NMOS 6510 Unintended Opcodes

no more secrets

(v0.93 - 24/12/18)

(w) 2013-2018 groepaz/solution, all rights reversed

Contents

Contents

Contents

Preface

'Back in the days' so called 'illegal' opcodes were researched independently by different parties, and detail knowledge about them was considered 'black magic' for many conventional programmers. They first appeared in the context of copy protection schemes, so keeping the knowledge secret was crucial.

When some time later some of these opcodes were documented by various book authors and magazines, a lot of misinformation was spread and a number of weird myths were born. It took another few years until some brave souls started to systematically investigate each and every opcode, and until the mid 90s that Wolfgang Lorenz came up with his test suite that finally contained elaborated test programs for them.

Still, a few opcodes were considered witchcraft for a while (the so called 'unstable' ones), until other people finally de-capped an actual CPU and solved the remaining riddles.

This document tries to present the current state of the art in a readable form, and is in large parts the result of pasting existing documents together and editing them (see References)

24/12/18 groepaz/solution

Scope of this Document

To make things simple, the rest of this document refers specifically to the MOS6510 (and the CSG8500) in the Commodore 64, and to the CSG8502 found in the Commodore 128.

However, most of the document applies to MOS6502 as well. Also MOS Technology licensed Rockwell and Synertek to second source the 6502 microprocessor and support components, meaning they used the same masks for manufacturing, so their chips should behave (exactly) the same. The 6502C “Sandy” found in Atari 8-bit computers also seems to work the same.

Some of the 'unstable' opcodes are known to work slightly different on 6502 equipped machines, but that is just the result of the RDY line not being used in them.

This document does **not** apply to the 65C02, 652SC02, 65CE02, 65816 etc. (These are all not 100% 6502 compatible)

Whether related CPUs like the 7501/8501 used in the CBM264 series behaves the same has not been tested (but is likely – feedback welcomed).

Intended Audience

This document is not for beginners (such as yourself) *. The reader should be familiar with 6502 assembly, and in particular is expected to know how the regular opcodes and CPU flags work exactly. For those that do not feel confident enough, having a reference to the regular opcodes, flags behaviour and things like decimal mode at hand is probably highly recommended.

*) Wording change suggested by Poopmaster

License

This documentation is free as in free beer. All rights reversed.

If using the information contained here results in ultra realistic smoke effects and/or loss of mental health, it is entirely your fault. ***You have been warned.***

What you get

- Reference chart of all 'illegal' opcodes
- Cycle by cycle breakdown of the 'illegal' addressing modes
- For every 'illegal' opcode:
  - Formal description of each opcode, including flags etc.
  - General description of operation and eventual quirks
  - equivalent 'legal' code
  - All documented behaviour backed up by test code. The referenced test code can be found in the VICE test-programs repository at

    http://vice-emu.svn.sourceforge.net/viewvc/vice-emu/testprogs/
  - examples for real world usage, if available

Naming Conventions

A	Accumulator
X	X-register
Y	Y-register
SP	Stack-pointer
PC	Program Counter
NV-BDIZC	Flags in the status-register
{imm}	An immediate value
{addr}	Effective address given in the opcode (including indexing)
{H+1}	High byte of the address given in the opcode, plus 1
{CONST}	'Magic' chip and/or temperature dependent constant value
&	Binary AND
|	Binary OR
^	Binary XOR
+	Integer Addition
-	Integer Substraction
*	Integer Multiplication (powers of two work like a bitshift)
/	Integer Division (powers of two work like a bitshift)

In the various tables colours GREEN, YELLOW and RED are used in the following way:

GREEN indicates all completely stable opcodes, which can be used without special precautions, YELLOW marks partially unstable opcodes which need some special care and RED is reserved for the remaining few which are highly unstable and can only be used with severe restrictions.

Address-Mode Abbreviations

AA	Absolute Address
AAH	Absolute Address High
AAL	Absolute Address Low
DO	Direct Offset

Mnemonics

This document lists all previously used mnemonics for each opcode in the headlines of their description, and then one variant which the author was most familiar with is used throughout the rest of the text. A table that shows which mnemonics are supported by some popular assemblers can be found in the appendix.

Processor Flags

Standard notation is used for the processor flags:

N	**N**egative
V	o**V**erflow
-	*bit5 of the status register is unused*
B	**B**reak
D	**D**ecimal
I	**I**nterrupt
Z	**Z**ero
C	**C**arry

To indicate what processor flags are used and/or modified by the respective instructions this document uses a slightly different notation than many other existing ones. In particular this will allow to indicate directly in the tables whether an instruction depends on, modifies, or just sets a flag.

i	The instruction depends on this flag (takes it as INPUT) but does not change it. In this document this applies to the decimal flag only.
o	The instruction does not depend on this flag, but does set or clear it (it is OUTPUT only). The zero flag is a typical example for this (only branches depend on it, other instruction would only set it)
x	The instruction depends on this flag, and does change it too. The carry flag is a typical example for this (although not generally in all instructions).
	The instruction does not depend on, nor change, this flag

Unintended Opcodes

Overview

Opc.	imp	imm	zp	zpx	zpy	izx	izy	abs	abx	aby	Function	N	V	-	B	D	I	Z	C
SLO			$07	$17		$03	$13	$0F	$1F	$1B	{addr} = {addr} * 2 A = A or {addr}	o						o	o
RLA			$27	$37		$23	$33	$2F	$3F	$3B	{addr} = rol {addr} A = A and {addr}	o						o	x
SRE			$47	$57		$43	$53	$4F	$5F	$5B	{addr} = {addr} / 2 A = A eor {addr}	o						o	o
RRA			$67	$77		$63	$73	$6F	$7F	$7B	{addr} = ror {addr} A = A adc {addr}	o	o			i		o	x
SAX			$87		$97	$83		$8F			{addr} = A & X								
LAX			$A7		$B7	$A3	$B3	$AF		$BF	A,X = {addr}	o						o	
DCP			$C7	$D7		$C3	$D3	$CF	$DF	$DB	{addr} = {addr} - 1 A cmp {addr}	o						o	o
ISC			$E7	$F7		$E3	$F3	$EF	$FF	$FB	{addr} = {addr} + 1 A = A - {addr}	o	o			i		o	x
ANC		$0B									A = A & #{imm}	o						o	o
ANC		$2B									A = A & #{imm}	o						o	o
ALR		$4B									A = (A & #{imm}) / 2	o						o	o
ARR		$6B									A = (A & #{imm}) / 2	o	o			i		o	o
SBX		$CB									X = A & X - #{imm}	o						o	o
SBC		$EB									A = A - #{imm}	o	o			i		o	x
SHA							$93			$9F	{addr} = A & X & {H+1}								
SHY									$9C		{addr} = Y & {H+1}								
SHX										$9E	{addr} = X & {H+1}								
TAS										$9B	SP = A & X {addr} = SP & {H+1}								
LAS										$BB	A,X,SP = {addr} & SP	o						o	
LAX		$AB									A,X = (A | CONST) & #{imm}	o						o	
ANE		$8B									A = (A | CONST) & X & #{imm}	o						o	

Opc.	imp	imm	zp	zpx	zpy	izx	izy	abs	abx	aby	Function	N	V	-	B	D	I	Z	C
NOP	$1A	$80	$04	$14				$0C	$1C		No effect								
NOP	$3A	$82	$44	$34					$3C		No effect								
NOP	$5A	$C2	$64	$54					$5C		No effect								
NOP	$7A	$E2		$74					$7C		No effect								
NOP	$DA	$89		$D4					$DC		No effect								
NOP	$FA			$F4					$FC		No effect								

Opc.	-	-	-	-	-	-	-	-	-	-	-	-	Function	N	V	-	B	D	I	Z	C
JAM	$02	$12	$22	$32	$42	$52	$62	$72	$92	$B2	$D2	$F2	CPU lock-up								

Types

Combinations of two operations with the same addressing mode

Opcode	Function
SLO {addr}	ASL {addr} + ORA {addr}
RLA {addr}	ROL {addr} + AND {addr}
SRE {addr}	LSR {addr} + EOR {addr}
RRA {addr}	ROR {addr} + ADC {addr}
SAX {addr}	STA {addr} + STX {addr} store A & X into {addr}
LAX {addr}	LDA {addr} + LDX {addr}
DCP {addr}	DEC {addr} + CMP {addr}
ISC {addr}	INC {addr} + SBC {addr}

Combinations of an immediate and an implied command

Opcode	Function
ANE #{imm}	TXA + AND #{imm}
LAX #{imm}	LDA #{imm} + TAX
ANC #{imm}	AND #{imm} + (ASL)
ANC #{imm}	AND #{imm} + (ROL)
ALR #{imm}	AND #{imm} + LSR
ARR #{imm}	AND #{imm} + ROR
SBX #{imm}	CMP #{imm} + DEX put A & X minus #{imm} into X
SBC #{imm}	SBC #{imm} + NOP

Combinations of STA/STX/STY

Opcode	Function
SHA {addr}	stores A & X & H into {addr}
SHX {addr}	stores X & H into {addr}
SHY {addr}	stores Y & H into {addr}

Combinations of STA/TXS and LDA/TSX

Opcode	Function
TAS {addr}	stores A & X into SP and A & X & H into {addr}
LAS {addr}	stores {addr} & SP into A, X and SP

No effect

Bit configuration does not allow any operation on these ones:

Opcode	Function
NOP	no effect
NOP #{imm}	Fetches #{imm} but has no effects.
NOP {addr}	Fetches {addr} but has no effects.

Lock-up

Opcode	Function
JAM	Halt the CPU. The buses will be set to $FF.

Stable Opcodes

SLO (ASO)

Type: Combination of two operations with the same addressing mode (Sub-instructions: ASL, ORA)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$07	SLO zp	{addr} = {addr} * 2 A = A or {addr}	2	5	o						o	o
$17	SLO zp, x		2	6	o						o	o
$03	SLO (zp, x)		2	8	o						o	o
$13	SLO (zp), y		2	8	o						o	o
$0F	SLO abs		3	6	o						o	o
$1F	SLO abs, x		3	7	o						o	o
$1B	SLO abs, y		3	7	o						o	o

Operation: Shift left one bit in memory, then OR accumulator with memory.

Example:

```
SLO $C010      ;0F 10 C0
```

Equivalent Instructions:

```
ASL $C010
ORA $C010
```

Test code: `Lorenz-2.15/asoa.prg, Lorenz-2.15/asoax.prg, Lorenz-2.15/asoay.prg, Lorenz-2.15/asoix.prg, Lorenz-2.15/asoiy.prg, Lorenz-2.15/asoz.prg, Lorenz-2.15/asozx.prg`

Example: Multibyte left shift and load leftmost byte

Instead of:

```
          ASL data+2          ; A is zero before reaching here
          ROL data+1
          ROL data+0
          LDA data+2
```

you can write: (which is shorter)

```
          SLO data+2          ; A is zero before reaching here
          ROL data+1
          ROL data+0
```

RLA (RLN)

Type: Combination of two operations with the same addressing mode (Sub-instructions: ROL, AND)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$27	RLA zp	{addr} = rol {addr} A = A and {addr}	2	5	o						o	x
$37	RLA zp, x		2	6	o						o	x
$23	RLA (zp, x)		2	8	o						o	x
$33	RLA (zp), y		2	8	o						o	x
$2F	RLA abs		3	6	o						o	x
$3F	RLA abs, x		3	7	o						o	x
$3B	RLA abs, y		3	7	o						o	x

Operation: Rotate one bit left in memory, then AND accumulator with memory.

Example:

```
RLA $FC,X      ;37 FC
```

Equivalent Instructions:

```
ROL $FC,X
AND $FC,X
```

Test code: `Lorenz-2.15/rlaa.prg, Lorenz-2.15/rlaax.prg, Lorenz-2.15/rlaay.prg, Lorenz-2.15/rlaix.prg, Lorenz-2.15/rlaiy.prg, Lorenz-2.15/rlaz.prg, Lorenz-2.15/rlazx.prg`

Example: scroll over a background layer

Lets say you want to create a scroller that moves text over some fixed background graphics. Suppose the data of the sliding text is stored at `scrollgfx` and the data of the fixed background at `backgroundgfx`. The actual data that is displayed is located at `buffer`.

Combining the sliding and fixed data without RLA would go something like (for the rightmost byte of the top line of the gfx data) this:

```
ROL scrollgfx       ; shift left (with carry)
LDA scrollgfx
AND backgroundgfx   ; combine with background
STA buffer
```

… which takes 18 cycles in 16 bytes

instead you can write:

```
LDA backgroundgfx
RLA scrollgfx       ; shift left and combine with bg
STA buffer
```

… which takes 14 cycles in 12 bytes

SRE (LSE)

Type: Combination of two operations with the same addressing mode (Sub-instructions: LSR, EOR)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$47	SRE zp	{addr} = {addr} / 2 A = A eor {addr}	2	5	o						o	o
$57	SRE zp, x		2	6	o						o	o
$43	SRE (zp, x)		2	8	o						o	o
$53	SRE (zp), y		2	8	o						o	o
$4F	SRE abs		3	6	o						o	o
$5F	SRE abs, x		3	7	o						o	o
$5B	SRE abs, y		3	7	o						o	o

Operation: Shift right one bit in memory, then EOR accumulator with memory.

Example:

```
SRE $C100,X          ;5F 00 C1
```

Equivalent Instructions:

```
LSR $C100,X
EOR $C100,X
```

Test code: `Lorenz-2.15/lsea.prg, Lorenz-2.15/lseax.prg, Lorenz-2.15/lseay.prg, Lorenz-2.15/lseix.prg, Lorenz-2.15/lseiy.prg, Lorenz-2.15/lsez.prg, Lorenz-2.15/lsezx.prg`

Example: 8bit 1-of-8 counter

SRE shifts the content of a memory location to the right and EORs the content with A, while SLO shifts to the left and does an OR instead of EOR.

So this is nice to combine the previous described 8 bit counter with for e.g. setting pixels:

```
          LDA #$80
          STA pix
          ...
          LDA (zp),y
          SRE pix             ;shift mask one to the right
                              ;and eor mask with A
          BCS advance_column  ;did the counter under-run?
                              ;so advance column
          STA (zp),y
          ...

advance_column:

          ROR pix             ;reset counter
          ORA #$80            ;set first pixel
          STA (zp),y

          LDA zp              ;advance column
          ;CLC                ;is still clear
          ADC #$08
          STA zp
          BCC +
          INC zp+1
+
```

RRA (RRD)

Type: Combination of two operations with the same addressing mode (Sub-instructions: ROR, ADC)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$67	RRA zp	{addr} = ror {addr} A = A adc {addr}	2	5	o	o			i		o	x
$77	RRA zp, x		2	6	o	o			i		o	x
$63	RRA (zp, x)		2	8	o	o			i		o	x
$73	RRA (zp), y		2	8	o	o			i		o	x
$6F	RRA abs		3	6	o	o			i		o	x
$7F	RRA abs, x		3	7	o	o			i		o	x
$7B	RRA abs, y		3	7	o	o			i		o	x

Operation: Rotate one bit right in memory, then add memory to accumulator (with carry).

This instruction inherits the decimal flag dependency from ADC.

Example:

```
RRA $030C      ;6F 0C 03
```

Equivalent Instructions:

```
ROR $030C
ADC $030C
```

Test code: `Lorenz-2.15/rraa.prg, Lorenz-2.15/rraax.prg, Lorenz-2.15/rraay.prg, Lorenz-2.15/rraix.prg, Lorenz-2.15/rraiy.prg, Lorenz-2.15/rraz.prg, Lorenz-2.15/rrazx.prg, 64doc/roradc.prg`

SAX (AXS, AAX)

Type: Combination of two operations with the same addressing mode (Sub-instructions: STA, STX)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$87	SAX zp	{addr} = A & X	2	3								
$97	SAX zp, y		2	4								
$83	SAX (zp, x)		2	6								
$8F	SAX abs		3	4								

Operation: AND the contents of the A and X registers (without changing the contents of either register) and stores the result in memory.

'The SAX instruction decodes to two instructions (STA and STX) whose behaviour is identical except that one hits the output-enable signal for the accumulator, and the other hits the output-enable signal for the X register. Although it would seem that this would cause ambiguous behaviour, it turns out that during one half of each cycle the internal operand-output bus is set to all '1's, and the read-enable signals for the accumulator and X register (and Y register, stack pointer, etc.) only allow those registers to set the internal operand-output bus bits to '0'. Thus, if a bit is zero in either the accumulator or the X register, it will be stored as zero; if it's set to '1' in both, then nothing will pull down the bus so it will output '1'.'

Example:

```
SAX $FE         ;87 FE
```

Equivalent Instructions:

```
PHP             ; save flags and accumulator
PHA
STX $FE
AND $FE
STA $FE
PLA             ; restore flags and accumulator
PLP
```

Note that SAX does not affect any flags in the processor status register, and does not modify A/X. It would also not actually use the stack, which is only needed to mimic the behaviour with legal opcodes in this example.

Test code: `Lorenz-2.15/axsa.prg, Lorenz-2.15/axsix.prg, Lorenz-2.15/axsz.prg, Lorenz-2.15/axszy.prg`

Note that two addressing modes that SAX is missing, absolute Y indexed and indirect Y indexed, can be simulated by using the SHA instruction, see SHA (AXA, AHX).

Example: store values with mask

This opcode is ideal to set up a permanent mask and store values combined with that mask:

```
LDX #$aa       ;set up mask
LDA $1000,y    ;load A
SAX $80,y      ;store A & $aa
```

Example: update Sprite Pointers

Often you need to set up all 8 sprite pointers in as few cycles as possible, this could be done like this:

```
LDA #$01
LDX #$fe
SAX screen + $3f8    ;00
STA screen + $3f9    ;01
LDA #$03
SAX screen + $3fa    ;02
STA screen + $3fb    ;03
LDA #$05
SAX screen + $3fc    ;04
STA screen + $3fd    ;05
LDA #$07
SAX screen + $3fe    ;06
STA screen + $3ff    ;07
```

LAX

Type: Combination of two operations with the same addressing mode (Sub-instructions: LDA, LDX)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$A7	LAX zp	A,X = {addr}	2	3	o						o	
$B7	LAX zp, y		2	4	o						o	
$A3	LAX (zp, x)		2	6	o						o	
$B3	LAX (zp), y		2	5 (+1)	o						o	
$AF	LAX abs		3	4	o						o	
$BF	LAX abs, y		3	4 (+1)	o						o	

Operation: Load both the accumulator and the X register with the contents of a memory location.

Example:

```
LAX $8400,Y          ;BF 00 84
```

Equivalent Instructions:

```
LDA $8400,Y
TAX
```

Test code: `Lorenz-2.15/laxa.prg, Lorenz-2.15/laxay.prg, Lorenz-2.15/laxix.prg, Lorenz-2.15/laxiy.prg, Lorenz-2.15/laxz.prg, Lorenz-2.15/laxzy.prg`

Example: load A and X with same value

Loading A and X with the same value is ideal if you manipulate the original value, but later on need the value again. Instead of loading it again you can either transfer it again from the other register, or combine A and X again with another illegal opcode.

```
LAX $1000,y    ;load A and X with value from $1000,y
EOR #$80       ;manipulate A
STA ($fd),y    ;store A
LDA #$f8       ;load mask
SAX jump+1     ;store A & X
```

Also one could so:

```
LAX $1000,y    ;load A and X with value from $1000,y
EOR #$80       ;manipulate A
STA ($fd),y    ;store A
TXA            ;fetch value again
EOR #$40       ;manipulate
STA ($fb),y    ;store
```

DCP (DCM)

Type: Combination of two operations with the same addressing mode (Sub-instructions: DEC, CMP)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$C7	DCP zp	{addr} = {addr} - 1 A cmp {addr}	2	5	o						o	o
$D7	DCP zp, x		2	6	o						o	o
$C3	DCP (zp, x)		2	8	o						o	o
$D3	DCP (zp), y		2	8	o						o	o
$CF	DCP abs		3	6	o						o	o
$DF	DCP abs, x		3	7	o						o	o
$DB	DCP abs, y		3	7	o						o	o

Operation: Decrement the contents of a memory location and then compare the result with the A register.

Example:

```
DCP $FF         ;C7 FF
```

Equivalent Instructions:

```
DEC $FF
CMP $FF
```

Test code: `Lorenz-2.15/dcma.prg, Lorenz-2.15/dcmax.prg, Lorenz-2.15/dcmay.prg, Lorenz-2.15/dcmix.prg, Lorenz-2.15/dcmiy.prg, Lorenz-2.15/dcmz.prg, Lorenz-2.15/dcmzx.prg, 64doc/dincsbc-deccmp.prg`

Example: decrementing loop counter

```
X1:       .byte $07
x2:       .byte $1a

          ;an effect
-
          ...
          DEC x2
          LDA x2
          CMP x1
          BNE -
```

can be written as:

```
          ;an effect
-
          ...
          LDA x1
          DCP x2    ;decrements x2 and compares x2 to A
          BNE -
```

Example: decrementing 16bit counter

For decrementing a 16 bit pointer it is also of good use:

```
          LDA #$ff
          DCP ptr
          BNE *+4
          DEC ptr+1
          ;carry is set always for free
```

ISC (ISB, INS)

Type: Combination of two operations with the same addressing mode (Sub-instructions: INC, SBC)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$E7	ISC zp	{addr} = {addr} + 1 A = A - {addr}	2	5	o	o			i		o	x
$F7	ISC zp, x		2	6	o	o			i		o	x
$E3	ISC (zp, x)		2	8	o	o			i		o	x
$F3	ISC (zp), y		2	8	o	o			i		o	x
$EF	ISC abs		3	6	o	o			i		o	x
$FF	ISC abs, x		3	7	o	o			i		o	x
$FB	ISC abs, y		3	7	o	o			i		o	x

Operation: Increase memory by one, then subtract memory from accumulator (with borrow).

This instruction inherits the decimal flag dependency from SBC.

Example:

```
          ISC $FF         ;E7 FF
```

Equivalent Instructions:

```
          INC $FF
          SBC $FF
```

Test code: Lorenz-2.15/insa.prg, Lorenz-2.15/insax.prg, Lorenz-2.15/insay.prg, Lorenz-2.15/insix.prg, Lorenz-2.15/insiy.prg, Lorenz-2.15/insz.prg, Lorenz-2.15/inszx.prg, 64doc/dincsbc.prg

Example: incrementing loop counter

Instead of:

```
INC counter
LDA counter
CMP #ENDVALUE
BNE next
```

you can write: (which saves a cycle when counter is in zero-page)

```
LDA #ENDVALUE
SEC
ISC counter
BNE next
```

Example: increment indexed and load value

Instead of:

```
; A is zero and C=0 before reaching here
INC buffer, x
LDA buffer, x
```

you can write: (which saves a byte if buffer is in regular memory, and is faster)

```
; A is zero and C=0 before reaching here
ISC buffer, x
EOR #$ff
```

ANC

Type: Combination of an immediate and an implied command (Sub-instructions: AND, ASL/ROL)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$0B	ANC #imm	A = A & #{imm}	2	2	o						o	o
$2B	ANC #imm	A = A & #{imm}	2	2	o						o	o

Operation: ANDs the contents of the A register with an immediate value and then moves bit 7 of A into the Carry flag.

This opcode works basically identically to AND #imm. except that the Carry flag is set to the same state that the Negative flag is set to. (bit 7 is put into the carry, as if the ASL/ROL would have been executed)

Example:

```
ANC #$AA        ;2B AA
```

Equivalent Instructions:

```
AND #$AA
; ROL A - not actually executed, set C as if it was
```

Test code: `Lorenz-2.15/ancb.prg`

Example: implicit enforcement of carry flag state

When using an AND instruction before an addition (or any other operation where you might want to know the state of the carry flag), you might save two cycles (not having to do CLC or SEC) by using ANC instead of AND. Since a cleared high bit in the value used with the ANC instruction always leads to a unset carry flag after this operation, you can take advantage of that. An example:

```
LDA value
ANC #$0f        ;Carry flag is always set to 0
                ;after this op.
ADC value2      ;Add a value. CLC not needed!
STA result
```

Another case like this is when you want to set the A register to #$00 specifically, and also happen to want to have the carry cleared:

```
ANC #0          ;Carry always cleared after this op,
                ;and A register always set to zero.
```

Example: remembering a bit

You can use ANC to simply putting the highest bit of a byte into the carry flag without affecting a register (by using ANC #$FF). This can be useful sometimes since not that many instructions destroy the (C)arry flag as well as the (N)egative flag (mainly mathematical operations, shifting operations and comparison operations), in order to 'remember' this information during the execution of other code (such as some LDA/STA stuff).

A command that does this too is CMP #$80 (as well as CPX and CPY), which non destructively puts the high bit of a register into Carry as well.

ALR (ASR)

Type: Combination of an immediate and an implied command (Sub-instructions: AND, LSR)

Opc.	Mnemonic	Function	Size	Cycles	N	V	B	D	I	Z	C
$4B	ALR #imm	A = (A & #{imm}) / 2	2	2	o					o	o

Operation: AND the contents of the A register with an immediate value and then LSRs the result.

Example:

```
ALR #$FE        ;4B FE
```

Equivalent Instructions:

```
AND #$FE
LSR A
```

Test code: `Lorenz-2.15/alrb.prg`

Example: right shift and mask

Whenever you need to shift and influence the carry afterwards, you can use ALR for that, and if you even need to apply an and-mask beforehand, you are extra lucky and can do 3 commands by that:

```
ALR #$fe  ;-> A & $fe = $fe -> lsr -> carry is cleared
          ; as bit 0 was not set before lsr
```

… same as …

```
AND #$ff
LSR
CLC
```

Example: fetch 2 bits from a byte

```
          LDA #%10110110
          LSR
          ALR #$03*2
```

This will mask out and shift down bits 2 and 3. Note that the mask is applied before shifting, therefore the mask is multiplied by two.

Example: add offset depending on LSB

Another nice trick to transform a single bit into a new value (good for adding offsets depending on the value of a single bit) offset is the following:

```
          LDA xposl      ;load a value
          ALR #$01       ;move LSB to carry and clear A
          BCC +
          LDA #$3f       ;carry is set
+
          ADC #stuff     ;things will work sane, as offset
                         ;includes already the carry
```

As you can see we have now either loaded $00 or $40 (carry!) to A depending on the state of bit 0, that is ideal for e.g. when we want to load from a different bank depending on if a position is odd or even. As you see, the above example is even faster than this (as the shifting always takes 6 cycles, whereas the above example takes 5/6 cycles):

```
          LDA xposl
          ALR #$01
          ROR
          LSR
          ADC #stuff     ;things will work sane as carry is
                         ;always clear (upper bits are masked)
```

ARR

Type: Combination of an immediate and an implied command (Sub-instructions: AND, ROR)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$6B	ARR #imm	A = (A & #{imm}) / 2	2	2	o	o			i		o	o

note to ARR: part of this command are some ADC mechanisms. following effects appear after AND but before ROR: the V-Flag is set according to (A and #{imm})+#{imm}, bit 0 does NOT go into carry, but bit 7 is exchanged with the carry.

ARR ANDs the accumulator with an immediate value and then rotates the content right. The resulting carry is however not influenced by the LSB as expected from a normal rotate. The Carry and the state of the overflow-flag depend on the state of bit 6 and 7 before the rotate occurs, but after the and-operation has happened, and will be set like shown in the following table:

Bit 7	Bit 6	Carry	Overflow
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Example:

```
          ARR #$7F       ;6B 7F
```

Equivalent Instructions:

```
          AND #$7F
          ROR A     ; flags are different with ARR, see the
                    ; above table
```

Test code: `CPU/asap/cpu_decimal.prg, Lorenz-2.15/arrb.prg`

The following assumes the decimal flag is clear: The sub-instruction for ARR ($6B) is in fact ADC ($69), not AND. While ADC is not performed, some of the ADC mechanics are evident. Like ADC, ARR affects the overflow flag. The following effects occur after ANDing but before RORing: The V flag is set to the result of exclusive ORing bit 7 with bit 6. Unlike ROR, bit 0 does not go into the carry flag. The state of bit 7 is exchanged with the carry flag. Bit 0 is lost. All of this may appear strange, but it makes sense if you consider the probable internal operations of ADC itself.

Example: rotating 16 bit values

```
        LDA #>addr
        LSR
        STA $fc
        ARR #$00  ;A = A & $00 -> ror A
        STA $fb
```

… is the same as …

```
        LDA #>addr
        LSR
        STA $fc
        LDA #$00
        ROR
        STA $fb
```

Note: Again, you can influence the final state of the carry by either using #$00 or #$01 for the LDA ($00 or $80 in case of ARR, but the later only if A has bit 7 set as well, so be carefully here).

Example: load register depending on carry

If you need to load a register depending on some branch, you might be able to save some cycles. Imagine you have the following to load Y depending on the state of the carry:

```
          CMP $1000
          BCS +
          LDY #$00
          BEQ ++          ; jump always
+
          LDY #$80
++
```

This can be solved in less cycles and less memory:

```
          CMP $1000
          ARR #$00
          TAY
```

Example: shift zeros or ones into accumulator

Due to the fact that the carry resembles the state of bit 7 after ARR is executed, one can continuously shift in zeroes or ones into a byte:

```
          LDA #$80
          SEC
          ARR #$ff        ; -> A = $c0 -> sec
          ARR #$ff        ; -> A = $e0 -> sec
          ARR #$ff        ; -> A = $f0 -> sec
          ...
          LDA #$7f
          CLC
          ARR #$ff        ; -> A = $3f -> clc
          ARR #$ff        ; -> A = $1f -> clc
          ARR #$ff        ; -> A = $0f -> clc
```

SBX (AXS, SAX)

Type: Combination of an immediate and an implied command (Sub-instructions: CMP, DEX)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$CB	SBX #imm	X = A & X - #{imm}	2	2	o						o	o

Operation: SBX ANDs the contents of the A and X registers (leaving the contents of A intact), subtracts an immediate value, and then stores the result in X. ... A few points might be made about the action of subtracting an immediate value. It actually works just like the CMP instruction, except that CMP does not store the result of the subtraction it performs in any register. This subtract operation is not affected by the state of the Carry flag, though it does affect the Carry flag. It does not affect the Overflow flag. (Flags are set like with CMP, not SBC)

Another property of this opcode is that it doesn't respect the decimal mode, since it is derived from CMP rather than SBC. So if you need to perform table lookups and arithmetic in a tight interrupt routine there's no need to clear the decimal flag in case you've got some code running that operates in decimal mode.

Example:

```
SBX #$5A       ;CB 5A
```

Equivalent Instructions:

```
STA $02   ; save accumulator
TXA       ; hack because there is no 'AND WITH X'
AND $02   ; instruction
CMP #$5A  ; set flags like CMP
PHP       ; save flags
SEC
CLD       ; subtract without being affected by
SBC #$5A  ; decimal mode
TAX
LDA $02   ; restore accumulator
PLP       ; restore flags
```

Note: SBX is not easily expressed entirely correct using legal opcodes. Memory location $02 would not be altered by the SBX opcode, and it would not use the stack.

Test code: `Lorenz-2.15/sbxb.prg, 64doc/sbx.prg, 64doc/vsbx.prg, 64doc/sbx-c100.prg`

Example: decrement X by more than 1

Sometimes you need/want to decrease the X register by more than one. That is often done by the following piece of code:

```
TXA
SEC
SBC #$xx  ;where xx is (obviously) the value
          ;to decrease by
TAX
```

This procedure takes 8 cycles (and 5 bytes in memory). If the value of the carry flag is always known at this point in the code, it can be removed and the snippet would then take 6 cycles (and 4 bytes in memory). However, you can use SBX like this:

And the modified code snippet using SBX instead looks like this:

```
LDA #$ff  ;Next opcode contains a implicit AND with
          ;the A register, so turn all bits ON!
SBX #$xx  ;where xx is the value to decrease by
```

This code kills the A register of course, but so does the 'standard' version above. It can be made even shorter by using a 'txa' instruction instead of the 'lda #$ff'. That works since X and A will be equal after the 'txa', and ANDing a value with itself produces no change, hence the AND effect of SBX is 'disarmed' and the subtraction will proceed as expected:

```
TXA
SBX #$xx
```

Note that in this case you do not have to worry about the carry flag at all, and all in all the whole procedure takes only 4 cycles (and 3 bytes in memory)

Example: decrement nibbles

Imagine you have a byte that is divided into two nibbles (just what you often use in 4×4 effects), now you want to decrement each nibble, but when the low nibble underflows, this will decrement the high nibble as well, here the SBX command can help to find out about that special case:

```
          LDA $0400,y    ;load value
          LDX #$0f       ;set up mask
          SBX #$00       ;check if low nibble underflows
                         ; -> X = A & $0f
          BNE +          ;all fine, decrement both nibbles
                         ;the cheap way, carry is set!
          SBC #$f0       ;do wrap around by hand
          SEC
+
          SBC #$11       ;decrement both nibbles,
                         ;carry is set already by sbx
```

… can be substituted by …

```
          LDA #$0f       ;set up mask beforehand,
                         ;can be reused for each turn
          STA $02

          LDA $0400,y
          BIT $02        ;apply mask without destroying A
          BNE +
          CLC
          ADC #$10
+
          SEC            ;we need to set carry
          SBC #$11
```

Example: apply a mask to an index

Furthermore, the SBX command can also be used to apply a mask to an index easily:

```
LDX #$03        ;mask
LDA val1        ;load value
SBX #$00        ;mask out lower 2 bits -> X
LSR             ;A is untouched, so we can continue
                ;doing stuff with A
LSR
STA val1
LDA colours,x   ;fetch colour from table
```

instead of (which takes 3 cycles more):

```
LDA val1
AND #$03
TAX             ;set up index
LSR val1        ;A is clobbered, so shift direct
LSR val1
LDA colours,x
```

The described case makes it easy to decode 4 multicolour pixel pairs by always setting up an index from the lowest two bits and fetching the appropriate colour from a previously set up table.

SBC (USBC)

Type: Combination of an immediate and an implied command (Sub-instructions: SBC, NOP)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$EB	SBC #imm	A = A - #{imm}	2	2	o	o			i		o	x

Operation: subtract immediate value from accumulator with carry. Same as the regular SBC.

Test code: `Lorenz-2.15/sbcb-eb.prg`

LAS (LAR)

Type: Combinations of STA/TXS and LDA/TSX

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$BB	LAS abs,y	A,X,SP = {addr} & SP	3	4 (+1)	o						o	

Operation: AND memory with stack pointer, transfer result to accumulator, X register and stack pointer.

Example:

```
LAS $C000, Y        ;BB AA
```

Equivalent Instructions:

```
TSX
TXA
AND $C000, Y
TAX
TXS
```

Test code: `CPU/asap/cpu_las.prg, Lorenz-2.15/lasay.prg`

Note: LAS is called as 'probably unreliable' in one source - this does not seem to be the case though

It can be the case that the stack is not used in a main routine, since it is cheaper to store things in the zeropage. Of course when a subroutine is called or an interrupt triggers the return address (and status register in case of an IRQ) is stored on the stack, but after returning to the main loop the stackpointer (SP) is back to the same value again. This means that you can change the SP at will in the main loop without messing things up. For example, you can use it as temporary storage of the X register with TXS/TSX. This makes it possible to use LAS (and TAS).

Example: cycle an index within bounds

If you want to cycle an index and wrap around to zero at a number that is a power of two, you could do that with LAS. For example to cycle from 0-15, suppose we start with SP=$f7 (any value will work):

```
          LAS mask,y     ; if Mask is one page filled with $0f,
                         ; this brings the SP to $07 (and A and X
                         ; as well) for any Y.
          DEX            ; X = $06
          TXS            ; SP is now $06, so the next time
                         ; 'lda List,x' will pick the next value
          LDA table, x   ; use X as index
```

SP and X after the LAS instruction will always remain in the range 0-$0f, no need to check for that!

Instead of the lda Table,x one could use pla if the data is on the stack and no interrupt can take place during this code snippet.Then dex should be replaced by e.g. sbx #$11 to bring the SP to a safe area, to ensure the data on the stack is not messed up in other parts of the code.

NOP (NPO)

Type: no effect

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$1A	NOP	No operation	1	2								
$3A	NOP	No operation	1	2								
$5A	NOP	No operation	1	2								
$7A	NOP	No operation	1	2								
$DA	NOP	No operation	1	2								
$FA	NOP	No operation	1	2								

NOP (DOP, SKB)

Type: no effect

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$80	NOP #imm	Fetch #imm	2	2								
$82	NOP #imm	Fetch #imm	2	2								
$C2	NOP #imm	Fetch #imm	2	2								
$E2	NOP #imm	Fetch #imm	2	2								
$89	NOP #imm	Fetch #imm	2	2								

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$04	NOP zp	Fetch {addr}	2	3								
$44	NOP zp	Fetch {addr}	2	3								
$64	NOP zp	Fetch {addr}	2	3								

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$14	NOP zp, x	Fetch {addr}	2	4								
$34	NOP zp, x	Fetch {addr}	2	4								
$54	NOP zp, x	Fetch {addr}	2	4								
$74	NOP zp, x	Fetch {addr}	2	4								
$D4	NOP zp, x	Fetch {addr}	2	4								
$F4	NOP zp, x	Fetch {addr}	2	4								

Operation: `NOP zp` and `NOP zp,x` actually perform a read operation. It's just that the value read is not stored in any register.

Note: `NOP` opcodes $82, $C2, $E2 may be `JAM`s. Since only one source claims this, and no other sources corroborate this, it must be true on very few machines. On all others, these opcodes always perform 'no operation'. It is perhaps a good idea to avoid using them anyway.

NOP (TOP, SKW)

Type: no effect

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$0C	NOP abs	Fetch {addr}	3	4								

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$1C	NOP abs, x	Fetch {addr}	3	4 (+1)								
$3C	NOP abs, x	Fetch {addr}	3	4 (+1)								
$5C	NOP abs, x	Fetch {addr}	3	4 (+1)								
$7C	NOP abs, x	Fetch {addr}	3	4 (+1)								
$DC	NOP abs, x	Fetch {addr}	3	4 (+1)								
$FC	NOP abs, x	Fetch {addr}	3	4 (+1)								

Operation: These actually perform a read operation. It's just that the value read is not stored in any register. Further, opcode $0C uses the absolute addressing mode. The two bytes which follow it form the absolute address. All the other 3 byte `NOP` opcodes use the absolute indexed X addressing mode. If a page boundary is crossed, the execution time of one of these `NOP` opcodes is upped to 5 clock cycles.

Test code: `Lorenz-2.15/nopa.prg, Lorenz-2.15/nopax.prg, Lorenz-2.15/nopb.prg, Lorenz-2.15/nopn.prg, Lorenz-2.15/nopz.prg, Lorenz-2.15/nopzx.prg`

Example: acknowledge IRQ

If for some reason you want to acknowledge a timer IRQ and can not afford changing a register or the CPU status, you can use the fact that some of these NOPs actually perform a read operation:

```
NOP $DCOD       ;0C 0D DC
```

JAM (KIL, HLT, CIM)

Type: lock-up

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$02	JAM	CPU lock-up	1	-								
$12	JAM	CPU lock-up	1	-								
$22	JAM	CPU lock-up	1	-								
$32	JAM	CPU lock-up	1	-								
$42	JAM	CPU lock-up	1	-								
$52	JAM	CPU lock-up	1	-								
$62	JAM	CPU lock-up	1	-								
$72	JAM	CPU lock-up	1	-								
$92	JAM	CPU lock-up	1	-								
$B2	JAM	CPU lock-up	1	-								
$D2	JAM	CPU lock-up	1	-								
$F2	JAM	CPU lock-up	1	-								

Operation: When one of these opcodes is executed, the byte following the opcode will be fetched, data- and address bus will be set to $ff (all 1s) and program execution ceases. No hardware interrupts will execute either. Only a reset will restart execution. This opcode leaves no trace of any operation performed! No registers or flags affected.

Test code: `CPU/cpujam/cpujamXX.prg`

Example: stop execution

Sometimes in a very memory constrained situation (like a 4k demo), you may want to stop execution of whatever is running with least effort – this can be achieved by using one of the JAM opcodes. Keep in mind though that only the CPU will stop.

```
LDA #0
STA $D418
JAM             ;02
```

Unstable Opcodes

Out of all opcodes, just **seven** fall into the so called 'unstable' category. They can be used, but extra care must be taken to work around the unstable behaviour, so read the following carefully :)

'unstable address high byte' group

There are five opcodes in this group. None of these opcodes affect the accumulator, the X register, the Y register, or the processor status register. They have two instabilities which have to be 'disarmed' by careful programming.

- If the target address crosses a page boundary because of indexing, the instruction may not store at the intended address, instead the high byte of the target address turns into the value stored. (some sources say *'or at a different address altogether'*). For this reason **you should generally keep your index in a range that page boundaries are not crossed**. This anomaly seems to exist on some chips only.

- Sometimes the actual value is stored in memory and the AND with <addrhi+1> part drops off (ex. SHY becomes true STY). This happens when the RDY line is used to stop the CPU (pulled low), i.e. either a 'bad line' or sprite DMA starts, in the second half of the cycle following the opcode fetch. '*For example, it never seems to occur if either the screen is blanked or C128 2MHz mode is enabled.*' For this reason **you will have to choose a suitable target address based on what kind of values you want to store**. *'For $fe00 there's no problem, since anding with $ff is the same as not anding. And if your values don't mind whether they are anded, e.g. if they are all $00-$7f for shy $7e00,x, there is also no difference whether the and works or not.'*

'To explain what's going on take a look at LDA ABX and STA ABX first.

LDA ABX takes 4 cycles unless a page wrap occurred (address+X lies in another page than address) in which case the value read during the 4th cycle (which was read with the original high byte) is discarded and in the 5th cycle a read is made again, this time from the correct address. During the 4th cycle the high address byte is incremented in order to have a correct high byte if the 5th cycle is necessary. The byte read from memory is buffered and copied to A during the read of the next command's opcode.

But there's a problem with storage commands: they need to put the value to write on the internal bus which is used for address computations as well. To avoid collisions STA ABX contains a fix-up which makes it always take 5 cycles (the value is always written in the 5th cycle as the high byte is computed in the 4th cycle).

This fix-up requires some transistors on the CPU; the guys at MOS forgot (or were unable?) to make them detect STX ABY (which becomes SHX) and a few others, they are missing that fix-up so this results in a collision between the value and high address byte computation.'

SHA (AXA, AHX)

Type: Combinations of STA/STX/STY

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$93	SHA (zp),y	{addr} = A & X & {H+1}	2	6								
$9F	SHA abs,y	{addr} = A & X & {H+1}	3	5								

Operation: This opcode stores the result of A AND X AND the high byte of the target address of the operand +1 in memory.

Instabilities:

- sometimes the &{H+1} drops off.
- page boundary crossing may not work as expected (the page where the value is stored may be equal to the value stored).

Example:

```
          SHA $7133,Y         ;9F 33 71
```

Equivalent Instructions:

```
          PHP             ; save flags and accumulator
          PHA
          STX $02         ; hack which is needed because there is
          AND $02         ; no 'AND-WITH-X' instruction
          AND #$72        ; High-byte of Address + 1
          STA $7133,Y
          LDX $02         ; restore X-register
          PLA             ; restore flags and accumulator
          PLP
```

Note: Memory location $02 would not be altered by the SHA opcode and it would not use the stack.

Test code:

- general: `Lorenz-2.15/shaay.prg, Lorenz-2.15/shaiy.prg`
- &H drop off: `CPU/sha/shazpy2.prg CPU/sha/shazpy3.prg CPU/sha/shaabsy2.prg CPU/sha/shaabsy3.prg`
- page boundaries: `CPU/sha/shazpy1.prg CPU/sha/shaabsy1.prg`

Example: SAX abs, y

When using $FE00 as address, the value stored would be ANDed by $FF and the SHA turns into a SAX:

```
          SHA $FE00,Y    ; SAX $FE00,Y
```

Example: SAX (zp), y

When using $FE00 as address, the value stored would be ANDed by $FF and the SHA turns into a SAX:

```
          LDA #$FE
          STA $03
          LDA #$00
          STA $02
          ...
          SHA ($02),Y    ; SAX ($02),Y
```

SHX (A11, SXA, XAS)

Type: Combinations of STA/STX/STY

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$9E	SHX abs,y	{addr} = X & {H+1}	3	5								

Operation: AND X register with the high byte of the target address of the argument + 1. Store the result in memory.

Instabilities:

- sometimes the &{H+1} drops off.
- page boundary crossing may not work as expected (the page where the value is stored may be equal to the value stored).

Example:

```
          SHX $6430,Y    ;9E 30 64
```

Equivalent Instructions:

```
          PHP            ; save flags and accumulator
          PHA
          TXA
          AND #$65       ; High byte of Address + 1
          STA $6430,Y
          PLA            ; restore flags and accumulator
          PLP
```

Note: The SHX opcode would not use the stack.

Test code:

- general: `CPU/asap/cpu_shx.prg, Lorenz-2.15/shxay.prg`
- &H drop off: `CPU/shxy/shxy2.prg, CPU/shxy/shxy3.prg`
- page boundaries: `CPU/shxy/shxy1.prg`

Simulation links:

- &{H+1} drop off:
  http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=20&d=a20fa0009e0211&logmore=rdy&rdy0=15&rdy1=16
- page boundary crossing anomaly:
  http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=18&d=a20Fa0ff9e0211

Example: STX abs, y

When using $FE00 as address, the value stored would be ANDed by $FF and the SHX turns into a STX:

```
          SHX $FE00,Y    ; STX $FE00,Y
```

SHY (A11, SYA, SAY)

Type: Combinations of STA/STX/STY

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$9C	SHY abs,x	{addr} = Y & {H+1}	3	5								

Operation: AND Y register with the high byte of the target address of the argument + 1. Store the result in memory.

Instabilities:

- sometimes the &{H+1} drops off.
- page boundary crossing may not work as expected (the page where the value is stored may be equal to the value stored).

Example:

```
SHY $7700,X    ;9C 00 77
```

Equivalent Instructions:

```
PHP            ; save flags and accumulator
PHA
TYA
AND #$78       ; High byte of Address + 1
STA $7700,X
PLA            ; restore flags and accumulator
PLP
```

Note: the SHY opcode would not use the stack.

Test code:

- general: `CPU/asap/cpu_shx.prg, Lorenz-2.15/shyax.prg`
- &H drop off: `CPU/shxy/shyx2.prg, CPU/shxy/shyx3.prg`
- page boundaries: `CPU/shxy/shyx1.prg`

Example: STY abs, x

When using $FE00 as address, the value stored would be ANDed by $FF and the SHY turns into a STY:

```
          SHY $FE00,X    ; STY $FE00,X
```

TAS (XAS, SHS)

Type: Combinations of STA/TXS and LDA/TSX

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$9B	TAS abs,y	SP = A & X {addr} = SP & {H+1}	3	5								

Operation: This opcode ANDs the contents of the A and X registers (without changing the contents of either register) and transfers the result to the stack pointer. It then ANDs that result with the contents of the high byte of the target address of the operand +1 and stores that final result in memory.

Instabilities:

- sometimes the &{H+1} drops off.
- page boundary crossing may not work as expected (the page where the value is stored may be equal to the value stored).

Example:

```
          TAS $7700,Y    ;9B 00 77
```

Equivalent Instructions:

```
          PHA            ; save accumulator
          STX $02        ; hack, since there is no 'AND WITH X'
          AND $02        ; instruction
          TAX
          AND #$78       ; High-byte of Address + 1
          STA $7700,Y
          PLA            ; restore accumulator
          TXS
          LDX $02        ; TAS would not modify the flags
```

Note: The above code does in many ways not accurately resemble how the TAS opcode works exactly, memory location $02 would not be altered, the stack would not be used, and no processor flags would be modified.

Test code:

- general: `Lorenz-2.15/shsay.prg`
- &H drop off: `CPU/shs/shsabsy2.prg, CPU/shs/shsabsy3.prg`
- page boundaries: `CPU/shs/shsabsy1.prg`

'Magic Constant' group

The two opcodes in this group are combinations of an immediate and an implied command, and involve a highly unstable 'magic constant', which is chip and/or temperature dependent. The behaviour also depends on the RDY line, which needs extra caution.

ANE (XAA)

Type: Combination of an immediate and an implied command

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$8B	ANE #imm	A = (A | {CONST}) & X & #{imm}	2	2	o						o	

Operation: This opcode ORs the A register with CONST, ANDs the result with X. ANDs the result with an immediate value, and then stores the result in A.

Instability: CONST is chip- and/or temperature dependent (common values may be $00, $ff, $ee …). Some dependency on the RDY line (**This is not yet emulated in VICE 3.3**)

for some very detailed info on how this opcode works look here: http://visual6502.org/wiki/index.php?title=6502_Opcode_8B_(XAA,_ANE)

Do not use ANE with any other immediate value than 0, or when the accumulator value is $ff (both takes the magic constant out of the equation)! (Or, more generalized, these are safe if all bits that could be 0 in A are 0 in either the immediate value or X or both.)

Example:

```
ANE #{IMM}          ;8B {IMM}
```

Equivalent Instructions:

```
ORA #{CONST}
AND #{IMM}
STX $02         ; hack because there is no 'AND WITH X'
AND $02         ; instruction
```

Note: Memory location $02 would not be altered by the ANE opcode.

Test code:

- `general: CPU/asap/cpu_ane.prg, Lorenz-2.15/aneb.prg`
- `temperature dependency: general/ane-lax/ane-lax.prg`
- `dependency on RDY line: CPU/ane/ane.prg`

Example: clear A

```
          ANE #0          ; 8B 00
```

is equivalent to

```
          LDA #0
```

… and is safe to use as using 0 as the immediate value takes the 'magic constant' out of the equation.

Example: A = X AND immediate

```
          ;LDA #$ff   assuming A=$ff from previous operation
          ANE #$0f        ; 8B 0f  A = (A | const) & X & $0f
```

is equivalent to

```
          TXA
          AND #$0f
```

… and is safe to use as a value of $ff in accumulator takes the 'magic constant' out of the equation.

Example: read the 'magic constant'

To determine the 'magic constant' which is in effect on your particular machine, you can do this:

```
          LDA #0
          LDX #$ff
          ANE #$ff        ; A contains the magic constant
```

This is mostly useful for experimenting and proving the constant is actually different on different set-ups. **Do not rely on this value!** It may not be stable even on the same chip and depend on temperature and/or the supplied voltage.

LAX #imm (ATX, LXA, OAL, ANX)

Type: Combination of an immediate and an implied command

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$AB	LAX #imm	A,X = (A | {CONST}) & #{imm}	2	2	o						o	

Operation: This opcode ORs the A register with CONST, ANDs the result with an immediate value, and then stores the result in both A and X.

Instability: CONST is chip- and/or temperature dependent (common values may be $00, $ff, …). Some dependency on the RDY line. (**This is not yet emulated in VICE 3.3**)

Do not use LAX #imm with any other immediate value than 0, or when the accumulator value is $ff (both takes the magic constant out of the equation)! (Or, more generalized, these are safe if all bits that could be 0 in A are 0 in the immediate value.)

'The problem with LAX immediate is that its decode is a combination of LDA, LDX, and TAX. This causes the current contents of the accumulator to be merged in with the value loaded from the data bus. Normally, during an LDA or LDX instruction, it doesn't matter if the operand-input bus is stable during the whole half-cycle for which they're enabled. Nothing is reading from the registers while they are being loaded; as long as the bus has stabilized before the load-enable signal goes away, the registers will end up with the correct value. The LAX opcode, however, enables the 'output accumulator' signal as well as the 'feed output bus to input bus' signal. My 6507 documentation doesn't show which buses have 'true' or 'inverted' logic levels, but a natural implementation would likely use the opposite signal polarity for the output bus and input bus (so the connections between them would be inverting buffers). Under that scenario, LAX would represent a race condition to see which bus got a 'low' signal first. A variety of factors could influence 'who wins' such a race.'

Example:

```
LAX #{IMM}          ;AB {IMM}
```

Equivalent Instructions:

```
ORA #{CONST}
AND #{IMM}
TAX
```

Test code:

- general: CPU/asap/cpu_anx.prg, Lorenz-2.15/lxab.prg
- temperature dependency: general/ane-lax/ane-lax.prg
- dependency on RDY line: CPU/lax/lax.prg

Example: clear A and X

```
LAX #0          ; AB 00
```

is equivalent to:

```
LDA #0
TAX
```

… and is safe to use as using 0 as the immediate value takes the 'magic constant' out of the equation.

Example: load A and X with same value

```
; assuming A=$ff from previous operation
LAX #<value>        ; AB <value>
```

is equivalent to:

```
LDA #<value>
TAX
```

… and is safe to use as a value of $ff in accumulator takes the 'magic constant' out of the equation.

Example: read the 'magic constant'

To determine the 'magic constant' which is in effect on your particular machine, you can do this:

```
LDA #0
LAX #$ff        ; A,X contain the magic constant
```

This is mostly useful for experimenting and proving the constant is actually different on different set-ups. **Do not rely on this value!** It may not be stable even on the same chip and depend on temperature and/or the supplied voltage.

Unintended addressing modes

Absolute Y Indexed (R-M-W)

- 3 bytes, 7 cycles

```
db lo hi   DCP abs, y
fb lo hi   ISC abs, y
7b lo hi   RRA abs, y
1b lo hi   SLO abs, y
5b lo hi   SRE abs, y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4	< AAH, AAL + Y >	< Internal Operation >	R
5	AA + Y	Data Low	R
6	AA + Y	Old Data Low	W
7	AA + Y	New Data Low	W

Simulation Link: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=22&d=a0d0db10eaeaeaeaeaeaeaeaeaeaea1280

equivalent legal mode: Absolute X Indexed (R-M-W)

- 3 bytes, 7 cycles

```
ASL abs, x  DEC abs, x  INC abs, x  LSR abs, x  ROL abs, x  ROR abs, x
2f lo hi   RLA abs, x
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4	< AAH, AAL + X >	< Internal Operation >	R
5	AA + X	Data Low	R
6	AA + X	Old Data Low	W
7	AA + X	New Data Low	W

Zeropage X Indexed Indirect (R-M-W)

- 2 bytes, 8 cycles

```
C3 zp   DCP (zp, x)
E3 zp   ISC (zp, x)
23 zp   RLA (zp, x)
63 zp   RRA (zp, x)
03 zp   SLO (zp, x)
43 zp   SRE (zp, x)
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Direct Offset	R
3	< PC + 1 >	< Internal Operation >	R
4	DO + X	Absolute Address Low	R
5	DO + X + 1	Absolute Address High	R
6	AA	Data Low	R
7	AA	Old Data Low	W
8	AA	New Data Low	W

Simulation Link: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=22&d=a2d0c310eaeaeaeaeaeaeaeaeaeaeaea1280

related legal mode: Zeropage X Indexed Indirect

- 2 bytes, 6 cycles

```
ADC (zp, x)  AND (zp, x)  CMP (zp, x)  EOR (zp, x)  LDA (zp, x)  ORA (zp, x)  SBC (zp, x)
STA (zp, x)

a3 zp   LAX (zp, x)
83 zp   SAX (zp, x)
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Direct Offset	R
3	< PC + 1 >	< Internal Operation >	R
4	DO + X	Absolute Address Low	R
5	DO + X + 1	Absolute Address High	R
6	AA	Data Low	R/W

Zeropage Indirect Y Indexed (R-M-W)

- 2 bytes, 8 cycles

```
D3 zp   DCP (zp), y
F3 zp   ISC (zp), y
33 zp   RLA (zp), y
73 zp   RRA (zp), y
13 zp   SLO (zp), y
53 zp   SRE (zp), y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Direct Offset Fetch	R
3	DO	Absolute Address Low	R
4	DO + 1	Absolute Address High	R
5	< AAH, AAL + Y >	< Internal Operation >	R
6	AA + Y	Data Low	R
7	AA + Y	Old Data Low	W
8	AA + Y	New Data Low	W

Simulation Link: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=22&d=a0d0d310eaeaeaeaeaeaeaeaeaeaea1280

related legal mode: Zeropage Indirect Y Indexed

- 2 bytes, 5+1 cycles

```
ADC (zp), y  AND (zp), y  CMP (zp), y  EOR (zp), y  LDA (zp), y  ORA (zp), y  SBC (zp), y
STA (zp), y
b7 zp   LAX (zp), y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Direct Offset	R
3	DO	Absolute Address Low	R
4	DO + 1	Absolute Address High	R
+1 (*)	< AAH, AAL + Y >	< Internal Operation >	R
5	AA	Data Low	R/W

(*) Add 1 cycle for indexing across page boundaries, or write

Unintended decimal mode

The decimal mode (or “BCD mode”) of the 6502 family is an often ignored artefact of the instruction set. Since it turned out not to be very useful in many practical situations, many programmers never use it, which contributes to the state of it being ignored :)

The decimal mode is described here because

- The behaviour of operations on invalid BCD values is officially undocumented. The following exactly describes the behaviour for all values, valid BCD or not, by giving exact pseudocode for each instruction.
- Some undocumented instructions inherit dependency on decimal mode from ADC or SBC. The main part of this document refers to binary mode, the following exactly describes how these instructions work in decimal mode.
- Last not least because decimal mode is ignored by so many programmers

Like the rest of the document, the following applies specifically to the 6510 MOS chips. 65C02 or 65816 as well as other derivates behave totally different when it comes to details such as flag behaviour and invalid BCD values.

Decimal mode in a nutshell

The decimal mode is ment to aid in making calculations with BCD encoded values (“packaged” BCD, one digit per nybble). A BCD encoded value is a hex number with both its upper and lower nybble equal to 0-9. All other values are invalid BCD values.

When the D flag is set, only (!) the ADC and SBC instructions (and undocumented instructions derived from them) will work differently than in binary mode.

The ALU works differently than in binary mode:

- the low and high nybble of the Akku will be treated as a BCD value, and when performing operations on it intermediate values will be BCD fixed and carry will be generated on BCD overflows.

The Processor Flags work differently than in binary mode:

- C will work as a carry for multi-byte operations as expected
- N will be equal to bit 7 of some intermediate result (see the respective instruction below)
- V will used the same logic as in binary mode, but some intermediate results will be used (see the respective instruction below)
- Z will be set when the non-BCD operation, before the BCD fixup, would have resulted in $00, no matter what value the result of the BCD operation is.

  example:

```
SED
CLC
LDA #$80
ADC #$80
; A = $60, C = 1, Z = 1
```

invalid BCD

Since only nibble values from 0 to 9 are valid in BCD, it's interesting to see what happens when using A to F. For example:

```
$00+$1F=$25 ("ok" since 10 + $0F = 25)
$10+$1F=$35 ("ok")
$05+$1F=$2A (a non-BCD result, "ok" since 5 + 10 + $0F = 20 + $0A)
$0F+$0A=$1F ("ok", since $0F + $0A = $0F + 10)
$0F+$0B=$10 (?!)
```

… refer to the pseudocode below for details

affected instructions

Surprisingly, only two instructions actually depend on the decimal mode flag: ADC and SBC.

However, all undocumented instructions derived from them are also affected: ARR, RRA, ISC (and the undocumented $eb SBC).

Test code: `CPU/decimalmode/scanner.prg`

ADC

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$79	ADC abs, y	A = A + {addr}	3	4 (+1)	o	o			i		o	x
$7d	ADC abs, x		3	4 (+1)	o	o			i		o	x
$6d	ADC abs		3	4	o	o			i		o	x
$71	ADC (zp),y		2	5 (+1)	o	o			i		o	x
$61	ADC (zp, x)		2	6	o	o			i		o	x
$75	ADC zp, x		2	4	o	o			i		o	x
$65	ADC zp		2	3	o	o			i		o	x
$69	ADC #imm	A = A + #{imm}	2	2	o	o			i		o	x

Operation: add immediate value from accumulator with carry.

Flags

- The N and V flags are set after fixing the lower nybble but before fixing the upper one. They use the same logic as binary mode ADC.
- Z flag is not affected by decimal mode, it will be set if the binary operation would become zero, regardless of the BCD result.
- C flag works as a carry for multi byte operations as expected

Test code: `CPU/decimalmode/adc00.prg CPU/decimalmode/adc01.prg CPU/decimalmode/adc02.prg CPU/decimalmode/adc10.prg CPU/decimalmode/adc11.prg CPU/decimalmode/adc12.prg`

pseudocode

```
/* A = value in Akku, imm = immediate argument, C = carry */

/* Calculate the lower nybble. */
tmp = (A & 0x0f) + (imm & 0x0f) + C;

/* BCD fixup for lower nybble. */
if (tmp > 9) { tmp += 6; }
if (tmp <= 15) {
    tmp = (tmp & 0x0f) + (A & 0xf0) + (imm & 0xf0);
}else{
    tmp = (tmp & 0x0f) + (A & 0xf0) + (imm & 0xf0) + 0x10;
}

/* Zero flag is set just  like in Binary mode. */
Z = ((A + imm + C) & 0xff) ? 0 : 1;

/* Negative and Overflow flags are set with the same logic than in
   Binary mode, but after fixing the lower nybble. */
N = (tmp & 0x80) >> 7;
V = ((A ^ tmp) & 0x80) && !((A ^ imm) & 0x80);

/* BCD fixup for higher nybble. */
if ((tmp & 0x1f0) > 0x90) {
    tmp += 0x60;
}

/* Carry is the only flag set after fixing the result. */
C = (tmp & 0xff0) > 0xf0;

A = tmp;
```

Example: convert a hex digit to ASCII

```
SED
CMP #$0A
ADC #$30
CLD
```

This code converts a hex digit 0 to F (i.e. the accumulator $00 to $0F) to $30 to $39 (for 0 to 9) and $41 to $46 (for A to F). However, this can also be done without using BCD arithmetic, as follows:

```
     CMP #$0A
     BCC SKIP
     ADC #$66 ; Add $67 (the carry is set), convert $0A to $0F --> $71 to $76
SKIP EOR #$30 ; Convert $00 to $09, $71 to $76 --> $30 to $39, $41 to $46
```

Which takes 2 more bytes, but the same number of cycles (or one less if the BCC is taken to the same page).

Example: convert a hex digit to BCD

```
; A contains 0-f (hex)
SED
CLC
ADC #$00
CLD
; A contains 0-15 (BCD)
```

Example: Distinguish NMOS 6502 from CMOS 65C02

```
SED
CLC
LDA #$99
ADC #$01
CLD
```

This code returns with the Z flag set on a 65C02 (the Z flag is valid), and returns with the Z flag clear on a 6502 (the Z flag is invalid, and in this case it does not match the result in the accumulator).

SBC (USBC)

Type: Combination of an immediate and an implied command (Sub-instructions: SBC, NOP)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$f9	SBC addr, y	A = A - {addr}	3	4 (+1)	o	o			i		o	x
$fd	SBC addr, x		3	4 (+1)	o	o			i		o	x
$ed	SBC addr		3	4	o	o			i		o	x
$f1	SBC (zp), y		2	5 (+1)	o	o			i		o	x
$e1	SBC (zp, x)		2	6	o	o			i		o	x
$f5	SBC zp, x		2	4	o	o			i		o	x
$e5	SBC zp		2	3	o	o			i		o	x
$E9	SBC #imm	A = A - #{imm}	2	2	o	o			i		o	x
$EB	SBC #imm		2	2	o	o			i		o	x

Operation: subtract immediate value from accumulator with carry.

The only difference in SBC's operation in decimal mode from binary mode is the result-fixup.

Decimal subtraction is easier than decimal addition, as you have to make the BCD fixup only when a nybble overflows. In decimal addition, you had to verify if the nybble was greater than 9. The processor has an internal "half carry" flag for the lower nybble, used to trigger the BCD fixup. When calculating with legal BCD values, the lower nybble cannot overflow again when fixing it.

So, the processor does not handle overflows while performing the fixup. Similarly, the BCD fixup occurs in the high nybble only if the value overflows, i.e. when the C flag will be cleared.

In binary mode, subtraction has a wraparound effect. For example $00 - $01 = $FF (and the carry is clear). In decimal mode, there is a similar wraparound effect: $00 - $01 = $99, and the carry is clear.

Flags

- The N and V flags are not affected by decimal mode.
- Z flag is not affected by decimal mode, it will be set if the binary operation would become zero, regardless of the BCD result.
- C flag works as a carry for multi byte operations as expected

Test code: `CPU/decimalmode/sbc00.prg CPU/decimalmode/sbc01.prg CPU/decimalmode/sbc02.prg CPU/decimalmode/sbc10.prg CPU/decimalmode/sbc11.prg CPU/decimalmode/sbc12.prg CPU/decimalmode/sbcEB00.prg CPU/decimalmode/sbcEB01.prg CPU/decimalmode/sbcEB02.prg CPU/decimalmode/sbcEB10.prg CPU/decimalmode/sbcEB11.prg CPU/decimalmode/sbcEB12.prg`

pseudocode

```
/* A = value in Akku, imm = immediate argument, C = carry */

tmp = A - imm - (C ^ 1);
tmp2 = (A & 0x0f) - (imm & 0x0f) - (C ^ 1);

C = (tmp < 0x100) ? 1 : 0;
N = (tmp & 0x80) >> 7;
Z = ((tmp & 0xff) == 0) ? 1 : 0;
V = (((A ^ tmp) & 0x80) && ((A ^ imm) & 0x80));

if (tmp2 & 0x10) {
    tmp2 = ((tmp2 - 6) & 0xf) | ((A & 0xf0) - (imm & 0xf0) - 0x10);
} else {
    tmp2 = (tmp2 & 0xf) | ((A & 0xf0) - (imm & 0xf0));
}
if (tmp2 & 0x100) {
    tmp2 -= 0x60;
}

A = tmp2;
```

ARR

Type: Combination of an immediate and an implied command (Sub-instructions: AND, ROR)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$6B	ARR #imm	A = (A & #{imm}) / 2	2	2	o	o			i		o	o

note to ARR: part of this command are some ADC mechanisms.

Operation: In Decimal mode the ARR instruction first performs the AND and ROR, just like in Binary mode. The N flag will be copied from the initial C flag, and the Z flag will be set according to the ROR result, as expected. The V flag will be set if the bit 6 of the accumulator changed its state between the AND and the ROR, cleared otherwise.

If the low nybble of the AND result, incremented by its lowmost bit, is greater than 5, the low nybble in the ROR result will be incremented by 6. The low nybble may overflow as a consequence of this BCD fixup, but the high nybble won't be adjusted. The high nybble will be BCD fixed in a similar way. If the high nybble of the AND result, incremented by its lowmost bit, is greater than 5, the high nybble in the ROR result will be incremented by 6, and the Carry flag will be set. Otherwise the C flag will be cleared.

pseudocode

```
/* A = value in Akku, imm = immediate argument, C = carry */

tmp = A & imm;  /* perform the AND */

/* perform ROR */
tmp2 = tmp | (C << 8);
tmp2 >>= 1;

N = C; /* original carry state is preserved in N */
Z = (tmp2 == 0 ? 1 : 0); /* Z is set when the ROR produced a zero result */
/* V is set when bit 6 of the result was changed by the ROR */
V = ((tmp2 ^ tmp) & 0x40) >> 6;

/* fixup for low nibble */
if (((tmp & 0xf) + (tmp & 0x1)) > 0x5) {
    tmp2 = (tmp2 & 0xf0) | ((tmp2 + 0x6) & 0xf);
}
/* fixup for high nibble, set carry */
if (((tmp & 0xf0) + (tmp & 0x10)) > 0x50) {
    tmp2 = (tmp2 & 0x0f) | ((tmp2 + 0x60) & 0xf0);
    C = 1;
} else {
    C = 0;
}

A = tmp2;
```

Test code: CPU/decimalmode/arr00.prg CPU/decimalmode/arr01.prg CPU/decimalmode/arr02.prg CPU/decimalmode/arr10.prg CPU/decimalmode/arr11.prg CPU/decimalmode/arr12.prg

ISC (ISB, INS)

Type: Combination of two operations with the same addressing mode (Sub-instructions: INC, SBC)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$E7	ISC zp	{addr} = {addr} + 1 A = A - {addr}	2	5	o	o			i		o	x
$F7	ISC zp, x		2	6	o	o			i		o	x
$E3	ISC (zp, x)		2	8	o	o			i		o	x
$F3	ISC (zp), y		2	8	o	o			i		o	x
$EF	ISC abs		3	6	o	o			i		o	x
$FF	ISC abs, x		3	7	o	o			i		o	x
$FB	ISC abs, y		3	7	o	o			i		o	x

Operation: Increase memory by one, then subtract memory from accumulator (with borrow).

This instruction works exactly like INC followed by SBC, with SBC inheriting the decimal mode as described above.

Test code: `CPU/decimalmode/isc00.prg CPU/decimalmode/isc01.prg CPU/decimalmode/isc02.prg CPU/decimalmode/isc03.prg CPU/decimalmode/isc10.prg CPU/decimalmode/isc11.prg CPU/decimalmode/isc12.prg CPU/decimalmode/isc13.prg`

RRA (RRD)

Type: Combination of two operations with the same addressing mode (Sub-instructions: ROR, ADC)

Opc.	Mnemonic	Function	Size	Cycles	N	V	-	B	D	I	Z	C
$67	RRA zp	{addr} = ror {addr} A = A adc {addr}	2	5	o	o			i		o	x
$77	RRA zp, x		2	6	o	o			i		o	x
$63	RRA (zp, x)		2	8	o	o			i		o	x
$73	RRA (zp), y		2	8	o	o			i		o	x
$6F	RRA abs		3	6	o	o			i		o	x
$7F	RRA abs, x		3	7	o	o			i		o	x
$7B	RRA abs, y		3	7	o	o			i		o	x

Operation: Rotate one bit right in memory, then add memory to accumulator (with carry).

This instruction works exactly like ROR followed by ADC, with ADC inheriting the decimal mode as described above.

Test code: `CPU/decimalmode/rra00.prg CPU/decimalmode/rra01.prg CPU/decimalmode/rra02.prg CPU/decimalmode/rra03.prg CPU/decimalmode/rra10.prg CPU/decimalmode/rra11.prg CPU/decimalmode/rra12.prg CPU/decimalmode/rra13.prg`

Appendix

Opcode naming in different Assemblers

Opc	imp	imm	zp	zpx	zpy	izx	izy	abs	abx	aby	ind	rel	KickAss	Acme	ca65	dasm	64tass
SLO			$07	$17		$03	$13	$0F	$1F	$1B			SLO	SLO	SLO	SLO	SLO
RLA			$27	$37		$23	$33	$2F	$3F	$3B			RLA	RLA	RLA	RLA	RLA
SRE			$47	$57		$43	$53	$4F	$5F	$5B			SRE	SRE	SRE	SRE	SRE
RRA			$67	$77		$63	$73	$6F	$7F	$7B			RRA	RRA	RRA	RRA	RRA
SAX			$87		$97	$83		$8F					SAX	SAX	SAX	SAX	SAX
LAX			$A7		$B7	$A3	$B3	$AF		$BF			LAX	LAX	LAX	LAX	LAX
DCP			$C7	$D7		$C3	$D3	$CF	$DF	$DB			DCP	DCP	DCP	DCP	DCP, DCM
ISC			$E7	$F7		$E3	$F3	$EF	$FF	$FB			ISC	ISC	ISC	ISB	ISC, INS, ISB
ANC		$0B											ANC	ANC	ANC	ANC	ANC
ANC		$2B											ANC2				
ALR		$4B											ALR	ASR	ALR	ASR	ALR, ASR
ARR		$6B											ARR	ARR	ARR	ARR	ARR
SBX		$CB											AXS	SBX	AXS	SBX	AXS, SBX
SBC		$EB											SBC2				
SHA							$93			$9F			AHX	SHA	SHA	SHA	AHX, SHA
SHY									$9C				SHY	SHY	SHY	SHY	SHY
SHX										$9E			SHX	SHX	SHX	SHX	SHX
TAS										$9B			TAS	TAS	TAS	SHS	TAS, SHS
LAS										$BB			LAS	LAS	LAS	LAS	LAS, LAE, LDS
LAX		$AB											LAX	LXA	LAX	LXA	LAX, LXA
ANE		$8B											XAA	ANE	ANE	ANE	XAA, ANE
NOP								$0C	$1C					TOP	NOP	NOP	NOP
NOP		$80	$04	$14										DOP	NOP	NOP	NOP
NOP	$1A																
NOP	$3A	$82	$44	$34					$3C								
NOP	$5A	$C2	$64	$54					$5C								
NOP	$7A	$E2		$74					$7C								
NOP	$DA	$89		$D4					$DC								
NOP	$FA			$F4					$FC								
JAM	$02	$12	$22	$32	$42	$52	$62	$72	$92	$B2	$D2	$F2		JAM	JAM		JAM

Combined Examples

negating a 16bit number

Another trick that makes use of the SBX command is the negation of a 16 bit number:

```
LAX #$00  ;should be safe, as #$00 is loaded
SBX #lo   ;sets carry automatically for upcoming sbc
SBC #hi
; negated value is in A/X
```

One might also think of extending this trick to negate two 8 bit numbers (A, X) at a time.

a smart addition

A second case in which to use SBX is in combination with LAX, for example when doing:

```
LDA $02
CLC
ADC #$08
TAX
```

that can be easily substituted by:

```
LAX $02         ;A = X = M [$02]
SBX #$f8        ;X = (A & X) - -8
```

So we saved 4 cycles here, as the state of the carry is of no interest for the subtract done by SBX, which is one of its big advantages. Thus we could also fake an ADD or SUB with that command. The and-operation is not needed here, but does not harm. If there's use for it, just let A or X be loaded with the right value for the and-mask.

Multiply 8bit * 2 ^ n with 16bit result

If you want to set up a reference into a table of 8-byte objects use:

```
LAX Index,y         ; 4 A,X = (index+Y)
AND #%00000111      ; 2
STA AddressHi       ; 3 store A & %00000111
LDA #%11111000      ; 2
SAX AddressLo       ; 3 store X & %11111000
                    ; = 14 cycles
```

Which is a hell of a lot faster than multiplying by 8, and just means storing the values in the index in a funny bit order (43210765)

6 sprites over FLI

The '6 sprites over FLI' routine used in 'Darwin' is based on the following code. It uses unintended Read-Modify-Write opcodes since they have a side-effect on the accumulator. This is needed because there is no time to load it explicitly with LDA #. *'Finding this combination with usable side-effects took 6 months (duration, not effort) and the game to find a second solution has been rightfully named FLI-Sudoku :)'*

First column in the comments show cycles, second the actual value written, and third the effective bits.

```
; A=$A0 X=$36 Y=$21
; $d018=$1f $dd00=$3d $dd02=$36

STA $D011      ;4 A0 (20)
SRE $DD02      ;6 1b (03) A:A0 -> BB
STY $D011      ;4 21 (21)
ASL $D018      ;6 3f (38)
SAX $D011      ;4 32 (22)
STY $DD02      ;4 21 (01)
STA $D011      ;4 BB (23)
SRE $D018      ;6 1f (18) A:BB -> A4
STA $D011      ;4 A4 (24)
RRA $DD02      ;6 90 (00) A:A4 -> 35
STA $D011      ;4 35 (25)
SLO $D018      ;6 3f (38) A:35 -> 3F
STX $D011      ;4 36 (26)
STX $DD02      ;4 36 (02)
STA $D011      ;4 3F (27)
SRE $D018      ;6 1f (18) A:3F -> 20
```

This block is repeated for every 8 lines of the graphics area, with every second block using $20 as a start value for the accumulator like this:

```
; A=$20 $d018=$1f $dd02=$36

STA $D011      ;4 20 (20)
SRE $DD02      ;6 1b (03) A:20 -> 3B
STY $D011      ;4 21 (21)
ASL $D018      ;6 3f (38)
SAX $D011      ;4 32 (22)
STY $DD02      ;4 21 (01)
STA $D011      ;4 3B (23)
SRE $D018      ;6 1f (18) A:3B -> 24
STA $D011      ;4 24 (24)
RRA $DD02      ;6 90 (00) A:24 -> B5
STA $D011      ;4 B5 (25)
SLO $D018      ;6 3f (38) A:B5 -> BF
STX $D011      ;4 36 (26)
STX $DD02      ;4 36 (02)
STA $D011      ;4 BF (27)
SRE $D018      ;6 1f (18) A:BF -> A0
; A=$A0 $d018=$1f $dd02=$36
```

References

- http://www.oxyron.de/html/opcodes02.html
- http://www.ataripreservation.org/websites/freddy.offenga/illopc31.txt
- http://www.ffd2.com/fridge/docs/6502-NMOS.extra.opcodes
- http://visual6502.org/wiki/index.php?title=6502_Unsupported_Opcodes
- http://www.pagetable.com/?p=39
- http://codebase64.org/doku.php?id=base:decrease_x_register_by_more_than_1
- http://codebase64.org/doku.php?id=base:some_words_about_the_anc_opcode
- http://codebase64.org/doku.php?id=base:advanced_optimizing
- http://www.atariage.com/forums/topic/168616-lxa-stable/#entry2092077
- Emulator Test-suite by Wolfgang Lorenz
- Test programs by Poitr Fusik
- First CSDb "Unintended OpCode coding challenge": http://csdb.dk/event/?id=2417

Greets and Thanks

- Segher
- Bitbreaker/Oxyron
- Graham/Oxyron
- Mist/R.O.L.E.
- pwsoft
- Ninja/The Dreams
- Count Zero/Cyberpunx
- Unseen/VICE Team
- TLR/VICE Team
- Krill/Plush
- Wolfgang Lorenz
- WoMo
- 0xF/Taquart
- Soci/Singular^VICE Team
- Peiselulli/TRSI^Oxyron
- SvOlli/XayaX
- Marco Baye
- Wilfred Bos
- Kabuto/Latency
- Color Bar
- JAC!
- … and all contributors to codebase64, visual6502, VICE and last not least the dark knights behind the scenes who shall remain unmentioned - you know who you are.

Wanted

This document could still be improved and extended, contributions welcome! if you want to help send your contributions to groepaz@gmx.net !

- More examples (most interesting would be examples for the opcodes that still have none, i.e. RRA, and especially the weird TAS)
- More/better test cases:
  - Some opcodes, such as ARR, should also be tested on a disk drive while data is being read.
  - 'unstable address hi byte' opcodes' page boundary crossing behaviour needs to be verified on more CPUs. The anomaly described in some sources seems to exist on some machines only.
  - ‘magic constant’ group still needs more testing on real gear. At least some of the supposed instabilities might simply be the RDY line interaction being misinterpreted.
- Examples of interesting (ab)use of the decimal mode
- Experienced 6502 coders from other platforms (Atari 2600/800, Apple II, VIC-20, Plus 4 …) who port the test cases and check them on other 6502 variants and platforms.

History

- December 24th, 2018 (V0.93) – added description on cpu flags naming, flag usage is a bit more detailed in tables, added some details on decimal mode, In some descriptions flipped the order of sub instructions around to match the logical order, added missing note on the RDY line dependency of ANE and LAX, last not least all sections have proper headers now.
- December 24th, 2017 (V0.92) – Added a couple unusual Mnemonics used by the Atari-centric MAD-Assembler, use “Andale Mono” instead of “Aerial Mono” - the later would produce broken ligatures. A few formatting fixes. Fixed description of the page-crossing anomaly of “unstable address high byte” group.
- December 24th, 2016 (V0.91) – Fixed some typos, added a few more examples.
- December 24th, 2015 (V0.9) – fixed cosmetical issues (justification), fixed link(s) in references, added notes on ANE/LAX#imm usage, added chapter about unintended addressing modes, added references to test code from 64doc.txt, added note on decimal flag for RRA and ISC, fixed error in ANE example, added examples for RLA and LAS (including great explanation by Color Bar, thanks!)
- December 24th, 2014 – first public release
- November 2014 – finally found the time to clean up this document and showed it to a bunch of people for proof reading (unreleased)
- some time 2013 – started pasting together various information for personal use

The Truth is out there