stack pointer won't be near $0100 when pushing or $01FF when pulling.

Another interesting fact is that direct page wrapping occurs (in emulation mode) when the DL register is $00; the DH register need not be zero. In fact, this can be very useful, since it allows you locate the direct page on any bank 0 page. For instance, if you have a program that makes heavy use of the direct page (e.g. a programming language interpreter, like BASIC, or an operating system) and you wish to avoid zero page conflicts with I/O routines, then you can adjust the D register when calling an I/O routine so that the program and the I/O routines use different pages for the direct page.

If, for whatever reason, you do not wish to have direct page wrap around in emulation mode, it will not occur when the DL register is nonzero.

Finally, emulation mode code does not have to be in bank 0, as long as there are no interrupts. As noted in sections 6.3.1, 6.3.1.1, and 6.3.2, interrupts do not push (and RTI does not pull) the bank byte on the stack, but as long as you can guarantee that interrupts will not occur when in a nonzero bank (other than RESET, since you wouldn't return from that), then nonzero banks can be used for code as well as data.

Last Updated September 28, 2015.