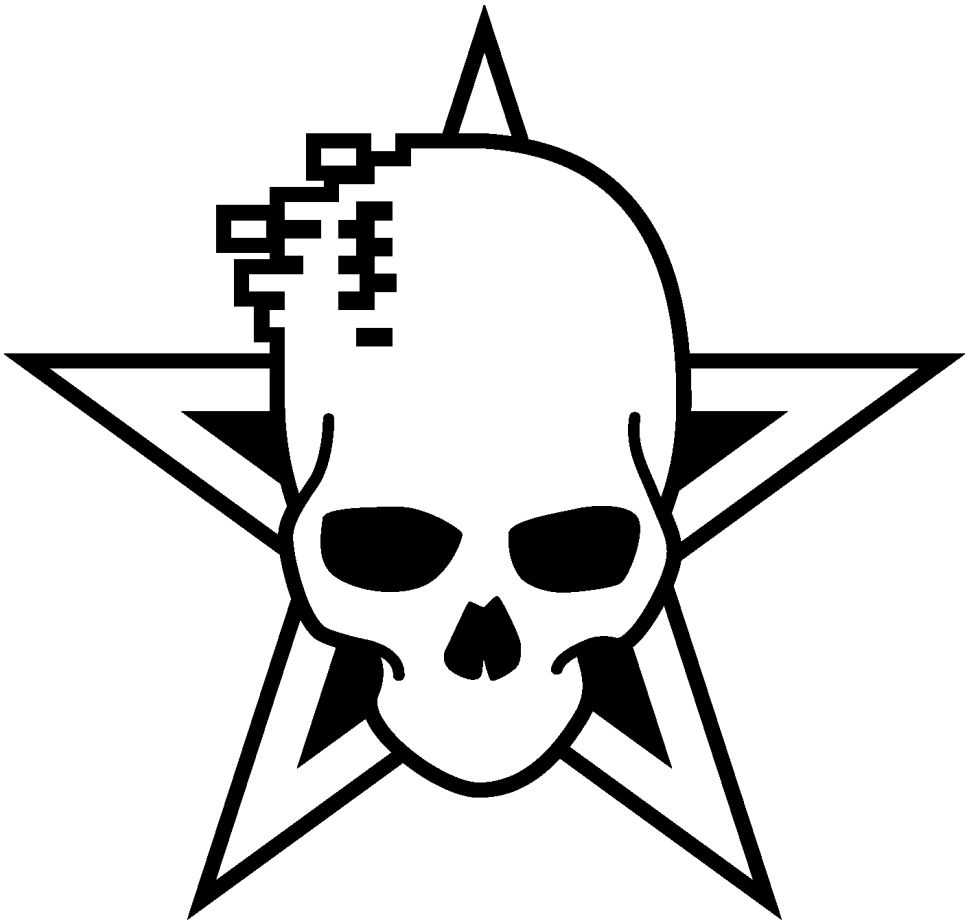
NMOS 6510 Unintended Opcodes

no more secrets

(v0.99 - 24/12/24)



Contents

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)

24/12/24 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 “SALLY” 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)

The 6507 used in the Atari 2600 VCS appears to work the same with some limitations: the address bus is limited to 13 bits - the top three bits are not available on the chip package. Also missing on the chip package are the IRQ and NMI lines, which are hard wired to "+5V" internally.

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

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

    https://sourceforge.net/p/vice-emu/code/HEAD/tree/testprogs/
  - examples for real world usage, if available
- Some hints on using decimal mode in (not only) unintended ways
- Description of the so called “dummy” memory accesses and some examples on how to (ab)use them
- A short description of all other unintended bugs and quirks of the CPU

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 Subtraction
*	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

Opcode Matrix

The instructions of the 6502 are compressed into a 130-entry decode ROM. Instead of 256 entries telling how to process each separate opcode, it's encoded as combinatorial logic post-processing the output of a "sparse" ROM that acts in some ways like a programmable logic array (PLA).

Many instructions activate multiple lines of the decode ROM at once. Often this is on purpose, such as one line for the addressing mode and one for the opcode part. But many of the unintended opcodes simultaneously trigger parts of the ROM that were intended for completely unrelated instructions.

If we arrange the opcode matrix in a slightly different way than it is usually done, we can show some interesting symmetries:

	A: Control Instructions + Load/Store Y								**B: ALU Operations + Load/Store A**							
	+00	+04	+08	+0C	+10	+14	+18	+1C	+01	+05	+09	+0D	+11	+15	+19	+1D
00	BRK	NOP zp	PHP	NOP abs	BPL rel	NOP zp,x	CLC	NOP abs,x	ORA (zp,x)	ORA zp	ORA #imm	ORA abs	ORA (zp),y	ORA zp,x	ORA abs,y	ORA abs,x
20	JSR abs	BIT zp	PLP	BIT abs	BMI rel	NOP zp,x	SEC	NOP abs,x	AND (zp,x)	AND zp	AND #imm	AND abs	AND (zp),y	AND zp,x	AND abs,y	AND abs,x
40	RTI	NOP zp	PHA	JMP abs	BVC rel	NOP zp,x	CLI	NOP abs,x	EOR (zp,x)	EOR zp	EOR #imm	EOR abs	EOR (zp),y	EOR zp,x	EOR abs,y	EOR abs,x
60	RTS	NOP zp	PLA	JMP (ind)	BVS rel	NOP zp,x	SEI	NOP abs,x	ADC (zp,x)	ADC zp	ADC #imm	ADC abs	ADC (zp),y	ADC zp,x	ADC abs,y	ADC abs,x
80	NOP #imm	STY zp	DEY	STY abs	BCC rel	STY zp,x	TYA	SHY abs,x	STA (zp,x)	STA zp	NOP #imm	STA abs	STA (zp),y	STA zp,x	STA abs,y	STA abs,x
A0	LDY #imm	LDY zp	TAY	LDY abs	BCS rel	LDY zp,x	CLV	LDY abs,x	LDA (zp,x)	LDA zp	LDA #imm	LDA abs	LDA (zp),y	LDA zp,x	LDA abs,y	LDA abs,x
C0	CPY #imm	CPY zp	INY	CPY abs	BNE rel	NOP zp,x	CLD	NOP abs,x	CMP (zp,x)	CMP zp	CMP #imm	CMP abs	CMP (zp),y	CMP zp,x	CMP abs,y	CMP abs,x
E0	CPX #imm	CPX zp	INX	CPX abs	BEQ rel	NOP zp,x	SED	NOP abs,x	SBC (zp,x)	SBC zp	SBC #imm	SBC abs	SBC (zp),y	SBC zp,x	SBC abs,y	SBC abs,x
	+02	+06	+0A	+0E	+12	+16	+1A	+1E	+03	+07	+0B	+0F	+13	+17	+1B	+1F
00	JAM	ASL zp	ASL	ASL abs	JAM	ASL zp,x	NOP	ASL abs,x	SLO (zp,x)	SLO zp	ANC #imm	SLO abs	SLO (zp),y	SLO zp,x	SLO abs,y	SLO abs,x
20	JAM	ROL zp	ROL	ROL abs	JAM	ROL zp,x	NOP	ROL abs,x	RLA (zp,x)	RLA zp	ANC #imm	RLA abs	RLA (zp),y	RLA zp,x	RLA abs,y	RLA abs,x
40	JAM	LSR zp	LSR	LSR abs	JAM	LSR zp,x	NOP	LSR abs,x	SRE (zp,x)	SRE zp	ALR #imm	SRE abs	SRE (zp),y	SRE zp,x	SRE abs,y	SRE abs,x
60	JAM	ROR zp	ROR	ROR abs	JAM	ROR zp,x	NOP	ROR abs,x	RRA (zp,x)	RRA zp	ARR #imm	RRA abs	RRA (zp),y	RRA zp,x	RRA abs,y	RRA abs,x
80	NOP #imm	STX zp	TXA	STX abs	JAM	STX zp,y	TXS	SHX abs,y	SAX (zp,x)	SAX zp	ANE #imm	SAX abs	SHA (zp),y	SAX zp,y	TAS abs,y	SHA abs,y
A0	LDX #imm	LDX zp	TAX	LDX abs	JAM	LDX zp,y	TSX	LDX abs,y	LAX (zp,x)	LAX zp	LAX #imm	LAX abs	LAX (zp),y	LAX zp,y	LAS abs,y	LAX abs,y
C0	NOP #imm	DEC zp	DEX	DEC abs	JAM	DEC zp,x	NOP	DEC abs,x	DCP (zp,x)	DCP zp	SBX #imm	DCP abs	DCP (zp),y	DCP zp,x	DCP abs,y	DCP abs,x
E0	NOP #imm	INC zp	NOP	INC abs	JAM	INC zp,x	NOP	INC abs,x	ISC (zp,x)	ISC zp	SBC #imm	ISC abs	ISC (zp),y	ISC zp,x	ISC abs,y	ISC abs,x
	C: RMW Operations + Load/Store X								**D: Unintended Operations**							

- Variants of the same instruction in a block are mostly in the same row.
- Instructions in the same column have (mostly) the same addressing mode, with the following exceptions:
  - `JSR abs` (expected implied) (but 00,20,40,60 are all „stack“)
  - `JMP (ind)` (expected abs)
  - `STX zp,y` and `LDX zp,y` (can‘t be `zp,x`)
  - `SHX abs,y` and `LDX abs,y` (can‘t be `abs,x`)
  - `SAX zp,y` and `LAX zp,y` (can‘t be `zp,x`)
  - `SHA abs,y` and `LAX abs,y` (can‘t be `abs,x`)

Other conclusions:

- all `JAM`s are empty „stack“ or „relative“ instructions
- In blocks A, B and C all unused instructions turn into `NOP`s (except for the `JAM`s) with the expected addressing modes. The only exceptions from this are opcodes 9C and 9E, which appear to be „non working“ `STY abs,x` and `STX abs,y` respectively.
- `NOP #imm` in block B is “`STA #imm`” (which makes no sense)
- All instructions in block D are unintended instructions. These “combine” (not necessarily all of) the sub-operations of instructions from the ALU operation at the same position in the same column and RMW operation at the same position in the same row, all of them having the same addressing mode as the corresponding ALU operation in the same column, with the four exceptions listed above.
- `LAX #imm` combines `LDA #imm` with `TAX`, which makes *some* sense at least (but still does not explain the weird unstable behaviour and/or the “magic constant”)
- `ANE #imm` combines “`STA #imm`” with `TXA`, which makes no sense at all on a first look, but might contribute to the ANDing of the Akkumulator, X-Register and immediate value.

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	x
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.

- The leftmost bit is shifted into the carry flag
- N and Z are set after the ORA

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 arithmetic left shift and load lowest byte

Instead of:

```
ASL data+0          ; (6) lo byte
ROL data+1          ; (6)
ROL data+2          ; (6) hi byte
LDA data+0          ; (4)
```

you can write: (which is shorter, and faster)

```
LDA #0              ; (2)
; A must be zero before reaching here
SLO data+0          ; (6) lo byte
ROL data+1          ; (6)
ROL data+2          ; (6) hi byte
```

Example: simulate extra addressing modes for ASL

If you can afford clobbering `A` and the flags (or even use the final value), `SLO` turns into `ASL` and makes some addressing modes available that do not exist for regular `ASL`:

```
; 0 is shifted into the LSB
SLO abs, y      ; like ASL abs, y
SLO (zp), y     ; like ASL (zp), y
SLO (zp, x)     ; like ASL (zp, x)
; A was changed as with additional ORA
; N,Z were set as with additional ORA
```

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.

- Carry is shifted in as LSB and bit 7 is shifted into Carry
- N and Z are set according to the AND

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

Example: simulate extra addressing modes for ROL

If you can afford clobbering `A` and the flags (or even use the final value), `RLA` turns into `ROL` and makes some addressing modes available that do not exist for regular `ROL`:

```
; C is shifted into the LSB
RLA abs, y     ; like ROL abs, y
RLA (zp), y    ; like ROL (zp), y
RLA (zp, x)    ; like ROL (zp, x)
; A was changed as with additional AND
; N,Z were set as with additional AND
```

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.

- LSB is shifted into the carry flag
- N and Z are set after the EOR

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
+
```

Example: compute the parity of a byte

Parity means counting the number of set bits, if the number of set bit is odd the parity is odd, if the number of set bit is even the parity is even (Note that zero is even). This is often used in transfer protocols as a simple error detection mechanism.

```
; byte is in A
STA temp
SRE temp
SRE temp
SRE temp
SRE temp
SRE temp
SRE temp
SRE temp
SRE temp
AND #$01
; A=0 and flag Z=1 if even parity
; A=1 and flag Z=0 if odd parity
```

Example: simulate extra addressing modes for LSR

If you can afford clobbering `A` and the flags (or even use the final value), `SRE` turns into `LSR` and makes some addressing modes available that do not exist for regular `LSR`:

```
; 0 is shifted into the MSB
SRE abs, y     ; like LSR abs, y
SRE (zp), y    ; like LSR (zp), y
SRE (zp, x)    ; like LSR (zp, x)
; A was changed as with additional EOR
; N,Z were set as with additional EOR
```

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).

- LSB is shifted into the carry flag and carry flag is shifted into bit 7 by the ROR
- then all flags are set according to the ADC

This instruction inherits the decimal flag dependency from ADC. For the behaviour in decimal mode see Unintended decimal mode: RRA (RRD).

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/droradc.prg`

Example: noise LFSR

If you need a fast “noise” generator, something like this could work:

```
LDA  #$e4 ; initial seed
STA  zp1
LDA  #$01 ; initial seed
CLC

...
; restore accu and carry
RRA  zp1
EOR  #$01
ROR
; “noise” value in accu
; preserve accu and carry
...
```

Example: simulate extra addressing modes for ROR

If you can afford clobbering A and the flags (or even use the final value), RRA turns into ROR and makes some addressing modes available that do not exist for regular ROR:

```
; C is shifted into the MSB
RRA abs, y     ; like ROR abs, y
RRA (zp), y    ; like ROR (zp), y
RRA (zp, x)    ; like ROR (zp, x)
; A was changed as with additional ADC
; N,V,Z,C were set as with additional ADC
```

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.

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, TEA).

'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: 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 #$01              ;A=%00000001
LDX #$fe              ;X=%11111110
SAX screen + $3f8     ;$00
STA screen + $3f9     ;$01
LDA #$03              ;A=%00000011
SAX screen + $3fa     ;$02
STA screen + $3fb     ;$03
LDA #$05              ;A=%00000101
SAX screen + $3fc     ;$04
STA screen + $3fd     ;$05
LDA #$07              ;A=%00000111
SAX screen + $3fe     ;$06
STA screen + $3ff     ;$07
```

Alternatively you can swap the roles of A and X, and you get the following:

```
LDX #$04          ;X=%00000100
LDA #$fb          ;A=%11111011
SAX screen + $3f8 ;$00
STX screen + $3fc ;$04
INX               ;X=%00000101
SAX screen + $3f9 ;$01
STX screen + $3fd ;$05
INX               ;X=%00000110
SAX screen + $3fa ;$02
STX screen + $3fe ;$06
INX               ;X=%00000111
SAX screen + $3fb ;$03
STX screen + $3ff ;$07
```

This does not save any cycles, but 3 bytes.

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: simulate LDA zp, y

In situations where you can afford trashing the X register, you can save one byte by using `LAX zp,y` instead of `LDA abs, y`. For example the following code copies some code to the zeropage and at the same time saves the zeropage contents on the stack:

Instead of this:

```
                LDY #PAYLOAD_LENGTH
-
                LDA <ZEROPAGE_ ADDRESS – 1,y      ; LDA abs, y
                LDX PAYLOAD_ADDRESS - 1,y
                STX <ZEROPAGE_ ADDRESS - 1,y
                PHA
                DEY
                BNE -
```

you can do this, which is one byte shorter:

```
                LDY #PAYLOAD_LENGTH
-
                LAX <ZEROPAGE_ ADDRESS – 1,y      ; LAX zp,y
                LDX PAYLOAD_ADDRESS - 1,y
                STX <ZEROPAGE_ ADDRESS - 1,y
                PHA
                DEY
                BNE -
```

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 do:

```
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.

- N / Z / C are set according to the compare, after the decrement

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 +
          DEC ptr+1
+
          ;carry is set always for free
```

Example: decrement indirect indexed memory

The 6502 has no `DEC (zp), y` instruction, so with legal opcodes one would write:

```
LDY #$00
LDA (zp),Y
SEC
SBC #$01
STA (zp),Y
```

instead you can use:

```
LDY #$00
DCP (zp), y
```

which is 5 bytes smaller, 7 cycles faster and leaves A untouched (but changes N, Z, C flags).

As a bonus, you get a compare with the result for free, so instead of:

```
LDY #$00
LDA (zp),Y
SEC
SBC #$01
STA (zp),Y
LDA #$42       ; value to compare with in A
CMP (zp),Y
; react on N, Z, C
```

You can use:

```
LDA #$42       ; value to compare with in A
LDY #$00
DCP (zp), y
; react on N, Z, C
```

which is 7 bytes smaller, 12 cycles faster and you can load A any time before.

Example: decrementing 16bit value with lowbyte in Akku and Carry always set

In the case when you want to decrement a 16bit value with the lowbyte in the akkumulator and you need the carry set after the operation, you can also use DCP:

instead of this:

```
                ; lowbyte is in A, carry is set
                SBC #$01
                BCS +
                ; carry is cleared
                DEC highbyte
                SEC                 ; set carry again
+
```

you can write this, which saves one byte and two cycles:

```
                ; lowbyte is in A, carry is set
                SBC #$01
                BCS +
                ; carry is cleared, Akku is $FF
                DCP highbyte        ; since Akku is $FF, the CMP
                                    ; in DCP will set the carry
+
```

Example: simulate extra addressing modes for DEC

If you can afford clobbering the flags (or even use the result of the CMP), DCP turns into DEC and makes some addressing modes available that do not exist for regular DEC:

```
          DCP abs, y     ; like DEC abs, y
          DCP (zp), y    ; like DEC (zp), y
          DCP (zp, x)    ; like DEC (zp, x)
          ; N,Z,C were set as with additional CMP
```

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).

- C is affecting the SBC, and SBC sets N / V / Z / C as expected

This instruction inherits the decimal flag dependency from SBC. For the behaviour in decimal mode see Unintended decimal mode: ISC (ISB, INS).

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 nooverflow
```

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

```
LDA #ENDVALUE
SEC
ISC counter
BNE nooverflow
STA counter    ; Bonus: A is always 0 here
```

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
```

Example: simulate extra addressing modes for INC

If you can make sure that A is 0 and C is cleared, ISC turns into INC and makes some addressing modes available that do not exist for regular INC:

```
; A is zero and C=0 before reaching here
ISC abs, y     ; like INC abs, y
ISC (zp), y    ; like INC (zp), y
ISC (zp, x)    ; like INC (zp, x)
```

ANC (ANC2, ANA, ANB)

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.

- Bit 1 (after the AND) is shifted into the carry flag
- N and Z are set after the shift

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	x

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.

The following applies to when the decimal flag is clear, for the behaviour in decimal mode see Unintended decimal mode: ARR.

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 will be equal to the state of bit 7 before (or bit 6 after) the rotate, the state of the overflow-flag depends on whether the rotate changes bit 6, and will be set like shown in the following table:

Input before ROR			Output			
Carry	Bit 7	Bit 6	Carry = Input Bit 7	Overflow = Input Bit 7 ^ Input Bit 6	Bit 7 = Input Carry	Bit 6 = Input Bit 7
0	0	0	0	0	0	0
0	0	1	0	1	0	0
0	1	0	1	1	0	1
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	1	0	1	1	0
1	1	0	1	1	1	1
1	1	1	1	0	1	1

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

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, XMA)

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)
- N and Z are set according to the value ending up in X

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 #$0f       ;2 set up mask beforehand,
                         ;  can be reused for each turn
          STA $02        ;2

          LDA $0400,y    ;4
          BIT $02        ;2 apply mask without destroying A
          BNE +          ;2
          CLC            ;2
          ADC #$10       ;2
+
          SEC            ;2 we need to set carry
          SBC #$11       ;2
                         ;= 20
```

… can be substituted by …

```
          LDA $0400,y    ;4 load value
          LDX #$0f       ;2 set up mask
          SBX #$00       ;2 check if low nibble underflows
                         ;  -> X = A & $0f
          BNE +          ;2 all fine, decrement both nibbles
                         ;  the cheap way, carry is set!
          SBC #$f0       ;2 do wrap around by hand
          SEC            ;2
+
          SBC #$11       ;2 decrement both nibbles,
                         ;  carry is set already by sbx
                         ;=16
```

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.

Example: toggle one bit and set another depending on carry

The different behaviour regarding flags can be useful in tight loops where you need to toggle one bit in a value on each loop iteration, and another bit depending on the state of the carry flag.

A good usecase for this is when dealing with a clock- and a data bit:

```
          ; A contains the data
          sec
          ror ; shift one 1 into the bitpattern, MSB to carry
          sta databits

          lda #$3f ; startvalue, data bit must be 1 here
loop:
          eor #$20 ; toggle clock bit
          ; move value to X to disarm the AND in SBX
          tax                 ; X = $1f/$3f (data bit set)
          bcc +
          ; substract without carry (unset data bit)
          sbx #$10            ; X = $0f/$2f
+
          ; do something with the value in X (preserve A!)

          lsr databits ; shift next bit into the carry, break
          bne loop     ; the loop when all bits are shifted out
```

SBC (SBC2, USBC, USB)

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.

- N and Z are set as expected by a load instruction

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 table,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, UNP)

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								

Note: One of the “classic” sources claims that NOP opcodes $82, $C2, $E2 may be JAMs. Since neither looking at the way these opcodes are decoded can back this up, nor any other sources corroborate this, it is probably plain wrong, or at least must be true only 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 (DOP, SKB, IGN)

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.

NOP (TOP, SKW, IGN)

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
```

Example: skip instructions

For skipping instructions, the undocumented NOPs are better than the commonly used BIT ($24, $2c) as they do not touch any registers or flags, perfect for merging two code pathes:

```
entry1:
          LDX #$00
          TOP       ; $0C instead of $2C to skip next instruction
entry2:
          LDX #$20  ; two byte instruction
common_path:
```

or

```
entry1:
          SEC
          DOP       ; $04 instead of $24 to skip next instruction
entry2:
          CLC       ; one byte instruction
common_path:
```

Example: skip instructions

Sometimes you want to “disable” an instruction, like above using the undocumented NOPs is better for this than other instructions. It may also be an advantage that some instructions only differ by one bit – eg JMP ($4C) and TOP ($0C)

```
          JMP function        ;enabled ($4c)
     →    TOP function        ;disabled ($0c)
```

Example: interleave code and data

Sometimes code uses tables that have “holes” in them, ie only certain offsets in the table will actually be used. In such cases you might be able to use less memory overall by interleaving such tables with code, at the cost of 2 cycles per data value that must be skipped.

```
          ; non performance critical code start
something:
          LDX foo
          LDA coltable1,x
          STA $D020
          .byte $80      ; NOP #IMM (DOP)

          ; data values at offset $00,$08,$10 interleaved
          ; into the code
sometable:
          .byte $55

          LDA coltable2, x
          STA $D021
          .byte $80      ; NOP #IMM (DOP)
          .byte $aa

          LDA coltable3, x
          STA $D022
          .byte $80      ; NOP #IMM (DOP)
          .byte $ff
          ...

otherthing:
          ...
          ; X contains $00, $08, $10...
          LDA sometable, x
          ...
```

JAM (KIL, HLT, CIM, CRP)

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
```

Simulation link:
http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=12&d=027fff00&loglevel=4

Unstable Opcodes

Out of all opcodes, just **seven** fall into the so called 'unstable' category. This is where in earlier documents often the half esoteric black magic started, and what took most time and effort to update and fix for the current state of this version of the truth: **Only two of those seven opcodes are actually unstable in the sense that they may produce a truly unpredictable result.** The other five opcodes actually produce predictable results – but the conditions under which they do that and the produced results are a bit unexpected. **All seven opcodes can be used safely when certain preconditions are met.**

'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 will get incremented as expected, and then ANDed with the value stored. For this reason **you should generally keep your index in a range that page boundaries are not crossed**.

- 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 last cycle of the instruction. '*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.'* **If you make sure no DMA starts when any of these opcodes executes, the value written is always ANDed with the highbyte of the target address, plus one.**

	SHA (zp), y	TAS abs, y	SHY abs, x	SHX abs, y	SHA abs,y
Opcode	$93	$9b	$9c	$9e	$9f
Value	A & X	A & X	Y	X	A & X
Cycle N	5	4	4	4	4

	Highbyte of address written to		Value written	
	No DMA in Cycle N	DMA in Cycle N	No DMA in Cycle N	DMA in Cycle N
Page not crossed	{H}	{H}	Value & {H+1}	Value
Page crossed	{H+1} & Value	{H+1} & Value	Value & {H+1}	Value

'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, TEA)

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:

- The value written is ANDed with `&{H+1}`, except when the RDY line goes low in the 4th (opcode `$9f`) or 5th (opcode `$93`) cycle.
- When adding `Y` to the target address causes a page boundary crossing, the highbyte of the target address is incremented by one (as expected), and then ANDed with `(A & X)`.

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+1} drop off: `CPU/sha/shazpy2.prg CPU/sha/shazpy3.prg CPU/sha/shaabsy2.prg CPU/sha/shaabsy3.prg CPU/sha/shazpy4.prg CPU/sha/shaabsy4.prg`
- page boundaries: `CPU/sha/shazpy1.prg CPU/sha/shaabsy1.prg CPU/sha/shazpy5.prg CPU/sha/shaabsy5.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, TEX)

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:

- The value written is ANDed with &{H+1}, except when the RDY line goes low in the 4th cycle.
- When adding Y to the target address causes a page boundary crossing, the highbyte of the target address is incremented by one (as expected), and then ANDed with X.

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+1} drop off: CPU/shxy/shxy2.prg, CPU/shxy/shxy3.prg, CPU/shxy/shxy4.prg, CPU/shxy/shx-t2.prg, CPU/shxy/shx-test.prg
- page boundaries: CPU/shxy/shxy1.prg, CPU/shxy/shxy5.prg

Simulation links:

- &{H+1} drop off: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=20&d=a27fa0f39e0211&logmore=rdy&rdy0=15&rdy1=16
- page boundary crossing anomaly: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=18&d=a27fa0f39e0f11
- &{H+1} drop off, plus page boundary crossing anomaly: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=20&d=a27fa0f39e0f11&logmore=rdy&rdy0=15&rdy1=16

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
```

Example: Sync with raster beam (remove cycle variance)

The following snippets can be used as a replacement for the commonly used “half variance” $D012 polling loop, to synchronise the code to a fixed position before setting up a timer that is then synced to the raster beam.

You can sync to the raster beam in order to remove jitter in just 9 bytes by doing this:

```
          * = $0f00 ; Some address with (H+1) & 1 = 0
                    ; and (H+1) & $10 = $10

          LDY #$00
loop:     LDX #$11
          SHX cont, y
cont:     BPL loop
```

It uses the fact that we will AND the written value with H+1 unless a badline pauses the CPU between the third and fourth cycle of shx. The latter then changes the "bpl" into an "ora" and drops us out of the loop at horizontal position 61.

This variant works at any address. It is required that $A0 holds a value <$80 before the routine is started. A good init value would be $01, since this will be restored by the last loop iteration.

The code can be written in two ways, the following two snippets are the same piece of code. The first shows what executes when the code was started:

```
loop = * + 1
          LDX #$B5        ; initialize X
          SBC #$9E        ; SBC-opcode $eb (Accu does not matter)
          LDY #$00        ; initialize Y
          BPL loop
```

now the BPL branches to the operand of the LDX, so the loop looks as follows:

```
          !byte $a2       ; LDX
loop:     LDA $EB,X
          SHX $00A0,Y
          BPL loop
```

The SHX stores a value to zp-adress $A0. Now the unintended SBC-opcode reveals its real magic: the hex-value $EB. Performing the LDA $EB,X, with X=$B5, ends up reading the value from $A0. Since SHX does not influence the flags, the branch solely depends on what was read with LDA $EB,X – which in turn is the value that was written by the SHX in the loop iteration before. Now as long as we have the &H+1 in play when executing the SHX, the value stored to $A0 will be $01 (since H+1=$01 and X=$B5). When the &H+1 does not occur, the full value $B5 is written to $A0, which is read in the next loop iteration, effectively ending the whole loop ($B5 -> Accu sets the N-flag!).

This syncing approach works just like the variant above: when a badline pauses the CPU on the 4th SHX cycle, value $B5 is written to $A0, so the cycle when this happens is known, and consequently also the cycle position off the loop's end.

SHY (A11, SYA, SAY, TEY)

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:

- The value written is ANDed with &{H+1}, except when the RDY line goes low in the 4th cycle.
- When adding X to the target address causes a page boundary crossing, the highbyte of the target address is incremented by one (as expected), and then ANDed with Y.

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+1} drop off: CPU/shxy/shyx2.prg, CPU/shxy/shyx3.prg, CPU/shxy/shyx4.prg
- page boundaries: CPU/shxy/shyx1.prg, CPU/shxy/shyx5.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
```

Example: Sync with raster beam (remove cycle variance)

The following snippet can be used as a replacement for the commonly used "half variance" $D012 polling loop, to synchronise the code to a fixed position before setting up a timer that is then synced to the raster beam.

This variant works at any address. It is required that $A2 holds a value <$80 before the routine is started. A good init value would be $01, since this will be restored by the last loop iteration.

The code can be written in two ways, the following two snippets are the same piece of code. The first shows what executes when the code was started:

```
loop = * + 1
          LDY #$B5        ; initialize Y
          LDX #$9C
          LDX #$00        ; initialize X
          BPL loop
```

now the BPL branches to the operand of the LDY, so the loop looks as follows:

```
          !byte $a0       ; LDY
loop:     LDA $A2,X
          SHY $00A2,X
          BPL loop
```

The SHY stores a value to zp-adress $A2. Performing the LDA $A2,X, with X=$00, ends up reading the value from $A2. Since SHY does not influence the flags, the branch solely depends on what was read with LDA $A2,X – which in turn is the value that was written by the SHY in the loop iteration before. Now as long as we have the &H+1 in play when executing the SHY, the value stored to $A2 will be $01 (since H+1=$01 and Y=$B5). When the &H+1 does not occur, the full value $B5 is written to $A2, which is read in the next loop iteration, effectively ending the whole loop ($B5 -> Accu sets the N-flag!).

This syncing approach works just like the variants given above: when a badline pauses the CPU on the 4th SHX cycle, value $B5 is written to $A2, so the cycle when this happens is known, and consequently also the cycle position of the loop's end.

Sometimes, you want to set up a timer that loops once per frame, not per line, and on a stable cycle. The snippet above gives us a constant x position, but on a random y, so won't work for that.

```
        LDY #$01
loop:
        LDX #243  ; (or any other badline)
        CPX $D012
        BNE *-3
        SHY $FFFF, X
        LDA $FF, X
        BEQ loop
```

This always drops us out on line 243 cycle 62, and takes up to seven frames to do so.

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} = A & X & {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:

- The value written is ANDed with `&{H+1}`, except when the RDY line goes low in the 4th cycle.
- When adding `Y` to the target address causes a page boundary crossing, the highbyte of the target address is incremented by one (as expected), and then ANDed with `(A & X)`.

Example:

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

Equivalent Instructions:

```
          ; save flags, A, X
          PHP
          STA $03        ; save A
          PLA
          STA $02        ; save flags
          STX $04        ; save X

          LDA $03        ; A
          AND $04        ; and with X
          TAX            ; remember A & X
          AND #$78       ; High-byte of Address + 1
          STA $7700,Y    ; addr = A & X & H+1
          TXS            ; sp = A & X

          ; restore flags, A, X
          LDX $04        ; X
          LDA $03        ; flags
          PHA
          LDA $02        ; akku
          PLP            ; restore flags
```

Note: The above code does in many ways not accurately resemble how the `TAS` opcode works exactly, memory location `$02-$04` would not be altered and the stack would not be used.

Test code:

- general: `Lorenz-2.15/shsay.prg`
- &{H+1} drop off: `CPU/shs/shsabsy2.prg, CPU/shs/shsabsy3.prg, CPU/shs/shsabsy4.prg`
- page boundaries: `CPU/shs/shsabsy1.prg, CPU/shs/shsabsy5.prg`

Example: SAX abs, y with SP=A & X

When using `$FE00` as address, the value stored would be `AND`ed by `$FF` and the TAS turns into a `SAX`, plus it moves the result of `AND`ing A and X into the stackpointer. This can be extremely powerful if you can afford trashing the stackpointer (ie saving/restoring it) in a piece of code where you want to compute A & X and reuse the resulting value a few times, preferably in the X register.

```
          TSX             ; save stackpointer
          STX  temp

          LDA  GLOBALMASK
          LDX  LOCALMASK
          TAS  $FE00,Y    ; SAX $FE00,Y stores A & X & ($FE + 1)
                          ; also sets SP = A & X
          ...
          TSX             ; get A & X
          LDY  data0,x
          STY  bitmap+0
          ...
          TSX             ; get A & X
          LDY  data1,x
          STY  bitmap+1
          ...

          LDX  temp       ; restore stackpointer
          TXS
```

'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 (and thus time!) dependent. The behaviour also depends on the RDY line, which needs extra caution.

These two opcodes are the only ones that can be considered truly unstable.

- The ‘magic constant’ must be considered to be totally random. Although often reported as being eg 0xee, 0xef or 0xff, you should not rely on any of this being the case. **You must use these opcodes in a way so the ‘magic constant’ is taken out of the equation. Do not rely on reading the ‘magic constant’ either, as it may change with time and temperature.**
- The ‘magic constant’ somehow interacts with the RDY line. In particular bits 0 and 4 seem to be “weaker” than the other bits, and may drop to 0 when a DMA starts. **It may be notable that this behaviour can not be reproduced in visual6502, which hints on it being some analogue side effect that the simulation does not cover.** This also contributes to the instabilities.

ANE (XAA, AXM)

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 $ee, $00, $ff …). Some dependency on the RDY line. Bit 0 and Bit 4 are "weaker" than the other bits, and may drop to 0 in the first cycle of DMA when RDY goes low.

Do not use ANE with any immediate value other than 0, or when the accumulator value is $ff (both take the magic constant out of the equation)! (Or, more accurately, 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, CPU/ane/ane-none.prg, CPU/ane/ane-border.prg`

Simulation Links:

- read magic constant: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=13&d=a200a9ff8bffea

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

Real world code

Interestingly, the unstable `ANE #imm` opcode can actually be found in code “in the wild”:

- The Ocean/Imagine tape loader (Rambo II, Comic Bakery, Yie Ar Kung Fu) uses it, albeit in a way that is considered stable.

- The Mastertronic variant of the “burner” tape loader (used eg for the games “Spectipede”, “BMX Racer”) uses it in a way that is considered unstable. For the game to load, the high nibble of the “magic constant” must be `$4`, `$5`, `$E` or `$F` and bit 0 must be 1, bits 3, 2, 1 are "don't care". That means the commonly assumed value `$EE` for the “magic constant” will **not** work, but `$EF` does.

- The game “Turrican 3” by Smash Designs uses `ANE #imm` in an unstable way in the scrolling routine of levels 1 and 2. It looks like the code expects the “magic constant” to be `$EF`, but luckily it will still “work” when it is `$EE` (but will break completely when other bits are different).

Emulation

For Emulation the best compromise between "proper emulation" and "making things work" seems to be to use a "magic contant" of $EF in regular cycles, and $EE in the RDY cycle.

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 $ee, $00, $ff, …). Some dependency on the RDY line. Bit 0 and Bit 4 are "weaker" than the other bits, and may drop to 0 in the first cycle of DMA when RDY goes low.

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

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, CPU/lax/lax-border.prg, CPU/lax/lax-none.prg`

Simulation Links:

- read magic constant: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=13&d=a200a900abffea

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.'

A surprising discovery

Despite the instability mentioned before, it was discovered that a very popular C-64 game – Wizball - actually uses the `LAX #imm` opcode in a way that is considered unstable (actually in the most unstable way possible). The location in Wizball where `LAX #imm` is executed is at `$b58b` (this is used after pressing fire in the "get ready" screen, and during actual gameplay):

```
b589  A9 00     LDA #$00
b58b  AB FF     LAX #$FF    ; A = X = (($00 | CONST) & $ff) = $EE
b58d  DF 97 FF  DCP $FF97,X ; decrement mem (=$85), compare with
                            ; akku (=$EE)
b590  60        RTS
```

Some different Wizball binaries have been tested (in Emulation) with all possible "magic constants", and the following values result in misbehaviour (crash) of the game: `$63`, `$64`, `$67`, `$68`, `$69`, `$6A`, `$D1`, `$D2`, `$EF` – So we can assume these (`$EF` in particular) did not occur on real C64s - at least no horror stories from the 80s about Wizball not working (randomly) are known to exist.

Emulation

Due to the above, emulators are advised to use `$EE` (**not** $EF) for the "magic constant" in both normal and RDY cycles. This **will** break at least one other program that is known to use `LAX #imm` – the "Blackmail FLI" mentioned in the appendix – however, that program was known to be flaky even back in the days, and the use of `LAX #imm` might very well be the reason for that.

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
3b lo hi   RLA abs, y
1b lo hi   SLO abs, y
5b lo hi   SRE abs, y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4	< AAH, AAL + Y >	Byte at target address before high byte was corrected	R
5	AA + Y	Old Data	R
6	AA + Y	Old Data	W
7	AA + Y	New Data	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
```

```
df lo hi   DCP abs, x
ff lo hi   ISC abs, x
7f lo hi   RRA abs, x
3f lo hi   RLA abs, x
1f lo hi   SLO abs, x
5f lo hi   SRE abs, x
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4	< AAH, AAL + X >	Byte at target address before high byte was corrected	R
5	AA + X	Old Data	R
6	AA + X	Old Data	W
7	AA + X	New Data	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	Opcode fetch	R
2	PC + 1	Direct Offset	R
3	DO	Byte at direct offset	R
4	DO + X	Absolute Address Low	R
5	DO + X + 1	Absolute Address High	R
6	AA	Old Data	R
7	AA	Old Data	W
8	AA	New Data	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	Opcode fetch	R
2	PC + 1	Direct Offset	R
3	PC + 1	Byte at direct offset	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	Opcode fetch	R
2	PC + 1	Direct Offset	R
3	DO	Absolute Address Low	R
4	DO + 1	Absolute Address High	R
5	< AAH, AAL + Y >	Byte at target address before high byte was corrected	R
6	AA + Y	Old Data	R
7	AA + Y	Old Data	W
8	AA + Y	New Data	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
```

```
b3 zp   LAX (zp), y
93 zp   SHA (zp), y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode 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 >	Byte at target address before high byte was corrected	R
5	AA	Data	R/W

(*) Add 1 cycle for indexing across page boundaries, or when writing to memory.

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.

Test code: `CPU/Acid800/cpu_decimal.prg CPU/bclark/decimalmode.prg CPU/asap/cpu_decimal.prg CPU/64doc/dsbc-cmp-flags.prg CPU/64doc/dsbc.prg CPU/64doc/dadc.prg`

Decimal mode in a nutshell

The decimal mode is meant to aid in making calculations with BCD encoded values (“packaged” BCD, one digit per nibble). A BCD encoded value is a hex number with both its upper and lower nibble 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.

1. The ALU works differently than in binary mode:

   The low and high nibble 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.

   When decimal-correcting a nibble for addition, following rules apply:

   ```
   if ((nibble > 0x9) | (C’ == 1)) { nibble += 6 }
   if ((nibble > 0xF) { C’’ = 1 } else { C’’ = C’ }
   ```

   When decimal-correcting a nibble for subtraction, following rules apply:

   ```
   if (C’ == 0) { nibble -= 6 }
   if (nibble < 6) { C’’ = 1 } else { C’’ = C’ }
   ```

   Thus, $F + $F in decimal mode is $14, not $24. Also, decimal correction can result in nibbles ranging from $A-$F. For example, $C + $D results in $19 before correction, $1F after.

The Processor Flags work differently than in binary mode:

- C will work as a carry for multi-byte operations as expected (for valid BCD values, for other values see the rules above)
- N and V are set after the high-order nibble is added or subtracted but before it is decimal-corrected, according to binary rules (see the respective instruction below).
  - N will be equal to bit 7 of some intermediate result
  - V will use the same logic as in binary mode, but some intermediate results will be used
- Z is always set according to binary mode. So it 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 nibble 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 nibble. */
tmp = (A & 0x0f) + (imm & 0x0f) + C;

/* BCD fixup for lower nibble. */
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 nibble. */
N = (tmp & 0x80) >> 7;
V = ((A ^ tmp) & 0x80) && !((A ^ imm) & 0x80);

/* BCD fixup for higher nibble. */
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 nibble overflows. In decimal addition, you had to verify if the nibble was greater than 9. The processor has an internal "half carry" flag for the lower nibble, used to trigger the BCD fixup. When calculating with legal BCD values, the lower nibble 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 nibble 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 */

/* set flags like in a binary subtraction */
tmp = A - imm - (C ^ 1);
C = (tmp < 0x100) ? 1 : 0;
N = (tmp & 0x80) >> 7;
Z = ((tmp & 0xff) == 0) ? 1 : 0;
V = (((A ^ tmp) & 0x80) && ((A ^ imm) & 0x80));

/* Calculate the lower nibble. */
tmp2 = (A & 0x0f) - (imm & 0x0f) - (C ^ 1);
/* BCD correction */
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 nibble of the AND result, incremented by its lowmost bit, is greater than 5, the low nibble in the ROR result will be incremented by 6. The low nibble may overflow as a consequence of this BCD fixup, but the high nibble won't be adjusted. The high nibble will be BCD fixed in a similar way. If the high nibble of the AND result, incremented by its lowmost bit, is greater than 5, the high nibble 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`

Unintended memory accesses

Due to how the 6502 works internally, every execution cycle is always a memory access – either read or write. In some cases those accesses are “dummies” and are not related to what the instruction is actually supposed to do. Usually these accesses can be ignored, however they can also be (ab)used and sometimes they may cause unwanted side effects, eg when dealing with I/O registers.

Dummy fetches

Single byte instructions

Single byte instructions will always fetch the PC+1 after the opcode fetch (like any other instruction).

Akkumulator

```
ASL  LSR  ROL  ROR
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*)	PC + 1	Byte after opcode	R

(*) fetch after opcode

Implied

```
CLC  CLD  CLI  CLV  DEX  DEY  INX  INY  NOP  SEC  SED  SEI  TAX  TAY  TSX  TXA  TYA  TXS
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*)	PC + 1	Byte after opcode	R

(*) fetch after opcode

Stack (push)

```
PHA  PHP
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*)	PC + 1	Byte after opcode	R
3	S	Pushed value	W

(*) fetch after opcode

Stack (software interrupts)

BRK

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*)	PC + 1	Byte after opcode (“Signature”)	R
3	S – 0	Program counter High	W
4	S - 1	Program counter low	W
5	S - 2	Status register	W
6	VA	IRQ vector address low (= $fffe)	R
7	VA + 1	IRQ vector address high (= $ffff)	R

(*) fetch after opcode

Stack (RTI)

RTI

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*1)	PC + 1	Byte after opcode	R
3 (*2)	S + 0	Byte from Stack	R
4	S + 1	Status register	R
5	S + 2	Program counter low	R
6	S + 3	Program counter High	R

(*1) Dummy fetch from PC + 1
(*2) Dummy fetch from S + 0

Example: acknowledge CIA interrupts

```
JMP $DD0C
;DD0C    RTI
```

This will execute the RTI instruction at $DD0C, but since it will also fetch the next “opcode” it will also perform a read on $DD0D, which will acknowledge the NMI. This is one cycle faster than doing

```
LDA $DD0D
RTI
```

Hardware interrupts

A special case are the hardware interrupts (IRQ, NMI, RESET).

Cycle	Address-Bus	Data-Bus	Read/Write
1 (*1)	PC	Byte at PC	R
2 (*2)	PC + 1	Byte at PC + 1	R
3 (*3)	S - 0	Program counter High	W
4 (*3)	S - 1	Program counter low	W
5 (*3)	S - 2	Status register	W
6	VA	vector address low	R
7	VA + 1	vector address high	R

(*1) Dummy fetch from PC

(*2) Dummy fetch from PC + 1

(*3) R/W remains high during reset, ie reset does not write to the stack

Simulation Link: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=38&a=fffe&d=2000&a=20&d=40&a=0&d=58eaea&logmore=Execute,irq&irq0=5&irq1=6

Indexed instructions

Indexed instructions read from the target address before the highbyte was incremented, if the indexing causes a page boundary crossing.

Absolute indexed

```
ADC abs, x  AND abs, x  CMP abs, x  EOR abs, x  LDA abs, x  LDY abs, x  NOP abs, x  ORA abs, x
SBC abs, x  SHY abs, x  STA abs, x
```

```
ADC abs, y  AND abs, y  CMP abs, y  EOR abs, y  LAS abs, y  LAX abs, y  LDA abs, y  LDX abs, y
ORA abs, y  SBC abs, y  SHA abs, y  TAS abs, y  SHX abs, y  STA abs, y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	DO + 1	Absolute Address High	R
3a (*)	< AAH, AAL + IL >	Byte at target address before high byte was corrected	R
4	AA	Data	R/W

(*) Add one cycle for indexing across page boundaries or write. A dummy read happens to the target address before the high byte was corrected.

Example: acknowledge both CIA interrupts

This will do a dummy read at `$DC0D` and a normal read at `$DD0D`, this way you can acknowledge both CIA interrupts in one instruction:

```
LDX #$F0
LDA $DC1D, X
```

… and is one cycle faster, and one byte shorter, than doing

```
LDA $DC0D
LDA $DD0D
```

Example: 5 cycle wide rastersplits

If you want to have raster splits that are exactly 5 cycles wide, you can use:

```
LDX #$FF
LDY #$05
LDA #$00
STY $D021
STA $CF22, X
```

Example: Sprites far right in the border

To feed data to the sprite pattern pipe for a sprite that is displayed “far right” so it did not yet have its DMA cycles before you can use a

```
STA VIC_REG, X
```

at the correct position in the rasterline, s.t. the 4th cycle occurs at the first sprite DMA-cycle and the 5th (the W-cycle) at the 2nd sprite DMA-cycle. This way the sprite pattern byte is filled with:

1. byte read in 4th cycle from the (uncorrected!) VIC-address
2. ghostbyte
3. byte stored in 5th cycle

Testcode:

- `VICII/sb_sprite_fetch/sbsprf24.prg`

Zeropage Indirect Y Indexed

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

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Direct Offset	R
3	DO	Absolute Address Low	R
4	DO + 1	Absolute Address High	R
4a (*)	< AAH, AAL + Y >	Byte at target address before high byte was corrected	R
5	AA	Data	R/W

(*) Add 1 cycle for indexing across page boundaries, or write. Dummy read from target address before the high byte is incremented.

ZP indexed instructions

ZP indexed instructions read from the target address before the index was added

Zeropage indexed

```
ADC zp, x  AND zp, x  CMP zp, x  EOR zp, x  LDA zp, x  LDY zp, x  ORA zp, x  SBC zp, x  STA zp, x
STY zp, x
LAX zp, y  LDX zp, y  SAX zp, y  STX zp, y
```

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Direct Offset	R
3 (*)	DO	Byte at direct offset	R
4	AA	Data	R/W

(*) Dummy fetch from direct offset, before the index was added

Zeropage X Indexed Indirect

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

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Direct Offset	R
3 (*)	DO	Byte at direct offset	R
4	DO + X	Absolute Address Low	R
5	DO + X + 1	Absolute Address High	R
6	AA	Data	R/W

(*) Dummy fetch from direct offset

Simulation Link: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=18&d=a201a111eaeaeaeaeaeaeaeaeaeaea4212230f00

Stack

Some instructions that use the stack do dummy fetches from the stack.

Absolute (JSR)

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	New PC low	R
3 (*)	S	Byte from stack	R
4	S	PC high	W
5	S - 1	PC low	W
6	PC + 2	New PC high	R

(*) Dummy fetch from stack

Stack (RTS)

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*1)	PC + 1	Byte after opcode	R
3 (*2)	S	Byte from stack	R
4	S + 1	New PC - 1 low	R
5	S + 2	New PC - 1 high	R
6 (*3)	New PC - 1	Byte at target address	R

(*1) dummy fetch from PC + 1
(*2) dummy fetch from stack
(*3) dummy fetch from target address (New PC – 1)

Stack (Pull)

PLA PLP

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2 (*1)	PC + 1	Byte after opcode	R
3 (*2)	S	Byte from stack	R
4	S + 1	Pulled value	R

(*1) dummy fetch from PC + 1
(*2) dummy fetch from Stack

Branches

Branches read from various intermediate addresses

BCC BCS BNE BEQ BMI BPL BVC BVS

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Offset	R
+1 (*1)	PC + 2	Byte at PC + 2	R
+1 (*2)	PC + 2 + offset (Old hi)	Byte at PC + 2 + offset (Old hi)	R

(*1) Add one cycle if branch is taken, dummy read from PC + 2 (This would usually be the next instruction)
(*2) Add one cycle if branch is taken across page boundaries (that means PC + 2 and PC + 2 + offset are on different pages), dummy read from PC + 2 + offset (before the high byte was corrected to point to the new/correct page)

Dummy writes

Read-Modify-Write

Read-Modify-Write instructions will write back the unmodified value before writing the modified value.

Absolute (R-M-W)

`ASL abs  DCP abs  DEC abs  INC abs  ISC abs  LSR abs  RLA abs  ROL abs  ROR abs  RRA abs  SLO abs  SRE abs`

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4	AA	Old Data	R
5 (*1)	AA	Old Data	W
6	AA	New Data	W

(*1) Unmodified original data is written back to memory

Example: acknowledge VIC-II interrupt

This is probably the most popular usage of the dummy writes. Usually you would have to do

```
LDA $D019
STA $D019
```

to acknowledge the VIC-II interrupt. However, you can use any RMW instruction instead, eg

```
INC $D019
```

Example: acknowledge and disable timer interrupt

```
DEC $DC0D
```

after a Timer A IRQ occurred would acknowledge that IRQ and stop further ones.

Example: write two values to I/O one cycle apart

The music routine from Fred Gray performs a read and write on IO with:

```
LDA #$40
STA $D404
INC $D404
```

Which will toggle the gate bit of the control register of the SID chip.

The same idea can be used to create “grey dots” spaced 8 pixels apart:

```
INC $D020
```

Example: ghostbyte under ROM

When reading a byte from ROM, the CPU reads from the ROM, but when writing to a byte in the ROM, the write falls through to the RAM beneath it. So with an RMW instruction you can actually write 2 values to a byte in RAM, 1 cycle apart, where none of the two written values are the value that was already present. Usually not a useful thing to do, but together with the VIC we could exploit this:

Put the VIC in bank 2 or 3 and enable the KERNAL/BASIC ROM. Then an `INC` (for example) can write to the ghostbyte twice, 1 cycle apart - and the first write doesn't necessarily have to write what was already there!

Unfortunately what you can write at the first dummy cycle is limited to what is in the ROM at the chosen ghostbyte address (4 possibilities). What you can write at the 2nd write cycle also depends on that value as well as which RMW-instruction you use (so we have 6 possibilities per ghostbyte address for the second write-cycle).

Let's look at which possibilities of pixels we have:

ROM	First Cycle	Second Cycle					
		INC	DEC	ASL	ROL	LSR	ROR
$B9FF = $A0	%10100000	%10100001	%10011111	%01000000	%0100000C	%01010000	%C1010000
$BFFF = $E0	%11100000	%11100001	%11011111	%11000000	%1100000C	%01110000	%C1110000
$F9FF = $D2	%11010010	%11010011	%11010001	%10100100	%1010010C	%01101001	%C1101001
$FFFF = $FF	%11111111	%00000000 (*)	%11111110	%11111110	%1111111C	%01111111	%C1111111

(*) this might be useful in practise to create a single cycle wide `$00` → `$FF` → `$00` pattern, just do an `INC $FFFF` somewhere the ghostbyte is visible. (And init the ghostbyte to `$00` in advance).

Instead of the ghostbyte, the same could be done for charset/bitmaps (but not sprites or the screen), for example (using precise timing) a charset-byte could could be set to `$FF` at the exact time it is read by the VIC, using the dummy-write of an `INC`, and then to `$00` immediately after, at the second write-cycle, so that it is `$00` next time it is rendered by the VIC (instead of `LDA #$FF`, `STA $xxxx`, `LDA #$00`, `STA $xxxx`).

When repeating the 7th pixel-line of a text-line using linecrunch, for example, this could make the charset-byte `$FF` on one raster line and `$00` on the next with only one `INC $xxxx` instruction.

That could of course be repeated again and again, every 2nd line, so that the charset-byte alternates between `$00` and `$FF` every rasterline.

Example: start a REU transfer

It is possible to start a REU transfer by writing to address `$FF00`, which is useful when you want to transfer to or from memory in the `$D000-$DFFF` range. But sometimes you don't want to trash the byte at $FF00, so you end up starting the transfer like this:

```
LDA $FF00
STA $FF00
```

However, it turns out you can use any RMW instruction:

```
INC $FF00
```

The dummy write causes the REU to immediately take over the bus, so the second write-request from the CPU doesn't reach the memory chips. The incremented value never gets written into RAM - Three cycles saved.

Test code:

- `REU/rmw-trigger/rmwtrigger-ram.prg`, `REU/rmw-trigger/rmwtrigger-rom.prg`

The 6502 has two inputs, /RDY (Ready) and /AEC (Address Enable Control). RDY tells the CPU to pause execution, but it is only obeyed during read cycles. AEC immediately disconnects the CPU from the buses (address, data, and the read/write signal).

The VIC chip has two outputs, BA (Bus Available) and AEC (Address Enable Control). During normal operation, VIC asserts AEC (which is connected to AEC on the CPU) on every other half-cycle in order to read e.g. font bits. It has to work immediately, i.e. asynchronously, because it needs to be fast enough for half-cycle operations.

When VIC needs to halt the CPU, it first pulls BA low for three cycles, to ensure that the CPU is on a read cycle. Then it asserts AEC in order to access memory on both half-cycles.

The expansion port has an output, BA, and an input, /DMA. BA comes from the VIC. But /DMA is connected to both /RDY and /AEC. That is, it tells the CPU to pause, but it also immediately disconnects the CPU from the buses.

The REU monitors BA so it can pause an ongoing transfer during badlines and sprite fetches. But otherwise, it pulls /DMA and just assumes that the bus is free. The engineers must have assumed (wrongly) that the CPU will always trigger a transfer on the last cycle of an instruction, so that the next cycle is guaranteed to be a read (to fetch the next instruction).

Instead, due to the double-write of our RMW instruction, part of the CPU will attempt to place an address and data value on the buses, and set the read/write line to write. But the CPU is disconnected from the buses because /DMA is held low, and therefore /AEC. The bits never reach the actual bus lines; they dissipate into a small amount of heat.

Zeropage (R-M-W)

`ASL zp  DCP zp  DEC zp  INC zp  ISC zp  LSR zp  RLA zp  ROL zp  ROR zp  RRA zp  SLO zp  SRE zp`

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	AA	Old Data	R
4 (*1)	AA	Old Data	W
5	AA	New Data	W

(*1) Unmodified original data is written back to memory

Indexed Read-Modify-Write

Indexed Read-Modify-Write instructions will do a dummy read and write back the unmodified value before writing the modified value.

Absolute X Indexed (R-M-W)

`ASL abs, x  DEC abs, x  INC abs, x  LSR abs, x  ROL abs, x  ROR abs, x  DCP abs, x  ISC abs, x`
`RRA abs, x  RLA abs, x  SLO abs, x  SRE abs, x`

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4 (*1)	< AAH, AAL + X >	Byte at target address before high byte was corrected	R
5	AA + X	Old Data	R
6 (*2)	AA + X	Old Data	W
7	AA + X	New Data	W

(*1) Dummy fetch from target address before the high byte was incremented
(*2) Unmodified data is written back to the target address

Absolute Y Indexed (R-M-W)

DCP abs, y ISC abs, y RRA abs, y RLA abs, y SLO abs, y SRE abs, y

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Absolute Address Low	R
3	PC + 2	Absolute Address High	R
4 (*1)	< AAH, AAL + Y >	Byte at target address before high byte was corrected	R
5	AA + Y	Old Data	R
6 (*2)	AA + Y	Old Data	W
7	AA + Y	New Data	W

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

(*1) Dummy fetch from target address before the high byte was incremented
(*2) Unmodified data is written back to the target address

Zeropage X indexed (R-M-W)

ASL zp, x DCP zp, x DEC zp, x INC zp, x ISC zp, x LSR zp, x RLA zp, x ROL zp, x ROR zp, x
RRA zp, x SLO zp, x SRE zp, x

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Direct offset	R
3 (*1)	DO	Byte at direct offset before index was added	R
4	DO + X	Old Data	R
5 (*2)	DO + X	Old Data	W
6	DO + X	New Data	W

(*1) Dummy fetch from direct offset before the index was added
(*2) Unmodified data is written back to the target address

Zeropage Indirect Y Indexed (R-M-W)

`DCP (zp), y  ISC (zp), y  RLA (zp), y  RRA (zp), y  SLO (zp), y  SRE (zp), y`

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Opcode fetch	R
2	PC + 1	Direct Offset	R
3	DO	Absolute Address Low	R
4	DO + 1	Absolute Address High	R
5 (*1)	< AAH, AAL + Y >	Byte at target address before high byte was corrected	R
6	AA + Y	Old Data	R
7 (*2)	AA + Y	Old Data	W
8	AA + Y	New Data	W

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

(*1) Dummy read from target address before high byte is incremented

(*2) Unmodified data is written back to the target address

Zeropage X Indexed Indirect (R-M-W)

`DCP (zp, x)  ISC (zp, x)  RLA (zp, x)  RRA (zp, x)  SLO (zp, x)  SRE (zp, x)`

Cycle	Address-Bus	Data-Bus	Read/Write
1	PC	Op Code Fetch	R
2	PC + 1	Direct Offset	R
3 (*1)	DO	Byte at direct offset	R
4	DO + X	Absolute Address Low	R
5	DO + X + 1	Absolute Address High	R
6	AA	Old Data	R
7 (*2)	AA	Old Data	W
8	AA	New Data	W

Simulation Link: http://visual6502.org/JSSim/expert.html?graphics=f&a=0&steps=22&d=a2d0c310eaeaeaeaeaeaeaeaeaeaea1280

(*1) Dummy read from direct offset before index was added

(*2) Unmodified data is written back to the target address

Unintended bugs and quirks

This chapter covers the remaining rest of weird and/or possibly undocumented and unintended things left.

Zeropage addressing modes & page wraps

If you use an indexed-zeropage addressing mode, either direct or indirect, it is not able to leave the zeropage on page-wraps.

Examples:

```
LDX #$01
LDA $FF,X
```

will fetch from adress `$0000` and not `$0100`.

```
LDA ($FF),Y

LDX #$00
LDA ($FF,X)

LDX #$FF
LDA ($00,X)
```

will all fetch the low-byte from `$00FF` and the high-byte from `$0000`.

Indirect addressing mode & page wraps

If you use the indirect addressing mode, `PCH` will not be incremented on page wraps. Example:

```
JMP ($C0FF)
```

will fetch the low-byte from `$C0FF` and the high-byte from `$C000`.

Interrupts do not affect Flags

When an interrupt fires, the CPU will push the status register to the stack, so the interrupt handler can modify these and restore them later to continue the main program. However, the CPU does no further preparations – in particular it will not set up the decimal flag. That means if the main program changes the decimal flag, and the interrupt handler uses instructions that may be affected by decimal mode, the interrupt handler must make sure to clear (or perhaps set) the decimal flag as needed.

The Break Flag is always set

During normal program execution, the B flag is always set – even in an interrupt handler it can not be used directly to determine whether an interrupt was caused by the BRK instruction or not. Ie the obvious PHP/PLA sequence can not be used for this, it will always read the B flag as 1. The reason for this is, that the B flag doesn't actually exist in the CPU – it's just the current state of the internal interrupt line, and as soon as the CPU is done handling an external interrupt, it goes inactive again. This is also the reason for why neither PLP nor RTI would restore the flag to 0.

The only case when this is not the case, ie the flag in the status byte is 0, is when a BRK instruction is executed, and while that happens the current status register is pushed to the stack by the CPU. This pushed value may contain a cleared B flag, and examining this pushed value on the stack is the only way to distinguish a software interrupt (BRK) and a hardware interrupt (IRQ or NMI).

A simple rule for this is: any hardware interrupt (IRQ or NMI) pushes the B flag as 0, while any CPU instruction (BRK or PHP) pushes the B flag as 1.

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	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					NOP, TOP	NOP	NOP	NOP
NOP		$80	$04	$14										NOP, 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)

Read rightmost set bit, and set it to 0

```
; Set the rightmost set bit to zero:
; A = *addr;
; *addr &= *addr - 1
     LAX addr  ; A = X = *addr;
     DEX       ; X--;
     SAX addr  ; *addr = A & X
; Retrieve said bit:
; A = A ^ *addr
     EOR addr
```

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
```

Blackmail FLI

In 1989 ASP of Blackmail released “FLI Graph v2.2”, which for the first time included a FLI display routine exploiting that in the first 3 columns of a FLI picture the VICII would fetch the colour-ram colour from “open” bus, with the result that whatever opcode comes after the write to $D011 in the FLI displayer will define said colour.

The following shows the extracted code snippets used for creating the various colours in the first 3 columns, including a small fix that makes it possible to use all 16 colours.

First let’s outline what this display routine needs to do:

- for each line use 23 cycles (each line is a badline)
- in each line change the vram base address (via $D018)
- in each line force a badline (via $D011) at the same cycle within the line
- in each line alter the colour ram colour in the first 3 column by placing a specific opcode right after the store to $D011
- in each line change the background colour ($D021)

The first half of code for each line is always the same, first the X register is preloaded with the value that will be stored to $D011. The same value will also be used for indirect-x indexed addressing in some variants of the second half of the code. After that the Akku is loaded with the value used for $D018, then $D018 is written, and finally $D011 is stored. These 4 opcodes take 13 cycles total:

```
(3) a6 xx     ldx zp           ; $69/$6b/$6d/$6f/$71/$73/$75/$77
                               ; → loaded value is $b8,$b9..$bf
(2) a9 xx     lda # <screen>   ; original code uses $08..$78 here.
                               ; however if we use $0f-$7f instead,
                               ; the LAX#imm used in one variant
                               ; of the second half will always
                               ; work as intended, and for all 16
                               ; colours
(4) 8d 18 d0  sta $d018
(4) 8e 11 d0  stx $d011
```

After this follows a different part of code, depending on the colour that should be used for the colour ram in the first 3 columns. This was achieved by using one of the opcodes in the $aX row of the opcode matrix, which are all loads. The opcode used selects the colour fetched for the colour ram, the value fetched by this opcode selects the colour stored to $D021. This part of the code alwas takes 10 cycles.

0: black as colour ram colour

```
(2) a0 xx     ldy # <background colour>
(4) 8c 21 d0  sty $d021
(2) ea        nop
(2) ea        nop
```

1: white as colour ram colour

```
(6) a1 xx     lda (zp,x)        ; ((background colour << 1) + $59)
                                ; - ((line & 7) | $b8)
(4) 8d 21 d0  sta $d021
```

2: red as colour ram colour

```
(2) a2 xx     ldx # <background colour>
(4) 8e 21 d0  stx $d021
(2) ea        nop
(2) ea        nop
```

3: cyan as colour ram colour

```
(6) a3 xx     lax (zp,x)        ; ((background colour << 1) + $59)
                                ; - ((line & 7) | $b8)
(4) 8f 21 d0  sax $d021
```

4: violet as colour ram colour

```
(3) a4 xx     ldy zp            ; <(background colour << 1) + $59>
(4) 8c 21 d0  sty $d021
(3) 24 24     bit $24
```

5: green as colour ram colour

```
(3) a5 xx     lda zp            ; <(background colour << 1) + $59>
(4) 8d 21 d0  sta $d021
(3) 24 24     bit $24
```

6: blue as colour ram colour

```
(3) a6 xx     ldx zp            ; <(background colour << 1) + $59>
(4) 8e 21 d0  stx $d021
(3) 24 24     bit $24
```

7: yellow as colour ram colour

```
(3) a7 xx     lax zp            ; <(background colour << 1) + $59>
(4) 8f 21 d0  sax $d021
(3) 24 24     bit $24
```

8: orange as colour ram colour

```
(2) a8        tay
(2) a0 xx     ldy # <background colour>
(4) 8c 21 d0  sty $d021
(2) ea        nop
```

9: brown as colour ram colour

```
(2) a9 xx     lda # <background colour>
(4) 8d 21 d0  sta $d021
(2) ea        nop
(2) ea        nop
```

a: l.red as colour ram colour

```
(2) aa        tax
(2) a2 xx     ldx # <background colour>
(4) 8e 21 d0  stx $d021
(2) ea        nop
```

b: d.grey as colour ram colour

```
(2) ab xx     lax # <background colour> ; the original code uses
                                        ; A=$08,$18...$78 for the
                                        ; values written to $d011,
                                        ; which makes this rely on
                                        ; bit0-2 of the “magic
                                        ; constant” being set.
                                        ; However if we use A=$0f,
                                        ; $1f...$7f instead, that
                                        ; takes the constant out
                                        ; of the equation and this
                                        ; lax#imm will always work
                                        ; for all 16 colours.
(4) 8f 21 d0  sax $d021
(2) ea        nop
(2) ea        nop
```

c: m.grey as colour ram colour

```
(4) ac xx 03  ldy abs           ; $03b0 + <background colour>
(4) 8c 21 d0  sty $d021
(2) ea        nop
```

d: l.green as colour ram colour

```
(4) ad xx 03  lda abs           ; $03b0 + <background colour>
(4) 8d 21 d0  sta $d021
(2) ea        nop
```

e: l.blue as colour ram colour

```
(4) ae xx 03  ldx abs           ; $03b0 + <background colour>
(4) 8e 21 d0  stx $d021
(2) ea        nop
```

f: l.grey as colour ram colour

```
(4) af xx 03  lax abs           ; $03b0 + <background colour>
(4) 8f 21 d0  sax $d021
(2) ea        nop
```

last not least, for the above code snippets to work, the following tables are used:

```
; values read by indirect x indexed loads
; values read by absolute loads ($03b0 + <background colour>)
03b0  b0 b1 b2 b3  b4 b5 b6 b7  b8 b9 ba bb  bc bd be bf

; addresses for indirect x indexed loads ($03b0...$03bf)
; every other value used for zp loads ($b0,$b1...$bf)
; second half also used as $d011 values ($b8, $b9...$bf)
0059  b0 03  b1 03 b2 03  b3 03 b4 03  b5 03 b6 03 b7 03
0069  b8 03  b9 03 ba 03  bb 03 bc 03  bd 03 be 03 bf 03
```

References

Everything in this document has been verified and is backed up by various test programs:

- VICE test-programs: https://sourceforge.net/p/vice-emu/code/HEAD/tree/testprogs/
  - Emulator Test-suite by Wolfgang Lorenz
  - Test programs by Poitr Fusik
- First CSDb "Unintended OpCode coding challenge": http://csdb.dk/event/?id=2417
- Blackmail “FLI Graph v2.2”
- Wizball (`LAX#imm` usage)
- Spectipede tape loader (`ANE#imm` usage)
- Turrican 3 scroll routine (`ANE#imm` usage)

Various older documents were used to create the first merged version of this document:

- 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

And some more bits of info were nicked from these places:

- http://www.atariage.com/forums/topic/168616-lxa-stable/#entry2092077
- http://www.pagetable.com/?p=39
- cbmhackers mailing list
- https://wiki.nesdev.com/w/index.php/CPU_unofficial_opcodes

Last not least, some example code snippets were borrowed from elsewhere too:

- 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

And of course the excellent visual6502 project provides lots of detail info in their wiki (currently offline)

- https://web.archive.org/web/20210405071236/http://visual6502.org/wiki/index.php?title=Special%3AAllPages

Please don’t mind the few unattributed anecdotal quotes in the text (*printed in italics*) – The text was not meant for publication initially and I forgot who posted what. The respective authors are probably present in the following list afterall:

Greets and Thanks

- 0xF/Taquart
- AndroSID
- Atomcode
- Bitbreaker/Oxyron
- ChristopherJam
- Clint Gamlin
- Color Bar
- Copyfault/The Solution
- Count Zero/Cyberpunx
- Cyberbrain/Noname
- Fungus/Nostalgia
- Graham/Oxyron
- Geir Straume
- Hoogo/Padua
- JAC!
- Kabuto/Latency
- Kakka
- Krill/Plush
- LFT/Kryo
- Marco Baye
- Mist/R.O.L.E.
- Ninja/The Dreams
- NoobTracker/The Solution
- Peiselulli/TRSI^Oxyron
- pwsoft
- Quiss/Reflex
- Segher Boessenkool
- S.E.S./Crest
- Se7en/Digital Excess
- Soci/Singular
- Strobe/VICE Team
- SvOlli/XayaX
- TLR/VICE Team
- Unseen/VICE Team
- Wil
- Wilfred Bos
- Wolfgang Lorenz
- WoMo

… 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 !

- While the dependency on RDY of `ANE#imm` and `LAX#imm` can be verified on the C-64 by test programs, it can not be explained 100% properly (yet). More investigation by someone who is able to read die shots may help with that.
  - More examples of where those two have been used in real world code, which will help to determine the “best” value (eg for using it in emulation).
  - `LAX#imm` appears to be more stable than commonly assumed – it would be interesting to find more programs that use it in a “non stable” way so we can narrow down the value of the magic constant more.
- 'unstable address hi byte' opcodes' page boundary crossing and RDY behaviour needs to be verified on more CPUs.
- Some opcodes, such as ARR, should also be tested on a disk drive while data is being read.
- More example code snippets would be great (sharing is caring!)
  - Examples of interesting (ab)use of the decimal mode
  - Examples of interesting (ab)use of the dummy memory accesses
- 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, 2024 (V0.99) – removed incorrect note about N and Z from ANE#imm and LAX#imm (thanks to Geir Straume for pointing out the error), fixed opcode $73 in the opcode matrix, it should be RRA, not SRE (thanks to Kakka for reporting the error). Updated ISC loopcounter example (thanks to Quiss). Added "set rightmost set bit to 0" example (thanks to Quiss).
- December 24th, 2023 (V0.98) – added another raster sync snippet (posted by Quiss), added note on D flag not being cleared by interrupts (inspired by Fungus), added note of B flag always being set (inspired by Fungus), fixed details in the description of the dummy fetches done by branches (thanks to Mac Bacon for reporting the errors)
- December 24th, 2022 (V0.97) – added more DOP/TOP/LAX/DCP examples (provided by Bitbreaker), fixed errors in Zeropage X Indexed Indirect and RTS access cycle tables (thanks to OldWoman37 for reporting), fixed LSB misnamed as bit 1 in RRA, added examples for simulating addressing modes to ISC/DCP/RRA/RLA/SRE/SLO (inspired by Bitbreaker), added SBX example (inspired by Bitbreaker)
- December 24th, 2021 (V0.96) – added note on the 6507 CPU, added parity example for SRE, added indirect indexed decrement example for DCP, fixed the note on LAX #imm in Wizball (thanks to Atomcode for pointing out the obvious mistake), fixed execution cycle table for hardware interrupts (thanks to NoobTracker for pointing out the mistake), made SLO example more clear (thanks to Strobe for pointing it out), fixed execution cycle table for RTI (thanks to NoobTracker for pointing it out)
- December 24th, 2020 (V0.95) – removed invalid RLA example, added chapter about unintended memory accesses, added opcode matrix with comments, added note on LAX #imm in Wizball, added notes on ANE#imm in real world code, added note on the supposedly unstable NOPs, corrected some spelling errors, added raster sync SHX and SHY examples, added additional mnemonics for NOP that are apparently used in NES land, fixed link(s) to sourceforge, fixed some simulation links, fixed a few more details (thanks to Copyfault and Mac Bacon for proofreading).
- December 24th, 2019 (V0.94) – added more detailed description of flags behaviour for some opcodes, updated some ANE/LAX details, fixed description of carry flag for ARR, more precise description of the SHA/SHX/SHY/TAS unstable behaviour, added references to new tests to SHA/SHX/SHY/TAS, updated TAS example code, added more details to decimal mode BCD fixup and flags behaviour, added cross references to the decimal mode chapter to opcodes that depend on the decimal flag, added chapter about bugs and quirks, added description of "Blackmail FLI" to combined examples, added more example code snippets so each opcode has at least one, added some missing references to test programs, Added alternative Mnemonics found in AEGs patched Turbo Assembler, sorted greetings alphabetically
- 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