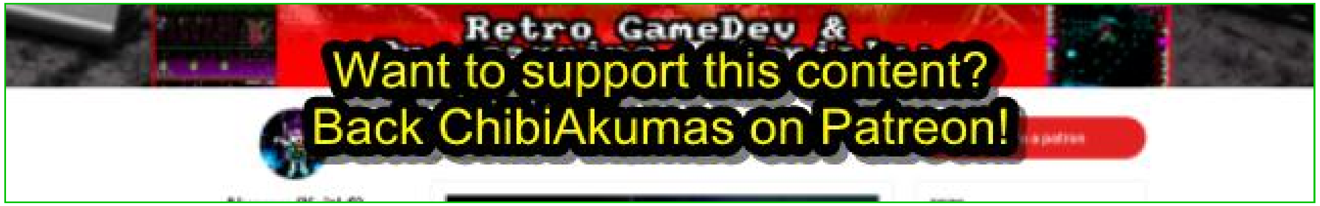
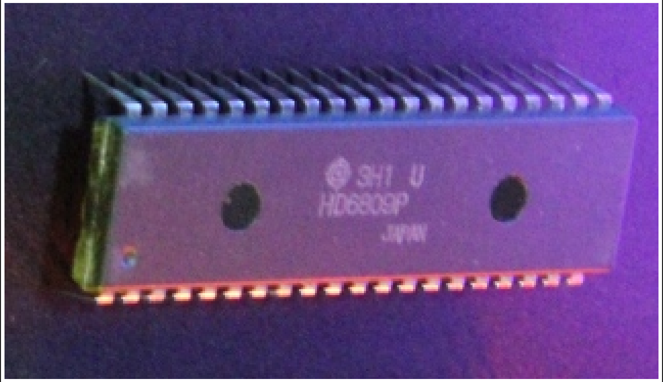


Learn Multi platform 6809 Assembly Programming... 8 bit resurrection!

The 6800 was too expensive for the mainstream, and it had many of its features cut, and was released as the 6502... it's second accumulator gone, it's command set cut back - and everyone forgot about the 6800.... But the 6800 came back - as the 6809... with new previously unheard-of powers!... armed with twin stack pointers , 16 bit Stack,X and Y registers- 16 bit capabilities and advanced addressing modes and even a MULTiply command (unheard of in most 8 bits)... the 6809 is the 'missing link' between the 6502 and the 68000! Powering the Dragon 32, the FM-7 machines - and the unique Vectrex... Lets see what the 6809 can do!	

Platforms Covered in these tutorials

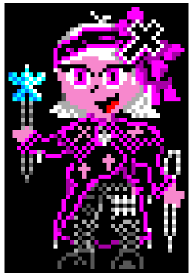
View Options
Default Dark
Simple (Hide this menu)
Print Mode (white background)

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

If you want to learn 6809 get the Cheatsheet! it has all the 6809 commands, It will give you a quick reference when you're stuck or confused, which will probably happen a lot in the early days!

There's also a enhanced 6309 cheatsheet if you're can use the extra opcodes!



We'll be using Macroassembler AS for our assembly in these tutorials... VASM is an assembler which supports rarer CPU's like 6809 and 65816 and many more, and also supports multiple syntax schemes...

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

What is the 6809 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 6809 is an 8-Bit processor with a 16 bit Address bus!... it has two 8 bit accumulatrors, A and B, that can be combined to make up one 16 bit accumulator D (AB)
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 four bytes is 1024 bytes
64 kilobytes is the amount of memory a basic 8-bit system can access

the 6809 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

Z80 Platforms
Amstrad CPC
Elan Enterprise
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

basement dwelling losers who wrote them!!!
6809 code is small, fast, and super efficient - with ASM you can do things in 1k that will amaze you!

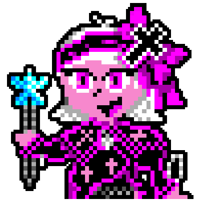
Numbers in Assembly can be represented in different ways.
A 'Nibble' (half a byte) can be represented as Binary (0000-1111) , Decimal (0-15) or Hexadecimal (0-F)...
unfortunately, you'll need to learn all three for programming!

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, some use $FF or #FF or even 0x, but this guide uses & - as this is how hexadecimal is represented in CPC basic
All the code in this tutorial is designed for compiling with WinApe's assembler - if you're using something else you may need to change a few things!
But remember, whatever compiler you use, while the text based source code may need to be slightly different, the compiled "BYTES' will be the same!



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** !							

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

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 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 6809 Registers

Compared to the 6502, , the 6809 is seriously powerful - and even gives the Z80 something to think about!

	8 Bit	16 Bit	Use cases
accumulator **A**	**A**		8 Bit Accumulator
accumulator **B**	**B**		8 Bit Accumulator
16-Bit Accumulator **D**	**A**	**B**	A+B combined to make a 16 bit accumulator

Flags: EFHINZVC

	Name	Meaning
E	Entire Flag	Regular/Fast Interrupt flag
F	FIRQ Mask	Fast Interrupt Flag

ARM Platforms
Gameboy Advance
Nintendo DS
Risc Os

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

Condition Code Register	CCR	Flags
Indirect X	X	Indirect Register
Indirect Y	Y	Indirect Register
User Stack Pointer	U	User Stack
Hardware Stack Pointer	S	Stack
Program Counter	PC	Running Command
Direct Page	DP	Zero page is relocatable on 6809

H	Half Carry	Bit 3/4 carry for BCD
I	IRQ Mask	Interrupt Flag
N	Negative	
Z	Zero	
V	oVerflow	
C	Carry	

Fujitsu FM7
TRS-80 Coco 3
Vectrex

My Game projects
Chibi Aliens
Chibi Akumas

Work in Progress
Learn 65816 Assembly
Learn eZ80 Assembly

Misc bits
Ruby programming

The Direct page is like the 6502 Zero Page, however it does not need to be at zero!

We can load A with a value, then TFR A,DP to set the direct page... we need to tell the assembler where the direct page is, otherwise some commands may malfunction, we do this with ASSUME dpr:$xx - this is called SETDP on some assemblers

Like the 68000, the 6809 is **BIG ENDIAN**... this means a 16 bit pair stored to an address like $6000 will save the high byte to $6000, and the low byte to $6001

Special Memory addresses on the 6809

Unlike the 6502, The 6809 has full 16 bit Stack pointers, U and S.... the 'Zero Page' (AKA Direct Page) can also be re positioned

Like the 6502, there are a variety of 'Interrupt Vectors' with fixed addresses...

Address	Vector (Address)	Registers Auto-pushed onto stack
$FFF2	SWi 3 Vector	D,X,Y,U,DP,CC
$FFF4	SWI 2 Vector	D,X,Y,U,DP,CC
$FFF6	FIRQ Vector	CC (E flag cleared)
$FFF8	IRQ Vector	D,X,Y,U,DP,CC
$FFFA	SWI 1 Vector	D,X,Y,U,DP,CC
$FFFC	NMI Vector	D,X,Y,U,DP,CC
$FFFE	RESET Vector	NA

The 6809 Addressing Modes

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

Inherent Addressing	Commands that don't take a parameter	ABX
Register Addressing	Commands that only use register	TFR A,DP
Immediate Addressing	Direct Address of command	ADDA #$10 ADDD #$1000
Direct Page addressing	Read from DP (zero page)	ADDA $10
Extended Direct addressing	Read from an address	ADDA $1234
Extended Indirect Addressing	Read from the address specified... then get the value from that address	ADDA [$1234]
Indexed Addressing	Uses a 2nd setting byte - allows for Autoinc	,R offset,R label,pcr ,R+ ,-R []
Indexed Addressing: Zero Offset	Just use the address in the register	LDA ,Y LDA 0,Y LDA Y
Indexed Addressing: 5 bit offset	-16 to +15 offset	LDA -1,Y
Indexed Addressing: Consant offset from base register	8 / 16 bit offset from X,Y,U,S ... Can be negative or positive	LDA 1000,Y
Indexed Addressing: Constant Offset From PC	8 / 16 bit offset from PC	LDA $10,PC
Program counter relative	PCR is like PC, but is calculated by the assembler	ADDA label,PCR
Indirect with constant offset from base register	Load from the address in the register + offset	LDA [1,X]
Accumulator offset from Base register	Add accumulator (A/B/D) to a X,Y,U,S (not PC)	LDA B,Y
Indirect Accumulator offset from Base register	Load from the address made up of a X,Y,U,S Plus the accumulator	LD [B,Y]
AutoIncrement	Add 1 or 2 to the register	ADDA ,X+ ADDA ,X++
AutoDecrement	Subtract 1 or 2 from the register	ADDA ,-X ADDA ,--X

Indirect AutoIncrement	Load from the address in Register, then add 1 or 2	ADDA [,X+] ADDA [,X++]
Indirect AutoDecrement	Subtract 1 or 2 then Load from the address in Register	ADDA [,-X] ADDA [,--X]
Program relative	Offset to PC	BRA label

Hints

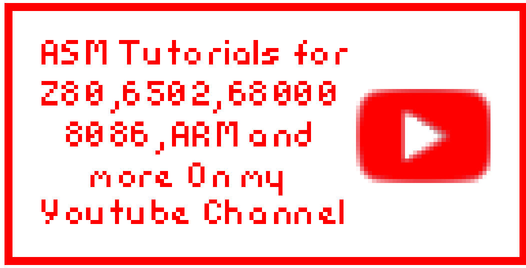
Saving a byte on return:

Rather than returning, if your last command is a pop, just pop the PC with your other registers:
PULS B , X , PC

Addresses, Numbers and Hex... 6809 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!

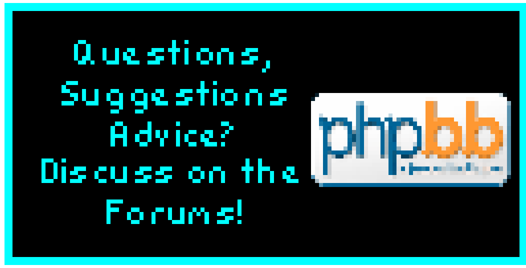
Prefix	Example	Z80 equivalent	Meaning
#	#16384	16384	Decimal Number
#%	#%00001111	%00001111	Binary Number
#$	#$4000	&4000	Hexadecimal number
#'	#'a	'a'	ascii value
	12345	(16384)	decimal memory address
$	$4000	(&4000)	Hexadecimal memory address



Missing Commands!

Commands we don't have, but might want!

DEX/DEY/INX/INY	Tfr X,D ;replace X with Y if required DecB ;or IncB as required Tfr D,X ;replace X with Y if required	
CLC	AndCC #%11111110	
SEC	OrCC #%00000001	
DeX	LEAX -1,X	



Lesson 1 - Getting started with 6809

Lets learn the basics of using 6809... In this lesson we'll set

some registers, and do a few simple maths operations.
We'll be testing on a Dragon 32 or XM7 (XM7 shown)

Structure of an ASM source file

Lets look at a simple file (Minimal.asm)

we have a **header** - We're including a header to do our setup

In our body we're running a simple monitor program - this is where you would put your code

In our footer we're including some useful files (with include statements)

This example will show a hello world message, the status of the registers, and dump some bytes of memory.

```
    include "\srcALL\v1_header.asm"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ldD #$1234              ;Test Value
    jsr Monitor             ;Show the Register

    ldy #32                 ;Bytes to dump
    ldx #$C000              ;Address to dump
    jsr Memdump

    ldy #Hello              ;255 terminated Message
    jsr PrintString         ;Show String to screen

InfLoop:
    jmp InfLoop             ;Infinite Loop

PrintString:                ;Print 255 terminated string
    lda ,Y+
    cmpa #255
    beq PrintStringDone
    jsr printchar
    jmp PrintString
PrintStringDone:
    rts

Hello:
    dc.b "HellO WORLD!?",255

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    include "\srcALL\v1_Monitor.asm"
    include "\srcALL\v1_Functions.asm"
    include "\srcALL\v1_Footer.asm"
```

This example will work fine on the Dragon 32 and the FM7

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!

It will not work on the Vectrex!... that's because the Vectrex is a vector based system, and the text routines will not work on that system.

Loading Values From Immediates

The 6809 has a 16 bit accumulator (D - made up of AB), Two index registers (X,Y) and two stack pointers (U,S)

These are LoaDed with LD?...where ? is a register name... an immediate value can be loaded with # followed by a number.

We can specify **Hex Values using $**
We can specify **Decimal without the $**
we can specify **binary with %**

we can **even load the Stack**.. but beware, the stack is used by calls, so we need to be sure the new value will work OK

```
TestBasics:
    ldD #$1234          ;Load Dual Accumulator (A=High B=Low)
                        ;$=Hex

    ldx #5678           ;Load X
                        ;no $=Decimal

    ldy #%101011110000  ;Load Y
                        ;%=Binary

  ifdef BuildXM7
      lds #$70F0        ;Load Stack
  endif
  ifdef BuildDGN
      lds #$7EF0        ;Load Stack
  endif

    ldu #$DEF0          ;Load User Stack
    jsr Monitor
```

The 16 bit Accumulator D is made up of two 8 bit accumulators A and B (A is the High byte of D - B is the low byte)

We can load these from an 8 bit immediate. **We can specify an Ascii value with '**

```
    lda #'A'         ;Store A (8 bit)
                     ;' = Ascii
    ldb #$22         ;Store B (8 bit)
    jsr Monitor
```

Here are the results

```
D:1234 X:162E Y:0AF0
S:70F0 U:DEF0 P:2199 C:00 Dp:00
D:4122 X:162E Y:0AF0
```

You'll see lots of JSR command - this is like a GOSUB in basic, it jumps to a subroutine...
RTS is the equivalent of RETURN...
JMP is like GOTO... don't worry too much though... we'll see these again later.

Loading and saving Values From Addresses in memory

Just like the 6502, If we don't specify the # then the number

[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)]

specified is an address... we can use **ST?** to Store register ? To an address

```
;NO # - Save to address
	std $6000		;Save D to $6000 ...
					;A->&$6000, A->&$6001
	stx $6002		;Store X (16 bit)
	sty $6004		;Store Y (16 bit)
	sts $6006		;Store Stack
	stu $6008		;Store User Stack

	sta $6009		;Store A (8 bit)
	stb $600A		;Store B (8 bit)

	ldy #16			;Bytes to show
	ldx #$6000		;Address to show
	jsr Memdump		;Show Memory
```

Here are the results

```
D:4122 X:162E Y:0AF0
S:70F0 U:DEF0 P:2199 C:00 Dp:00
6000:
41 22 16 2E 0A F0 70 F0
DE 41 22 00 00 00 00 00
```

Loading works the same way...

We don't need to specify a two byte address... if we **specify just one byte** (like $60) then the data will be loaded from the DIRECT PAGE.

This is like the 6502 Zero page, though register DP is used for the top byte of the address, and it doesn't have to be $00xx

```
;NO # - Load from address
	ldx $6000
;NO $ - Decimal value
	ldY 24576		;$6000 in decimal
	jsr Monitor

	ldd #$1234
	std $60			;Store in Direcpage address $0060
	ldd #$FEDC
	std $62

	ldy #16			;Bytes to show
	ldx #$0060		;Address to show
	jsr Memdump		;Show Memory
```

We've **loaded bytes** from memory,

And stored two words to the **Direct Page** (Zero Page)

```
6000:
41 22 16 2E 0A F0 70 F0
DE 41 22 00 00 00 00 00
D:4122 X:4122 Y:4122
S:70F0 U:DEF0 P:2199 C:00 Dp:00
0060:
12 34 FE DC 00 00 00 00
00 00 00 00 00 03 48 00
```

Specifying addresses in the direct page uses One byte, compared to the normal Two, so memory in this area saves program code and is faster.

The Direct page should be used as a 'Temporary store' for values you don't have enough register for.

Transferring values between registers

As well as immediates and memory, we have commands to transfer between registers.

We can **Copy the value** from one register to another with **TFR**... the source register is on the LEFT of the comma, the Destination is on the RIGHT

We can **swap registers** with **EXG**... the two registers values are swapped.

```
ldd #$FEDC        ;Test Value
jsr MonitorABP
tfr D,X           ;Store D in X
jsr MonitorABP
exg Y,X           ;Swap Y and X
jsr MonitorABP
```

Here are the results

```
D:FEDC X:0060 Y:0000
D:FEDC X:FEDC Y:0000
D:FEDC X:0000 Y:FEDC
```


INCrease, DECrease and CLeaR!

We can use **INC?** and **DEC?** to add 1 or subtract 1 from a register - these are great when doing loops

We can also zero a register using **CLR?**...

these only work on 8 Bit Registers A or B, not X,Y or D

```
TestMore:
    ldd #$8888
    jsr MonitorABP
    inca              ;Add 1 to A
    decb              ;Subtract 1 from B
    jsr MonitorABP
    clra              ;Zero A
    clrb              ;Zero B
    jsr MonitorABP
```

Here are the result

```
D:8888 X:2000 Y:001A P:2120 C:00
D:8987 X:2000 Y:001A P:2120 C:00
D:0000 X:2000 Y:001A P:2120 C:00
```

We have versions for memory addresses too... **INC**, **DEC** and **CLR** - these are also all 8 bit.

```
jsr MonitorMem
inc $6000         ;Add one to value at $6000
dec $6001         ;Subtract one from value at $6000
clr $6002         ;Store Zero to $6000
jsr MonitorMem
```

Here are the results

```
6000:
00 00 00 00 00 00 00 00
6000:
01 FF 00 00 00 00 00 00
```


Addition and Subtraction

We have **ADD?** and **SUB?** commands... these can work with 8 bit registers...

We also have **ADDD** and **SUBD** which work with the 16 bit accumulator

We also have **ABX** - which adds B to X (X=X+B)... A is not used...

ABX is the only such command, there's no AAX, ABY or ADX

```
TestAddSub:
	ldd #$8888
	jsr MonitorABP
	adda #$4			;Add 4 to A
	addb #$4			;Add 4 to B
	jsr MonitorABP
	suba #$4			;Sutract 4 from A
	subb #$4			;Sutract 4 from B
	jsr MonitorABP

	jsr NewLine

	jsr MonitorABP
	addd #$1234			;Add $1234 to D
	jsr MonitorABP
	subd #$1234			;Subtract $1234 to D
	jsr MonitorABP

	jsr NewLine
	ldx #$1234
	ldb #$4
	jsr MonitorABP
	abx					;Add B to X... X=X+B
	jsr MonitorABP			;There's no AAX!

	jmp InfLoop
```

Here is the result

```
D:8888 X:2000 Y:001A P:2120 C:00
D:8C8C X:2000 Y:001A P:2120 C:00
D:8888 X:2000 Y:001A P:2120 C:00

D:8888 X:2000 Y:001A P:2120 C:00
D:9ABC X:2000 Y:001A P:2120 C:00
D:8888 X:2000 Y:001A P:2120 C:00

D:8804 X:1234 Y:001A P:2120 C:00
D:8804 X:1238 Y:001A P:2120 C:00
```


JuMPs... Jump to SubRoutine... ReTurn from Subroutine... Labels

There will be many times in our program that we want to call subroutines, or jump to different places in our code.

When we want to mark part of our code as a destination for a jump or a subroutine we use a Label - these are always at the far left of the source - whereas normal commands are tab indented.

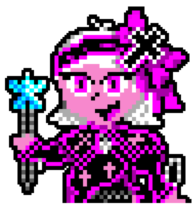
If we want to jump to a different position in our code we can use

JMP - this sets the program counter to the address of the specified **label** (of course a fixed address can be used too)

If we want to jump to a subroutine - and come back once the subroutine is done we use **JSR**.... the end of the subroutine is marked by a **RTS** return command... execution will continue on the line after the JSR command

```
TestSubroutine:
	lda #'A'
	jsr printchar
	lda #'B'
	jsr printchar

	jmp Skip			;Jump to label Skip

	lda #'X'			;This never happens
	jsr printchar
Skip:
	lda #'C'
	jsr printchar

	jsr TestJsr			;Call subroutine

	lda #'E'			;This line happens after sub RTS
	jsr printchar

	jmp InfLoop

TestJsr:				;Sub routine
	lda #'D'
	jsr printchar
	rts				;Return from subroutine
```

```
ABCDE
```

You'll notice that many of the commands on the 6809 are the same as the 6502, but many are different on the 6809

Both have their 'roots' in the 6800, and the 6809 has some compatibility with the old cpu, but code would need recompiling for the 6809...

Next time we'll take a look at addressing modes - and we'll learn about all the impressive options the 6809 offers.

Lesson 2 - Addressing modes on 6809

Lets look into more detail of the 6809... our commands need to load from or save to somewhere, and the 6809 offers us a wide range of 'addressing mode' choices for this purpose,

Lets try them all out, and learn about them.

Inherant Addressing

Inherent addressing is commands which do not take a parameter, the 'destination' of the command is built into the command.

In this case... **INCA** will always use A as a source and destination

```
TestSimple:
;Inherant Addressing
    jsr MonitorABP
    inca
    jsr MonitorABP
    jsr NewLine
```

here is the result

```
D:8800 X:1A18
:1A18 Y:0000
D:8900 X:1A18
```

Register Addressing

Register addressing is where we specify a register by name as a source or destination

```
;Regsiter Addressing
    jsr MonitorABP
    tfr a,b                 ;Transfer A->B
    jsr MonitorABP
    jsr NewLine
```

The source and destination are registers - we copied A to B!

```
D:8900 X:1A18
:1A18 Y:0000
D:8989 X:1A18
```

Immediate Addressing

We've seen **Immediate addressing** before, this is where a fixed value is specified with the command with a # symbol, and that value is used as the parameter

```
;Immediate Addressing
    jsr MonitorABP
    lda #$1F               ;load 1F into the register
    jsr MonitorABP
    jsr NewLine
    jmp InfLoop
```

Here is the result

```
D:8989 X:1A18
D:1F89 X:1A18
```

Direct Page addressing

Direct Page Addressing is basically the same as Zero Page addressing... however thanks to the DP register the Direct Page can be at any address!

We specify a single byte (without a #)... this is used as the bottom byte of an address - the top byte is taken from the DP register... that address contains the source parameter for the command

```
;Direct Page addressing
TestDP:
    jsr MonitorABP
    lda $80                ;Load A from ZP address
    jsr MonitorABP         ;$80 = $0080
    jsr NewLine
    jmp InfLoop
```

In this example we specified $80 , and DP=00... so we loaded from address $0080

```
0080:
11 12 13 14 15
D:8800 X:1A18
D:1100 X:1A18
```


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)

Indirect Direct Page Addressing

Indirect Direct Page Addressing also uses the DP - however the two bytes at the specified DP address are used as an address... and the parameter is loaded from that address! The one byte DP address is in square brackets	;Indirect Direct Page Addressing lda [$81] ;Load A from address at DP addr jsr MonitorABP ;$80 = $0080
The address specified is $81 in the example... so as DP=00 we load in from $0081... the two bytes here are $1213... this becomes the source address for our parameter - at address $1213 is $42 - so this is the value that ends up being loaded into A	0080: 11 12 13 14 15 16 17 1210: 00 40 41 42 43 44 45 D:0000 X:1A18 Y:0008 D:1100 X:1A18 Y:0008 D:4200 X:1A18 Y:0008

Don't forget! Unlike the 6502 the 6809 is BIG ENDIAN... the value $1234 will be stored in memory as $12 $34

It sounds logical if you're used to the 68000 - but it will be a shock to the 6502 or z80 users!

Extended Direct addressing

Extended Direct Addressing sounds complex, but it's not... If we specify a one byte address, we load from the direct page, but if extend our address to two bytes, we specify the Direct address to load from	TestExtended: ;Extended Direct addressing lda $2000 jsr MonitorABP
In this case we specify $2000 - so the parameter is read from address $2000	2000: 1A 1B 1C 1D 1E D:1A00 X:1A18

Extended Indirect Addressing

Extended indirect Addressing is where we load our parameter from the address at an address (specified in 16 bits - a full address) The two byte DP address is in square brackets	;Extended Indirect Addressing lda [$2000] jsr MonitorABP jmp InfLoop
In this example we specified [$2000]... at this address is $1A1B (Big Endian)... so the parameter loaded into A is read from $1A1B ($CC)	

```
0080:
11 12 13 14 15 16 17
2000:
1A 1B 1C 1D 1E 1F 20
1A18:
FF EE DD CC BB AA 99
D:1A00 X:1A18 Y:0000
D:CC00 X:1A18 Y:0000
```

Indexed Addressing

Indexed Addressing uses a register plus an offset... the offset comes first, followed by a comma, then the register

The **offset can be Zero** - in which case we can omit it.
The offset can be **positive** or **negative**, and the register can be X,Y,S or U (not D)

```
TestIndexed:
;Indexed Addressing
    ldy #$2000
    ldu #$2002
    jsr MonitorABP

    lda ,Y              ;Zero Offset
    ldb 1,Y             ;Parameter loaded from address in Y +1
    ldx 2,U             ;Can Also use Y,X,S,U
    ldy -1,U            ;Can be negative!
    jsr MonitorABP
```

Here are the results.

Y was pointing to $2000 (the 1A)... U was pointing to $2002 (the 1C)

```
0080:
11 12 13 14 15 16 17 18
2000:
1A 1B 1C 1D 1E 1F 20 21
D:8800 X:1A18 Y:2000 P:20BD
D:1A1B X:1E1F Y:1B1C P:20C9
```

We can use a **symbol** to give a numbered offset a label.

On the 6809 we can even have a **16 bit offset**!

```
    ldy #2
    jsr MonitorABP

testoff equ $80

    lda testoff,y       ;Symbol used

    ldb $2000,y         ;Very large offsets possible!

    jsr MonitorABP
```

Both these work just fine!

```
2000:
1A 1B 1C 1D 1E 1F 20 21
0080:
11 12 13 14 15 16 17 18
D:1A1B X:0080 Y:0002 P:20E7
D:131C X:0080 Y:0002 P:20F2
```

There are **multiple ways** of specifying a zero offset

```
TestZero:
;Indexed Addressing - Zero Offset
    jsr MonitorABP
    ldy #$2000

    lda ,Y              ;All these use value at address Y
    ldb 0,Y             ;All these use value at address Y
    ldx Y               ;All these use value at address Y

    jsr MonitorABP
```

Here are the results

```
2000:
1A 1B 1C 1D 1E
D:8800 X:1A18
D:1A1A X:1A1B
```

Actually, the offset doesn't have to be a fixed number - we can use the Accumulator as an offset!

This has a different name though... "Accumulator offset from Base register"

Program Counter Relative

We can specify a label as a parameter by offset using **PCR** as the parameter, an the label as the offset,

the assembler will calculate the offset for us.

```
;Program counter relative
ProgramCounterRelative:
    jsr MonitorABP
    ldd TestLabel,PCR       ;Calculate offset from Program counter
    jsr MonitorABP              ;to Testlabel

    jmp InfLoop
TestLabel:
    dc.w $FEDC              ;Test Data
```

The data has been loaded in

```
D:8800 X:1A18
D:FEDC X:1A18
```

Indirect With Constant Offset

Rather than reading a parameter from a register+offset, we can read the parameter from the address at the address register+offset...

We specify an offset and a register in Square brackets []

this is **Indirect With Constant Offset Addressing**

```
;Indirect with constant offset from base register
IndirectWithConstantOffset:
    ldx #1                  ;Load X with 1
    jsr MonitorABP          ;Call the monitor
    lda [$80,X]             ;Preindex with constant ($0080+X)
                                ;so load data from address at ($0081)
    ldb [$2000,X]           ;Preindex with constant ($2000+X)
                                ;so load data from address at ($2001)
    jsr MonitorABP          ;Call the monitor
    jmp InfLoop             ;Inf Loop
```

in the first example we loaded from **[$80,X]**... at address $0080+1 is $1213 - this becomes the source address of the parameter... at address $1213 is value $42

in the second example we loaded from **[$2000,X]**... at address $2000+1 is $1B1C - this becomes the source address of the parameter... at address $1B1C is value $AB

```
0080:
11 12 13 14 15 16 17 18
2000:
1A 1B 1C 1D 1E 1F 20 21
1210:
00 40 41 42 43 44 45 46
1B18:
EF DE CD BC AB 9A 89 78
D:7800 X:0001 Y:0000 P:214D C:20
D:42AB X:0001 Y:0000 P:2158 C:00
```


Accumulator offset from Base register

Rather than a fixed offset, we can use an accumulator (**A**,**B** or D) as an offset from an index register (X,Y,S,U) this is known as Accumulator Offset From Base Register

in this example The parameter will be loaded from the address in **A+X** or **B+X**

```
;Accumulator offset from Base register
AccumulatorOffsetFromBase:
    ldx #$2000          ;Load X with $2000
    lda #4              ;Offset 4
    ldb #5              ;Offset 5
    jsr MonitorABP      ;Call the monitor

    ldy A,X             ;Load from address (X+A)

    ldd B,X             ;Can also use B or D (X+B)

    jsr MonitorABP      ;Call the monitor
    jmp InfLoop         ;Inf Loop
```

Here are the results

```
2000:
1A 1B 1C 1D 1E 1F 20 21
D:0405 X:2000 Y:0000 P:215F
D:1F20 X:2000 Y:1E1F P:2167
```


Indirect Accumulator Offset From Base Register

Rather than use the address Accumulator+Base as the address of the parameter... we can indirectly use the address at that address as the source.

```
;Indirect Accumulator offset from Base register
IndirectAccumulatorOffsetFromBase:
    ldx #$0080          ;Load X with $0080
    lda #1
    ldb #2
    jsr MonitorABP      ;Call the monitor

    ldy [A,X]           ;Load from Address at Address (A+X)
                            ;Can also use B or D
    ldd [B,X]

    jsr MonitorABP      ;Call the monitor
    jmp InfLoop         ;Inf Loop
```

For Y
X=$0080 and A=1... so we look at address $0080 - at this

address is $1213... so this becomes the source address of the parameter - at address $1213 is value $4243

For D

X=$0080 and A=2... so we look at address $0080 - at this address is $1314... so this becomes the source address of the parameter - at address $1314 is value $3334

```
0080:
11 12 13 14 15 16 17 18
1210:
00 40 41 42 43 44 45 46
1310:
00 30 31 32 33 34 35 36
D:0102 X:0080 Y:0008 P:2177
D:3334 X:0080 Y:4243 P:217F
```


Post Increment and Pre Decrement

AutoIncrement will read from the address in a register, and add 1 or 2 to the register AFTER the read.

We just put a **+** or **++** at the end of the register - the register can be X,Y,S or U

```
;AutoIncrement
AutoInc:
    ldx #$0080          ;Load X with 1
    ldy #$0080          ;Load X with 1
    jsr MonitorABP
    lda X+              ;Load from address in X, Add 1 to X
    ldb Y++             ;Load from address in Y, Add 2 to Y
    jsr MonitorABP
    lda X+              ;Load from address in X, Add 1 to X
    ldb Y++             ;Load from address in Y, Add 2 to Y
    jsr MonitorABP
    lda X+              ;Load from address in X, Add 1 to X
    ldb Y++             ;Load from address in Y, Add 2 to Y
    jsr MonitorABP
    lda X+              ;Load from address in X, Add 1 to X
    ldb Y++             ;Load from address in Y, Add 2 to Y
    jsr MonitorABP
    jmp InfLoop
```

Here are the results

```
0080:
11 12 13 14 15 16 17 18
D:7800 X:0080 Y:0080 P:2186
D:1111 X:0081 Y:0082 P:218D
D:1213 X:0082 Y:0084 P:2194
D:1315 X:0083 Y:0086 P:219B
D:1417 X:0084 Y:0088 P:21A2
```

AutoDecrement will decrease the register BEFORE the read by 1 or 2

We just put a **-** or **--** at the end of the register - the register can be X,Y,S or U

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!

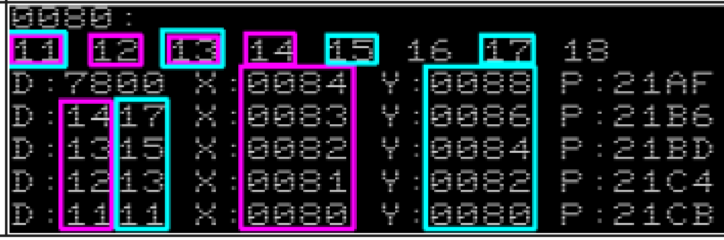
Mouse reading on the MSX

Hello World on RISC-OS

Atari 800 / 5200 - ASM PSET and

```
;AutoDecrement
AutoDec:
    ldx #$0084
    ldy #$0088
    jsr MonitorABP              ;Call the monitor
    lda -X
    ldb --Y
    jsr MonitorABP              ;Call the monitor
    lda -X
    ldb --Y
    jsr MonitorABP              ;Call the monitor
    lda -X
    ldb --Y
    jsr MonitorABP              ;Call the monitor
    lda -X
    ldb --Y
    jsr MonitorABP              ;Call the monitor
    jmp InfLoop
```

Here are the results



Indirect Post Increment and Pre Decrement

Just like with Post Increment we can change the index register - but we can use that indirectly, and the address at the address that register points to becomes the address of the parameter

We just put a **+-** or **++** at the end of the register for an Inc of 1 or 2... we use Square brackets [] to denote indirection - the register can be X,Y,S or U

```
IndirectAutoInc:
    ldx #$0080
    ldy #$0080
    jsr MonitorABP
    lda [X+]                ;Load from Address at Address X - Add 1 to X
    ldb [Y++]               ;Load from Address at Address Y - Add 2 to Y
    jsr MonitorABP
    lda [X+]                ;Load from Address at Address X - Add 1 to X
    ldb [Y++]               ;Load from Address at Address Y - Add 2 to Y
    jsr MonitorABP
    lda [X+]                ;Load from Address at Address X - Add 1 to X
    ldb [Y++]               ;Load from Address at Address Y - Add 2 to Y
    jsr MonitorABP
    lda [X+]                ;Load from Address at Address X - Add 1 to X
    ldb [Y++]               ;Load from Address at Address Y - Add 2 to Y
    jsr MonitorABP
```

Here are the results...

The registers X and Y are Incremented... and the address in those registers is used to generate an address that is the source of the parameter

[POINT for Pixel Plotting]

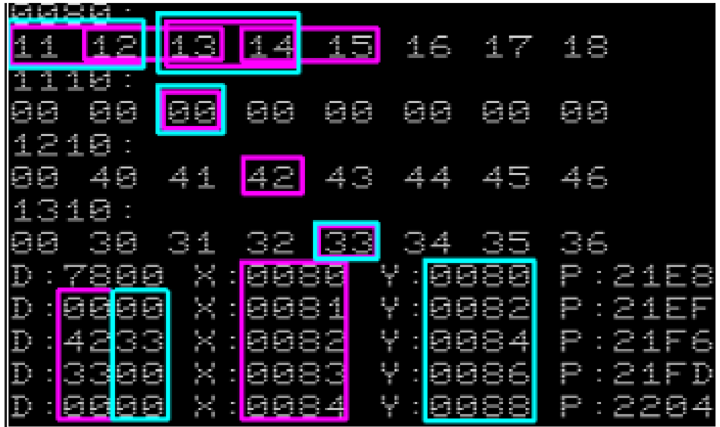
[Apple 2 - ASM PSET and POINT for Pixel Plotting]

[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)]



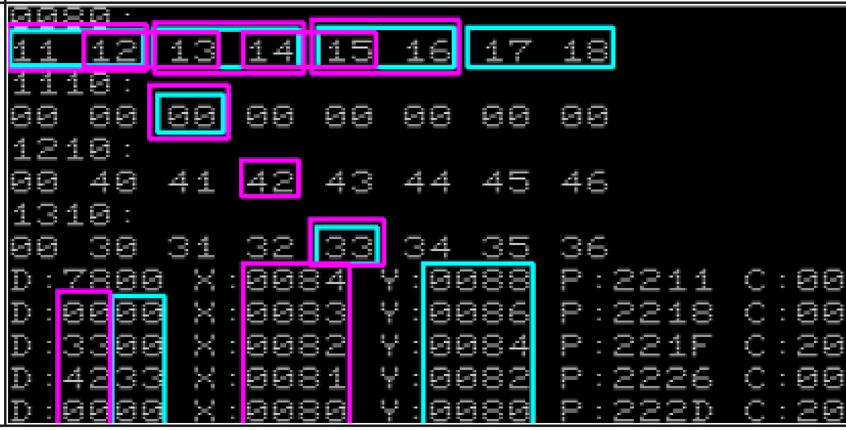
We can also use indirection with Pre-Decrement, and the address at the address that register points to becomes the address of the parameter

We just put a **-** or **--** at the start of the register for an Dec of 1 or 2... we use Square brackets [] to denote indirection - the register can be X,Y,S or U

```
;Indirect AutoDecrement
IndirectAutoDec:
    ldx #$0084
    ldy #$0088
    jsr MonitorABP
    lda [-X]            ;Subtract 1 from X - Load from Address at Address X
    ldb [--Y]           ;Subtract 2 from Y - Load from Address at Address Y
    jsr MonitorABP
    lda [-X]            ;Subtract 1 from X - Load from Address at Address X
    ldb [--Y]           ;Subtract 2 from Y - Load from Address at Address Y
    jsr MonitorABP
    lda [-X]            ;Subtract 1 from X - Load from Address at Address X
    ldb [--Y]           ;Subtract 2 from Y - Load from Address at Address Y
    jsr MonitorABP
    lda [-X]            ;Subtract 1 from X - Load from Address at Address X
    ldb [--Y]           ;Subtract 2 from Y - Load from Address at Address Y
    jsr MonitorABP
    jmp InfLoop
```

Here are the results...

The registers X and Y are Incremented.. and the address in those registers is used to generate an address that is the source of the parameter



Program Relative Addressing

Branch commands use a 'relative offset' to

the destination, represented in the byte code as a positive or negative number.
This is calculated automatically by the assembler...

BRA uses a single byte offset , so can only jump short distances... LBRA (Long Branch) can jump long distances

```
;Program relative
ProgramRelative:
    lda #'A'
    jsr PrintChar
    lda #'B'
    jsr PrintChar

    bra Skipped          ;Branch to relative location (-128 to +127)

    ;lbra Skipped        ;Works for Long distances
    ;ds 129          ;Relative Branch can only be a short distance (use LBRA)

    lda #'X'
    jsr PrintChar
Skipped:
    lda #'C'
    jsr PrintChar
```

Here are the results!

```
ABC
```

Load Effective Address

All the commands above will calculate an an address, and use the parameter at that address.

However, there may be times when we want to calculate the address, but save it for later... Load Effective Address does this for us!

This is a handy 'time saver' if we need to use a calculated address several times - it's faster to store the address in a register, then use that pre-calculated address for future commands.

```
LoadEffectiveAddress:
    ldy #$2000

    leaU $100,y ;Load Address of $100+$2000...  U=$2100

    jsr Monitor
    jsr NewLine

    leaX --y    ;Load Address of $2000--...     X=$1FFE ,Y=$1FFE

    leaS [$80]  ;Load Address of Indirect $0080 S=$1122

    jsr Monitor

    jmp InfLoop
```

In the **First example**, we used an offset of $100 from Y. As Y=$2000, the result loaded into U is $2100

In the **Second Example**, we PreDecremented Y by 2. As Y was $2000, Y Went down to $1FFE, and that value is loaded int X

In the **Third Example**, we loaded the indirect address at $0080... at address $0080 is $1112 - so this value is loaded into S

```
0080:
11 12 13 14 15 16 17 18
D:7800 X:1B18 Y:2000
S:70A5 U:2100 P:235E C:20

D:7800 X:1FFE Y:1FFE
S:1112 U:2100 P:235E C:20
```

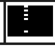
Lesson 3 - Carry, Branch, Test

We need to learn about conditions and branching... which brings up the topic of Flags!

Lets learn about it all!

Branch to Subroutine, and Long branch

The 6809 has a wide variety of Branch commands - these use a single byte relative offset to the current code position.
 	
While most of these are conditional, the 6809 actually has a '**Branch to Subroutine**' command (BSR) - this is the same as JSR, but uses a single byte for the address (saving 1 byte over JSR)
 	
on the 6809 All branch commands have a 'Long' Equivalent (in this case **LBSR**), this uses a 16 bit offset, allowing a branch to anywhere in ram.
	`TestJsrBsr:`
	`    bsr SubTest      ;Short distances only Branch to Sub (-128 to +127)`
	`    ;lbsr SubTest     ;Long distance Branch to Sub (Same as JSR)`
	`    jmp InfLoop`
	`    ;ds.b 256         ;Long branch test (BSR will fail)`
	`SubTest:`
	`    lda #'!'`
	`    jsr PrintChar`
	`    rts`

The subroutine will show an ! onscreen	

Add with Carry, and Subtract with Carry

When we do a mathematical operation like ADD or SUB, there is the possibility that the mathematical operation will cause the value in the register to go over 255 or under 0, causing a 'Carry' or 'Borrow'

If we want, we can use an extra register as a second byte (in a 16 bit pair - or more)

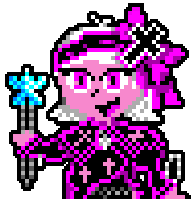
An ADD or SUB command will set the carry (also used as borrow)... we then use **ADC for Add with Carry**, or **SBC for Subtract with carry** - which will Add or subtract a value PLUS the Carry flag (if set)

```
TestAdcSbc:
    lda #6
    sta $1              ;Counter
    lda #1              ;High Byte
    ldb #253            ;Low  Byte

;Add with Carry Test
AdcRep:
    addb #1             ;Add 1 to Low Byte
    jsr MonitorB
    jsr MonitorCC
    adca #0             ;Add 0 + Carry to High byte
    jsr MonitorA
    jsr NewLine
    dec $1              ;Decrease counter and repeat
    bne AdcRep

    jsr NewLine
    pshs d
        lda #6          ;Reset the loop counter
        sta $1
    puls d

;Subtract with Carry Test
SbcRep:
    subb #1             ;Subtract 1 from the Low Byte
    jsr MonitorB
    jsr MonitorCC
    sbca #0             ;Subtract 0 + Carry from High byte
    jsr MonitorA            ;(Carry acts as Borrow)
    jsr NewLine
    dec $1              ;Decrease counter and repeat
    bne SbcRep
```

Here are the results... **when the Carry flag was set**, the High byte (A) is affected by the ADC or SBC.

```
B:FE ----N--- A:01
B:FF ----N--- A:01
B:00 --H--Z-C A:02
B:01 -------- A:02
B:02 -------- A:02
B:03 -------- A:02

B:02 -------- A:02
B:01 -------- A:02
B:00 -----Z-- A:02
B:FF ----N--C A:01
B:FE ----N--- A:01
B:FD ----N--- A:01
```

We've looked at ADD and SUB, but many other commands affect and use the carry flag,



Rotate commands will shift bits around the register, and some will use the carry too.

Branch on Carry

We've learned ADD can set the carry... but what if we want to do a different command depending on the Carry flag?

We can use **BCC** to Branch if Carry Clear (no carry)
We can use **BCS** to Branch if Carry Set (carry)

```
TestCarry:
    lda #253
    adda #1             ;254
    jsr MonitorACC
    adda #1             ;255
    jsr MonitorACC
    adda #1             ;0 - Carry!
    jsr MonitorACC
    adda #1             ;1 - Rem this out to cause a carry
    jsr MonitorACC

    bcc ShowNc          ;Branch if Carry Clear (NC)

    bcs ShowC           ;Branch if Carry Set (C)
```

Depending on the **Carry flag**, one of the two branches will occur!

```
A:FE ----N---
A:FF ----N---
A:00 --H--Z-C
A:01 --------
NC
```

To properly learn about how commands affect these flags, and the branches, you really need to download the source, and change the commands.
If you REM (;) out the last ADDA, you'll leave the carry flag set, and the BCC branch will happen.

What? you thought you didn't need to try things yourself? Good luck with that!

Compare, and Branch on Equal / Not Equal (Zero Flag)

When we want to test a value in a register, we can do this with Compare (**CMP**).... this will set the flags, but not change any register - we have a whole range of CMP commands for different registers.

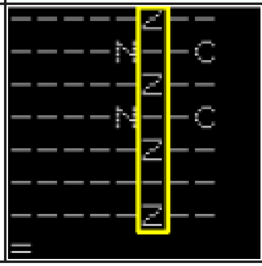
The easiest Branch we can try out is BEQ and BNE... our

Compare command will set the Zero flag if the difference between the register and the compared value is Zero...

BEQ (Branch if EQual) will branch if the Zero flag is set
BNE (Branch if Not Equal) will branch if the Zero flag is set

```
TestCmpEqNe:
      lda #1
      ldb #2
      ldx #3
      ldy #4
      ldu #5

      cmpa #1                        ;Compare A
      jsr MonitorCCNL
      cmpb #22                       ;Compare B
      jsr MonitorCCNL
      cmpd #$0102                    ;Compare D (AB)
      jsr MonitorCCNL
      cmpx #$33                      ;Compare X
      jsr MonitorCCNL
      cmpy #4                        ;Compare Y
      jsr MonitorCCNL
      cmps #4                        ;Compare S
      jsr MonitorCCNL

      cmpu #$55                      ;Compare U (NE Test)
      cmpu #$5                       ;Compare U (EQ Test)
      jsr MonitorCCNL

      beq ShowEq                     ;Branch on Z flag Set
      bne ShowNe                     ;Branch on Z flag clear

      ;lbeq ShowEq                       ;Long Branch on Z flag Set
      ;lbne ShowNe                       ;Long Branch on Z flag clear

      jmp InfLoop
      ;ds.b 2048                     ;Long branch test
ShowNe
      lda #'!'
      jsr PrintChar
ShowEq
      lda #'='
      jsr PrintChar
      jmp InfLoop
```

Which Branch occurs will depend on whether the **Zero flag** was set or not.



The CMP commands are actually a 'simulated subtraction' - in the sense that they set the flags the same as a SUB would, but they leave the register unchanged.

That's why Z is set if the two compared values are the same.

Comparisons of Unsigned numbers

Depending if the values in our registers are Signed or Unsigned numbers we need to use different branch commands for our comparison

If our number is UNSIGNED (0 to 255 or 0 to 65535) then we use the following commands:

BHI will branch if Higher (>)
BHS will branch if Higher or Same (>=)
BLO will branch if Lower (<)
BLS will branch if Lower or Same (<=)

```
TestUnsigned:
    ldd #$7000          ;28672

    ;cmpd #$7000        ;=
    ;cmpd #$7800        ;D < 30720
    ;cmpd #$8000        ;D < 32768   (-32768 Signed)
    cmpd #$6000         ;D > 24576

    bhi ShowGreater     ;Branch if Higher
    bhs ShowGreater     ;Branch if Higher or Same

    bls ShowLess        ;Branch if Lower or Same
    blo ShowLess        ;Branch if Lower
```

In this example D contained 28672... we compared to 24576, so BHI occurred - showing a >

Comparisons of Signed numbers

If our number is SIGNED (-128 to 127 or -32768 to 32767) then we use the following commands:

BGT will branch if Greater Than(>)
BGE will branch if Greater or Even (>=)
BLT will branch if Less Than (<)
BLE will branch if Less or Even (<=)

```
TestSigned:
    ldd #$7000          ;28672

    ;cmpd #$7000        ;=
    ;cmpd #$7800        ;D < 30720
    cmpd #$8000     ;D > -32768 (32768 Unsigned)
    ;cmpd #$6000        ;D > 24576

    bgt ShowGreater     ;Branch if Greater Than (Signed)
    bge ShowGreater     ;Branch if Greater or Even (Signed)

    blt ShowLess        ;Branch if Less Than (Signed)
    ble ShowLess        ;Branch if Less or Even (Signed)
```

In this example D contained 28672... we compared to -32768, so BGT occurred - showing a >

Note, If we had

Checking if Signed or Unsigned

We're going to do some tests on positive and negative numbers.

We'll see how the flags change as a number goes from Positive to Negative

The Negative flag denotes if a register is Negative - effectively if it's top bit is 1

the oVerflow flag denotes if a register just changed sign - for example if we add 1 to 127 ($7F) it will become -128 ($80)

BMI will Branch if MInus (N flag set - top bit 1)
BPL will Branch if PLus (N flag clear - top bit 0)

```
TestNegativePositive:
    lda #-1                     ;Negative Number
    jsr MonitorACC
    lda #1                      ;Positive Number
    jsr MonitorACC
    jsr NewLine

    lda #6
    sta $10                     ;Loop Count
    lda #124
TestNegativePositiveAgain:
    inca                        ;Increse A
    jsr MonitorACC
    dec $10
    bne TestNegativePositiveAgain

    jsr NewLine

    ldd #-100                   ;Negative Number
    jsr MonitorDCC
    ldd #100                    ;Positive Number
    jsr MonitorDCC

    bmi BranchMinus             ;Branch if top bit 1 (N flag)
    bpl BranchPlus              ;Branch if top bit 0 (N flag)
```

the N flag will be set when a number is negative (the top bit is 1)

When the sign changes, the oVerflow flag is set.

In this example the sign was positive, so BPL occurred, and a + was shown

```
A:FF ----N---
A:01 --------

A:7D --------
A:7E --------
A:7F --------
A:80 ----N-V-
A:81 ----N---
A:82 ----N---

D:FF9C ----N---
D:0064 --------
+
```

Checking if Overflowed

If our mathematical operation accidentally changes the sign of a register we'll have a problem with our maths!

In order to work around this, we can check the overflow flag (V) and see if the sign changed.

We have two commands:

BVS will Branch if oVerflow Set
BVC will Branch if oVerflow Clear
	`TestOverflow:`
	`    lda #$7D              ;              125`
	
	`    inca                  ;Add 1 = $7E 126`
	`    jsr MonitorDCC`
	`    inca                  ;Add 1 = $7F 127`
	`    jsr MonitorDCC`
	`    inca                  ;Add 1 = $80 -128 oVerflow!`
	`    jsr MonitorDCC`
	`    ;inca                 ;Add 1 = $81 127`
	`    jsr MonitorDCC`
	
	`    bvs ShowOverflow      ;Branch if Overflow Set`
	`    bvc ShowNoOverflow    ;Branch if Overflow Clear`

Here's the result, in this case the V flag was set, so BVS occurred	`D:7E00 ---------`
	`D:7F00 ---------`
	`D:8000 ----N-V-`
	`D:8000 ----N-V-`
	`V`

BRanch Always and Never!

Finally we have to exceptions that don't look at the flags
 	
BRA will BRanch Always
BRN will BRanch Never
 	
BRN is useless really, it's just a 'quirk' of the instruction set.
	`TestAlwaysNever:`
	`    brn ShowNever         ;Branch Never`
	`    bra ShowBranch        ;Branch Always`
	`    jmp InfLoop`

Unsurprisingly BRA occurred!	`B`

BRN is pretty useless, but it could be handy for self modifying code!

Self modifying code is code that re-writes itself, for example, you may wish to modify a jump to turn it off.. changing BRA to BRN will do this!

Lesson 4 - The StackS! (Yes... there's two!) *

In Assembly, We often need to temporarily store values for a short while and bring them back later, we use the Stack for this!

Unlike most 8 bits, however the 6809 doesn't just have one stack... it has TWO! S and U - Twice the stacky goodness!

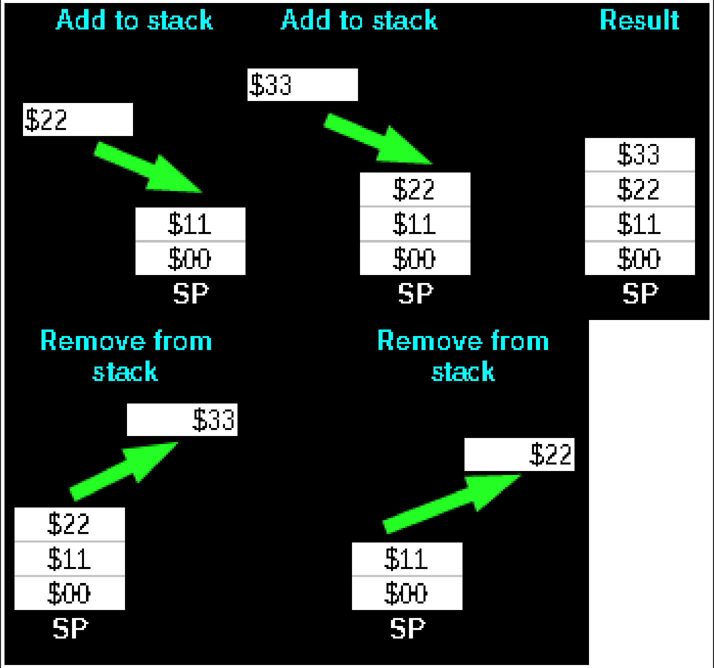
Stack attack!

'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



Using Stacks S and U

Like most CPU's, the stack on the 6809 moves DOWN memory as items are 'pushed' onto it... We'll need to point our stack pointers to the top of a spare area of memory.

When we want to set the stack up we use commands LDS and LDU (LoaD Stack and LoaD User stack)

```
ldS #$3E00          ;Set Stack S (Main)
ldU #$3D00          ;Set Stack U (User)
```

When we want to put an item on the stack we use a PUSH command PSHS/PSHD... we get it back with PULL command
PULS/PULU....

The stack is in **reverse order**, so the items are pulled off the stack in the opposite order they were pushed...

Pushed items can be pulled into **different registers** later.

Multiple registers can be pushed or pulled at the same time... the order we list the registers in the source **does not affect** the order they are processed by the command.

We can push one register onto the stack

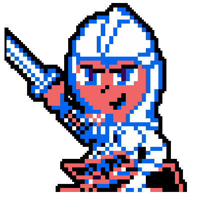
```
ldd #$FEDC
pshs d                  ;Push D onto the S Stack
    ldd #$1234
    pshs d                  ;Push D onto the S Stack
        ldd #$FFEE
        pshu d                  ;Push D onto the U Stack
        ldd #$DDCC
        pshu d                  ;Push D onto the U Stack

        jsr MonitorStacks

        jsr MonitorXYSU
        pulu x,y                ;Pull two 16 bit registers onto the stack

        pshu x,y                ;Push XY
        jsr MonitorXYSU
        pulu y,x                ;Pull YX Order doesn't matter
        jsr MonitorXYSU             ;Same effect as pulu x,y
        jsr NewLine
    puls x
    jsr MonitorXYSU
puls y
jsr MonitorXYSU
```

Here are the results.

With each push the stack pointer S or U will go down... with each pull S or U will go up

```
D:0000 X:2000 Y:001A
S:3E00 U:3D00 P:2176 C:00 Dp:00
3DF0:
00 3D F2 00 DD CC 3D F0
20 C9 20 24 12 34 FE DC
3CF0:
00 00 00 00 00 00 00 00
00 00 00 00 DD CC FF EE

X:3CF0 Y:0000 S:3DFC U:3CFC
X:DDCC Y:FFEE S:3DFC U:3CFC
X:DDCC Y:FFEE S:3DFC U:3D00

X:1234 Y:FFEE S:3DFE U:3D00
X:1234 Y:FEDC S:3E00 U:3D00
```

We can push and pull any of the registers onto the stack... 8 Bit accumulators A or B, 16 bit registers X, Y or D, the Stack pointers themselves S, U the program counter PC, or the 8 bit flags (CC)



Combining Bytes and Words, and pushing Stack registers!

We can **push the 8 bit accumulators** onto the stack - when we do so a single byte is pushed onto the stack
We can even then **pull them off into a 16 bit register.**

We should always make sure our routines leave the stack as they found them - with the stack pointer in the same position at the end as the start,

however we don't actually need to pull all the items off the stack - in this case we **pushed the Stack register onto the User stack - and popped it off at the end...** restoring the Stack pointer to it's starting state!

```
pshu s              ;Push the S stack onto the U Stack

lda #$FF
pshs a              ;Push A onto the S Stack (1 Byte)
    lda #$EE
    pshs a              ;Push A onto the S Stack (1 Byte)
        lda #$DD
        pshs a              ;Push A onto the S Stack (1 Byte)
            lda #$CC
            pshs a              ;Push A onto the S Stack (1 Byte)
                jsr MonitorStacks
                jsr MonitorXYSU
    puls x              ;Pull 2 bytes off the S stack into X
    jsr MonitorXYSU

pulu s              ;Pull the S stack pointer off the U Stack
                    ;We didn't Pull all the pushed data off the stack,
                        ;but now it doesn't matter
jsr MonitorXYSU
```

Here are the results.

```
3DF0:
00 3D F2 00 CC 00 3D F0
20 CD 20 5B CC DD EE FF
3CF0:
00 00 00 00 00 00 00 00
00 00 00 00 00 00 3E 00

X:3CF0 Y:0000 S:3DFC U:3CFE
X:CCDD Y:0000 S:3DFE U:3CFE
X:CCDD Y:0000 S:3E00 U:3D00
```

Subroutines and stacks

Subroutines and returns also use the main stack (S)

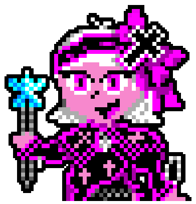
When we use **JSR**, the return address is pushed onto the stack...

for this reason we need to ensure our stack is the same at the end of the subroutine as when it started.

```
StackTests3:
    lda #$FF
    pshs a              ;Push A onto the S Stack (1 Byte)
    jsr MonitorA
        jsr SubTest         ;Sub call puts return address
        jsr MonitorStack         ;onto stack
    puls a
    jsr MonitorA
    jmp InfLoops

SubTest:
    lda #$EE
    pshs a              ;Push A onto the S Stack (1 Byte)
    jsr MonitorA
        lda #$DD
        pshs a          ;Push A onto the S Stack (1 Byte)
            jsr MonitorA
            jsr NewLine
            jsr MonitorStack
        puls a
        jsr MonitorA
    puls a
    jsr MonitorA
    jsr NewLine
    rts                 ;Stack needs to be in same position as
                            ;when sub started
```

In this example we can see the **Return address**, surrounded by the **Other values pushed** onto the stack

```
A:FF A:EE A:DD
3DF0:
21 F1 00 DD 00 3D F0 20
D3 20 94 DD EE 20 75 FF
A:DD A:EE
3DF0:
21 CC 21 44 3D F5 00 EE
00 3D F0 20 D3 20 78 FF
A:FF
```



While JSR pushes the address of the next command onto the stack, RTS effectively pops an item off the main stack (S) into the Program counter (PC)

JSR always uses the main stack (S) not the user stack (U).

Look Ma!... no RTS

we can also back up the flags (CC)!

In this example, our subroutine backs up **X**,**Y** and **CC**

Normally our subroutine would end with **RTS**... but as RTS effectively pops the **program counter (PC)** off the stack, we can do this with the PULS, and save the command!

```
StackTests4:
    ldY #$1234              ;Test Values
    ldX #$5678              ;Test Values
    lda #0
    jsr Monitor

    jsr SubTest2            ;Call Sub (Return addr onto stack)

    jsr Monitor
    jmp InfLoops

SubTest2:
    pshs x,y,cc
        ldY #$FFEE          ;More test values
        ldX #$DDCC
        jsr MonitorXYSU
        jsr MonitorStack
    puls x,y,cc,pc      ;Pop Return into PC - no need for RTS
```

Here are the results

```
D:0000 X:5678 Y:1234
S:3DFF U:3D00 P:218F C:04 Dp:00
X:DDCC Y:FFEE S:3DF9 U:3D00
3DF0:
00 00 00 3D F0 20 D5 20
C6 04 56 78 12 34 20 B1
D:0000 X:5678 Y:1234
S:3DFF U:3D00 P:218F C:04 Dp:00
```

No matter how many or few registers we push or pull the command always ends up being two bytes.

Therefore, if we need to end our sub with a PULS we can save one byte by adding the PC to the list... PC is always the last register popped off the stack if it's in the PULS list.

Lesson 5 - More Maths - Logical Ops, Bit shifts and more

We've covered a wide range of of commands, but we've overlooked a wide range of common maths commands.

Lets look at them now and learn what they do

Logical Operations... 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 0 1 0	1 1 0 0	1 0 0 0
ORA	1 0 1 0	1 1 0 0	1 1 1 0
EOR	1 0 1 0	1 1 0 0	0 1 1 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

Our commands need to specify the register we want to work on.

We can only operate on A or B (Not D)... we can also use ANDCC to set Condition codes, or ORCC to clear them.

AND will clear bits where the parameter bit is 0

OR will set bits where the parameter bit is 1

EOR will flip bits where the perameter bit is 1

COM will flip all the bits (complement) - it doesn't take a parameter

NEG will negate a number (turn a positive to negative or vice versa) - effectively flipping the bits and adding 1

```
TestLogicalOps:
    lda #%11001101
    ldb #$CD
    pshs d,cc
        jsr MonitorACCBits
        anda   #%11110000    ;Return 1 when both bits 1, else 0
        andb   #%11110000    ;Return 1 when both bits 1, else 0
        andcc  #%11110000    ;Return 1 when both bits 1, else 0
        jsr MonitorACCBits
        jsr NewLine
    puls d,cc
    pshs d,cc
        jsr MonitorACCBits
        ora   #%00001111     ;Return 1 when either bit 1, else 0
        orb   #%00001111     ;Return 1 when either bit 1, else 0
        orcc  #%00001111     ;Return 1 when either bit 1, else 0
        jsr MonitorACCBits
        jsr NewLine
    puls d,cc
    pshs d,cc
        jsr MonitorACCBits
        eora #%00001111      ;Return opposite when bit 1
        eorb #%00001111      ;Return opposite when bit 1
        jsr MonitorACCBits
        jsr NewLine
    puls d,cc
    pshs d,cc
        jsr MonitorACCBits
        coma                 ;Flip bits
        comb                 ;Flip bits
        jsr MonitorACCBits
        jsr NewLine
    puls d,cc
    lda #10
    ldb #-10
    jsr MonitorACCBits
    nega                     ;Negate number
    negb                     ;Negate number
    jsr MonitorACCBits
    nega                     ;Negate number
    negb                     ;Negate number
    jsr MonitorACCBits
```

The bottom bits of the **AND** Test were 0, so these were cleared in the test

The bottom bits of the **OR** test were 1, so these were set in the test

The bottom bits of the **EOR** test were 1, so these were flipped in the test
 	
When tested **COM** flipped all the bits
 	
When tested **NEG** negated the value - of course if we run it twice, we end up with the value we started
	`A:11001101 B:CD CC:----N---`
	`A:11000000 B:C0 CC:--------`
	
	`A:11001101 B:CD CC:----N---`
	`A:11001111 B:CF CC:----NZOC`
	
	`A:11001101 B:CD CC:----N---`
	`A:11000010 B:C2 CC:----N---`
	
	`A:11001101 B:CD CC:----N---`
	`A:00110010 B:32 CC:-------C`
	
	`A:00001010 B:F6 CC:----N---`
	`A:11110110 B:0A CC:-------C`
	`A:00001010 B:F6 CC:----N--C`

Rotations and Shifts

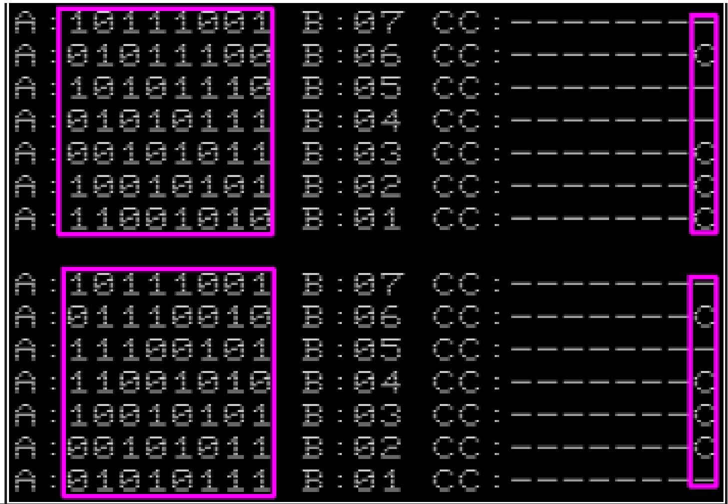
The 6809 has 3 kinds of rotate command, these can work to the Left, or Right, and can work on A, B or an address... they always work on a single byte

Operation	Left	Right
Rotate bits around through Carry	ROL	ROR
Arithmetic Shift (Signed)	ASL	ASR
Logical Shift (Unsigned)	LSL	LSR

```
TestRotation:
    lda #%10111001
    ldb #7
    pshs d
TestRotationAgain:
        jsr MonitorACCBits

        rora                    ;Rotate Right

        asra                    ;Arithematic Shift Right

        lsra                    ;Logical Shift Right

        decb
        bne TestRotationAgain
    puls d
    jsr NewLine
TestRotationAgainB:
    jsr MonitorACCBits

    rola                        ;Rotate Left

    asla                        ;Aritematic Shift Left

    lsla                        ;Logical Shift Left

    decb
    bne TestRotationAgainB
```

ROL and ROR rotate the bits by 1 to either the left or the right...

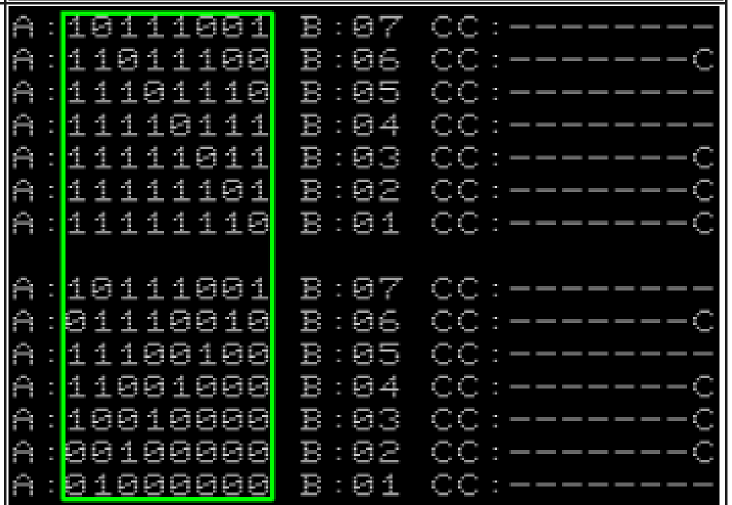
Of course the register contains 8 bits, but the Carry acts as a 9th bit - so any bit pushed out of the register goes into the carry, and any carry is pushed into the "other side" of the registers



ASL and ASR are designed to work with signed numbers...

When the bits are shifted to the right with ASR, any new bit will be the same as the previous top bit (in this case 1)...
ASL will fill any new bits with 0 ... though it's effectively the same as LSL

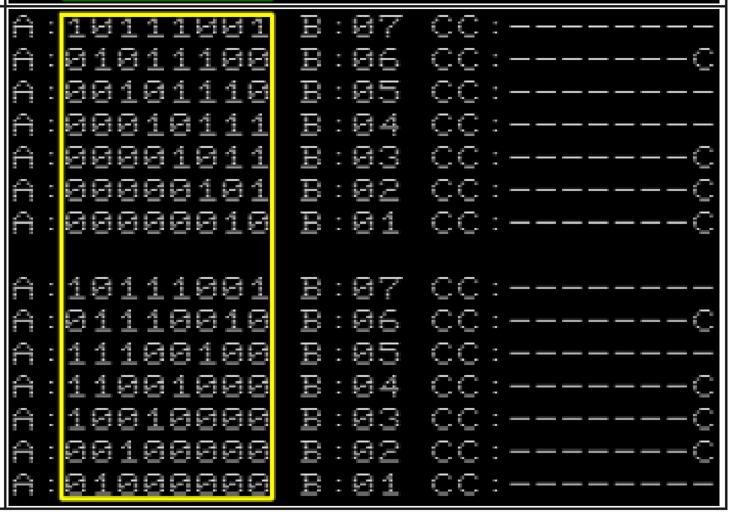
ASR effectively halves signed number, but keeps the sign intact... so 8 will turn to 4, or -8 will turn to -4
ASL effectively doubles numbers so 4 becomes 8



LSL and LSR are designed to work with unsigned numbers...

When the bits are shifted to the right with LSR or ASL, any new bits with 0 ...

LSR effectively halves unsigned number, but breaks signed numbers... so 8 will turn to 4
ASL effectively doubles numbers so 4 becomes 8



You need to download the source code and unrem the alternate shifts to see them in operation!

What do you mean, you can't be bothered! Grr… I don't know, the youth of today are sooo lazy!

Sign EXtending a register with...SEX (oh dear!)

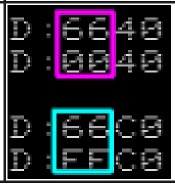
There may be times when you want to sign extend an 8 bit register to 16 bits - effectively filling the top byte with the top bit of the low byte.

We can do this with the dubiously named '**SEX**' command.... this sign extends B into A... so the D register now contains a 16 bit version of the signed number

In this example we load some junk into A - then we load a test number into B - and sign extend it.

```
TestSEX:
    lda #$66
    ldb #64
    jsr MonitorD
    sex                 ;Sign EXtend B->D (B->AB)
    jsr MonitorD            ;Bits of A to top bit of B

    jsr NewLine

    lda #$66
    ldb #-64
    jsr MonitorD
    sex                 ;Sign EXtend B->D (B->AB)
    jsr MonitorD            ;Bits of A to top bit of B

    jmp InfLoop
```

When the top bit was **zero**... A was set to %00000000 ($00)
When the top bit was **one**... A was set to %11111111 ($FF)



Test bits and set flags

We have some commands to set flags based on a register without changing it

the **BIT** commands allow us to provide a bitmask - this is effectively the equivalent of an AND command - but does not actually change the register

TST sets the flags according to a register (or memory address) and sets the flags accordingly - we can use it to check if a register contains zero

```
TestBits:
    lda #%00000011
    bita #%00000001        ;logical AND (only changes Flags)
    jsr MonitorACCBits
    bita #%00000010        ;logical AND (only changes Flags)
    jsr MonitorACCBits
    bita #%00000100        ;logical AND (only changes Flags)
    jsr MonitorACCBits
    bita #%00000011        ;logical AND (only changes Flags)
    jsr MonitorACCBits
    bita #%10000000        ;logical AND (only changes Flags)
    jsr MonitorACCBits

    jsr NewLine

    lda #0
    tsta                   ;Set Flags From A
    jsr ShowFlagsN
    lda #1
    tsta                   ;Set Flags From A
    jsr ShowFlagsN
    lda #-1
    tsta                   ;Set Flags From A
    jsr ShowFlagsN
```

BIT will compare the selected bits, and set the Z flag if the bits that were 1 in the parameter are 0 in the tested register.

TST will set the flags according to the register or memory address... in this case we set the Zero flag when A=0... and the Negative flag when A was <0

```
A:00000011 B:00 CC:--------
A:00000011 B:00 CC:--------
A:00000011 B:00 CC:-----Z--
A:00000011 B:00 CC:--------
A:00000011 B:00 CC:-----Z--

CC:-----Z--
CC:--------
CC:----N---
```

DAA - Decimal Adjust Accumulator

Binary coded decimal is where we use a byte to store two decimal digits (one per nibble)....
Actually they are stored as 'hexadecimal' however the digits never go over 9... for example $09 + 1 = $10 and $0099 +1 = $0100

DAA will decimal adjust the A accumulator (it cannot work on B)... this should be done after addition, and will correct the accumulator, converting numbers like $0A to $10

```
TestDaa
    ldb #16
    lda #08
DAA_Again:
    jsr ShowAN

    inca                ;Test #1
    ;adda #2            ;Test #2

    daa      ;Decimal Adjust Accumulator (1 nibble per digit)

    decb
    bne DAA_Again

    jmp InfLoop
```

The Two nibbles act as a pair of decimal numbers (even though they are actually base 16)	A:08 A:09 A:10 A:11 A:12 A:13 A:14

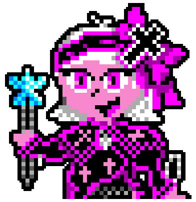
Multiplication!

The **MUL** command will multiply A * B and store the result in D

```
TestMult:
    lda #$10
    ldb #$69
    jsr MonitorD

    mul                     ;Multiply... D=A*B

    jsr MonitorD
```

in this example we multiplied $69 by $10 ... The result? $0690

```
D:1069
D:0690
```



We've learned how to Multiply... Wondering where the Divide command is?... Well, um, there isn't one!

You've got a MUL command - that's more than the Z80 or 6502 had... stop being so demanding!