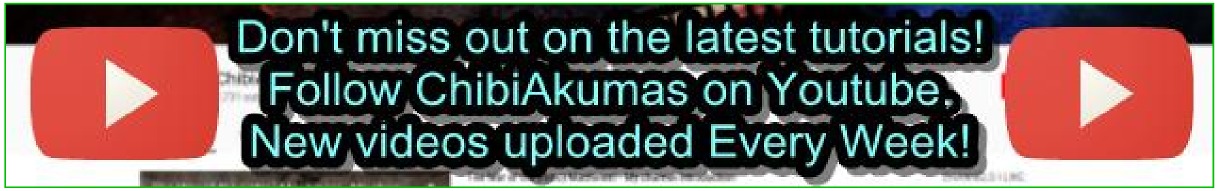

Don't miss out on the latest tutorials!
Follow ChibiAkumas on Youtube.
New videos uploaded Every Week!

Learn Multi platform 6502 Assembly Programming... For Monsters!

Don't like to read? you can learn while you watch and listen instead!

Every Lesson in this series has a matching YOUTUBE video... with commentary and practical examples

Visit the authors Youtube channel, or Click the icons to the right when you see them to watch the Lessons video!

Welcome To the Dark Side!... I grew up with the Amstrad CPC, and I started learning Assembly with the Z80, however as my experience with Z80 assembly grew, I wanted to start learning about other architctures, and see how they compared!

The 6502, and it's varients powered many of the biggest systems from the 80's and 90'... From the ubiquitous C64... to the Nintendo Entertainment System, as well as the BBC Micro, PC-Engine and Atari Lynx... even the Super Nintendo used a 16 bit varient of the 6502 known as the 65816

The 6502's origins are somewhat odd, a cost reduced version of the 8-bit '6800' (which was the predecessor to the venerable16-bit 68000)... the 6502 sacrificed some functions for a cheaper unit price, which allowed it such wide support... the 6510 which powered the C64 had a few added features...

the 6502

View Options
Default Dark
Simple (Hide this menu)
Print Mode (white background)

Top Menu
Main Menu
Youtube channel
Forum
AkuSprite Editor
Dec/Bin/Hex/Oct/Ascii Table

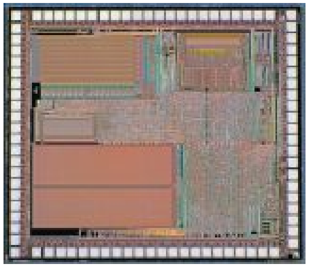
Z80 Content
Z80 Tutorial List
Learn Z80 Assembly
Hello World
Advanced Series
Multiplatform Series
Platform Specific Series
ChibiAkumas Series
Grime Z80
Z80 Downloads
Z80 Cheatsheet
Sources.7z
DevTools kit
Z80 Platforms
Amstrad CPC
Elan Enterprise

A later version, the 65C02 added more commands (Used in systems like the Apple IIc and the Atari Lynx) ... and HudsonSoft made a custom version of the 65C02 with even more features, called the HuC6280 and exclusively used in the PC Engine

All these CPU variants are 8 bit, and the basic 6502 command set works in the same way on all these sysems, and it's that instruction set we'll be learning in these tutorials...

These tutorials will be written from the perspective of a Z80 programmer learning 6502, but they will not assume any prior knowledge of Z80, so if you're starting out in assembly, these tutorials will also be fine for you!

In these tutorials we'll start from the absolute basics... and teach you to become a multiplatform 6502 monster!... Let's begin!


The 65C02 die

If you want to learn 6502 get the Cheatsheet! it has all the 6502 commands, it also covers the extra commands used by the 65c02 and PC-Engine HuC6280

We'll be using the excellent VASM for our assembly in these tutorials... VASM is an assembler which supports Z80, 6502, 68000, ARM and many more, and also supports multiple syntax schemes...

*You can get the source and documentation for VASM from the official website **HERE***

Table of Contents

Beginners Series - lets learn the basic 6502 commands by example!

Gameboy & Gameboy Color ▶
Master System & GameGear ▶
MSX & MSX2 ▶
Sam Coupe ▶
TI-83 ▶
ZX Spectrum ▶
Spectrum NEXT
Camputers Lynx ▶

6502 Content
6502 Tutorial List
Learn 6502 Assembly ▶
Advanced Series
Platform Specific Series
Hello World Series
Grime 6502 ▶
6502 Downloads
6502 Cheatsheet
Sources.7z
DevTools kit
6502 Platforms
Apple IIe ▶
Atari 800 and 5200 ▶
Atari Lynx ▶
BBC Micro ▶
Commodore 64 ▶
Commander x16 ▶
Super Nintendo (SNES) ▶
Nintendo NES / Famicom ▶
PC Engine (Turbografx-16) ▶
Vic 20 ▶

68000 Content
68000 Tutorial List
Learn 68000 Assembly ▶
Hello World Series
Platform Specific Series
Grime 68000 ▶
68000 Downloads

Advanced Series - More advanced topics

	Lesson A1 - Extra commands in the 65c02 (Snes,Lynx & Apple II) and 6280 (PC Engine) processor

Hello World Series - Get Hello World on your machine with a single ASM file!

	Lesson H1 - Hello World on the BBC Micro!
	Lesson H2 - Hello World on the C64
	Lesson H3 - Hello World on the VIC-20
	Lesson H4 - Hello World on the Atari 800 / 5200
	Lesson H5 - Hello World on the Apple II
	Lesson H6 - Hello World on the Atari Lynx
	Lesson H7 - Hello World on the Nes / Famicom
	Lesson H8 - Hello World on the SNES / Super Famicom
	Lesson H9 - Hello World on the PC Engine/TurboGrafx-16 Card

Simple Samples

	Lesson S1 - Bitmap Drawing on the BBC
	Lesson S2 - Bitmap Drawing on the C64
	Lesson S3 - Bitmap Drawing on the VIC-2
	Lesson S4 - Bitmap Drawing on the Atari 800 / 52000
	Lesson S5 - Bitmap Drawing on the Apple II
	Lesson S6 - Bitmap Drawing on the Atari Lynx
	Lesson S7 - Bitmap Drawing on the Nes / Famicom
	Lesson S8 - Bitmap Drawing on the SNES / Super Famicom
	Lesson S9 - Bitmap Drawing on the on the PC Engine/TurboGrafx-16 Card
	Lesson S10 - Joystick Reading on the BBC
	Lesson S11 - Joystick reading on the C64
	Lesson S12 - Joystick Reading on the VIC-20
	Lesson S13 - Joystick Reading on the Atari 800 / 5200
	Lesson S14 - Joystick Reading on the Apple II
	Lesson S15 - Joypad Reading on the Atari Lynx
	Lesson S16 - Joypad Reading on the Nes / Famicom
	Lesson S17 - Joypad Reading on the SNES / Super Famicom

68000 Cheatsheet
Sources.7z
DevTools kit
68000 Platforms
Amiga 500
Atari ST
Neo Geo
Sega Genesis / Mega Drive
Sinclair QL
X68000 (Sharp x68k)

8086 Content
Learn 8086 Assembly
Platform Specific Series
Hello World Series
8086 Downloads
8086 Cheatsheet
Sources.7z
DevTools kit
8086 Platforms
Wonderswan
MsDos

ARM Content
Learn ARM Assembly
Platform Specific Series
ARM Downloads
ARM Cheatsheet
Sources.7z
DevTools kit
ARM Platforms
Gameboy Advance
Nintendo DS
Risc Os

Risc-V Content
Learn Risc-V Assembly
Risc-V Downloads
Risc-V Cheatsheet

Platforms Covered in these tutorials:

Apple IIe
Atari 800 and 5200
Atari Lynx
BBC B
Commodore 64
Super Nintendo (SNES)
Nintendo Entertainment System / Famicom
PC Engine
Vic 20

Recommended PDF resources:

6502 CPU Manual
6502 Getting started
6502 Tricks

What is the 6502 and what are 8 'bits'

You can skip this if you know about binary and Hex (This is a copy of the same section in the Z80 tutorial)
The 6502 is an 8-Bit processor with a 16 bit Address bus!
What's 8 bit... well, one 'Bit' can be 1 or 0
four bits make a Nibble (0-15)
two nibbles (8 bits) make a byte (0-255)
two bytes (16 bits) make a word (0-65535)

And what is 65535? well that's 64 kilobytes ... in computers 'Kilo' is 1024, because binary works in powers of 2, and 2^10 is 1024
64 kilobytes is the amount of memory a basic 8-bit system can access

6502 is 8 bit so it's best at numbers less than 256... it can do numbers up to 65535 too more slowly... and really big numbers will be much harder to do! - we can design our game round small numbers so these limits aren't a problem.

You probably think 64 kilobytes doesn't sound much when a small game now takes 8 gigabytes, but that's 'cos modern games are sloppy, inefficient, fat and lazy - like the basement dwelling losers who wrote them!!!
6502 code is small, fast, and super efficient - with ASM you can do things in 1k that will amaze you!

Numbers in Assembly can be represented in different ways.

Sources.7z
DevTools kit

PDP-11 Content
Learn PDP-11 Assembly
PDP-11 Downloads
PDP-11 Cheatsheet
Sources.7z
DevTools kit

TMS9900 Content
Learn TMS9900 Assembly
TMS9900 Downloads
TMS9900 Cheatsheet
Sources.7z
DevTools kit
TMS9900 Platforms
Ti 99

6809 Content
Learn 6809 Assembly
6809 Downloads
6809/6309 Cheatsheet
Sources.7z
DevTools kit
6809 Platforms
Dragon 32/Tandy Coco
Fujitsu FM7
TRS-80 Coco 3
Vectrex

My Game projects
Chibi Aliens
Chibi Akumas

Work in Progress
Learn 65816 Assembly
Learn eZ80 Assembly

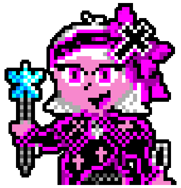
A 'Nibble' (half a byte) can be represented as Binary (0000-1111) , Decimal (0-15) or Hexadecimal (0-F)... unfortunately, you'll need to learn all three for programming!

Also a letter can be a number... Capital 'A' is stored in the computer as number 65!

Think of Hexadecimal as being the number system invented by someone wit h 15 fingers, ABCDEF are just numbers above 9! Decimal is just the same, it only has 1 and 0.

In this guide, Binary will shown with a % symbol... eg %11001100 ... hexadecimal will be shown with $ eg.. $FF.

Assemblers will use a symbol to denote a hexadecimal number, in 6502 programming $ is typically used to denote hex, and # is used to tell the assembler to tell the assembler something is a number (rather than an address), so $# is used to tell the assembler a value is a Hex number
In this tutorial VASM will be used for all assembly, if you use something else, your syntax may be different!



Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	255
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111		11111111
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		FF

Another way to think of binary is think what each digit is 'Worth' ... each digit in a number has it's own value... lets take a look at %11001100 in detail and add up it's total

Bit position	7	6	5	4	3	2	1	0
Digit Value (**D**)	128	64	32	16	8	4	2	1
Our number (**N**)	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**
D x **N**	128	64	0	0	8	4	0	0
128+64+8+4= 204	So %11001100 = **204** !							

If a binary number is small, it may be shown as %11 ... this is the same as %00000011
Also notice in the chart above, each bit has a number, the bit on the far right is no 0, and the far left is 7... don't worry about it now, but you will need it one day!

If you ever get confused, look at Windows Calculator, Switch to 'Programmer Mode' and it has binary and Hexadecimal view, so you can change numbers from one form to another!
If you're an Excel fan, Look up the functions DEC2BIN and DEC2HEX... Excel has all the commands to you need to convert one thing to the other!

But wait! I said a Byte could go from 0-255 before, well what happens if you add 1 to 255? Well it overflows, and goes back to 0!...
The same happens if we add 2 to 254... if we add 2 to 255, we will end up with 1
this is actually usefull, as if we want to subtract a number, we can use this to work out what number to add to get the effect we

Misc bits
Ruby programming

want

Negative number	-1	-2	-3	-5	-10	-20	-50	-254	-255
Equivalent Byte value	255	254	253	251	246	236	206	2	1
Equivalent Hex Byte Value	FF	FE	FD	FB	F6	EC	CE	2	1

All these number types can be confusing, but don't worry! Your Assembler will do the work for you! You can type %11111111 , &FF , 255 or -1 ... but the assembler knows these are all the same thing! Type whatever you prefer in your ode and the assembler will work out what that means and put the right data in the compiled code!

The 6502 Registers

Compared to the Z80, the 6502 has a more limited register set...

The Z80 has Accumulator, 3 pairs of 8 bit regsiters (BC,DE,HL), usable for 16 bit maths and 2 16-bit indirect registers (IX,IY), it also has a 16 bit Stack pointer, and there are 'Shadow Regsiters' for special purposes

The 6502 is very different, it has an 8 bit Accumulator, two 8 bit indirect registers (X,Y) and an 8 bit stack pointer... it also has a 16 bit Program Counter... it has no Shadow Registers

	8 Bit	16 Bit	Use cases
Accumulator	A		
Flags	F		
Indirect X	X		Preindex register , stack pointer manipulation
Indirect Y	Y		Postindex register
Stack Pointer	SP		Stack
Program Counter		PC	

Flags: NV-BDIZC

	Name	Meaning
N	Negative	1=Negative
V	Overflow	1=True
-	unused	
B	BRK command	
D	Decimal mode	1=True
I	IRQ disable	1=Disable
Z	Zero	1=Result Zero
C	Carry	1=Carry

At a glance this may make the 6502 seem significantly inferior to the Z80, however the 6502 has some tricks up it's sleeve!... Where as the fastest command on the Z80 takes 4 ticks, on the 6502 it takes only 1... and the 6502 makes up for it's lack of registers with superior addressing modes!

Special Memory addresses on the 6502

Compared to the Z80, two things are apparent about the 6502... firstly the stack pointer is only 8 bit...

and secondly we have very few registers!

The way the Stack pointer works is simple... the stack is always positioned beween $0100 and $01FF... Where xx is the SP register, the stack pointer will point to $01xx

The 'solution' to the lack of registers is special addressing options... the first 256 bytes between &0000 and &00FF are called the 'Zero Page', and the 6502 has many special functions which allow data in this memory range to be quickly used with the accumulator and other functions as if they were 'registers'!

Note: the PC-Engine has different Zeropage and Stackpointer addresses... and the 65816 can relocate them!... in this case the Zeropage (ZP) is often referred to as the Direct page (DP)

From	To	Meaning
$0000	$00FF	Zero Page (zp)
$0100	$01FF	Stack Pointer
$0200	$FFFF	Normal memory (and mapped registers)

The 6502 Addressing Modes

The 6502 has 11 different addrssing modes... many have no comparable equivalent on the Z80

Mode	Description		Sample Command	Z80 Equivalent	effective result
Implied / Inherant	A command that needs no paprameters	SEC	SEC (set carry)	SCF	
Relative	A command which uses the program counter PC with and offset nn (-128 to +127)	BEQ #$nn	BEQ [label] (branch if equal)	JR Z,[label]	
Accumulator	A command which uses the Accumulator as the parameter	ROL	ROL (ROtate bits Left)	RLCA	
Immediate	A command which takes a byte nn as a parameter	ADC #$nn	ADC #1	ADC 1	&nn
Absolute	Take a parameter from a two byte memory address $nnnn	LDA $nnnn	LDA $2000	LD a,(&2000)	(&nnnn)
Absolute Indexed	Take a parameter from a two byte memory address $nnnn+X (or Y)	LDA $nnnn,X	LDA $2000,X		(&nnnn+X)
Zero Page	Take a parameter from the zero page address $00nn	ADC $nn	ADC $32		(&00nn)
Zero Page Indexed	Takes a parameter from memory address $00nn+X	ADC $nn,X	ADC $32,X		(&00nn+X)
Indirect	Take a parameter from pointer at address $nnnn... if $nnnn contains $1234 the parameter would come from the address at $1234	JMP ($1000)		LD HL,(&1000) JP (HL)	(&nnnn)
indirect ZP	The 65c02 has an extra feature, where it can read from an unindexed Zero page	LDA ($80)			((&00nn))

Pre Indexed (Indirect,X)	Take a paramenter from pointer at address $nnnn+X if $nnnn contains $1234, and X contained 4 the parameter would come from the address at $1238	ADC ($nn,X)	ADC ($32,X)		((&00nn+X))
Postindexed (Indirect),Y	Take pointer from address $nnnn, add Y... get the parameter from the result if $nnnn contains $1234, and Y contained 4, the address would be read from $1234... then 4 would be added... and the parameter would be read from ther resulting address	ADC ($nn),Y	ADC ($32),Y		((&00nn)+Y)

CMP

If we do the comparison

```
    LDA #val1
    CMP #val2
```

We can test the result with the following commands

Basic command	Comparison	6502 command	Z80 equivalent	68000 equivalent
if Val1>=Val2 then goto label	>=	BCS label	JP NC,label	BGE label
if Val1<Val2 then goto label	<	BCC label	JP C,label	BLT label
if Val1=Val2 then goto label	=	BEQ label	JP Z,label	BEQ label
if Val1<>Val2 then goto label	<>	BNE label	JP NZ,label	BNE label

Addresses, Numbers and Hex... 6502 notification

We'll be using VASM for our assembler, but most other 6502 assemblers use the same formats... however coming from Z80, they can be a little confusing, so lets make it clear which is which!

Prefix	Example	Z80 equivalent	Meaning
#	#16384	16384	Decimal Number
#%	#%00001111	%00001111	Binary Number
#$	#$4000	&4000	Hexadecimal number
#'	#'a	'a'	ascii value
	12345	(16384)	decimal memory address
$	$4000	(&4000)	Hexadecimal memory address

If you forget the # in a command like ADC #3... you will end up adding from the zeropage address $0003 - and your program will malfunction

With VASM you do not need to put a # where it is always a number, like on jump commands or data declaractions like "DB $3" or "BRA 3"

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Ti-84 Plus CE (eZ80 cpu)

Low and High Byte

Because the 6502 has no 16 bit registers, it's often nesassary to split an address into its High and Low byte parts, by prefixing a label with < or > it's low or high bytes will be extracted and used in the compiled code, lets take a look!

Symbol	Meaning	Example	Result
<	Low Byte	#<$1234	#$34
>	High Byte	#>$1234	#$12

Testing Bits!

In some cases, there are tricks we can do to 'quickly' test a bit!

	7	6	5	4	3	2	1	0
anytime	ASL A BCC/BCC Dest	ASL A BPL/BMI Dest	AND #32	AND #16	AND #8	AND #4	AND #2	LSR A BCC Dest
After a BIT command	BPL/BMI Dest	BVS/BVC Dest						

Important commands that don't exist!

The 6502 lacks some surprisingly common commands that other processors have, but we can 'fake' them with the commands we do have!

Missing command	Meaning	6502 alternative
ADD #5	ADD a number without carry	CLC (Clear carry for add) ADC #5 (Clear carry)
ASR		BPL scalenegativeP SEC (Top bit 1) scalenegativeP: ROR
SUB #5	Subtract a number without carry	SEC (Clear carry for sub) SBC #5 (Clear carry)
NEG	convert positive value in Accumulator to negative value in Accumulator	EOR #255 (XOR/Flip bits) CLC (Clear carry) ADC #1 (add 1)
SWAP A	Swap two Nibbles in A	ASL (shift left - bottom bit zero) ADC #$80 (pop top bit off) ROL (shift carry in) ASL (shift left - bottom bit zero) ADC #$80 (pop top bit off) ROL (shift carry in)
RLCA	Rotate left with wrap	CLC (Clear the carry) ADC #$80 (pop top bit off) ROL(shift carry in)
RRCA	Rotate right with wrap	PHA (Backup A) ROR (Rotate Ritght - get bit)

		PLA (Restore A) ROR (Rotate Ritght - set bit)
BRA r	Jump to PC relative location +r (Use instead of JMP for relocatable code)	CLV Clear Overflow BVC n Branch if overflow clear
CALL NZ,subroutine	Skip over subroutine command if Zero	BEQ 3 Skip the JSR command JSR subroutine Csubroutine to call if nonzero
RET Z	Skip over return command if Zero	BNE #1 Skip the RET command RTS Return if zero
PHX / PHY	Push X (PHX does exist on 65c02) (do opposite for PLX)	TXA PHA
HALT	infinite loop until next Interrupt	CLV BVC -2
LDA (zp)	Load a from the address in (zp) (not needed on 65c02... use LDA (00zp) (do same for STA etc)	LDX #0 LDA (zp,X) or LDY #0 LDA (zp),Y

If you're used to the Z80, don't go looking for INC A or DEC A on the 6502 ... they don't exist either, so you'll have to CLC, ADD #1 instead!... however they DO exist on the 65C02 and HU6280 as DEA and INA

Shifting without carry

ROL / ROR shift with carry

Use ASL to shift bits left, if you don't want the carry (and bottom bit can be 0)
use LSR to shift bits right without the carry

Skip over parameters

We may call a subroutine, and pass some parameters, there are two ways we can do this

Using Zeropage	Using X (takes 7 more bytes)
JSR TestSub db $11,$22,$33 ;Parameters TestSub: ... PLA CLC ADC #3+1 ;(parameter bytes+1... so 3+1) STA retaddr PLA ADC #0 STA retaddr+1 JMP (retaddr)	JSR TestSub db $11,$22,$33 ;Parameters TestSub: ... TSX LDA $0101,X CLC ADC #3 ;(parameter bytes... so 3) STA $0101,X BCC 3 ;Skip over inc command (3 byte cmd) INC $0102,X RTS

Pretending we have 16 bit!

We can use Zero page pointers to fake the Z80's 16 bit operations!

INC (inc de)	DEC (dec de)	ADD (add bc to hl)	SUB
INC z_E BNE IncDE_Done INC z_D IncDE_Done:	LDA z_E BNE DecDE DEC z_D DecDe: DEC z_E	clc lda z_c adc z_l sta z_l lda z_b adc z_h sta z_h	lda z_l sbc z_c sta z_l lda z_h sbc z_b sta z_h

Fast 16 bit loop

```
fontchar_loop:
   lda (z_hl),y
....
   iny
   bne fontchar_loop
   inc z_hl+1
   dex
   bne fontchar_loop
```

RTS

Unlike the Z80, RTS adds 1 to the value on the stack before setting the PC

Status Register bits

7	6	5	4	3	2	1	0
Negative	Overflow	Unused	Break	Decimal mode	Interrupt state	Zero	Carry
1=Negative 0=Positive	1=Overflow 0=No Overflow		1=BRK occured 0=Normal	1=Dec 0=Bin	1=on 0=disabled	1=Zero 0=Nonzero	1=NoCarry 0=Carry

Get 16 bits from a Lookup Table

lookup 16 bit value A in [table]	
ASL A TAX LDA table,X STA destval INX	ASL A TAX LSA BASE+1,X PHA LSA BASE,X

LDA table,X STA destval+1 16 bit value is now in destval	PHA RTS (because RET adds 1 to address - you must subtract 1 from pointers in table)

Lesson 1 - Getting started with 6502

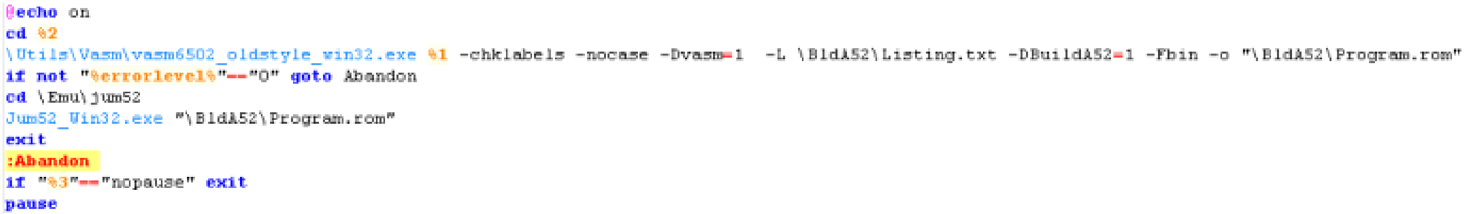
I Learned Assembly on the Z80 systems, and the 6502 seemed strange and scary!... but there's really nothing to worry about, while you have to use it a little bit differently, programming 6502 is no harder than Z80!

Lets start from the basics and learn how to use 6502!

Vasm, Build scripts and Emulators

In these tutorials, we'll be using VASM for our assembly, VASM is free, open source and supports 6502,Z80 and 68000!

We will be testing on various 6502 systems, and you may need to do extra steps (such as adding a header or checksum)... if you download my DevTools, batch files are provided to create the resulting files tested on the emulators used in these tutorials.



```
@echo on
cd %2
\Utils\Vasm\vasm6502_oldstyle_win32.exe %1 -chklabels -nocase -Dvasm=1  -L \BldA52\Listing.txt -DBuildA52=1 -Fbin -o "\BldA52\Program.rom"
if not "%errorlevel%"=="0" goto Abandon
cd \Emu\jum52
Jum52_Win32.exe "\BldA52\Program.rom"
exit
:Abandon
if "%3"=="nopause" exit
pause
```

My sources will use a symbolic definition to define the platform we're buiilding for, if you use my batch files this will occur automatically, but if you're using your own scripts, you need to define this with an EQU statement.

Here's the platform, symbol I use, and emulators we'll be looking at!

Platform	Symbol Definition Required	Emulator used
Apple IIe	BuildAP2 equ 1	AppleWin
Atari 5200	BuildA52 equ 1	Jum52
Atari 800	BuildA80 equ1	Atari800win
BBC Micro B	BuildBBC equ1	BeebEm
C64	BuildC64 equ1	Vice
Atari Lynx	BuildLNX equ 1	Handy
Nintendo NES/Famicom	BuildNES equ 1	Nestopia
PC Engine	BuildPCE equ 1	Ootake
Super Nintendo (SNES)	BuildSNS equ 1	Snes9x
Vic 20	BuildVIC equ 1	Vice

For these tutorials, I have provided a basic set of include files that will allow us to look at the technicalities of each platform and just worry about the workings of 6502 for now...

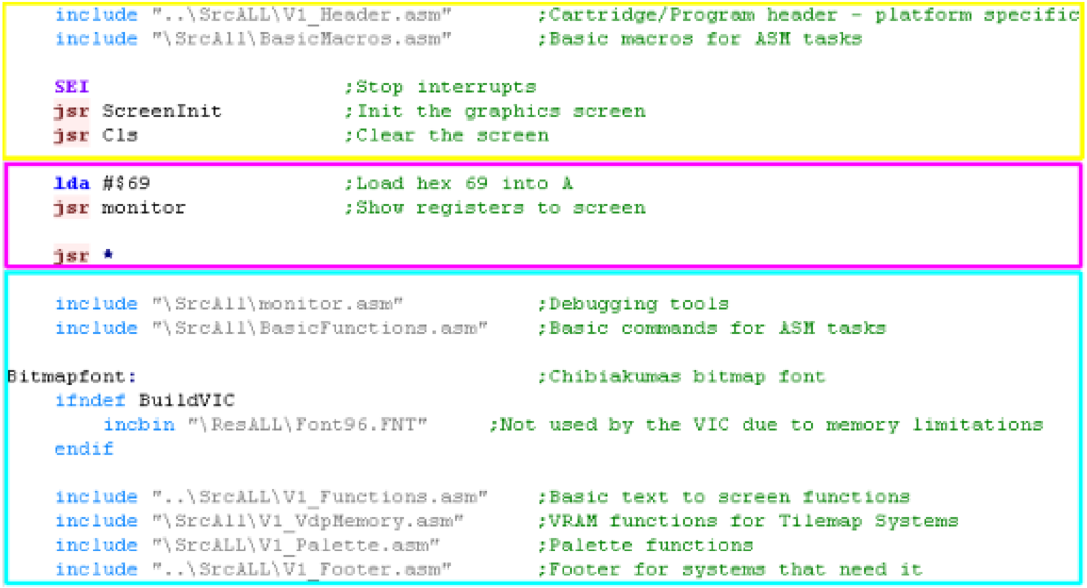
We will look at ALL of this code later, in the Platform specific series... but we can't do that until we understand 6502 itself!

The example shown to the right will load the A register with $69 (69 in hexadecimal)

We will then call the 'Monitor' function - which will show the state of the CPU registers to screen!

in this way, whatever the 6502 system you're learning and what emulator you're using, we'll be able to do things in a common way!

The example to the right is split into 3 parts:
The generic header - this will set up the system to a text screen
The program - this is where we do our work
The generic footer - The functions and



```
    include "..\SrcALL\V1_Header.asm"       ;Cartridge/Program header - platform specific
    include "\SrcAll\BasicMacros.asm"       ;Basic macros for ASM tasks

    SEI                         ;Stop interrupts
    jsr ScreenInit              ;Init the graphics screen
    jsr Cls                     ;Clear the screen

    lda #$69                    ;Load hex 69 into A
    jsr monitor                 ;Show registers to screen

    jsr *

    include "\SrcAll\monitor.asm"           ;Debugging tools
    include "\SrcAll\BasicFunctions.asm"    ;Basic commands for ASM tasks

Bitmapfont:                                 ;Chibiakumas bitmap font
    ifndef BuildVIC
        incbin "\ResALL\Font96.FNT"     ;Not used by the VIC due to memory limitations
    endif

    include "..\SrcALL\V1_Functions.asm"    ;Basic text to screen functions
    include "\SrcAll\V1_VdpMemory.asm"      ;VRAM functions for Tilemap Systems
    include "\SrcALL\V1_Palette.asm"        ;Palette functions
    include "..\SrcALL\V1_Footer.asm"       ;Footer for systems that need it
```

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

resources needed for the example to work

It's important to notice all the commands are inset by one tab... otherwise the Assembler will interpret them as labels.

Ti-84 Plus CE (eZ80 cpu)

The sample scripts provided with these tutorials will allow us to just look at the commands for the time being... we'll look at the contents of the Header+Footer in another series...

Of course if you want to do everything yourself that's cool... We're lerning the fundamentals of the 6502 - and they will work on any system with that processor... but you'll need to have some other kind of debugger/monitor or other way to view the results of the commands if you're going it alone!... Good luck!

Registers and Numbers

The 6502 has 3 main registers...

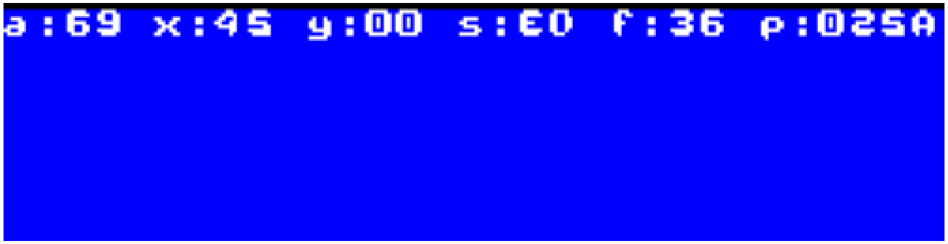
A is known as the Accumulator - we use it for all our maths
X and **Y** are our other 2 registers... we can use them as loop counters, temporary stores, and for special address modes... but we'll look at that later!

Lets learn our first commands... **LDA** stands for LoaD A... it sets A to a value... we can also do **LDX** or **LDY** to load X or Y registers!

Take a look at the example to the right... we're going to load A, X and Y... but notice... we're going to load them in different ways... A will be loaded with #$69... X will be loaded with #69... and Y will be loaded with 69... what will the difference be??

```
lda #$69                ;Load A with Hex $69
ldx #69                 ;Load A with Decimal 69
ldy 69                  ;Load A from memory address 0069
jsr monitor             ;Show the monitor
jmp *                   ;Infinite Loop
```

Well here's the result... the values are shown in Hex... so A=69... because specifying #$69 tells the assembler to use a HEX VALUE
but X=45... this is because without the $ the assembler used a Decimal value (45 hex = 69 decimal)
Y=0... why? well when we don't use a # the assembler gets the memory address.... so we read from memory address decimal 00069!... of course we can do $69 or $0069 to read from address hex 0069 too!



```
a:69 x:45 y:00 s:E0 f:36 p:025A
```

So #$xx = hex value .... #xx = decimal value.... and xx means read from address!

If you forget the # you're code is going to malfunction - as the assembler will use an address rather than a fixed value!

It's an easy mistake to make, and it'll mean your code won't work... so make sure you ALWAYS put a # at the start of fixed values!... or you WILL regret it!

Here are all the 6502 Assembler ways of representing values, and how they will be treated.

Prefix	Example	Z80 equivalent	Meaning
#	#16384	16384	Decimal Number
#%	#%00001111	%00001111	Binary Number
#$	#$4000	&4000	Hexadecimal number
#'	#'a	'a'	ascii value
	12345	(16384)	decimal memory address
$	$4000	(&4000)	Hexadecimal memory address

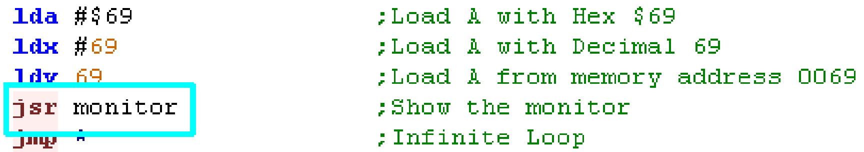
What's this JSR thing?... Jump to SubRoutine!

We've been using this **JSR** command... but what does it do?

Well JSR jumps to a subroutine... in this case **JSR monitor** will run the 'monitor' debugging subroutine... when the subroutine is done, the processor runs the next command

In this case that command is 'JMP *' which tricks the 6502 into an infinite loop!

JSR in 6502 is the equivalent of **GOSUB** in basic or **CALL** in z80.... we'll look at how to make our own subroutine in a later lesson!



```
lda #$69                        ;Load A with Hex $69
ldx #69                         ;Load A with Decimal 69
ldy 69                          ;Load A from memory address 0069
jsr monitor                     ;Show the monitor
jmp *                           ;Infinite Loop
```

*JMP is a jump command ... and * is a special command that means 'the current line' to the assembler... so 'JMP *' means jump to this line...*

This causes the 6502 to jump back to the start of the line... so it ends up running the jump command forever!... it's an easy way to stop the program for testing!

Adding and subtracting

The 6502 is a cut down version of the 6800... and would you believe it, one of the things they removed was the ADD and SUBtract commands!... so how can we do maths? well they did leave us some other commands... **ADC** and **SBC**... these add and subtract a value plus the 'Carry'....

The Carry is a single bit which is the overflow from a previous calculation... you see, in 8 bit maths you can't go over 255... so if you set A=255, then add 1... then A will become Zero, but the Carry will be 1... effectively the Carry is the 9th bit!

Don't worry if you don't understand that now... the important thing is we need to deal with the carry before we try to add or subtract with ADC and SBC!

Note... there is no way to add or subtract with X or Y... you have to store to memory, and use a command like ADC $0013.... which would ADD the 8 bit value in memory address $0013

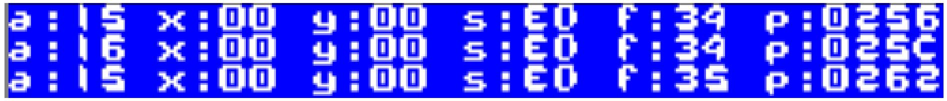
In this example, we're going to set A to Hex 15...
then we'll show it by calling the Monitor
then we'll add 1... and show it again with the monitor
then we'll subtract 1... and show it again with the monitor

We don't want the Carry affecting things so we have to **CL**ear the **C**arry with **CLC** before the **ADC** command...

However strangely if we don't want the Carry to affect subtraction, we have to **SE**t the **C**arry with **SEC**... before the **SBC** command - this is the opposite of the z80 command, but it's just the way the 6502 does things!

```
lda #$15                   ;Set A to Hexa
jsr monitor                ;Show the monitor

clc                        ;Clear the carry (need to do this before ADC to simulate ADD)
adc #1                     ;ADD decimal 1 with carry
jsr monitor                ;Show the monitor

sec                        ;Set the carry (need to do this before SBC to simulate SUB)
sbc #1                     ;Subtract Decimal 1 with the carry
jsr monitor                ;Show the monitor
```

Here is the result... you can see we go from 15, to 16, then back to 15!



```
a:15  x:00  y:00  s:E0  f:34  p:0256
a:16  x:00  y:00  s:E0  f:34  p:025C
a:15  x:00  y:00  s:E0  f:35  p:0262
```

Moving data between registers

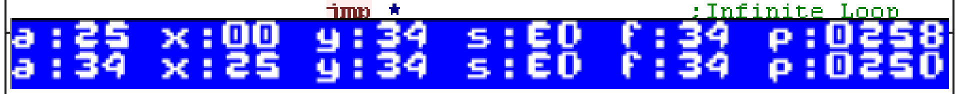
We know how to set all the registers, but what if we have a value in one register, and we want to transfer it to another...
Well, we can use **TAX** and **TAY** to **T**ransfer **A** to **X**...or **T**ransfer **A** to **Y**!

We can also use **TXA** or **TYA** to **T**ransfer **X** to **A**... or **T**ransfer **Y** to **A**!

What if we want to transfer **X** to **Y**? (or Y to X) ... well we can't directly, so we'd have to do **TXA**... then **TAY**

```
lda #$25                    ;Set A to $25
ldy #$34                    ;Set Y to $34
jsr monitor                 ;Show the monitor

tax                         ;Transfer A to X
tya                         ;Transfer Y to A
jsr monitor                 ;Show the monitor

jmp *                       ;Infinite Loop
```

You can see the result here... First we set A to $25 and Y to $34 - the result is shown on the first line
Then we transfer A to X... and Y to A... the result is shown on the second line.



```
a:25 x:00 y:34 s:E0 f:34 p:0258
a:34 x:25 y:34 s:E0 f:34 p:0250
```

Storing back to memory!

Remember we learned that using LDA with a number without a # means it will load from that numbered address? - so LDA $13 will LoaD A from hex address $0013?
Well we can also STore A with the **STA** command!... we can also STore X with **STX**, or Store Y with **STY**!

In this example we'll use STA to store some values to memory addresses $0011 and $0012

We'll then set the Accumulator to $13 and add these two memory addresses to the accumulator.... finally we'll use STA again to store the result to memory address $0013

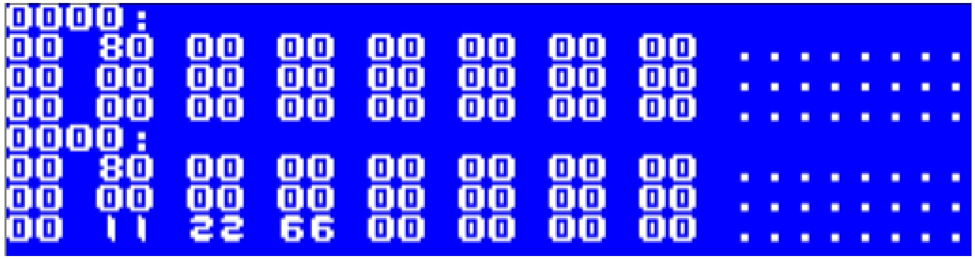
When it comes to showing the result, we'll use another debugging subroutine I wrote called **MemDump**... this will dump a few lines of data to the screen... in this case we'll show 3 lines (of 8 bytes) from memory address $0000-$0018... In this example, we'll show the memory before, and after we do the writes.

* **Warning** * If you're not using my sample code, these commands may overwrite system variables - and cause something strange to happen!

```
jsr MemDump                 ;Dump an address to screen
    dw $0000                ;Address to show
    db $3                   ;Lines to show

lda #$11                    ;Load A with Hex 11
sta $0011                   ;Save to memory address $0011
lda #$22                    ;Load A with Hex 22
sta $0012                   ;Save to memory address $0022

lda #$33                    ;Load A with Hex 33
clc                         ;Clear the carry (we don't want to add it!)
adc $0011                   ;Add the value at address $0011
adc $0012                   ;Add the value at address $0012
sta $0013                   ;Store the result to address $0013

jsr MemDump                 ;Dump an address to screen
    dw $0000                ;Address to show
    db $3                   ;Lines to show

jmp *                       ;Infinite Loop
```

Here's the result of the programm running... you can see the bytes $11, $22 and $66 were written... these are the two values stored at the start... and then the result of these two added to the $33 loaded into the accumulator

Want to try something else?? Why not change CLC to SEC and ADC to SBC... and see what happens!



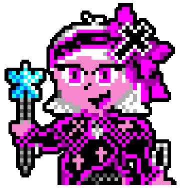
```
0000:
00 80 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
0000:
00 80 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 11 22 66 00 00 00 00 ........
```

The first 256 bytes of memory $0000-$00FF are special on the 6502... in fact there's a lot we're not mentioning about reading and writing memory... but it's coming soom!

Also the memory from $0100-$01FF is also special... it's used by the stack!... don't know what that is? don't worry... we'll come to that!

Be Careful writing to memory on different systems... This example may not work write on some systems... The PC-Engine is weird... unlike every 6502... the range $0000-$01FF is NOT memory... that area is at $2000-$21FF

Why? because it's not actually a 6502... its a HuC6280... it's almost the same as a 6502... but it has some extras and weirdness!



Lesson 2 - Addressing modes on the 6502

The 6502 has very few registers - but it makes up for this with a mind boggling number of addressing modes!

You won't need them all at first, but you should at least understand what they all do - lets see some examples of how they work!

Lets try them all out with some simple examples!

Prepearation...

In order to run these examples we're going to need to set up some areas of memory, by filling them with test values.

The code to the right will do the work (via a Function called LDIR - which copies memory areas)... don't worry how it works for now, it's too complex at this time!

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

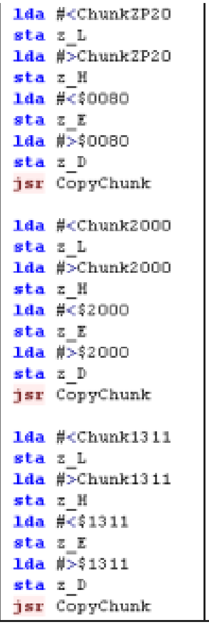
Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

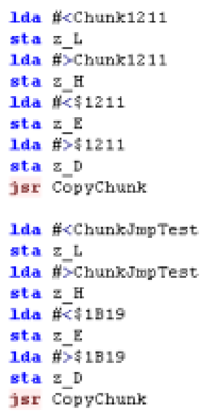
Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the



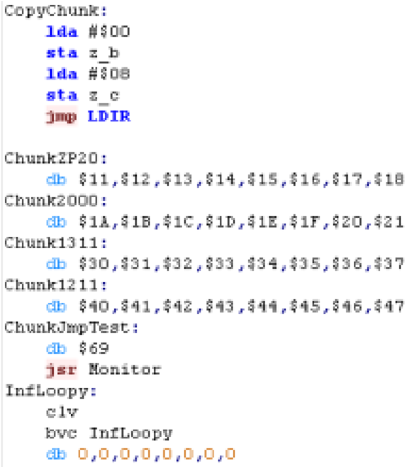
```
lda #<ChunkZP20
sta z_L
lda #>ChunkZP20
sta z_H
lda #<$0080
sta z_E
lda #>$0080
sta z_D
jsr CopyChunk

lda #<Chunk2000
sta z_L
lda #>Chunk2000
sta z_H
lda #<$2000
sta z_E
lda #>$2000
sta z_D
jsr CopyChunk

lda #<Chunk1311
sta z_L
lda #>Chunk1311
sta z_H
lda #<$1311
sta z_E
lda #>$1311
sta z_D
jsr CopyChunk
```



```
lda #<Chunk1211
sta z_L
lda #>Chunk1211
sta z_H
lda #<$1211
sta z_E
lda #>$1211
sta z_D
jsr CopyChunk

lda #<ChunkJmpTest
sta z_L
lda #>ChunkJmpTest
sta z_H
lda #<$1B19
sta z_E
lda #>$1B19
sta z_D
jsr CopyChunk
```

Ti-84 Plus CE (eZ80 cpu)

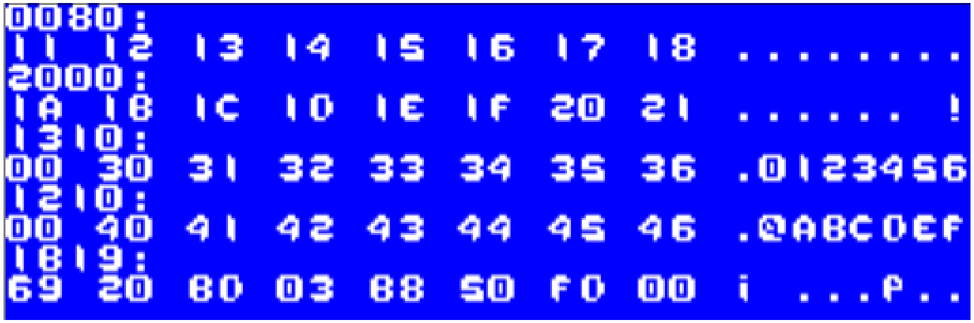
Here is the rest of the Chunk copying code, and the data copied... again, you don't need to worry about this for now.



```
CopyChunk:
    lda #$00
    sta z_b
    lda #$08
    sta z_c
    jmp LDIR

ChunkZP20:
    db $11,$12,$13,$14,$15,$16,$17,$18
Chunk2000:
    db $1A,$1B,$1C,$1D,$1E,$1F,$20,$21
Chunk1311:
    db $30,$31,$32,$33,$34,$35,$36,$37
Chunk1211:
    db $40,$41,$42,$43,$44,$45,$46,$47
ChunkJmpTest:
    db $69
    jsr Monitor
InfLoopy:
    clv
    bvc InfLoopy
    db 0,0,0,0,0,0,0,0
```

Prepearation... the result...

Here is the important bit... THIS is the data as it appears in memory when the program runs... you may want to refer back to this if you wish!



```
0080:
11 12 13 14 15 16 17 18 ........
2000:
1A 1B 1C 1D 1E 1F 20 21 .......!
1310:
00 30 31 32 33 34 35 36 .0123456
1210:
00 40 41 42 43 44 45 46 .@ABCDEF
1B19:
69 20 80 03 B8 50 FD 00 i ...P..
```

Note: These tutorials will not work on all systems... for example most will not work on the PC engine, because the zero page is not at &0000!
They may also not work on the NES or SNES, because the &2000 area has a special purpose on those systems.

They have all been tested on the BBC.... but don't worry... the theory shown here is based on the principals of the 6502 - so will work on ANY 6502 based system!

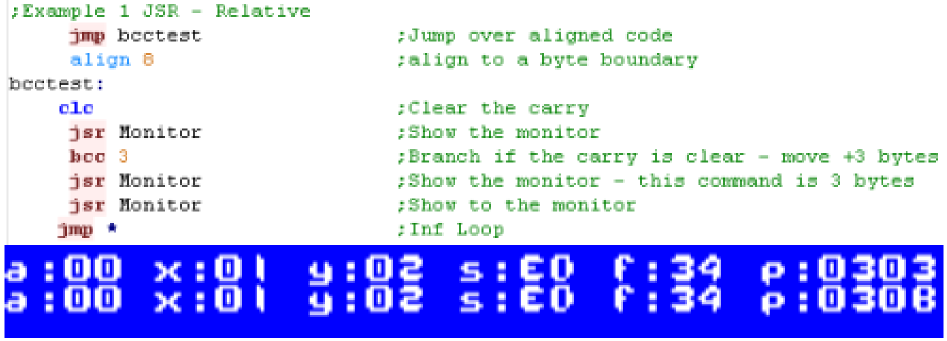
We're all set up now... lets try out all the addressing options... we'll look at the theory, and an example program... then we'll see the result in the registers in a screenshot from the BBC version
We'll be reading in all these examples... but many of the commands can be used for other commands.. please see the Cheatsheet for more details.

1.Relative Addressing

Relative Addressing is where execution (the program counter) jumps to a position relative to the current address - it can be 127 bytes after the calling line, or 128 bytes before.... This means the code will be 'relocatable' - we can move it in memory and it will still work, but we can't jump more than 128 bytes! There are all kinds of 'Branch' commands... here we've used 'Branch if Carry Clear'... we'll look at the others in a later lesson BCC ALWAYS takes a fixed number (not an address), so we don't have to use # with BCC in vasm!... that said, we can just use labels (names that appear at the far left, and let the assembler work out the maths.	 BCC 3 (assuming CLC) Before PC=$1002 **Result** PC=$1008
Take a look at the example to the right... there are 3 Monitor commands... but only	

2 show on the screen... this is because the BCC skips over one

The "Program Counter" (shown as P) stores the byte of the end of the last command.... A "JSR Monitor" takes 3 bytes, "BCC 3" takes 2... hopefully the numbers the program counter shows will now make sense if you add up the commands!



```
;Example 1 JSR - Relative
     jmp bcctest             ;Jump over aligned code
     align 8                 ;align to a byte boundary
bcctest:
    clc                      ;Clear the carry
     jsr Monitor             ;Show the monitor
     bcc 3                   ;Branch if the carry is clear - move +3 bytes
     jsr Monitor             ;Show the monitor - this command is 3 bytes
     jsr Monitor             ;Show to the monitor
    jmp *                    ;Inf Loop
```

```
a:00 x:01 y:02 s:E0 f:34 p:0303
a:00 x:01 y:02 s:E0 f:34 p:0308
```

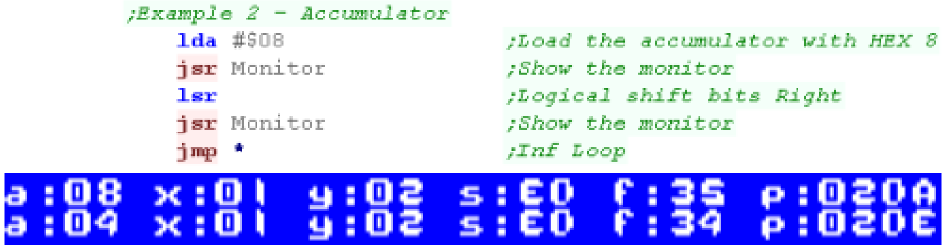
2.Accumulator Addressing

Accumulator addressing sounds more complex than it is!

Effectively it's a command with no parameters - it just changes the accumulator in some way....

Accumulator works on A	LSR	Before A=8	**Result** A=4

For Example LSR shifts the bits to the left... don't worry if you don't understand it, we'll look at it later!



```
;Example 2 - Accumulator
    lda #$08                ;Load the accumulator with HEX 8
    jsr Monitor             ;Show the monitor
    lsr                     ;Logical shift bits Right
    jsr Monitor             ;Show the monitor
    jmp *                   ;Inf Loop
```

```
a:08 x:01 y:02 s:E0 f:35 p:020A
a:04 x:01 y:02 s:E0 f:34 p:020E
```

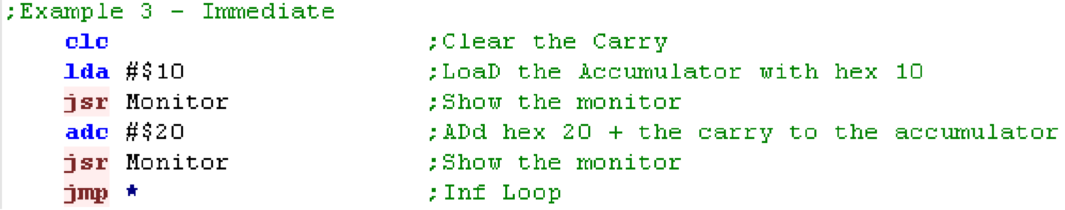
3.Immediate Addressing

Again, Immediate sound scary... but it's really easy... it's just a simple number in the code, specified with a #
As we've already learned... we can use # followed by $ to sepcify a hexadecimal number.

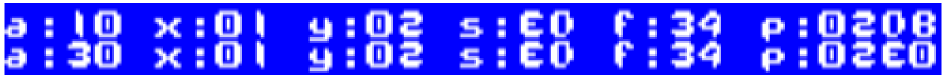
Immediate #nn &nn	ADC #$20 (assuming CLC)	Before A=$10	**Result** A=$30

In this example we will add Hex 10 and Hex 20... the result is obviously 30!

Why not try using different numbers,remove the $ to stop using hexadecimal..., or SBC... don't forget to change CLC to SEC if you do!



```
;Example 3 - Immediate
    clc                      ;Clear the Carry
    lda #$10                 ;LoaD the Accumulator with hex 10
    jsr Monitor              ;Show the monitor
    adc #$20                 ;ADd hex 20 + the carry to the accumulator
    jsr Monitor              ;Show the monitor
    jmp *                    ;Inf Loop
```



```
a:10 x:01 y:02 s:E0 f:34 p:0208
a:30 x:01 y:02 s:E0 f:34 p:02E0
```

4.Zero Page Addressing

The Zero Page is the 6502's special trick... addresses between **$0000 and $00FF** are called the 'Zero Page'... these can be stored as a single byte... so $FF would refer to address $00FF

Because the address is stored as a single byte - it's fast, and the Zero page can do things that other addresses cannot!

The 6502 uses this 'zero page' like a bank of 255 registers - allowing the 6502 with it's just 3 registers to do the things the Z80 did with over a dozen!

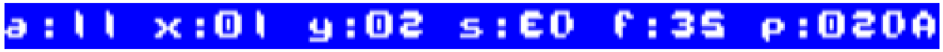
Zero Page
$zp
($00zp)

LDA $80	$0080	11	**Result**
	$0081	12	A=$11
	$0082	13	

In this example we'll load from zero page address $80.... note that if we did LDA #$80 then we would load the Value $80 not from the address...

This is important - you don't want to make that mistake (too often!)

```
;Example 4 - Zero Page / Direct page
    lda $80                  ;Load A from ZP address $80 = $0080
    jsr Monitor              ;Show the monitor
    jmp *                    ;Inf Loop
```



```
a:11 x:01 y:02 s:E0 f:35 p:020A
```

The Zero Page (Sometimes called the Direct Page - usually when it's not at $0000) is effectively the 'tepmporary store' for all the data we can't get into the A,X and Y registers...

We can use different numbered addresses for different purposes, but many may be used by the machines firmware!

5. Zero Page Indexed X (or Y with LDX / STX) Addressing

When we specify ,X or ,Y after an address it becomes an offset... the register is added to the address in the zero page... and the value is retrieved from the resulting address...

Note - you typically have to use X for this addressing mode... however LDX and

Zero Page Indexed X
$zp,X
($00zp+X)

LDA $80,X	$0080	X	11	**Result**
X=1	$0081		12	A=$12
	$0082		13	

Zero Page Indexed Y
$zp,Y
($00zp+Y)

LDX $20,Y	$0080	Y	11	**Result**
Y=2	$0081		12	A=$13
(only for LDX,STX)	$0082		13	

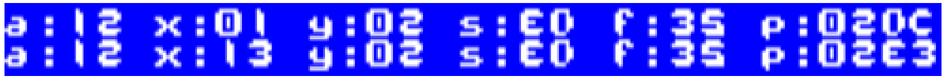
STX are as special case, and we can use Y because we can't use X if it's the source or destination of the command

Note... LDA $20,Y is not a valid command... however the assembler will covert it to LDA $0020,Y which IS... but it takes an extra byte, so is not as efficient!

As you can see here we're using the Zero Page, and X and Y register....

take a look at the values we wrote to the Zero Page at the start, and try changing X,Y and the source location ($80) to other values.

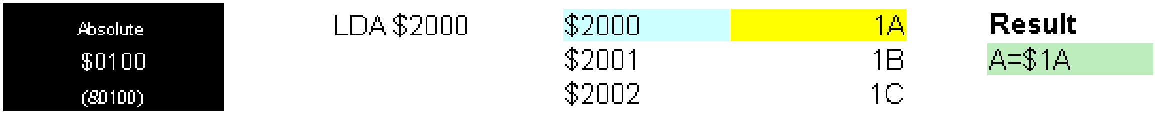
```
;Example 5 - Zero Page Indexed X,Y
    ldx #1                  ;Load X with 1
    lda $80,X               ;Load A from Zeropage $80 + X (so load from Zeropage $81)
    jsr Monitor             ;Show the monitor
    ldy #2                  ;Load Y with 1
    ldx $80,Y               ;ZP,Y only works with LDX,STX...
    jsr Monitor             ;Show the monitor
    jmp *                   ;Inf Loop
```



```
a:12 x:01 y:02 s:E0 f:35 p:020C
a:12 x:13 y:02 s:E0 f:35 p:02E3
```

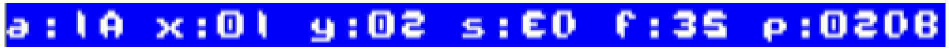
6. Absolute Addressing

Of course we can't always read and write in the zero page... we'll want to specify the whole address... this takes an extra byte - so the command will be 3 bytes total and is slower, but we can get data from the whole 64k range ($0000-$FFFF)



Absolute addressing is good for variables we're not storing in the zero page (often most of the Zero page is used by the firmware!)... but isn't very good for reading in lots of data (like sprite images)... for that we want indirect addressing - which we'll look at soon!

```
;Example 6 - Absolute
    lda $2000               ;Load from address $2000
    jsr Monitor             ;Call the monitor
    jmp *                   ;Inf Loop
```

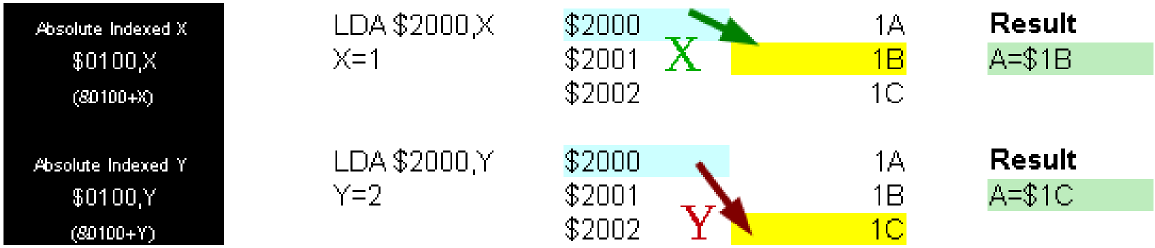


```
a:1A x:01 y:02 s:E0 f:35 p:0208
```

7. Absolute Indexed Addressing With X,Y

When we want to read from multiple addresses, we can used Indexed addressing... this adds X or Y to an address - so we can change X/Y to read in from a range using a Loop!... we'll learn how to do a loop very soon!

$xxxx,Y can be used with many commands, but $xxxx,X has more



Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

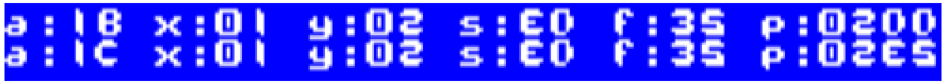
$150 calculator: Unboxing the

options... check out the cheatsheet for more info!

Ti-84 Plus CE (eZ80 cpu)

Changing X and Y allow you to change the source address without changing the LDA line.... we'll learn how to do this in loops and functions later.

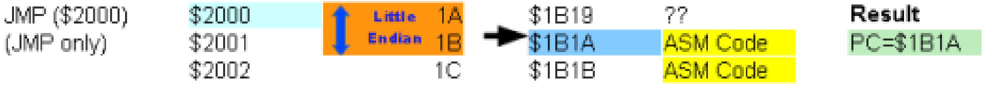
```
; Example 7 - Absolute Indexed
    ldx #1                    ;Load X with 1
    lda $2000,X               ;Load A from address ($2000+X) so ($2001)
    jsr Monitor

    ldy #2                    ;Load Y with 2
    lda $2000,Y               ;Load A from address ($2000+X) so ($2001)
    jsr Monitor               ;Call the monitor
    jmp *                     ;Inf Loop
```



```
a:1B x:01 y:02 s:E0 f:35 p:0200
a:1C x:01 y:02 s:E0 f:35 p:02E5
```

8. Absolute Indirect

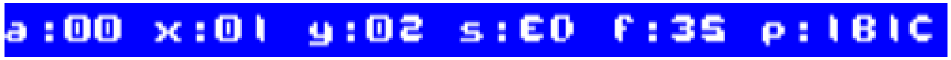
We can directly read a 16-bit value from another 16-bit address ($0000-$FFFF) In one special... the JuMP command (for all other cases we need to use the zero page. This can be used to reprogram parts of your progam - allowing alternate routines to be 'switched' in.



In this example we use ($2000)... this loads in two bytes $1B1A and then jumps to that address (sets the PC to 1B1A)...

Our setup put a "JSR MONITOR" at this address... so we see the contents of the registers... notice P (the program counter) is $1B1C... the last byte of the 3 byte "JSR MONITOR" command

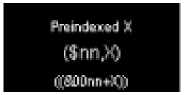
```
; Example 8 - Absolute Indirect
    jmp ($2000) ; ($nnnn) only works with Jump
    jmp *                      ;Inf Loop
```

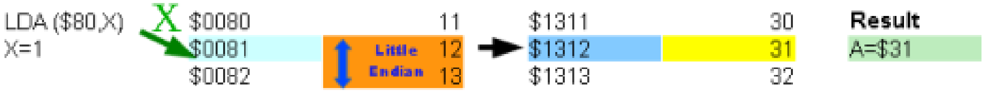


```
a:00 x:01 y:02 s:E0 f:35 p:1B1C
```

9. Preindexed Indirect Addressing with X

Pre-inxexed Indirect with X regsiter uses the ZeroPage... X is added to the ZeroPage.... the two consecutive bytes are read in from the zero page, and these are used





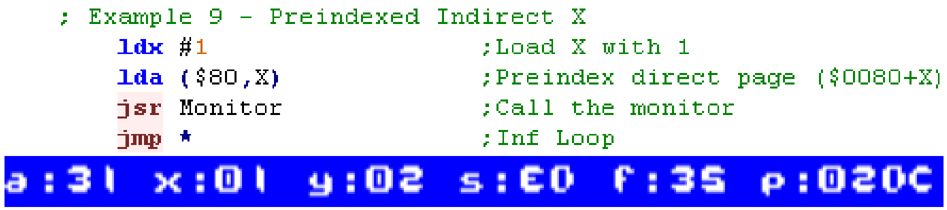
as an address... a byte is read from that address... Note... the data is stored in 'Little Endian' format... meaning the lower value byte comes first

This is all very cofusing!... but think of it like this... two bytes of the zeropage are a 'temporary address' pointing to the actual data we will read

We can use these to simulate 'Z80 registers'... by setting one as an L register for the low byte, and the next as the H register for the high byte....
This is how we get around the 6502's lack of registers!... don't worry about it if you don't understand yet... we'll see this a lot later!

In this example we've got X set to 1... so we end up loading a byte from the address made up of bytes at $0081 and $0082 - remember they are in reverse order because it's little endian!

we then show the result to screen.... of course setting X to 0... and changing $80 to $81 would have the same effect.

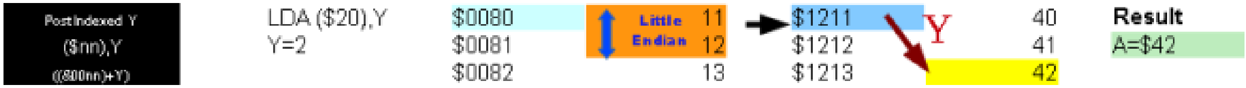


```
; Example 9 - Preindexed Indirect X
    ldx #1                  ;Load X with 1
    lda ($80,X)             ;Preindex direct page ($0080+X)
    jsr Monitor             ;Call the monitor
    jmp *                   ;Inf Loop
```

```
a:31 x:01 y:02 s:E0 f:35 p:020C
```

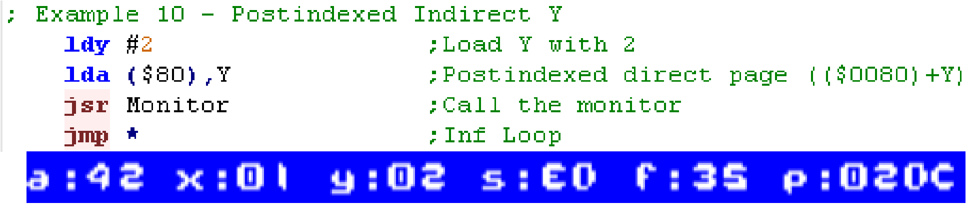
10. Postindexed Indirect Addressing with Y

Post-Indexed with the Y register also use the Zero Page... two concecutive bytes are read in from the Zero page to make an address... but the Y register is then added to THAT address... and the final value is read from the resulting address.



With this option, Effectively, if we store an address in the Zero page... we can use Y as a counter and read from consecutive addresses... we can use this in a loop - we'll learn how to do that later

Y is 2 in this example, so 2 is added to the address in ZeroPage ($0080-$0081)... if we change Y then the final address will change by the same amount



```
; Example 10 - Postindexed Indirect Y
    ldy #2                         ;Load Y with 2
    lda ($80),Y                    ;Postindexed direct page (($0080)+Y)
    jsr Monitor                    ;Call the monitor
    jmp *                          ;Inf Loop
  a:42 x:01 y:02 s:E0 f:35 p:020C
```

11. Indirect Addressing (65c02 only)

This is a special mode only available on 65c02 used by the Lynx, Snes, PcEngine and Apple II.... Effectively it's the same as Preindexed when X=0... or PostIndexed when Y=0... this is how we can simulate this addressing mode if we need to do this on the other machines!

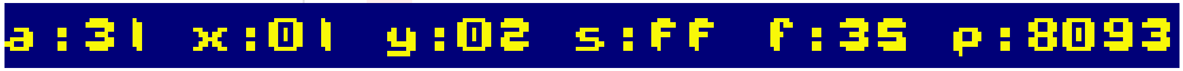
Indirect (65C02) ($nn) (($00nn))	LDA ($81)	$0080		11	$1311	30	**Result**
		$0081	Little Endian	12	$1312	31	A=$31
		$0082		13	$1313	32	

It uses a pair of bytes in the Zero page as an address, and uses that address for the result

It would be nice to have this mode on the other CPU's, but we don't... however we can simulate it!

to fake it on other machines we set X=0 then use LDA ($81,X) or we set Y=0 and then use LDA ($81),Y

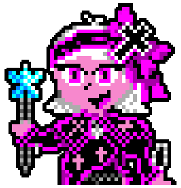
```
; Example 11 - Indirect (65C02 only)
    lda ($81)                   ;Load the address from ($0081)
    jsr Monitor                 ;Call the monitor
    jmp *                       ;Inf Loop
```



```
a:31 x:01 y:02 s:ff f:35 p:8093
```

You won't see much '65c02 only' code in these tutorials - so all the code will work on all systems, we only use the basic 6502 commands

Of course you're free to use them if you wish, just remember - it will mean you can't port your code to another system as easily!



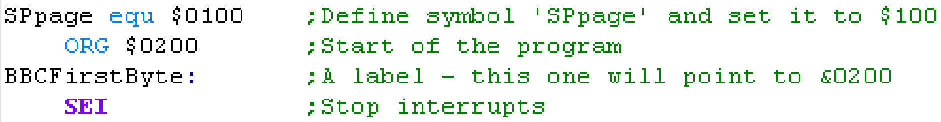
Lesson 3 - Loops and Conditions

We've had a breif introduction to 6502, and now we understand the Addressing modes we can look properly at 6502, lets take a look at some more commands, an how to do 'IF Then' type condions and Loops!

Some overlooked fundamentals!

We've been cheating a little, we've overlooked a few important commands - they're hidden in the header, but we really need to know them!... before we start the proper lesson, lets look at them now!

We're going to need to know ALL the details of assembly to create a working program, and something have been hidden until now! but we need to ensure we know everything.



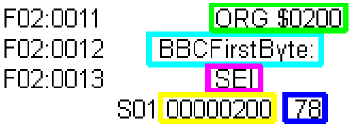
```
SPpage equ $0100     ;Define symbol 'SPpage' and set it to $100
    ORG $0200        ;Start of the program
BBCFirstByte:        ;A label - this one will point to &0200
    SEI              ;Stop interrupts
```

ORG and Labels - Positioning data in memory

Because we're compiling to a 8-bit cpu with a 16-bit address bus, our compiled code filles maps to a fixed address within the memory space... this is important, because while branch commands like BCC are an 'offset'... JMP commands will 'Jump' to a specific numbered address

to the right, you can see how the code will compile - this is the 'Listing.txt' file, showing the source code and the resulting binary output.

The **SEI** command is compiled to the byte **$78** - this is the command as the CPU sees it... because of the **ORG** command, the code is compiled to the address $**0200**...

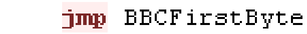


```
F02:0011        ORG $0200
F02:0012      BBCFirstByte:
F02:0013        SEI
         S01 00000200 78
```

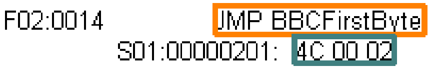
Using Labels

We also have a **Label**... Labels must be at the far left of the screen... all other commands must be inset

n this example, the label will be defined as address $0200 - so if we use it in a **Jump** command (hex $4C) , it will be **compiled to** that address (in reverse endian - so $0200 becomes $00 $02)



```
    jmp BBCFirstByte
```

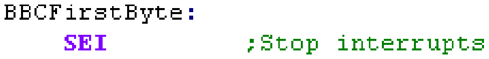


```
F02:0014          JMP BBCFirstByte
         S01:00000201: 4C 00 02
```

SEI - Disabling interrupts

Interrupts are where the CPU does other tasks whenever it wants!

For simplicity at this stage, we want to stop that, so we use SEI to "Set the Interrupt Mask"



```
BBCFirstByte:
    SEI          ;Stop interrupts
```

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Don't worry about interrupts yet, we'll look at them later... so for now we just need to know how to turn them off

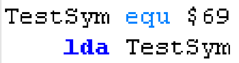
Ti-84 Plus CE (eZ80 cpu)

Symbol definitions

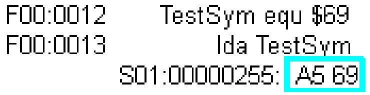
Symbols are similar to labels... they allow us to give 'name' (like TestSym) a 'Value' ... rather than using the value later, we can just use the symbol... Using symbols makes it easy for us to program, as we can use explainatory text rather than meaningless numbers.

the assembler will convert the symbol name to its original value... we just use EQU to define the definition... in the example **once assembled** LDA converts to byte $A5... and TestSym has a value of $69

In VASM, like labels, symbol definitions must be at the far left of the screen



```
TestSym equ $69
    lda TestSym
```

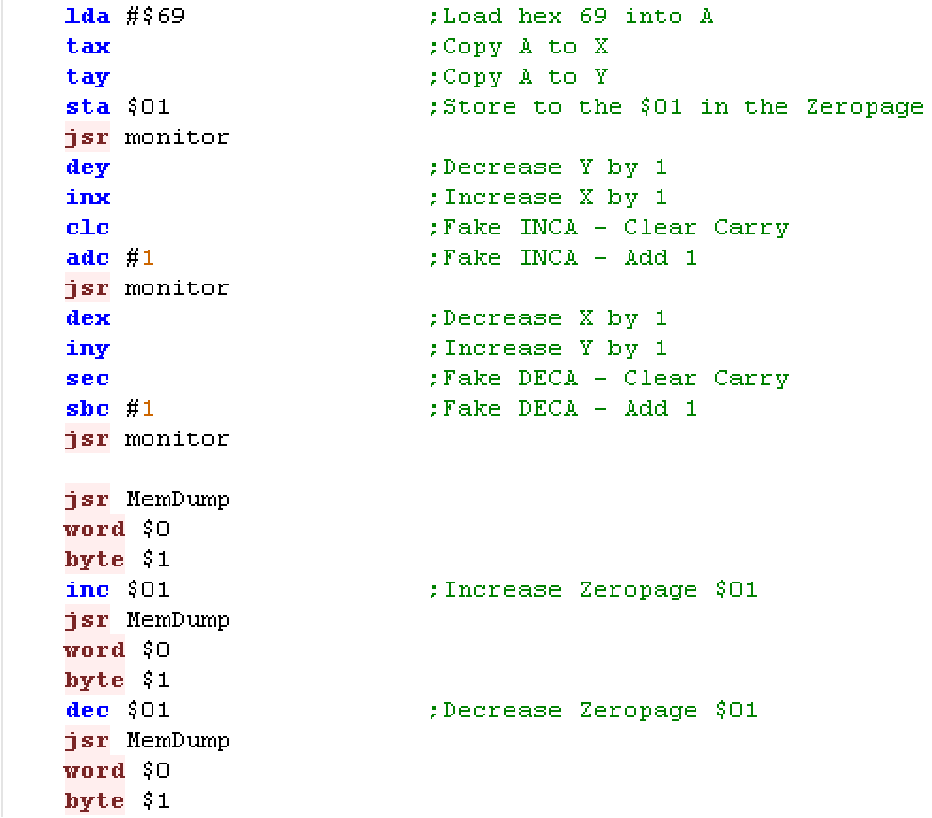


```
F00:0012       TestSym equ $69
F00:0013               lda TestSym
               S01:00000255:  A5 69
```

INC and DEC

There will be frequent times when we need to increase and decrease values by just 1
For the X or Y registers we can do this with INX and DEX
We can increase values in the ZeroPage by using INC $01 or DEC $01

rather annoyingly there is no INC or DEC command on the 6502... so we have to simulate it, by clearing the carry, and adding one (CLC, ADC #1)

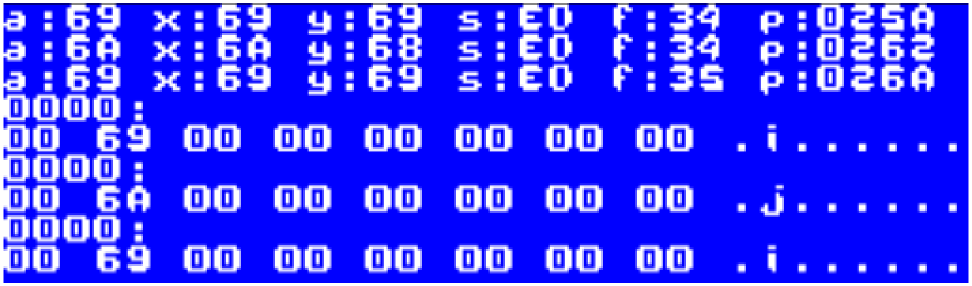


```
    lda #$69                ;Load hex 69 into A
    tax                     ;Copy A to X
    tay                     ;Copy A to Y
    sta $01                 ;Store to the $01 in the Zeropage
    jsr monitor
    dey                     ;Decrease Y by 1
    inx                     ;Increase X by 1
    clc                     ;Fake INCA - Clear Carry
    adc #1                  ;Fake INCA - Add 1
    jsr monitor
    dex                     ;Decrease X by 1
    iny                     ;Increase Y by 1
    sec                     ;Fake DECA - Clear Carry
    sbc #1                  ;Fake DECA - Add 1
    jsr monitor

    jsr MemDump
    word $0
    byte $1
    inc $01                 ;Increase Zeropage $01
    jsr MemDump
    word $0
    byte $1
    dec $01                 ;Decrease Zeropage $01
    jsr MemDump
    word $0
    byte $1
```

Here you can see the results of the program...

The first thee lines show the status of the registers at each stage.... and we can see how A,X and Y are affected by each stage of the program

The lower half shows the zero page - and we can see how $01 goes up and down as we do INC and DEC commands

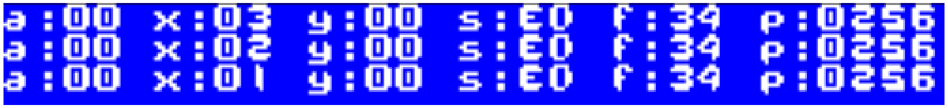


```
a:69 x:69 y:69 s:E0 f:34 p:025A
a:6A x:6A y:68 s:E0 f:34 p:0262
a:69 x:69 y:69 s:E0 f:35 p:026A
0000:
00 69 00 00 00 00 00 00 .i......
0000:
00 6A 00 00 00 00 00 00 .j......
0000:
00 69 00 00 00 00 00 00 .i......
```

Branch on condition

Branches allow us to do things depending on a condition... we can use this to create a loop!
Because we don't have a DEC command for the accumulator, it's often easier to use X or Y as a loop counter.
if we use DEX to decrement the counter, and BNE will jump back until the counter reaches zero... note that BNE needs to be immediately after the decrement command as other commands may alter the Z flag

```
    ldx #3                    ;Set X to 3
DecTestAgain:
    jsr monitor
    dex                       ;Decrease X by one
    bne DecTestAgain          ;Jump back until Zero flag is set
    jmp *                     ;Infinite Loop
```



```
a:00 x:03 y:00 s:E0 f:34 p:0256
a:00 x:02 y:00 s:E0 f:34 p:0256
a:00 x:01 y:00 s:E0 f:34 p:0256
```

There are a wide variety of Branch commands for different condition codes.

Command	Meaning	Literal Meaning	Description
BCC	Branch if Carry Clear	flag C=1	Is there any carry caused by last command?*
BCS	Branch if Carry Set	flag C=0	Is there any carry caused by last command?*
BEQ	Branch if Equal	flag Z=1	Is the result of the last command zero?
BMI	Branch if Minus	flag S=1	Is the result of the last command <128
BNE	Branch if Not Equal	flag Z=0	Is the result of the last command zero?
BPL	Branch if Plus	flag S=0	Is the result of the last command >=128
BVC	Branch if Overflow Clear	flag V=0	Is there any overflow caused by there last command?*
BVS	Branch if Overflow Set	flag V=1	Is there any overflow caused by there last command?*

If a previous addition command caused a value over 255 then Carry will be set... Overflow is a bit odd... it's affected if Addition/Subtraction goes over the 128 boundary (if it changes from positive to negative) it's also set by BIT commands

Comparing to another value with CMP, CPX and CPY

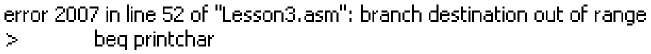
If you don't want to see if a register is zero, you can compare to a different value with CMP... then perform one of the

commands.... effectively, CMP 'simulates' a subtraction

Basic command	Comparison	6502 command	Z80 equivalent	68000 equivalent
if Val1>=Val2 then goto label	>=	**BCS** label	JP NC,label	BGE label
if Val1<Val2 then goto label	<	**BCC** label	JP C,label	BLT label
if Val1=Val2 then goto label	=	**BEQ** label	JP Z,label	BEQ label
if Val1<>Val2 then goto label	<>	**BNE** label	JP NZ,label	BNE label

Conditional Jumping far away with JMP, or calling a subroutine with JSR

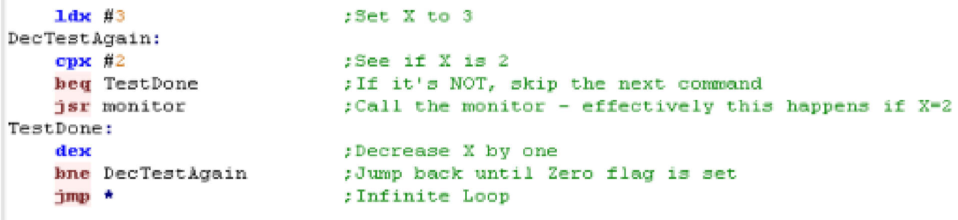
Branch commands are pretty limited, they can only jump 128 bytes away, if you try to jump further you will get an error



```
error 2007 in line 52 of "Lesson3.asm": branch destination out of range
>          beq printchar
```

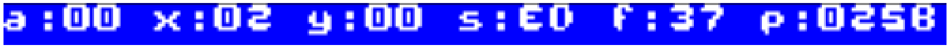
If you need to jump further, or you want to use JSR with a condition you have to do things backwards!.... jump OVER the JSR or JMP command if the condition is NOT met

For example... if you want to call the Monitor if X=2... then you have to use a branch command to jump OVER the call if X is not 2...



```
    ldx #3                  ;Set X to 3
DecTestAgain:
    cpx #2                  ;See if X is 2
    beq TestDone            ;If it's NOT, skip the next command
    jsr monitor             ;Call the monitor - effectively this happens if X=2
TestDone:
    dex                     ;Decrease X by one
    bne DecTestAgain        ;Jump back until Zero flag is set
    jmp *                   ;Infinite Loop
```

The result is that the monitor is called only when X=2... we've faked a 'Jump to SubRoutine on Equal' command... we can also do the same with a JMP to get further than 128 bytes away!



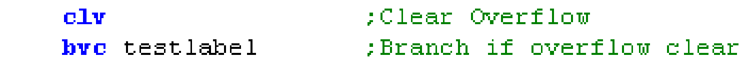
```
a:00 x:02 y:00 s:E0 f:37 p:0258
```

Using BVC to simulate BRA

JMP jumps to a specific memory address, where as BEQ and other branch commands jump to a relative position...
There may be cases where you want to write code that can be relocated... copied to a new memory address and still executable... JMP will not work in this case, but branch will...

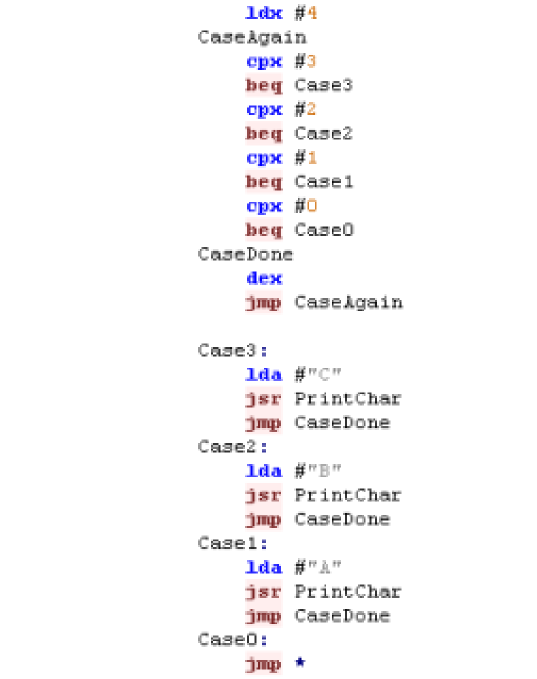
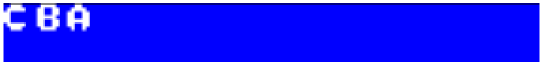
the 65c02 has a BRA command for this purpose (branch always)... but the 6502 does not... we can however simulate it by clearing the rarely used overflow with CLV, then using BVC

Don't worry if you don't see any reason to do this - you may never need to! if you don't know why you'd need relocatable code - then you don't need it!



```
    clv             ;Clear Overflow
    bvc testlabel   ;Branch if overflow clear
```

Multiple conditions for a Case statement

It's important to understand that ALL other languages convert to assembly... so anything Basic or C++ can do can be done in ASM! We can chain multiple branches together to create 'If Then ElseIf' commands or even create 'Case' Statements in assembly, just by chaining multiple branch commands together.	
The result will be the program will branch out to each of the subsections depending on X	

```
    ldx #4
CaseAgain
    cpx #3
    beq Case3
    cpx #2
    beq Case2
    cpx #1
    beq Case1
    cpx #0
    beq Case0
CaseDone
    dex
    jmp CaseAgain

Case3:
    lda #"C"
    jsr PrintChar
    jmp CaseDone
Case2:
    lda #"B"
    jsr PrintChar
    jmp CaseDone
Case1:
    lda #"A"
    jsr PrintChar
    jmp CaseDone
Case0:
    jmp *
```

```
CBA
```

Through a combination of conditions we can do any condition in assembly that C++ or Basic can do... that's because those languages compile DOWN to assembly...

That said, it will take a lot more work in assembly!

Lesson 4 - Stacks and Math

Now we know how to do conditions, jumping and the other basics, it's time to look at some more advanced commands and principles of Assembly..

Lets take a look!

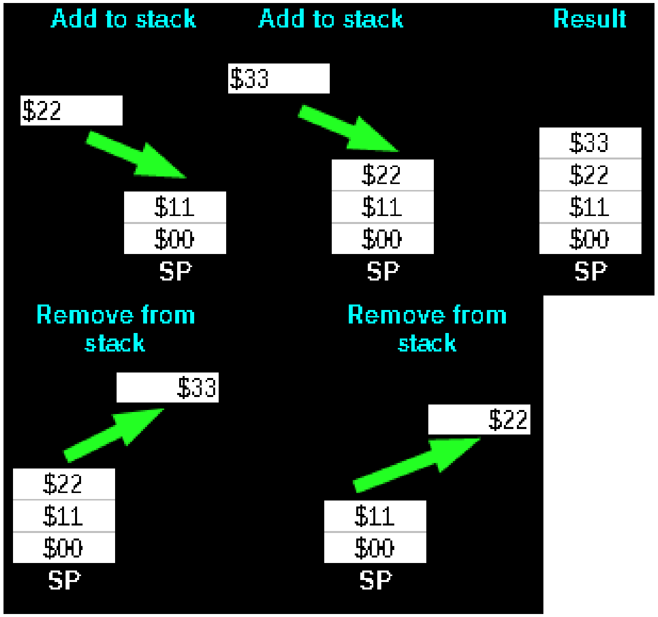
Stack Attack!

'Stacks' in assembly are like an 'In tray' for temporary storage...	

Imagine we have an In-Tray... we can put items in it, but only ever take the top item off... we can store lots of paper - but have to take it off in the same order we put it on!... this is what a stack does!

If we want to temporarily store a register - we can put it's value on the top of the stack... but we have to take them off in the same order...

The stack will appear in memory, and the stack pointer goes DOWN with each push on the stack... so if it starts at $01FF and we push 1 byte, it will point to $01FE



Push me - Pull me!

on the Z80 we have Push and Pop, but on the 6502 it's Push and Pull!

We PUSH values onto the top of the stack to back them up, and PULL them off!

Our 6502 has 4 registers we may want put onto the stack A, X, Y and the 'Flags' ... unfortunately the basic 6502 can only directly do A and the Flags - so we will have to Transfer X/Y to A first ... but the 65C0C can do it directly.

When it comes to setting the 'Stack pointer' we have to do it via the X register - Remember, the stack HAS to be between $0100 and $01FF on the 6502

Action	6502 command	65C02 Command	Action	6502 Command	6502 Command
Push A	PHA	PHA	Pull A	PLA	PLA
Push X	TXA PHA	PHX	Pull X	PLA TAX	PLX
Push Y	TYA PHA	PHY	Pull Y	PLA TAY	PLY
Push Flags	PHP	PHP	Pull Flags	PLP	PLP
Set SP to X	TXS	TXS	Set X to SP	TSX	TSX

Let's try out the stack!

We're going to set A,X and Y to various values, and push them onto the stack,

Because we can't do this directly for X and Y, we'll have to transfer them to A first

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Once we've done that, we'll show the contents of the stack...

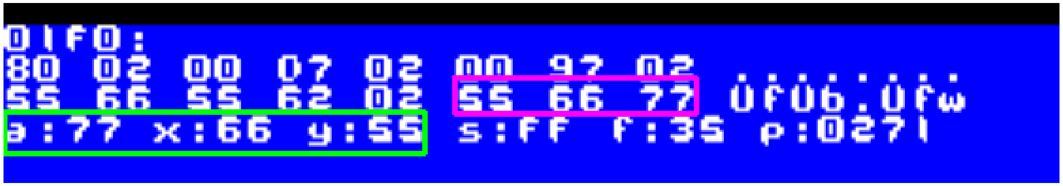
We'll then clear all the registers - and pull them from the stack - it's important we pull them in the same order!

Finally we'll show all the register contents

```
SEI                             ;Stop interrupts
jsr ScreenInit                  ;Init the graphics screen
jsr Cls                         ;Clear the screen

ldx #$FF                        ;Set Stack Pointer to $01FF
txs

lda #$77                        ;Set AXY to test values
ldx #$66
ldy #$55
pha                             ;Push A onto the stack
    txa                         ;Transfer X to A and push
    pha
        tya                     ;Transfer Y to A and push
        pha
            jsr MemDump         ;Show the Stack
            word $01F0          ;We should see pushed AXY
            byte $2

            lda #0              ;Clear XYA
            tax
            tay
        pla                     ;Pull A and move to Y
        tay
    pla                         ;Pull A and move to X
    tax
pla                             ;Pull A
jsr monitor                     ;Show Registers
jmp *                           ;Infinite Loop
```

We can see the **3 bytes at the top of the stack** - remember the stack pointer goes down with each push, so they are backwards

Provided we restore them in the correct order - **the registers are restored** - even though we cleared them before



```
01F0:
80 02 00 07 02 00 97 02
55 66 55 62 02 55 66 77 ÙfÙb:Ùfw
a:77 x:66 y:55 s:FF f:35 p:0271
```

[Ti-84 Plus CE (eZ80 cpu)](#)

The Stack and JSR

We can use the stack pointer to backup and restore register values ... the processor uses it too, to handle calling Subroutines!... lets take a look!

Subroutines are sections of code that will be executed, and then execution will resume after they complete
On the 6502 we call a sub with JSR (Jump SubRoutine).... and the last command of the sub is RTS (ReTurn from Subroutine)
if you're familiar with basic **JSR** is the equivalent of GOSUB... and **RTS** is the equivalent of RETURN

We're going to do a test here... we'll show the stack to the screen... first we'll push the flags onto the stack,

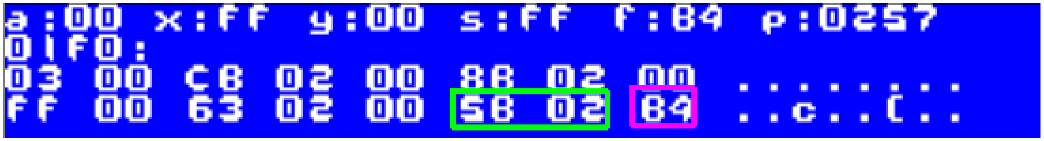
Then we're going to use JSR to jumpt to subroutine StackTest.... we'll show the stack again... and for reference, we'll also see the address of 'ReturnPos'

Then we'll return to the main program and show the stack again... what will happen?

```
        ldx #$FF                ;Set Stack Pointer to $01FF
        txs
        jsr monitor
        php                     ;Push flags onto the stack
            jsr SubTest
        plp                     ;Pull flags from the stack
        jmp *                   ;Infinite Loop

SubTest:
        pha
            jsr MemDump ;Show the Stack
            word $01F0  ;We should see pushed AXY
            byte $2
        pla
        rts
```

The flags are pushed onto the stack first... Next we can see the 'Return address' , that was pushed onto the stack by the JSR command

Effectively **JSR** pushes the program counter onto the stack, and **RTS** pulls the Program Counter off the stack



```
a:00  x:FF  y:00  s:FF  f:84  p:0257
01F0:
03 00 C8 02 00 88 02 00  ........
FF 00 63 02 00 58 02 84  ..c..[..
```

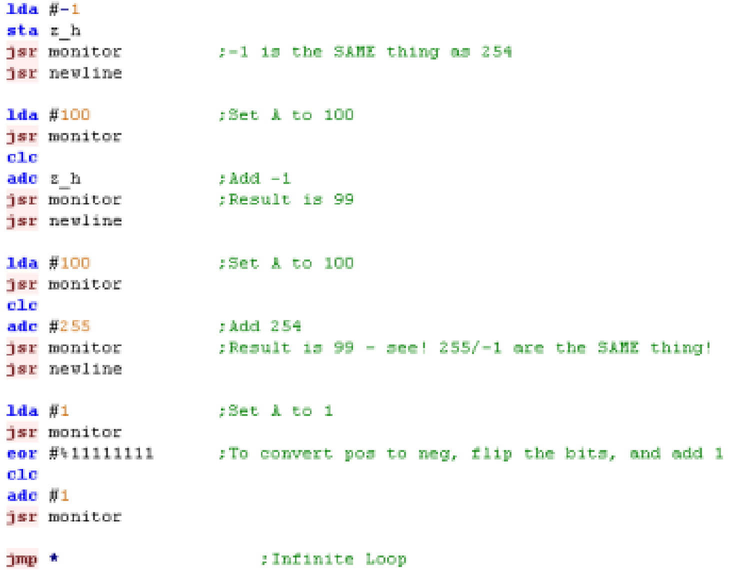
Because the JSR and RTS commands use the stack to maintain the program counter, it's important that the stack is the same when a subroutine ends as it was when it starterd... ne need to ensure we pull everything off the stack that we pushed on at the start... otherwise some 'other data' will be mistaken for the return address - and anything could happen!

Negative numbers in Assembly

Negative numbers in HEX are weird!... when we subtract 1 from 0 we get 255... this means 255 IS -1... in the same way, 254 is -2 and so on - meaning a 'Signed' byte can go from -128 to +127

The CPU doesn't 'Know ' whether it's working with signed or unsigned numbers - it all depends how we use the data...

The psuedocode for converting to positive to negative is to **invert all the bits**, **and add one**... or subtract the value from zero of course!

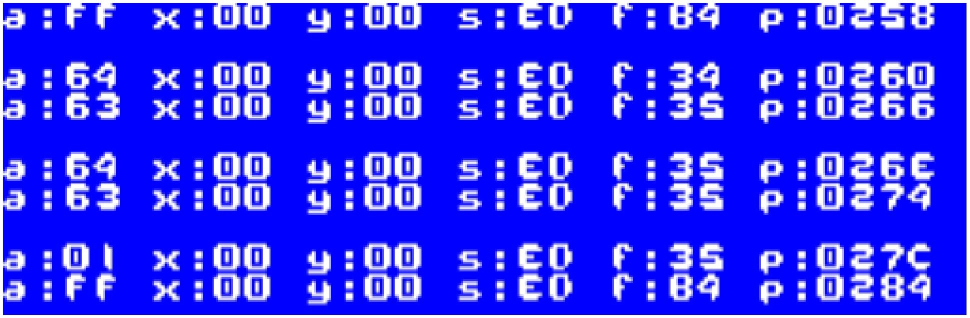


```
lda #-1
sta z_h
jsr monitor             ;-1 is the SAME thing as 254
jsr newline

lda #100                ;Set A to 100
jsr monitor
clc
adc z_h                 ;Add -1
jsr monitor             ;Result is 99
jsr newline

lda #100                ;Set A to 100
jsr monitor
clc
adc #255                ;Add 254
jsr monitor             ;Result is 99 - see! 255/-1 are the SAME thing!
jsr newline

lda #1                  ;Set A to 1
jsr monitor
eor #%11111111          ;To convert pos to neg, flip the bits, and add 1
clc
adc #1
jsr monitor

jmp *                       ;Infinite Loop
```

When we put a #-1 in the source, its converted to 255...

Because the numbers wrap around, adding 255 to a number decreases it by 1... so 255 IS -1

if we want to negate a number, we flip all the bits and add one... this converts 01 to $FF



```
a:FF  x:00  y:00  s:E0  f:84  p:0258

a:64  x:00  y:00  s:E0  f:34  p:0260
a:63  x:00  y:00  s:E0  f:35  p:0266

a:64  x:00  y:00  s:E0  f:35  p:026E
a:63  x:00  y:00  s:E0  f:35  p:0274

a:01  x:00  y:00  s:E0  f:35  p:027C
a:FF  x:00  y:00  s:E0  f:84  p:0284
```

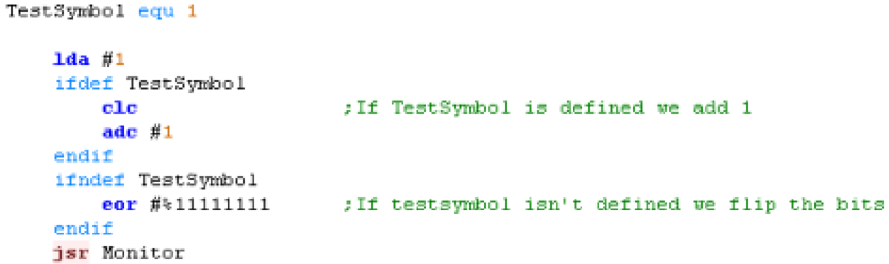
Conditional Assembly

We learned about using Labels for Jumps, and Symbols for values before... but symbols have another use!

We can put **IFDEF** statements in our code, and have parts of the assembly only compile if a symbol is defined - or not defined with **IFNDEF**

It's important to understand, it's not the CPU doing ths, the assembler simply skips over the excluded code - so it never appears in the outputted binary!

This allows us to build multiple versions of a program from a single source, in fact it's how these tutorials support so many systems!

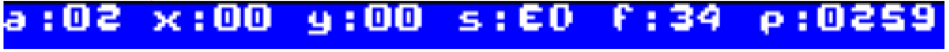


```
TestSymbol equ 1

    lda #1
    ifdef TestSymbol
        clc                     ;If TestSymbol is defined we add 1
        adc #1
    endif
    ifndef TestSymbol
        eor #%11111111          ;If testsymbol isn't defined we flip the bits
    endif
    jsr Monitor
```

To disable a definition we can just rem it out with a semicolon ; - we can even define symbols on the Vasm Command line!

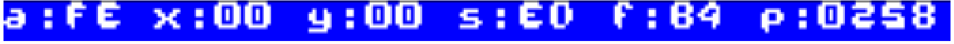
The output will of course be completely different depending on whether TestSymbol is defined or not.

With TestSymbol Defined



```
a:02 x:00 y:00 s:E0 f:34 p:0259
```

Without TestSymbol Defined



```
a:FE x:00 y:00 s:E0 f:B4 p:0258
```

Macros... for less typing!

Subroutines are great - but there's times they may be too slow (because of the JSR/RTS) .... and if you want to do things with the stack, they may not be possible.

Alternatively, we can use a **Macro**... this is a chunk of code that we can give a simple name... then whenever we use that name - the assembler will insert the code... we can even use parameters in the macro.

Because the assembler does the work, it's faster than a call, but saves us typing all the commands... however it will make the code larger - so you will want to call to subroutines for big chunks of code where you can rather than use macros.

```
macro PushPair,ra   ;Push a pair onto the stack (eg PushPair z_HL)
    lda \ra
    pha             ;Push lower Zpage entry
    lda \ra+1
    pha
endm                ;Push higher Zpage entry

macro PullPair,ra   ;Pull a pair onto the stack (eg PullPair z_HL)
    pla
    sta \ra+1       ;Pull lower Zpage entry
    pla
    sta \ra         ;Pull higher Zpage entry
endm


pushpair z_hl
```

16 bits.. When 8 Bits aren't enough!

Unlike the Z80, we don't have pairs of registers which we can use for 16 bit commands,

the easiest solution to this is to use concecutive bytes of the **Zero Page** as a pair to make up a 16 bit 'Zero Page Register'

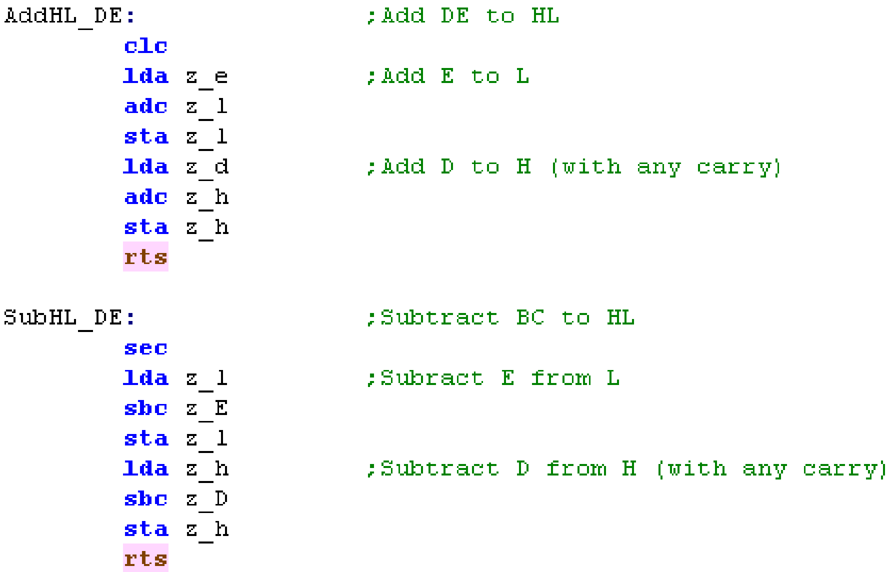
For ease of use, we'll use Symbols to define these with a name - and we'll mimic the Z80 register pairs... for example HL is High Low... but because the 6502 is little endian, L comes first in the zero page

```
z_Regs equ &0020

z_HL  equ z_Regs
z_L   equ z_Regs
z_H   equ z_Regs+1

z_BC  equ z_Regs+2
z_C   equ z_Regs+2
z_B   equ z_Regs+3

z_DE  equ z_Regs+4
z_E   equ z_Regs+4
z_D   equ z_Regs+5
```

When it comes to Addition or Subtraction - we use the Carry flag...

The Carry flag stores the 'overflow' of an addition, or the 'borrow' of a subtraction.

By using two **ADC** we can add 16 bit (or more) numbers, and two **SBC**'s can do a 16 bit subtract



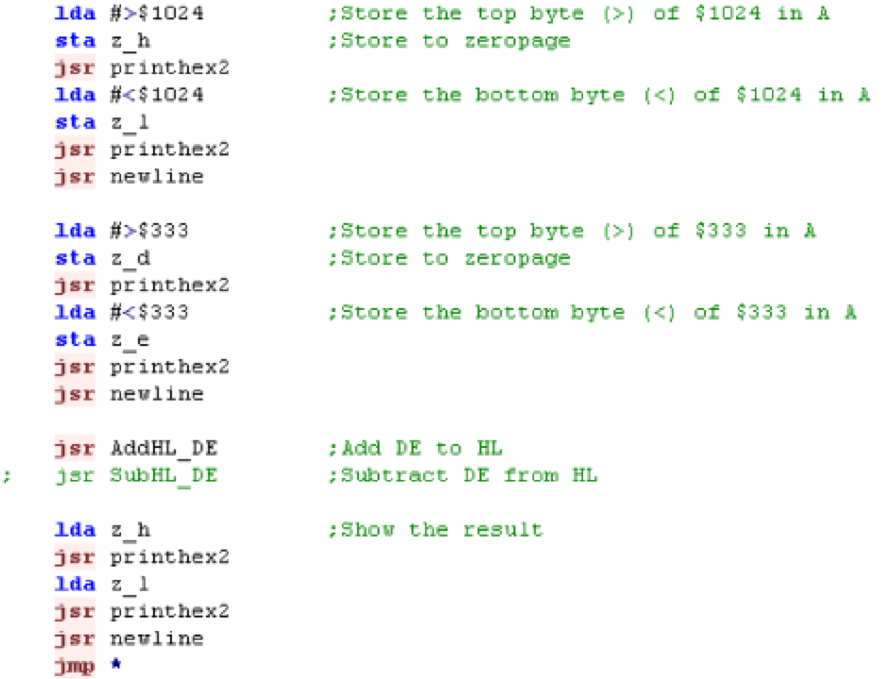
```
AddHL_DE:                        ;Add DE to HL
          clc
          lda z_e                ;Add E to L
          adc z_l
          sta z_l
          lda z_d                ;Add D to H (with any carry)
          adc z_h
          sta z_h
          rts

SubHL_DE:                        ;Subtract BC to HL
          sec
          lda z_l                ;Subract E from L
          sbc z_E
          sta z_l
          lda z_h                ;Subtract D from H (with any carry)
          sbc z_D
          sta z_h
          rts
```

When we want to use a 16 bit value, we have to split it into it's High byte, and it's Low byte

Forunately 6502 assemblers have us covered... we can use a > to calculate the high byte of a number, and < to calculate the low byte

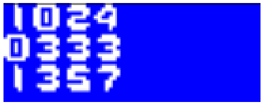
Once we've set 16 bit pairs Z_DE and Z_HL, we can call the addition or subtraction function

Note: many of the 'Printchar' functions use the same 'Z Page' values... so we're using a special 'PrintHex' function that backs them up.



```
    lda #>$1024          ;Store the top byte (>) of $1024 in A
    sta z_h              ;Store to zeropage
    jsr printhex2
    lda #<$1024          ;Store the bottom byte (<) of $1024 in A
    sta z_l
    jsr printhex2
    jsr newline

    lda #>$333           ;Store the top byte (>) of $333 in A
    sta z_d              ;Store to zeropage
    jsr printhex2
    lda #<$333           ;Store the bottom byte (<) of $333 in A
    sta z_e
    jsr printhex2
    jsr newline

    jsr AddHL_DE         ;Add DE to HL
;   jsr SubHL_DE         ;Subtract DE from HL

    lda z_h              ;Show the result
    jsr printhex2
    lda z_l
    jsr printhex2
    jsr newline
    jmp *
```

Addition:



```
1024
0333
1357
```

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

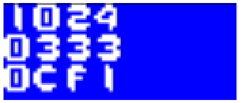
Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Subtraction:



```
1024
0333
0CF1
```

Ti-84 Plus CE (eZ80 cpu)

There's no needs to stop at 16 bits, you can just keep doing ADC's to get up to 32 bits or more...

Of course it will be slower!... another option is 'floating point'... but that's a too complex to cover here!

These tutorials use Zero page registers to mimic the function of Z80 registers where the 6502 can't directly do the job... this is because the author of these tutorials started on the Z80, and found that the most logical way to do things...

Other Tutorials may do things differenty, and if you don't like this way of using the Zero page, you should probably follow another tutorial instead.

Mult/Div... Where's my Maths!

The Z80 and 6502 have something in common... they have no Multiply or Divide commands... yes, you read that right!

We can, however simulate them!... the simplest way to multiply is repeately add a value, or subtract one to divide...

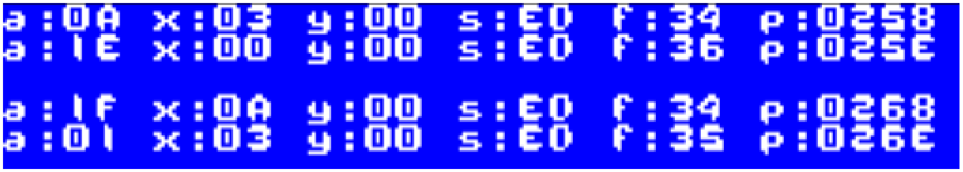
There are faster ways of doing things - and we'll look at them later!

In our Multiply example we'll multiply A by X, and store the result in A

In our Divide example we'll Divide A by X, and store the successfull divisions in X, and the remainder in A

```
    ldx #3
    lda #10
    jsr Monitor
    jsr Multiply                 ;Multiply 10 by 3
    jsr Monitor
    jsr newline

    ldx #10
    lda #31
    jsr Monitor
    jsr Divide                   ;Divide 31 by 10
    jsr Monitor

    jmp *                        ;Infinite Loop
Multiply:
    sta z_h                      ;Value to multiply by
    lda #0
MultiplyAgain:
    clc
    adc z_h                      ;add again
    dex                          ;Decrease counter
    bne MultiplyAgain
    rts

Divide:
    stx z_h                      ;divisor
    ldx #0                       ;Set count to zero
DivideAgain:
    sec
    sbc z_h                      ;Subtract one of divisor
    bcc DivideDone               ;Have we gone below zero?
    inx                          ;Add 1 to count of sucessfull subs
    jmp DivideAgain
DivideDone:
    clc
    adc z_h                      ;We've gone below zero - so fix that!
    rts
```

You can see we've effected a simple Multiply and Divide command!



```
a:0A x:03 y:00 s:E0 f:34 p:0258
a:1E x:00 y:00 s:E0 f:36 p:025E

a:1F x:0A y:00 s:E0 f:34 p:0268
a:01 x:03 y:00 s:E0 f:35 p:026E
```

Lesson 5 - Bits and Shifts

We've learned lots of maths commands, but we've still not covered the full range... this time lets take a look at how we can work with Bits on the 6502!

Get The DevTools!

File Available in sources.7z Click to Download

Discuss on the forums! Under Construction

Video Available Click to watch!

AND, OR and EOR!

There will be many times when we need to change some of the bits in a register, we have a range of commands to do this!

AND will return a bit as 1 where the bits of both the accumulator and parameter are 1
OR will set a bit to 1 where the bit of either the accumulator or the parameter is 1
EOR is nothing to do with donkeys... it means Exclusive OR... it will invert the bits of the accumulator with the parameter - it's called XOR on the z80!

Effectively, when a bit is 1 - AND will keep it... OR will set it, and EOR will invert it

A summary of each command can be seen below:

Command	Accumulator	Parameter	Result
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0
ORA	1	1	1
	0	1	1
	1	0	1
	0	0	0
EOR	1	1	0
	0	1	1
	1	0	1
	0	0	0

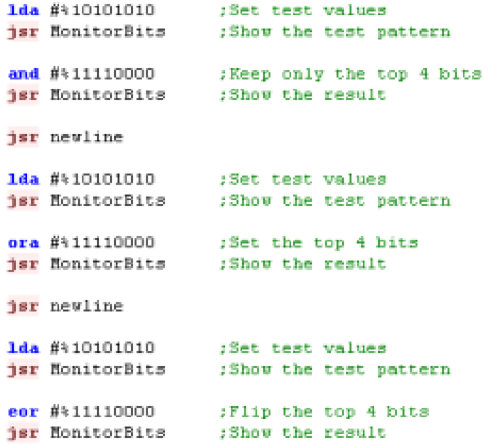
Command	lda #%10101010 **eor** #%11110000	lda #%10101010 **and** #%11110000	lda #%10101010 **ora** #%11110000
Result	#%**0101**1010	#%1010**0000**	#%**1111**1010
Meaning	Invert the bits where the mask bits are 1	return 1 where both bits are1	Return 1 when either bit is 1

In the Z80 tutorials, we saw a visual representation of how these commands changed the bits - it may help you understand each command.

Sample	**EOR %11110000** Invert Bits that are 1	**AND %11110000** Keep Bits that are 1	**ORA %11110000** Set Bits that are 1

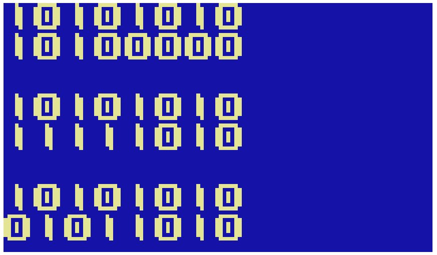
Lets try these commands on the 6502!

We'll use a test bit pattern, and try each command with the same %11110000 parameter, We're using a 'MontiorBits' function, which will show the contents of the Accumulators bits to screen!



```
lda #%10101010          ;Set test values
jsr MonitorBits         ;Show the test pattern

and #%11110000          ;Keep only the top 4 bits
jsr MonitorBits         ;Show the result

jsr newline

lda #%10101010          ;Set test values
jsr MonitorBits         ;Show the test pattern

ora #%11110000          ;Set the top 4 bits
jsr MonitorBits         ;Show the result

jsr newline

lda #%10101010          ;Set test values
jsr MonitorBits         ;Show the test pattern

eor #%11110000          ;Flip the top 4 bits
jsr MonitorBits         ;Show the result
```

The bits of the test pattern will be altered in each case according to the logical command!



```
10101010
10100000

10101010
11111010

10101010
01011010
```

Rotating and shifting bits with ROL,ROR, ASL and LSR

There will be many times when we want to shift bits around... If we shift all the bits in a byte left, we'll effectively double the number - if we shift them right, we'll halve it
We may want to use 3 bits from the middle of a byte or word as a 'lookup' - and we'll need to get them in the right position...

You may not immediately see the need for bit shifting - but as you program, you'll come across many times you need to do it...

One very important use of ASL/LSR is for halving and doubling numbers... our CPU has no **multiply or divide** commands, but effectively it can quickly do x2 or /2... and you want to try to take advantage of this when designing your code!

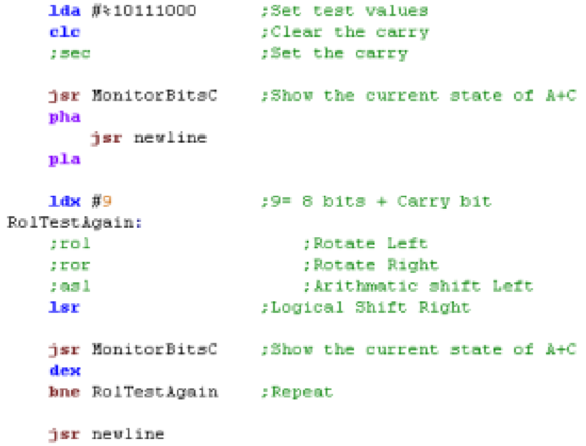
The 6502 has 2 options - shift a bits within the Accumulator using ASL or LSR - which will fill any new bits with 0 and lose any bits pushed out of the accumulator,
or 'Rotate it through the carry flag' with ROL and ROR... where the carry is put into the new bit, and any bits pushed out go into the carry flag

Command	**Left**	**Right**
ROtate	ROL	ROR
Arithmatic Shift / Logical Shift	ASL	LSR

We're going to test the shifting commands... we'll use a new testing function 'MonitorBitsC' will show the Accumulator and Carry flag.

We'll set the accumulator to %10111000, and we'll clear the carry flag...

Then we'll see what happens when we use each of the rotate commands 9 times!



```
        lda #%10111000          ;Set test values
        clc                     ;Clear the carry
        ;sec                    ;Set the carry

        jsr MonitorBitsC        ;Show the current state of A+C
        pha
            jsr newline
        pla

        ldx #9                  ;9= 8 bits + Carry bit
    RolTestAgain:
        ;rol                        ;Rotate Left
        ;ror                        ;Rotate Right
        ;asl                        ;Arithmatic shift Left
        lsr                     ;Logical Shift Right

        jsr MonitorBitsC        ;Show the current state of A+C
        dex
        bne RolTestAgain        ;Repeat

        jsr newline
```

So what does each command do?

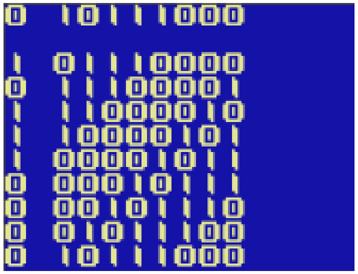
Well ROL rotates all the bits Left, the carry ends up in Bit 0 - and what WAS in Bit 7 ends up in the carry.

ROR is the opposite... it rotates all the bits Right, the carry ends up in Bit 7 - and what WAS in Bit 0 ends up in the carry.

ASL shifts all the bits left - but Bit 0 is zero - and the what was in Bit 7 is lost

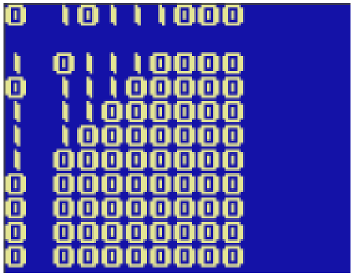
LSR is the opposite, it shifts all the bits right - but Bit 7 is zero - and the what was in Bit 0 is lost

ROL:



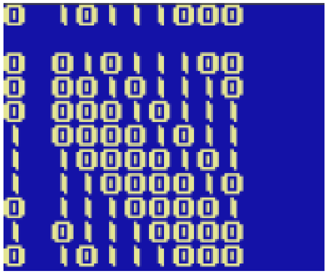
```
0  10111000

1  01110000
0  11100001
1  11000010
1  10000101
1  00001011
0  00010111
0  00101110
0  01011100
0  10111000
```

ASL:



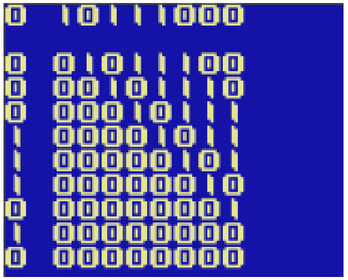
```
0  10111000

1  01110000
0  11100000
1  11000000
1  10000000
1  00000000
0  00000000
0  00000000
0  00000000
0  00000000
```

ROR:



```
0  10111000

0  01011100
0  00101110
0  00010111
1  00001011
1  10000101
1  11000010
0  11100001
1  01110000
0  10111000
```

LSR:



```
0  10111000

0  01011100
0  00101110
0  00010111
1  00001011
1  00000101
1  00000010
0  00000001
1  00000000
0  00000000
```

The 6502 doesn't have as many bit shift options as the Z80... but we can 'fake' others!.

If we want to shift 1's into the empty bits we can just set the carry with SEC before the rotate command,

If we want to rotate the 8 bits in the accumulator without the carry... we can back up A with PHA, do the rotate, then restore A with PLA, and do another rotate

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

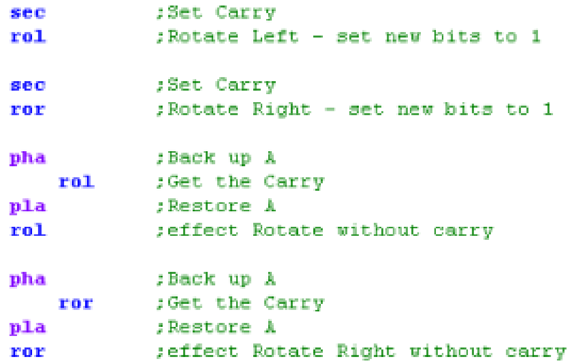
Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Ti-84 Plus CE (eZ80 cpu)

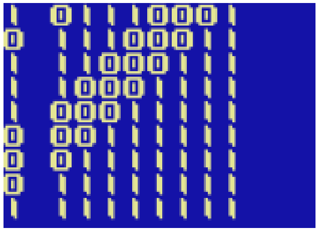


```
sec             ;Set Carry
rol             ;Rotate Left - set new bits to 1

sec             ;Set Carry
ror             ;Rotate Right - set new bits to 1

pha             ;Back up A
    rol         ;Get the Carry
pla             ;Restore A
rol             ;effect Rotate without carry

pha             ;Back up A
    ror         ;Get the Carry
pla             ;Restore A
ror             ;effect Rotate Right without carry
```

Now we're able to set the new bits to a 1, or able to rotate the bits within A

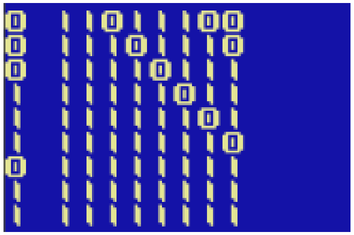
There's other ways to do this, and other combinations of commands to do things like swap nibbles... see here

SEC -ROL



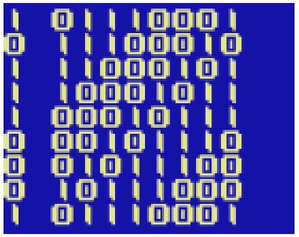
```
1 01110001
0 11100011
1 11000111
1 10001111
1 00011111
0 00111111
0 01111111
0 11111111
1 11111111
```

SEC-ROR



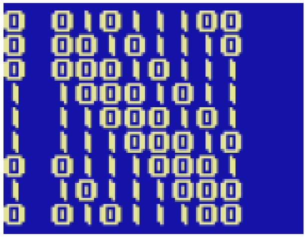
```
0 11011100
0 11101110
0 11110111
1 11111011
1 11111101
1 11111110
0 11111111
1 11111111
1 11111111
```

PHA-ROL-PLA-ROL



```
1 01110001
0 11100010
1 11000101
1 10001011
1 00010111
0 00101110
0 01011100
0 10111000
1 01110001
```

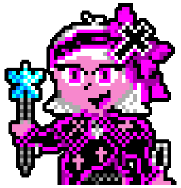
PHA-ROR-PLA-ROR



```
0 01011100
0 00101110
0 00010111
1 10001011
1 11000101
1 11100010
0 01110001
1 10111000
0 01011100
```

There's lots of commands we'd like to have that are 'missing' on the 6502 - and this is just one possible solution

See Here for more examples of combinations of commands to effect the result you want.



Bit testing

There will be many times when we want to test a single bit of a register, and make a decision based on it's content....
We could use the AND command, but that will change the accumulator - and we may want it to stay the same... for this we have the BIT command

BIT has the same effect as AND on the Z flag - but doesn't change the Accumulator... unlike AND, we have to use a memory address as the parameter... so we'll define a set of bitmasks...

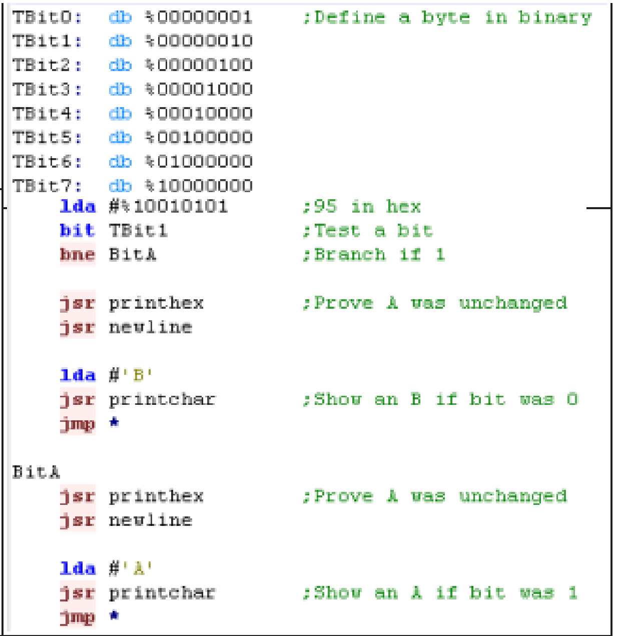
Because the BIT command needs to work with an address, we need to define some bitmasks...	

To define a byte of data in our program code we use DB - then we specify the value for the byte... we're using % and defining the definitions in bits

We're giving each of these a label, so we can use them easilly later.

We can use the BIT command with a label pointing to one of these defined bytes, and then use BNE or BEQ to branch depending on if the bit was Zero or not...

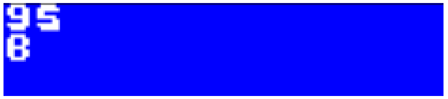
Note, the Accumulator is unchanged when we do this



```
TBit0:  db %00000001          ;Define a byte in binary
TBit1:  db %00000010
TBit2:  db %00000100
TBit3:  db %00001000
TBit4:  db %00010000
TBit5:  db %00100000
TBit6:  db %01000000
TBit7:  db %10000000
    lda #%10010101            ;95 in hex
    bit TBit1                 ;Test a bit
    bne BitA                  ;Branch if 1

    jsr printhex              ;Prove A was unchanged
    jsr newline

    lda #'B'
    jsr printchar             ;Show an B if bit was 0
    jmp *

BitA
    jsr printhex              ;Prove A was unchanged
    jsr newline

    lda #'A'
    jsr printchar             ;Show an A if bit was 1
    jmp *
```

We'll branch and show a B if the bit is Zero... or an A if the bit is One

Hint: Try changing the TBit1 to a TBit0 in the example code!



```
95
B
```

Specifying Addresses in this way will use 3 bytes per command - which is wasteful - if possible, it would be better to store these bitmasks in the Zero page, so we only use 2 bytes per command if we can.

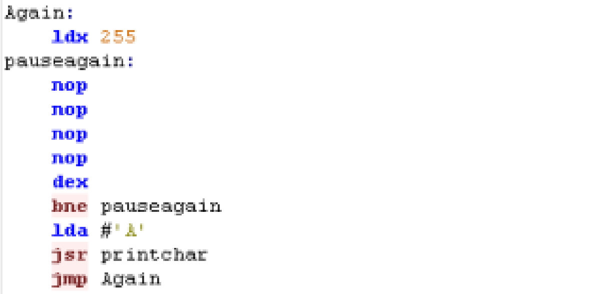
Whatever bit you test, two other flags are set at the same time....as well as the Z flag being set to the tested bit, N flag is set to bit 7 , and the V flag is set to bit 6

So you can branch on conditions relating to bit 7 and 6 without any more testing commands!

NOP - Slacking in 8 bits!

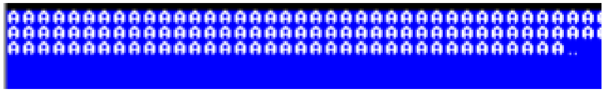
NOP (No OPeration) is a strange command... it does absolutely nothing!

Why would we want to use it? well it's handy for a short delay - and if we do something called 'Self Modifying code' (code that rewrites itself) it can be useful for disabling commands



```
Again:
    ldx 255
pauseagain:
    nop
    nop
    nop
    nop
    dex
    bne pauseagain
    lda #'A'
    jsr printchar
    jmp Again
```

The more NOPs we add, the slower the screen will fill



```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA..
```

Lots of NOP commands aren't really a good way of slowing things down - It's far better to nest loops to slow things down or use some kind of firmware function...

NOP's are more useful for self modifying code - we'll learn about that next time!

Lesson 6 - Defined data, Aligned data... Lookup Tables, Vector Tables, and Self-modifying code!

Now we've learned all the basic maths commands, it's time to start looking at some clever tricks!

Get The DevTools!

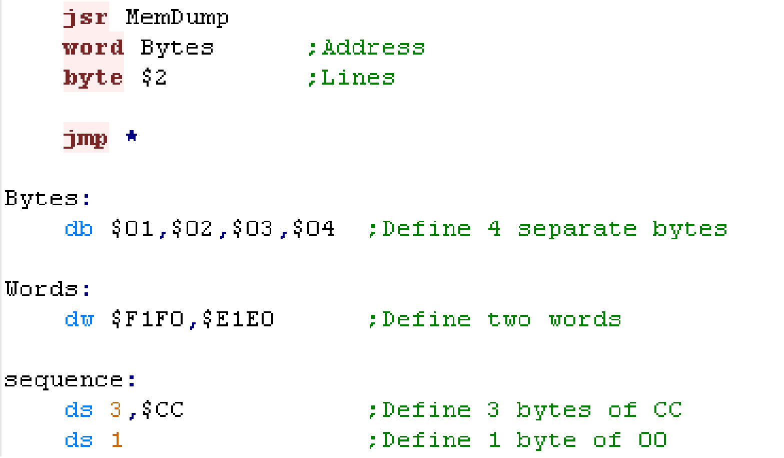
File Available in sources.7z Click to Download

Video Available Click to watch!

Defining Data with DB DW and DS

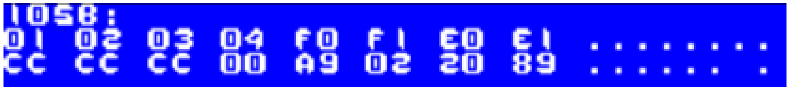
There will be times we need to define data for use within our code areas... we can use three commands to do this...

DB will define one or more bytes
DW will define one or more words (in little endian)
DS will define sequences of defined length in bytes - if only one parameter is specified, then all the bytes are zero, if two are specified they will all be the specified value



```
    jsr MemDump
    word Bytes      ;Address
    byte $2         ;Lines

    jmp *

Bytes:
    db $01,$02,$03,$04  ;Define 4 separate bytes

Words:
    dw $F1F0,$E1E0      ;Define two words

sequence:
    ds 3,$CC            ;Define 3 bytes of CC
    ds 1                ;Define 1 byte of 00
```

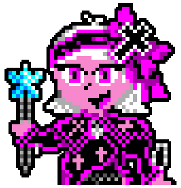
The contents of the defined bytes will be shown... notice that the bytes with DW are backwards, because



```
1058:
01 02 03 04 F0 F1 E0 E1 ........
CC CC CC 00 A9 02 20 89 ....... .
```

DB, DW and DS are assembler commands not 6502 opcodes... they will work in VASM and other assemblers, but depending on your assembler the commands may be different.

Check your documentation if the commands do not work as you expect!



Lookup Tables

A Lookup table is just a set of data for some purpose, we can lookup a numbered entry and use the result for some purpose...

For Example, if we want to draw a sine wave, but don't want to try to calculate a sine wave, we can just read the needed values from a 'Lookup Table'

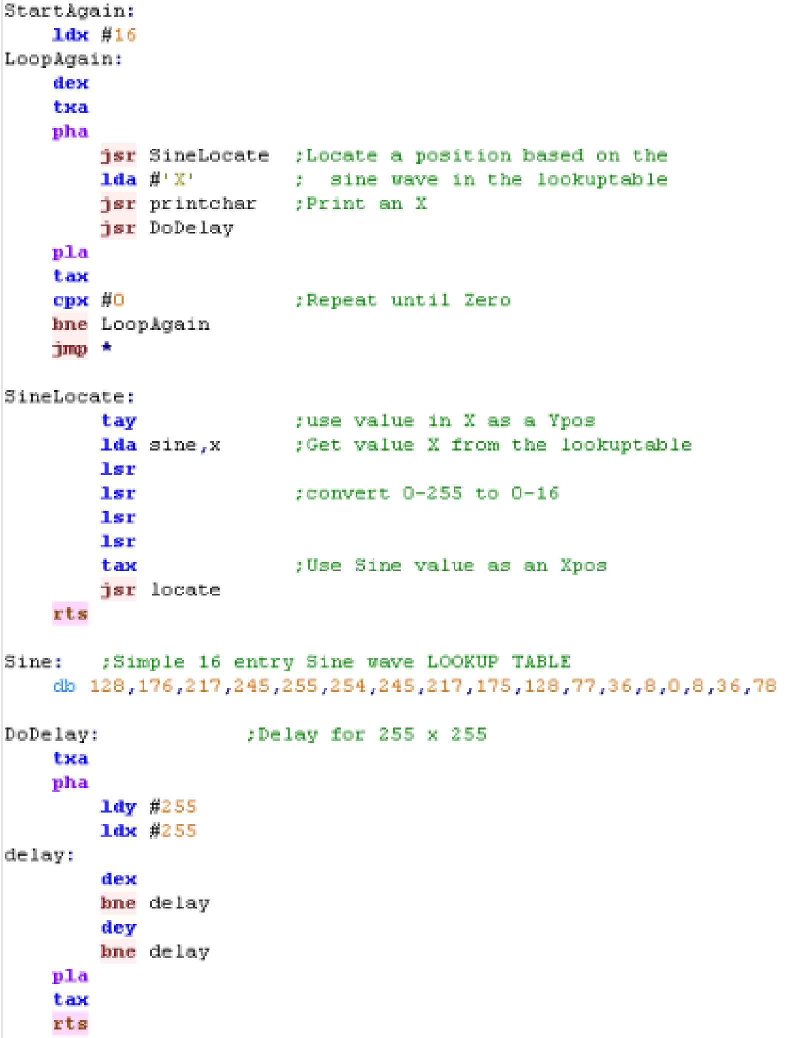
```
Sine:    ;Simple 16 entry Sine wave LOOKUP TABLE
    db 128,176,217,245,255,254,245,217,175,128,77,36,8,0,8,36,78
```

We're going to use this lookup table to set an X position, and repeatedly decrement the Y - so we can draw a sinewave in X'es

The 6502's Indexed addressing mode is perfect for this kind of work!

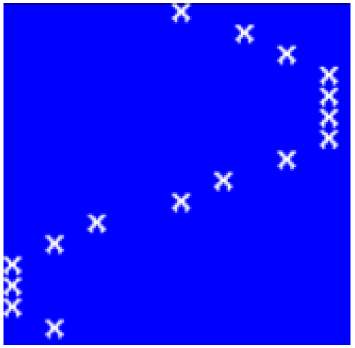
We LDA sine,X to read in entry X from the sine lookup table!

Note... the Lookuptable has values 0-255 - we need to scale it down by dividing it by 16 - we do that with 4 LSR's



```
StartAgain:
    ldx #16
LoopAgain:
    dex
    txa
    pha
        jsr SineLocate  ;Locate a position based on the
        lda #'X'        ;  sine wave in the lookuptable
        jsr printchar   ;Print an X
        jsr DoDelay
    pla
    tax
    cpx #0              ;Repeat until Zero
    bne LoopAgain
    jmp *

SineLocate:
        tay             ;use value in X as a Ypos
        lda sine,x      ;Get value X from the lookuptable
        lsr
        lsr             ;convert 0-255 to 0-16
        lsr
        lsr
        tax             ;Use Sine value as an Xpos
        jsr locate
    rts

Sine:   ;Simple 16 entry Sine wave LOOKUP TABLE
    db 128,176,217,245,255,254,245,217,175,128,77,36,8,0,8,36,78

DoDelay:            ;Delay for 255 x 255
    txa
    pha
        ldy #255
        ldx #255
delay:
        dex
        bne delay
        dey
        bne delay
    pla
    tax
    rts
```

Our sine wave will be shown to screen... it's not very high resolution, but we could add extra steps if we wanted.



The entries in a lookup table don't have to just be 1 byte it can be as many bytes as you want - though if

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

you use X to read in the entries ... your total lookup table has to be 256 bytes in total, so if each entry is 4 bytes (2 words), then the Lookuptable can only have 64 entries!

You can always calculate the address to read from manually rather than using X if you need more

Ti-84 Plus CE (eZ80 cpu)

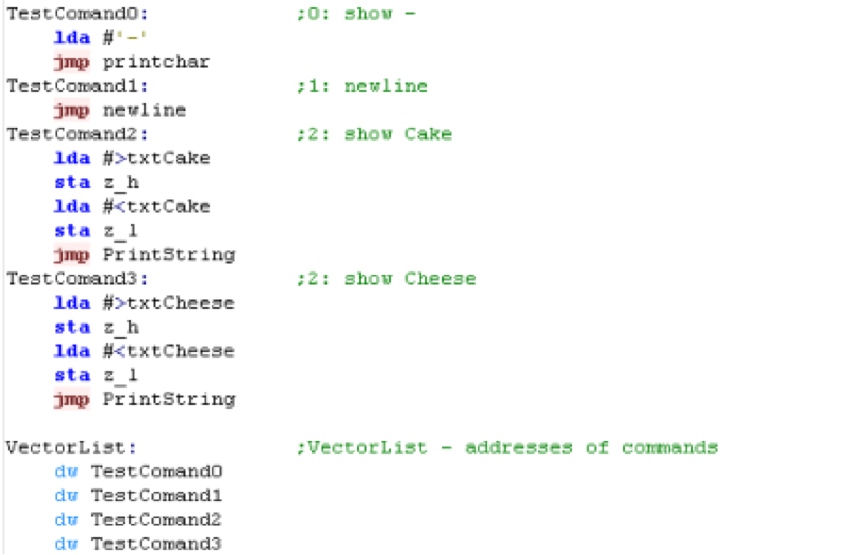
Vector Tables

One special kind of Lookup Table is sometimes called a 'Vector table'...

This is a table of 16 bit words... each of which is an address... we use our lookup table code to read in an address - then execute the data at that address!

Effectively, this allows us to execute commands based on single byte 'command numbers'... this can save memory if we need

In this example, we'll define 4 silly commands to try out - they'll just show simple text to screen

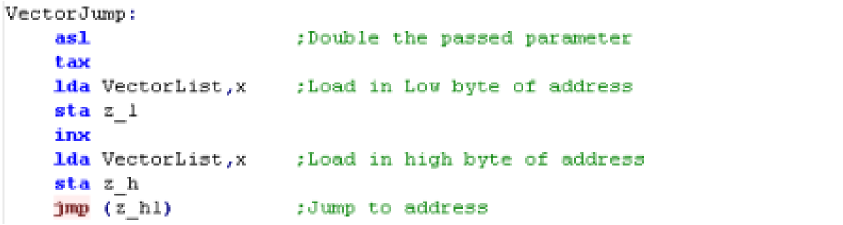


```
TestComand0:                ;0: show -
    lda #'-'
    jmp printchar
TestComand1:                ;1: newline
    jmp newline
TestComand2:                ;2: show Cake
    lda #>txtCake
    sta z_h
    lda #<txtCake
    sta z_l
    jmp PrintString
TestComand3:                ;2: show Cheese
    lda #>txtCheese
    sta z_h
    lda #<txtCheese
    sta z_l
    jmp PrintString

VectorList:                 ;VectorList - addresses of commands
    dw TestComand0
    dw TestComand1
    dw TestComand2
    dw TestComand3
```

We need to define a function to execute a numbered command from this list .

We'll take a number in via the Accumulator - double it with ASL, and load a pair of bytes from that offset in the Vector Table...

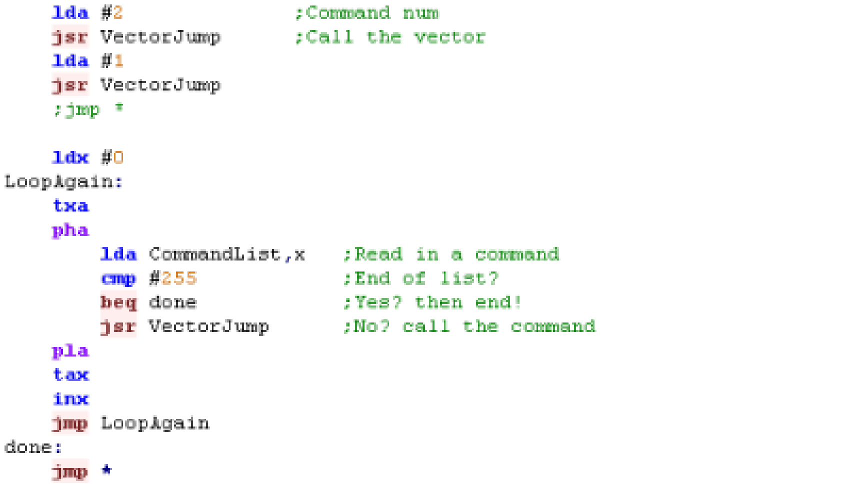
The address we got will be where we want to go, so we'll use it with an indirect jump via JMP (Z_HL)



```
VectorJump:
    asl                     ;Double the passed parameter
    tax
    lda VectorList,x        ;Load in Low byte of address
    sta z_l
    inx
    lda VectorList,x        ;Load in high byte of address
    sta z_h
    jmp (z_hl)              ;Jump to address
```

We can call our 'VectorJump' command just by passing a value in A,

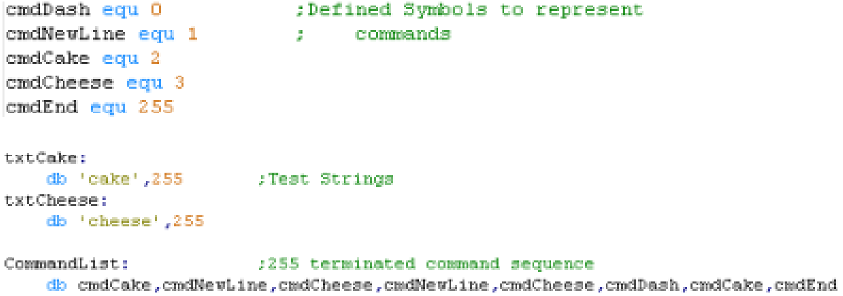
But if we want to be really powerful, we can process a 'CommandList'... with a set of numbered commands!



```
    lda #2              ;Command num
    jsr VectorJump      ;Call the vector
    lda #1
    jsr VectorJump
    ;jmp *

    ldx #0
LoopAgain:
    txa
    pha
        lda CommandList,x   ;Read in a command
        cmp #255            ;End of list?
        beq done            ;Yes? then end!
        jsr VectorJump      ;No? call the command
    pla
    tax
    inx
    jmp LoopAgain
done:
    jmp *
```

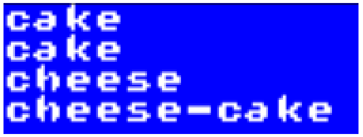
We'll need to define this command list, and also a few strings...

If we want, we can use Symbols defined with EQU to give 'names' to these numbered commands!



```
cmdDash equ 0           ;Defined Symbols to represent
cmdNewLine equ 1        ;    commands
cmdCake equ 2
cmdCheese equ 3
cmdEnd equ 255

txtCake:
    db 'cake',255       ;Test Strings
txtCheese:
    db 'cheese',255

CommandList:            ;255 terminated command sequence
    db cmdCake,cmdNewLine,cmdCheese,cmdNewLine,cmdCheese,cmdDash,cmdCake,cmdEnd
```

The result of the calls at the start, and the command list are shown here... you can try changing the command numbers and see the results



```
cake
cake
cheese
cheese-cake
```

Vector tables have AWESOME POWER! They allow us to turn a number into a executed command - in this case we've effectively created a scripting language!... because each command is just one byte... we could have hundreds of calls and save lots of space compared to sets of JSR's!

Aligned code and Self Modification

Self Modifying code is where our program overwrites parts of itself... why would we want to do this? well rather than a condition and a branch, there may be times where we can just reprogram a jump - and rather than loading A from a memory address, we could just reprogram a LDA command...

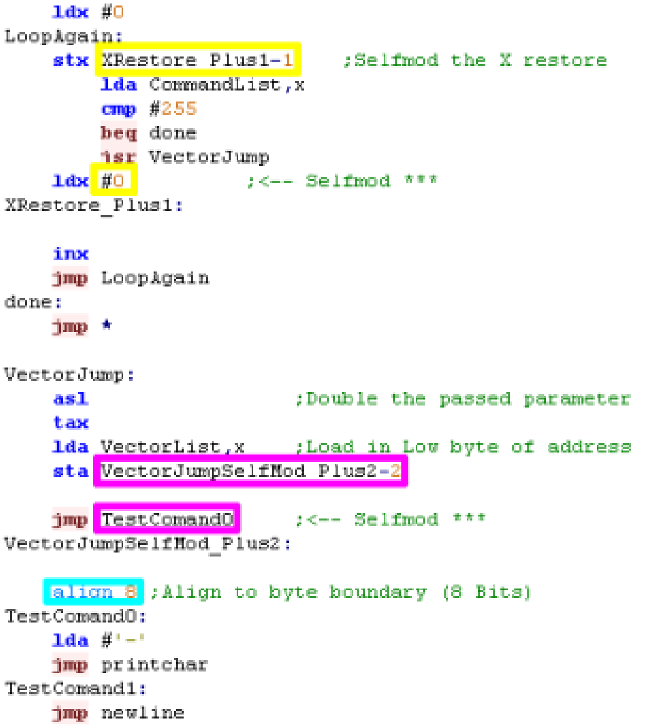
The reasons we may want to do this are twofold - saving speed, and saving bytes (though saving bytes will also usually save speed!)

This routine has two pieces of self modifying code... rather than PHA/PLA and TXA/TAX - we'll use self modifying code to restore X by **replacing the byte at the end of LDX with the correct value**

Also we'll self modify the **last byte of a Jump to cause the Vector jump** - this is much simpler than the indirect jump we used before, but **relies on all the addresses of the @ to have the same top byte**

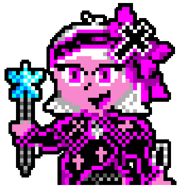
How can we makes sure all the commands have the same top byte? well we need to pad our code with 0000's until a new byte starts (for example $1200 or $1300)

With VASM - the **Align** command takes a parameter which is a number of bits to align by - for example ALIGN 2 will align to a 32 bit boundary - and **ALIGN 8** will do what we need - and align to a byte boundary - note, this command will be different on other assemblers.



```
    ldx #0
LoopAgain:
    stx XRestore Plus1-1      ;Selfmod the X restore
        lda CommandList,x
        cmp #255
        beq done
        jsr VectorJump
    ldx #0                ;<-- Selfmod ***
XRestore_Plus1:

    inx
    jmp LoopAgain
done:
    jmp *

VectorJump:
    asl                     ;Double the passed parameter
    tax
    lda VectorList,x        ;Load in Low byte of address
    sta VectorJumpSelfMod Plus2-2

    jmp TestComand0         ;<-- Selfmod ***
VectorJumpSelfMod_Plus2:

   align 8 ;Align to byte boundary (8 Bits)
TestComand0:
    lda #'-'
    jmp printchar
TestComand1:
    jmp newline
```

Self Modifying code allows for extra speed and saves memory - but it's complex and only works from RAM - so if your program is running in ROM it won't work.



We can use vector tables to create 'modules' of code and execute them with a single call - with a 'parameter' which defines the command number - The calling code doesn't need to know the internals, so long as each numbered command does the same job it will work fine... this allows you to have different loadable modules, and the internals can change so long as the base call and functions of each numbered command does not.

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Ti-84 Plus CE (eZ80 cpu)

Learn Multiplatform Assembly Programming
with ChibiAkumas!
Order now at
amazon.com

Want to help support
my content creation?
BECOME A PATRON

Buy ChibiAkumas
merchandise from
Teespring &
Support my content

Recent New Content

Amiga - ASM PSET and POINT for Pixel Plotting

Learn 65816 Assembly: 8 and 16 bit modes on the 65816

SNES - ASM PSET and POINT for Pixel Plotting

ARM Assembly Lesson H3

Lesson P65 - Mouse reading on the Sam Coupe

Mouse Reading in MS-DOS

Risc-V Assembly Lesson 3 - Bit ops and more maths!

Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and POINT for Pixel Plotting

Apple 2 - ASM PSET and POINT for Pixel Plotting

Making a 6502 ASM Tron game... Photon1 - Introduction and Data Structures

Gaming + more:

Emily The Strange (DS) - Live full playthrough

$150 calculator: Unboxing the

Ti-84 Plus CE (eZ80 cpu)