6502 Instruction Set

HI	*LO-NIBBLE*															
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	BRK impl	ORA X,ind	JAM	SLO X,ind	NOP zpg	ORA zpg	ASL zpg	SLO zpg	PHP impl	ORA #	ASL A	ANC #	NOP abs	ORA abs	ASL abs	SLO abs
1-	BPL rel	ORA ind,Y	JAM	SLO ind,Y	NOP zpg,X	ORA zpg,X	ASL zpg,X	SLO zpg,X	CLC impl	ORA abs,Y	NOP impl	SLO abs,Y	NOP abs,X	ORA abs,X	ASL abs,X	SLO abs,X
2-	JSR abs	AND X,ind	JAM	RLA X,ind	BIT zpg	AND zpg	ROL zpg	RLA zpg	PLP impl	AND #	ROL A	ANC #	BIT abs	AND abs	ROL abs	RLA abs
3-	BMI rel	AND ind,Y	JAM	RLA ind,Y	NOP zpg,X	AND zpg,X	ROL zpg,X	RLA zpg,X	SEC impl	AND abs,Y	NOP impl	RLA abs,Y	NOP abs,X	AND abs,X	ROL abs,X	RLA abs,X
4-	RTI impl	EOR X,ind	JAM	SRE X,ind	NOP zpg	EOR zpg	LSR zpg	SRE zpg	PHA impl	EOR #	LSR A	ALR #	JMP abs	EOR abs	LSR abs	SRE abs
5-	BVC rel	EOR ind,Y	JAM	SRE ind,Y	NOP zpg,X	EOR zpg,X	LSR zpg,X	SRE zpg,X	CLI impl	EOR abs,Y	NOP impl	SRE abs,Y	NOP abs,X	EOR abs,X	LSR abs,X	SRE abs,X
6-	RTS impl	ADC X,ind	JAM	RRA X,ind	NOP zpg	ADC zpg	ROR zpg	RRA zpg	PLA impl	ADC #	ROR A	ARR #	JMP ind	ADC abs	ROR abs	RRA abs
7-	BVS rel	ADC ind,Y	JAM	RRA ind,Y	NOP zpg,X	ADC zpg,X	ROR zpg,X	RRA zpg,X	SEI impl	ADC abs,Y	NOP impl	RRA abs,Y	NOP abs,X	ADC abs,X	ROR abs,X	RRA abs,X
8-	NOP #	STA X,ind	NOP #	SAX X,ind	STY zpg	STA zpg	STX zpg	SAX zpg	DEY impl	NOP #	TXA impl	ANE #	STY abs	STA abs	STX abs	SAX abs
9-	BCC rel	STA ind,Y	JAM	SHA ind,Y	STY zpg,X	STA zpg,X	STX zpg,Y	SAX zpg,Y	TYA impl	STA abs,Y	TXS impl	TAS abs,Y	SHY abs,X	STA abs,X	SHX abs,Y	SHA abs,Y
A-	LDY #	LDA X,ind	LDX #	LAX X,ind	LDY zpg	LDA zpg	LDX zpg	LAX zpg	TAY impl	LDA #	TAX impl	LXA #	LDY abs	LDA abs	LDX abs	LAX abs
B-	BCS rel	LDA ind,Y	JAM	LAX ind,Y	LDY zpg,X	LDA zpg,X	LDX zpg,Y	LAX zpg,Y	CLV impl	LDA abs,Y	TSX impl	LAS abs,Y	LDY abs,X	LDA abs,X	LDX abs,Y	LAX abs,Y
C-	CPY #	CMP X,ind	NOP #	DCP X,ind	CPY zpg	CMP zpg	DEC zpg	DCP zpg	INY impl	CMP #	DEX impl	SBX #	CPY abs	CMP abs	DEC abs	DCP abs
D-	BNE rel	CMP ind,Y	JAM	DCP ind,Y	NOP zpg,X	CMP zpg,X	DEC zpg,X	DCP zpg,X	CLD impl	CMP abs,Y	NOP impl	DCP abs,Y	NOP abs,X	CMP abs,X	DEC abs,X	DCP abs,X
E-	CPX #	SBC X,ind	NOP #	ISC X,ind	CPX zpg	SBC zpg	INC zpg	ISC zpg	INX impl	SBC #	NOP impl	USBC #	CPX abs	SBC abs	INC abs	ISC abs
F-	BEQ rel	SBC ind,Y	JAM	ISC ind,Y	NOP zpg,X	SBC zpg,X	INC zpg,X	ISC zpg,X	SED impl	SBC abs,Y	NOP impl	ISC abs,Y	NOP abs,X	SBC abs,X	INC abs,X	ISC abs,X

☑ show illegal opcodes

Description

Address Modes

A	.... Accumulator	OPC A	*operand is AC (implied single byte instruction)*
abs	.... absolute	OPC $LLHH	*operand is address $HHLL **
abs,X	.... absolute, X-indexed	OPC $LLHH,X	*operand is address; effective address is address incremented by X with carry ***
abs,Y	.... absolute, Y-indexed	OPC $LLHH,Y	*operand is address; effective address is address incremented by Y with carry ***
#	.... immediate	OPC #$BB	*operand is byte BB*
impl	.... implied	OPC	*operand implied*
ind	.... indirect	OPC ($LLHH)	*operand is address; effective address is contents of word at address: C.w($HHLL)*
X,ind	.... X-indexed, indirect	OPC ($LL,X)	*operand is zeropage address; effective address is word in (LL + X, LL + X + 1), inc. without carry: C.w($00LL + X)*
ind,Y	.... indirect, Y-indexed	OPC ($LL),Y	*operand is zeropage address; effective address is word in (LL, LL + 1) incremented by Y with carry: C.w($00LL) + Y*
rel	.... relative	OPC $BB	*branch target is PC + signed offset BB ****
zpg	.... zeropage	OPC $LL	*operand is zeropage address (hi-byte is zero, address = $00LL)*
zpg,X	.... zeropage, X-indexed	OPC $LL,X	*operand is zeropage address; effective address is address incremented by X without carry ***
zpg,Y	.... zeropage, Y-indexed	OPC $LL,Y	*operand is zeropage address; effective address is address incremented by Y without carry ***

* 16-bit address words are little endian, lo(w)-byte first, followed by the hi(gh)-byte.
(An assembler will use a human readable, big-endian notation as in $HHLL.)

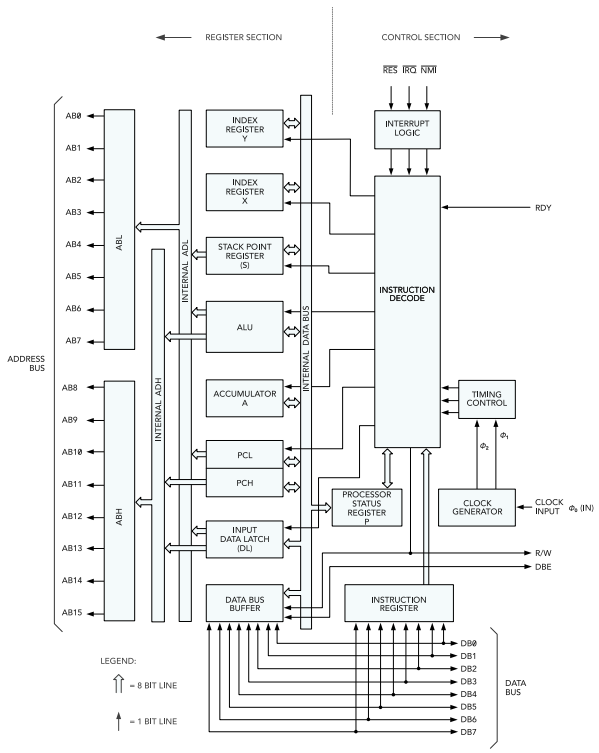
** The available 16-bit address space is conceived as consisting of pages of 256 bytes each, with address hi-bytes represententing the page index. An increment with carry may affect the hi-byte and may thus result in a crossing of page boundaries, adding an extra cycle to the execution. Increments without carry do not affect the hi-byte of an address and no page transitions do occur. Generally, increments of 16-bit addresses include a carry, increments of zeropage addresses don't. Notably this is not related in any way to the state of the carry bit of the accumulator.

*** Branch offsets are signed 8-bit values, -128 ... +127, negative offsets in two's complement. Page transitions may occur and add an extra cycle to the exucution.

Instructions by Name

ADC .... add with carry
AND .... and (with accumulator)
ASL .... arithmetic shift left
BCC .... branch on carry clear
BCS .... branch on carry set
BEQ .... branch on equal (zero set)
BIT .... bit test
BMI .... branch on minus (negative set)
BNE .... branch on not equal (zero clear)
BPL .... branch on plus (negative clear)
BRK .... break / interrupt
BVC .... branch on overflow clear
BVS .... branch on overflow set
CLC .... clear carry
CLD .... clear decimal
CLI .... clear interrupt disable
CLV .... clear overflow
CMP .... compare (with accumulator)
CPX .... compare with X
CPY .... compare with Y
DEC .... decrement
DEX .... decrement X
DEY .... decrement Y
EOR .... exclusive or (with accumulator)
INC .... increment
INX .... increment X
INY .... increment Y
JMP .... jump
JSR .... jump subroutine
LDA .... load accumulator
LDX .... load X
LDY .... load Y
LSR .... logical shift right
NOP .... no operation

ORA .... or with accumulator
PHA .... push accumulator
PHP .... push processor status (SR)
PLA .... pull accumulator
PLP .... pull processor status (SR)
ROL .... rotate left
ROR .... rotate right
RTI .... return from interrupt
RTS .... return from subroutine
SBC .... subtract with carry
SEC .... set carry
SED .... set decimal
SEI .... set interrupt disable
STA .... store accumulator
STX .... store X
STY .... store Y
TAX .... transfer accumulator to X
TAY .... transfer accumulator to Y
TSX .... transfer stack pointer to X
TXA .... transfer X to accumulator
TXS .... transfer X to stack pointer
TYA .... transfer Y to accumulator



Block diagram of the NMOS 6502 CPU.
After (6501 specifics omitted):
MCS6502 Microcomputer Family Hardware Manual; Jannuary 1976.
MOS Technology, Inc., Norristown/PA, 1976.

Registers

```
  PC .... program counter            (16 bit)
  AC .... accumulator                 (8 bit)
  X  .... X register                  (8 bit)
  Y  .... Y register                  (8 bit)
  SR .... status register [NV-BDIZC]  (8 bit)
  SP .... stack pointer               (8 bit)
```

Note: The status register (SR) is also known as the P register, the accumulator (AC) as just A, the stack pointer (SP) as S, and X and Y registers as XR and YR respectively.

Machine language monitors show them typically like,

```
     PC  IRQ  SR AC XR YR SP
    0401 E62E 32 04 5E 00 F8
```

("IRQ" is not a register, but the interrupt request vector, see below.)

- The **accumulator** is the main rgister of the 6502. Its content is typically used by the arithmetic logic unit (ALU) for the first operand and results are deposited in the accumulator again. Thus its name, as results accumulate in this register. Most arithmetic and logical operations interact with this register.
- The **X** and **Y** registers are auxiliary registers. Like the accumulator, they can be loaded directly with values, both immediatedly (as literal constants) or from memory. Additionally, they can be incremented and decremented, and their contents may be transferred to and from the acuumulator. Their main purpose is the use as index registers, where their contents is added to a base memory location, before any values are either stored to or retrieved from the resulting address, which is known as the *effective address*. This is commonly used for loops and table lookups at a given index, hence the name. (See address modes, below.)
- The **program counter** keeps track of the memory location holding the current instruction code. Its contents is automatically stepped up as the program is executed and is modified by branch and jump operations. As it must be able to address the full 16-bit address range of 64K bytes, it's the only 16-bit register of the 6502.
- The **stack pointer** points to the current *top of stack* (or rather, to its bottom, as the stack grows top-down.) The *processor stack* is located on memory page #1 ($0100–$01FF), a 256 bytes *last-in-first-out* (LIFO) stack, which enables subroutines and also serves as a quick intermediate storage. As a 8-bit register, the stack pointer holds just the low-byte of this address (the offset from $0100.)
  Be aware that this will just wrap around, in case that the stack underflows.
- The **status register** holds the status of the processor, consisting of flags reflecting results of previous operations, configuration flags, like disabeling (blocking) interrupts or setting up binary encoded decimal mode (BCD), and the carry flag, which enables multi-byte arithmetics.

Status Register Flags (bit 7 to bit 0)

```
N  .... Negative
V  .... Overflow
-  .... ignored
```

```
B  .... Break
D  .... Decimal (use BCD for arithmetics)
I  .... Interrupt (IRQ disable)
Z  .... Zero
C  .... Carry
```

- The **zero flag** (Z) indicates a value of all zero bits and the **negative flag** (N) indicates the presence of a set sign bit in bit-position 7. These flags are always updated, whenever a value is transferred to a CPU register (A,X,Y) and as a result of any logical ALU operations. The Z and N flags are also updated by increment and decrement operations acting on a memory location.
- The **carry flag** (C) flag is used as a buffer and as a borrow in arithmetic operations. Any comparisons will update this additionally to the Z and N flags, as do shift and rotate operations.
- All arithmetic operations update the Z, N, C and V flags.
- The **overflow flag** (V) indicates overflow with signed binary arithmetics. As a signed byte represents a range of -128 to +127, an overflow can never occur when the operands are of opposite sign, since the result will never exceed this range. Thus, overflow may only occur, if both operands are of the same sign. Then, the result must be also of the same sign. Otherwise, overflow is detected and the overflow flag is set. (I.e., both operands have a zero in the sign position at bit 7, but bit 7 of the result is 1, or, both operands have the sign-bit set, but the result is positive.)
- The **decimal flag** (D) sets the ALU to binary coded decimal (BCD) mode for additions and subtractions (ADC, SBC).
- The **interrupt inhibit flag** (I) blocks any maskable interrupt requests (IRQ).
- The **break flag** (B) is not an actual flag implemented in a register, and rather appears only, when the status register is pushed onto or pulled from the stack. When pushed, it will be 1 when transfered by a BRK or PHP instruction, and zero otherwise (i.e., when pushed by a hardware interrupt). When pulled into the status register (by PLP or on RTI), it will be ignored.
  In other words, the break flag will be inserted, whenever the status register is transferred to the stack by software (BRK or PHP), and will be zero, when transferred by hardware. Since there is no actual slot for the break flag, it will be always ignored, when retrieved (PLP or RTI). The break flag is not accessed by the CPU at anytime and there is no internal representation. Its purpose is more for patching, to discern an interrupt caused by a BRK instruction from a normal interrupt initiated by hardware.
- Any of these flags (but the break flag) may be set or cleared by dedicated instructions. Moreover, there are branch

instructions to conditionally divert the control flow
depending on the respective state of the Z, N, C or V flag.

Processor Stack

LIFO, top-down, 8 bit range, 0x0100 - 0x01FF

Bytes, Words, Addressing

8 bit bytes, 16 bit words in lobyte-hibyte representation (Little-Endian).
16 bit address range, operands follow instruction codes.

Signed values are two's complement, sign in bit 7 (most significant bit).
(%11111111 = $FF = -1, %10000000 = $80 = -128, %01111111 = $7F = +127)
Signed binary and binary coded decimal (BCD) arithmetic modes.

System Vectors

$FFFA, $FFFB ... NMI (Non-Maskable Interrupt) vector, 16-bit (LB, HB)
$FFFC, $FFFD ... RES (Reset) vector, 16-bit (LB, HB)
$FFFE, $FFFF ... IRQ (Interrupt Request) vector, 16-bit (LB, HB)

Start/Reset Operations

An active-low reset line allows to hold the processor in a known disabled state, while the system is initialized. As the reset line goes high, the processor performs a start sequence of 7 cycles, at the end of which the program counter (PC) is read from the address provided in the 16-bit reset vector at $FFFC (LB-HB). Then, at the eighth cycle, the processor transfers control by performing a JMP to the provided address.
Any other initializations are left to the thus executed program. (Notably, instructions exist for the initialization and loading of all registers, but for the program counter, which is provided by the reset vector at $FFFC.)

Instructions by Type

- **Transfer Instructions**

  Load, store, interregister transfer

  LDA .... load accumulator
  LDX .... load X
  LDY .... load Y
  STA .... store accumulator
  STX .... store X
  STY .... store Y
  TAX .... transfer accumulator to X
  TAY .... transfer accumulator to Y
  TSX .... transfer stack pointer to X

  TXA .... transfer X to accumulator
  TXS .... transfer X to stack pointer
  TYA .... transfer Y to accumulator

- **Stack Instructions**

  These instructions transfer the accumulator or status register (flags) to and from the stack. The processor stack is a last-in-first-out (LIFO) stack of 256 bytes length, implemented at addresses $0100 - $01FF. The stack grows down as new values are pushed onto it with the current insertion point maintained in the stack pointer register.
  (When a byte is pushed onto the stack, it will be stored in the address indicated by the value currently in the stack pointer, which will be then decremented by 1. Conversely, when a value is pulled from the stack, the stack pointer is incremented. The stack pointer is accessible by the TSX and TXS instructions.)

  PHA .... push accumulator
  PHP .... push processor status register (with break flag set)
  PLA .... pull accumulator
  PLP .... pull processor status register

- **Decrements & Increments**

  DEC .... decrement (memory)
  DEX .... decrement X
  DEY .... decrement Y
  INC .... increment (memory)
  INX .... increment X
  INY .... increment Y

- **Arithmetic Operations**

  ADC .... add with carry (prepare by CLC)
  SBC .... subtract with carry (prepare by SEC)

  See the Primer of 6502 Arithmetic Instructions below for details.

- **Logical Operations**

  AND .... and (with accumulator)
  EOR .... exclusive or (with accumulator)
  ORA .... (inclusive) or with accumulator

- **Shift & Rotate Instructions**

  All shift and rotate instructions preserve the bit shifted out in the carry flag.

  ASL .... arithmetic shift left (shifts in a zero bit on the right)
  LSR .... logical shift right (shifts in a zero bit on the left)
  ROL .... rotate left (shifts in carry bit on the right)
  ROR .... rotate right (shifts in zero bit on the left)

- **Flag Instructions**

CLC .... clear carry
CLD .... clear decimal (BCD arithmetics disabled)
CLI .... clear interrupt disable
CLV .... clear overflow
SEC .... set carry
SED .... set decimal (BCD arithmetics enabled)
SEI .... set interrupt disable

- **Comparisons**

Generally, comparison instructions subtract the operand from the given register without affecting this register. Flags are still set as with a normal subtraction and thus the relation of the two values becomes accessible by the Zero, Carry and Negative flags.
(See the branch instructions below for how to evaluate flags.)

Relation R - Op	Z	C	N
Register < Operand	0	0	*sign bit of result*
Register = Operand	1	1	0
Register > Operand	0	1	*sign bit of result*

CMP .... compare (with accumulator)
CPX .... compare with X
CPY .... compare with Y

- **Conditional Branch Instructions**

Branch targets are relative, signed 8-bit address offsets. (An offset of #0 corresponds to the immedately following address — or a rather odd and expensive NOP.)

BCC .... branch on carry clear
BCS .... branch on carry set
BEQ .... branch on equal (zero set)
BMI .... branch on minus (negative set)
BNE .... branch on not equal (zero clear)
BPL .... branch on plus (negative clear)
BVC .... branch on overflow clear
BVS .... branch on overflow set

- **Jumps & Subroutines**

JSR and RTS affect the stack as the return address is pushed onto or pulled from the stack, respectively.
(JSR will first push the high-byte of the return address [PC+2] onto the stack, then the low-byte. The stack will then contain, seen from the bottom or from the most recently added byte, [PC+2]-L [PC+2]-H.)

JMP .... jump
JSR .... jump subroutine

RTS .... return from subroutine

- **Interrupts**

A hardware interrupt (maskable IRQ and non-maskable NMI), will cause the processor to put first the address currently in the program counter onto the stack (in HB-LB order), followed by the value of the status register. (The stack will now contain, seen from the bottom or from the most recently added byte, SR PC-L PC-H with the stack pointer pointing to the address below the stored contents of status register.) Then, the processor will divert its control flow to the address provided in the two word-size interrupt vectors at $FFFA (IRQ) and $FFFE (NMI).
A set interrupt disable flag will inhibit the execution of an IRQ, but not of a NMI, which will be executed anyways.
The break instruction (BRK) behaves like a NMI, but will push the value of PC+2 onto the stack to be used as the return address. Also, as with any software initiated transfer of the status register to the stack, the break flag will be found set on the respective value pushed onto the stack. Then, control is transferred to the address in the NMI-vector at $FFFE.
In any way, the interrupt disable flag is set to inhibit any further IRQ as control is transferred to the interrupt handler specified by the respective interrupt vector.

The RTI instruction restores the status register from the stack and behaves otherwise like the JSR instruction. (The break flag is always ignored as the status is read from the stack, as it isn't a real processor flag anyway.)

BRK .... break / software interrupt
RTI .... return from interrupt

- **Other**

BIT .... bit test (accumulator & memory)
NOP .... no operation

6502 Address Modes in Detail

(This section, especially the diagrams included, is heavily inspired by the Acorn Atom manual "Atomic Theory and Practice" by David Johnson Davies, Acorn Computers Limited, 2nd ed. 1980, p 118-121.)

- **Implied Addressing**

These instructions act directly on one or more registers or flags internal to the CPU. Therefor, these instructions are principally single-byte instructions, lacking an explicit operand. The operand is implied, as it is already provided by the very instruction.

Instructions targeting exclusively the contents of the accumulator may or may not be denoted by using an explicit "A" as the operand, depending on the flavor of syntax. (This may be regarded as a special address mode of its own, but it is really a special case of

an implied instruction. It is still a single-byte instruction and no operand is provided in machine language.)

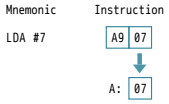
Mnemonic Examples:

```
CLC ......... clear the carry flag
ROL A ....... rotate contents of accumulator left by one position
ROL ......... same as above, implicit notation (A implied)
TXA ......... transfer contents of X-register to the accumulator
PHA ......... push the contents of the accumulator to the stack
RTS ......... return from subroutine (by pulling PC from stack)
```

Mind that some of these instructions, while simple in appearance, may be quite complex operations, like "PHA", which involves the accumulator, the stack pointer and memory access.

- **Immediate Addressing**

Here, a literal operand is given immediately after the instruction. The operand is always an 8-bit value and the total instruction length is always 2 bytes. In memory, the operand is a single byte following immediately after the instruction code. In assembler, the mode is usually indicated by a "#" prefix adjacent to the operand.



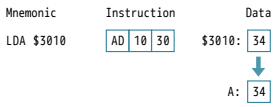
Mnemonic Examples:

```
LDA #$07 .... load the literal hexidecimal value "$7" into the accumulator
ADC #$A0 .... add the literal hexidecimal value "$A0" to the accumulator
CPX #$32 .... compare the X-register to the literal hexidecimal value "$32"
```

- **Absolute Addressing**

Absolute addressing modes provides the 16-bit address of a memory location, the contents of which used as the operand to the instruction. In machine language, the address is provided in two bytes immediately after the instruction (making these 3-byte instructions) in low-byte, high-byte order (LLHH) or little-endian. In assembler, conventional numbers (HHLL order or big-endian words) are used to provide the address.

Absolute addresses are also used for the jump instructions JMP and JSR to provide the address for the next instruction to continue with in the control flow.



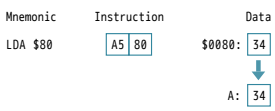
Mnemonic Examples:

```
LDA $3010 ... load the contents of address "$3010" into the accumulator
ROL $08A0 ... rotate the contents of address "$08A0" left by one position
JMP $4000 ... jump to (continue with) location "$4000"
```

- **Zero-Page Addressing**

The 16-bit address space available to the 6502 is thought to consist of 256 "pages" of 256 memory locations each ($00…$FF). In this model the high-byte of an address gives the page number and the low-byte a location inside this page. The very first of these pages, where the high-byte is zero (addresses $0000…$00FF), is somewhat special.

The zero-page address mode is similar to absolute address mode, but these instructions use only a single byte for the operand, the low-byte, while the high-byte is assumed to be zero by definition. Therefore, these instructions have a total length of just two bytes (one less than absolute mode) and take one CPU cycle less to execute, as there is one byte less to fetch.



Mnemonic Examples:

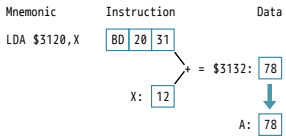
```
LDA $80 ..... load the contents of address "$0080" into the accumulator
BIT $A2 ..... perform bit-test with the contents of address "$00A2"
ASL $9A ..... arithmetic shift left of the contents of location "$009A"
```

(One way to think of the zero-page is as a page of 256 additional registers, somewhat slower than the internal registers, but with zero-page instructions also faster executing than "normal" instructions. The zero-page has a few more tricks up its sleeve, making these addresses perform more like real registers, see below.)

- **Indexed Addressing: Absolute,X and Absolute,Y**

Indexed addressing adds the contents of either the X-register or the Y-register to the provided address to give the *effective address*, which provides the operand.

These instructions are usefull to e.g., load values from tables or to write to a continuous segment of memory in a loop. The most basic forms are "absolute,X" and "absolute,X", where either the X- or the Y-register, respectively, is added to a given base address. As the base address is a 16-bit value, these are generally 3-byte instructions. Since there is an additional operation to perform to determine the effective address, these instructions are one cycle slower than those using absolute addressing mode.*



Mnemonic Examples:

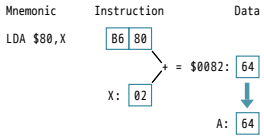
```
LDA $3120,X ... load the contents of address "$3120 + X" into A
LDX $8240,Y ... load the contents of address "$8240 + Y" into X
INC $1400,X ... increment the contents of address "$1400 + X"
```

*) If the addition of the contents of the index register effects in a change of the high-byte given by the base address so that the effective address is on the next memory page, the additional operation to increment the high-byte takes another CPU cycle. This is also known as a crossing of page boundaries.

- **Indexed Addressing: Zero-Page,X (and Zero-Page,Y)**

  As with absolute addressing, there is also a zero-page mode for indexed addressing. However, this is generally only available with the X-register. (The only exception to this is LDX, which has an indexed zero-page mode utilizing the Y-register.)
  As we have already seen with normal zero-page mode, these instructions are one byte less in total length (two bytes) and take one CPU cycle less than instructions in absolute indexed mode.

  Unlike absolute indexed instructions with 16-bit base addresses, zero-page indexed instructions never affect the high-byte of the effective address, which will simply wrap around in the zero-page, and there is no penalty for crossing any page boundaries.



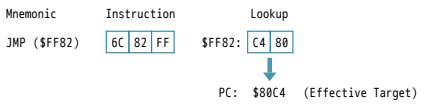
Mnemonic Examples:

```
LDA $80,X ... load the contents of address "$0080 + X" into A
LSR $82,X ... shift the contents of address "$0082 + X" left
LDX $60,Y ... load the contents of address "$0060 + Y" into X
```

- **Indirect Addressing**

  This mode looks up a given address and uses the contents of this address and the next one (in LLHH little-endian order) as the effective address. In its basic form, this mode is available for the JMP instruction only. (Its generally use is jump vectors and jump tables.)
  Like the absolute JMP instruction it uses a 16-bit address (3 bytes in total), but takes two additional CPU cycles to execute, since there are two additional bytes to fetch for the lookup of the effective jump target.

  Generally, indirect addressing is denoted by putting the lookup address in parenthesis.



Mnemonic Example:

```
JMP ($FF82) ... jump to address given in addresses "$FF82" and "$FF83"
```

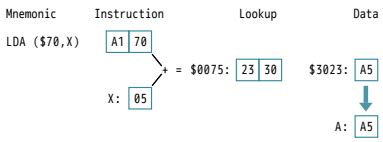
- **Pre-Indexed Indirect, "(Zero-Page,X)"**

  Indexed indirect address modes are generally available only for instructions supplying an operand to the accumulator (LDA, STA, ADC, SBC, AND, ORA, EOR, etc). The placement of the index register inside or outside of the parenthesis indicating the address lookup will give you clue what these instructions are doing.

  Pre-indexed indirect address mode is only available in combination with the X-register. It works much like the "zero-page,X" mode, but, after the X-register has been added to the base address, instead of directly accessing this, an additional lookup is performed, reading the contents of resulting address and the next one (in LLHH little-endian order), in order to determine the effective address.

  Like with "zero-page,X" mode, the total instruction length is 2 bytes, but there are two additional CPU cycles in order to fetch the effective 16-bit address. As "zero-page,X" mode, a lookup address will never overflow into the next page, but will simply wrap around in the zero-page.

  These instructions are useful, whenever we want to loop over a table of pointers to disperse addresses, or where we want to apply the same operation to various addresses, which we have stored as a table in the zero-page.



Mnemonic Examples:

```
LDA ($70,X) ... load the contents of the location given in addresses
                "$0070+X" and "$0070+1+X" into A
```

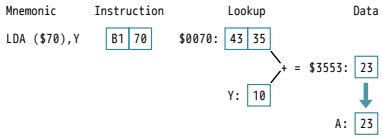
```
STA ($A2,X) ... store the contents of A in the location given in
                addresses "$00A2+X" and "$00A3+X"
EOR ($BA,X) ... perform an exlusive OR of the contents of A and the contents
                of the location given in addresses "$00BA+X" and "$00BB+X"
```

- **Post-Indexed Indirect, "(Zero-Page),Y"**

Post-indexed indirect addressing is only available in combination with the Y-register. As indicated by the indexing term ",Y" being appended to the outside of the parenthesis indicating the indirect lookup, here, a pointer is first read (from the given zero-page address) and resolved and only then the contents of the Y-register is added to this to give the effective address.

Like with "zero-page,Y" mode, the total instruction length is 2 bytes, but there it takes an additional CPU cycles to resolve and index the 16-bit pointer. As with "absolute,X" mode, the effective address may overflow into the next page, in the case of which the execution uses an extra CPU cycle.

These instructions are useful, wherever we want to perform lookups on varying bases addresses or whenever we want to loop over tables, the base address of which we have stored in the zero-page.



Mnemonic Examples:

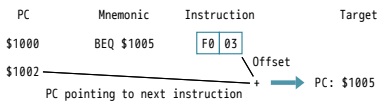
```
LDA ($70),Y ... add the contents of the Y-register to the pointer provided in
                "$0070" and "$0071" and load the contents of this address into A
STA ($A2),Y ... store the contents of A in the location given by the pointer
                in "$00A2" and "$00A3" plus the contents of the Y-register
EOR ($BA),Y ... perform an exlusive OR of the contents of A and the address
                given by the addition of Y to the pointer in "$00BA" and "$00BB"
```

- **Relative Addressing (Conditional Branching)**

This final address mode is exlusive to conditional branch instructions, which branch in the execution path depending on the state of a given CPU flag. Here, the instruction provides only a relative offset, which is added to the contents of the program counter (PC) as it points to the immediate next instruction. The relative offset is a signed single byte value in two's complement encoding (giving a range of -128…+127), which allows for branching up to half a page forwards and backwards.
On the one hand, this makes these instructions compact, fast and relocatable at the same time. On the other hand, we have to mind that our branch target is no farther away than half a memory page.

Generally, an assembler will take care of this and we only have to provide the target address, not having to worry about relative addressing.

These instructions are always of 2 bytes length and perform in 2 CPU cycles, if the branch is not taken (the condition resolving to 'false'), and 3 cycles, if the branch is taken (when the condition is true). If a branch is taken and the target is on a different page, this adds another CPU cycle (4 in total).



Mnemonic Examples:

```
BEQ $1005 ... branch to location "$1005", if the zero flag is set.
              if the current address is $1000, this will give an offset of $03.
BCS $08C4 ... branch to location "$08C4", if the carry flag is set.
              if the current address is $08D4, this will give an offset of $EE (-$12).
BCC $084A ... branch to location "$084A", if the carry flag is clear.
```

Vendor

MOS Technology, 1975

Image: Wikimedia Commons.

6502 Instructions in Detail

ADC Add Memory to Accumulator with Carry

```
     A + M + C -> A, C                N Z C I D V
                                      + + + - - +

     addressing    assembler    opc  bytes cycles
     ---------------------------------------------
     immediate     ADC #oper     69     2      2
     zeropage      ADC oper      65     2      3
     zeropage,X    ADC oper,X    75     2      4
     absolute      ADC oper      6D     3      4
     absolute,X    ADC oper,X    7D     3      4*
     absolute,Y    ADC oper,Y    79     3      4*
     (indirect,X)  ADC (oper,X)  61     2      6
     (indirect),Y  ADC (oper),Y  71     2      5*
```

AND AND Memory with Accumulator

A AND M -> A

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	AND #oper	29	2	2
zeropage	AND oper	25	2	3
zeropage,X	AND oper,X	35	2	4
absolute	AND oper	2D	3	4
absolute,X	AND oper,X	3D	3	4*
absolute,Y	AND oper,Y	39	3	4*
(indirect,X)	AND (oper,X)	21	2	6
(indirect),Y	AND (oper),Y	31	2	5*

ASL Shift Left One Bit (Memory or Accumulator)

C <- [76543210] <- 0

N	Z	C	I	D	V
+	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
accumulator	ASL A	0A	1	2
zeropage	ASL oper	06	2	5
zeropage,X	ASL oper,X	16	2	6
absolute	ASL oper	0E	3	6
absolute,X	ASL oper,X	1E	3	7

BCC Branch on Carry Clear

branch on C = 0

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BCC oper	90	2	2**

BCS Branch on Carry Set

branch on C = 1

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BCS oper	B0	2	2**

BEQ Branch on Result Zero

branch on Z = 1

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BEQ oper	F0	2	2**

BIT Test Bits in Memory with Accumulator

bits 7 and 6 of operand are transfered to bit 7 and 6 of SR (N,V);
the zero-flag is set according to the result of the operand AND
the accumulator (set, if the result is zero, unset otherwise).
This allows a quick check of a few bits at once without affecting
any of the registers, other than the status register (SR).

A AND M -> Z, M7 -> N, M6 -> V

N	Z	C	I	D	V
M7	+	-	-	-	M6

addressing	assembler	opc	bytes	cycles
zeropage	BIT oper	24	2	3
absolute	BIT oper	2C	3	4

BMI Branch on Result Minus

branch on N = 1

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BMI oper	30	2	2**

BNE Branch on Result not Zero

branch on Z = 0

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BNE oper	D0	2	2**

BPL Branch on Result Plus

branch on N = 0

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BPL oper	10	2	2**

BRK Force Break

BRK initiates a software interrupt similar to a hardware
interrupt (IRQ). The return address pushed to the stack is
PC+2, providing an extra byte of spacing for a break mark
(identifying a reason for the break.)
The status register will be pushed to the stack with the break
flag set to 1. However, when retrieved during RTI or by a PLP
instruction, the break flag will be ignored.
The interrupt disable flag is not set automatically.

interrupt,
push PC+2, push SR

N	Z	C	I	D	V
-	-	-	1	-	-

addressing	assembler	opc	bytes	cycles
implied	BRK	00	1	7

BVC Branch on Overflow Clear

branch on V = 0

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
relative	BVC oper	50	2	2**

BVS Branch on Overflow Set

branch on V = 1

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles

addressing	assembler	opc	bytes	cycles
relative	BVS oper	70	2	2**

CLC Clear Carry Flag

0 -> C

N	Z	C	I	D	V
-	-	0	-	-	-

addressing	assembler	opc	bytes	cycles
implied	CLC	18	1	2

CLD Clear Decimal Mode

0 -> D

N	Z	C	I	D	V
-	-	-	-	0	-

addressing	assembler	opc	bytes	cycles
implied	CLD	D8	1	2

CLI Clear Interrupt Disable Bit

0 -> I

N	Z	C	I	D	V
-	-	-	0	-	-

addressing	assembler	opc	bytes	cycles
implied	CLI	58	1	2

CLV Clear Overflow Flag

0 -> V

N	Z	C	I	D	V
-	-	-	-	-	0

addressing	assembler	opc	bytes	cycles
implied	CLV	B8	1	2

CMP Compare Memory with Accumulator

A - M

N	Z	C	I	D	V
+	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	CMP #oper	C9	2	2
zeropage	CMP oper	C5	2	3
zeropage,X	CMP oper,X	D5	2	4
absolute	CMP oper	CD	3	4
absolute,X	CMP oper,X	DD	3	4*
absolute,Y	CMP oper,Y	D9	3	4*
(indirect,X)	CMP (oper,X)	C1	2	6
(indirect),Y	CMP (oper),Y	D1	2	5*

CPX Compare Memory and Index X

X - M

N	Z	C	I	D	V
+	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	CPX #oper	E0	2	2
zeropage	CPX oper	E4	2	3
absolute	CPX oper	EC	3	4

CPY Compare Memory and Index Y

Y - M

N	Z	C	I	D	V
+	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	CPY #oper	C0	2	2
zeropage	CPY oper	C4	2	3
absolute	CPY oper	CC	3	4

DEC Decrement Memory by One

M - 1 -> M

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
zeropage	DEC oper	C6	2	5
zeropage,X	DEC oper,X	D6	2	6
absolute	DEC oper	CE	3	6
absolute,X	DEC oper,X	DE	3	7

DEX Decrement Index X by One

X - 1 -> X

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	DEX	CA	1	2

DEY Decrement Index Y by One

Y - 1 -> Y

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	DEY	88	1	2

EOR Exclusive-OR Memory with Accumulator

A EOR M -> A

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	EOR #oper	49	2	2
zeropage	EOR oper	45	2	3
zeropage,X	EOR oper,X	55	2	4
absolute	EOR oper	4D	3	4
absolute,X	EOR oper,X	5D	3	4*
absolute,Y	EOR oper,Y	59	3	4*
(indirect,X)	EOR (oper,X)	41	2	6
(indirect),Y	EOR (oper),Y	51	2	5*

INC Increment Memory by One

M + 1 -> M

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
zeropage	INC oper	E6	2	5

zeropage,X	INC oper,X	F6	2	6
absolute	INC oper	EE	3	6
absolute,X	INC oper,X	FE	3	7

INX Increment Index X by One

X + 1 -> X

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	INX	E8	1	2

INY Increment Index Y by One

Y + 1 -> Y

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	INY	C8	1	2

JMP Jump to New Location

operand 1st byte -> PCL
operand 2nd byte -> PCH

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
absolute	JMP oper	4C	3	3
indirect	JMP (oper)	6C	3	5

JSR Jump to New Location Saving Return Address

push (PC+2),
operand 1st byte -> PCL
operand 2nd byte -> PCH

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
absolute	JSR oper	20	3	6

LDA Load Accumulator with Memory

M -> A

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	LDA #oper	A9	2	2
zeropage	LDA oper	A5	2	3
zeropage,X	LDA oper,X	B5	2	4
absolute	LDA oper	AD	3	4
absolute,X	LDA oper,X	BD	3	4*
absolute,Y	LDA oper,Y	B9	3	4*
(indirect,X)	LDA (oper,X)	A1	2	6
(indirect),Y	LDA (oper),Y	B1	2	5*

LDX Load Index X with Memory

M -> X

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	LDX #oper	A2	2	2
zeropage	LDX oper	A6	2	3
zeropage,Y	LDX oper,Y	B6	2	4
absolute	LDX oper	AE	3	4
absolute,Y	LDX oper,Y	BE	3	4*

LDY Load Index Y with Memory

M -> Y

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	LDY #oper	A0	2	2
zeropage	LDY oper	A4	2	3
zeropage,X	LDY oper,X	B4	2	4
absolute	LDY oper	AC	3	4
absolute,X	LDY oper,X	BC	3	4*

LSR Shift One Bit Right (Memory or Accumulator)

0 -> [76543210] -> C

N	Z	C	I	D	V
0	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
accumulator	LSR A	4A	1	2
zeropage	LSR oper	46	2	5
zeropage,X	LSR oper,X	56	2	6
absolute	LSR oper	4E	3	6
absolute,X	LSR oper,X	5E	3	7

NOP No Operation

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	NOP	EA	1	2

ORA OR Memory with Accumulator

A OR M -> A

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
immediate	ORA #oper	09	2	2
zeropage	ORA oper	05	2	3
zeropage,X	ORA oper,X	15	2	4
absolute	ORA oper	0D	3	4
absolute,X	ORA oper,X	1D	3	4*
absolute,Y	ORA oper,Y	19	3	4*
(indirect,X)	ORA (oper,X)	01	2	6
(indirect),Y	ORA (oper),Y	11	2	5*

PHA Push Accumulator on Stack

push A

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles

addressing	assembler	opc	bytes	cycles
implied	PHA	48	1	3

PHP Push Processor Status on Stack

The status register will be pushed with the break flag and bit 5 set to 1.

push SR

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	PHP	08	1	3

PLA Pull Accumulator from Stack

pull A

N	Z	C	I	D	V
+	+	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	PLA	68	1	4

PLP Pull Processor Status from Stack

The status register will be pulled with the break flag and bit 5 ignored.

pull SR

N	Z	C	I	D	V
from stack					

addressing	assembler	opc	bytes	cycles
implied	PLP	28	1	4

ROL Rotate One Bit Left (Memory or Accumulator)

C <- [76543210] <- C

N	Z	C	I	D	V
+	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
accumulator	ROL A	2A	1	2
zeropage	ROL oper	26	2	5
zeropage,X	ROL oper,X	36	2	6
absolute	ROL oper	2E	3	6
absolute,X	ROL oper,X	3E	3	7

ROR Rotate One Bit Right (Memory or Accumulator)

C -> [76543210] -> C

N	Z	C	I	D	V
+	+	+	-	-	-

addressing	assembler	opc	bytes	cycles
accumulator	ROR A	6A	1	2
zeropage	ROR oper	66	2	5
zeropage,X	ROR oper,X	76	2	6
absolute	ROR oper	6E	3	6
absolute,X	ROR oper,X	7E	3	7

RTI Return from Interrupt

The status register is pulled with the break flag and bit 5 ignored. Then PC is pulled from the stack.

pull SR, pull PC

N	Z	C	I	D	V
from stack					

addressing	assembler	opc	bytes	cycles
implied	RTI	40	1	6

RTS Return from Subroutine

pull PC, PC+1 -> PC

N	Z	C	I	D	V
-	-	-	-	-	-

addressing	assembler	opc	bytes	cycles
implied	RTS	60	1	6

SBC Subtract Memory from Accumulator with Borrow

A - M - $\overline{C}$ -> A

N	Z	C	I	D	V
+	+	+	-	-	+

addressing	assembler	opc	bytes	cycles
immediate	SBC #oper	E9	2	2
zeropage	SBC oper	E5	2	3
zeropage,X	SBC oper,X	F5	2	4
absolute	SBC oper	ED	3	4
absolute,X	SBC oper,X	FD	3	4*
absolute,Y	SBC oper,Y	F9	3	4*
(indirect,X)	SBC (oper,X)	E1	2	6
(indirect),Y	SBC (oper),Y	F1	2	5*

SEC Set Carry Flag

1 -> C

N	Z	C	I	D	V
-	-	1	-	-	-

addressing	assembler	opc	bytes	cycles
implied	SEC	38	1	2

SED Set Decimal Flag

1 -> D

N	Z	C	I	D	V
-	-	-	-	1	-

addressing	assembler	opc	bytes	cycles
implied	SED	F8	1	2

SEI Set Interrupt Disable Status

1 -> I

N	Z	C	I	D	V
-	-	-	1	-	-

addressing	assembler	opc	bytes	cycles
implied	SEI	78	1	2

STA Store Accumulator in Memory

N	Z	C	I	D	V

A -> M - - - - - -

addressing	assembler	opc	bytes	cycles
zeropage	STA oper	85	2	3
zeropage,X	STA oper,X	95	2	4
absolute	STA oper	8D	3	4
absolute,X	STA oper,X	9D	3	5
absolute,Y	STA oper,Y	99	3	5
(indirect,X)	STA (oper,X)	81	2	6
(indirect),Y	STA (oper),Y	91	2	6

STX Store Index X in Memory

X -> M N Z C I D V
- - - - - -

addressing	assembler	opc	bytes	cycles
zeropage	STX oper	86	2	3
zeropage,Y	STX oper,Y	96	2	4
absolute	STX oper	8E	3	4

STY Sore Index Y in Memory

Y -> M N Z C I D V
- - - - - -

addressing	assembler	opc	bytes	cycles
zeropage	STY oper	84	2	3
zeropage,X	STY oper,X	94	2	4
absolute	STY oper	8C	3	4

TAX Transfer Accumulator to Index X

A -> X N Z C I D V
+ + - - - -

addressing	assembler	opc	bytes	cycles
implied	TAX	AA	1	2

TAY Transfer Accumulator to Index Y

A -> Y N Z C I D V
+ + - - - -

addressing	assembler	opc	bytes	cycles
implied	TAY	A8	1	2

TSX Transfer Stack Pointer to Index X

SP -> X N Z C I D V
+ + - - - -

addressing	assembler	opc	bytes	cycles
implied	TSX	BA	1	2

TXA Transfer Index X to Accumulator

X -> A N Z C I D V
+ + - - - -

addressing	assembler	opc	bytes	cycles
implied	TXA	8A	1	2

TXS Transfer Index X to Stack Register

X -> SP N Z C I D V
- - - - - -

addressing	assembler	opc	bytes	cycles
implied	TXS	9A	1	2

TYA Transfer Index Y to Accumulator

Y -> A N Z C I D V
+ + - - - -

addressing	assembler	opc	bytes	cycles
implied	TYA	98	1	2

* add 1 to cycles if page boundary is crossed

** add 1 to cycles if branch occurs on same page
add 2 to cycles if branch occurs to different page

Legend to Flags: + ... modified
- .... not modified
1 .... set
0 .... cleared
M6 .... memory bit 6
M7 .... memory bit 7

Note on assembler syntax:
Some assemblers employ "OPC *oper" or a ".b" extension to the mneomonic for forced zeropage addressing.

Note on Read-Modify-Write instructions *(NMOS 6502 only)*:
Some instructions like EOR, ASL, ROL, DEC, INC, etc., fetch a value from memory to modify it and to write the modified value back to the originating address. The original NMOS 6502 switches immediately into write mode after the read of the value, resulting in the unmodified value being written back to the address (while the value is modified in the next cycle), before the modified value is finally written to the destination. This may cause issues when writing to devices attached to the address bus that may trigger some action on this intermediate write operation.
This does not apply to the CMOS variants of the 6502.

"Illegal" Opcodes and Undocumented Instructions

The following instructions are undocumented are not guaranteed to work. Some are highly unstable, some may even start two asynchronous threads competing in race condition with the winner determined by such miniscule factors as temperature or minor differences in the production series, at other times, the outcome depends on the exact values involved and the chip series.

Use with care and at your own risk.

There are several mnemonics for various opcodes. Here, they are (mostly) the same as those used by the ACME and DASM assemblers with known synonyms provided in parentheses:

- ALR (ASR)
- ANC
- ANC (ANC2)
- ANE (XAA)
- ARR
- DCP (DCM)
- ISC (ISB, INS)
- LAS (LAR)
- LAX
- LXA (LAX immediate)
- RLA
- RRA
- SAX (AXS, AAX)
- SBX (AXS, SAX)
- SHA (AHX, AXA)
- SHX (A11, SXA, XAS)
- SHY (A11, SYA, SAY)
- SLO (ASO)
- SRE (LSE)
- TAS (XAS, SHS)
- USBC (SBC)
- NOPs (including DOP, TOP)
- JAM (KIL, HLT)

"Illegal" Opcodes in Details

Legend to markers used in the instruction details:

*	add 1 to cycles if page boundary is crossed
†	unstable
††	highly unstable

ALR (ASR)

AND oper + LSR

A AND oper, 0 -> [76543210] -> C

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
immediate	ALR #oper	4B	2	2

ANC

AND oper + set C as ASL

A AND oper, bit(7) -> C

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
immediate	ANC #oper	0B	2	2

ANC (ANC2)

AND oper + set C as ROL

effectively the same as instr. 0B

A AND oper, bit(7) -> C

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
immediate	ANC #oper	2B	2	2

ANE (XAA)

* OR X + AND oper

Highly unstable, do not use.

A base value in A is determined based on the contets of A and a constant, which may be typically $00, $ff, $ee, etc. The value of this constant depends on temerature, the chip series, and maybe other factors, as well.
In order to eliminate these uncertaincies from the equation, use either 0 as the operand or a value of $FF in the accumulator.

(A OR CONST) AND X AND oper -> A

```
                                N Z C I D V
                                + + - - - -
```

addressing	assembler	opc	bytes	cycles
immediate	ANE #oper	8B	2	2 ††

ARR

AND oper + ROR

This operation involves the adder:
V-flag is set according to (A AND oper) + oper
The carry is not set, but bit 7 (sign) is exchanged with the carry

A AND oper, C -> [76543210] -> C

```
                                N Z C I D V
                                + + + - - +
```

addressing	assembler	opc	bytes	cycles
immediate	ARR #oper	6B	2	2

DCP (DCM)

DEC oper + CMP oper

M - 1 -> M, A - M

Decrements the operand and then compares the result to the accumulator.

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
zeropage	DCP oper	C7	2	5
zeropage,X	DCP oper,X	D7	2	6
absolute	DCP oper	CF	3	6
absolut,X	DCP oper,X	DF	3	7
absolut,Y	DCP oper,Y	DB	3	7
(indirect,X)	DCP (oper,X)	C3	2	8
(indirect),Y	DCP (oper),Y	D3	2	8

ISC (ISB, INS)

INC oper + SBC oper

M + 1 -> M, A - M - $\overline{C}$ -> A

```
                                N Z C I D V
                                + + + - - +
```

addressing	assembler	opc	bytes	cycles
zeropage	ISC oper	E7	2	5
zeropage,X	ISC oper,X	F7	2	6
absolute	ISC oper	EF	3	6
absolut,X	ISC oper,X	FF	3	7
absolut,Y	ISC oper,Y	FB	3	7
(indirect,X)	ISC (oper,X)	E3	2	8
(indirect),Y	ISC (oper),Y	F3	2	8

LAS (LAR)

LDA/TSX oper

M AND SP -> A, X, SP

```
                                N Z C I D V
                                + + - - - -
```

addressing	assembler	opc	bytes	cycles
absolut,Y	LAS oper,Y	BB	3	4*

LAX

LDA oper + LDX oper

M -> A -> X

```
                                N Z C I D V
                                + + - - - -
```

addressing	assembler	opc	bytes	cycles
zeropage	LAX oper	A7	2	3
zeropage,Y	LAX oper,Y	B7	2	4
absolute	LAX oper	AF	3	4
absolut,Y	LAX oper,Y	BF	3	4*
(indirect,X)	LAX (oper,X)	A3	2	6
(indirect),Y	LAX (oper),Y	B3	2	5*

LXA (LAX immediate)

Store * AND oper in A and X

Highly unstable, involves a 'magic' constant, see ANE

(A OR CONST) AND oper -> A -> X

```
                                N Z C I D V
                                + + - - - -
```

addressing	assembler	opc	bytes	cycles
immediate	LXA #oper	AB	2	2 ††

RLA

ROL oper + AND oper

M = C <- [76543210] <- C, A AND M -> A

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
zeropage	RLA oper	27	2	5
zeropage,X	RLA oper,X	37	2	6
absolute	RLA oper	2F	3	6
absolut,X	RLA oper,X	3F	3	7
absolut,Y	RLA oper,Y	3B	3	7
(indirect,X)	RLA (oper,X)	23	2	8
(indirect),Y	RLA (oper),Y	33	2	8

RRA

ROR oper + ADC oper

M = C -> [76543210] -> C, A + M + C -> A, C

```
                                N Z C I D V
                                + + + - - +
```

addressing	assembler	opc	bytes	cycles
zeropage	RRA oper	67	2	5
zeropage,X	RRA oper,X	77	2	6
absolute	RRA oper	6F	3	6
absolut,X	RRA oper,X	7F	3	7
absolut,Y	RRA oper,Y	7B	3	7
(indirect,X)	RRA (oper,X)	63	2	8
(indirect),Y	RRA (oper),Y	73	2	8

SAX (AXS, AAX)

A and X are put on the bus at the same time (resulting effectively in an AND operation) and stored in M

A AND X -> M

```
                                N Z C I D V
                                - - - - - -
```

addressing	assembler	opc	bytes	cycles
zeropage	SAX oper	87	2	3
zeropage,Y	SAX oper,Y	97	2	4
absolute	SAX oper	8F	3	4
(indirect,X)	SAX (oper,X)	83	2	6

SBX (AXS, SAX)

CMP and DEX at once, sets flags like CMP

(A AND X) - oper -> X

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
immediate	SBX #oper	CB	2	2

SHA (AHX, AXA)

Stores A AND X AND (high-byte of addr. + 1) at addr.

unstable: sometimes 'AND (H+1)' is dropped, page boundary crossings may not work (with the high-byte of the value used as the high-byte of the address)

A AND X AND (H+1) -> M

```
                                N Z C I D V
                                - - - - - -
```

addressing	assembler	opc	bytes	cycles
absolut,Y	SHA oper,Y	9F	3	5 †
(indirect),Y	SHA (oper),Y	93	2	6 †

SHX (A11, SXA, XAS)

Stores X AND (high-byte of addr. + 1) at addr.

unstable: sometimes 'AND (H+1)' is dropped, page boundary crossings may not work (with the high-byte of the value used as the high-byte of the address)

X AND (H+1) -> M

```
                                N Z C I D V
                                - - - - - -
```

addressing	assembler	opc	bytes	cycles
absolut,Y	SHX oper,Y	9E	3	5 †

SHY (A11, SYA, SAY)

Stores Y AND (high-byte of addr. + 1) at addr.

unstable: sometimes 'AND (H+1)' is dropped, page boundary crossings may not work (with the high-byte of the value used as the high-byte of the address)

Y AND (H+1) -> M

```
                                N Z C I D V
                                - - - - - -
```

addressing	assembler	opc	bytes	cycles
absolut,X	SHY oper,X	9C	3	5 †

SLO (ASO)

ASL oper + ORA oper

M = C <- [76543210] <- 0, A OR M -> A

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
zeropage	SLO oper	07	2	5
zeropage,X	SLO oper,X	17	2	6
absolute	SLO oper	0F	3	6
absolut,X	SLO oper,X	1F	3	7
absolut,Y	SLO oper,Y	1B	3	7
(indirect,X)	SLO (oper,X)	03	2	8
(indirect),Y	SLO (oper),Y	13	2	8

SRE (LSE)

LSR oper + EOR oper

M = 0 -> [76543210] -> C, A EOR M -> A

```
                                N Z C I D V
                                + + + - - -
```

addressing	assembler	opc	bytes	cycles
zeropage	SRE oper	47	2	5
zeropage,X	SRE oper,X	57	2	6
absolute	SRE oper	4F	3	6
absolut,X	SRE oper,X	5F	3	7
absolut,Y	SRE oper,Y	5B	3	7
(indirect,X)	SRE (oper,X)	43	2	8
(indirect),Y	SRE (oper),Y	53	2	8

TAS (XAS, SHS)

Puts A AND X in SP and stores A AND X AND (high-byte of addr. + 1) at addr.

unstable: sometimes 'AND (H+1)' is dropped, page boundary crossings may not work (with the high-byte of the value used as the high-byte of the address)

A AND X -> SP, A AND X AND (H+1) -> M

```
                                N Z C I D V
                                - - - - - -
```

addressing	assembler	opc	bytes	cycles
absolut,Y	TAS oper,Y	9B	3	5 †

USBC (SBC)

SBC oper + NOP

effectively same as normal SBC immediate, instr. E9.

A - M - C¯ -> A

```
                                N Z C I D V
                                + + + - - +
```

addressing	assembler	opc	bytes	cycles
immediate	USBC #oper	EB	2	2

NOPs (including DOP, TOP)

Instructions effecting in 'no operations' in various address modes. Operands are ignored.

```
                                N Z C I D V
                                - - - - - -
```

opc	addressing	bytes	cycles
1A	implied	1	2
3A	implied	1	2
5A	implied	1	2
7A	implied	1	2
DA	implied	1	2
FA	implied	1	2
80	immediate	2	2
82	immediate	2	2
89	immediate	2	2
C2	immediate	2	2
E2	immediate	2	2
04	zeropage	2	3
44	zeropage	2	3
64	zeropage	2	3
14	zeropage,X	2	4
34	zeropage,X	2	4
54	zeropage,X	2	4
74	zeropage,X	2	4
D4	zeropage,X	2	4
F4	zeropage,X	2	4
0C	absolute	3	4
1C	absolut,X	3	4*
3C	absolut,X	3	4*
5C	absolut,X	3	4*
7C	absolut,X	3	4*
DC	absolut,X	3	4*
FC	absolut,X	3	4*

JAM (KIL, HLT)

These instructions freeze the CPU.

The processor will be trapped infinitely in T1 phase with $FF on the data bus. — Reset required.

Instruction codes: 02, 12, 22, 32, 42, 52, 62, 72, 92, B2, D2, F2

Have a look at this table of the instruction layout in order to see how most of these "illegal" instructions are a result of executing both instructions at *c=1* and *c=2* in a given slot (same column, rows immediately above) at once.
Where *c* is the lowest two bits of the instruction code. E.g., "SAX abs", instruction code $8F, binary 100011**11**, is "STA abs", 100011**01** ($8D) and "STX abs", 100011**10** ($8E).

Honorable Mention: Rev. A 6502 ROR, Pre-June 1976

Famously, the Rev. A 6502 as delivered from September 1975 to June 1976 had a *"ROR bug"*. However, the "ROR" instruction isn't only missing from the original documentation, as it turns out, the chip is actually missing crucial control lines, which would have been required to make this instruction work. The instruction is simply not implemented and it wasn't even part of the design. (This was actually added on popular demand in Rev. B, as rumor has it, demand by Steve Wozniak. Even, if not true, this makes for a good story. And how could there be a page on the 6502 without mentioning "Woz" once?) So, for all means, "ROR" is an undocumented or "illegal" instruction on the Rev. A 6502.

And this is how **ROR** behaves on these Rev. A chips, much like **ASL**: it shifts all bits to the left, shifting in a zero bit at the LSB side, but, unlike **ASL**, it does not shift the high-bit into the carry. (So there are no connections to the carry at all.)

ROR Rev. A (pre-June 1976)

As ASL, but does not update the carry.

N and Z flags are set correctly for the operation performed.

[76543210] <- 0

```
                                N Z C I D V
                                + + - - - -
```

addressing	assembler	opc	bytes	cycles
accumulator	ROR A	6A	1	2
zeropage	ROR oper	66	2	5
zeropage,X	ROR oper,X	76	2	6
absolute	ROR oper	6E	3	6
absolute,X	ROR oper,X	7E	3	7

Compare Instructions

The 6502 MPU features three basic compare instructions in various address modes:

Instruction	Comparison
CMP	Accumulator and operand
CPX	X register and operand
CPY	Y register and operand

The various compare instructions subtract the operand from the respective register (as if the carry was set) without setting the result and adjust the N, Z, and C flags accordingly to this operation.
Flags will be set as follows:

Relation	Z	C	N
register < operand	0	0	*sign-bit of result*
register = operand	1	1	0
register > operand	0	1	*sign-bit of result*

And for the derivative relations *"less/greater than or equal"*:

Relation	Z	C	N
register ≤ operand	1	0	*sign-bit of result*
register ≥ operand	1	1	*sign-bit of result*

Mind that the negative flag is not significant and all conditions may be evaluated by checking the carry and/or zero flag(s).

The BIT Instruction

The BIT instruction may be the most obscure instruction of the 6502:
While other instruction serve a very clear purpose, like transferring values or performing basic arithmetic or logical operations, this one serves a rather specialized purpose, but it does so in a very general way.
This purpose is bit testing.

Generally, testing of a particular bit is achieved by masking (isolating) this bit (or multiple bits) by an AND operation and then checking the zero flag (Z) by a BNE or BEQ instruction. This, however, destroys the contents of the accumulator. This is, where the BIT instruction comes in: much like the comparisons perform a subtraction without setting the result, the BIT instruction performs a logical AND without setting the result, but still reflects the result in the state of the zero flag (Z). Which allows for the same checks using the BNE or BEQ instructions, without affecting the contents of the accumulator.

Since the sign-bit is often used as a flag, testing this is also covered by the BIT instruction, which additionally to setting the zero flag also transfers bits 7 and 6 of the operand into the corresponding bits of the status register — which happen to be the negative (N) and overflow (V) flags. Therefore, bits 7 and 6 of the operand may be tested independently using the BMI/BPL and BVS/BVC instructions.

```
accumulator       operand
[76543210]  AND  [76543210]  == 0?
                  ↓↓            ↓
                  NV            Z
```

A Primer of 6502 Arithmetic Operations

The 6502 processor features two basic arithmetic instructions, **ADC**, *ADd with Carry*, and **SBC**, *SuBtract with Carry*. As the names suggest, these provide addition and subtraction for single byte operands and results. However, operations are not limited to a single byte range, which is where the carry flag comes in, providing the means for a single-bit carry (or borrow), to combine operations over several bytes.

In order to accomplish this, the carry is included in each of these operations: for additions, it is added (much like another operand); for subtractions, which are just an addition using the inverse of the operand (complement value of the operand), the role of the carry is inverted, as well.
Therefore, it is crucial to set up the carry appropriatly: fo additions, the carry has to be initially cleared (using **CLC**), while for subtractions, it must be initally set (using **SEC** — more on **SBC** below).

```
              ;ADC: A = A + M + C
CLC           ;clear carry in preparation
LDA #2        ;load 2 into the accumulator
ADD #3        ;add 3 -> now 5 in accumulator
```

```
              ;SBC: A = A - M - C‾   ("C‾": "not carry")
SEC           ;set carry in preparation
LDA #15       ;load 15 into the accumulator
SBC #8        ;subtract 8 -> now 7 in accumulator
```

Note: Here, we used immediate mode, indicated by the prefix "#" before the operand, to directly load a literal value. If there is no such "#" prefix, we generally mean to use the value stored at the address, which is given by the operand. As we will see in the next example.)

To combine this for 16-bit values (2 bytes each), we simply chain the instructions for the next bytes to operate on, but this time without setting or clearing the carry.

Supposing the following locations for storing 16-bit values:

```
                    low-byte  high-byte

first argument ....  $1000     $1001

second argument ...  $1002     $1003

result ............  $1004     $1005
```

we perform a 16-bit addition by:

```
CLC           ;prepare carry for addition
LDA $1000     ;load value at address $1000 into A (low byte of first argument)
ADC $1002     ;add low byte of second argument at $1002
STA $1004     ;store low byte of result at $1004
LDA $1001     ;load high byte of first argument
ADC $1003     ;add high byte of second argument
STA $1005     ;store high byte of result (result in $1004 and $1005)
```

and, conversely, for a 16-bit subtraction:

```
SEC           ;prepare carry for subtraction
LDA $1000     ;load value at address $1000 into A (low byte of first argument)
SBC $1002     ;subtract low byte of second argument at $1002
STA $1004     ;store low byte of result at $1004
LDA $1001     ;load high byte of first argument
SBC $1003     ;subtract high byte of second argument
STA $1005     ;store high byte of result (result in $1004 and $1005)
```

Note: Another, important preparatory step is to set the processor into binary mode by use of the **CLD** *(CLear Decimal flag)* instruction. (Compare the section on decimal mode below.) This has to be done only once.

Signed Values

Operations for unsigned and signed values are principally the same, the only difference being in how we interpret the values. Generally, the 6502 uses what is known as *two's complement* to represent negative values.

(In earlier computers,something known as ones' complement was used, where we simply flip all bits to their opposite state to represent a negative value. While simple, this came with a few drawbacks, like an additional value of negative zero, which are overcome by two's complement.)

In two's complement representation, we simply flip all the bits in a byte to their opposite (the same as an XOR by **$FF**) and then add 1 to this.

E.g., to represent -4:,
(We here use "$" to indicate a hexadecimal number and "%" for binary notation. A dot is used to separate the high- and low-nibble, i.e. group of 4 bits.)

```
     %0000.0100     4
 XOR %1111.1111   255
 -------------
     %1111.1011   complement (all bits flipped)
  +          1
 -------------
     %1111.1100    -4, two's complement
```

Thus, in a single byte, we may represent values in the range

```
 from  -128  (%1000.0000 or $80)
 to    +127  (%0111.1111 or $7F)
```

A notable feature is that the highest value bit (first bit from the left) will always be 1 for a negative value and always be 0 for a positive one, for which it is also known as the *sign bit*. Whenever we interpret a value as a signed number, a set sign bit indicates a negative value.
This works just the same for larger values, e.g., for a signed 16-bit value:

```
 -512  =  %1111.1110.0000.0000  =  $FE $00
 -516  =  %1111.1101.1111.1100  =  $FD $FC (mind how the +1 step carries over)
```

Notably, the binary operations are still the same as with unsigned values and provide the expected results:

```
   dec     binary     hex

   100   %0110.0100   $64
 + -24   %1110.1000   $E8
 ------------------------
    76   %0100.1100   $4C  (+ carry)
```

Note: We may now see how **SBC** actually works, by adding *ones' complement* of the operand to the accumulator. If we add 1 from the carry to the result, this effectively results in a subtraction in *two's complement* (the inverse of the operand + 1). If the carry happens to be zero, the result falls short by 1 in terms of two's complement, which is equivalent to adding 1 to the operand before the subtraction. Thus, the carry either provides the correction required for a valid two's complement representation or, if missing, results in a subtraction including a binary borrow.

FLags with ADC and SBC

Besides the carry flag (C), which allows us to chain multi-byte operations, the CPU sets the following flags on the result of an arithmetic operation:

```
 zero flag (Z) ........ set if the result is zero, else unset
 negative flag (N)  ... the N flag always reflects the sign bit of the result
 overflow flag (V)  ... indicates overflow in signed operations
```

The latter may require explanation: how is signed overflow different from the carry flag? The overflow flag is about a certain ambiguity of the sign bit and the negative flag in signed context: if operands are of the same sign, the case may occure, where the sign bit flips (as indicated by a change of the negative flag), while the result is still of the same sign. This condition is indicated by the overflow flag. Notably, such an overflow can never occur, when the operands are of opposite signs.

E.g., adding positive $40 to positive $40:

```
             acc.      acc.     flags
             hex      binary    NVDIZC

 LDA #$40    $40   %0100.0000   000000
 ADC #$40    $80   %1000.0000   110000
```

Here, the change of the sign bit is unrelated to the actual value in the accumulator, it is merely a consequence of carry propagation from bit 6 to bit 7, the sign bit. Since both operands are positive, the result must be positive, as well.
The overflow flag (V) is of interest in signed context only and has no meaning in unsigned context.

Decimal Mode (BCD)

Besides binary arithmetic, the 6502 processor supports a second mode, binary coded decimal (BCD), where each byte, rather than representing a range of 0…255, represents two decimal digits packed into a single byte. For this, a byte is thought divided into two sections of 4 bits, the high- and the low-nibble. Only values from 0…9 are used for each nibble and a byte can represent a range of a 2-digit decimal value only, as in 0…99.
E.g.,

```
 dec     binary     hex

  14   %0001.0100   $14
  98   %1001.1000   $98
```

Mind how this intuitively translates to hexadecimal notation, where figures A…F are never used.

Whether or not the processor is in decimal mode is determined by the decimal flag (D). If it is set (using **SED**) the processor will use BCD arithmetic. If it is cleared (using **CLD**), the processor is in binary mode. Decimal mode only affects instructions **ADC** and **SBC** (but not **INC** or **DEC**.)

Examples:

```
  SED
  CLC
  LDA #$12
  ADC #$44  ;accumulator now holds $56

  SED
  CLC
  LDA #$28
  ADC #$14  ;accumulator now holds $42
```

Mind that BCD mode is always unsigned:

```
            acc. NVDIZC
  SED
  SEC
  LDA #0    $00  001011
  SBC #1    $99  101000
```

The carry flag and the zero flag work in decimal mode as expected.
The negative flag is set similar to binary mode (and of questionable value.)
The overflow flag has no meaning in decimal mode.

Multi-byte operations are just as in decimal mode: We first prepare the carry and then chain operations of the individual bytes in increasing value order, starting with the lowest value pair.

(It may be important to note that Western Design Center (WDC) version of the processor, the 65C02, always clears the decimal flag when it enters an interrupt, while the original NMOS version of the 6502 does not.)

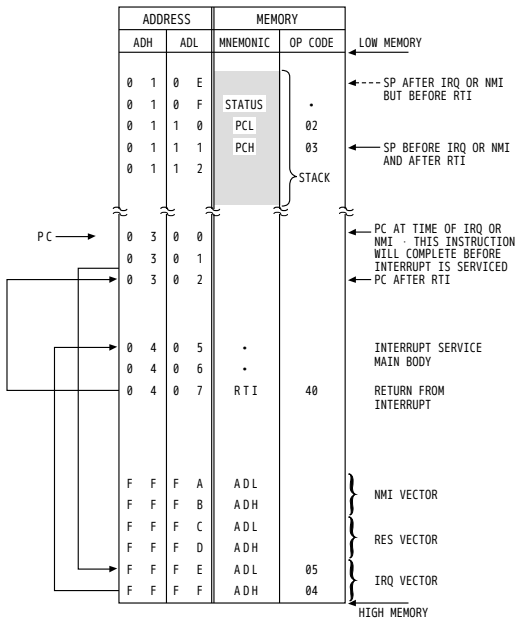
6502 Jump Vectors and Stack Operations

The 256 bytes processor stack of the 6502 is located at $0100 ... $01FF in memory, growing down from top to bottom.

There are three 2-byte address locations at the very top end of the 64K address space serving as jump vectors for reset/startup and interrupt operations:

```
  $FFFA, $FFFB ... NMI (Non-Maskable Interrupt) vector
  $FFFC, $FFFD ... RES (Reset) vector
  $FFFE, $FFFF ... IRQ (Interrupt Request) vector
```

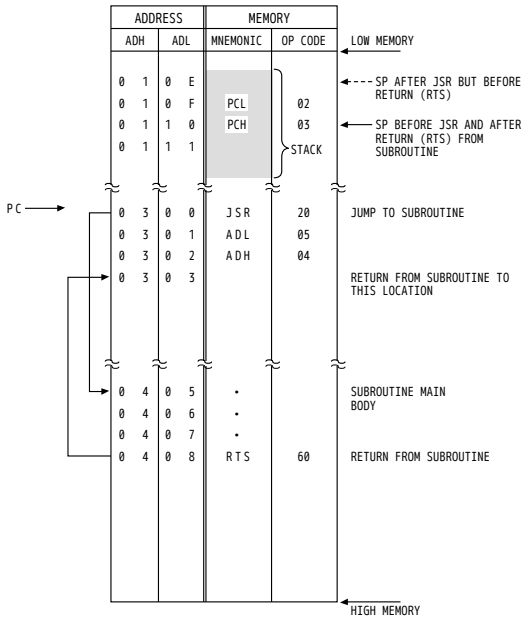
As an interrupt occurs, any instruction currently processed is completed first. Only then, the value of the program counter (PC) is put in high-low order onto the stack, followed by the value currently in the status register, and control will be transferred to the address location found in the respective interrupt vector. The registers stored on the stack are recovered at the end of an interrupt routine, as control is transferred back to the interrupted code by the RTI instruction.



IRQ, NMI, RTI, BRK OPERATION

(Reset after: MCS6502 Instruction Set Summary, MOS Technology, Inc.)

Similarly, as a JSR instruction is encountered, PC is dumped onto the stack and recovered by the JSR instruction. (Here, the value stored is actually the address *before* the location, the program will eventually return to. Thus, the effective return address is PC+1.)



JSR, RTS OPERATION

(Reset after: MCS6502 Instruction Set Summary, MOS Technology, Inc.)

Curious Interrupt Behavior

- If the instruction was a taken branch instruction with 3 cycles execution time (without crossing page boundaries), the interrupt will trigger only after an extra CPU cycle.
- On the NMOS6502, an NMI hardware interrupt occuring at the start of a BRK intruction will hijack the BRK instruction, meanining, the BRK instruction will be executed as normal, but the NMI vector will be used instead of the IRQ vector.
- The 65C02 will clear the decimal flag on any interrupts (and BRK).

The Break Flag and the Stack

Interrupts and stack operations involving the status register (or P register) are the only instances, the break flag appears (namely on the stack). It has no representation in the CPU and can't be accessed by any instruction.

- The break flag will be set to on (1), whenever the transfer was caused by software (BRK or PHP).
- The break flag will be set to zero (0), whenever the transfer was caused by a hardware interrupt.
- The break flag will be masked and cleared (0), whenever transferred from the stack to the status register, either by PLP or during a return from interrupt (RTI).

Therefore, it's somewhat difficult to inspect the break flag in order to discern a software interrupt (BRK) from a hardware interrupt (NMI or IRQ) and the mechanism is seldom used. Accessing a break mark put in the extra byte following a BRK instruction is even more cumbersome and probably involves indexed zeropage operations.

Bit 5 (unused) of the status register will be set to 1, whenever the register is pushed to the stack. Bits 5 and 4 will always be ignored, when transferred to the status register.

E.g.,

1)

```
         SR: N V - B D I Z C
             0 0 - - 0 0 1 1

    PHP  ->  0 0 1 1 0 0 1 1  =  $33

    PLP  <-  0 0 - - 0 0 1 1  =  $03

  but:

    PLA  <-  0 0 1 1 0 0 1 1  =  $33
```

2)

```
    LDA #$32 ;00110010

    PHA  ->  0 0 1 1 0 0 1 0  =  $32

    PLP  <-  0 0 - - 0 0 1 0  =  $02
```

3)

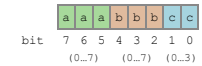
```
    LDA #$C0
    PHA  ->  1 1 0 0 0 0 0 0  =  $C0
    LDA #$08
    PHA  ->  0 0 0 0 1 0 0 0  =  $08
    LDA #$12
    PHA  ->  0 0 0 1 0 0 1 0  =  $12

    RTI
         SR: 0 0 - - 0 0 1 0  =  $02
         PC: $C008
```

Mind that most emulators are displaying the status register (SR or P) in the state as it would be currently pushed to the stack, with bits 4 and 5 on, adding

a bias of $30 to the register value. Here, we chose to rather omit this virtual presence of these bits, since there isn't really a slot for them in the hardware.

6502 Instruction Layout

The 6502 instruction table is laid out according to a pattern *a*-*b*-*c*, where *a* and *b* are an octal number each, followed by a group of two binary digits *c*, as in the bit-vector *"aaabbbcc"*.



Example:
All ROR instructions share *a* = 3 and *c* = 2 (3*b*2) with the address mode in *b*.
At the same time, all instructions addressing the zero-page share *b* = 1 (*a*1*c*).
abc = 312 => (3 << 5 | 1 << 2 | 2) = %011.001.10 = $66 "ROR zpg".

Notably, there are no legal opcodes defined where *c* = 3, accounting for the empty columns in the usual, hexadecimal view of the instruction table.
(For compactness empty rows where *c* = 3 are omitted from the tables below.)

The following table lists the instruction set, rows sorted by *c*, then *a*.

Generally, instructions of a kind are typically found in rows as a combination of *a* and *c*, and address modes are in columns *b*.
However, there are a few exception to this rule, namely, where bits 0 of both *c* and *b* are low (*c* = 0, 2; *b* = 0, 2, 4, 6) and combinations of *c* and *b* select a group of related operations. (E.g., c=0 ∧ b=4: branch, c=0 ∧ b=6: set flag)

c	*a*	*b* = 0	1	2	3	4	5	6	7
0	0	$00 BRK impl		$08 PHP impl		$10 BPL rel		$18 CLC impl	
	1	$20 JSR abs	$24 BIT zpg	$28 PLP impl	$2C BIT abs	$30 BMI rel		$38 SEC impl	
	2	$40 RTI impl		$48 PHA impl	$4C JMP abs	$50 BVC rel		$58 CLI impl	
	3	$60 RTS impl		$68 PLA impl	$6C JMP ind	$70 BVS rel		$78 SEI impl	
	4		$84 STY zpg	$88 DEY impl	$8C STY abs	$90 BCC rel	$94 STY zpg,X	$98 TYA impl	
	5	$A0 LDY #	$A4 LDY zpg	$A8 TAY impl	$AC LDY abs	$B0 BCS rel	$B4 LDY zpg,X	$B8 CLV impl	$BC LDY abs,X
	6	$C0 CPY #	$C4 CPY zpg	$C8 INY impl	$CC CPY abs	$D0 BNE rel		$D8 CLD impl	
	7	$E0 CPX #	$E4 CPX zpg	$E8 INX impl	$EC CPX abs	$F0 BEQ rel		$F8 SED impl	
1	0	$01 ORA X,ind	$05 ORA zpg	$09 ORA #	$0D ORA abs	$11 ORA ind,Y	$15 ORA zpg,X	$19 ORA abs,Y	$1D ORA abs,X
	1	$21 AND X,ind	$25 AND zpg	$29 AND #	$2D AND abs	$31 AND ind,Y	$35 AND zpg,X	$39 AND abs,Y	$3D AND abs,X
	2	$41 EOR X,ind	$45 EOR zpg	$49 EOR #	$4D EOR abs	$51 EOR ind,Y	$55 EOR zpg,X	$59 EOR abs,Y	$5D EOR abs,X

c	*a*	*b* = 0	1	2	3	4	5	6	7
	3	$61 ADC X,ind	$65 ADC zpg	$69 ADC #	$6D ADC abs	$71 ADC ind,Y	$75 ADC zpg,X	$79 ADC abs,Y	$7D ADC abs,X
	4	$81 STA X,ind	$85 STA zpg		$8D STA abs	$91 STA ind,Y	$95 STA zpg,X	$99 STA abs,Y	$9D STA abs,X
	5	$A1 LDA X,ind	$A5 LDA zpg	$A9 LDA #	$AD LDA abs	$B1 LDA ind,Y	$B5 LDA zpg,X	$B9 LDA abs,Y	$BD LDA abs,X
	6	$C1 CMP X,ind	$C5 CMP zpg	$C9 CMP #	$CD CMP abs	$D1 CMP ind,Y	$D5 CMP zpg,X	$D9 CMP abs,Y	$DD CMP abs,X
	7	$E1 SBC X,ind	$E5 SBC zpg	$E9 SBC #	$ED SBC abs	$F1 SBC ind,Y	$F5 SBC zpg,X	$F9 SBC abs,Y	$FD SBC abs,X
2	0		$06 ASL zpg	$0A ASL A	$0E ASL abs		$16 ASL zpg,X		$1E ASL abs,X
	1		$26 ROL zpg	$2A ROL A	$2E ROL abs		$36 ROL zpg,X		$3E ROL abs,X
	2		$46 LSR zpg	$4A LSR A	$4E LSR abs		$56 LSR zpg,X		$5E LSR abs,X
	3		$66 ROR zpg	$6A ROR A	$6E ROR abs		$76 ROR zpg,X		$7E ROR abs,X
	4		$86 STX zpg	$8A TXA impl	$8E STX abs		$96 STX zpg,Y	$9A TXS impl	
	5	$A2 LDX #	$A6 LDX zpg	$AA TAX impl	$AE LDX abs		$B6 LDX zpg,Y	$BA TSX impl	$BE LDX abs,Y
	6		$C6 DEC zpg	$CA DEX impl	$CE DEC abs		$D6 DEC zpg,X		$DE DEC abs,X
	7		$E6 INC zpg	$EA NOP impl	$EE INC abs		$F6 INC zpg,X		$FE INC abs,X

Note: The operand of instructions like "ASL A" is often depicted as implied, as well. Mind that, for any practical reasons, the two notations are interchangeable for any instructions involving the accumulator. — However, there are subtle differences.

A rotated view, rows as combinations of *c* and *b*, and columns as *a*:

c	*b*	*a* = 0	1	2	3	4	5	6	7
0	0	$00 BRK impl	$20 JSR abs	$40 RTI impl	$60 RTS impl		$A0 LDY #	$C0 CPY #	$E0 CPX #
	1		$24 BIT zpg			$84 STY zpg	$A4 LDY zpg	$C4 CPY zpg	$E4 CPX zpg
	2	$08 PHP impl	$28 PLP impl	$48 PHA impl	$68 PLA impl	$88 DEY impl	$A8 TAY impl	$C8 INY impl	$E8 INX impl
	3		$2C BIT abs	$4C JMP abs	$6C JMP ind	$8C STY abs	$AC LDY abs	$CC CPY abs	$EC CPX abs
	4	$10 BPL rel	$30 BMI rel	$50 BVC rel	$70 BVS rel	$90 BCC rel	$B0 BCS rel	$D0 BNE rel	$F0 BEQ rel
	5					$94 STY zpg,X	$B4 LDY zpg,X		
	6	$18 CLC impl	$38 SEC impl	$58 CLI impl	$78 SEI impl	$98 TYA impl	$B8 CLV impl	$D8 CLD impl	$F8 SED impl
	7						$BC LDY abs,X		
1	0	$01 ORA X,ind	$21 AND X,ind	$41 EOR X,ind	$61 ADC X,ind	$81 STA X,ind	$A1 LDA X,ind	$C1 CMP X,ind	$E1 SBC X,ind
	1	$05 ORA zpg	$25 AND zpg	$45 EOR zpg	$65 ADC zpg	$85 STA zpg	$A5 LDA zpg	$C5 CMP zpg	$E5 SBC zpg
	2	$09 ORA #	$29 AND #	$49 EOR #	$69 ADC #		$A9 LDA #	$C9 CMP #	$E9 SBC #
	3	$0D ORA abs	$2D AND abs	$4D EOR abs	$6D ADC abs	$8D STA abs	$AD LDA abs	$CD CMP abs	$ED SBC abs

c	b	a 0	a 1	a 2	a 3	a 4	a 5	a 6	a 7
0	4	$11 ORA ind,Y	$31 AND ind,Y	$51 EOR ind,Y	$71 ADC ind,Y	$91 STA ind,Y	$B1 LDA ind,Y	$D1 CMP ind,Y	$F1 SBC ind,Y
	5	$15 ORA zpg,X	$35 AND zpg,X	$55 EOR zpg,X	$75 ADC zpg,X	$95 STA zpg,X	$B5 LDA zpg,X	$D5 CMP zpg,X	$F5 SBC zpg,X
	6	$19 ORA abs,Y	$39 AND abs,Y	$59 EOR abs,Y	$79 ADC abs,Y	$99 STA abs,Y	$B9 LDA abs,Y	$D9 CMP abs,Y	$F9 SBC abs,Y
	7	$1D ORA abs,X	$3D AND abs,X	$5D EOR abs,X	$7D ADC abs,X	$9D STA abs,X	$BD LDA abs,X	$DD CMP abs,X	$FD SBC abs,X
2	0						$A2 LDX #		
	1	$06 ASL zpg	$26 ROL zpg	$46 LSR zpg	$66 ROR zpg	$86 STX zpg	$A6 LDX zpg	$C6 DEC zpg	$E6 INC zpg
	2	$0A ASL A	$2A ROL A	$4A LSR A	$6A ROR A	$8A TXA impl	$AA TAX impl	$CA DEX impl	$EA NOP impl
	3	$0E ASL abs	$2E ROL abs	$4E LSR abs	$6E ROR abs	$8E STX abs	$AE LDX abs	$CE DEC abs	$EE INC abs
	4								
	5	$16 ASL zpg,X	$36 ROL zpg,X	$56 LSR zpg,X	$76 ROR zpg,X	$96 STX zpg,Y	$B6 LDX zpg,Y	$D6 DEC zpg,X	$F6 INC zpg,X
	6					$9A TXS impl	$BA TSX impl		
	7	$1E ASL abs,X	$3E ROL abs,X	$5E LSR abs,X	$7E ROR abs,X		$BE LDX abs,Y	$DE DEC abs,X	$FE INC abs,X

Finally, a more complex view, the instruction set listed by rows as combinations of *a* and *c*, and *b* in columns:

Address modes are either a property of *b* (even columns) or combinations of *b* and *c* (odd columns with aspecific row-index modulus 3; i.e., every third row in a given column). In those latter columns, first and third rows (*c* = 0 and *c* = 2) refer to the same kind of general operation.

Load, store and transfer instructions as well as comparisons are typically found in the lower half of the table, while most of the arithmetical and logical operations as well as stack and jump instructions are found in the upper half. (However, mind the exception of SBC as a "mirror" of ADC.)

a	c	b 0	b 1	b 2	b 3	b 4	b 5	b 6	b 7
0	0	$00 BRK impl		$08 PHP impl		$10 BPL rel		$18 CLC impl	
	1	$01 ORA X,ind	$05 ORA zpg	$09 ORA #	$0D ORA abs	$11 ORA ind,Y	$15 ORA zpg,X	$19 ORA abs,Y	$1D ORA abs,X
	2		$06 ASL zpg	$0A ASL A	$0E ASL abs		$16 ASL zpg,X		$1E ASL abs,X
1	0	$20 JSR abs	$24 BIT zpg	$28 PLP impl	$2C BIT abs	$30 BMI rel		$38 SEC impl	
	1	$21 AND X,ind	$25 AND zpg	$29 AND #	$2D AND abs	$31 AND ind,Y	$35 AND zpg,X	$39 AND abs,Y	$3D AND abs,X
	2		$26 ROL zpg	$2A ROL A	$2E ROL abs		$36 ROL zpg,X		$3E ROL abs,X
2	0	$40 RTI impl		$48 PHA impl	$4C JMP abs	$50 BVC rel		$58 CLI impl	
	1	$41 EOR X,ind	$45 EOR zpg	$49 EOR #	$4D EOR abs	$51 EOR ind,Y	$55 EOR zpg,X	$59 EOR abs,Y	$5D EOR abs,X

a	c	b 0	b 1	b 2	b 3	b 4	b 5	b 6	b 7
	2		$46 LSR zpg	$4A LSR A	$4E LSR abs		$56 LSR zpg,X		$5E LSR abs,X
3	0	$60 RTS impl		$68 PLA impl	$6C JMP ind	$70 BVS rel		$78 SEI impl	
	1	$61 ADC X,ind	$65 ADC zpg	$69 ADC #	$6D ADC abs	$71 ADC ind,Y	$75 ADC zpg,X	$79 ADC abs,Y	$7D ADC abs,X
	2		$66 ROR zpg	$6A ROR A	$6E ROR abs		$76 ROR zpg,X		$7E ROR abs,X
4	0		$84 STY zpg	$88 DEY impl	$8C STY abs	$90 BCC rel	$94 STY zpg,X	$98 TYA impl	
	1	$81 STA X,ind	$85 STA zpg		$8D STA abs	$91 STA ind,Y	$95 STA zpg,X	$99 STA abs,Y	$9D STA abs,X
	2		$86 STX zpg	$8A TXA impl	$8E STX abs		$96 STX zpg,Y	$9A TXS impl	
5	0	$A0 LDY #	$A4 LDY zpg	$A8 TAY impl	$AC LDY abs	$B0 BCS rel	$B4 LDY zpg,X	$B8 CLV impl	$BC LDY abs,X
	1	$A1 LDA X,ind	$A5 LDA zpg	$A9 LDA #	$AD LDA abs	$B1 LDA ind,Y	$B5 LDA zpg,X	$B9 LDA abs,Y	$BD LDA abs,X
	2	$A2 LDX #	$A6 LDX zpg	$AA TAX impl	$AE LDX abs		$B6 LDX zpg,Y	$BA TSX impl	$BE LDX abs,Y
6	0	$C0 CPY #	$C4 CPY zpg	$C8 INY impl	$CC CPY abs	$D0 BNE rel		$D8 CLD impl	
	1	$C1 CMP X,ind	$C5 CMP zpg	$C9 CMP #	$CD CMP abs	$D1 CMP ind,Y	$D5 CMP zpg,X	$D9 CMP abs,Y	$DD CMP abs,X
	2		$C6 DEC zpg	$CA DEX impl	$CE DEC abs		$D6 DEC zpg,X		$DE DEC abs,X
7	0	$E0 CPX #	$E4 CPX zpg	$E8 INX impl	$EC CPX abs	$F0 BEQ rel		$F8 SED impl	
	1	$E1 SBC X,ind	$E5 SBC zpg	$E9 SBC #	$ED SBC abs	$F1 SBC ind,Y	$F5 SBC zpg,X	$F9 SBC abs,Y	$FD SBC abs,X
	2		$E6 INC zpg	$EA NOP impl	$EE INC abs		$F6 INC zpg,X		$FE INC abs,X

"Illegal" Opcodes Revisited

So, how do the "illegal" opcodes fit into this decoding scheme?
Let's have a look — "illegals" are shown on grey background.

The first view — rows by *c* and *a* and columns as *b* — reveals a strict relation beetween address modes and columns:

c	a	b 0	b 1	b 2	b 3	b 4	b 5	b 6	b 7
0	0	$00 BRK impl	$04 NOP zpg	$08 PHP impl	$0C NOP abs	$10 BPL rel	$14 NOP zpg,X	$18 CLC impl	$1C NOP abs,X
	1	$20 JSR abs	$24 BIT zpg	$28 PLP impl	$2C BIT abs	$30 BMI rel	$34 NOP zpg,X	$38 SEC impl	$3C NOP abs,X
	2	$40 RTI impl	$44 NOP zpg	$48 PHA impl	$4C JMP abs	$50 BVC rel	$54 NOP zpg,X	$58 CLI impl	$5C NOP abs,X
	3	$60 RTS impl	$64 NOP zpg	$68 PLA impl	$6C JMP ind	$70 BVS rel	$74 NOP zpg,X	$78 SEI impl	$7C NOP abs,X
	4	$80 NOP #	$84 STY zpg	$88 DEY impl	$8C STY abs	$90 BCC rel	$94 STY zpg,X	$98 TYA impl	$9C SHY abs,X
	5	$A0 LDY #	$A4 LDY zpg	$A8 TAY impl	$AC LDY abs	$B0 BCS rel	$B4 LDY zpg,X	$B8 CLV impl	$BC LDY abs,X
	6	$C0 CPY #	$C4 CPY zpg	$C8 INY impl	$CC CPY abs	$D0 BNE rel	$D4 NOP zpg,X	$D8 CLD impl	$DC NOP abs,X

c	a	b 0	1	2	3	4	5	6	7
	7	$E0 CPX #	$E4 CPX zpg	$E8 INX impl	$EC CPX abs	$F0 BEQ rel	$F4 NOP zpg,X	$F8 SED impl	$FC NOP abs,X
1	0	$01 ORA X,ind	$05 ORA zpg	$09 ORA #	$0D ORA abs	$11 ORA ind,Y	$15 ORA zpg,X	$19 ORA abs,Y	$1D ORA abs,X
	1	$21 AND X,ind	$25 AND zpg	$29 AND #	$2D AND abs	$31 AND ind,Y	$35 AND zpg,X	$39 AND abs,Y	$3D AND abs,X
	2	$41 EOR X,ind	$45 EOR zpg	$49 EOR #	$4D EOR abs	$51 EOR ind,Y	$55 EOR zpg,X	$59 EOR abs,Y	$5D EOR abs,X
	3	$61 ADC X,ind	$65 ADC zpg	$69 ADC #	$6D ADC abs	$71 ADC ind,Y	$75 ADC zpg,X	$79 ADC abs,Y	$7D ADC abs,X
	4	$81 STA X,ind	$85 STA zpg	$89 NOP #	$8D STA abs	$91 STA ind,Y	$95 STA zpg,X	$99 STA abs,Y	$9D STA abs,X
	5	$A1 LDA X,ind	$A5 LDA zpg	$A9 LDA #	$AD LDA abs	$B1 LDA ind,Y	$B5 LDA zpg,X	$B9 LDA abs,Y	$BD LDA abs,X
	6	$C1 CMP X,ind	$C5 CMP zpg	$C9 CMP #	$CD CMP abs	$D1 CMP ind,Y	$D5 CMP zpg,X	$D9 CMP abs,Y	$DD CMP abs,X
	7	$E1 SBC X,ind	$E5 SBC zpg	$E9 SBC #	$ED SBC abs	$F1 SBC ind,Y	$F5 SBC zpg,X	$F9 SBC abs,Y	$FD SBC abs,X
2	0	$02 JAM	$06 ASL zpg	$0A ASL A	$0E ASL abs	$12 JAM	$16 ASL zpg,X	$1A NOP impl	$1E ASL abs,X
	1	$22 JAM	$26 ROL zpg	$2A ROL A	$2E ROL abs	$32 JAM	$36 ROL zpg,X	$3A NOP impl	$3E ROL abs,X
	2	$42 JAM	$46 LSR zpg	$4A LSR A	$4E LSR abs	$52 JAM	$56 LSR zpg,X	$5A NOP impl	$5E LSR abs,X
	3	$62 JAM	$66 ROR zpg	$6A ROR A	$6E ROR abs	$72 JAM	$76 ROR zpg,X	$7A NOP impl	$7E ROR abs,X
	4	$82 NOP #	$86 STX zpg	$8A TXA impl	$8E STX abs	$92 JAM	$96 STX zpg,Y	$9A TXS impl	$9E SHX abs,Y
	5	$A2 LDX #	$A6 LDX zpg	$AA TAX impl	$AE LDX abs	$B2 JAM	$B6 LDX zpg,Y	$BA TSX impl	$BE LDX abs,Y
	6	$C2 NOP #	$C6 DEC zpg	$CA DEX impl	$CE DEC abs	$D2 JAM	$D6 DEC zpg,X	$DA NOP impl	$DE DEC abs,X
	7	$E2 NOP #	$E6 INC zpg	$EA NOP impl	$EE INC abs	$F2 JAM	$F6 INC zpg,X	$FA NOP impl	$FE INC abs,X
3	0	$03 SLO X,ind	$07 SLO zpg	$0B ANC #	$0F SLO abs	$13 SLO ind,Y	$17 SLO zpg,X	$1B SLO abs,Y	$1F SLO abs,X
	1	$23 RLA X,ind	$27 RLA zpg	$2B ANC #	$2F RLA abs	$33 RLA ind,Y	$37 RLA zpg,X	$3B RLA abs,Y	$3F RLA abs,X
	2	$43 SRE X,ind	$47 SRE zpg	$4B ALR #	$4F SRE abs	$53 SRE ind,Y	$57 SRE zpg,X	$5B SRE abs,Y	$5F SRE abs,X
	3	$63 RRA X,ind	$67 RRA zpg	$6B ARR #	$6F RRA abs	$73 RRA ind,Y	$77 RRA zpg,X	$7B RRA abs,Y	$7F RRA abs,X
	4	$83 SAX X,ind	$87 SAX zpg	$8B ANE #	$8F SAX abs	$93 SHA ind,Y	$97 SAX zpg,Y	$9B TAS abs,Y	$9F SHA abs,Y
	5	$A3 LAX X,ind	$A7 LAX zpg	$AB LXA #	$AF LAX abs	$B3 LAX ind,Y	$B7 LAX zpg,Y	$BB LAS abs,Y	$BF LAX abs,Y
	6	$C3 DCP X,ind	$C7 DCP zpg	$CB SBX #	$CF DCP abs	$D3 DCP ind,Y	$D7 DCP zpg,X	$DB DCP abs,Y	$DF DCP abs,X
	7	$E3 ISC X,ind	$E7 ISC zpg	$EB USBC #	$EF ISC abs	$F3 ISC ind,Y	$F7 ISC zpg,X	$FB ISC abs,Y	$FF ISC abs,X

And, again, as a rotated view, rows as combinations of *c* and *b*, and columns as *a*.
We may observe a close relationship between the legal and the undocumented instructions in the vertical (quarter-)segements of each column.

c	b	a 0	1	2	3	4	5	6	7
0	0	$00 BRK impl	$20 JSR abs	$40 RTI impl	$60 RTS impl	$80 NOP #	$A0 LDY #	$C0 CPY #	$E0 CPX #

c	b	a 0	1	2	3	4	5	6	7
	1	$04 NOP zpg	$24 BIT zpg	$44 NOP zpg	$64 NOP zpg	$84 STY zpg	$A4 LDY zpg	$C4 CPY zpg	$E4 CPX zpg
	2	$08 PHP impl	$28 PLP impl	$48 PHA impl	$68 PLA impl	$88 DEY impl	$A8 TAY impl	$C8 INY impl	$E8 INX impl
	3	$0C NOP abs	$2C BIT abs	$4C JMP abs	$6C JMP ind	$8C STY abs	$AC LDY abs	$CC CPY abs	$EC CPX abs
	4	$10 BPL rel	$30 BMI rel	$50 BVC rel	$70 BVS rel	$90 BCC rel	$B0 BCS rel	$D0 BNE rel	$F0 BEQ rel
	5	$14 NOP zpg,X	$34 NOP zpg,X	$54 NOP zpg,X	$74 NOP zpg,X	$94 STY zpg,X	$B4 LDY zpg,X	$D4 NOP zpg,X	$F4 NOP zpg,X
	6	$18 CLC impl	$38 SEC impl	$58 CLI impl	$78 SEI impl	$98 TYA impl	$B8 CLV impl	$D8 CLD impl	$F8 SED impl
	7	$1C NOP abs,X	$3C NOP abs,X	$5C NOP abs,X	$7C NOP abs,X	$9C SHY abs,X	$BC LDY abs,X	$DC NOP abs,X	$FC NOP abs,X
1	0	$01 ORA X,ind	$21 AND X,ind	$41 EOR X,ind	$61 ADC X,ind	$81 STA X,ind	$A1 LDA X,ind	$C1 CMP X,ind	$E1 SBC X,ind
	1	$05 ORA zpg	$25 AND zpg	$45 EOR zpg	$65 ADC zpg	$85 STA zpg	$A5 LDA zpg	$C5 CMP zpg	$E5 SBC zpg
	2	$09 ORA #	$29 AND #	$49 EOR #	$69 ADC #	$89 NOP #	$A9 LDA #	$C9 CMP #	$E9 SBC #
	3	$0D ORA abs	$2D AND abs	$4D EOR abs	$6D ADC abs	$8D STA abs	$AD LDA abs	$CD CMP abs	$ED SBC abs
	4	$11 ORA ind,Y	$31 AND ind,Y	$51 EOR ind,Y	$71 ADC ind,Y	$91 STA ind,Y	$B1 LDA ind,Y	$D1 CMP ind,Y	$F1 SBC ind,Y
	5	$15 ORA zpg,X	$35 AND zpg,X	$55 EOR zpg,X	$75 ADC zpg,X	$95 STA zpg,X	$B5 LDA zpg,X	$D5 CMP zpg,X	$F5 SBC zpg,X
	6	$19 ORA abs,Y	$39 AND abs,Y	$59 EOR abs,Y	$79 ADC abs,Y	$99 STA abs,Y	$B9 LDA abs,Y	$D9 CMP abs,Y	$F9 SBC abs,Y
	7	$1D ORA abs,X	$3D AND abs,X	$5D EOR abs,X	$7D ADC abs,X	$9D STA abs,X	$BD LDA abs,X	$DD CMP abs,X	$FD SBC abs,X
2	0	$02 JAM	$22 JAM	$42 JAM	$62 JAM	$82 NOP #	$A2 LDX #	$C2 NOP #	$E2 NOP #
	1	$06 ASL zpg	$26 ROL zpg	$46 LSR zpg	$66 ROR zpg	$86 STX zpg	$A6 LDX zpg	$C6 DEC zpg	$E6 INC zpg
	2	$0A ASL A	$2A ROL A	$4A LSR A	$6A ROR A	$8A TXA impl	$AA TAX impl	$CA DEX impl	$EA NOP impl
	3	$0E ASL abs	$2E ROL abs	$4E LSR abs	$6E ROR abs	$8E STX abs	$AE LDX abs	$CE DEC abs	$EE INC abs
	4	$12 JAM	$32 JAM	$52 JAM	$72 JAM	$92 JAM	$B2 JAM	$D2 JAM	$F2 JAM
	5	$16 ASL zpg,X	$36 ROL zpg,X	$56 LSR zpg,X	$76 ROR zpg,X	$96 STX zpg,Y	$B6 LDX zpg,Y	$D6 DEC zpg,X	$F6 INC zpg,X
	6	$1A NOP impl	$3A NOP impl	$5A NOP impl	$7A NOP impl	$9A TXS impl	$BA TSX impl	$DA NOP impl	$FA NOP impl
	7	$1E ASL abs,X	$3E ROL abs,X	$5E LSR abs,X	$7E ROR abs,X	$9E SHX abs,Y	$BE LDX abs,Y	$DE DEC abs,X	$FE INC abs,X
3	0	$03 SLO X,ind	$23 RLA X,ind	$43 SRE X,ind	$63 RRA X,ind	$83 SAX X,ind	$A3 LAX X,ind	$C3 DCP X,ind	$E3 ISC X,ind
	1	$07 SLO zpg	$27 RLA zpg	$47 SRE zpg	$67 RRA zpg	$87 SAX zpg	$A7 LAX zpg	$C7 DCP zpg	$E7 ISC zpg
	2	$0B ANC #	$2B ANC #	$4B ALR #	$6B ARR #	$8B ANE #	$AB LXA #	$CB SBX #	$EB USBC #
	3	$0F SLO abs	$2F RLA abs	$4F SRE abs	$6F RRA abs	$8F SAX abs	$AF LAX abs	$CF DCP abs	$EF ISC abs
	4	$13 SLO ind,Y	$33 RLA ind,Y	$53 SRE ind,Y	$73 RRA ind,Y	$93 SHA ind,Y	$B3 LAX ind,Y	$D3 DCP ind,Y	$F3 ISC ind,Y
	5	$17 SLO zpg,X	$37 RLA zpg,X	$57 SRE zpg,X	$77 RRA zpg,X	$97 SAX zpg,Y	$B7 LAX zpg,Y	$D7 DCP zpg,X	$F7 ISC zpg,X
	6	$1B SLO abs,Y	$3B RLA abs,Y	$5B SRE abs,Y	$7B RRA abs,Y	$9B TAS abs,Y	$BB LAS abs,Y	$DB DCP abs,Y	$FB ISC abs,Y
	7	$1F SLO abs,X	$3F RLA abs,X	$5F SRE abs,X	$7F RRA abs,X	$9F SHA abs,Y	$BF LAX abs,Y	$DF DCP abs,X	$FF ISC abs,X

And, finally, in a third view, we may observe how each of the rows of "illegal" instructions at *c = 3* inherits behavior from the two rows with *c = 1* and *c = 2* immediately above, combining the operations of these instructions with the address mode of the respective instruction at *c = 1*.
(Mind that in binary 3 is the combination of 2 and 1, bits 0 and 1 both set.)

We may further observe that additional NOPs result from non-effective or non-sensical combinations of operations and address modes, e.g., instr. $89, which would be "STA #", storing the contents of the accumulator in the operand. Some other instructions, typically combinations involving indirect indexed addressing, fail over unresolved timing issues entirely, resulting in a "JAM".

(We me also observe that there is indeed a difference in accumulator mode — as in "OPC A" — and immediate addressing. E.g., $6A, "ROR A", is a valid instruction, while instruction $7A, "ROR implied", is a NOP.
We may also note how "ROR X,ind" at $62 and "ROR ind,Y" at $72 fail entirely and result in a JAM.)

a	*c*	*b* = 0	1	2	3	4	5	6	7
0	0	$00 BRK impl	$04 NOP zpg	$08 PHP impl	$0C NOP abs	$10 BPL rel	$14 NOP zpg,X	$18 CLC impl	$1C NOP abs,X
	1	$01 ORA X,ind	$05 ORA zpg	$09 ORA #	$0D ORA abs	$11 ORA ind,Y	$15 ORA zpg,X	$19 ORA abs,Y	$1D ORA abs,X
	2	$02 JAM	$06 ASL zpg	$0A ASL A	$0E ASL abs	$12 JAM	$16 ASL zpg,X	$1A NOP impl	$1E ASL abs,X
	3	$03 SLO X,ind	$07 SLO zpg	$0B ANC #	$0F SLO abs	$13 SLO ind,Y	$17 SLO zpg,X	$1B SLO abs,Y	$1F SLO abs,X
1	0	$20 JSR abs	$24 BIT zpg	$28 PLP impl	$2C BIT abs	$30 BMI rel	$34 NOP zpg,X	$38 SEC impl	$3C NOP abs,X
	1	$21 AND X,ind	$25 AND zpg	$29 AND #	$2D AND abs	$31 AND ind,Y	$35 AND zpg,X	$39 AND abs,Y	$3D AND abs,X
	2	$22 JAM	$26 ROL zpg	$2A ROL A	$2E ROL abs	$32 JAM	$36 ROL zpg,X	$3A NOP impl	$3E ROL abs,X
	3	$23 RLA X,ind	$27 RLA zpg	$2B ANC #	$2F RLA abs	$33 RLA ind,Y	$37 RLA zpg,X	$3B RLA abs,Y	$3F RLA abs,X
2	0	$40 RTI impl	$44 NOP zpg	$48 PHA impl	$4C JMP abs	$50 BVC rel	$54 NOP zpg,X	$58 CLI impl	$5C NOP abs,X
	1	$41 EOR X,ind	$45 EOR zpg	$49 EOR #	$4D EOR abs	$51 EOR ind,Y	$55 EOR zpg,X	$59 EOR abs,Y	$5D EOR abs,X
	2	$42 JAM	$46 LSR zpg	$4A LSR A	$4E LSR abs	$52 JAM	$56 LSR zpg,X	$5A NOP impl	$5E LSR abs,X
	3	$43 SRE X,ind	$47 SRE zpg	$4B ALR #	$4F SRE abs	$53 SRE ind,Y	$57 SRE zpg,X	$5B SRE abs,Y	$5F SRE abs,X
3	0	$60 RTS impl	$64 NOP zpg	$68 PLA impl	$6C JMP ind	$70 BVS rel	$74 NOP zpg,X	$78 SEI impl	$7C NOP abs,X
	1	$61 ADC X,ind	$65 ADC zpg	$69 ADC #	$6D ADC abs	$71 ADC ind,Y	$75 ADC zpg,X	$79 ADC abs,Y	$7D ADC abs,X
	2	$62 JAM	$66 ROR zpg	$6A ROR A	$6E ROR abs	$72 JAM	$76 ROR zpg,X	$7A NOP impl	$7E ROR abs,X
	3	$63 RRA X,ind	$67 RRA zpg	$6B ARR #	$6F RRA abs	$73 RRA ind,Y	$77 RRA zpg,X	$7B RRA abs,Y	$7F RRA abs,X
4	0	$80 NOP #	$84 STY zpg	$88 DEY impl	$8C STY abs	$90 BCC rel	$94 STY zpg,X	$98 TYA impl	$9C SHY abs,X
	1	$81 STA X,ind	$85 STA zpg	$89 NOP #	$8D STA abs	$91 STA ind,Y	$95 STA zpg,X	$99 STA abs,Y	$9D STA abs,X
	2	$82 NOP #	$86 STX zpg	$8A TXA impl	$8E STX abs	$92 JAM	$96 STX zpg,Y	$9A TXS impl	$9E SHX abs,Y
	3	$83 SAX X,ind	$87 SAX zpg	$8B ANE #	$8F SAX abs	$93 SHA ind,Y	$97 SAX zpg,Y	$9B TAS abs,Y	$9F SHA abs,Y

a	*c*	*b* = 0	1	2	3	4	5	6	7
5	0	$A0 LDY #	$A4 LDY zpg	$A8 TAY impl	$AC LDY abs	$B0 BCS rel	$B4 LDY zpg,X	$B8 CLV impl	$BC LDY abs,X
	1	$A1 LDA X,ind	$A5 LDA zpg	$A9 LDA #	$AD LDA abs	$B1 LDA ind,Y	$B5 LDA zpg,X	$B9 LDA abs,Y	$BD LDA abs,X
	2	$A2 LDX #	$A6 LDX zpg	$AA TAX impl	$AE LDX abs	$B2 JAM	$B6 LDX zpg,Y	$BA TSX impl	$BE LDX abs,Y
	3	$A3 LAX X,ind	$A7 LAX zpg	$AB LXA #	$AF LAX abs	$B3 LAX ind,Y	$B7 LAX zpg,Y	$BB LAS abs,Y	$BF LAX abs,Y
6	0	$C0 CPY #	$C4 CPY zpg	$C8 INY impl	$CC CPY abs	$D0 BNE rel	$D4 NOP zpg,X	$D8 CLD impl	$DC NOP abs,X
	1	$C1 CMP X,ind	$C5 CMP zpg	$C9 CMP #	$CD CMP abs	$D1 CMP ind,Y	$D5 CMP zpg,X	$D9 CMP abs,Y	$DD CMP abs,X
	2	$C2 NOP #	$C6 DEC zpg	$CA DEX impl	$CE DEC abs	$D2 JAM	$D6 DEC zpg,X	$DA NOP impl	$DE DEC abs,X
	3	$C3 DCP X,ind	$C7 DCP zpg	$CB SBX #	$CF DCP abs	$D3 DCP ind,Y	$D7 DCP zpg,X	$DB DCP abs,Y	$DF DCP abs,X
7	0	$E0 CPX #	$E4 CPX zpg	$E8 INX impl	$EC CPX abs	$F0 BEQ rel	$F4 NOP zpg,X	$F8 SED impl	$FC NOP abs,X
	1	$E1 SBC X,ind	$E5 SBC zpg	$E9 SBC #	$ED SBC abs	$F1 SBC ind,Y	$F5 SBC zpg,X	$F9 SBC abs,Y	$FD SBC abs,X
	2	$E2 NOP #	$E6 INC zpg	$EA NOP impl	$EE INC abs	$F2 JAM	$F6 INC zpg,X	$FA NOP impl	$FE INC abs,X
	3	$E3 ISC X,ind	$E7 ISC zpg	$EB USBC #	$EF ISC abs	$F3 ISC ind,Y	$F7 ISC zpg,X	$FB ISC abs,Y	$FF ISC abs,X

As a final observation, the two highly unstable instructions "ANE" (XAA) and "LXA" (LAX immediate) involving a "magic constant" are both combinations of an accumulator operation and an inter-register transfer between the accumulator and the X register:

```
$8B (a=5, c=3, b=2): ANE # = STA # (NOP) + TXA
(A OR CONST) AND X AND oper -> A

$AB (a=4, c=3, b=2): LXA # = LDA # + TAX
(A OR CONST) AND oper -> A -> X
```

In the case of ANE, the contents of the accumulator is put on the internal data lines at the same time as the contents of the X-register, while there's also the operand read for the immediate operation, with the result transferred to the accumulator.

In the case of LXA, the immediate operand and the contents of the accumulator are competing for the imput lines, while the result will be transferred to both the accumulator and the X register.

The outcome of these competing, noisy conditions depends on the production series of the chip, and maybe even on environmental conditions. This effects in an OR-ing of the accumulator with the "magic constant" combined with an AND-ing of the competing inputs. The final transfer to the target register(s) then seems to work as may be expected.

(This AND-ing of competing values susggests that the 6502 is working internally in active negative logic, where all data lines are first set to high and then cleared for any zero bits. This also suggests that the "magic constant" stands merely for a partial transfer of the contents of the accumulator.)

Much of this also applies to "TAS" (XAS, SHS), $9B, but here the extra cycles for indexed addressing seem to contribute to the conflict being resolved without this "magic constant". However, TAS is still unstable.

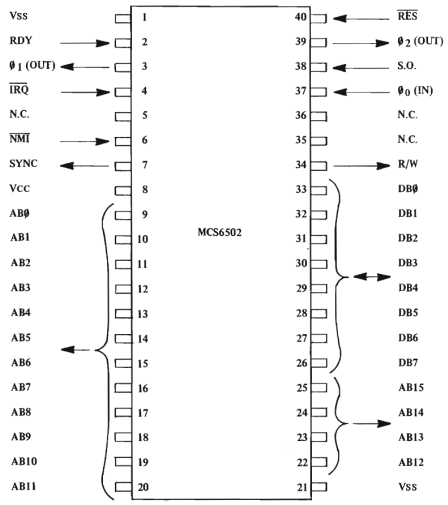
Simlarly the peculiar group involving the high-byte of the provided address + 1 (as in "H+1") — SHA (AHX, AXA), SHX (A11, SXA, XAS), SHY (A11, SYA, SAY) — involves a conflict of an attempt to store the accumulator and another register being put on the data lines at the same time, and the operations required to determine the target address for for indexed addressing. Again, the competing values are AND-ed and the instructions are unstable.

We may also observe that SHY is really the unimplemented instruction "STY abs,X" and SHX is "STX abs,Y" with SHA being the combination of "LDA abs,X" and SHX.

We may conclude that these "illegal opcodes" or "undocumented instructions" are really a text-book example of undefined behavior for undefined input patterns. Generally speaking, for any instructions xxxxxx11 *(c=3)* both instructions at xxxxxx01 *(c=1)* and xxxxxx10 *(c=2)* are started in a thread, with competing ouput values on the internal data lines AND-ed. For some combinations, this results in a fragile race condition, while others are showing mostly stable behavior. The addressing mode is generally determined by that of the instruction at *c=1*.

(It may be interesting that is doesn't matter, if any of the two threads jams, as long as the timing for the other thread resolves. So there is no "JAM" instruction at *c=3*.)

Pinout (NMOS 6502)



Vcc, Vss	supply voltage (Vcc: +5 V DC ± 5%, Vss: max. +7 V DC)
$\Phi_{0...2}$	clock
AB0-AB15	address bus
DB0-DB7	data bus
R/W	read/write
RDY	ready
S.O.	set overflow (future I/O interface)
SYNC	sync (goes high on opcode fetch phase)
IRQ	interrupt request (active low)
NMI	non maskable interrupt (active low)
RES	reset (active low)
N.C.	no connection

Source: MCS6500 Microcomputer Family Hardware Manual. MOS Technology, Inc., 1976.

The 65xx-Family:

Type	Features, Comments
6502	NMOS, 16 bit address bus, 8 bit data bus
6502A	accelerated version of 6502
6502C	accelerated version of 6502, additional halt pin, CMOS
65C02	WDC version, additional instructions and address modes, up to 14MHz
6503, 6505, 6506	12 bit address bus [4 KiB]
6504	13 bit address bus [8 KiB], no NMI
6507	13 bit address bus [8 KiB], no interrupt lines
6509	20 bit address bus [1 MiB] by bankswitching
6510	as 6502 with additional 6 bit I/O-port
6511	integrated micro controler with I/O-port, serial interface, and RAM (Rockwell)
65F11	as 6511, integrated FORTH interpreter
7501	as 6502, HMOS
8500	as 6510, CMOS
8502	as 6510 with switchable 2 MHz option, 7 bit I/O-port
65816 (65C816)	16 bit registers and ALU, 24 bit address bus [16 MiB], up to 24 MHz (Western Design Center)
65802 (65C802)	as 65816, pin compatible to 6502, 64 KiB address bus, up to 16 MHz

Site Notes

Disclaimer

See also the “Virtual 6502” suite of online-programs

>> Virtual 6502 (6502/6510 emulator)
>> 6502 Assembler
>> 6502 Disassembler

External Links

>> 6502.org — the 6502 microprocessor resource
>> visual6502.org — visual transistor-level simulation of the 6502 CPU
>> The Western Design Center, Inc. — designers of the 6502 (still thriving)

Presented by mass:werk.