6502 Microprocessor Family

Notes about assembly language

Peter Mount, Area51.dev & Contributors

6502 Microprocessor Family

Notes about assembly language

Title 6502 Microprocessor Family
Subtitle Notes about assembly language
Author Peter Mount, Area51.dev & Contributors
Copyright CC BY-SA

CC BY-SA version 4.0 license

You are free to:

1. **Share** — copy and redistribute the material in any medium or format
2. **Adapt** — remix, transform, and build upon the material or any purpose, even commercially.

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

1. **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
2. **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
3. **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

You can read the full license here: https://creativecommons.org/licenses/by-sa/4.0/

Table of Contents

This section covers assembly language for the 6502 Microprocessor family including the 6510, 65C02 & 65816 processors.

1 - Opcodes

Instruction Set

In this section we cover every available instruction for both 8-bit & 16-bit processors.

1.1 - Arithmetic

Arithmetic operations

1.1.1 - ADC Add With Carry

Add With Carry

Adds the data in the operand with the contents of the accumulator. Add 1 to the result if the carry flag is set. Store the final result in the accumulator.

Binary/Decimal mode

If the d flag is clear then binary addition is performed. If the d flag set then Binary Coded Decimal (BCD) addition is performed.

Data size

On all processors, the data added from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data added is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Multi-precision arithmetic

In multi-precision (multi-word) arithmetic, the carry flag should be cleared before the low-order words are added. The addition will generate a new carry flag value based on that addition which will then be passed on to the next word.

For example, to add 1 to a 16-bit value at &70 on 8-bit processors:

```
1   CLC      ; Clear carry before first addition
2   LDA &70  ; Add 1 to low-order byte
3   ADC #1
4   STA &70
5   LDA &71  ; Add 0 to high order byte
6   ADC #0   ; This will add 1 if carry was set
7   STA &71  ; in the low-order byte
```

Flags Affected

Flags	n	v	-	-	-	-	z	c

- **n** Set if most-significant bit of result is set
- **v** Set if signed overflow
- **z** Set if result is zero
- **c** Set if unsigned overflow, clear if valid unsigned result

Instructions

Syntax	Opcode (hex)	Available on: 6502	65C02	65816	# of bytes	# of cycles	Addressing Mode
ADC *#const*	69	x	x	x	2[1]	2[2, 5]	Immediate
ADC *addr*	6D	x	x	x	3	4[2, 5]	Absolute
ADC *long*	6F			x	4	5[2, 5]	Absolute Long
ADC *dp*	65	x	x	x	2	3[2, 3, 5]	Direct Page
ADC (*dp*)	72		x	x	2	5[2, 3, 5]	Direct Page Indirect
ADC [*dp*]	67			x	2	6[2, 3, 5]	Direct Page Indirect Long
ADC *addr*,X	7D	x	x	x	3	4[2, 4, 5]	Absolute Indexed X
ADC *long*,X	7F			x	4	5[2, 5]	Absolute Long Indexed X
ADC *addr*,Y	79	x	x	x	3	4[2, 4, 5]	Absolute Indexed Y
ADC *dp*,X	75	x	x	x	2	4[2, 3, 5]	Direct Page Indexed X
ADC (*dp*,X)	61	x	x	x	2	6[2, 3, 5]	Direct Page Indexed Indirect X
ADC (*dp*),Y	71	x	x	x	2	5[2, 3, 4, 5]	Direct Page Indirect Indexed Y
ADC [*dp*],Y	77			x	2	6[2, 3, 5]	Direct Page Indirect Long Indexed Y
ADC *sr*,S	63			x	2	4[2, 5]	Stack Relative
ADC (*sr*,S),Y	73			x	2	7[2, 5]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary
5. 65C02: Add 1 cycle if d=1

1.1.2 - Decrement

Decrement by one a register or a memory location

The decrement instructions add one to either a register or a memory location.

Unlike subtracting 1 with ADC, these instructions does not use the Carry flag in any way. You can test for wraparound only by testing after every decrement to see if the result is zero or negative.

The d flag does not affect these instructions. The decrement is always in binary mode.

For all processors, the decrement is an 8-bit operation unless m=0 on the 65816 in which case the decrement is 16-bit.

DEC - Decrement

Decrement by 1 the contents of the memory location or accumulator.

DEX - Decrement Index Register X

Decrement by 1 the X index register.

DEY - Decrement Index Register Y

Decrement by 1 the Y index register.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	-

- **n** Set if most significant bit of the result is set
- **z** Set if result is zero

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
DEC A	3A		x	x	1	2	Accumulator
DEC *addr*	CE	x	x	x	3	6[1]	Absolute
DEC *dp*	C6	x	x	x	2	5[1, 2]	Direct Page
DEC *addr*,X	DE	x	x	x	3	7[1, 3]	Absolute Indexed X
DEC *dp*,X	D6	x	x	x	2	6[1, 2]	Direct Page Indexed X
DEX	CA	x	x	x	1	2	Implied
DEY	88	x	x	x	1	2	Implied

Notes:

1. 65816: Add 2 cycles if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0
3. 65C02: Subtract 1 cycle if no page boundary is crossed

1.1.3 - Increment

Increment by one a register or a memory location

The increment instructions add one to either a register or a memory location.

Unlike adding 1 with ADC, these instructions does not use the Carry flag in any way. You can test for wraparound only by testing after every increment to see if the result is zero or positive.

The d flag does not affect these instructions. The increment is always in binary mode.

For all processors, the increment is an 8-bit operation unless m=0 on the 65816 in which case the increment is 16-bit.

INC - Increment

Increment by 1 the contents of the memory location or accumulator.

INX - Increment Index Register X

Increment by 1 the X index register.

INY - Increment Index Register Y

Increment by 1 the Y index register.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	-

- **n** Set if most significant bit of the result is set
- **z** Set if result is zero

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
INC A	1A		x	x	1	2	Accumulator
INC *addr*	EE	x	x	x	3	6[1]	Absolute
INC *dp*	E6	x	x	x	2	5[1, 2]	Direct Page
INC *addr*,X	FE	x	x	x	3	7[1, 3]	Absolute Indexed X
INC *dp*,X	F6	x	x	x	2	6[1, 2]	Direct Page Indexed X
INX	E8	x	x	x	1	2	Implied
INY	C8	x	x	x	1	2	Implied

Notes:

1. 65816: Add 2 cycles if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0
3. 65C02: Subtract 1 cycle if no page boundary is crossed

1.1.4 - SBC Subtract with Borrow from Accumulator

Subtract with Borrow

Subtracts the data in the operand with the contents of the accumulator. Subtract 1 from the result if the carry flag is clear. Store the final result in the accumulator.

Binary/Decimal mode

If the d flag is clear then binary subtraction is performed. If the d flag set then Binary Coded Decimal (BCD) subtraction is performed.

Data size

On all processors, the data subtracted from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data subtracted is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Multi-precision arithmetic

In multi-precision (multi-word) arithmetic, the carry flag should be set before the low-order words are subtracted. The subtraction will generate a new carry flag value based on that subtraction which will then be passed on to the next word.

For example, to subtract 1 from a 16-bit value at &70 on 8-bit processors:

```
1   SEC      ; Set carry before first subtraction
2   LDA &70  ; Subtract 1 from low-order byte
3   SBC #1
4   STA &70
5   LDA &71  ; Subtract 0 to high order byte
6   SBC #0   ; This will subtract 1 if carry was clear
7   STA &71  ; from the low-order byte
```

Flags Affected

Flags	n	v	-	-	-	-	z	c

- **n** Set if most-significant bit of result is set
- **v** Set if signed overflow
- **z** Set if result is zero
- **c** Set if unsigned borrow not required, clear if required

Instructions

Syntax	Opcode *(hex)*	Available on: 6502	65C02	65816	# of bytes	# of cycles	Addressing Mode
SBC *#const*	E9	x	x	x	2[1]	2[2, 5]	Immediate
SBC *addr*	ED	x	x	x	3	4[2, 5]	Absolute
SBC *long*	EF			x	4	5[2, 5]	Absolute Long
SBC *dp*	E5	x	x	x	2	3[2, 3, 5]	Direct Page
SBC (*dp*)	F2		x	x	2	5[2, 3, 5]	Direct Page Indirect
SBC [*dp*]	E7			x	2	6[2, 3, 5]	Direct Page Indirect Long
SBC *addr*,X	FD	x	x	x	3	4[2, 4, 5]	Absolute Indexed X
SBC *long*,X	FF			x	4	5[2, 5]	Absolute Long Indexed X
SBC *addr*,Y	F9	x	x	x	3	4[2, 4, 5]	Absolute Indexed Y
SBC *dp*,X	F5	x	x	x	2	4[2, 3, 5]	Direct Page Indexed X
SBC (*dp*,X)	E1	x	x	x	2	6[2, 3, 5]	Direct Page Indexed Indirect X
SBC (*dp*),Y	F1	x	x	x	2	5[2, 3, 4, 5]	Direct Page Indirect Indexed Y
SBC [*dp*],Y	F7			x	2	6[2, 3, 5]	Direct Page Indirect Long Indexed Y
SBC *sr*,S	E3			x	2	4[2, 5]	Stack Relative
SBC (*sr*,S),Y	F3			x	2	7[2, 5]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary
5. 65C02: Add 1 cycle if d=1

1.2 - Binary operations

Operations that work in Binary or individual Bits

1.2.1 - AND

And Accumulator with Memory

AND performs a logical And of the value in the accumulator with that of the memory location with the result stored in the accumulator.

The result will be each bit in the accumulator will be set ONLY if that same bit was set in the original accumulator value and the memory. If the bits were different then the resultant bit will be 0.

		Second Operand	
		0	**1**
First Operand	**0**	0	0
	1	0	1

AND truth table

For 8-bit processors n has the value of bit 7 and v the value of bit 6 of the memory location.

For 16-bit processors, when m=0, n has the value of bit 15 and v the value of bit 14 of the memory location.

Second it performs a logical AND of the memory and the accumulator. If the result is zero the z flag is set.

In both operations, the contents of the accumulator and memory are not modified.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	-

- **n** Set if most significant bit of result is set
- **z** Set if result is zero, otherwise clear

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
AND *#const*	29	x	x	x	2[1]	2[2]	Immediate
AND *addr*	2D	x	x	x	3	4[2]	Absolute
AND *long*	2F			x	4	5[2]	Absolute Long
AND *dp*	25	x	x	x	2	3[2, 3]	Direct Page
AND (*dp*)	32		x	x	2	5[2, 3]	Direct Page Indirect
AND [*dp*]	27			x	2	6[2, 3]	Direct Page Indirect Long
AND *addr*,X	3D	x	x	x	3	4[2, 4]	Absolute Indexed X
AND *long*,X	3F			x	4	5[2]	Absolute Long Indexed X
AND *addr*,Y	39	x	x	x	3	4[2, 4]	Absolute Indexed Y
AND *dp*,X	35	x	x	x	2	4[2, 3]	Direct Page Indexed X
AND (*dp*,X)	21	x	x	x	2	6[2, 3]	Direct Page Indexed Indirect X
AND (*dp*),Y	31	x	x	x	2	5[2, 3, 4]	Direct Page Indirect Indexed Y
AND [*dp*],Y	37			x	2	6[2, 3]	Direct Page Indirect Long Indexed Y
AND *sr*,S	23			x	2	4[2]	Stack Relative
AND (*sr*,S),Y	33			x	2	7[2]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.2.2 - BIT

Test Memory Bits against Accumulator

Bit is a dual-purpose instruction which performs operations against the accumulator and memory. It is usually used immediately preceding a conditional branch instruction

First it set's the n flag to reflect the value of the high bit of the data in memory and the v flag to the next-to-highest bit of that data.

For 8-bit processors n has the value of bit 7 and v the value of bit 6 of the memory location.

For 16-bit processors, when m=0, n has the value of bit 15 and v the value of bit 14 of the memory location.

Second it performs a logical AND of the memory and the accumulator. If the result is zero the z flag is set.

In both operations, the contents of the accumulator and memory are not modified.

Flags Affected

Flags	**n**	**v**	-	-	-	-	**z**	-

- **n** Takes value of most significant bit of memory data, not in immediate addressing
- **v** Takes value of the next-to-highest bit of memory data, not in immediate addressing
- **z** Set if logical AND of memory & accumulator is zero, otherwise clear

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
BIT *#const*	89		x	x	2[1]	2[2]	Immediate
BIT *addr*	2C	x	x	x	3	4[2]	Absolute
BIT *dp*	24	x	x	x	2	5[2, 3]	Direct Page
BIT *addr*,X	3C		x	x	3	4[2, 4]	Absolute Indexed X
BIT *dp*,X	34		x	x	2	4[2, 3]	Direct Page Indexed X

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 658116: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.2.3 - EOR - Exclusive OR

Exclusive-OR Accumulator with Memory

EOR performs a bitwise logical Exclusive-OR of the value in the accumulator with that of the memory location with the result stored in the accumulator.

The result will be each bit in the accumulator will be set ONLY if that same bit in the original accumulator value and the memory differ. If the bits were the same then the resultant bit will be 0.

		Second Operand	
		0	**1**
First Operand	**0**	0	1
	1	1	0

Exclusive OR truth table

For 8-bit processors n has the value of bit 7 and v the value of bit 6 of the memory location.

For 16-bit processors, when m=0, n has the value of bit 15 and v the value of bit 14 of the memory location.

Second it performs a logical AND of the memory and the accumulator. If the result is zero the z flag is set.

In both operations, the contents of the accumulator and memory are not modified.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	-

- **n** Set if most significant bit of result is set
- **z** Set if result is zero, otherwise clear

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
EOR *#const*	49	x	x	x	2[1]	2[2]	Immediate
EOR *addr*	4D	x	x	x	3	4[2]	Absolute
EOR *long*	4F			x	4	5[2]	Absolute Long
EOR *dp*	45	x	x	x	2	3[2, 3]	Direct Page
EOR (*dp*)	52		x	x	2	5[2, 3]	Direct Page Indirect
EOR [*dp*]	47			x	2	6[2, 3]	Direct Page Indirect Long
EOR *addr*,X	5D	x	x	x	3	4[2, 4]	Absolute Indexed X
EOR *long*,X	5F			x	4	5[2]	Absolute Long Indexed X
EOR *addr*,Y	59	x	x	x	3	4[2, 4]	Absolute Indexed Y
EOR *dp*,X	55	x	x	x	2	4[2, 3]	Direct Page Indexed X
EOR (*dp*,X)	41	x	x	x	2	6[2, 3]	Direct Page Indexed Indirect X
EOR (*dp*),Y	51	x	x	x	2	5[2, 3, 4]	Direct Page Indirect Indexed Y
EOR [*dp*],Y	57			x	2	6[2, 3]	Direct Page Indirect Long Indexed Y
EOR *sr*,S	43			x	2	4[2]	Stack Relative
EOR (*sr*,S),Y	53			x	2	7[2]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.2.4 - ORA - OR Accumulator with memory

OR Accumulator with Memory

ORA performs a bitwise logical OR of the value in the accumulator with that of the memory location with the result stored in the accumulator.

The result will be each bit in the accumulator will be set if either the same bit in the original accumulator value and the memory are set.

		Second Operand	
		0	**1**
First Operand	**0**	0	1
	1	1	1

OR truth table

For 8-bit processors n has the value of bit 7 and v the value of bit 6 of the memory location.

For 16-bit processors, when m=0, n has the value of bit 15 and v the value of bit 14 of the memory location.

Second it performs a logical AND of the memory and the accumulator. If the result is zero the z flag is set.

In both operations, the contents of the accumulator and memory are not modified.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	-

- **n** Set if most significant bit of result is set
- **z** Set if result is zero, otherwise clear

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
ORA *#const*	09	x	x	x	2[1]	2[2]	Immediate
ORA *addr*	0D	x	x	x	3	4[2]	Absolute
ORA *long*	0F			x	4	5[2]	Absolute Long
ORA *dp*	05	x	x	x	2	3[2, 3]	Direct Page
ORA (*dp*)	12		x	x	2	5[2, 3]	Direct Page Indirect
ORA [*dp*]	07			x	2	6[2, 3]	Direct Page Indirect Long
ORA *addr*,X	1D	x	x	x	3	4[2, 4]	Absolute Indexed X
ORA *long*,X	1F			x	4	5[2]	Absolute Long Indexed X
ORA *addr*,Y	19	x	x	x	3	4[2, 4]	Absolute Indexed Y
ORA *dp*,X	15	x	x	x	2	4[2, 3]	Direct Page Indexed X
ORA (*dp*,X)	01	x	x	x	2	6[2, 3]	Direct Page Indexed Indirect X
ORA (*dp*),Y	11	x	x	x	2	5[2, 3, 4]	Direct Page Indirect Indexed Y
ORA [*dp*],Y	17			x	2	6[2, 3]	Direct Page Indirect Long Indexed Y
ORA *sr*,S	03			x	2	4[2]	Stack Relative
ORA (*sr*,S),Y	13			x	2	7[2]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.2.5 - Rotate Bits

Test Memory Bits against Accumulator

The rotate instructions shifts the contents of the accumulator or memory location one bit either to the left or right.

The ROL & ROR instructions will shift in the carry flag into the value. The ASL & LSR instructions shift in 0. The carry flag is set to the bit that was shifted out of the value.

On all processors the data shifted is 8 bits.

On the 65816 with m=0, the data shifted is 16 bits.

Operation	Opcode	7 6 5 4 3 2 1 0
Shift left	ASL	C←←←←←←←←0
Shift left with carry	ROL	C←←←←←←←←C
Shift right	LSR	0→→→→→→→→C
Shift right with carry	ROR	C→→→→→→→→C

Effect on memory for 8 bit operations.

ASL - Shift Memory or Accumulator Left

Shift the value left one bit. The left most bit is transferred into the carry flag. The right most bit is cleared.

The arithmetic result of the operation is an unsigned multiplication by two.

LSR - Logical Shift Memory or Accumulator Right

Shift the value right one bit. The right most bit is transferred into the carry flag. The left most bit is cleared.

The arithmetic result of the operation is an unsigned division by two.

ROL - Rotate Memory or Accumulator Left

Shift the value left one bit. The right most bit is set to the initial value of the carry flag. The left most bit is transferred into the carry flag.

ROR - Rotate Memory or Accumulator Right

Shift the value right one bit. The left most bit is set to the initial value of the carry flag. The right most bit is transferred into the carry flag.

Multi-word shifts

These instructions can be combined to handle multiple word values:

Multi-word shift left

To shift left multiple words, use ASL for the first operation then ROL for the subsequent words.

```
1   ; Shift 16-bit value at &70 left 1 bit.
2   ; This is effectively a multiplication by 2
3   ASL &70 ; Shift left low-order byte
4   ROL &71 ; Shift left high-order byte
5   ; Carry will be set if we overflowed
6
```

For higher precision simply add an additional ROL for the next order byte.

Multi-word shift right

To shift right multiple words, use LSR for the first operation then ROR for the subsequent words. Unlike shifting left, here we have to start with the high-order byte first.

```
1   ; Shift 16-bit value at &70 right 1 bit.
2   ; This is effectively a division by 2
3   LSR &71 ; Shift right high-order byte
4   ROR &70 ; Shift right low-order byte
5   ; Carry will have the remainder
6
```

For higher precision just start the LSR on the higher order byte & use ROL for each lower order.

Flags Affected

Flags	n	-	-	-	-	-	z	c

- **n** Set if the most significant bit of the result is set
- **z** Set if the result is zero
- **c** The value of the bit shifted out of the result

Instructions

	Opcode	Available on:			# of	# of	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
ASL A	0A	x	x	x	1	2	Accumulator
ASL *addr*	0E	x	x	x	3	6	Absolute
ASL *dp*	06	x	x	x	2	5[1, 2]	Direct Page
ASL *addr*,X	1E	x	x	x	3	7[1, 3]	Absolute Indexed X
ASL *dp*,X	16	x	x	x	2	6[1, 2]	Direct Page Indexed X
LSR A	4A	x	x	x	1	2	Accumulator
LSR *addr*	4E	x	x	x	3	6	Absolute
LSR *dp*	46	x	x	x	2	5[1, 2]	Direct Page
LSR *addr*,X	5E	x	x	x	3	7[1, 3]	Absolute Indexed X
LSR *dp*,X	56	x	x	x	2	6[1, 2]	Direct Page Indexed X
ROL A	2A	x	x	x	1	2	Accumulator
ROL *addr*	2E	x	x	x	3	6	Absolute
ROL *dp*	26	x	x	x	2	5[1, 2]	Direct Page
ROL *addr*,X	3E	x	x	x	3	7[1, 3]	Absolute Indexed X
ROL *dp*,X	36	x	x	x	2	6[1, 2]	Direct Page Indexed X
ROR A	6A	x	x	x	1	2	Accumulator
ROR *addr*	6E	x	x	x	3	6	Absolute
ROR *dp*	66	x	x	x	2	5[1, 2]	Direct Page
ROR *addr*,X	7E	x	x	x	3	7[1, 3]	Absolute Indexed X
ROR *dp*,X	76	x	x	x	2	6[1, 2]	Direct Page Indexed X

Notes:

1. 65816: Add 2 cycles if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0
3. 65C02: Subtract 1 cycle if no page boundary is crossed

1.2.6 - TRB & TSB

Test & Set/Reset Memory Bits against Accumulator

TRB - Test & Reset memory against Accumulator

TRB logically AND's the *complement* of the accumulator with the data at an address and stores the result in that address.

This has the effect of clearing each memory bit which is set in the accumulator, leaving the other bits unchanged.

The z flag is set based on a different operation. It's set if the memory location once set logically AND the accumulator (not it's compliment) is zero.

For 8-bit processors or when m=1, the values in the accumulator & memory are 8-bit.

For 16-bit processors, when m=0, the values in the accumulator & memory are 16-bit.

TSB - Test & Set memory against Accumulator

TSB is identical to TRB except it sets the bits defined in the Accumulator not reset them.

Flags Affected

Flags	-	-	-	-	-	-	**z**	-

z Set if logical AND of memory & accumulator is zero, otherwise clear

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
TRB *addr*	1C		x	x	3	6[1]	Absolute
TRB *dp*	14		x	x	2	5[1, 2]	Direct Page
TSB *addr*	0C		x	x	3	6[1]	Absolute
TSB *dp*	04		x	x	2	5[1, 2]	Direct Page

Notes:

1. 65816: Add 2 cycles if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0

1.3 - Program Flow

Branch & Jumps

1.3.1 - Flags

Flag manipulation

The flag instructions manipulate some of the flags in the status register.

CLC - Clear Carry Flag

CLC is used prior to addition with the ADC instruction to keep the carry flag affecting the result.

On the 6502 a CLC before a BCC instruction can be used to implement a branch always, which is relocatable. This is unnecessary since the 65C02 with it's BRA instruction.

On the 16-bit processors a CLC followed by XCE instruction is used to switch the 65802 & 65816 processors into native mode.

SEC - Set Carry Flag

SEC is used prior to subtraction using the SBC instruction to keep the carry flag affecting the result.

On the 16-bit processors a SEC followed by XCE instruction is used to switch the 65802 & 65816 processors into 6502 emulation mode.

CLD - Clear Decimal Mode

CLD is used to switch the processors into binary mode so that the ADC & SBC instructions will perform binary not BCD arithmetic.

SED - Set Decimal Mode

SED is used to switch the processors into decimal mode so that the ADC & SBC instructions will perform BCD not binary arithmetic.

CLI - Clear Interrupt Disable Flag

CLI is used to re-enable hardware interrupts.

When the processor starts the interrupt handler it sets the i flag to prevent another interrupt to occur during that handler. If the handler want's to allow interrupts to happen whilst it's handling a previous one it can use CLI to re-enable them. The handler doesn't need to use CLI as the RTI (ReTurn from Interrupt) instruction will clear the i flag automatically.

In user code, CLI can be used to re-enable interrupts after an SEI instruction. This is usually used during time-critical code or code that cannot be interrupted.

SEI - Clear Interrupt Disable Flag

SEI is used to disable hardware interrupts.

When the i bit is set, maskable hardware interrupts are ignored. When the processor starts the interrupt handler it sets the i flag to prevent another interrupt to occur during that handler. If the handler want's to allow interrupts to happen whilst it's handling a previous one it can use CLI to re-enable them. The handler doesn't need to use CLI as the RTI (ReTurn from Interrupt) instruction will clear the i flag automatically.

In user code, SEI can be used to disable interrupts when it needs to run time-critical code or code that cannot be interrupted. It should then use CLI once it's finished that time-critical code.

CLV - Clear Overflow Flag

CLV clears the overflow flag.

Unlike other clear flag instructions, there is no set overflow flag available. The only way the overflow flag can be set is either:

- The BIT instruction will set overflow if bit 6 of the mask & memory is set
- The 65816 REP instruction can clear the overflow
- Use the Overflow pin on the processor. This is rarely used & is often not even connected.

On the 6502 a CLC before a BVC instruction can be used to implement a branch always, which is relocatable. This is unnecessary since the 65C02 with it's BRA instruction.

REP - Reset Status Bits

For each bit set in the operand byte, reset the corresponding bit in the status register. For each bit not set in the operand byte leaves the corresponding bit unchanged.

This instruction lets you clear any flag or flags in a single instruction. It is the only direct means of resetting the m & x flags.

In 6502 emulation mode (e=1) neither the b flag or bit 5 (the 6502's non-flag bit) is affected by this instruction.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
6502 emulation mode e=1	n	v			d	i	z	c
65816 native mode e=0	n	v	m	x	d	i	z	c

Flags Affected

SEP - Set Status Bits

For each bit set in the operand byte, set the corresponding bit in the status register. For each bit not set in the operand byte leaves the corresponding bit unchanged.

This instruction lets you set any flag or flags in a single instruction. It is the only direct means of setting the m & x flags.

In 6502 emulation mode (e=1) neither the b flag or bit 5 (the 6502's non-flag bit) is affected by this instruction.

The bit's in the operand & their relationship with the status register is the same as the REP instruction.

Instructions

		Opcode	**Available on:**			**# of**	**# of**	
Syntax	**Action**	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
CLC	Clear Carry	18	x	x	x	1	2	Implied
SEC	Set Carry	38	x	x	x	1	2	Implied
CLD	Clear Decimal	D8	x	x	x	1	2	Implied
SED	Set Decimal	F8	x	x	x	1	2	Implied
CLI	Enable hardware interrupts	58	x	x	x	1	2	Implied
SEI	Disable hardware interrupts	78	x	x	x	1	2	Implied
CLV	Clear Overflow	B8	x	x	x	1	2	Implied
REP *#const*	Reset status bits	C2			x	2	3	Immediate
SEP *#const*	Set status bits	E2			x	2	3	Immediate

1.3.2 - Compare Accumulator

Compare Accumulator with Memory

CMP subtracts the data at the address in the operand from the contents of the accumulator, setting the n, z & c flags based on the result. The Accumulator & Memory are unaffected by this operation.

Data size

On all processors, the data added from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data added is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	**c**

- **n** Set if most-significant bit of result is set
- **z** Set if result is zero
- **c** Set if register value greater than or equal or Cleared if less than memory value

Instructions

Syntax	**Opcode** *(hex)*	**Available on: 6502**	**65C02**	**65816**	**# of bytes**	**# of cycles**	**Addressing Mode**
CMP *#const*	C9	x	x	x	2[1]	2[2]	Immediate
CMP *addr*	CD	x	x	x	3	4[2]	Absolute
CMP *long*	CF			x	4	5[2]	Absolute Long
CMP *dp*	C5	x	x	x	2	3[2, 3]	Direct Page
CMP (*dp*)	D2		x	x	2	5[2, 3]	Direct Page Indirect
CMP [*dp*]	C7			x	2	6[2, 3]	Direct Page Indirect Long
CMP *addr*,X	DD	x	x	x	3	4[2, 4]	Absolute Indexed X
CMP *long*,X	DF			x	4	5[2]	Absolute Long Indexed X
CMP *addr*,Y	D9	x	x	x	3	4[2, 4]	Absolute Indexed Y
CMP *dp*,X	D5	x	x	x	2	4[2, 3]	Direct Page Indexed X
CMP (*dp*,X)	C1	x	x	x	2	6[2, 3]	Direct Page Indexed Indirect X
CMP (*dp*),Y	D1	x	x	x	2	5[2, 3, 4]	Direct Page Indirect Indexed Y
CMP [*dp*],Y	D7			x	2	6[2, 3]	Direct Page Indirect Long Indexed Y
CMP *sr*,S	C3			x	2	4[2]	Stack Relative
CMP (*sr*,S),Y	D3			x	2	7[2]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.3.3 - Compare Index Register

Compare Index Register with Memory

The CPX & CPY instructions subtracts the data at the address in the operand from the contents of the relevant index register, setting the n, z & c flags based on the result. The register & Memory are unaffected by this operation.

The primary use of the CPX or CPY instructions is to test the value of the index register against loop boundaries.

Data size

On all processors, the data added from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data added is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	**c**

- **n** Set if most-significant bit of result is set
- **z** Set if result is zero
- **c** Set if register value greater than or equal or Cleared if less than memory value

Instructions

Syntax	**Opcode** *(hex)*	**Available on: 6502**	**65C02**	**65816**	**# of bytes**	**# of cycles**	**Addressing Mode**
CPX *#const*	E0	x	x	x	2[1]	2[2]	Immediate
CPX *addr*	EC	x	x	x	3	4[2]	Absolute
CPX *dp*	E4	x	x	x	2	3[2, 3]	Direct Page
CPY *#const*	C0	x	x	x	2[1]	2[2]	Immediate
CPY *addr*	CC	x	x	x	3	4[2]	Absolute
CPY *dp*	C4	x	x	x	2	3[2, 3]	Direct Page

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0

1.3.4 - Branch

Perform a test & branch based on that test

The branch instructions perform a test against one of the processor's flags. Depending on the instruction a branch is taken if it is either clear or set.

If the branch is taken, a 1-byte signed displacement in the second byte of the instruction is sign-extended to 16-bits and added to the Program Counter. If the branch is not taken then the instruction immediately following the 2-byte instruction is executed.

The allowable range of the displacement is -128 to +127 from the instruction immediately following the branch.

BCC - Branch if Carry Clear

BCC tests the Carry flag and branches if it is clear.

It can be used in several ways:

- Test the result of a shift into the carry
- Determine if the result of a comparison is *less than*

Some assemblers accept **BLT** *(Branch if Less Than)* as an alternate mnemonic for **BCC**.

BCS - Branch if Carry Set

BCS tests the Carry flag and branches if it is set.

It can be used in several ways:

- Test the result of a shift into the carry
- Determine if the result of a comparison is *greater than or equal*

Some assemblers accept **BGE** *(Branch if Greater Than or Equal)* as an alternate mnemonic for **BCS**.

BEQ - Branch if Equal

BEQ tests the Zero flag and branches if it is set.

It can be used in several ways:

- Test the result of a comparison is equal
- Test the result of an Increment or Decrement operation is zero, useful in loops.
- Test the value just loaded is zero
- Test the result of an arithmetic operation is zero

BNE - Branch if Not Equal

BNE tests the Zero flag and branches if it is clear.

It can be used in several ways:

- Test the result of a comparison is not equal
- Test the result of an Increment or Decrement operation is not zero
- Test the value just loaded is not zero
- Test the result of an arithmetic operation is not zero

BMI - Branch if Minus

BMI tests the Negative flag and branches if it is set. The high bit of the value most recently affected will set the N flag. On 8-bit operations this is bit 7. On 16-bit operations (65816 only) this is bit 15.

This is normally used to determine if a two's-complement value is negative but can also be used in a loop to determine if zero has been passed when looping down through zero (the initial value must be positive)

BPL - Branch if Positive

BPL tests the Negative flag and branches if it is clear. The high bit of the value most recently affected will set the N flag. On 8-bit operations this is bit 7. On 16-bit operations (65816 only) this is bit 15.

This is normally used to determine if a two's-complement value is positive or if the high bit of the value is clear.

BVC - Branch if Overflow Clear

BVC tests the Overflow flag and branches if it is clear.

On the 6502 only 3 instructions alter the overflow flag: ADC, SBC & CLV.

On the 65C02 the BIT instruction also alters the overflow flag.

The PLP & RTI alter the flags as they restore all flags from the stack.

On the 65816 the SEP & REP instructions modify the v flag.

On some processors there's a Set Overflow hardware signal available, but on many systems there is no connection to that pin.

BVS - Branch if Overflow Set

BVS tests the Overflow flag and branches if it is set. It has the same limitations as the BVC instruction.

Flags Affected

None.

Instructions

		Opcode	Available on:			# of	# of	
Syntax	**Branch if**	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
BCC *nearlabel*	Carry clear	90	x	x	x	2	2[1, 2]	Program Counter Relative
BCS *nearlabel*	Carry set	B0	x	x	x	2	2[1, 2]	Program Counter Relative
BEQ *nearlabel*	Equal, z=1	F0	x	x	x	2	2[1, 2]	Program Counter Relative
BNE *nearlabel*	Not Equal, z=0	D0	x	x	x	2	2[1, 2]	Program Counter Relative
BMI *nearlabel*	Minus, n=1	30	x	x	x	2	2[1, 2]	Program Counter Relative
BPL *nearlabel*	Positive, n=0	10	x	x	x	2	2[1, 2]	Program Counter Relative
BVC *nearlabel*	Overflow clear, v=0	50	x	x	x	2	2[1, 2]	Program Counter Relative
BVS *nearlabel*	Overflow set, v=1	70	x	x	x	2	2[1, 2]	Program Counter Relative

Notes:

1. Add 1 cycle if branch taken
2. Add 1 more cycle if branch taken crosses page boundary on a 6502, 65C02 or a 65816 in 6502 emulation mode (e=1)

1.3.5 - Jump to location

Transfer control to the address specified by the operand field.

The branch instructions sets the Program Counter to a new value from which the next instruction will be taken.

JMP - Jump to location

The program counter is loaded with the target address. If a long JMP is executed the bank is loaded from the third byte of the address.

Some assemblers accept **JML** as an alternate mnemonic for **JMP long**.

BRA - Branch Always

A branch is always taken, no test is performed. It is equivalent to a JMP instruction, except that as it uses a signed displacement it is only 2 bytes in length instead of 3 for JMP. In addition, because it uses displacements, code using BRA is relocatable.

The 1-byte signed displacement in the second byte of the instruction is sign-extended to 16-bits and added to the Program Counter. If the branch is not taken then the instruction immediately following the 2-byte instruction is executed.

The allowable range of the displacement is -128 to +127 from the instruction immediately following the branch.

BRA was introduced with the 65C02 processor.

BRL - Branch Always Long

A branch is always taken, no test is performed. It is equivalent to a BRA instruction, except that BRL is a 3 byte instruction. The two bytes after the opcode form a 16-bit signed displacement from the Program Counter.

The allowable range of the displacement is anywhere within the current 64K program bank.

The advantage of BRL is that it makes code relocatable, although it is 1 cycle slower than the absolute JMP instruction.

BRL was introduced with the 65802 processor.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
BRA *nearlabel*	80		x	x	2	3[3]	Program Counter Relative
BRL *label*	82			x	3	4	Program Counter Relative Long
JMP *addr*	4C	x	x	x	3	3	Absolute
JMP (*addr*)	6C	x	x	x	3	5[1, 2]	Absolute Indirect
JMP (*addr*,X)	7C		x	x	3	6	Absolute Indexed Indirect
JMP *long*	5C			x	4	4	Absolute Long
JMP [*addr*]	DC			x	3	6	Absolute Indirect Long

Notes:

1. Add 1 cycle if 65C02
2. 6502: If low byte of address is 0xFF yields incorrect result
3. Add 1 more cycle if branch taken crosses page boundary on a 6502, 65C02 or a 65816 in 6502 emulation mode (e=1)

1.3.6 - Subroutines

Calling subroutines

The JSR & RTS instructions allows for subroutines to be implemented. The work by utilising 2 bytes on the stack consisting of the address before the next instruction to execute when the subroutine returns - not the actual address of that instruction.

On the 16-bit 65816 there are the JSL & RTL instructions. These use 3 bytes on the stack. The extra byte is the return bank address. Like RTS the address on the stack is the address before the next instruction not the actual instruction

For Interrupt routines there's the RTI instruction. That instruction is on the Interrupt page.

JSR - Jump to Subroutine

Transfer control to a subroutine, pushing the return address onto the stack. The 16-bit address placed on the stack is the address of the 3rd byte of the instruction, not the address of the next instruction.

Subroutines called by JSR must return using the RTS instruction.

Some assemblers recognise JSR as an alternate to the 65816 JSL instruction where if the address is greater than &FFFF then the 24 bit JSL instruction is used instead.

RTS - Return from Subroutine

Returns from a subroutine called by JSR. It pulls the 16-bit program counter from the stack, incrementing it by one so that the next instruction is the one immediately after the calling JSR instruction.

JSL - Jump to Subroutine Long

Transfer control to a subroutine, pushing the return address onto the stack. The 24-bit address placed on the stack is the address of the 4th byte of the instruction, not the address of the next instruction.

Subroutines called by JSL must return using the RTL instruction.

RTL - Return from Subroutine Long

Returns from a subroutine called by JSL. It pulls the 24-bit program counter from the stack, incrementing it by one so that the next instruction is the one immediately after the calling JSL instruction.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
JSL *long*	22			x	4	8	Absolute Long
JSR *addr*	20	x	x	x	3	6	Absolute
JSR (*addr*,X)	FC			x	3	8	Absolute Indexed Indirect
RTL	6B			x	1	6	Implied
RTS	60	x	x	x	1	6	Implied

1.4 - Registers

Register operations

1.4.1 - LDA

Load Accumulator from Memory

Load the accumulator with data from memory.

On all processors, the data loaded from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data added is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

Flags	**n**	-	-	-	-	-	**z**	-

- **n** Set if most-significant bit of result is set
- **z** Set if result is zero

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
LDA *#const*	A9	x	x	x	2[1]	2[2]	Immediate
LDA *addr*	AD	x	x	x	3	4[2]	Absolute
LDA *long*	AF			x	4	5[2]	Absolute Long
LDA *dp*	A5	x	x	x	2	3[2, 3]	Direct Page
LDA (*dp*)	B2		x	x	2	5[2, 3]	Direct Page Indirect
LDA [*dp*]	A7			x	2	6[2, 3]	Direct Page Indirect Long
LDA *addr*,X	BD	x	x	x	3	4[2, 4]	Absolute Indexed X
LDA *long*,X	BF			x	4	5[2]	Absolute Long Indexed X
LDA *addr*,Y	B9	x	x	x	3	4[2, 4]	Absolute Indexed Y
LDA *dp*,X	B5	x	x	x	2	4[2, 3]	Direct Page Indexed X
LDA (*dp*,X)	A1	x	x	x	2	6[2, 3]	Direct Page Indexed Indirect X
LDA (*dp*),Y	B1	x	x	x	2	5[2, 3, 4]	Direct Page Indirect Indexed Y
LDA [*dp*],Y	B7			x	2	6[2, 3]	Direct Page Indirect Long Indexed Y
LDA *sr*,S	A3			x	2	4[2]	Stack Relative
LDA (*sr*,S),Y	B3			x	2	7[2]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.4.2 - LDX

Load Index Register X from Memory

Load index register X with data from memory.

On all processors, the data loaded from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data added is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

Flags	n	-	-	-	-	-	z	-

- **n** Set if most-significant bit of result is set
- **z** Set if result is zero

Instructions

	Opcode	Available on:			# of	# of	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
LDX *#const*	A2	x	x	x	2[1]	2[2]	Immediate
LDX *addr*	AE	x	x	x	3	4[2]	Absolute
LDX *dp*	A6	x	x	x	2	3[2, 3]	Direct Page
LDX *addr*,X	BE	x	x	x	3	4[2, 4]	Absolute Indexed X
LDX *dp*,X	B6	x	x	x	2	4[2, 3]	Direct Page Indexed X

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.4.3 - LDY

Load Index Register Y from Memory

Load index register Y with data from memory.

On all processors, the data loaded from memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data added is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

Flags	n	-	-	-	-	-	z	-

- **n** Set if most-significant bit of result is set
- **z** Set if result is zero

Instructions

	Opcode	Available on:			# of	# of	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
LDY *#const*	A0	x	x	x	2[1]	2[2]	Immediate
LDY *addr*	AC	x	x	x	3	4[2]	Absolute
LDY *dp*	A4	x	x	x	2	3[2, 3]	Direct Page
LDY *addr*,X	BC	x	x	x	3	4[2, 4]	Absolute Indexed X
LDY *dp*,X	B4	x	x	x	2	4[2, 3]	Direct Page Indexed X

Notes:

1. 65816: Add 1 byte if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if low byte of Direct Page register is not 0
4. Add 1 cycle if adding index crosses a page boundary

1.4.4 - STA

Store Accumulator to Memory

Stores the accumulator into memory.

On all processors, the data written to memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data written is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
STA *addr*	8D	x	x	x	3	4[1]	Absolute
STA *long*	8F			x	4	5[1]	Absolute Long
STA *dp*	85	x	x	x	2	3[1, 2]	Direct Page
STA (*dp*)	92		x	x	2	5[1, 2]	Direct Page Indirect
STA [*dp*]	87			x	2	6[1, 2]	Direct Page Indirect Long
STA *addr*,X	9D	x	x	x	3	4[1]	Absolute Indexed X
STA *long*,X	9F			x	4	5[1]	Absolute Long Indexed X
STA *addr*,Y	99	x	x	x	3	4[1]	Absolute Indexed Y
STA *dp*,X	95	x	x	x	2	4[1, 2]	Direct Page Indexed X
STA (*dp*,X)	81	x	x	x	2	6[1, 2]	Direct Page Indexed Indirect X
STA (*dp*),Y	91	x	x	x	2	5[1, 2]	Direct Page Indirect Indexed Y
STA [*dp*],Y	97			x	2	6[1, 2]	Direct Page Indirect Long Indexed Y
STA *sr*,S	83			x	2	4[1]	Stack Relative
STA (*sr*,S),Y	93			x	2	7[1]	Stack Relative Indirect Indexed Y

Notes:

1. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0

1.4.5 - STX

Store Index Register X to Memory

Stores the index register X into memory.

On all processors, the data written to memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data written is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
STX *addr*	8E	x	x	x	3	4[1]	Absolute
STX *dp*	86	x	x	x	2	3[1, 2]	Direct Page
STX *dp*,Y	96	x	x	x	2	4[1, 2]	Direct Page Indexed Y

Notes:

1. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0

1.4.6 - STY

Store Index Register X to Memory

Stores the index register Y into memory.

On all processors, the data written to memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data written is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
STY *addr*	8C	x	x	x	3	4[1]	Absolute
STY *dp*	84	x	x	x	2	3[1, 2]	Direct Page
STY *dp*,X	94	x	x	x	2	4[1, 2]	Direct Page Indexed X

Notes:

1. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0

1.4.7 - STZ

Store Zero to Memory

Stores zero to memory.

On all processors, the data written to memory is 8-bit. However, for 16-bit processors with the m flag is clear then the data written is 16-bit with the low-order 8-bits at the effective address and the high-order 8-bits at the effective address plus one.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
STZ *addr*	9C		x	x	3	4[1]	Absolute
STZ *dp*	64		x	x	2	3[1, 2]	Direct Page
STZ *addr*,X	9E		x	x	3	5[1]	Absolute Indexed X
STZ *dp*,X	74		x	x	2	4[1, 2]	Direct Page Indexed X

Notes:

1. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
2. 65816: Add 1 cycle if low byte of Direct Page register is not 0

1.4.8 - Transfer

Transfer data between registers

The transfer register set of instructions allows for data to be passed between different registers.

In all of these transfer instructions, the source register is unchanged.

For example, on the 6502 to save the X register on the stack you would need to use the following:

```
1 TXA      ; Transfer X into A
2 PHA      ; Push A to the stack
```

Note: On the 65C02 and later this is replaced by the PHX instruction which doesn't touch the accumulator.

TAX - Transfer Accumulator to Index Register X

Transfers the Accumulator to X. On the 8-bit processors the registers are all the same size, however on the 16-bit processors the registers can be of different sizes. The following table describes how the data is transferred when a size mismatch occurs:

Source Size	Dest Size	m	x	Action performed
8	8	All types		Value transferred is 8-bit
				Value transferred is 16-bit.
8	16	1	0	The 8-bit A becomes the low byte of the index register.
				The 8-bit hidden B register becomes the high byte of the index register.
16	8	0	1	The low byte of A is transferred to the index register
16	16	0	0	The full 16-bit A is transferred to the index register

TAY - Transfer Accumulator to Index Register Y

Transfers the Accumulator to Y. It follows the same rules as TAX.

TCD - Transfer 16-bit accumulator to Direct Page Register

TCD transfers the 16-bit accumulator C to the direct page register D, regardless of the accumulator/memory flag.

The C accumulator is used to indicate that 16-bits are transferred regardless of the m flag. If the A accumulator is 8-bit due to m=1 or in 6502 emulation mode then C = A as the low 8-bits and the hidden B accumulator as the high 8-bits.

TCS - Transfer Accumulator to Stack Pointer

TCS transfers the 16-bit accumulator C to the stack pointer S, regardless of the accumulator/memory flag. The C register is defined above for TCD. An alternate mnemonic for TCS is TAS.

Note: Unlike most transfer instructions, TCS does not affect any flags.

TDC - Transfer Direct Page Register tp 16-bit Accumulator

TDC transfers the Direct Page Register to the 16-bit accumulator C. The C register is defined above for TCD. An alternate mnemonic for TDC is TDA.

TSC - Transfer Stack Pointer to Accumulator

TSC transfers the stack pointer S to the 16-bit accumulator C, regardless of the accumulator/memory flag. The C register is defined above for TCD. An alternate mnemonic for TCS is TSA.

TSX - Transfer Stack Pointer to Index Register X

TSX transfers the stack pointer to X. The stack pointer is unchanged. On 8-bit processors only the low byte is transferred to X. On 16-bit processors (x=0) the full 16-bit value is tranferred to X.

TXA - Transfer Index Register X to Accumulator

TXA transfers X into the accumulator. On the 8-bit processors the registers are all the same size, however on the 16-bit processors the registers can be of different sizes. The following table describes how the data is transferred when a size mismatch occurs:

Source Size	Dest Size	m	x	Action performed
8	8	All types		Value transferred is 8-bit
				Value transferred is 16-bit.
8	16	1	0	The 8-bit index register becomes the low byte of the accumulator.
				The high-byte of the accumulator is set to 0.
				Value transferred is 8-bit.
16	8	0	1	The low 8-bits of the index register becomes the low byte of the accumulator.
				The high-byte of the hidden accumulator B is not affected by the transfer.
16	16	0	0	The full 16-bit index register is transferred to the accumulator

TXS - Transfer Index Register X to the Stack Pointer

TXS transfers X to the stack pointer to. The X is unchanged. On 8-bit processors only the low byte is transferred to S. On 16-bit processors (x=1) the low 8-bits of X is transferred to S. The high 8-bits of S are zeroed. On 16-bit processors (x=0) the full 16-bit value of X is transferred to S.

Note: Unlike most transfer instructions, TXS does not affect any flags.

TXY - Transfer index register X to Y

TXY transfers X to Y. X is unchanged. The registers are always the same size, so when 8-bit then that's what's transferred. When 16-bit (x=0) then 16-bits are transferred.

TYA - Transfer Index Register Y to Accumulator

TYA transfers Y into the accumulator. It follows the same rules as TXA above.

TYX - Transfer index register Y to X

TYX transfers Y to X. Y is unchanged. The registers are always the same size, so when 8-bit then that's what's transferred. When 16-bit (x=0) then 16-bits are transferred.

XBA - Exchange the B & A accumulators

On the 16-bit processors the 16-bit C accumulator is formed of two 8-bit accumulators, A for the low 8-bits and B for the upper 8-bits. XBA swaps the contents of the A & B registers.

In 8-bit memory mode, B is usually referred to as the hidden B accumulator, so the XBA instruction can be used to swap the accessible A with B, providing an in-processor scratch accumulator rather than pushing a value to the stack.

The flags are based on the value of the 8-bit A accumulator

Flags Affected

Flags	n	-	-	-	-	-	z	-

n Set if most significant bit of the transferred value is set

z Set if value transferred is zero

Instructions

Src	Dest	Syntax	Opcode *(hex)*	Available on: 6502	65C02	65816	# of bytes	# of cycles	Addressing Mode	Notes
A	X	TAX	AA	x	x	x	1	2	Implied	
A	Y	TAY	A8	x	x	x	1	2	Implied	
C	D	TCD	5B			x	1	2	Implied	
C	S	TCS	1B			x	1	2	Implied	Flags are unaffected
D	C	TDC	7B			x	1	2	Implied	
S	C	TSC	3B			x	1	2	Implied	
S	X	TSX	BA	x	x	x	1	2	Implied	
X	A	TXA	8A	x	x	x	1	2	Implied	
X	S	TXS	9A	x	x	x	1	2	Implied	Flags are unaffected
X	Y	TXY	9B			x	1	2	Implied	
Y	A	TYA	98	x	x	x	1	2	Implied	
Y	X	TYX	BB			x	1	2	Implied	
B	A	XBA	EB			x	1	2	Implied	Exchanges both registers, flags based on A post exchange

1.5 - Stack

Stack operations

1.5.1 - Pull

Stack pull operations

Flags Affected

Flags	n	-	-	-	-	-	z	-

n Set if most significant bit of the transferred value is set

z Set if value transferred is zero

Instructions

	Opcode	Available on:				# of	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
PLA	68	x	x	x	1	3[2]	Implied
PLB	AB			x	1	4	Implied
PLD	2B			x	1	5	Implied
PLP	28	x	x	x	1	4	Implied
PLX	FA	x	x	x	1	4[3]	Implied
PLY	7A	x	x	x	1	4[3]	Implied

Notes:

1. Add 1 cycle if low byte of Direct Page register is other than zero (DL<>0)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if x=0 (16-bit registers)

1.5.2 - Push

Stack push operations

Instructions

Syntax	Opcode *(hex)*	Available on: 6502	65C02	65816	# of bytes	# of cycles	Addressing Mode
PEA *addr*	F4			x	3	5	Stack Absolute
PEI (*dp*)	D4			x	2	6[1]	Stack Direct Page Indirect
PER *label*	62			x	3	6	Stack PC Relative Long
PHA	48	x	x	x	1	3[2]	Implied
PHB	8B			x	1	3	Implied
PHD	0B			x	1	4	Implied
PHK	4B			x	1	3	Implied
PHP	08	x	x	x	1	3	Implied
PHX	DA	x	x	x	1	3[3]	Implied
PHY	5A	x	x	x	1	3[3]	Implied

Notes:

1. Add 1 cycle if low byte of Direct Page register is other than zero (DL<>0)
2. 65816: Add 1 cycle if m=0 (16-bit memory/accumulator)
3. 65816: Add 1 cycle if x=0 (16-bit registers)

1.6 - Interrupts

Software & Hardware Interrupts

1.6.1 - BRK - Software Break

Perform a software break

BRK forces a software interrupt. It is unaffected by the i interrupt disable flag.

The BRK instruction is a single byte instruction. However, when it is invoked the Program Counter is incremented by 2. This allows for a one-byte signature value indicating which break caused the interrupt.

Even if the signature byte is not required, it must either be there or the RTI instruction which returns control to the caller must manually decrement the return address. As this can be tricky, most operating systems require BRK to take up 2 bytes in memory.

6502, 65C02 & Emulation Mode (e=1)

The program counter is incremented by two & pushed onto the stack. The status register (with b break flag set) is pushed onto the stack. The interrupt disable flag is then set, disabling interrupts. The program counter is loaded with the interrupt vector at &FFFE-&FFFF.

It's up to the interrupt handler pointed to by (&FFFE) to test the b flag to determine if the interrupt was from a software (BRK) rather than a hardware (IRQ) interrupt.

```
.handler PLA       ; copy status from stack
         PHA       ; but don't remove it else RTI will break

         AND #&10  ; check B flag
         BNE isBrk ; call break handler

.isIRQ             ; hardware handler here
         RTI       ; exit hardware handler

.isBrk             ; break handler here
         RTI       ; exit BRK handler
```

65802/65816 Native Mode (e=0)

The program bank register is pushed onto the stack. The program counter is incremented by two & pushed onto the stack. The status register is pushed onto the stack. The interrupt disable flag is then set, disabling interrupts. The program counter is loaded with the break vector at &00FFE6-&00FFE7.

Decimal Mode

On the 6502 the decimal d flag is not modified after a BRK is executed. On the 65C02 & 65816 the decimal d flag is reset to 0.

Flags Affected

Flags	-	-	-	**b**	**d**	**i**	-	-

- **b** Value of P register on the stack is set
- **d** On 65C02, 65816 in emulation mode (e=1) reset to 0 for binary arithmetic, unchanged on 6502
- **i** set to disable hardware IRQ interrupts

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
BRK	00	x	x	x	2[1]	7[2]	Stack Interrupt

Notes:

1. BRK is 1 byte but program counter is incremented by 2 allowing for an optional parameter
2. 65816: Add 1 cycle in 6502 emulation mode (e=1)

1.6.1.1 - BRK on the BBC Micro & Acorn Electron

Example of BRK on the BBC Micro or Acorn Electron

In the operating system for the BBC Micro (& Acorn Electron), the standard is to write BRK followed by an error number, then the error message ending with a 0

```
BRK           ; Software break
EQUB 0        ; Error code
EQUS "Silly"  ; This is a real error message in BBC BASIC, try: AUTO10,1000
EQUB 0        ; End of message marker
```

Service ROM's usually write error messages into RAM starting at &0100 and then do a JMP &0100 to run it. They do that as the handler is usually in a Language rom so they would be paged out if they ran BRK from their own ROM.

1.6.2 - COP - Co-Processor Enable

Perform a software interrupt with optional co-processor

COP causes a software interrupt similar to BRK but through a separate vector. Unlike BRK, it is possible for it to be trapped by an optional co-processor like a floating point processor or a graphics processor. It is unaffected by the i interrupt disable flag.

Like BRK, COP increments the Program Counter by 2. However assemblers require the second byte to be provided as part of the instruction.

Values &00-&7F are free for use by software handlers.

Values &80-&FF are reserved for hardware implementations.

65802/65816 in 6502 emulation mode (e=1)

The program counter is incremented by two & pushed onto the stack. The status register is pushed onto the stack. The interrupt disable flag is then set, disabling interrupts. The program counter is loaded with the interrupt vector at &FFF4-&FFF5. The d flag is reset to 0 after the instruction is executed.

65802/65816 in native mode (e=0)

The program bank register is pushed onto the stack. The program counter is incremented by two & pushed onto the stack. The status register is pushed onto the stack. The interrupt disable flag is then set, disabling interrupts. The program bank register is set to 0. The program counter is loaded with the break vector at &00FFE4-&00FFE5. The d flag is reset to 0 after the instruction is executed.

Flags Affected

Flags	-	-	-	-	**d**	**i**	-	-

d reset to 0

i set to disable hardware IRQ interrupts

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
COP *const*	02			x	2	7[1]	Stack Interrupt

Notes:

1. 65816: Add 1 cycle in 65816 native mode (e=0)

1.6.3 - RTI

Return from Interrupt

The RTI instruction is used at the end of an interrupt handler. It pulls both the status register and program counter from the stack. For 16bit processors running in native mode it also pulls the program bank register from the stack.

Unlike RTS, the address on the stack is the actual return address. (RTS expects it to be the address before the next instruction).

Instructions

	Opcode	Available on:			# of	# of	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
RTI	40	x	x	x	1	6[1]	Implied

Notes:

1. 65816: Add 1 cycle in 65816 native mode (e=0)

1.6.4 - WAI - Wait for Interrupt

Put the processor to sleep until a hardware interrupt occurs

WAI pulls the RDY pin low. Power consumption reduced to a minimum and RDY is kept low until an external hardware interrupt (NMI, IRQ, ABORT or RESET) is received.

When an interrupt is received

Interrupts enabled i=0

When a hardware interrupt is received, control is vectored though one of the hardware interrupt vectors. An RTI instruction in the invoked handler will return control back to the instruction immediately after the WAI.

Interrupts disabled i=1

If interrupts were disabled at the time WAI was invoked then when the interrupt is received then the relevant interrupt handler is not called and execution resumes immediately with the instruction after the WAI. This allows for processing to be synchronized with the interrupt.

The data bus

As WAI pulls RDY low it frees up the bus. If BE is also pulled low, the processor can be disconnected from the bus.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
WAI	CB			x	1	3[1]	Implied

Notes:

1. Uses 3 cycles to shut down the processor. Additional cycles required by interrupt to restart it

1.7 - Miscellaneous Instructions

Miscellaneous Instructions

1.7.1 - Block Move

Move (copy) memory block

The MVN & MVP instructions moves/copies a block of memory from one location to another.

The source, destination and length of the block are taken from the X, Y & C registers.

The source address is in X, the destination address is in Y.

The length of the block minus 1 is in the C double accumulator. So if you are moving 42 bytes then C should have 41.

The two bytes of the operand consists of the source bank in the first byte and the destination bank in the second.

Processor modes

These instructions should be run in 16-bit native mode. If the index registers are in 8-bit mode (x=1) or the processor is in 6502 emulation mode (e=1) then the blocks specified will be in zero page due to the high byte of the index registers will be 0.

Interrupts

If a block move instruction is interrupted, it may be resumed automatically when RTI is executed by the handler, as long as the registers are left intact. The address pushed to the stack when it is interrupted is the address of the block move instruction so it resumes where it left off. The byte currently being moved will complete first before the interrupt is serviced.

MVN

MVN copies a block from the start of the source block to the start of the destination block.

The source and destination addresses need to point to the first byte of each block to be moved.

When execution is complete, the C accumulator will be &FFFF X & Y will point to the byte after the end of the source & destination blocks respectively.

MVP - Block Move Previous

MVP copies a block from the end of the source block to the end of the destination block.

The source and destination addresses need to point to the last byte of each block to be moved.

When execution is complete, the C accumulator will be &FFFF X & Y will point to the byte before the start of the source & destination blocks respectively.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
MVN *srcbk*, *dstbk*	54			x	3	*[1]	Block Move
MVP *srcbk*, *dstbk*	44			x	3	*[1]	Block Move

Notes:

1. 7 cycles per byte moved

1.7.2 - XCE

Exchange Carry & Emulation Bits

This instruction is the only means to shift a 65802 or 65812 processor between 6502 emulation mode and full 16-bit native mode.

Switch to native 16-bit mode

To switch into native mode, clear the carry bit then invoke XCE

```
.goNative
    CLC      ; Clear Carry to indicate native mode
    XCE      ; Processor will be in 16-bit native mode once this completes
    RTS      ; Carry will now set if we were originally in emulation or clear if already native.
```

Once XCE has completed and the processor is in native mode, the following would have occurred.

- bit 5 of the flags stops being the b break flag. It's now the x mode select flag
- bit 6 is now the m memory mode flag (it's unused in 6502 emulation mode)
- Both x & m are set to 1

Switch to 6502 emulation mode

To switch into 6502 emulation mode, set the carry bit then invoke XCE

```
.goEmulation
    SEC      ; Set Carry to indicate native mode
    XCE      ; Processor will be in 16-bit native mode once this completes
    RTS      ; Carry will now set if we were already in emulation or clear if we were originally native.
```

Once XCE has completed and the processor is in 6502 emulation mode, the following would have occurred.

- The x & m flags are lost from the status register.
- bit 6 is unavailable as it's unused in 6502 emulation mode
- The accumulator is forced into 8-bit's, the high 8 bits are in the hidden B accumulator
- The index registers are forced into 8-bits. The high 8-bits are lost.
- The stack pointer is forced into page 1, losing the high byte of the address.

Flags Affected

Flags	-	-	**m**	-	-	-	-	**c**

m Set to 1 when switching to native mode, otherwise clear

c Takes emulations previous value

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
XCE	FB			x	1	2	Implied

1.7.3 - NOP

No Operation

A NOP takes no action and does not effect any registers except the program counter.

NOP's are usually used for timing loops as each NOP takes 2 cycles.

Flags Affected

None.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
NOP	EA	x	x	x	1	2	Implied

1.7.4 - Reserved

WDM Reserved for future expansion

> Do not use this instruction. It will break if/when a future processor is released with additional instructions.

The 65802 & 654816 processors use 255 out of the possible 256 8-bit opcodes. The remaining opcode is this one, labeled WDM which happens to be the initials of William D. Mensch who designed the processors.

To allow additional instructions to be added later this instruction act's as a prefix allowing an additonal 256 opcodes. This is a similar technique to the Z80 & 8080 processors which have 2-byte extension opcodes.

The actual number of bytes and cycles involved will be depended on those extensions, however the byte size will be a minimum of 2 bytes.

On the 65802 & 65816 this instruction will execute as a 2-byte NOP.

Instructions

	Opcode	**Available on:**			**# of**	**# of**	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
WDM	42			x	2[1]	?[1]	Implied

Notes:

1. byte & cycle count subject to change in future processors

1.7.5 - STP - Stop Processor

Stop the Processor until Reset

STP will stop the processors oscillator input, shutting down the processor until a reset occurs by pulling the $\overline{\text{RES}}$ pin low.

As power consumption is a function of frequency in CMOS circuits, stopping the clock cuts power to almost nothing.

Flags Affected

None.

Instructions

	Opcode	Available on:			# of	# of	
Syntax	***(hex)***	**6502**	**65C02**	**65816**	**bytes**	**cycles**	**Addressing Mode**
STP	DB			x	1	3[1]	Implied

Notes:

1. Uses 3 cycles to shut down the processor. Additional cycles required by reset to restart it

2 - reference

2.1 - Instruction List by name

ADC *addr*	6Dnnnn	CMP *addr*,Y	D9nnnn	LDA *addr*	ADnnnn	PHY	5A
ADC *addr*,X	7Dnnnn	CMP *long*	CFnnnnnn	LDA *addr*,X	BDnnnn	PLA	68
ADC *addr*,Y	79nnnn	CMP *long*,X	DFnnnnnn	LDA *addr*,Y	B9nnnn	PLB	AB
ADC *long*	6Fnnnnnn	CMP *dp*	C5nn	LDA *long*	AFnnnnnn	PLD	2B
ADC *long*,X	7Fnnnnnn	CMP (*dp*)	D2nn	LDA *long*,X	BFnnnnnn	PLP	28
ADC *dp*	65nn	CMP (*dp*,X)	C1nn	LDA *dp*	A5nn	PLX	FA
ADC (*dp*)	72nn	CMP (*dp*),Y	D1nn	LDA (*dp*)	B2nn	PLY	7A
ADC (*dp*,X)	61nn	CMP [*dp*]	C7nn	LDA (*dp*,X)	A1nn	REP *#const*	C2nn
ADC (*dp*),Y	71nn	CMP [*dp*],Y	D7nn	LDA (*dp*),Y	B1nn	ROL *addr*	2Ennnn
ADC [*dp*]	67nn	CMP *dp*,X	D5nn	LDA [*dp*]	A7nn	ROL *addr*,X	3Ennnn
ADC [*dp*],Y	77nn	CMP *#const*	C9nn	LDA [*dp*],Y	B7nn	ROL A	2A
ADC *dp*,X	75nn	CMP *sr*,S	C3nn	LDA *dp*,X	B5nn	ROL *dp*	26nn
ADC *#const*	69nn	CMP (*sr*,S),Y	D3nn	LDA *#const*	A9nn	ROL *dp*,X	36nn
ADC *sr*,S	63nn	COP *const*	02nn	LDA *sr*,S	A3nn	ROR *addr*	6Ennnn
ADC (*sr*,S),Y	73nn	CPX *addr*	ECnnnn	LDA (*sr*,S),Y	B3nn	ROR *addr*,X	7Ennnn
AND *addr*	2Dnnnn	CPX *dp*	E4nn	LDX *addr*	AEnnnn	ROR A	6A
AND *addr*,X	3Dnnnn	CPX *#const*	E0nn	LDX *addr*,X	BEnnnn	ROR *dp*	66nn
AND *addr*,Y	39nnnn	CPY *addr*	CCnnnn	LDX *dp*	A6nn	ROR *dp*,X	76nn
AND *long*	2Fnnnnnn	CPY *dp*	C4nn	LDX *dp*,X	B6nn	RTI	40
AND *long*,X	3Fnnnnnn	CPY *#const*	C0nn	LDX *#const*	A2nn	RTL	6B
AND *dp*	25nn	DEC *addr*	CEnnnn	LDY *addr*	ACnnnn	RTS	60
AND (*dp*)	32nn	DEC *addr*,X	DEnnnn	LDY *addr*,X	BCnnnn	SBC *addr*	EDnnnn
AND (*dp*,X)	21nn	DEC A	3A	LDY *dp*	A4nn	SBC *addr*,X	FDnnnn
AND (*dp*),Y	31nn	DEC *dp*	C6nn	LDY *dp*,X	B4nn	SBC *addr*,Y	F9nnnn
AND [*dp*]	27nn	DEC *dp*,X	D6nn	LDY *#const*	A0nn	SBC *long*	EFnnnnnn
AND [*dp*],Y	37nn	DEX	CA	LSR *addr*	4Ennnn	SBC *long*,X	FFnnnnnn
AND *dp*,X	35nn	DEY	88	LSR *addr*,X	5Ennnn	SBC *dp*	E5nn
AND *#const*	29nn	EOR *addr*	4Dnnnn	LSR A	4A	SBC (*dp*)	F2nn
AND *sr*,S	23nn	EOR *addr*,X	5Dnnnn	LSR *dp*	46nn	SBC (*dp*,X)	E1nn
AND (*sr*,S),Y	33nn	EOR *addr*,Y	59nnnn	LSR *dp*,X	56nn	SBC (*dp*),Y	F1nn
ASL *addr*	0Ennnn	EOR *long*	4Fnnnnnn	MVN *srcbk*, *dstbk*	54nnnn	SBC [*dp*]	E7nn
ASL *addr*,X	1Ennnn	EOR *long*,X	5Fnnnnnn	MVP *srcbk*, *dstbk*	44nnnn	SBC [*dp*],Y	F7nn
ASL A	0A	EOR *dp*	45nn	NOP	EA	SBC *dp*,X	F5nn
ASL *dp*	06nn	EOR (*dp*)	52nn	ORA *addr*	0Dnnnn	SBC *#const*	E9nn
ASL *dp*,X	16nn	EOR (*dp*,X)	41nn	ORA *addr*,X	1Dnnnn	SBC *sr*,S	E3nn
BCC *nearlabel*	90nn	EOR (*dp*),Y	51nn	ORA *addr*,Y	19nnnn	SBC (*sr*,S),Y	F3nn
BCS *nearlabel*	B0nn	EOR [*dp*]	47nn	ORA *long*	0Fnnnnnn	SEC	38
BEQ *nearlabel*	F0nn	EOR [*dp*],Y	57nn	ORA *long*,X	1Fnnnnnn	SED	F8
BIT *addr*	2Cnnnn	EOR *dp*,X	55nn	ORA *dp*	05nn	SEI	78
BIT *addr*,X	3Cnnnn	EOR *#const*	49nn	ORA (*dp*)	12nn	SEP *#const*	E2nn
BIT *dp*	24nn	EOR *sr*,S	43nn	ORA (*dp*,X)	01nn	STA *addr*	8Dnnnn
BIT *dp*,X	34nn	EOR (*sr*,S),Y	53nn	ORA (*dp*),Y	11nn	STA *addr*,X	9Dnnnn
BIT *#const*	89nn	INC *addr*	EEnnnn	ORA [*dp*]	07nn	STA *addr*,Y	99nnnn
BMI *nearlabel*	30nn	INC *addr*,X	FEnnnn	ORA [*dp*],Y	17nn	STA *long*	8Fnnnnnn
BNE *nearlabel*	D0nn	INC A	1A	ORA *dp*,X	15nn	STA *long*,X	9Fnnnnnn
BPL *nearlabel*	10nn	INC *dp*	E6nn	ORA *#const*	09nn	STA *dp*	85nn
BRA *nearlabel*	80nn	INC *dp*,X	F6nn	ORA *sr*,S	03nn	STA (*dp*)	92nn
BRK	00nn	INX	E8	ORA (*sr*,S),Y	13nn	STA (*dp*,X)	81nn
BRL *label*	82nnnn	INY	C8	PEA *addr*	F4nnnn	STA (*dp*),Y	91nn
BVC *nearlabel*	50nn	JMP *addr*	4Cnnnn	PEI (*dp*)	D4nn	STA [*dp*]	87nn
BVS *nearlabel*	70nn	JMP (*addr*)	6Cnnnn	PER *label*	62nnnn	STA [*dp*],Y	97nn
CLC	18	JMP (*addr*,X)	7Cnnnn	PHA	48	STA *dp*,X	95nn
CLD	D8	JMP [*addr*]	DCnnnn	PHB	8B	STA *sr*,S	83nn
CLI	58	JMP *long*	5Cnnnnnn	PHD	0B	STA (*sr*,S),Y	93nn
CLV	B8	JSL *long*	22nnnnnn	PHK	4B	STP	DB
CMP *addr*	CDnnnn	JSR *addr*	20nnnn	PHP	08	STX *addr*	8Ennnn
CMP *addr*,X	DDnnnn	JSR (*addr*,X)	FCnnnn	PHX	DA	STX *dp*	86nn

STX *dp*,Y	96nn	STZ *dp*,X	74nn	TRB *dp*	14nn	TXY	9B
STY *addr*	8Cnnnn	TAX	AA	TSB *addr*	0Cnnnn	TYA	98
STY *dp*	84nn	TAY	A8	TSB *dp*	04nn	TYX	BB
STY *dp*,X	94nn	TCD	5B	TSC	3B	WAI	CB
STZ *addr*	9Cnnnn	TCS	1B	TSX	BA	WDM	42nn
STZ *addr*,X	9Ennnn	TDC	7B	TXA	8A	XBA	EB
STZ *dp*	64nn	TRB *addr*	1Cnnnn	TXS	9A	XCE	FB

2.2 - Instruction List by opcode

BRK	00nn	TSC	3B	ROR *dp*,X	76nn	LDA (*dp*),Y	B1nn
ORA (*dp*,X)	01nn	BIT *addr*,X	3Cnnnn	ADC [*dp*],Y	77nn	LDA (*dp*)	B2nn
COP *const*	02nn	AND *addr*,X	3Dnnnn	SEI	78	LDA (*sr*,S),Y	B3nn
ORA *sr*,S	03nn	ROL *addr*,X	3Ennnn	ADC *addr*,Y	79nnnn	LDY *dp*,X	B4nn
TSB *dp*	04nn	AND *long*,X	3Fnnnnnn	PLY	7A	LDA *dp*,X	B5nn
ORA *dp*	05nn	RTI	40	TDC	7B	LDX *dp*,X	B6nn
ASL *dp*	06nn	EOR (*dp*,X)	41nn	JMP (*addr*,X)	7Cnnnn	LDA [*dp*],Y	B7nn
ORA [*dp*]	07nn	WDM	42nn	ADC *addr*,X	7Dnnnn	CLV	B8
PHP	08	EOR *sr*,S	43nn	ROR *addr*,X	7Ennnn	LDA *addr*,Y	B9nnnn
ORA *#const*	09nn	MVP *srcbk*, *dstbk*	44nnnn	ADC *long*,X	7Fnnnnnn	TSX	BA
ASL A	0A	EOR *dp*	45nn	BRA *nearlabel*	80nn	TYX	BB
PHD	0B	LSR *dp*	46nn	STA (*dp*,X)	81nn	LDY *addr*,X	BCnnnn
TSB *addr*	0Cnnnn	EOR [*dp*]	47nn	BRL *label*	82nnnn	LDA *addr*,X	BDnnnn
ORA *addr*	0Dnnnn	PHA	48	STA *sr*,S	83nn	LDX *addr*,X	BEnnnn
ASL *addr*	0Ennnn	EOR *#const*	49nn	STY *dp*	84nn	LDA *long*,X	BFnnnnnn
ORA *long*	0Fnnnnnn	LSR A	4A	STA *dp*	85nn	CPY *#const*	C0nn
BPL *nearlabel*	10nn	PHK	4B	STX *dp*	86nn	CMP (*dp*,X)	C1nn
ORA (*dp*),Y	11nn	JMP *addr*	4Cnnnn	STA [*dp*]	87nn	REP *#const*	C2nn
ORA (*dp*)	12nn	EOR *addr*	4Dnnnn	DEY	88	CMP *sr*,S	C3nn
ORA (*sr*,S),Y	13nn	LSR *addr*	4Ennnn	BIT *#const*	89nn	CPY *dp*	C4nn
TRB *dp*	14nn	EOR *long*	4Fnnnnnn	TXA	8A	CMP *dp*	C5nn
ORA *dp*,X	15nn	BVC *nearlabel*	50nn	PHB	8B	DEC *dp*	C6nn
ASL *dp*,X	16nn	EOR (*dp*),Y	51nn	STY *addr*	8Cnnnn	CMP [*dp*]	C7nn
ORA [*dp*],Y	17nn	EOR (*dp*)	52nn	STA *addr*	8Dnnnn	INY	C8
CLC	18	EOR (*sr*,S),Y	53nn	STX *addr*	8Ennnn	CMP *#const*	C9nn
ORA *addr*,Y	19nnnn	MVN *srcbk*, *dstbk*	54nnnn	STA *long*	8Fnnnnnn	DEX	CA
INC A	1A	EOR *dp*,X	55nn	BCC *nearlabel*	90nn	WAI	CB
TCS	1B	LSR *dp*,X	56nn	STA (*dp*),Y	91nn	CPY *addr*	CCnnnn
TRB *addr*	1Cnnnn	EOR [*dp*],Y	57nn	STA (*dp*)	92nn	CMP *addr*	CDnnnn
ORA *addr*,X	1Dnnnn	CLI	58	STA (*sr*,S),Y	93nn	DEC *addr*	CEnnnn
ASL *addr*,X	1Ennnn	EOR *addr*,Y	59nnnn	STY *dp*,X	94nn	CMP *long*	CFnnnnnn
ORA *long*,X	1Fnnnnnn	PHY	5A	STA *dp*,X	95nn	BNE *nearlabel*	D0nn
JSR *addr*	20nnnn	TCD	5B	STX *dp*,Y	96nn	CMP (*dp*),Y	D1nn
AND (*dp*,X)	21nn	JMP *long*	5Cnnnnnn	STA [*dp*],Y	97nn	CMP (*dp*)	D2nn
JSL *long*	22nnnnnn	EOR *addr*,X	5Dnnnn	TYA	98	CMP (*sr*,S),Y	D3nn
AND *sr*,S	23nn	LSR *addr*,X	5Ennnn	STA *addr*,Y	99nnnn	PEI (*dp*)	D4nn
BIT *dp*	24nn	EOR *long*,X	5Fnnnnnn	TXS	9A	CMP *dp*,X	D5nn
AND *dp*	25nn	RTS	60	TXY	9B	DEC *dp*,X	D6nn
ROL *dp*	26nn	ADC (*dp*,X)	61nn	STZ *addr*	9Cnnnn	CMP [*dp*],Y	D7nn
AND [*dp*]	27nn	PER *label*	62nnnn	STA *addr*,X	9Dnnnn	CLD	D8
PLP	28	ADC *sr*,S	63nn	STZ *addr*,X	9Ennnn	CMP *addr*,Y	D9nnnn
AND *#const*	29nn	STZ *dp*	64nn	STA *long*,X	9Fnnnnnn	PHX	DA
ROL A	2A	ADC *dp*	65nn	LDY *#const*	A0nn	STP	DB
PLD	2B	ROR *dp*	66nn	LDA (*dp*,X)	A1nn	JMP [*addr*]	DCnnnn
BIT *addr*	2Cnnnn	ADC [*dp*]	67nn	LDX *#const*	A2nn	CMP *addr*,X	DDnnnn
AND *addr*	2Dnnnn	PLA	68	LDA *sr*,S	A3nn	DEC *addr*,X	DEnnnn
ROL *addr*	2Ennnn	ADC *#const*	69nn	LDY *dp*	A4nn	CMP *long*,X	DFnnnnnn
AND *long*	2Fnnnnnn	ROR A	6A	LDA *dp*	A5nn	CPX *#const*	E0nn
BMI *nearlabel*	30nn	RTL	6B	LDX *dp*	A6nn	SBC (*dp*,X)	E1nn
AND (*dp*),Y	31nn	JMP (*addr*)	6Cnnnn	LDA [*dp*]	A7nn	SEP *#const*	E2nn
AND (*dp*)	32nn	ADC *addr*	6Dnnnn	TAY	A8	SBC *sr*,S	E3nn
AND (*sr*,S),Y	33nn	ROR *addr*	6Ennnn	LDA *#const*	A9nn	CPX *dp*	E4nn
BIT *dp*,X	34nn	ADC *long*	6Fnnnnnn	TAX	AA	SBC *dp*	E5nn
AND *dp*,X	35nn	BVS *nearlabel*	70nn	PLB	AB	INC *dp*	E6nn
ROL *dp*,X	36nn	ADC (*dp*),Y	71nn	LDY *addr*	ACnnnn	SBC [*dp*]	E7nn
AND [*dp*],Y	37nn	ADC (*dp*)	72nn	LDA *addr*	ADnnnn	INX	E8
SEC	38	ADC (*sr*,S),Y	73nn	LDX *addr*	AEnnnn	SBC *#const*	E9nn
AND *addr*,Y	39nnnn	STZ *dp*,X	74nn	LDA *long*	AFnnnnnn	NOP	EA
DEC A	3A	ADC *dp*,X	75nn	BCS *nearlabel*	B0nn	XBA	EB

CPX *addr*	ECnnnn	SBC (*dp*),Y	F1nn	INC *dp*,X	F6nn	XCE	FB
SBC *addr*	EDnnnn	SBC (*dp*)	F2nn	SBC [*dp*],Y	F7nn	JSR (*addr*,X)	FCnnnn
INC *addr*	EEnnnn	SBC (*sr*,S),Y	F3nn	SED	F8	SBC *addr*,X	FDnnnn
SBC *long*	EFnnnnnn	PEA *addr*	F4nnnn	SBC *addr*,Y	F9nnnn	INC *addr*,X	FEnnnn
BEQ *nearlabel*	F0nn	SBC *dp*,X	F5nn	PLX	FA	SBC *long*,X	FFnnnnnn

2.3 - Opcode Matrix

Instructions shown in an Opcode Matrix

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK 2 7 00nn	ORA (*dp*,X) 2 6 01nn	COP *const* 2 7 02nn	ORA *sr*,S 2 4 03nn	TSB *dp* 2 5 04nn	ORA *dp* 2 3 05nn	ASL *dp* 2 5 06nn	ORA [*dp*] 2 6 07nn	PHP 1 3 08	ORA *#const* 2 2 09nn	ASL A 1 2 0A	PHD 1 4 0B	TSB *addr* 3 6 0Cnnnn	ORA *addr* 3 4 0Dnnnn	ASL *addr* 3 6 0Ennnn	ORA *long* 4 5 0Fnnnnnn
1	BPL *nearlabel* 2 2 10nn	ORA (*dp*),Y 2 5 11nn	ORA (*dp*) 2 5 12nn	ORA (*sr*,S),Y 2 7 13nn	TRB *dp* 2 5 14nn	ORA *dp*,X 2 4 15nn	ASL *dp*,X 2 6 16nn	ORA [*dp*],Y 2 6 17nn	CLC 1 2 18	ORA *addr*,Y 3 4 19nnnn	INC A 1 2 1A	TCS 1 2 1B	TRB *addr* 3 6 1Cnnnn	ORA *addr*,X 3 4 1Dnnnn	ASL *addr*,X 3 7 1Ennnn	ORA *long*,X 4 5 1Fnnnnnn
2	JSR *addr* 3 6 20nnnn	AND (*dp*,X) 2 6 21nn	JSL *long* 4 8 22nnnnnn	AND *sr*,S 2 4 23nn	BIT *dp* 2 5 24nn	AND *dp* 2 3 25nn	ROL *dp* 2 5 26nn	AND [*dp*] 2 6 27nn	PLP 1 4 28	AND *#const* 2 2 29nn	ROL A 1 2 2A	PLD 1 5 2B	BIT *addr* 3 4 2Cnnnn	AND *addr* 3 4 2Dnnnn	ROL *addr* 3 6 2Ennnn	AND *long* 4 5 2Fnnnnnn
3	BMI *nearlabel* 2 2 30nn	AND (*dp*),Y 2 5 31nn	AND (*dp*) 2 5 32nn	AND (*sr*,S),Y 2 7 33nn	BIT *dp*,X 2 4 34nn	AND *dp*,X 2 4 35nn	ROL *dp*,X 2 6 36nn	AND [*dp*],Y 2 6 37nn	SEC 1 2 38	AND *addr*,Y 3 4 39nnnn	DEC A 1 2 3A	TSC 1 2 3B	BIT *addr*,X 3 4 3Cnnnn	AND *addr*,X 3 4 3Dnnnn	ROL *addr*,X 3 7 3Ennnn	AND *long*,X 4 5 3Fnnnnnn
4	RTI 1 6 40	EOR (*dp*,X) 2 6 41nn	WDM 2 0 42nn	EOR *sr*,S 2 4 43nn	MVP *srcbk*, *dstbk* 3 0 44nnnn	EOR *dp* 2 3 45nn	LSR *dp* 2 5 46nn	EOR [*dp*] 2 6 47nn	PHA 1 3 48	EOR *#const* 2 2 49nn	LSR A 1 2 4A	PHK 1 3 4B	JMP *addr* 3 3 4Cnnnn	EOR *addr* 3 4 4Dnnnn	LSR *addr* 3 6 4Ennnn	EOR *long* 4 5 4Fnnnnnn
5	BVC *nearlabel* 2 2 50nn	EOR (*dp*),Y 2 5 51nn	EOR (*dp*) 2 5 52nn	EOR (*sr*,S),Y 2 7 53nn	MVN *srcbk*, *dstbk* 3 0 54nnnn	EOR *dp*,X 2 4 55nn	LSR *dp*,X 2 6 56nn	EOR [*dp*],Y 2 6 57nn	CLI 1 2 58	EOR *addr*,Y 3 4 59nnnn	PHY 1 3 5A	TCD 1 2 5B	JMP *long* 4 4 5Cnnnnnn	EOR *addr*,X 3 4 5Dnnnn	LSR *addr*,X 3 7 5Ennnn	EOR *long*,X 4 5 5Fnnnnnn
6	RTS 1 6 60	ADC (*dp*,X) 2 6 61nn	PER *label* 3 6 62nnnn	ADC *sr*,S 2 4 63nn	STZ *dp* 2 3 64nn	ADC *dp* 2 3 65nn	ROR *dp* 2 5 66nn	ADC [*dp*] 2 6 67nn	PLA 1 3 68	ADC *#const* 2 2 69nn	ROR A 1 2 6A	RTL 1 6 6B	JMP (*addr*) 3 5 6Cnnnn	ADC *addr* 3 4 6Dnnnn	ROR *addr* 3 6 6Ennnn	ADC *long* 4 5 6Fnnnnnn
7	BVS *nearlabel* 2 2 70nn	ADC (*dp*),Y 2 5 71nn	ADC (*dp*) 2 5 72nn	ADC (*sr*,S),Y 2 7 73nn	STZ *dp*,X 2 4 74nn	ADC *dp*,X 2 4 75nn	ROR *dp*,X 2 6 76nn	ADC [*dp*],Y 2 6 77nn	SEI 1 2 78	ADC *addr*,Y 3 4 79nnnn	PLY 1 4 7A	TDC 1 2 7B	JMP (*addr*,X) 3 6 7Cnnnn	ADC *addr*,X 3 4 7Dnnnn	ROR *addr*,X 3 7 7Ennnn	ADC *long*,X 4 5 7Fnnnnnn
8	BRA *nearlabel* 2 3 80nn	STA (*dp*,X) 2 6 81nn	BRL *label* 3 4 82nnnn	STA *sr*,S 2 4 83nn	STY *dp* 2 3 84nn	STA *dp* 2 3 85nn	STX *dp* 2 3 86nn	STA [*dp*] 2 6 87nn	DEY 1 2 88	BIT *#const* 2 2 89nn	TXA 1 2 8A	PHB 1 3 8B	STY *addr* 3 4 8Cnnnn	STA *addr* 3 4 8Dnnnn	STX *addr* 3 4 8Ennnn	STA *long* 4 5 8Fnnnnnn
9	BCC *nearlabel* 2 2 90nn	STA (*dp*),Y 2 5 91nn	STA (*dp*) 2 5 92nn	STA (*sr*,S),Y 2 7 93nn	STY *dp*,X 2 4 94nn	STA *dp*,X 2 4 95nn	STX *dp*,Y 2 4 96nn	STA [*dp*],Y 2 6 97nn	TYA 1 2 98	STA *addr*,Y 3 4 99nnnn	TXS 1 2 9A	TXY 1 2 9B	STZ *addr* 3 4 9Cnnnn	STA *addr*,X 3 4 9Dnnnn	STZ *addr*,X 3 5 9Ennnn	STA *long*,X 4 5 9Fnnnnnn
A	LDY *#const* 2 2 A0nn	LDA (*dp*,X) 2 6 A1nn	LDX *#const* 2 2 A2nn	LDA *sr*,S 2 4 A3nn	LDY *dp* 2 3 A4nn	LDA *dp* 2 3 A5nn	LDX *dp* 2 3 A6nn	LDA [*dp*] 2 6 A7nn	TAY 1 2 A8	LDA *#const* 2 2 A9nn	TAX 1 2 AA	PLB 1 4 AB	LDY *addr* 3 4 ACnnnn	LDA *addr* 3 4 ADnnnn	LDX *addr* 3 4 AEnnnn	LDA *long* 4 5 AFnnnnnn
B	BCS *nearlabel* 2 2 B0nn	LDA (*dp*),Y 2 5 B1nn	LDA (*dp*) 2 5 B2nn	LDA (*sr*,S),Y 2 7 B3nn	LDY *dp*,X 2 4 B4nn	LDA *dp*,X 2 4 B5nn	LDX *dp*,X 2 4 B6nn	LDA [*dp*],Y 2 6 B7nn	CLV 1 2 B8	LDA *addr*,Y 3 4 B9nnnn	TSX 1 2 BA	TYX 1 2 BB	LDY *addr*,X 3 4 BCnnnn	LDA *addr*,X 3 4 BDnnnn	LDX *addr*,X 3 4 BEnnnn	LDA *long*,X 4 5 BFnnnnnn
C	CPY *#const* 2 2 C0nn	CMP (*dp*,X) 2 6 C1nn	REP *#const* 2 3 C2nn	CMP *sr*,S 2 4 C3nn	CPY *dp* 2 3 C4nn	CMP *dp* 2 3 C5nn	DEC *dp* 2 5 C6nn	CMP [*dp*] 2 6 C7nn	INY 1 2 C8	CMP *#const* 2 2 C9nn	DEX 1 2 CA	WAI 1 3 CB	CPY *addr* 3 4 CCnnnn	CMP *addr* 3 4 CDnnnn	DEC *addr* 3 6 CEnnnn	CMP *long* 4 5 CFnnnnnn
D	BNE *nearlabel* 2 2 D0nn	CMP (*dp*),Y 2 5 D1nn	CMP (*dp*) 2 5 D2nn	CMP (*sr*,S),Y 2 7 D3nn	PEI (*dp*) 2 6 D4nn	CMP *dp*,X 2 4 D5nn	DEC *dp*,X 2 6 D6nn	CMP [*dp*],Y 2 6 D7nn	CLD 1 2 D8	CMP *addr*,Y 3 4 D9nnnn	PHX 1 3 DA	STP 1 3 DB	JMP [*addr*] 3 6 DCnnnn	CMP *addr*,X 3 4 DDnnnn	DEC *addr*,X 3 7 DEnnnn	CMP *long*,X 4 5 DFnnnnnn
E	CPX *#const* 2 2 E0nn	SBC (*dp*,X) 2 6 E1nn	SEP *#const* 2 3 E2nn	SBC *sr*,S 2 4 E3nn	CPX *dp* 2 3 E4nn	SBC *dp* 2 3 E5nn	INC *dp* 2 5 E6nn	SBC [*dp*] 2 6 E7nn	INX 1 2 E8	SBC *#const* 2 2 E9nn	NOP 1 2 EA	XBA 1 2 EB	CPX *addr* 3 4 ECnnnn	SBC *addr* 3 4 EDnnnn	INC *addr* 3 6 EEnnnn	SBC *long* 4 5 EFnnnnnn
F	BEQ *nearlabel* 2 2 F0nn	SBC (*dp*),Y 2 5 F1nn	SBC (*dp*) 2 5 F2nn	SBC (*sr*,S),Y 2 7 F3nn	PEA *addr* 3 5 F4nnnn	SBC *dp*,X 2 4 F5nn	INC *dp*,X 2 6 F6nn	SBC [*dp*],Y 2 6 F7nn	SED 1 2 F8	SBC *addr*,Y 3 4 F9nnnn	PLX 1 4 FA	XCE 1 2 FB	JSR (*addr*,X) 3 8 FCnnnn	SBC *addr*,X 3 4 FDnnnn	INC *addr*,X 3 7 FEnnnn	SBC *long*,X 4 5 FFnnnnnn

Opcode Matrix Legend

Instruction Size bytes Cycle count Opcode hex	Register	Memory	Implicit	Math	Logic	Flow	Interrupt	Special	Extension