Learn Multiplatform Assembly Programming with ChibiAkumas!

Beginner's Introduction to
Assembly & Instruction set overview
for Z80, 6502, 68000, 8086 and ARM!

First Edition

Download the source code for the examples in this book from:
www.chibiakumas.com/book

Visit my YouTube channel for video tutorials:
https://www.youtube.com/chibiakumas

This book is dedicated to my Mother and Father.

Without their education, support and encouragement
I would have never gained my interest or ability in computers and programming.

Table of Contents

Introduction

What is Assembly Language?

Assembly Language is an early very Low level language. Its commands are more readable than the 'Machine Code' the CPU actually runs, but is still reasonably human readable to help the programmer.

Rather than a 'High Level Language' like C++ which does a lot to help us, Assembly compiles straight to the machine code that the CPU runs.

This makes it faster than High level languages, but it means the programmer has do do more of the work of writing the program.

That sounds difficult! Why should I use Assembly Language then?

Well, if you're writing a program for your job – you almost certainly should use something else! But if you're looking to learn programming for a hobby, or you're trying to make an impressive game on an old system with limited hardware, then assembly is worth taking a look at.

I would say It's like taking a train compared to running a marathon. Do you want to get somewhere quickly, or do you want a challenge?

Assembly programming will teach you new skills, and let you take a different look at the old computers you've used for a long time.

Assembly language is harder to get started than C++, and you'll almost certainly need to research the hardware you're programming, and plan your project well. But the extra effort will be worth it, as the result will be a program that you made 100% by yourself!

Unlike with High level languages, there will be no Unreal engine doing all the game engine, no SDL handling the sound and no OpenGL doing the graphics. You'll have done it all yourself and know how it all works!

What can Assembly Language Do?

Anything you want! Because it's so low level, other languages, like C++, Forth and Basic, all end up as Assembly. So in theory anything those languages can do, you can do in Assembly, if you have the persistence!

If you're just starting out, you should probably be aiming to make simple games like Pong, Space Invaders or something of that style. While it's possible to write your own Window based Operating System in Assembly, you'd probably be better off aiming for something more realistic to start unless you're very confident.

How do I learn Assembly Language?

The only way to learn Assembly is through a combination of study and practice. You'll need to understand the instruction set of the CPU you're interested in, the hardware of the platform you're wanting to develop for and probably the OS of the system as well.
This can only be achieved by research and study and all the documentation you'll need is out there free on the web. But you'll probably find it impossible to understand without trying it for yourself, so writing little test programs to put what you read into practice will help.

How will this book help me?

This book is intended as a basic introduction into Assembly for someone with no prior experience. It covers the terminology that you will need to know in programming Assembly and classic hardware, it includes some simple samples for one platform on each CPU to get you started (for example the Amstrad CPC easiest Z80 platform to begin with) and a list of the instruction set of that CPU with clear simple descriptions.

You should read the terminology section in its entirety, and the Instruction set for the CPU you are interested in, then start looking at the particular platform you are really interested in (for example the Genesis or SNES).

The individual platforms are outside of the scope of this book. It would take a book on each platform to attempt to cover the intricacies of each system.

Once you complete this book, if you're looking for a place to continue learning, there is detailed documentation and examples, along with video tutorials, on the website of this book's author.

Take a look at http://www.learnasm.net for more detailed, platform specific content by this book's author. All the online content is free, and the lessons have video tutorials and downloadable source code.

Why did you write this book?

Though I had written small programs of a dozen lines in the 1990s and early 2000, I really started learning Z80 assembly in 2016 as a hobby. Assembly had always had a 'reputation' as extremely difficult, but I didn't find it particularly so. It required good planning and determination, but everything worth doing does.

What's more, learning Assembly taught me things about computer hardware that a computer science degree and 20 years as a professional programmer did not.

Let's be honest, I'm not a particularly smart guy, and I got bad grades at maths at school! If someone like me can make fun and interesting games in Assembly, and learn a lot along the way, I'm pretty sure most people can and that's why I started making my tutorials.

What are 'ChibiAkumas'?

"Chibi Akuma" means "Little Demon" in Japanese.
The ChibiAkumas are the characters I created for the 8 bit Amstrad CPC game I wrote in Z80 Assembly in 2016.

After I wrote the game, I started creating YouTube videos covering Assembly programming content using the game as subject material, and was surprised by how well they were received.

I wanted to respond to this apparent demand, so I started learning more CPU types, and learning about more hardware platforms.

I now create small multiplatform games, and matching tutorials for a wide range of platforms and CPUs, with many containing the same 'ChibiAkumas' characters, which have become my mascot and branding for my retro content.

You can see the Chibiakumas YouTube channel here:
www.youtube.com/chibiakumas

I first learned programming from an 'Usborne - Creepy computer games' book when I was about 8. These books used little cartoon characters to annotate code and explain concepts.

It's an educational style I'm a personally a fan of. I think adding a few cartoons and a little light heartedness to a difficult topic can make it more accessible to people, and it's my hope that my educational content could help others learn in the same way those books helped me.

Figure 1: Meet the Chibis:The Chibi Akumas and Chibi Aliens.

Enough talk! Let's start learning Assembly!

Chapter 1: Technical Terminology of Assembly and retro programming

Overview of the system

Every computer is different, but the general structure of the classic 8 and 16 bits tends to follow some common patterns.

Let's take a look at an 'imaginary system' its parts and how they connect:

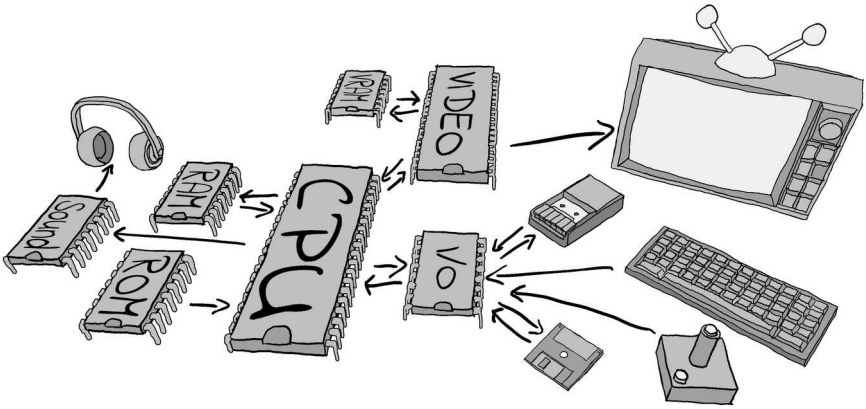


Figure 2: The layout of an imaginary 'typical' system.

Of course the part our commands will run on is the 'CPU', but the CPU can't do anything on its own.

To call the CPU the 'Brain' of the computer would be an overstatement, as it's a 'brain' that can't remember anything for very long, so it needs the Memory to store things for it. Computer systems will have a combination of RAM (Random Access Memory - for storing data) and ROM (Read Only Memory - for the operating system).

Some systems will have a Keyboard, but every system will have at least some kind of input like a gamepad. These don't connect directly to the CPU. There will be some kind of 'Input/Output chip' we'll have to 'ask' for the data.

Most systems will have some kind of sound. This will usually be some kind of chip we can tell what sound to make, but some have a 'beeper', where we basically have to make the shape of the sound with the CPU.

Depending on the system, there may be a tape or disk device we can read data from. The reading or writing procedure is too complex to do directly, so we probably want to ask the operating system to do the work for us.

Finally we have the graphics, because we'll want to 'see' some results!
How the graphics work will vary from system to system. Some systems have separate graphics memory and others do not. All will have some kind of 'graphics chips' which we can tell to do things, like change the colors on screen, or change the screen resolution.

CPU Terminology

Let's start by looking at some common terms you'll come across in Assembly. We'll discuss what they mean to you as a programmer.

CPU

The CPU is the 'Calculator' of our machine, it reads in numbers and commands, and calculates the results.

Commands and parameters will be usually read in from memory, the results will be stored back to memory. Sometimes the source or destination may be a 'device' like a joystick or speaker. How connection to such devices works depends on the CPU and the hardware.

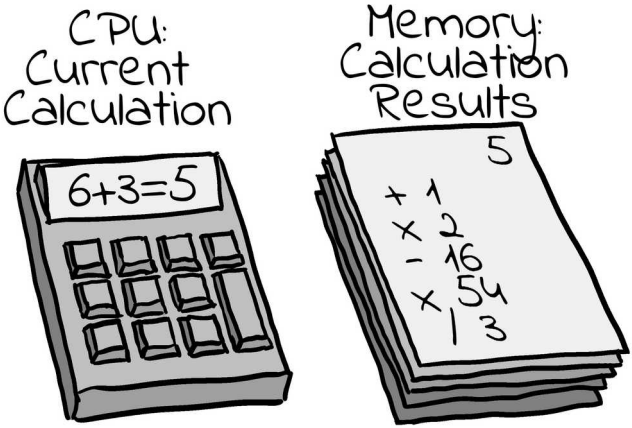


Figure 3: The CPU calculates the current calculation. The memory stores the results and future calculations.

Memory

Memory is our main storage for our program code, graphics data and values such as our score, the XY position of our player and everything else!

Each byte of memory has a numeric 'address'. Think of it like a huge set of lockers, each has a number and each 'locker' contains a byte of data which we can read or write.

The 'Address bus' is a set of wires that connects the CPU to the Memory.

Though its calculations are 8 bit, a Z80 system has a 16 bit 'address bus' so it can address 64K of memory (0-65535), though this can be extended with 'Bank Switching'.

Bank switching is where part of the memory map will 'switch' between different areas of memory. On the 128K CPC, the range &4000-&7FFF can switch between one of the 8 available 16k banks, giving our program access to more memory within the 64K address space.

Figure 4: Memory is like a bank of numbered 'lockers' that can each store 1 byte messages.

RAM and ROM!

There are two kinds of memory:
RAM is 'Random Access Memory', this just means it can be read or written.
ROM is 'Read only memory', like a CD ROM we can't change its contents.

Systems like the CPC and Spectrum have lots of RAM, and a bit of ROM for things like screen and tape routines.

On Systems like the Sega Master System and NeoGeo, the Cartridge which stores our program is read only ROM, and the game system has a small amount of writeable RAM for our variables and other stuff.

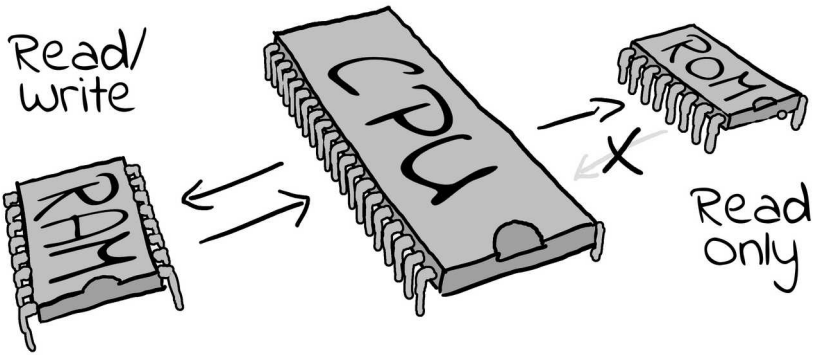


Figure 5: RAM can be read or written, ROM can only be read.

Registers

The CPU has a very small amount of built in memory to remember parameters for the calculations, usually just a few bytes (depending on the system).

This is 'short term storage'. It's much faster than RAM because it's inside the CPU but we have very few registers, so we have to use the slower normal memory a lot.

Systems like the 6502 have just 3 main registers, and systems like the Z80 have over a dozen. That's not to say the 6502 is 'worse', it just works differently. Though the 6502 has fewer registers it can access memory faster instead.

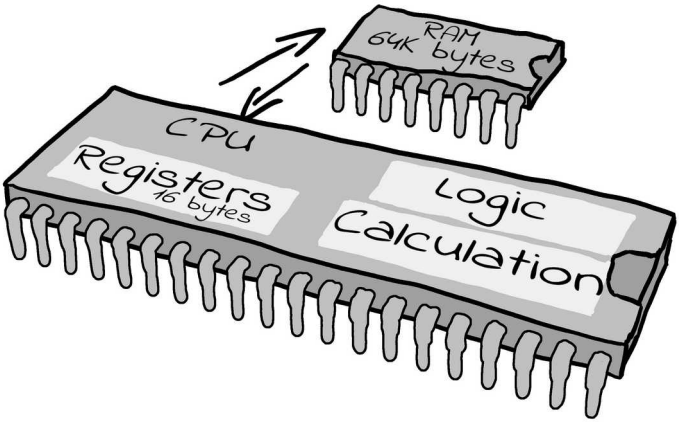


Figure 6: Registers are 'short term memory', just a few bytes are available. RAM is 'Long term memory' many Kilobytes or even Megabytes will be available.

Accumulator Register (A)

The Accumulator is the main register for calculations. It can only store one calculation result at a time, you can think of it like the screen on your calculator. The register is referenced by the letter 'A'.

Old CPU's, like the 6502 and Z80, had a single Accumulator.
More modern ones, like the ARM/68000, don't work like that. On the 68000 or ARM any register of the half dozen or so can do the calculations, so there's no single 'Accumulator'.

Program Counter Register (PC) / Instruction Pointer (IP)

The code of our program will be somewhere in memory. The Program Counter points to the current address of the line of code the CPU is running.

It's like following a 'to do list' and pointing your finger at each line as you move through the list. Without the program counter the CPU wouldn't know what bytes to read in next to work on.

Figure 7: The Program Counter points to the current 'job' being worked on.

Flags Register / Condition Codes

Whenever we do a calculation, the flag register will store the 'answers' to some questions we may ask.
For Example:
Z Flag Was the result Zero?
V Flag Did the result go too high to store correctly (oVerflow)?
C flag Did a calculation transfer data out of a register (Carry)?

There are also some special 'Flags' which change the way the CPU works, turning things on like 'Binary Coded Decimal' (a special calculation mode) or disabling 'Maskable Interrupts' (stopping the Operating System taking over the CPU).

Flags are called Condition Codes on some CPUs and are held in the 'Condition Code Register' (CCR).
Not all commands set the flags, you will need to check the instruction documentation to know if they do. Also some commands may leave a flag in an 'undetermined state', meaning it's changed but not predictable or useful.

We can take advantage of unchanged flags. On the Z80, "DEC A" changes the z flag, but "LD" does not. We can take advantage of this in our program code.
For Example: Consider the code "DEC A LD A,10 jr z,AccZero". This takes advantage of the fact "LD A,#" does not change the z flag. If DEC A sets the zero flag, the jump will occur, and the Accumulator will equal 10 after the jump.

In the instruction references of this book, the 'Flags Affected' section will show a minus '-' when an instruction leaves a flag unchanged, an uppercase letter when a flag is correctly updated (for example 'C'), and a lowercase letter when a flag is changed to an undetermined state (for example 'c').

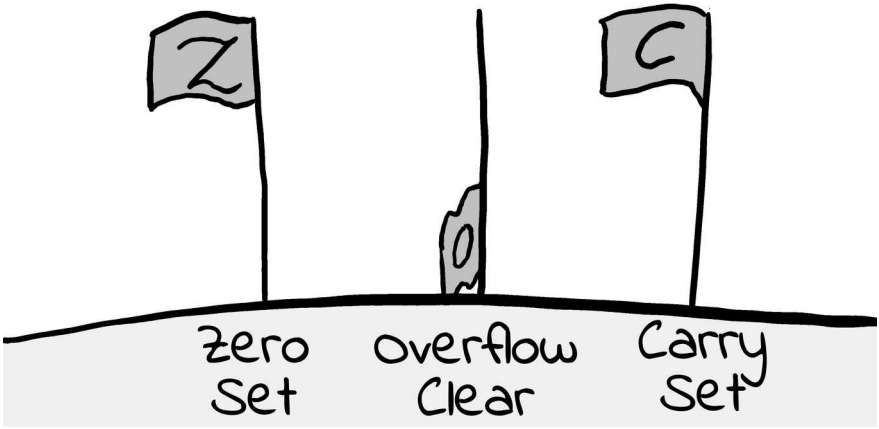


Figure 8: Flags can be set (1) or Clear (0).

The Carry Flag

The Carry flag is quite important on 8 bit systems, as it allows us to combine registers together to store larger values than the resister can contain. It works in a similar way to 'carrying' if you do long addition or multiplication on paper.

For Example: On the 8 bit 6502, suppose there is a 16 bit pair in $01,$02 and we want to add the pair at $03,$04. We do this by adding the two low bytes together ($01 and $03) then adding with the carry the high bytes ($02,$04).

Because the 6502 is a Little Endian CPU, $01 and $03 are the Low byte and $02 and $04 are the High byte.

If the first addition goes over 255, the carry will be set to 1, and this will be added to the high byte.

```
;Equivalent of ADD HL,DE
    clc         ;Clear the carry
    lda $01     ;Load Low Byte of first parameter (L)
    adc $03     ;Add Low Byte of Second Parameter (E)
    sta $01     ;Store Low Byte of first parameter (L)

    lda $02     ;Load High Byte of first parameter (H)
    adc $04     ;Add with carry High Byte of Second Parameter (D)
    sta $02     ;Store High Byte of first parameter (H)
```

Figure 9: The 6502 uses 8 bit registers, but we can use the 'Carry flag' to transfer the 'carry' during addition between two or more bytes to allow us to support 16 or more bit values.

I/O Ports – Input/Output Ports

There will be times we need to transfer data to a device other than RAM memory.
For Example: We may want to read the status of the Joysticks, or make a sound, and we use I/O ports to do this.

On the Z80 and 8086 system we have special commands to do this.
On systems like the 6502 and 68000 these are 'memory mapped' and appear as an address in our normal memory range.

Things can be very different depending on the computer. On the Z80 based CPC the VRAM (Video Ram) is part of the normal memory, but on the Z80 based MSX it is separate and we have to use I/O ports to access it.

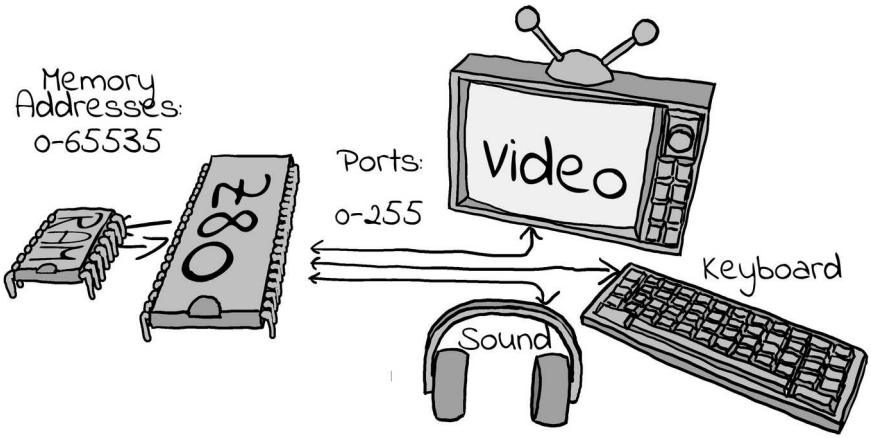


Figure 10: The RAM addresses and I/O ports are separate.

Stack

The stack is a temporary store for data. We don't have very many registers, and there will be times (like in a subroutine) where we need to put all the current data to one side and do another job for a while, then bring all the old data back. This is what the Stack is for.

When we have some data we need later, but need to do something else first, we put it on the stack. Later we'll take it off and use it again.

You can think of the stack like your office in-tray. We can put jobs in our 'in tray' when we want to do them later, and pull them out when we want to work on them again. We can put as many items into our in-tray as we want, but we have to take them off the top of the in-tray, not from the middle.

The same is true of the stack. We can put many items onto the stack, but they always go on top of the last one, and we have to remove them in the reverse order, taking out first the last

item put in. This is called a Last In First Out stack. This means we put items 1,2,3 onto the stack and we'll take them back in order 3,2,1.

The current position in the stack (the top of our in-tray) is marked by the Stack Pointer (SP).

Technically speaking, the stack actually goes DOWN in memory. If the Stack Pointer starts at $C000 and we put (PUSH) two bytes onto the stack, the Stack Pointer will point to $BFFE. When we take the bytes off (PULL/POP) the Stack Pointer will point to $C000. The Stack Pointer always points to the first empty byte of the stack.

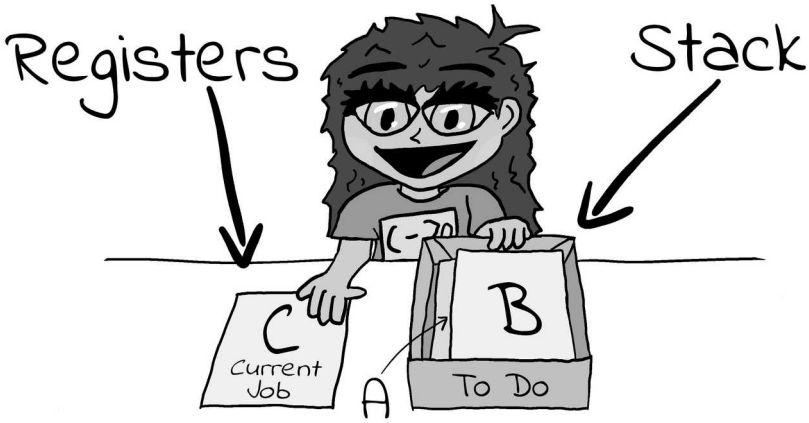


Figure 11: Items we'll look at later go on the top of the stack.

Interrupts

Interrupts are exactly what they sound like! Sometimes a device, Like a disk or mouse, will want to send some data to be processed RIGHT NOW!

The CPU will have to stop whatever it was doing and deal with the data. The current running program will be interrupted, and a 'sub program' will run (usually in system ROM) to deal with the interrupt. When the interrupt is done, the original running program will resume as if nothing happened.

On 8 bit systems we can turn interrupts off in many cases. On most systems we can create our own interrupt handlers to do clever things, like make the screen wobble by moving things as the screen is drawing.

There are two kinds of interrupt:
Maskable interrupts are interrupts that can be disabled (sometimes called IRQ – Interrupt ReQuest). For Example: IM1 calls to Address $0038 on the Z80 can be disabled by DI.
Non-Maskable Interrupts (NMI) cannot be disabled. On the Z80 NMI interrupts result in a CALL to address $0066.

Figure 12: NMI can't be stopped by Disabled Interrupts (DI).

Endian, Big Endian and Little Endian

Even on an 8 bit system there will be times when we need to store a 16 bit or more number, and whether our CPU is 8,16 or 32 bits, each memory address in RAM will only store 1 byte, so we'll need to split larger numbers up into individual bytes for storage.

Let's imagine we have 32 bits to store, this will take four bytes of RAM. But in what order should these four bytes be stored? It may be surprising to hear that there are two options.
'Big Endian' stores the highest byte first and the lowest byte last.
'Little Endian' stores the smallest byte first, and the largest byte last.

It's not really up to us what 'Endian' to use, as our CPU will have its 'Endian' built into its memory addressing.
Little Endian is used by the Z80, 6502, 8086, PDP-11 and ARM*.
Big Endian is used by the 68000, 6809 and TMS9900.

Fun Fact:
The terms 'Big Endian' and 'Little Endian' actually come from the fictional nation of "Lilliput" in "Gulliver's Travels" which split in two factions over from which 'end' an egg should be eaten! Take a look at this link for more details:
https://en.wikipedia.org/wiki/Gulliver%27s_Travels

*The ARM can theoretically be configured as Big Endian, but this is not normally the case.

Let's store 16 bit number $ABCD at address $1000

	$1000	$1001
Big Endian	$AB	$CD
Little Endian	$CD	$AB

Let's store 32 bit number $12345678 at address $1000

	$1000	$1001	$1002	$1003
Big Endian	$12	$34	$56	$78
Little Endian	$78	$56	$34	$12

Figure 13: Big and Little Endian compared.

RST / Traps

RST (ReSeT) and Traps are special 'commands' which cause a special subroutine to run, What these do depends on the machine.

RSTs and Traps often relate to Interrupts, but Traps are also used for debugging and error handling.

On many Z80 systems RST7 occurs when the screen starts drawing.
On the 68000 Traps are also used for errors (like Divide by zero) and even help with debugging! There is a special "Step" trap which will occur after each command so we can check what's happening.

Privilege Modes

In the 8 bit days, CPUs were pretty basic, and user applications and the operating system both had the same power and abilities, but the later CPUs like the 68000 had 'Privilege' modes.

On 16 and 32 bit CPUs Privilege would often be split into 'User' and 'Supervisor'
User mode would be for the running program.
Supervisor would be for the Interrupts and other system tasks. There are often 'alternate' registers and commands available only to supervisor mode. It's quite probable there will be many more levels of privilege on more modern CPUs.

Zero Page / Direct Page

On 8 bit CPUs with few registers, like the 6502 or 6809, the lack of registers is compensated for by the 'Zero page'.

This uses a block of 256 bytes for quick storage of values. Reading and writing to this range is faster than regular memory. The Zero Page uses the memory range $0000-$00FF. Rather than specifying a full address we just specify a single byte, so a command like "LDA $66" will load A from address $0066. The top byte is always Zero, hence the name!

On later systems like, the 65816, the top byte of the resulting address was configured by a one byte 'Direct Page Register', so the 'Zero Page' is referred to as the 'Direct Page' on these systems, though its function is basically the same.

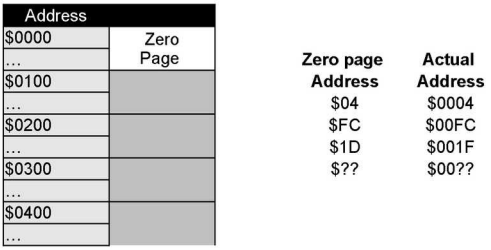


Figure 14: The Zero Page uses the first 256 bytes of RAM. Zero page addresses are specified with a single byte.

Segment Registers / Bank Registers

These only exist in CPUs with a 20/24 bit address bus, like the eZ80, 65816 or 8086. These have a 24 bit address bus, but keep their original 16 bit registers. This leaves a 'shortfall' of 8 bits for address calculation.

The solution to this is often an 8 bit register defining the top byte. This register is known as a 'Segment Register' on the 8086, or 'Bank register' on the 65816 and 'Mbase' on the eZ80.

The Segment register is actually 16 bit on the 8086, we'll cover it later in the 8086 chapter.

Segment Registers allow extra memory to be used, while retaining compatibility with the old programming methods. However they are more limited than a true 24 bit addressing system like the 68000 CPU.

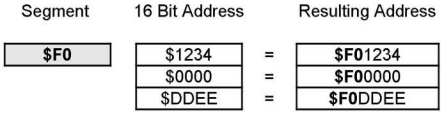


Figure 15: An 8 bit segment register is added to the top of a specified 16 bit address to make a 24 bit one.

Segments also relate to Logical and Physical addresses. On a CPU like the 65816, Our 64K program may only be able to see a 16 bit, 64K of memory with memory addresses $0000-$FFFF. In this case this would be referred to as the programs 'Logical Address'.

However, that range will be part of a much larger range of the actual resources the system offers. For Example: The true 24 bit address of that memory could be $7F0000-$7FFFFF. This 'True Address' is the 'Physical Address'.

Base Pointer + Offset

There are many times where the source or destination address for a command will be calculated from two separate values. The terminology may vary depending on CPU, but for this example we'll call them a Base and an Offset.

This is where the destination address will be calculated from two parts. The first may be a register (the base), and the second may be a fixed immediate number (the offset). The actual address used by the command will be the base plus the offset.

The advantage of this is improved efficiency. If we have 20 commands that use addresses in the range $200000-$200010, we can set a register to the 24 bit base address $200000, and specify the 8 bit offsets in the 20 commands, saving memory and time.

Also, if we later need to use the same offsets in the range $300000-$300010, we can just change the base register and reuse the code.

The above example is just an example. The concept of Base+Offset could be done by the processor with Index Registers on the Z80, the 6502 Zero page, or in software by our code.

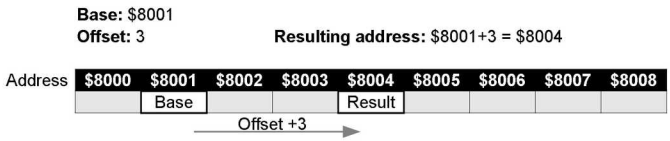


Figure 16: Using a Base plus an Offset, the resulting address used in the operation will be the sum of the two.

Index Registers

Many systems allow addresses to be specified with offsets.

For Example: The Z80 has IX as an Index register. Let's look at the Z80 command "LD A, (IX+7)" which sets the Accumulator. If IX=10 then the Accumulator will be read from address 17.

The advantage of this is that code using index registers can be 'reused' easily. If we have a 'ShowSprite' function that uses IX, we can point IX to each of our objects we want to show, and the code can be designed to find the relevant settings for that object as relative offsets.

The 6502 also has registers for this purpose. X and Y are used for similar purposes.

RISC and CISC CPUs

RISC stands for "Reduced Instruction Set Computer", CISC stands for "Complex Instruction Set Computer".
RISC processors tend to need more commands to do a single job than their CISC counterparts, but those commands will tend to occur faster, and the processor will be relatively simpler, meaning it's probably more power efficient.

Examples of RISC are the ARM, RISC-V, MIPS and POWER PC.
Examples of CISC are the Z80, 68000 and 8086.

These terms aren't particularly helpful to the programmer, especially as the "RISC ARM" instruction set is more advanced than the "CISC Z80", they are just included in here to explain the term.

If you find yourself looking at ARM or RISC-V asking yourself "why isn't it as easy to do X as on the 68000?" then the answer is probably "Because it's a RISC CPU!"

Data Representation Terminology

There are various forms of representation for numbers we'll want to know about. The representation we'll use in our code will vary depending on how we want to use it. We'll also need them to read documentation and manuals on the hardware we need to use.

Dec	Hex	Oct	Binary	Asc
0	00	000	00000000	
1	01	001	00000001	
2	02	002	00000010	
3	03	003	00000011	
4	04	004	00000100	
5	05	005	00000101	
6	06	006	00000110	
7	07	007	00000111	
8	08	010	00001000	
9	09	011	00001001	
10	0A	012	00001010	
11	0B	013	00001011	
12	0C	014	00001100	
13	0D	015	00001101	
14	0E	016	00001110	
15	0F	017	00001111	
16	10	020	00010000	
17	11	021	00010001	
18	12	022	00010010	
19	13	023	00010011	
20	14	024	00010100	
21	15	025	00010101	
22	16	026	00010110	
23	17	027	00010111	
24	18	030	00011000	
25	19	031	00011001	
26	1A	032	00011010	
27	1B	033	00011011	
28	1C	034	00011100	
29	1D	035	00011101	
30	1E	036	00011110	
31	1F	037	00011111	
32	20	040	00100000	
33	21	041	00100001	!
34	22	042	00100010	"
35	23	043	00100011	#
36	24	044	00100100	$
37	25	045	00100101	%
38	26	046	00100110	&
39	27	047	00100111	'
40	28	050	00101000	(
41	29	051	00101001	)
42	2A	052	00101010	*
43	2B	053	00101011	+
44	2C	054	00101100	,
45	2D	055	00101101	-
46	2E	056	00101110	.
47	2F	057	00101111	/
48	30	060	00110000	0
49	31	061	00110001	1
50	32	062	00110010	2
51	33	063	00110011	3
52	34	064	00110100	4
53	35	065	00110101	5
54	36	066	00110110	6
55	37	067	00110111	7
56	38	070	00111000	8
57	39	071	00111001	9
58	3A	072	00111010	:
59	3B	073	00111011	;
60	3C	074	00111100	<
61	3D	075	00111101	=
62	3E	076	00111110	>
63	3F	077	00111111	?

Figure 17: 0-63 in the common bases. See the Appendix for full 0-255 range.

Decimal

Decimal is what we're used to - the 0-9 numbers on our clock, our receipts, and our normal calculator.
It's known as Base 10, as each 'digit' has a value of 0-9.

Binary

Binary is Base 2. Each digit can only be 1 or 0.
"01" in binary is 1 in decimal, "10" in binary is 2 in decimal, "100" is 4, "101" is 5 and so on.
This works better for computers, which tend to work in only 'Off' or 'On'.

In Assembly, Binary is often shown starting % . You may see 2 in binary shown as "%00000010", though other assemblers used different terminology, so you may see the same value as "00000010b" or "0b00000010".

Hexadecimal

As computers work in Binary, it's hard to make 10 or 100 out of base 2. Hexadecimal is the way computers combine binary bits into digits we can easily use in our source code.
Hexadecimal is 'Base 16'. It uses the normal digits from 0-9, then uses letters 'ABCDEF' as 'digits' for 10-15.
"&10" in hexadecimal is "16" in decimal!

Hexadecimal is often shown starting with a "$" or "&", or sometimes "0x", or ending in "h".
You may see the Decimal value 31 shown in hex as "$1F", "&1F", "0x1F", or even "01Fh".

Octal

Octal is Base 8. It's not really used any more, but some old computers like the PDP-11 use it, so it's worth remembering the name as you'll hear it from time to time.

ASCII

This is what you're reading right now! It's just regular letters, numbers and symbols.
ASCII stands for "American Standard Code for Information Interchange".
ASCII defines the first 128 characters in the character set, but many computers allow 256. The second 128 are different on each system.

Note: some systems, like the C64, do not use ASCII, they have a different character set. As our assembler will probably convert letters to ASCII bytes, this may mean strings of text we put in our ASM code do not appear correctly on screen. We will either need to use an assembler that can work with this, or to write our code to convert ASCII to the matching letters on the system.

Byte (Bytes)

A byte is 8 bits. It can go from 0 to 255 in Decimal, which is $00 to $FF in Hexadecimal, or %00000000 to %11111111 in Binary.
A signed byte goes from -128 to +127 in Decimal.

A byte is the smallest unit of memory in a system. Registers on an 8 bit system work in bytes, and each CPU memory address refers to a byte of data. Address $0000 is the first byte of memory, and $0001 is the second, and so on.

Kilobyte

A Kilobyte is 1024 bytes. Why isn't it 1000? Well, because everything in computing is binary, 1000 wouldn't be very convenient. 1000 in hex is $3E8, but 1024 is a much more tidy $400!

Our 8 bit machine with its 16 bit address bus has a memory limit of 65536, 64 Kilobytes (64K).

Bit

A bit is a single binary number 1 or 0, there are 8 per byte.

Bits in a byte or word are numbered backwards from right to left.
In a byte, Bit 0 is the least significant (with a value of one), Bit 7 is the most significant (with a value of 128).

Example 16 bit Word (2 Bytes)… Range 0-65535

Bit Number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Figure 18: Numbered Bits in a Word and their respective value. You'll need to know the position numbers for bit testing commands, and reading hardware documentation.

Nibble (Nybble)

A Nibble is half a byte, that's 4 bits.
Note: There are always two in a byte, there is no such thing as a system that has a 1 nibble register, or 1 nibble per memory address.
The unit 'Nibble' is of course a pun on Bite and Bit, disappointingly there is no such unit as a 'Munch'!

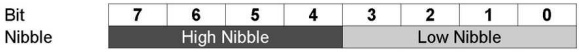


Figure 19: Nibbles in a byte. Nibbles are always 4 bits, so there are two per byte.

Word

The size of a word depends on the system. On 8 and 16 bit systems (Z80/6502/68000) it's 2 bytes, but on 32 bit systems, like ARM, it's 4 bytes.

A word on an 8 or 16 bit system goes from 0 to 65535, or $0000 to $FFFF in Hexadecimal.
A signed word on an 8 or 16 bit system goes from -32768 to +32767.
A word on a 32 bit system goes from 0 to 4,294,967,290, or $00000000 to $FFFFFFFF in Hexadecimal.

Long / Double Word

A Long is two words. On 8 and 16 bit systems this is 4 bytes. On systems like ARM it's 8 bytes.
A Long on an 8 bit system goes from 0 to 4,294,967,295, or $00000000 to $FFFFFFFF in Hexadecimal.
A signed Long goes from -2147483648 to +2147483647 .

Integer

An integer is a number with no decimal places.
1234 is an integer, 1234.2 is not.

Floating point number

Essentially a floating point number has decimal places. 1234.2 is a floating point number, 123 is not.
It's referred to as 'floating point' because the number is represented by a 'significand' and an 'Exponent', which define the basic number and the position of the decimal point.

These are complex and slow for classic computers to work with, so we usually want to find a way to work which only needs integer maths.

Signed Number

A signed number is a number that can be positive or negative.
An Unsigned 8 bit number can go from 0 to 255 but a signed number can go from -128 to +127.

The way signed numbers work in assembly is odd, and exploits a 'quirk' of the way the processor registers work.

A byte value in an 8 bit register can only go from 0 to 255. If we add a large number to a byte (for example adding 255 to 128) the byte will 'Overflow' going over up to 255, and wrapping back to 0. Therefore adding 255 to an 8 bit register is the same as subtracting 1!

So 255 is effectively -1, 254 is effectively -2 and so on.
The processor doesn't 'know' whether the 255 in a byte is -1 or 255 and it doesn't need to. We use special Condition statements for signed and unsigned numbers, and we code according to whether our byte is signed or not.

Don't worry if that sounds odd, it will make more sense later!

Converting a positive to a negative is easy. In code we flip all the bits via an XOR / EOR or other special command (like CPL), and add one to the result with an INC or ADDQ (or equivalent). This is known as Two's Complement.

For Example: Let's look at converting 1 to -1. Our register starts with a decimal value 1. After the bits are flipped and one is added this will result in a decimal value of 255 ($01 converted to $FF in hexadecimal).

On our calculator we take the maximum value plus 1 (256 for a byte, 65536 for a word) and subtract the number we want to negate.

Binary Coded Decimal

Working in hexadecimal is great for mathematics and calculating memory addresses, but it's difficult for showing decimal numbers on screen.

Suppose we use a 16 bit number for our 4 digit game score, how can we easily split the score 65535 into 4 decimal digits quickly for display? Unfortunately we can't! It would be slow and take lots of divides. This is where Binary Coded Decimal (BCD) comes in!

With normal register values, if a byte has the value $73 in hexadecimal, it's decimal value will be 115.

In Binary Coded Decimal, the value $73 in a byte is actually the value 73 in decimal! This makes it very easy to convert numbers for display.

We've effectively ignored the A-F part of the hex numbers. Binary Coded Decimal wastes some memory as now a byte can only store 0-99, but it's a much easier way to store data to allow us to show decimal numbers.

There are two Binary Coded Decimal formats:
Packed format stores two digits per byte. In this format decimal 1234 would be stored $12 $34. This is the most common BCD format.
Unpacked format would be 4 bytes – so it would be stored $01 $02 $03 $04.

Absolute Addresses and Relative Addresses

In our code there will be many times we will need to specify addresses, either for reading or writing data, calling subroutines, or branching on conditions. In 8 bit code, these will tend to be Absolute Addresses - addresses at a fixed location.

Relative addresses are specified as an offset to the current position in code (the PC register). These are only really used for jumps and branches in 8 bit code (not subroutines). 16 bit systems use relative code more often, as their programs are often 'relocatable' in memory.

Examples of Absolute addressing are:

Function	Z80 Version	6502 Version
Load a value from address $1000	LD A,($1000)	LDA $1000
Save it to $2000	LD ($2000),A	STA $2000
Jump to address $3000	JMP $3000	JMP $3000

Examples of Relative addressing are:

Function	Z80 Version	6502 Version
Jump to the address 16 bytes away if Z flag is set	JR Z,16	BEQ 16

Assembler Terminology.

Binary file

A binary file is pure data, the content could be any kind of data. Binary files could be an image, some sound or even a program our retro computer can run. We'll need to know what to do with it to make use of it, a sound file copied to video RAM won't help much!

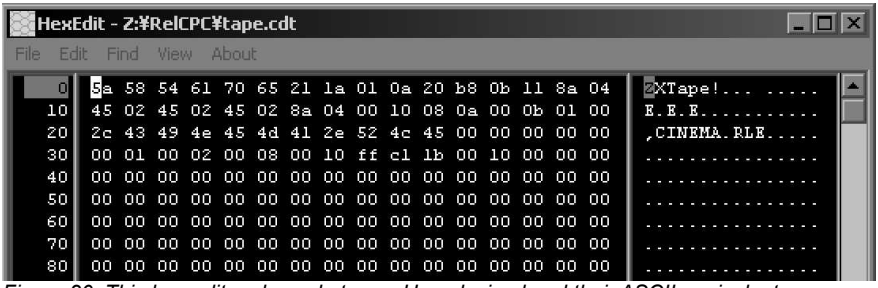


Figure 20: This hex editor shows bytes as Hexadecimal and their ASCII equivalent.

Assembly Source file

The source file is a text file which contains the commands that make up our program, usually with an ASM file extension. We can't run an ASM file on an emulator or computer, we'll need to convert it to a binary file with an Assembler.

We can edit the file with whatever we prefer, Notepad, Notepad++ or Visual Studio Code, it's the Assembler that does the 'real work' of making a runnable program.

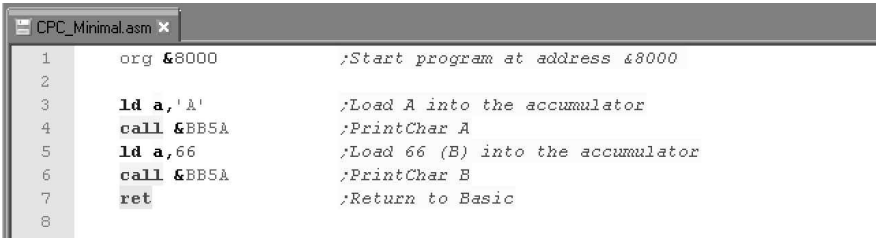


Figure 21: An ASM source file being edited with Notepad++

Assembler

The Assembler will take a source file (usually with an ASM file extension) and convert it to binary data.

On a cartridge based system we may end up with a usable game once it's assembled, but on home computers we may need to do more work, like add it to a disk or tape image for our emulator to run.

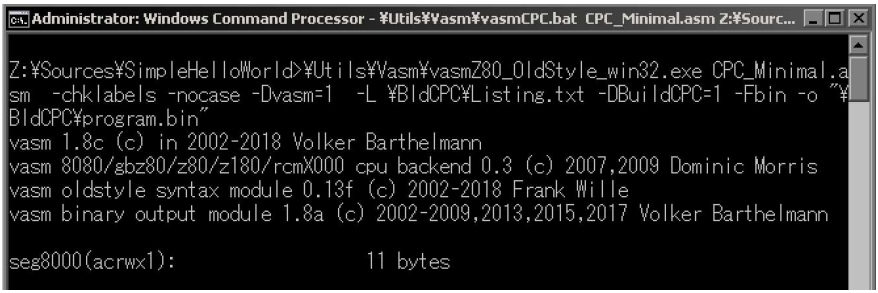


Figure 22: VASM and most other assemblers are command line tools.

Disk Images and Tape Images

Our home computers probably can't run a binary 'as is', we'll need to make it into something the computer can use.

We'll need to put our binary onto a 'Tape' or 'Disk'. We won't use a real one though, we'll use a fake 'Disk' or 'Tape' image.

We'll need special software to make this Disk or Tape image. Unfortunately, Disk, Tape and Cartridge formats are platform dependent, there's not 'one solution', so you'll need to check the documentation relating to the system you're interested in.

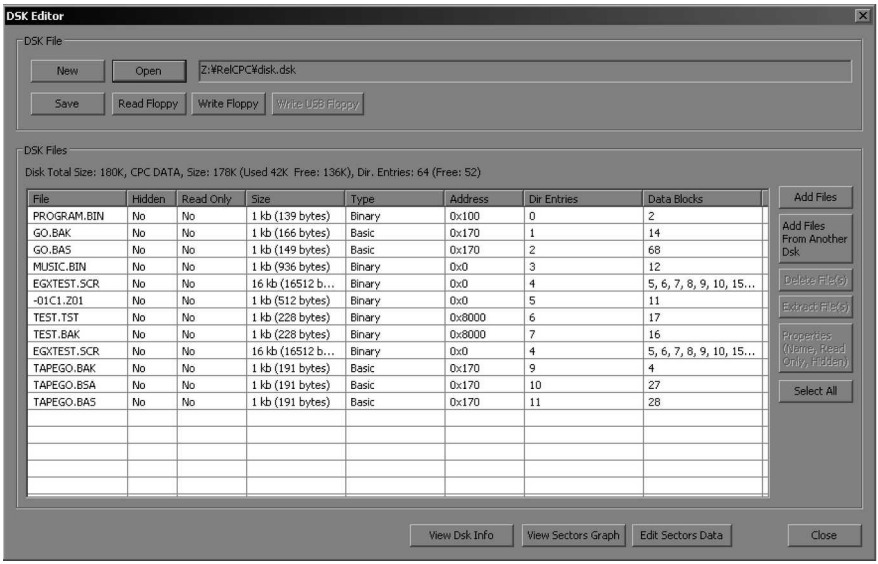


Figure 23: Amstrad CPC Disk Image Editor: CpcDiskXP. Unfortunately you typically need a different disk image editor for each system.

Immediate Value

An 'Immediate' is a fixed number parameter specified in the assembler source after the command.
For Example: Let's look at the commands "LD A,3" (Z80) or "LDA #3" (6502).
In these examples the value 3 is an 'Immediate'.

Other commands like the commands, "LD A,(3)" or "LDA 3", are loading from the address '3', so these are not Immediate values.

Don't worry if you can't tell Immediate values from values loaded from addresses at this stage, as the syntax varies depending on the assembly language you're using.

You'll notice on the 6502 and 68000, Immediate values start with a "#", and if you forget that symbol the value will be be treated as an address. This is a common mistake you're likely to make, so it's worth double checking when things go wrong!

Label

A label marks part of the code which we will refer to later, it's like a line number in Basic. Usually it's a destination for a jump (a command like GOTO in Basic) or some data we'll read or write elsewhere in our code (like sprites or variables).

Typically (unlike command opcodes) labels have to be at the far left with no tab indent, and have a colon after the label.

```
PrintString:

    ld a,(hl)           ;Get A char
    cp 255              ;Is it a 255?
    ret z               ;Yes? then return
```

Figure 24: A label called "PrintString" - we'll CALL this later in our program. Note the label is at the far left, but the code is indented by at least one tab.

Operator

An operator is a 'command' like ADD or SUB. The abbreviated commands used in Assembly are called 'OP Codes'
Commands in Assembly typically need to be 'indented' with a tab to identify them as not being labels, though this can vary depending on your assembler.

```
    ld a,(hl)                   ;Read a byte from HL Address
    inc hl                      ;Increse HL address
    out (TI_LCD_DATA),a         ;send the byte to the GPU
```

Figure 25: LD, INC and OUT are operators.

Operand

An operand is a parameter, often a Register or an Immediate value.

```
    ld a,(hl)                   ;Read a byte from HL Address
    inc hl                      ;Increse HL address
    out (TI_LCD_DATA),a         ;send the byte to the GPU
```

Figure 26: A, HL and TI_LCD_DATA are operands - the parameters of the operator.

Assembler Directives

Assembler directives are commands which instruct the Assembler to do something. These are not converted directly to command bytes for the CPU, instead they change the function of the Assembler and tell it how to compile the code.

There are a wide range of compiler directives, and their format will vary depending on the Assembler, but some examples are: "EQU" (Symbols) "IFDEF" (Conditional Compilation) "ORG" (tells the Assembler the address the code will run from) ".186" (tells the Assembler what version CPU we're compiling for) and "DB" (Defined Bytes of data).

You'll need to check your Assembler manual to know the commands available to you, and you'll probably only need a few of the wide range available. We'll discuss some of the general ones you're likely to need throughout this chapter.

```
1  LIST            ;Directive telling the assembler to output a listing file
2
3  ScreenMode equ 3        ;Directive defining Screenmode as 3
4      ORG &1000           ;Directive defining that the code origin is &1000
5      jp ProgramStart
6      db "File Header"    ;Directive defining a Byte sequence
7  ProgramStart:
```

Figure 27: There's a huge range of Assembler directives, depending on your CPU, Assembler and syntax choice. The only thing you can do is check your Assembler's manual.

Symbol

A symbol is a fixed value (it doesn't change during our code). "PI EQU 3.1415926" sets the symbol "PI" to the number 3.1415926. We can now use PI in our code, rather than typing all those numbers again.

It's the assembler that converts the symbol, the binary file will not change whether we use the number or the symbol in our code.

EQU stands for 'EQUate' or 'EQUivalence', it tells the assembler the symbol has the same value as the number which follows. The exact syntax of the command varies depending on your assembler.

In VASM, EQU statements have to be at the far left like labels.

```
TI_LCD_COMM equ &10
TI_LCD_DATA equ &11
```

Figure 28: Two Symbols defined for the TI graphics ports.

Code Comments

There will be times we will want to put notes within our code so we can remember how it works. These are known as 'Comments' or 'Remarks' (REM statements).

While the syntax varies depending on your assembler, comments in assembly usually start with a semicolon (;). They can be on a line on their own, or at the end of a line of code. After the semicolon, the assembler will ignore the rest of the line, so adding a semicolon to a line of code will quickly disable it.

These are totally optional, they make no difference to the resulting program, but you will probably find them essential to help make the function of the code you're writing clearer, as

when you're debugging it in a few weeks, or trying to reuse part of your old code many months later, you'll benefit from having spent the extra time to add at least a few comments to your code.

```
1  ;This is a code comment - it starts with a semicolon
2
3      ld a,128        ;This is also a comment - it's after a command
```

Figure 29: Code comments can be a whole line, or after a command. On most assemblers a comment starts with a semicolon (;).

Conditional Compilation

There may be times when we want to build multiple versions of our program from the same source.
For Example: Maybe we want two versions of our game, a 'Trial Version' and a 'Paid version' with more features.

We don't want to keep two separate copies of the source files, as this would increase our developing and maintenance time, instead what we do is use 'Conditional Compilation'.

By using assembler directives, such as "IFDEF mysymbol" (IFD on 68000) and ENDIF, we can define blocks of code which will only compile if a symbol is defined.

By defining only certain symbols, we can enable and disable these blocks, changing the code that is compiled, and the resulting program. Enabling and disabling symbols can be done by simply putting a semicolon (;) at the start of the line, turning them into comments, or symbols can often be defined on the assembler command line, meaning we can run different scripts to build different versions of our code.

```
1  IsBeta equ 1        ;Rem this out when not beta
2
3      org &1000           ;Start program at address &1000
4
5      ifdef IsBeta            ;Is IsBeta defined? (any value)
6
7          ld hl,BetaMessage   ;Show beta message for pre-release
8          Call PrintString
9      else
10         ld hl,SupportMessage    ;Show support message when out of beta
11         Call PrintString
12     endif
```

Figure 30: Conditional compilation allows us to make it possible to have one source file that can have different build versions.

Macro

A Macro is a bit like a symbol. It defines a set of commands that are given a 'name'.
We can then use that name in our code and the assembler will replace it with all the commands we defined.

For Example: We could create a macro that automatically prints a letter and call it PRINTCHAR. We can then use PRINTCHAR 'A' or PRINTCHAR 'B' in our code.

The Assembler will use our definition to produce the resulting program with the contents of the macro replacing this.

It's a bit like making our own commands. It saves us copying the same code over and over and it saves a bit of time, rather than writing a subroutine.

The syntax of a macro definition depends on the assembler. You'll need to check the documentation of your Assembler.

```
macro z_ld_iyh_l
    push af
        ld a,l
        ld (r_iyh),a
    pop af
endm
```

Figure 31: We've defined a macro "z_ld_iyh_l". When we use this name the assembler will 'put in' all the commands we specified here.

Disassembler

The Assembler converts a source file to a binary so we can run it, Binary files don't make much sense to humans though and we may want to see how a program works that we don't have the source for. That's what a Disassembler does!

The Disassembler takes a binary file and converts it to a Source File.

Unfortunately it doesn't do a perfect job. Label names, Symbols and Comments are not included in the binary, so these are all lost. Also it can be hard for the Disassembler to work out what parts of the binary are program code and what parts are images, sound or other data, and it may mix the two up.

If you have a "Symbol File" from the assembly stage, the Disassembler will be able to recover these label and symbol names, which can help for real time disassembly of a program while it's running.

```
0x00005c44      call fcn.00007949.32bitMult
0x00005c47      ld [0x4d31], hl
0x00005c4a      ld a, 0x01
0x00005c4c      bit 7, d
0x00005c4e      jr nz, 0x02
0x00005c50      ld a, 0xff
0x00005c52      ld [0x4d36], a
```

Figure 32: A disassembled file - but the labels have been lost for the addresses

Linker

Depending on your destination file format and Assembler, assembly may not be enough to produce a runnable program.

A linker can take multiple files, combine them together and produce a runnable file.

Debugger / Monitor

A Debugger is a tool which helps us figure out what is happening in our program, typically when there's a 'Bug' (something going wrong).

They will often show the contents of the CPU registers, the running code (via disassembly) and allow us to see parts of the memory.

They may also allow us to 'step' through the code, running just one line at a time to see what's really happening.

Debuggers are also called 'Monitors', in the sense they monitor the running state of the code.

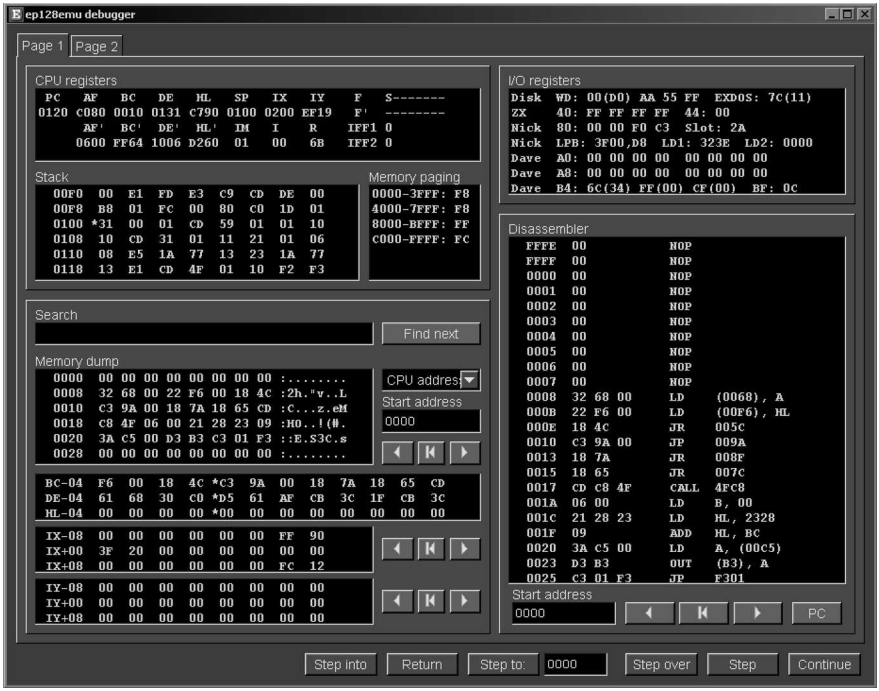


Figure 33: The very comprehensive debugger of ep128emu. Not all emulators have such good functionality!

Symbol file

A symbol file contains the names of Labels and Symbols, with the text name and resulting byte value.

A symbol file can be used with a Disassembler or debugger to keep the label names in disassembled code.
These are not needed, and your assembler won't typically output one by default, but they may help for debugging or disassembly.

CPC_SuckHunt.sym

```
1  a_bullets #0000
2  a_magazine #0001
3  animatedeadsprite #367E
4  animatedeadsprite_background #36EB
5  animatedeadsprite_corona #36B6
6  animatedeadsprite_enemy1 #369C
7  animatedeadsprite_enemy2 #36A9
8  animatedeadsprite_powerup #36C3
9  animatedeadsprite_reanimated #36F7
```

Figure 34: The optional Symbol file. This tracks all the values that the symbols and labels ended up with.

Listing File

A listing file is a summary of how the ASM source converted to output bytes, it also typically includes the symbols like a symbol file.

Assemblers don't usually output these by default, but they can be essential to figure out what's going wrong when things don't happen how you expect.

Listing.txt

```
1  F00:0001        org &8000           ;Start program at address &8000
2  F00:0002
3  F00:0003        ld a,'A'            ;Load A into the accumulator
4                  S01:00008000:  3E 41
5  F00:0004        call &BB5A          ;PrintChar A
6                  S01:00008002:  CD 5A BB
7  F00:0005        ld a,66             ;Load 66 (B) into the accumulator
8                  S01:00008005:  3E 42
9  F00:0006        call &BB5A          ;PrintChar B
10                 S01:00008007:  CD 5A BB
11 F00:0007        ret                 ;Return to Basic
12                 S01:0000800A:  C9
```

Figure 35: A Listing file, the source code and the resulting bytes are shown.

Defined Data

There will be times when we want to put sections with byte data in our code. These could be values for lookup tables, addresses for indirect jumps, or bitmap data.

The assembler directives we use to do this will vary depending on our assembler, but 8 bit assemblers often use DB and DW for Define Byte and Define Word.
DS (Define Storage) will define a block of data (usually initialized to zero). For example to define a block of 512 bytes DS 512 could be used.

On 16 bit systems the syntax is often DC.B, DC.W and DC.L for Define Constant Byte, Word or Long.
DS.B, DS.W and DS.L can be used to define a block of 8, 16 or 32 bit data.
For example: "DS.W 100" will define 100*16 bit words, 200 bytes total (the same as "DC.B 200" would).

A similar concept on 68000 systems is the BSS Section. This stands for 'Block Started by Symbol', though that doesn't really make its purpose clear. The BSS section is an area of memory which is allocated to our program but starts with zero values, so we can use it to store our sprites or level data, but we could use it for a screen buffer.

Programming Techniques

Assembly programming has some common programming techniques and methods it's worth pointing out, as they may help you design your programs.

Lookup Table

A Lookup table is a table of pre-calculated values for some purpose. They are used to save time where the calculation job would be too slow.

Common uses for a lookup table would include Sine values, Multiplication calculations or transparency masking colors of sprite data, though there are any number of possible practical uses for lookup tables.

For Example: Suppose we want to divide numbers between 0 and 279 by 7, and get a whole number (quotient) and remainder result. On an 8 bit CPU this will be very slow.

If we can spare 280*2 bytes then we can pre-calculate the whole number and remainder for each value 0-279, and store them in pairs in the lookup table.
When we need the answer, we just look at the correct offset, and the two values are there.

```
Sine:
    db 128,176,217,245,255,245,217,175,128,77,36  ,8  ,0  ,8  ,36 ,78
Linear1:
    db 0  ,32 ,64 ,96 ,128,160,192,224,255,223,191,159,127,95 ,63 ,31
Randoms1:
    db $0A,$9F,$F0,$1B,$69,$3D,$E8,$52,$C6,$41,$B7,$74,$23,$AC,$8E,$D5
```

Figure 36: 16 Byte Look up tables for generated movements.

Jump Block

A Jump block is a set of Jump commands at a specific location. These jumps will have a defined 'purpose'.

For example: If we had a jump block at address $3000, we could have a jump to a ClearScreen function at $3000, and a Jump to DrawCharacter at $3003. Our code would call these addresses to perform the task required.

The advantage of this is a later revision of the program could have completely different internal structure, but, provided the address and defined purpose of the jumps was unchanged, anything that uses the jump block would still work the same.

For this reason Jump blocks are often used in system ROM, but we may want to use them in our own programs as part of our 'Game Engine', especially if we're making a 'Multi Load' game that loads each level separately, as it means we can make improvements later without changing (and recompiling) all our levels.

```
jp ShowSprite       ;8000
jp ExecuteBootStrap ;8003
jp LoadDiscSector   ;8006
jp StarArray_Redraw ;8009
jp SetLevelTime     ;800C
jp Player_Handler   ;800F
```

Figure 37: The ChibiAkumas Jump block with functions provided by the game core.

Vector Table

A vector table is a set of addresses. One such example is the one used by the 6502 for interrupt handling. We can create our own vector table as a way to define 'numbered commands'.

For Example: On an 8 bit machine we could write a function that takes a command number in the Accumulator, We would then look up a two byte 'Vector' from the table, based on that number, and call that address.

In a similar way to the 'Jump Block', this allows us to change the internals of our code, but keep the way the functions are called consistent. It's also useful for creating a 'Virtual machine', as we can look up and execute numbered commands from a stream of bytes.

Relocatable Code

When we compile our program, often it will have 'Absolute addresses'. This means it must be loaded and executed at the address the code was compiled for (defined by an ORG statement or similar). If the code is loaded to a different address, then the Jumps will go to memory addresses which do not contain the commands they should.

For Example: Let's suppose our program starts at $1000, and there is a jump to absolute address $1010 ("JMP $1010" in 6502).

If we load this program to address $2000 then the "JMP $1010" command will still go to address $1010, and the code will not be there. (it would be at address $2020).

Sometimes we'll want our code to be able to move, and be run at any address in memory. This is known as 'Relocatable code'. To achieve this we need to ensure we only use 'Relative addresses' in our code *.

If our example program starts at address $1000, with a relative jump to +$10 ("JR $10 in 6502), but we load the program to $2000 the JR command will now go to the correct $2010. The program is 'Relocatable' and will work from any address.

The assembler can't help us with this, we need to only use relative commands in our code. If we always use commands like JR instead of JMP on the Z80, and BRA instead of JMP on the 65c02, then we can create a relocatable program.

* Note: Technically speaking some 68000 based systems CAN relocate absolute addresses, they have a 'Relocation' table which contains pointers to the bytes of code which are

absolute addresses, and these are modified by the operating system before the program is executed.

Assembly Programming Terminology

Indirection / Pointers

Indirection / Pointers are a common way of using registers for 'lookups'. This is where a register contains an address to look at for the value.

It's kind of like going to the cupboard, and finding a note saying 'The pickles are in the fridge'!

Direct reading from address $0081

LDA $81 A loaded from address $81 in zero page ($**0081**)
Result A=**$12**

	$0080	$0081	$0082	$0083	$0084	$0085	$0086	$0087
Zero page	$11	$12	$13	$14	$15	$16	$17	$18

Read indirectly from $0081

LDA ($81) Address **$1312** is loaded from in zero page ($0081)
A is loaded from address $1312
Result A=**$31**

	$0080	$0081	$0082	$0083	$0084	$0085	$0086	$0087
Zero page	$11	$12	$13	$14	$15	$16	$17	$18

	$1310	$1311	$1312	$1313	$1314	$1315	$1316	$1317
Ram	$29	$30	$31	$32	$33	$34	$35	$36

Figure 38: Indirection on the 65c02. The data is read from the address at an address.

Jumps and Branches

Normally after each command the CPU will run the next command, but there will be times we need to jump elsewhere.

Unconditional jumps will ALWAYS jump to a different place, this is like a GOTO command in Basic.

Conditional jumps will SOMETIMES jump, depending usually on one of the flags registers. This is like an "IF THEN GOTO" command in Basic.

Branches are the same as Jumps (on the Z80 these are called Relative Jumps - "JR"). On many systems they result in smaller files, but can't jump very 'far' forward or backward in code (-128 bytes to +127 on most 8 bit systems).

```
 7      ld a,'A'
 8      call &BB5A      ;PrintChar A
 9
10      jp JPTest   ;Jump to label "JPTest"
11      call &BB5A      ;This command is skipped
12  JPTest:
13      ld a,'B'
14      call &BB5A      ;PrintChar B
15
16      jr JRTest   ;This compiles to JR $03
17      call &BB5A      ;This command is skipped (3 bytes)
18  JRTest:
19      ld a,'C'
20      call &BB5A      ;PrintChar C
21      ret
```

Figure 39: Absolute and Relative jumps in Z80 code. The Assembler converts the labels to numeric Addresses and relative offsets.

Subroutines

Subroutines are a bit like a JUMP – however these will run part of the code, and then come back when return occurs. This is like GOSUB / RETURN in Basic.

Subroutines are completely essential! When we write our program, we'll need to break up the problem.

Suppose we want to show the message 'Hello'. We'll probably run a 'PrintString' subroutine, that will probably make multiple calls to a 'PrintChar' subroutine which may call a 'DrawPixel' subroutine!

We write and test each subroutine separately. Once we get 'PrintChar' working right, it's no harder to use than the PRINT command in Basic.

```
11      call Newline    ;Call Newline Sub (GOSUB)
12
13      ret             ;Return to basic
14
15  NewLine:            ;Newline Subroutine
16          ld a,13         ;Carriage return
17          call &BB5A      ;PrintChar A
18          ld a,10         ;Line Feed
19          call &BB5A      ;PrintChar A
20      ret             ;RETURN to callee
```

Figure 40: This example calls a subroutine called Newline.

Self Modifying Code

Self Modifying Code is code which changes itself. This is usually done to improve speed, but also saves memory.
For Example: Suppose during our game we have to check either the keys or joystick, depending on the option selected.
We could read the 'controller' byte, and call the Key routine, or the Joystick routine, but it would be quicker to call the Key routine and rewrite the call if the player enables joystick. This will save a few bytes, and a little CPU power, but it makes the code harder to read.

```
3       ld a,%11110000  ;1 byte of yellow
4       call FillBytes  ;Fill to the right
5
6       ld a,&2B    ;&2B = "DEC HL" in bytecode
7       ld (ChangeCommand),a    ;Modify sub
8
9       ld a,%00001111  ;1 byte of cyan
10      call FillBytes  ;Fill to the Left
11      ret     ;return
12
13  FillBytes:
14      ld b,8          ;Bytecount
15      ld hl,&C010     ;Screen RAM
16  FillBytesAgain:
17
18  ChangeCommand: inc hl       ;This command will be changed
19
20      ld (hl),a       ;Write byte
21      djnz FillBytesAgain ;Repeat
22      ret         ;return
```

Figure 41: Here we've got a routine that draws 8 bytes of pixels to the right. The "INC HL" is changed to a "DEC HL" via self modifying code, and 8 bytes are drawn to the left

Aligned Code

There may be times when we want to specify our code to be aligned on a certain byte boundary.
For Example: We may want the bottom byte of our data to be a $00. So $1100 or $1200 is OK for our purposes, but $1180 is no good. We can do this with a command like 'ALIGN 8', which will align the data to an 8 bit boundary.

This is useful for times we're reading data from lookup tables, and when we want to only increase the low byte of a 16 bit address ("INC L" is faster than "INC HL").

The 68000 can only read words (including commands) on even boundaries (where the bottom bit is 0). This has a special EVEN command to do this (equivalent of ALIGN 2).

Note: Other assemblers may use other syntax. For Example: VASM uses ALIGN 8, but WinApe uses ALIGN 256.

```
11      ret                     ;C9 in hex
12
13      align 4                 ;Align to next 4 bit boundary
14
15  Message: db 'Hello World 123!',255  ;String to show
16
```

```
70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
80  21 10 10 cd 21 10 cd 2b 10 c9 00 00 00 00 00 00  !...!..+........
90  48 65 6c 6c 6f 20 57 6f 72 6c 64 20 31 32 33 21  Hello World 123!
a0  ff 7e fe ff c8 23 cd 5a bb 18 f6 3e 0d cd 5a bb  .~...#.Z...>..Z.
```

Figure 42: An example Align statement and resulting compiled binary. The "Align 4" has caused the 'Hello World' to start at $90.

Graphics Terminology

Sprite

A sprite is an image used in our game. There are two types:
Hardware sprites are shown by the hardware. They are very fast, and removing them is very easy, we just turn it 'off', but there's a limit to how many are on the screen.

Software sprites are done by 'us'. We have to basically plot each pixel to the screen to draw the image to the screen. When we want to remove them we have to redraw the background where the sprite was.

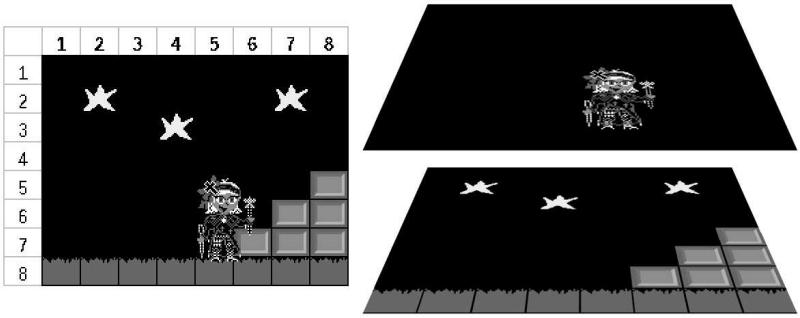


Figure 43: Left: An imaginary platform game screen with tiles and sprites.
Right: The final image is made up of 2 layers. The top layer is hardware sprites, The second is the tile map.

Tiles, Patterns and Tile maps

Hardware Tile maps are very common on console and arcade hardware.
The Tile map is a grid type 'layer' of square 'tiles', usually 8x8 pixel squares.
Each entry in the tile map is not a picture, it's a number which refers to a Pattern which is the tile bitmap itself. This saves a lot of memory, as a 32x32 tile map will take 1024 or 2048 bytes, but means there's often a limit to how many 'unique' tiles the screen can have.

For Example: The Sega Master System would require 768 tile patterns to have a completely unique 256x192 pixel screen, but the master system only supports 512 tiles. This means any 'full screen image' will have to have duplicated squares (probably blank).

8 bit systems usually have just one tile layer, but 16 bit systems often have 2 or more. This allows for 'Parallax' where there's a foreground and a background that move at different speeds. You may think you've seen this on 8 bit systems, as many games with just one layer do clever tricks to simulate this effect!

Just like with sprites, bitmap based computers can also use 'tile maps' but these will be software based, and therefore much slower than the hardware based ones.

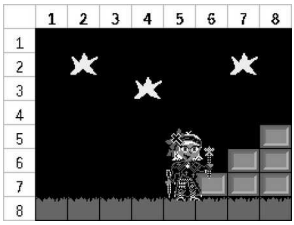


Figure 44: An imaginary platformer game screen.

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	3	0	0	0	0	3	0
3	0	0	0	3	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	1	1	1
8	2	2	2	2	2	2	2	2

	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	9	10	11	12	13	14	15	16
3	17	18	19	20	21	22	23	24
4	25	26	27	28	29	30	31	32
5	33	34	35	36	37	38	39	40
6	41	42	43	44	45	46	47	48
7	49	50	51	52	53	54	55	56
8	57	58	59	60	61	62	63	64

Figure 45: The tile map uses repeating objects, a star, a grass pattern, and a block - these are repeats of the same pattern. A 1 byte tile number can define this large area.

Bitmap screens

A bitmap screen is an image where the visible contents of the screen are defined by a block of byte data in memory, where (unlike the Tile map systems) a cluster of pixels will be defined by a byte.

Depending on the pixel depth, the number of pixels per byte will differ.
A 256 color system will have 1 pixel per byte.
A 16 color system will have 2 pixels per byte.
A 4 color system will have 4 pixels per byte.
A 2 color system will have 8 pixels per byte.

This is only a general rule, some systems like the spectrum have 'color attributes' and there's a few bytes of 'wasted space' in the CPC screen memory area.

Let's compare different system memory usage:
The 320x200 4 color screen of the CPC takes about 16k.
The 256x192 16 color screen of the MSX2 and SAM Coupe takes 24k.

The Spectrum's 256x192 screen takes 6k.
This is part of the reason MSX2 games tend to be slow, and Spectrum games are much faster than the CPC!

Color Attributes

Color attributes are a way of 'saving' memory but giving a more colorful screen.
The Spectrum's black and white screen uses 6k of bitmap data and 768 bytes of color attributes. The extra 768 bytes turn a black and white screen into a color screen with 2 colors per 8x8 square.

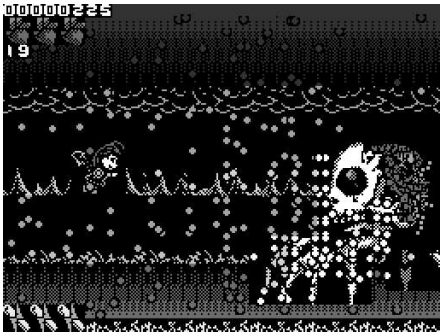


Figure 46: Color attributes on ChibiAkumas on the ZX Spectrum, each 8x8 square only has 2 colors.

Palettes

A Palette is a set of colors.
Typically our screen will be limited to 4 or 16 colors, but we can choose those colors from a wider range (27 on the CPC).

Palettes are also relevant to consoles. A console like the SNES uses 16 color sprites, but each sprite can choose its 16 color palette from a wider range, giving 256 on screen colors! Doing so saves memory, as a 16 color per pixel image takes half the memory of a 256 color one.

Raster Beam

Cathode ray tubes (old non flat-screen TVs) use a 'Raster beam' that scans the screen in a zigzag from top left, to top right and down the screen.

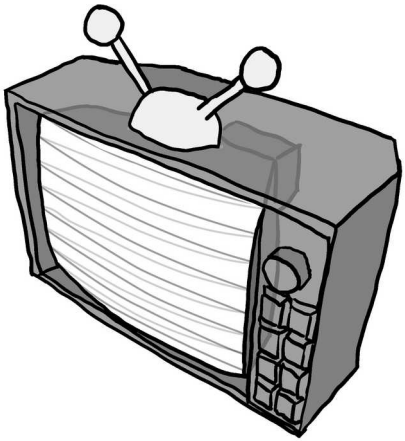


Figure 47: The raster beam scans the screen from left to right, top to bottom - it happens so fast the eye cannot see the redraw.

Many old games took advantage of this to perform 'clever tricks', changing things as the screen was redrawing.

For Example: ChibiAkumas used this to get 16 colors on a 4 color screen. Other games use it to make the screen 'wavy' and systems like the C64, which has an 8 sprite limit, move those sprites the line after they've drawn to a new position, overcoming the sprite limit (ChibiAkumas also did this on the CPC+).

Figure 48: ChibiAkumas Ep2 Title screen. The screen uses 4 color mode, but 2 colors are changed 4 times during the raster redraw to make 12 colors appear.

VBlank

Vertical blank is the time after the screen has finished drawing, but the next screen hasn't started yet.
On many systems this is the only time we can alter Video RAM, but 'waiting for VBlank' is also an easy way to limit the maximum speed of our game and stop it running too fast.

There is also a 'HBlank' (Horizontal blank) between lines, however it is very short so not as useful.

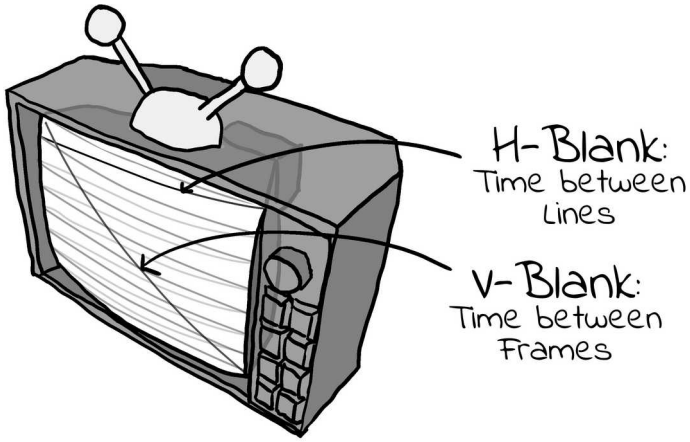


Figure 49: VBlank and HBlank.

CRTC

Cathode Ray Tube Controller. This is the chip used on some systems to turn bytes of memory into a picture. Changing the settings of this will change the shape of the screen, the address used for the screen, and other effects.

VDP

Video Display Processor. This term is used by some systems to refer to the 'graphics card' of the machine.
The VDP will often have separate memory from the main CPU, which cannot be accessed in the same way.

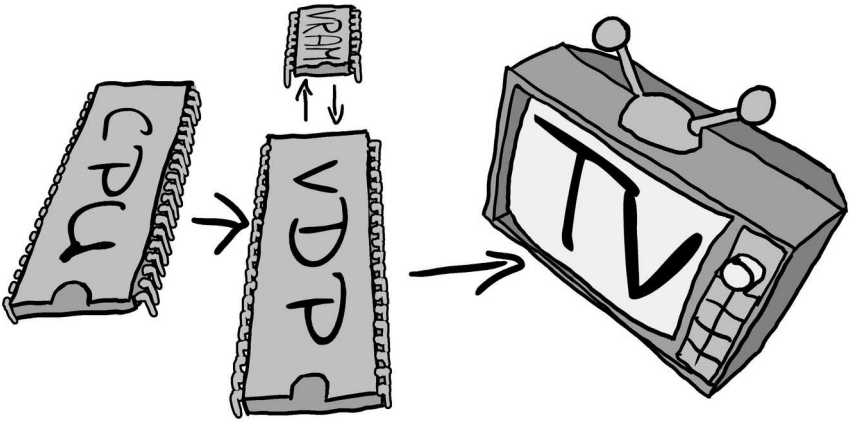


Figure 50: The CPU talks to the VDP which has its own memory, and generates the image for display.

Bitplanes and Linear data.

These are two different ways of storing image data.
Let's suppose we have a system with 4 color sprites, 4 colors requires 2 bits per pixel (2bpp).
If we want to store 8 pixels (two bytes), we have two choices:

1. Store the two pixels for each byte together. This could be called 'Linear' data.

2. Store all the bit 0s for each of the 8 pixels in one byte, store all the bit 1s in a second byte. This is known as storing in 'Bitplanes'.

Example:

2 bit per pixel	7	6	5	4	3	2	1	0
Pixels	0	1	2	3	1	1	2	2

As bitplanes	7	6	5	4	3	2	1	0
Bitplane 0	0	1	0	1	1	1	0	0
Bitplane 1	0	0	1	1	0	0	1	1

As Linear data	7	6	5	4	3	2	1	0
2 bytes	0	0	0	1	1	0	1	1

7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	0

Figure 51: Example of 2bpp as bitplanes or 'linear' data.
Note: Some non-bitplane bitmap screens use other orders for pixel bits (Like Mode 0 of the CPC).

Screen Buffer

A Screen buffer is the memory that makes up the bitmap screen. If we write data into this, the visible screen will change.

Double Buffering is where we use a pair of bitmap screens. One is the 'Visible screen' we show to the player, the other is the 'Drawing screen' which we update. Once the Drawing screen is completed, we flip the two buffers, showing the newly drawn screen, and using the previously visible screen as the new drawing screen.

Other Hardware Terminology

ULA / Gate array / ASIC

ULA stands for Uncommitted Logic Array – this is a chip which performs multiple tasks relating to memory and graphics on the Spectrum.

The Gate Array on the Amstrad CPC is similar, performing Graphics and Memory mapping.

The ASIC (Application Specific Integrated Circuit) on the CPC+ takes on the extra capabilities of that system.

Strobe

A sequence sent to a data port (or single bit of a port) to signal to a device. This is often done to initialize a device and prepare it to start sending data.

For Example: We may strobe a joypad port to tell it to start sending the data for the first of four joypads, subsequent reads will get joypads 2 3 and so on.

Analog Joystick

An Analog Joystick is one that is not just 'On or Off', it will give a range of values depending on the precise position of the joystick.

If the range was 8 bit, the centre would be a value like 128, full left would be 0 and 'a bit left' would be 100.

They give smooth movement, but unfortunately they're often harder to read than a digital joystick, which is only On or Off. This is partially because reading the numeric position is often more complex, and because our imaginary example is unrealistic.

An Analog joystick will be slightly 'unreliable', so on a joystick with a centre of 128, you will find it won't actually return to this value every time. The 'centre' would probably be a range of values, and we may consider any value in the range 112-144 to be treated as the 'center', this is known as a Joystick "Dead Zone".

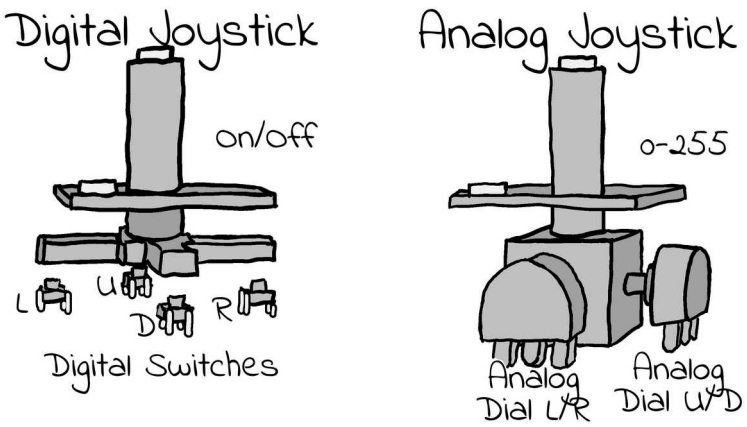


Figure 52: Analog and Digital Joysticks.

DAC / ADC

A Digital to Analog Converter converts a digital signal to an Analog one, for example taking bytes and converting them to sound. ADC is the reverse, taking Analog and converting it to Digital.
Converting between Analog and Digital is needed for sound, non digital joysticks and other such tasks.

Multiplexer

A device which allows for multiple devices to share a single line.
For Example: On the Tandy COCO, the multiplexer shares the sound DAC between the joystick, speaker and cassette.

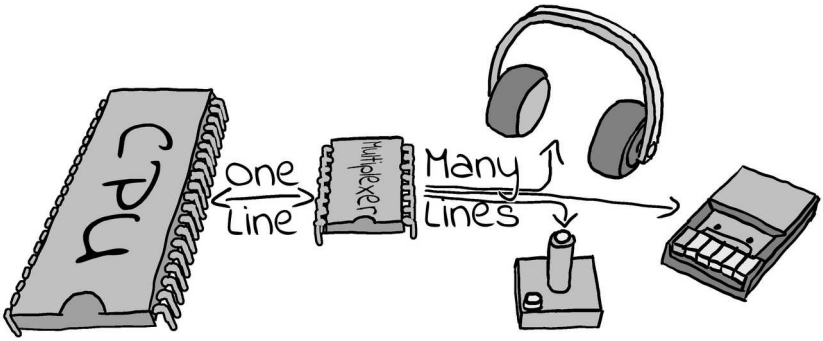


Figure 53: A multiplexer allows multiple devices to connect to one line.

Hints and Tips for starting Assembly

(Also known as: Things I wished I'd been told before I had started!)

1. Labels are always at the far left of the source code. They usually need to end with a colon. Commands are indented with a tab.

2. Numbers are usually decimal by default. Depending on your assembler, numbers starting "&" "$" or "0x" will be hexadecimal. Some assemblers have numbers ending "h" as a hexadecimal number. This causes problems where the first digit is a letter, so you may need to put an extra zero at the start.
Examples of hexadecimal numbers are: &F000 $F000 0xF000 0F000h

3. When you're starting programming assembly, aim small, very small!
You should start by just getting a few characters on the screen, then a few pixels, then read some input, move the pixels etc. Soon you'll have something resembling a game, but it's all about being satisfied with the small wins and moving slowly towards the big program.

4. Read all the technical documentation of the platform you're aiming at, there's tons of documentation out there for free on these old systems on Archive.org and other free websites. Download them, put them on your phone/tablet and read them whenever you get a chance. At first it won't mean much to you, but over time it will start to sink in, and you'll need to know it all some day.

5. This book does not teach everything you need, it's just aimed as an introduction in 'down to earth' language. There are detailed manuals on the Z80, 6502, 68000 and other CPUs available free straight from the manufacturers, for the detailed function of the instructions. There are dozens of old books on every platform out there that give all the details.
Also bear in mind everyone is different, if you read one book and can't understand it, try another! Different people learn in different ways, and no one style is perfect for everyone. This book is written in the style I find the easiest, it's possible you do not, so don't get disheartened if you're struggling to understand one particular reference book.

6. On the 6502: If you are confused about the Zero Page, it's best to think of it as a bank of registers. The bytes of Zero Page RAM on the 6502 have the same purpose as the HL, DE and BC registers of the Z80. This was very true of the TMS-9900, which used a 'workspace' of 16 registers, which was literally held in memory like a Zero page.

7. It's perfectly normal to struggle with Hexadecimal and Binary when you're getting started. It's worth taking a look at Windows Calculator's 'Programmer mode', which will allow you to convert numbers to and from Hexadecimal, Binary, Decimal and Octal. Microsoft Excel and LibreOffice also have conversion functions like Dec2Hex and Bin2Dec (and others) which you can use for converting bases.

8. On the 68000: VASM does NOT like spaces in between parameters with 68000 assembly – some other assemblers allow it, but VASM does NOT! If there's a space after the comma "MOVE.L D1, D2" will not compile on VASM because there is a space after the comma..

9. On the 68000: Commands can work at the Byte, Word or Long level, and this is specified with .B .W or .L at the end of a commands opcode. The command will default to .W, but it's

best to get in the habit of specifying the length even for words, otherwise will surely forget one time when it does matter.

Chapter 2: The Z80

Introducing the Z80.

The Z80 is an 8 bit processor, usually around 4 MHz, but 6mhz versions exist.
The MSX Turbo-R R800 is also 100% Z80 compatible, with 4x the effective speed.

Each Register can only store one byte (0-255), but some registers can be used in certain cases together as 16 bit pairs. For example HL together are 16 bits, and can store 0-65535.

Each of the registers has a 'purpose' it is intended for, but you can use any register for anything you want!
The different registers all have 'strengths' because many commands will only work with certain ones, and some commands may be slower or need more code if you use the wrong one.

The Z80's large number of registers makes Z80 programming very different to a system like the 6502.

The Z80 uses a 16 bit address bus. This means it can address $0000-$FFFF (0-65535), giving a total of 64K. Systems like the 128K Amstrad 6128 get around this limitation via bank switching.
In addition to the addressable memory, the Z80 also addresses hardware ports via OUT and IN.

On some systems, like the MSX, ports are 8 bit. These use register C as the port number in commands like "OUT (C),A" but on other systems, like the CPC or Spectrum, ports are 16 bit using BC as the port number. Annoyingly the command is still the rather misleading "OUT (C),A".
You may wonder why some use 8 bit ports, and some use 16 bit ones. The difference is not the Z80 itself, but how the Z80 is wired up in the computer.

On the Z80, when commands have two parameters, the parameter on the right is the source, the parameter on the left is the destination.
For Example: "ADD HL,DE" will add the source DE to the destination HL.
Finally it should be noted the eZ80 is a more enhanced Z80 with a 24 bit address bus, which is also backwards compatible with the Z80.

The Z80 Registers

A	The Accumulator is used during all our main calculations. You will use it all the time when adding up (Accumulation). A and F are combined in the stack operations - PUSH AF / POP AF will transfer the Accumulator and Flags as a 16 bit pair.
F	The Flags Register. Each bit defines a 'condition' we can test for. We don't tend to use this directly, it's set by mathematical operations, and we respond to its contents via conditional jumps and calls.
HL	The High Low pair. This often stores memory locations, as there are a lot of special commands that use it to quickly read or write whatever memory locations. It's also good at 16 bit maths, it's almost the Z80's 16 bit Accumulator, so if you want to add two numbers above 255, you'll probably need it.
BC	These are often used as a Byte Count or loop counter. Sometimes you'll use B and C together, for a count of up to 65535, or just B on its own for up to 255.
DE	DE is often used as a Destination. If you're reading from one place and writing to another, you'll probably use HL as the source, and DE as the destination.
IX	Sometimes we want to get to memory by specifying a relative position. Index Registers allow us to do this. For Wxample: If we have sprites, and each 4 bytes for X, Y, Width, Height. Just point IX to the start of the data for the sprite we want and read the rest out as IX+1, IX+2 etc. Don't worry about this, we'll explain it later. IX is actually a pair of two registers called IXH and IXL. We can use them alone for whatever we want, but they are slower than other registers.
IY	IY works the same as IX. Note: IY is used by the operating system on the Spectrum and TI-83 calculator, so won't be available for your use on those systems.
PC	This is the place in memory that the Z80 is running. We don't change this directly, but the CALL, JP and RET commands all affect it.
SP	This is the stack pointer. It points to a big temporary store that we'll use for 'backing up' values we need to remember for a short while.
R	This is the Refresh register. The system uses it to know when to refresh the memory, so don't change it, You could mess something up! But it can be used for getting simple 'random' numbers! The R register goes from 0-127, the top bit is always 0.
I	This is the Interrupt vector register, it's only used by Interrupt Mode 2 (IM2). IM1 is easier to work with than IM2, so, on systems like the CPC and MSX, I is unused. This means you can use the I register as a 'temporary' store if you wish. On the Spectrum you may want to use IM2, as IM1 is not reprogrammable since the &0038 address IM1 uses is read only ROM.

The Z80 Shadow Registers

The Shadow registers are 'extra' registers that are swapped with the main ones.
These are AF' (Accumulator and Flags) BC' DE' and HL'. There is no shadow IX/IY or SP.

These shadow registers are usually used during the interrupt handler, so the main registers are unchanged for when the interrupt handler ends. You probably won't be able to use them unless interrupts are permanently disabled, or you write your own interrupt handler.

There are two exchange commands: "EX AF,AF'" swaps the Accumulator+Flags, "EXX" swaps BC,DE and HL.

The Z80 Flags

The flag register contains 8 one bit 'states'. Each keeps track of an attribute of the last command we performed.
For Example: If we do a compare like "CMP #0", the Z flag will be set if the accumulator is zero. The S flag will be set if the accumulator is negative. We can use these later with conditions to branch to different parts of our code.

The Z80 'F' Register contains 8 bits: %SZ-H-PNC .

The main ones you will need to know about are Carry (C) and Zero (Z). Sign (S) is also useful in some cases, but you'll probably never actually need to know about P/V, H and N.

On the Z80 many commands do not change the flags. You need to check if the command you're using actually updates them.
For Example: "DEC B" will correctly update the zero flag, "DEC BC" will not, so with "DEC BC" you'll need to manually check if B and C are zero. This can be done with "LD A,B OR C".

Bit	Flag	Name	Description
7	**S**	Sign	Positive / Negative
6	**Z**	Zero	Zero Flag (0=zero)
5	**-**		
4	**H**	Half Carry	Used by DAA
3	**-**		
2	**P / V**	Parity / Overflow	Used for Overflow - if a sign changes because a register is too small Also used as Parity for error checking for bit operations
1	**N**	Add / Subtract	Used by DAA
0	**C**	Carry	Carry / Borrow

In the instruction references of this book, the 'Flags Affected' section will show a minus '-' when an instruction leaves a flag unchanged, an uppercase letter when a flag is correctly updated (for example 'C'), and a lowercase letter when a flag is changed to an undetermined state (for example 'c').

Z80 Conditions

The Z80 has a variety of condition codes, which can be applied to Jumps, Calls and Returns:

Condition	Meaning	Flag
Z	Zero	Z Set
NZ	Non Zero	Z Clear
C	Carry	C Set
NC	No Carry	C Clear
PO	Parity Odd	P/V Clear
PE	Parity Even	P/V Set
P	Positive Sign	S Clear
M	Minus Sign	S Set

Representing data types in source code

Here are the typical ways Z80 assemblers represent the different data types:

Prefix	Alternatives	Example	Meaning
		12	Immediate Decimal Value.
%		%10101010	Immediate Binary Value
&	# $	&FF	Immediate Hexadecimal Value
'		'A'	Immediate ASCII Value
()		-1000	Memory Address in Decimal
(&)	(#) ($)	(&1000)	Memory Address in Hexadecimal

Defining bytes of data on the Z80 in WinApe

There will be many times when we need to define bytes of data in our code. Examples of this would be bitmap data, the score of our player, a string of text, or the co-ordinates of an object.
The commands will vary depending on the CPU and your assembler, but the ones shown below are the ones used by the assembler covered in these tutorials.

For Example: If we want to define a 16bit sequence &1234 we would use "DW &1234".

Bytes	Z80	6502	68000	8086	ARM
1	DB	DB	DC.B	DB	.BYTE
2	DW	DW	DC.W	DW	.WORD
4			DC.L	DD	.LONG
n	DS n,x	DS n,x	DS n,x	n DUP (x)	.SPACE n,x

Z80 Addressing Modes

Immediate Addressing

A single byte fixed number parameter is specified as a parameter after the Opcode.

Usage Example: ADC 128 XOR 128

Immediate Extended Addressing

A pair of bytes is specified as a parameter.

Usage Example: LD BC,$1000 LD HL,$C000

Modified Page Zero Addressing

RST calls are effectively a jump to an address starting $00##. We specify the single ## byte for the call.

Usage Example: RST 0 RST $38

Relative Addressing

Relative addressing is where the parameter is an 8 bit signed byte offset (-128 to +127) relative to the current Program Counter position (the position AFTER the JR command).

Usage Example: JR 16 JR -100

Extended Addressing

A 16 bit address used as a destination, either for a jump, or for the address to load or store a parameter.
The parameter will be specified in brackets (##) to show the value is not an immediate.

Usage Example: JP $1000 LD A,($C000)

Indexed Addressing

One of the two index registers (IX or IY) plus a displacement offset are specified.
The displacement is an 8 bit byte, so has the possible range -128 to +127.

Usage Example: LD A,(IX+1) XOR (IY-4)

Register Addressing

This is simply where the source or destination parameter is a register.

Usage Example: LD C,4 LD H,B

Implied Addressing

This is where one of the parameters is implied (usually the Accumulator).

Usage Example: SUB 4 CPL

Register Indirect Addressing

This is where a 16 bit register is specified, but the address in that register is the source of the parameter, not the value in the register itself. The register will be specified in brackets.

Usage Example: JP (HL) XOR (HL)

Bit Addressing

This is where a single bit of a register is being read or altered in a register.

Usage Example: BIT 6,L RES 1,C

Addressing Mode Combinations

Multiple combinations of addressing modes can be used as the source and destination in some cases.

Usage Example: BIT 6,(HL) LD E,(IX+1)

Compiling and running Z80 code in WinApe

If you're looking to get started with Z80, Amstrad CPC emulator WinApe is the best for you to take a look at.

WinApe is a fine Amstrad CPC Emulator, however what makes it special is it has a built in Assembler and Debugger. This allows us to compile Assembly, run it and troubleshoot it all from a single program.

You can get WinApe from: http://www.winape.net/

Using Winape

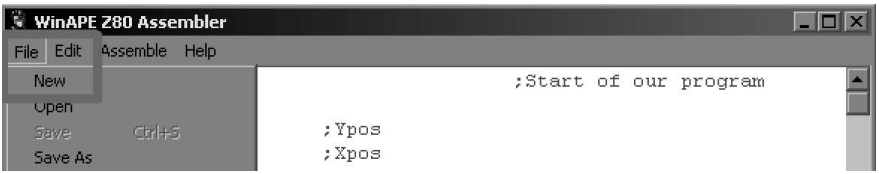
When you start WinApe you should see the emulator window open and show the Amstrad CPC blue screen. The CPC starts straight away into Basic, and we can use Basic to start our compiled program.

Don't worry if you've never used a CPC before, this tutorial assumes you know nothing about Basic or the CPC.
Let's learn how to compile a program with WinApe.

Click on the **Assembler menu** and select **"Show Assembler",** or press F3 on your keyboard.

Writing an Assembly program

Select the **File Menu**, then **New** to create an empty ASM program.



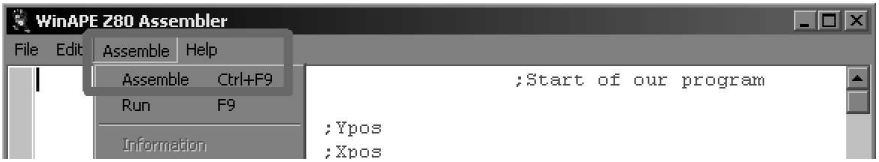
We type in our ASM code into the assembler. The program below will show a few characters to the screen.

```
1     org &8000           ;Start program at address &8000
2
3     ld a,'A'            ;Load A into the accumulator
4     call &BB5A          ;PrintChar A
5     ld a,66             ;Load 66 (B) into the accumulator
6     call &BB5A          ;PrintChar B
7     ret                 ;Return to Basic
```

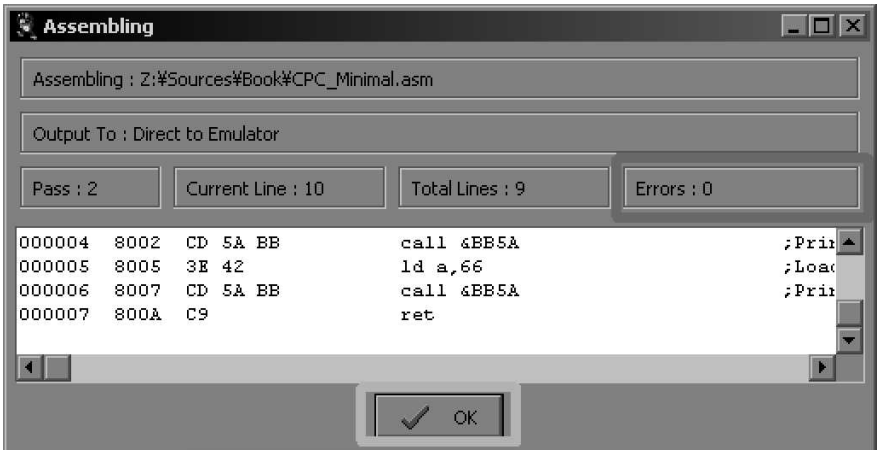
Note: WinApe uses "&" to mark Hexadecimal symbols not "$".

Don't worry about the case of your commands and labels, as WinApe isn't case sensitive, so "ORG &8000" is the same as "org &8000".

Select **Assemble** from the **Assemble menu** to compile the program, or press F9.



You will see the Assembler output. This shows the actual bytes of compiled code that were made from the code you typed.

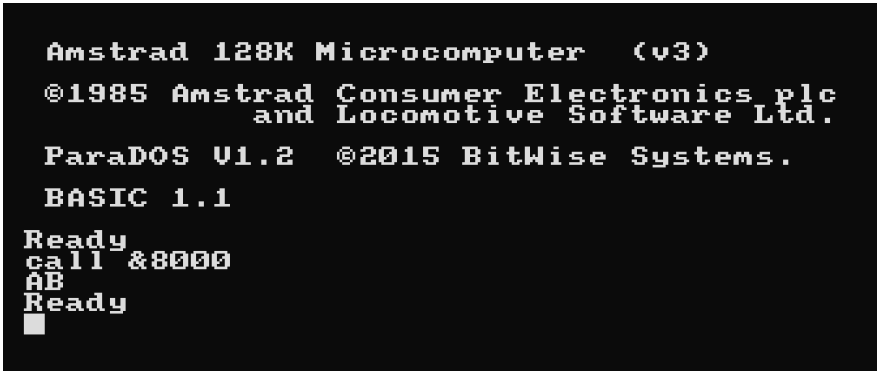


Check there are **Zero Errors.** If there are any errors you need to check your code and fix them.
Click on **OK.**

Now we learn about the magic of WinApe. Your code is now immediately in the emulators memory, and we can run it straight away!

Go back to the blue Basic screen, and type "Call &8000" into Basic and hit the Enter key.

Basic should show two characters 'AB' and return to the ready message.
We just wrote, compiled and ran our first Assembly program. Well done!



Z80 Examples

Example 1: Hello World

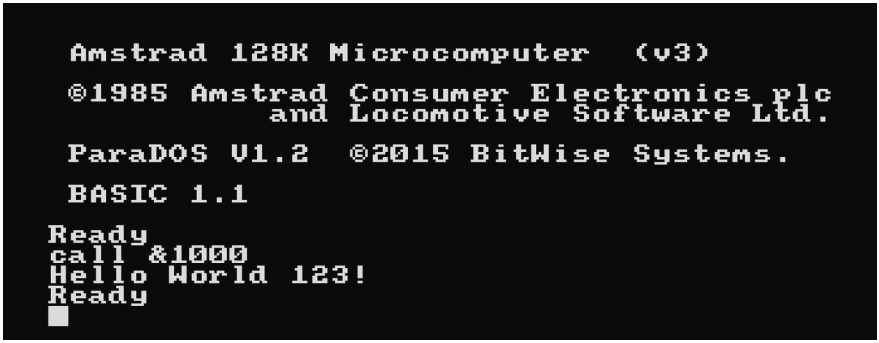
This Example will show a 'Hello World' Message on the CPC.

Type this into WinApe's Assembler and compile by pressing F9.

```
;Text after semicolon is comments, you don't need to type it in

;Run this program with CALL &1000 from basic

    org &1000           ;Start program at address &1000

    ld hl,Message       ;Address of string
    Call PrintString    ;Show String HL to screen
    call Newline        ;Move down a line
    ret                 ;Return to basic

PrintString:

    ld a,(hl)           ;Get A char
    cp 255              ;Is it a 255?
    ret z               ;Yes? then return
    inc hl              ;Move to next char
    call &BB5A          ;PrintChar A
    jr PrintString

Message: db 'Hello World 123!',255  ;String to show

NewLine:
    ld a,13             ;Carriage return
    call &BB5A          ;PrintChar A
    ld a,10             ;Line Feed
    call &BB5A          ;PrintChar A
    ret

```

To start this program, type "CALL &1000".

You should see the 'Hello World' message as shown.



How does it work?

Our program starts at memory address &1000. (Line 5)

We load the address of the message we want to show into HL. (Line 7)

We read in our string character by character, using HL as a pointer to the next character to show. We check if the character is 255, and return if it is. (Line 14-16)

We use firmware function &BB5A to print characters to the screen. This shows the Accumulator as an ASCII character. (Line 18)

To move the cursor down a line, we use this function to show a character 13, followed by character 10. (Line 24-28)

Example 2: Test Bitmap

This Example will show a Smiley bitmap on the CPC.

Type this into WinApe's Assembler and compile by pressing F9

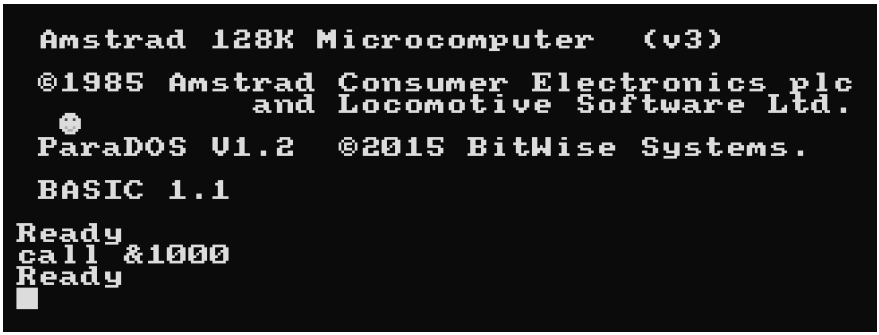
```
;Text after semicolon is comments, you don't need to type it in

;Run this program with CALL &1000 from basic

    org &1000           ;Start program at address &1000

    ld de,&0010         ;Xpos (in pixels)
    ld hl,&00A0         ;Ypos (in pixels)

    call &BC1D          ;Scr Dot Position - Returns address in HL

    ld de,TestSprite    ;Sprite Source
    ld b,8              ;Lines

SpriteNextLine:
    push hl
        ld a,(de)       ;Source Byte
        ld (hl),a       ;Screen Destination

        inc de          ;INC Source (Sprite) Address
        inc hl          ;INC Dest (Screen) Address

        ld a,(de)       ;Source Byte
        ld (hl),a       ;Screen Destination

        inc de          ;INC Source (Sprite) Address
        inc hl          ;INC Dest (Screen) Address
    pop hl
    call &BC26          ;Scr Next Line (Alter HL to move down a line)

    djnz SpriteNextLine ;Repeat for next line

    ret                 ;Return to basic

;Each byte holds 4 pixels,
;The top    nibble is bitplane 0
;The bottom nibble is bitplane 1

```

(Continued on the next page)

```
39  TestSprite:                     ;Test Smiley
40      db %00110000,%11000000      ;8 line sprite
41      db %01110000,%11100000      ;8 pixels per line
42      db %11110010,%11110100
43      db %11110000,%11110000
44      db %11110000,%11110000
45      db %11010010,%10110100
46      db %01100001,%01101000
47      db %00110000,%11000000
48
```

To start this program, type "CALL &1000".

You should see the Smiley below the copyright date as shown.



How does it work?

We need to transfer the bytes of our Smiley into VRAM. On the CPC VRAM is at memory address &C000+.

The CPC firmware function &BC1D calculates the VRAM address for us. This function takes an (X,Y) co-ordinate in (DE,HL). It returns a VRAM address for that co-ordinate in HL. (Line 10)

Now that HL points to screen RAM, we set DE to the sprite data, and transfer 2 bytes per line (8 pixels). (Lines 16-28)

Once we complete a line we use firmware function &BC26, which alters HL, moving it down one screen line. (Line 29)

We then repeat the procedure for all 8 lines of the Smiley. (Line 31)

The sprite is 2 bits per pixel, so 4 pixels are contained in a byte. The top nibble is bit 0 for the 4 pixels, the bottom nibble is the bit 1 for the same 4 pixels. (Line 39-47)

Example 3: Moving a sprite

This Example will show a Smiley bitmap on the CPC. You can move it round the screen with the joystick, but it cannot go off-screen, as we're checking the position before moving.

Type this into WinApe's Assembler and compile by pressing F9.

```
    org &1000           ;Start of our program

    ld hl,32    ;Ypos
    ld de,32    ;Xpos

    call DrawSprite     ;Show Srarting Pos

DrawLoop:
    push hl
    push de
        call &bb24  ;KM Get Joystick... Returns %---FRLDU
    pop de
    pop hl
    or a
    jr z,DrawLoop       ;See if no keys are pressed

StartDraw:
    push af
        call DrawSprite ;Remove old player sprite
    pop af
    ld b,a

    bit 0,b
    jr z,JoyNotUp       ;Jump if UP not presesd
    ld a,1
    cp 199              ;Check if already at top of screen
    jr z,JoyDone
    inc hl              ;Move Y Up the screen
JoyNotUp:

    bit 1,b
    jr z,JoyNotDown     ;Jump if DOWN not presesd
    ld a,1
    cp 8                ;Check if at bottom of screen
    jr z,JoyDone
    dec hl              ;Move Y Down the screen
JoyNotDown:
```

(Continued on the next page)

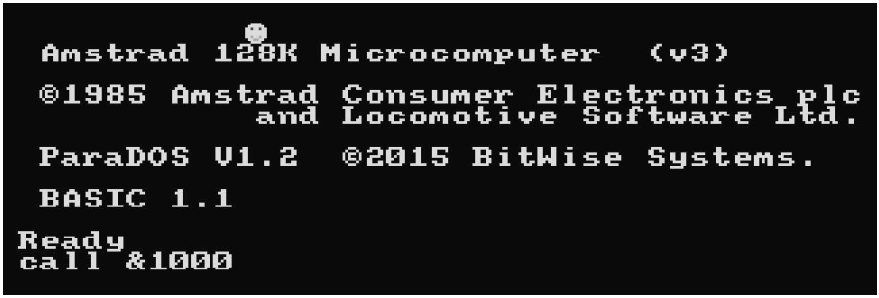
```
38
39      bit 2,b
40      jr z,JoyNotLeft      ;Jump if LEFT not presesd
41      ld a,d
42      cp 0
43      jr nz,JoyLOK         ;Check if at Left of screen
44      ld a,e
45      cp 0
46      jr z,JoyDone
47  JoyLOK:
48      dec de               ;Move X Left
49  JoyNotLeft:
50
51      bit 3,b
52      jr z,JoyNotRight     ;Jump if RIGHT not presesd
53      ld a,d
54      cp &1
55      jr nz,JoyROK         ;Check if at Right of Screen
56      ld a,e
57      cp &38
58      jr z,JoyDone
59  JoyROK:
60      inc de               ;Move X Right
61  JoyNotRight:
62
63  JoyDone:
64      call DrawSprite      ;Draw Player Sprite
65
66      ld bc,500            ;Delay
67  PauseBC:
68      dec c
69      jr nz,PauseBC
70      dec b
71      jr nz,PauseBC
72
73      jp DrawLoop          ;Repeat
74
75  ;Draw sprite at (X,Y) pos (DE,HL)
76  ;Sprite is XOR, so drawing twice will remove it
77
78  DrawSprite:
79      push hl
80      push de
81          ld bc,TestSprite     ;Player Sprite Source
82          push bc
83              call &BC1D   ;Scr Dot Position - Returns address in HL
84          pop de
85          ld b,8               ;Lines
86  SpriteNextLine:
87          push hl
88              ld a,(de)        ;Source Byte
89              xor (hl)         ;XOR current screen byte
90              ld (hl),a        ;Screen Destination
```

(Continued on the next page)

```
 91              inc de          ;INC Source (Sprite) Address
 92              inc hl          ;INC Dest (Screen) Address
 93              ld a,(de)       ;Source Byte
 94              xor (hl)
 95              ld (hl),a       ;Screen Destination
 96              inc de          ;INC Source (Sprite) Address
 97              inc hl          ;INC Dest (Screen) Address
 98          pop hl
 99          call &BC26          ;Scr Next Line (Alter HL to move down a line)
100          djnz SpriteNextLine ;Repeat for next line
101      pop de
102      pop hl
103      ret                     ;Finished
104
105  TestSprite:
106      db %00110000,%11000000
107      db %01110000,%11100000
108      db %11110010,%11110100
109      db %11110000,%11110000
110      db %11110000,%11110000
111      db %11010010,%10110100
112      db %01100001,%01101000
113      db %00110000,%11000000
```

To start this program, type "CALL &1000".

You should see the Smiley as before, but this time the joystick will move it around the screen.



How does it work?

The Smiley drawing routine has been changed, it now XORs the current screen byte with the Smiley bitmap data. (Line 87-98)

This means drawing the sprite in the same position twice will remove it, leaving the background unchanged.

We're using firmware function &BB24 to read the joystick. This returns a byte, where bits 0-4 are Up, Down, Left, Right and Fire. (Line 11)

We test these bits, and if a direction is pressed we check the current direction. We need to ensure our sprite does not go off the screen. (Line 23-61)

If moving in that direction will not push us off the screen, we INC or DEC the position in HL/DE to move the Smiley.

Once any movement has occurred we redraw the Smiley. (Line 64)

We then have a delay and repeat. (Line 66-73)

Learning Z80: Where to go from here

The documentation examples we've covered here should be enough to get you started with your first Z80 program.

Over the following pages of this chapter, you'll be presented with all the Z80 instructions, and a description of their purpose, with examples.

If you're looking for more, there's lots more content on the ChibiAkumas website!

If you're looking for more information on the Z80 instruction set, there are step by step tutorials, with videos and source code on the Z80 page:
http://www.chibiakumas.com/z80/

We've only covered the Amstrad CPC here. For details of the other systems covered, and detailed tutorials on the hardware of all the systems, take a look at the full list here:
https://www.assemblytutorial.com/

Don't forget to download the source code for this book from:
www.chibiakumas.com/book

And please subscribe to the official YouTube channel for weekly new videos:
https://www.youtube.com/chibiakumas

Z80 Instructions

ADC r

Add register **r** and the carry flag to the Accumulator A. A can be specified as destination or omitted.

Usage Example: ADC B ADC A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

ADC A,#

Add 8 bit number **#** and the carry to A. A can be specified as destination or omitted.

Usage Example: ADC 128 ADC A,128

Flags Affected: S Z H V N C

Valid values for '#': 0-255 ($00-$FF)

ADC HL,rr

Add 16 bit register **rr** and the carry to HL.

Usage Example: ADC HL,BC

Flags Affected: S Z H V N C

Valid registers for 'rr': BC DE HL SP

ADD rr2,rr1

Add 16 bit register **rr1** to 16 bit register **rr2**.
If **rr2** is HL, IX or IY, then **rr1** must also be the same register.
For Example: "ADD IX,IX" is valid, but "ADD HL,IX" or "ADD IY,IX" are not.

Any case where both registers are the same will double the value.
For Example: "ADD IX,IX" will effectively double the value in IX.

Usage Example: ADD HL,BC

Flags Affected: - - H - N C

Valid registers for 'rr1': HL IX IY
Valid registers for 'rr2': BC DE SP *HL IX IY*

ADD r

Adds 8 bit register **r** to A. The destination A can also be specified.

Usage Example: ADD B ADD A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

ADD

Adds 8 bit value **#** to A. The destination A can also be specified.

Usage Example: ADD B ADD A,B

Flags Affected: S Z H V N C

Valid values for '#': 0-255 ($00-$FF)

AND r

Logical AND of bits in register **r** with Accumulator A.
Where a bit in A and r are 1 the result will be 1, when they are not it will be 0.
For Example: If A=%10101010 and B=%11110000, if we use "AND B" this will result in A= %10100000.

Usage Example: AND B AND A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

AND

Logical AND of bits in 8 bit value **#** with Accumulator A.
Where a bit in A and # are 1 the result will be 1, when they are not it will be 0.
For Example: If A=%10101010, if we use "AND %11110000", then this will result in A= %10100000.

Usage Example: AND $64 AND A,%11110000

Flags Affected: S Z H V N C

Valid values for '#': 0-255 ($00-$FF)

BIT b,r

Test bit **b** from 8 bit register **r** and set the Z flag to that bit.
Bit 0 is the rightmost bit of R, Bit 7 is the leftmost. The byte's bits are numbered in the order %76543210 .

Usage Example: BIT 7,B

Flags Affected: s Z H v N -

Valid values for 'b': 0-7 (%76543210)
Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

CALL addr

Call Subroutine at address **addr** - this pushes the return address onto the stack, and changes the Program Counter to **addr**.

This is effectively the same as the GOSUB command in Basic.

Usage Example: CALL $1000 CALL TestLabel

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)

CALL c,addr

Call Subroutine at address **addr** only IF condition **c** is true in the flags register.
This is like an IF THEN GOTO statement in Basic.

Usage Example: CALL Z,$1000 CALL C,TestLabel

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)
Valid conditions for 'c': c m nc nz p po pe z

CCF

Complement the Carry Flag. If C=1 it will now be 0. If it was 0 it will now be 1.
You may have mistaken this for clear carry flag but it's not. If you want to clear the carry but change nothing else use "OR A".

Usage Example: CCF

Flags Affected: - - H - N C

CP r

Compare the Accumulator to register **r**.
This sets the flags the same as "SUB r" would, but the Accumulator is unchanged. This is like an IF statement in Basic.

Usage Example: CP B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

CP

Compare the Accumulator to 8 bit immediate value **#**.
This sets the flags the same as "SUB #" would, but the Accumulator is unchanged. Usually this command will be followed by a conditional branch like "JR Z,MatchedLabel".

Usage Example: CP 32 CP $5A

Flags Affected: S Z H V N C

Valid values for '#':0-255 ($00-$FF)

CPD

Compare A to the byte at address HL and decrease HL and BC.
This is the equivalent of the commands "CP (HL), DEC HL,DEC BC, PO set if BC=0".
This is like CPDR but without the 'repeat'.

Usage Example: CPD

Flags Affected: S Z H V N -

CPDR

Compare A to the byte at address HL and decrease HL and BC (Bytecount), and Repeat until a match occurs or BC=0.
This is the equivalent of the commands "CP (HL), DEC HL,DEC BC, until BC=0 or found".

Usage Example: CPDR

Flags Affected: S Z H V N -

CPI

Compare A to the byte at address HL and increase HL but decrease BC (Bytecount).
This is the equivalent of the commands "CP (HL), INC HL,DEC BC, PO set if BC=0".
This is like CPDR but without the 'repeat'.

Usage Example: CPI

Flags Affected: S Z H V N -

CPIR

Compare A to the byte at address HL and increase HL but decrease BC (Bytecount) and Repeat until a match occurs or BC=0.
This is the equivalent of the commands "CP (HL), INC HL,DEC BC, until BC=0 or found."
This is like CPIR but without the 'repeat'.

Usage Example: CPIR

Flags Affected: S Z H V N -

CPL

Invert all bits of A (this is known as 'One's Complement').
If A was equal to %11000000 it will now be %00111111.

Usage Example: CPL

Flags Affected: - - H - N -

DAA

If we're using Binary Coded Decimal, commands like ADD and SUB will of course 'break' our values. Adding $01 to the value $19 will cause the value $1A, not the $20 we need. DAA (Decimal Adjust Accumulator) fixes this, changing $1A to the correct $20.

Usage Example: DAA

Flags Affected: S Z H V - C

DEC r

Decrease value in 8 bit register **r** by one. This is faster than "SUB 1".

Usage Example: DEC B

Flags Affected: S Z H V N -

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

DEC rr

Decrease value in 16 bit register **rr** by one. Note: This does not set the flags.

Usage Example: DEC HL

Flags Affected: - - - - - -

Valid registers for 'rr': BC DE HL IX IY SP

DI

Disable Maskable Interrupts – in IM1 this stops calls to $0038. Note: Non Maskable interrupts cannot be stopped. You may want to disable interrupts to use Shadow Registers or to access hardware like a VDP.

Usage Example: DI

Flags Affected: - - - - - -

DJNZ

Decrease B and Jump if NonZero to address offset **#**. This is a relative jump like JR so must be close, the destination can only be -128 to +127 bytes away. This is usually used for loops, where B is the loop count.

Usage Example: DJNZ repeatlabel

Flags Affected: - - - - - -

EI

Enable Maskable Interrupts. In IM1 this resumes calls to $0038 when interrupts occur. On many systems this is once per frame at the start of VBlank.

Usage Example: EI

Flags Affected: - - - - - -

EX (SP),HL

Exchange HL with the top item of the stack (the item pointed to by address SP).
This may be used if we wish to alter the return address during a call.

Usage Example: EX (SP),HL

Flags Affected: - - - - - -

EX AF,AF'

Exchange the Accumulator and Flags with the shadow Accumulator and Flags.
This effectively backs up AF while they are used for another purpose and is faster than PUSH AF (usually used during interrupt handlers).

Usage Example: EX AF,AF'

Flags Affected: S Z H V N C

EX DE,HL

Exchange HL and DE. There's a lot of commands that only work on HL.
This is a quick way of allowing us to use those commands with the value in DE.

Usage Example: EX DE,HL

Flags Affected: S Z H V N C

EXX

Exchange the registers BC, DE and HL with the shadow registers BC' , DE' and HL' . This effectively backs up these registers for other purposes (usually used during interrupt handlers).
Note: There is no shadow IX or IY register – these must be pushed onto the stack.

Usage Example: EXX

Flags Affected: - - - - - -

HALT

Stop the CPU until an interrupt occurs. After the interrupt occurs, execution will resume after the HALT command. If interrupts were disabled by DI, then HALT will permanently stop the processor.

Usage Example: HALT

Flags Affected: - - - - - -

IM0

Enable Interrupt mode 0. In this mode a device causes an interrupt by putting a single byte command on the data bus (EG and RST).
This is not usable for programming on any system we're likely to encounter.

Usage Example: IM0

Flags Affected: - - - - - -

IM1

Enable Interrupt mode 1. In this mode an interrupt causes a CALL to &0038 (RST7).
The frequency of the interrupt depends on the system. On the MSX/Spectrum it occurs once per frame at VBLANK. On the CPC it occurs 6 times per frame.

Usage Example: IM1

Flags Affected: - - - - - -

IM2

Enable Interrupt mode 2. In this mode an address is formed by taking the top byte from the R register, and the bottom byte from the interrupting device. A 16 bit word is read from that address, and that address is called.

We can't effectively predict what the bottom byte will be, so using this command means reserving a block of 257 bytes.

The common solution is to fill memory addresses $8000-$8101 with the value $81. The effect of this is that all interrupts will call to $8181, and it's at that address we put the code for (or jump to) our actual interrupt handler.

This is not very convenient, and IM1 is far more useful. But on systems like the Spectrum, where low memory (Address &0038) is ROM, then this is the best solution for us.

Usage Example: IM2

Flags Affected: - - - - - -

IN A,(#)

Read in an 8 bit byte A from 8 bit address **#**. This command is used to read from external devices like a keyboard.

On systems which use 16 bit ports, like the CPC and Spectrum, the top byte of the port is taken from the previous value in the accumulator, so the effective address is ($AA##).
For Example: If A=$12 and we use the command "IN A,($34)" then A will be loaded from port ($1234) .

Usage Example: IN A,($10)

Flags Affected: S Z H V N -

Valid values for '#': 0-255 ($00-$FF)

IN r,(C)

On a system with 8 bit ports, this will read in an 8 bit byte into register **r** from port (C).
On a system with 16 bit ports (CPC/Spectrum/SAM), this will read in an 8 bit byte into register **r** from (BC). However the command is still "IN **r**,(C)" on these systems.

Usage Example: IN A,(C)

Flags Affected: S Z H V N -

Valid registers for 'r': A B C D E H L

INC r

Increase value in 8 bit register **r** by one. This is faster than 'ADD 1'.

Usage Example: INC B

Flags Affected: S Z H V N -

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

INC rr

Increase value in 16 bit register **r** by one. Note: This does not set the flags.

Usage Example: INC HL

Flags Affected: - - - - - -

Valid registers for 'rr': BC DE HL IX IY SP

IND

Read a byte IN from port (C) and save it to the address in HL, then Decrease HL and B. This is equivalent to the commands "IN (HL),(C) DEC HL DEC B".
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: IND

Flags Affected: s Z h v N -

INDR

Read a byte IN from port (C) and save it to the address in HL. Then Decrease HL and B, repeat until B=0.
This is equivalent to the commands "IN (HL),(C) DEC HL DEC B until B=0".
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: INDR

Flags Affected: s Z h v N -

INI

Read a byte IN from port (C) and save it to the address in HL, then increase HL and decrease B.
This is equivalent to the commands "IN (HL),(C) INC HL DEC B" .
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: INI

Flags Affected: s Z h v N -

INIR

Read a byte IN from port (C) and save it to the address in HL, then increase HL and decrease B, repeat until B=0.
This is equivalent to the commands "IN (HL),(C) INC HL DEC B until B=0".
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: INIR

Flags Affected: s Z h v N -

JP (HL)

Jump to the address in register HL. This effectively sets the program counter to HL.
Fun Fact: This is also the same effect as "PUSH HL RET".

Usage Example: JP (HL)

Flags Affected: - - - - - -

JP addr

Jump to the 16 bit address **addr**. This effectively sets the Program Counter to **addr**.

Usage Example: JP $4000 JP TestLabel

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)

JP c,addr

Jump to the 16 bit address **addr** only IF condition c is true in the flags register. This is like an IF THEN GOTO statement in Basic.

Condition	Meaning	Flag
JP Z,label	Jump to label if Zero	Z Set
JP NZ,label	Jump to label if Non Zero	Z Clear
JP C,label	Jump to label if Carry	C Set
JP NC,label	Jump to label if No Carry	C Clear
JP PO,label	Jump to label if Parity Odd	P/V Clear
JP PE,label	Jump to label if Parity Even	P/V Set
JP P,label	Jump to label if Positive Sign	S Clear
JP M,label	Jump to label if Minus Sign	S Set

Usage Example: JP Z,$4000 JP Z,TestLabel

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)
Valid conditions for 'c': c m nc nz p po pe z

JR ofst

Jump to the 8 bit offset **#**. This is added to the program counter. Usually **ofst** is a label, the assembler calculates the offset. This command takes less memory than JP.

Usage Example: JR TestLabel JR 16

Flags Affected: - - - - - -

Valid values for '#': -128 to +127

JR c,ofst

Jump to the 8 bit offset **ofst** IF condition **c** is true. Usually **ofst** is a label, the assembler calculates the offset.

Condition	Meaning	Flag
JR Z,label	Jump to label if Zero	Z Set
JR NZ,label	Jump to label if Non Zero	Z Clear
JR C,label	Jump to label if Carry	C Set
JR NC,label	Jump to label if No Carry	C Clear
JR PO,label	Jump to label if Parity Odd	P/V Clear
JR PE,label	Jump to label if Parity Even	P/V Set
JR P,label	Jump to label if Positive Sign	S Clear
JR M,label	Jump to label if Minus Sign	S Set

Usage Example: JR Z,TestLabel JR NC,16

Flags Affected: - - - - - -

Valid values for '#': -128 to +127
Valid conditions for 'c': c d nc nz z

LD (rr),A

Load the 8 bit value in the Accumulator into the address in register **rr**.

Usage Example: LD (DE),A

Flags Affected: - - - - - -

Valid registers for 'rr': BC DE HL IX+# IY SP

LD (rr),r

Load the 8 bit value in register **r** into the address in register **rr.**
rr can be HL IX+# or IY+#. You can even correctly load H or L from (HL). However the address in HL will be changed after the load.

Note: BC or DE cannot do this.

Usage Example: LD (HL),B LD (IX+3),D LD (IY-1),E

Flags Affected: - - - - - -

Valid registers for 'r': A B C D E H L
Valid registers for 'rr': HL IX+# IY+#

LD (addr),A

Load the 8 bit value in the Accumulator into memory address **addr**.
Note: The Accumulator is the only 8 bit register that can do this.

Usage Example: LD ($C000),A

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)

LD (addr),rr

Load the 16 bit value in register pair **rr** into memory address **addr**.

Usage Example: LD ($C000),BC

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)
Valid registers for 'rr': BC DE HL IX IY SP

LD A,(rr)

Load the 8 bit value from the address in register **rr** into the Accumulator.

Usage Example: LD A,(DE)

Flags Affected: - - - - - -

Valid registers for 'rr': BC DE HL IX+# IY SP

LD A,(addr)

Load the 8 bit value from memory address **addr** into the Accumulator.
Note: The Accumulator is the only 8 bit register that can do this.

Usage Example: LD A,($C000)

Flags Affected: - - - - - -

Valid values for '##': 0-65535 ($0000-$FFFF)

LD r,#

Load the 8 bit register **r** with value **#**.

Usage Example: LD B,32

Flags Affected: - - - - - -

Valid registers for 'r': A B C D E H L IXH IXL IYH IYL
Valid values for '#': 0-255 ($00-$FF)

LD A,I

Load the 8 bit value from the I register to the Accumulator.
The I register is only used in Interrupt Mode 2 (IM2), this can be used for 8 bit storage if you don't need IM2.

There is a 'quirk' of this function - the V flag is set to 1 if interrupts were enabled. This is the only way to check if interrupts are enabled or not.

Usage Example: LD A,I

Flags Affected: S Z H V N -

LD A,R

Load the 8 bit value from the R register to the Accumulator. The R register is used for refreshing memory and shouldn't be written to, but this may be useful for random number seeds.
Note: The R register will cycle between 0 and 127.

There is a 'quirk' of this function - the V flag is set to 1 if interrupts were enabled. This is the only way to check if interrupts are enabled or not.

Usage Example: LD A,R

Flags Affected: S Z H V N -

LD rr,(addr)

Load the 16 bit register pair **rr** from memory address **addr**.

Usage Example: LD BC,($C000)

Flags Affected: - - - - - -

Valid registers for 'rr': BC DE HL IX IY SP
Valid values for 'addr': 0-65535 ($0000-$FFFF)

LD rr,addr

Load the 16 bit register pair **rr** with immediate value **addr**.

Usage Example: LD BC,$C000

Flags Affected: - - - - - -

Valid registers for 'rr': BC DE HL IX IY SP
Valid values for 'addr': 0-65535 ($0000-$FFFF)

LD I,A

Load the 8 bit value from the Accumulator into the I register.
This command and "LD A,I" are the only commands to work with the I register.

Usage Example: LD I,A

Flags Affected: - - - - - -

LD R,A

*** WARNING *** The R register handles refreshing memory. Altering it could cause hardware damage!

Load the R register with the 8 bit value in the Accumulator.

Usage Example: LD R,A

Flags Affected: - - - - - -

LD SP,HL

Load the 16 bit Stack Pointer register SP with the value in HL.

Usage Example: LD SP,HL

Flags Affected: - - - - - -

LD r1,r2

Load the 8 bit register **r1** from register **r2**.
Note: You cannot transfer between HL, IX and IY directly. "LD H,L" or "LD IXH,IXL" is fine, but "LD IXH,IYL" is impossible.

Usage Example: LD H,B

Flags Affected: - - - - - -

Valid registers for 'r1' and 'r2': A B C D E H L IXH IXL IYH IYL

LD r,(rr)

Load the 8 bit register **r** from the address in register **rr**.
rr can be HL IX or IY. You can even correctly load H or L from (HL). However the address in HL will be changed after the load.

Usage Example: LD B,(HL) LD C,(IX+2) LD D,(IX-3)

Flags Affected: - - - - - -

Valid registers for 'r': A B C D E H L
Valid registers for 'rr': HL IX+# IY+#

LDD

Load and Decrement. This command copies bytes downwards from HL to DE with BC as a byte count (without repeat).
This command takes an 8 bit byte from the address in HL, stores it in the address in DE, and Decreases HL, DE and BC all by 1.

This command could be imagined as the commands "LD (DE),(HL) DEC HL DEC DE DEC BC"

Usage Example: LDD

Flags Affected: - - H V N -

LDDR

Load, Decrement and Repeat. This command copies bytes downwards from HL to DE with BC as a Byte count (with repeat).
This command takes an 8 bit byte from the address in HL, stores it in the address in DE, and Decreases HL, DE and BC all by 1. This is repeated until BC=0.

This command could be imagined as the commands: "LD (DE),(HL) DEC HL DEC DE DEC BC Until BC=0".

Usage Example: LDDR

Flags Affected: - - H V N -

LDI

Load and Increment. This command copies bytes upwards from HL to DE with BC as a byte count (without repeat).
This command takes an 8 bit byte from the address in HL, stores it in the address in DE, and Increases HL and DE by 1 and decreases BC by 1.

This command could be imagined as the commands: "LD (DE),(HL) INC HL INC DE DEC BC".
For a low byte count it is quicker to do a few LDI commands than a LDIR.

Usage Example: LDI

Flags Affected: - - H V N -

LDIR

Load, Decrement and Repeat. This command copies bytes upwards from HL to DE with BC as a byte count (with repeat).
This command takes an 8 bit byte from the address in HL, stores it in the address in DE, and Increases HL and DE by 1 and decreases BC by 1. This is repeated until BC=0.

This command could be imagined as the commands: "LD (DE),(HL) INC HL INC DE DEC BC Until BC=0"

Usage Example: LDIR

Flags Affected: - - H V N -

NEG

Negate the 8 bit value in the accumulator (Two's Complement of the number). This converts a positive number to a negative, or a negative to a positive.

Usage Example: NEG

Flags Affected: S Z H V N C

NOP

No Operation. This command has no effect on any registers or memory. "NOP" can be used as a short delay, or as a 'spacer' for bytes that will be altered via self modifying code.

Usage Example: NOP

Flags Affected: - - - - - -

OR r

Logical OR of bits in register **r** with Accumulator A. Where a bit in either A or **r** is 1 the result will be 1, when both are 0 the result will be 0.
For Example: If A=%10101010 and B=%11110000, the command "OR B" will result in A= %11111010 .

Usage Example: OR B OR A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

OR

Logical OR of bits in 8 bit value **#** with Accumulator A. Where a bit in A or **#** is 1 the result will be 1, when both are 0 it will be 0.
For Example: If A=%10101010 and we use the command "OR %11110000", the result will be A=%11111010 .

Usage Example: OR $64 OR A,%11110000

Flags Affected: S Z H V N C

Valid values for '#': 0-255 ($00-$FF)

OTDR

Out Decrement Repeat. This command transfers B bytes from HL to port (C) moving downwards.
It reads a byte from the address in HL and OUTs it to port (C), then Decreases HL and B, this will repeat until B=0.

This is equivalent to the commands: "OUT (C),(HL) DEC HL DEC B until B=0".

Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: OTDR

Flags Affected: s Z h v N -

OTIR

Out Increment Repeat. This command transfers B bytes from HL to port (C) moving upwards.
It reads a byte from the address in HL and OUTs it to port (C), then Increases HL and decreases B, this will repeat until B=0.
This is equivalent to the commands: "OUT (C),(HL) INC HL DEC B until B=0".
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: OTIR

Flags Affected: S Z H V N -

OUT (#),A

Output an 8 bit byte from A to 8 bit port **#**. This command is used to write to external devices like a sound chip.

On systems which use 16 bit ports, like the CPC and Spectrum, the top byte of the port is taken from the value in the accumulator, so the effective address is ($AA##).
For Example: If A=$12 and we use the command "OUT ($34),A" then A will be sent to port ($1234). Because this means the top byte of the port and the value sent are the same, this often won't be usable.

Usage Example: OUT ($10),A

Flags Affected: - - - - - -

Valid values for '#': 0-255 ($00-$FF)

OUT (C),r

On a system with 8 bit ports, this will output an 8 bit byte from register **r** to port (C) .
On a system with 16 bit ports (CPC/Spectrum/SAM), this will output an 8 bit byte from register **r** to port (BC). The command is still "OUT (C),r" though.

Usage Example: OUT (C),r

Flags Affected: - - - - - -

Valid registers for 'r': A B C D E H L

OUT (C),0

On a system with 8 bit ports, this will output an 8 bit byte zero to port (C). Note: Only zero is possible, not other numbers.
On a system with 16 bit ports (CPC/Spectrum/SAM), this will output an 8 bit byte zero to port (BC) . The command is still "OUT (C),0" though.

Usage Example: OUT (C),0

Flags Affected: - - - - - -

Valid registers for 'r': A B C D E H L

OUTD

Out and Decrement. This command transfers a byte from HL to port (C) moving downwards. It reads a byte from the address in HL and OUTs it to port (C), then Decreases HL and B.

This is equivalent to the commands:"OUT (C),(HL) DEC HL DEC B" .
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: OUTD

Flags Affected: s Z h v N -

OUTI

Out and Increment. This command transfers a byte from HL to port (C) moving upwards. It reads a byte from the address in HL and OUTs it to port ©, then Increases HL and decreases B.

This is equivalent to the commands "OUT (C),(HL) INC HL DEC B" .
Port (BC) is used on systems with 16 bit ports (CPC/Spectrum/SAM).

Usage Example: OUTI

Flags Affected: s Z h v N -

POP rr

Pop a pair of bytes off the stack into 16 bit register **rr**. Two bytes are taken from the top of the stack, and 2 is added to the SP register. This has the effect of 'restoring' the previous value of **rr**.

Usage Example: POP AF POP BC

Flags Affected when AF is popped: S Z H V N C
Flags Affected otherwise: - - - - - -

PUSH rr

Push a pair of bytes from 16 bit register **rr** onto the top of the stack. The two bytes from **rr** are put onto the top of the stack, and 2 is subtracted from the SP register. This has the effect of 'Backing up' the current value of **rr**.

Usage Example: PUSH AF PUSH BC

Flags Affected: - - - - - -

RES b,r

Reset bit **b** from 8 bit register **r** to 0. Bit 0 is the rightmost bit of **r** – bit 7 is the leftmost, so the byte's bits are numbered in the order %76543210.

Usage Example: RES 7,B RES 0,A

Flags Affected: - - - - - -

Valid values for 'b': 0-7 (%76543210)
Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

RET

Return from a subroutine. The return address is taken from the top of the stack and put in the program counter. This is effectively like a "POP PC" command.

Usage Example: RET

Flags Affected: - - - - - -

RET c

Return from a subroutine only if condition **c** is true. This is effectively like "IF THEN RETURN" in Basic.

Usage Example: RET z

Flags Affected: - - - - - -

Valid conditions for 'c': c m nc nz p po pe z

RETI

Return from an interrupt. This will signal devices that an interrupt is processed.
This command is only needed for interrupts on certain systems, it's not needed for IM1 on the CPC for example.

Usage Example: RETI

Flags Affected: - - - - - -

RETN

Return from a non maskable interrupt (NMI). This restores the state of interrupts (whether they were enabled or disabled) as they were before the NMI occurred.

Usage Example: RETN

Flags Affected: - - - - - -

RL r

Rotate bits in register **r** Left with Carry. The bits in the register are shifted left by 1, and the Carry acts like an 8th bit.
Let's see how bits change with multiple rotates:

Carry	76543210
0	11000010
1	10000100
1	00001001
0	00010011

There is a special version RLA which only works on the Accumulator, but is faster.

Usage Example: RL B RLA

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

RLC r

Rotate bits in register **r** Left and Copy the top bit to the Carry. The bits in the register are shifted left by 1, and the Carry copies the old 7th bit.
Let's see how bits change with multiple rotates:

Carry	76543210
0	11000010
1	10000101
1	00001011
0	00010110

There is a special version RLCA which only works on the Accumulator, but is faster.

Usage Example: RLC B RLCA

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

RLD

Rotate Left for binary coded Decimal. This command shifts the nibbles in the address HL left one nibble, and shifts the bottom nibble of A into the new position (like a Carry). The top nibble of A is unaffected.

For Example: If HL=$1000 and we use repeated RLD commands, A and the memory address $1000 will be affected in the following way:

A	($1000)
$01	$23
$02	$31
$03	$12

Usage Example: RLD

Flags Affected: S Z H V N -

RR r

Rotate bits in register **r** Right with carry. The bits in the register are shifted right by 1, and the Carry acts like an 8th bit.
Let's see how bits change with multiple rotates:

Carry	76543210
1	11000010
0	11100001
1	01110000
0	10111000

There is a special version RRA which only works on the accumulator, but is faster

Usage Example: RR B RRA

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

RRC r

Rotate bits in register **r** Right and Copy the top bit to the Carry. The bits in the register are shifted right by 1, and the Carry copies the old bit 0.
Let's see how bits change with multiple rotates:

Carry	76543210
1	11000010
0	01100001
1	10110000
0	01011000

There is a special version RLCA which only works on the accumulator, but is faster.

Usage Example: RLC B RLCA

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

RRD

Rotate Right for binary coded Decimal.
This command shifts the nibbles in the address HL right one nibble, and shifts the bottom nibble of A into the new position (like a Carry). The top nibble of A is unaffected.

For example: If HL=$1000 and we use repeated RRD commands, A and the memory address $1000 will be affected in the following way:

A	($1000)
$01	$23
$03	$12
$02	$31

Usage Example: RRD

Flags Affected: S Z H V N -

RST

ReSeT function. RST is a single byte call to $00xx address. Depending on the assembler the number **#** will either be 0,1,2,3,4,5,6,7 (WinApe style) or $0,$8,$10,$18,$20,$28,$30,$38 (VASM style).

RST **#** has the same function as CALL **#**, however it's a single byte command rather than 3 bytes for the call.

Usage Example: RST 7 (winape) RST $38 (vasm)

Flags Affected: - - - - - -

SBC r

Subtract register **r** and the carry flag from the Accumulator A. A can be specified as destination or omitted.

Usage Example: SBC B SBC A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

SBC A,#

Subtract 8 bit number **#** and the carry from A. A can be specified as destination or omitted.

Usage Example: SBC 128 SBC A,128

Flags Affected: S Z H V N C

Range of values for '#': 0-255 ($00-$FF)

SBC HL,rr

Subtract 16 bit register **rr** and the carry from HL.

Usage Example: SBC HL,BC

Flags Affected: S Z H V N C

Valid registers for 'rr': BC DE HL SP

SCF

Set the carry flag to 1. Note: There's no command to clear the carry - use "OR A".

Usage Example: SCF

Flags Affected: - - H - N C

SET b,r

Set bit **b** from 8 bit register **r** to 1. Bit 0 is the rightmost bit of r – bit 7 is the leftmost, so the byte's bits are numbered in the order %76543210.

Usage Example: SET 7,B

Flags Affected: - - - - - -

Valid values for 'b': 0-7 (%76543210)
Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

SLA r

Shift the bits register **r** Left for Arithmetic. 'Arithmetic' means bits as shifted left but the sign of the register is maintained. The Carry flag will be set to the old 7th bit, all new Bit 0s will be Zero.

SLA can be combined with RL to move bits between two registers.
For Example: "SLA L RL H" will double the 16 bit value in HL, as bits shifted out of L by SLA L (through the Carry) will be moved into H via RL H.

Carry	76543210
0	11000010
1	10000100
1	00001000
0	00010000

Usage Example: SLA A SLA B

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

SLL r

Shift the bits in register **r** Left Logically (for unsigned numbers). Carry flag will be the old 7th bit. Bit Zero will be set to 1.

Note: On most CPUs a Logical shift left would set the bottom bit to 0, but this one sets it to 1. This is an 'undocumented opcode' on the Z80 and is not supported on the eZ80.

Carry	76543210
0	11000010
1	10000101
1	00001011
0	00010111

Usage Example: SLL A SLL B

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

SRA r

Shift the bits in register **r** Right for Arithmetic. 'Arithmetic' means bits as shifted left but the sign of the register is maintained. The Carry flag will be the old bit 0. All new bit 7s will be one.

This can be combined with RR to halve a 16 bit signed number.
For Example: "SRA H RR L" will halve the 16 bit value in HL, as bits shifted out of H by "SRA L" (through the carry) will be moved into L via "RR L".

76543210	Carry
11000010	0
11100001	0
11110000	1
11111000	0

Usage Example: SRA A SRA B

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

SRL r

Shift the bits in register **r** Right Logically. 'Logical' shifts are intended for unsigned numbers. The Carry flag will be set to the old bit 0, bit 7 will be set to zero.

76543210	Carry
11000010	0
01100001	0
00110000	1
00011000	0

Usage Example: SRL A SRL B

Flags Affected: S Z H P N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

SUB r

Subtract 8 bit register **r** from A. The destination A can also be specified.

Usage Example: SUB B SUB A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

SUB

Subtract 8 bit value **#** from A. The destination A can also be specified.

Usage Example: SUB B SUB A,B

Flags Affected: S Z H V N C

Valid values for '#': 0-255 ($00-$FF)

XOR r

Logical XOR (eXclusive OR) of bits in register **r** with Accumulator A. Where a bit in **r** is 1 the bit in A will be flipped, Where a bit in r is 0 the bit in A will be unchanged.
For Example: If A=%10101010 and B=%11110000, "XOR B" will result in A=%01011010 .

Usage Example: XOR B XOR A,B

Flags Affected: S Z H V N C

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L IXH IXL IYH IYL

XOR

Logical XOR (eXclusive OR) of bits in immediate value **#** with Accumulator A. Where a bit in **#** is 1 the bit in A will be flipped, Where a bit in **#** is 0 the bit in A will be unchanged.
For Example: If A=%10101010, "XOR %11110000" will result in A=%01011010 .

Usage Example: XOR $64 XOR A,%11110000

Flags Affected: S Z H V N C

Valid values for '#':0-255 ($00-$FF)

Chapter 3: The 6502

Introducing the 6502

The 6502 is an 8 bit processor, While it usually has a lower clock speed than the Z80, it processes instructions more quickly, meaning a 2mhz 6502 is probably at least as fast as a 4 mhz Z80.

The 6502 has very few registers, and, while the accumulator is used to store the results of calculations, most mathematical operations will require reading and writing to memory for parameters and results.

The Zero page is ideal for this. Its addresses are specified by a single byte, taking less memory than a normal two byte address.

Compared to the Z80, the Zero Page can be thought of as our 'extra registers'. The tasks performed by registers like B, C and HL of the Z80 would be performed by Zero Page entries on the 6502.

The 6502 also does not have the same '16 bit' commands like the 'ADD HL,DE' on the Z80, but again the Zero Page can be used to give the same result.

For Example: The Z80 command "ADD HL,DE" could be written as the following on the 6502 using Zero Page address $01,$02 as the HL pair and $03,$04 as the DE pair.

Because the 6502 is a Little Endian CPU, $01 and $03 are the Low byte (L/E) and $02 and $04 are the High byte (H/D).

```
;Equivalent of ADD HL,DE
    clc         ;Clear the carry
    lda $01     ;Load Low Byte of first parameter (L)
    adc $03     ;Add Low Byte of Second Parameter (E)
    sta $01     ;Store Low Byte of first parameter (L)

    lda $02     ;Load High Byte of first parameter (H)
    adc $04     ;Add with carry High Byte of Second Parameter (D)
    sta $02     ;Store High Byte of first parameter (H)
```

Figure 54: The 6502 uses 8 bit registers, but we can use the 'Carry flag' to transfer the 'carry' during addition between two or more bytes to allow us to support 16 or more bit values.

The 6502 does not have special commands for accessing hardware ports. They are 'Memory Mapped', meaning they appear as addresses which can be used like normal RAM.

The 'successor' to the 6502 is the 65c02, which was extended with a few extra commands. There is also a 16 bit version, the 65816, and PC Engine's 6280 is also based on the 6502. The 65c02 and 65816 are backwards compatible with the 6502.

Special Addresses on the 6502

There are some special addresses on the 6502 processor that are common to all 6502 systems:

From	To	Purpose
$0000	**$00FF**	Zero Page
$0100	**$01FF**	Stack
...	**...**	...
$FFFA	**$FFFB**	NMI: Non Maskable Interrupt Vector
$FFFC	**$FFFD**	Reset Vector
$FFFE	**$FFFF**	IRQ: Interrupt Vector

The 6502 Registers

A	The Accumulator is used during all our main calculations. You will use it all the time when adding up (Accumulation).
P	The Processor status Flags – we don't tend to use this directly, it's set by mathematical operations, and we respond to its contents via conditional jumps and calls.
X	Index Registers are a relative offset to an address in the Zero page, or an absolute address, the resulting address+offset will be the source of our data. They can also be used as loop counters or temporary storage.
Y	Y is similar to X. It's an Index register for offsets, but the addressing modes available for Y and X are different for many commands.
PC	This is the place in memory that the 6502 is running. We don't change this directly, but JSR, JP and RTS all affect it.
S	This is the stack pointer. It points to a big temporary store that we'll use for 'backing up' values we need to remember for a short while. The Stack register is just 1 byte. On the 6502 the stack is always in the range $0100-$01FF, so the S register is the low byte, and $01 is the high byte of the final address. So if S=$43 then the stack pointer will actually be $0143.

The 6502 Flags

The 6502 Processor flags have 8 bits. The meanings of these flags are shown below.

Bit	Flag	Name	Description
7	**N**	Negative	Positive / Negative
6	**V**	Overflow	1=True. Overflow is when the sign changes, for example adding 1 to 127
5	**-**		
4	**B**	Break	BRK Command
3	**D**	Decimal mode	Used for Binary Coded Decimal
2	**P / V**	IRQ	1=Disable Interrupts
1	**Z**	Zero	Zero Flag (1=zero)
0	**C**	Carry	Carry / Borrow

In the instruction references of this book, the 'Flags Affected' section will show a minus '-' when an instruction leaves a flag unchanged, an uppercase letter when a flag is correctly updated (for example 'C'), and a lowercase letter when a flag is changed to an undetermined state (for example 'c').

Representing data types in source code

Here are the typical ways 6502 assemblers represent the different data types:

Prefix	Example	Meaning
#	#12	Immediate Decimal Value.
#%	#%10101010	Immediate Binary Value
#$	#$FF	Immediate Hexadecimal Value
#'	#'A'	Immediate ASCII Value
	1000	Memory Address in Decimal
$	$1000	Memory Address in Hexadecimal

As the 6502 is an 8 bit processor, we'll need to often split 16 bit values into two parts.
The 6502 has some special additions, the symbols "<" and ">" will do this.

"**<**" will take the low byte of a 16 bit part. For Example: "LDA #<$1234" will load $34 into A.
"**>**" will take the high byte of a 16 bit part. For Example: "LDA #>$1234" will load $12 into A.

Defining bytes of data on the 6502 in VASM

There will be many times when we need to define bytes of data in our code. Examples of this would be bitmap data, the score of our player, a string of text, or the co-ordinates of an object.

The commands will vary depending on the CPU and your assembler, but the ones shown below are the ones used by the assembler covered in these tutorials.

For Example: If we want to define a 16 bit sequence $1234, we would use "DW $1234".

Bytes	Z80	6502	68000	8086	ARM
1	DB	DB	DC.B	DB	.BYTE
2	DW	DW	DC.W	DW	.WORD
4			DC.L	DD	.LONG
n	DS n,x	DS n,x	DS n,x	n DUP (x)	.SPACE n,x

6502 Addressing Modes

The 6502 has a wide variety of addressing modes available. Many commands can support a range of these as the source or destination for an instruction. The 6502 documentation in this book specifies **<ea>** where there is a range of options, and the possible addressing modes will be listed.

If in doubt, just give it a go. If you put a command in your code with an invalid addressing mode for that command your assembler will refuse to compile it.

Implied - Imp

This is where one of the parameters is implied by the command, or there are no parameters at all.

Usage Example: DEX CLD BRK

Program Counter Relative - Rel

Relative addressing is where the parameter is an 8 bit signed byte offset (-128 to +127) relative to the current Program Counter position.

Usage Example: BEQ 16 BRA -100 {65c02}

Accumulator - Accum

This is where the only parameter is the Accumulator itself.
No parameter will be specified for **<ea>**.
For Example: "ROR $1000" will rotate the byte at $1000, but "ROR" with no parameter will rotate the Accumulator.

Usage Example: ROR ASL DEC {65c02}

Immediate - Imm

A single byte fixed number parameter is specified as a parameter after the Opcode. An immediate value will be specified by a leading hash symbol "#". The "$" symbol can be used to denote hexadecimal.

Usage Example: ADC #128 EOR #$80

Zero Page - ZeroPg

The source parameter or destination for the instruction will be an address in the range $0000-$00FF.
This is specified by a number WITHOUT a hash symbol. The number will be 0-255 (or $00-$FF in hexadecimal).

Usage Example: ADC 128 EOR $80

Zero Page Indexed with X - ZeroPg,X

Like 'Zero Page' Addressing, a single byte address is specified. However the value in the X register is added to this, and the resulting address will be used.
For Example: If X=5, command "LDA $10,X" will load the Accumulator from address $0015.

Usage Example: ADC 128,X EOR $80,X

Zero Page Indexed with Y - ZeroPg,Y

This is essentially the same as 'Zero Page Indexed with X', only uses the Y register. This addressing mode is only used by a couple of commands where the X register is the source or destination.

Usage Example: LDX 128,Y STX $80,Y

Absolute (Abs)

A 16 bit address, used as a parameter either for a jump or for the address to load or store a parameter. A 16 bit address is a value from 0-65535 in decimal or $0000-$FFFF in hexadecimal.

Usage Example: JMP $1000 LDA $C000

Absolute Indexed with X - Abs,X

A 16 bit address used as a parameter plus the value in the X register. The resulting address is used to load or store a parameter. The 16 bit address is a value from 0-65535 in decimal or $0000-$FFFF in hexadecimal.

For Example: If X=5, command "LDA $1000,X" will load the Accumulator from address $1005

Usage Example: LDA $1000,X STA $C000,X

Absolute Indexed with Y - Abs,Y

A 16 bit address used as a parameter plus the value in the Y register. The resulting address is used to load or store a parameter. The 16 bit address is a value from 0-65535 in decimal or $0000-$FFFF in hexadecimal.

For Example: If Y=5, command "LDA $1000,Y" will load the Accumulator from address $1005 .

Usage Example: LDA $1000,Y STA $C000,Y

Absolute Indirect (ind)

This is only available for JMP on the 6502 processor.
A 16 bit address used as a parameter. Two bytes are read from that address, and that becomes the address used to load or store a parameter. The address is a value from 0-65535 in decimal or $0000-$FFFF in hexadecimal.

For Example: Command "JMP ($1000)" will load two bytes from address $1000. Let's suppose that address contains $4000, then the JMP will occur to address $4000 .

Usage Example: JMP ($1000)

Absolute Indexed Indirect - (Ind Abs,X) {65c02}

This is only available for JMP on the 65c02 processor.
A 16 bit address used as a parameter plus the value in the X register. Two bytes are read from the resulting address, and that becomes the address used to load or store a parameter.

For Example: If X=5, command "JMP ($1000,X)" will load two bytes from address $1005 . Let's suppose that address contains $4000, then the JMP will occur to address $4000 .

Usage Example: JMP ($1000,X)

Zero Page Indexed Indirect - (Ind,X)

An 8 bit zero page address used as a parameter plus the value in the X register. Two bytes are read from the zero page address resulting address, and that becomes the address used to load or store a parameter.

For Example: If X=5, and we use "LDA ($10,X)", the CPU will perform the calculation $10+5 = $15. This will be used as a zero page address, so the CPU will load two bytes from address $0015. Let's suppose $0015/6 address contains $4000, in that case A will be loaded from memory address $4000.

Usage Example: LDA ($10,X) STA ($C0,X)

Zero Page Indirect Indexed with Y - (Ind),Y

An 8 bit zero page address used as a parameter. Two bytes are read from that zero page address and then Y is added to that address, and that becomes the address used to load or store a parameter.

For Example: If Y=5, and we use "LDA ($10),Y", the CPU will load two bytes from address $0010. Let's suppose $0010/1 contains $4000, in that case Y will be added to $4000, and A will be loaded from address $4005 .

Usage Example: LDA ($10),Y STA ($C0),Y

Compiling 6502 with VASM, and running VICE c64 emulator

Writing an assembler program

An ASM file is just a text file. You can edit it with the text editor of your choice. I use Notepad++, but you can use Windows Notepad, Visual Studio Code, or anything else you like.

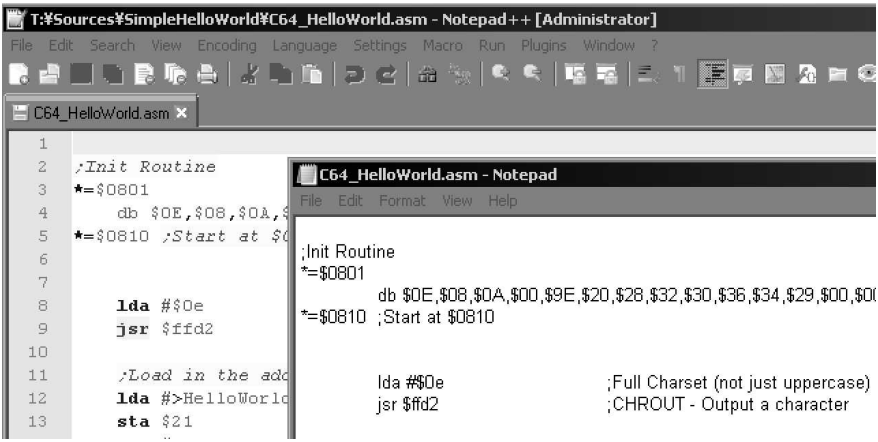


Figure 55: ASM files are just text files. Even classic Notepad can do the job, but an editor like Notepad++ offers syntax highlighting.

Using VASM

VASM is a fantastic Assembler for retro coding, as it supports 6502, Z80, 68000 and even ARM!

VASM is free, and can be found at:
http://sun.hasenbraten.de/vasm/

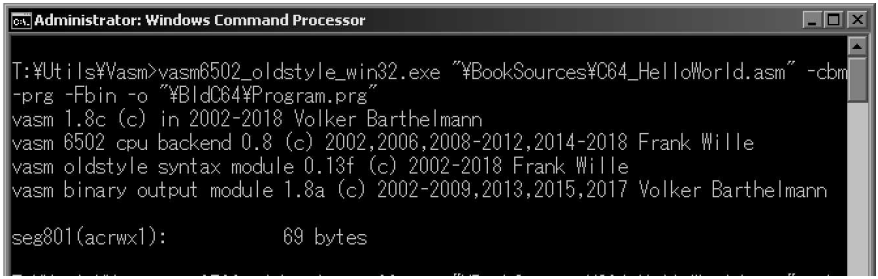
VASM is a Command line tool, we run it from a CMD prompt (Run CMD.exe in Windows).

Here is a minimal example of a script to compile an ASM file into a Commodore 64 program:

vasm6502_oldstyle_win32.exe "\BookSources\C64_HelloWorld.asm" -cbm-prg -Fbin -o "\BldC64\Program.prg"

This will compile the source file "C64_HelloWorld.asm" into a "PRG" file our c64 emulator can run. The built file will be saved to "\BldC64\Program.prg".

Here is a more detailed example with some extra features:



vasm6502_oldstyle_win32.exe "\BookSources\C64_HelloWorld.asm" -cbm-prg -chklabels -nocase -L Listing.txt -Fbin -o "\BldC64\Program.prg"

This will also compile the program, however this version has a few extra features that may help:
1. It will ignore case differences in labels, which may reduce your debugging.

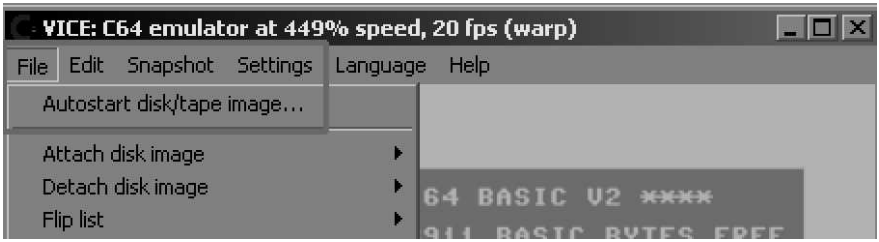
2. It will check for 'suspicious' label names. If you forget to put a tab before a command, it will be mistaken for a label, this chklabels switch will find those mistakes.

3. It will create a listing file called "Listing.txt". This is to help with debugging, it shows the source code and how the code compiles to bytes.

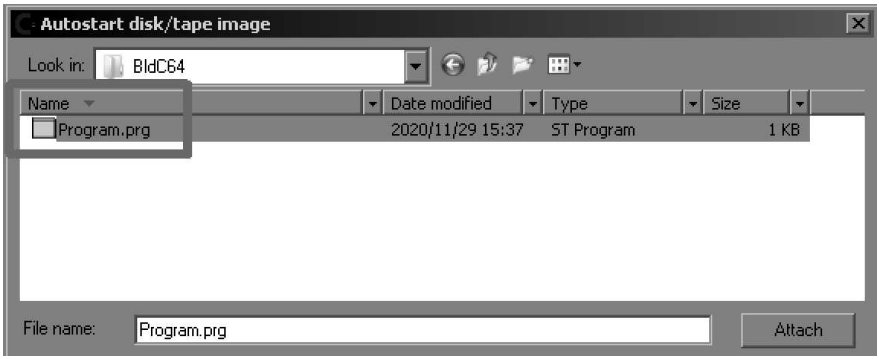
Either can be used to create the examples below, the second just has some helpful debugging options.

Running with VICE c64 Emulator

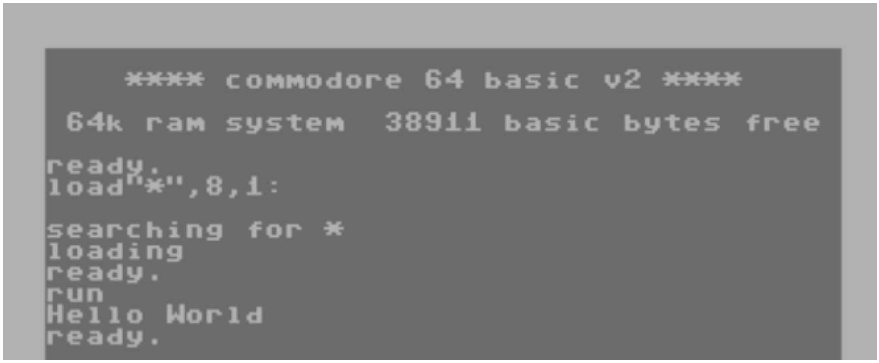
We're going to use VICE to run our compiled program. To start the program we just compiled, select the File menu – Autostart disk/tape image:



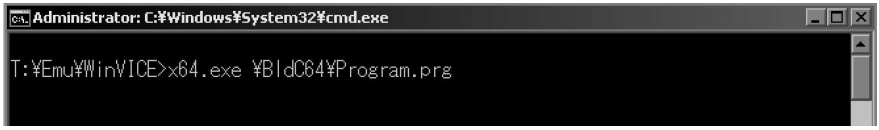
Select the PRG program we just compiled.



The emulator will reboot and run the program, the result is shown below:



You can also start a PRG from the command line:



6502 Examples

Example 1: Hello World

This Example will show a 'Hello World' Message on the C64.

Compile this with VASM in the way described on the previous pages.

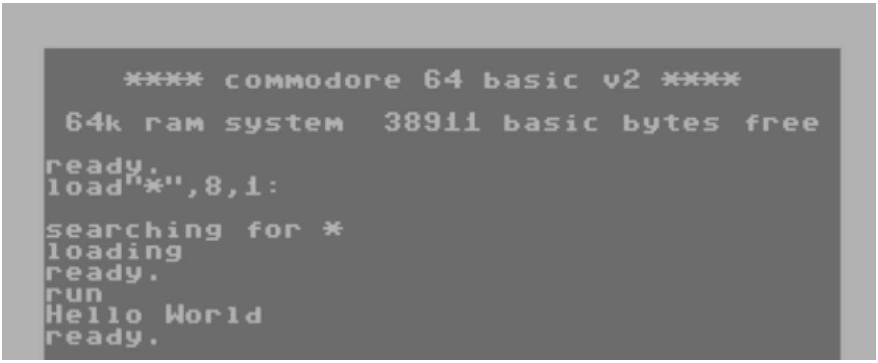
```
;Text after a semicolon is comments, you don't need to type it in

    org $0801           ;Header starts at address $0801

;Init Routine. This is the basic code for command to run our program
    db $0E,$08,$0A,$00,$9E,$20,$28,$32,$30,$36,$34,$29,$00,$00,$00

;Program Start at $0810

    lda #$0e            ;Full Charset (not just uppercase)
    jsr $ffd2           ;CHROUT - Output a character

;Load in the address of the Message into the zero page address $0020/21

    lda #>HelloWorld    ;> takes the High byte of a 16 bit address
    sta $21             ;H Byte of string address
    lda #<HelloWorld    ;< takes the Low byte of a 16 bit address
    sta $20             ;L Byte of string address

    jsr PrintStr        ;Show to the screen
    rts                 ;Return to basic

PrintStr:
    ldy #0              ;Reset Y
PrintStr_again:
    lda ($20),y         ;Read in a character
    cmp #255
    beq PrintStr_Done   ;Return if we've reached a 255
    jsr PrintChar       ;Print to screen if not
    iny
    jmp PrintStr_again  ;repeat
PrintStr_Done:
    rts                 ;Return when done

```

(Continued on the next page)

```
35  PrintChar:
36      cmp #64             ;Check if character >64
37      bcc PrintCharOKB
38      eor #%00100000      ;Convert to uppercase
39  PrintCharOKB:
40      jmp $FFD2           ;CHROUT - Output a character
41
42  HelloWorld:             ;Test string to show
43      db "Hello World"
44      db 255              ;End of string
```

To start this program, load it from the file menu as described.

You should see the 'Hello World' message as shown below:



How does it work?

Our program starts with a fixed header at &0801 . The actual code of our program starts at &0810. (Lines 3-6)

When we want to show a string in memory, we store the address of that string's address in two bytes of the Zero page. We use $20/$21 to store the two bytes, and we use the ">" and "<" symbols to tell the assembler to work out the high and low byte of the address. (Lines 15-18)

Because the 6502 is Little Endian the low byte of address "Hello World" goes in $20 and the high byte goes in $21.

We read in our string character by character, and sent them to our PrintChar routine. (Line 23-33)

The C64 does not use ASCII, so in our PrintChar routine we have to do some conversion of the character codes if the character is >64. (Line 35-39)

Once we have corrected the character codes, we use firmware function $FFD2 to show the character to the screen. (Line 40)

Example 2: Test Bitmap

This Example will show a Smiley bitmap on the C64.

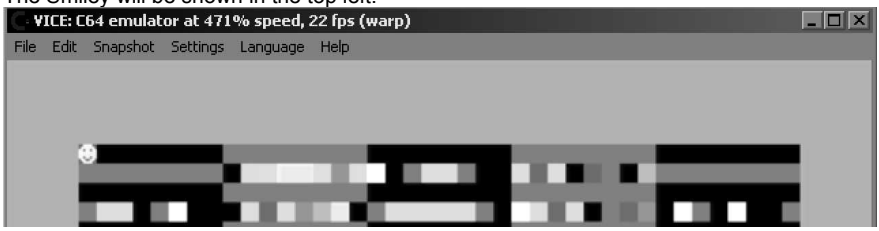
Compile this with VASM in the way described on the previous pages.

```
;Init Routine
    org $0801
    db $0E,$08,$0A,$00,$9E,$20,$28,$32,$30,$36,$34,$29,$00,$00,$00

;Set up our screen
        lda #%00111011  ;turn on graphics mode
        sta $D011

        lda #%11001000  ;Set 2 color mode
        sta $D016

        lda #%00011000  ;Set Screen base to $2000 (Color at $400)
        sta $D018

;Draw Bitmap Data
    lda #<Bitmap    ;Source Bitmap Data L byte
    sta $82
    lda #>Bitmap    ;Source Bitmap Data H byte
    sta $83

;Destination Screen address
    lda #<$2000     ;Dest L byte ($00)
    sta $80
    lda #>$2000     ;Dest H byte ($20)
    sta $81

BitmapNextLine:
    ldY #0          ;Offset for byte of graphic
BitmapNextByte:
    lda ($82),Y     ;Load in a byte from source offset with Y
    sta ($80),Y     ;Store it in screen ram offset with Y
    inY             ;INC the offset
    cpY #8          ;Our Smiley is 8 lines
    bne BitmapNextByte

    lda #$76        ;Yellow/Blue
    sta $400        ;Color in the smiley
    rts

```

(Continued on the next page)

```
40  ;1 bit per pixel smiley graphic
41  Bitmap:
42          db %00111100      ;  0
43          db %01111110      ;  1
44          db %11011011      ;  2
45          db %11111111      ;  3
46          db %11111111      ;  4
47          db %11011011      ;  5
48          db %01100110      ;  6
49          db %00111100      ;  7
```

Once compiled, you can start this program from the file menu as described before.
The Smiley will be shown in the top left.



How does it work?

We switched the screen into 2 color mode, and set the screen VRAM base to $2000. We needed to write the correct combination of bits to $D011,$D016 and $D018. (Lines 6-13)

We loaded in the address of our Bitmap data into zero page addresses $82/$83, and the destination address of the VRAM into $80/$81. (Lines 16-25)

We then transferred the 8 lines of our Smiley into VRAM, making it visible on the screen. (Lines 27-34)

We then set the matching block of color RAM at memory address $400, coloring it yellow. (Lines 36-37)

Example 3: Moving a sprite

This Example will show a Smiley bitmap on the C64.

Compile this with VASM in the way described on the previous pages.

```
1  Xpos equ $90         ;Zeropage entries for Current smiley position
2  Ypos equ $91
3
4  ;Init Routine
5      org $0801
6      db $0E,$08,$0A,$00,$9E,$20,$28,$32,$30,$36,$34,$29,$00,$00,$00
7      sei              ;Disable interrupts
8
9  ;Set up our screen
10     lda #%00111011   ;turn on graphics mode
11     sta $D011
12
13     lda #%11001000   ;Set 2 color mode
14     sta $D016
15
16     lda #%00011000   ;Set Screen base to $2000 (Color at $400)
17     sta $D018
18
19 ;Clear bitmap ram
20     lda #<$2000      ;Address to clear (Bitmap ram)
21     sta $80
22     lda #>$2000
23     sta $81
24     ldx #31          ;Bytes to clear *256
25     lda #0           ;Empty pixels
26     jsr Clear
27
28 ;Clear color ram
29     lda #<$400       ;Address to clear (Color ram)
30     sta $80
31     lda #>$400
32     sta $81
33     ldx #4           ;Bytes to clear *256
34     lda #$76         ;Yellow/Blue
35     jsr Clear
36
37     lda #10          ;Start pos
38     sta Xpos
39     sta Ypos
40
41     jsr ShowSprite   ;Show sprite
42
```

(Continued on the next page)

```
43  DrawLoop:
44      lda $DC00           ;Read in Joystick 1 ($DC01 = Joy2)
45
46      cmp #%11111111      ;%---FRLDU
47      beq DrawLoop        ;No Keys pressed?
48      sta $92
49
50      jsr ShowSprite      ;Remove old sprite
51
52      lda $92
53      and #%00000001      ;Test Up.
54      bne JoyNotUp
55      lda ypos
56      beq JoyNotUp        ;At top of screen?
57      dec ypos
58  JoyNotUp:
59
60      lda $92
61      and #%00000010      ;Test Down
62      bne JoyNotDown
63      lda ypos
64      cmp #24             ;At bottom of screen?
65      beq JoyNotDown
66      inc ypos
67  JoyNotDown:
68
69      lda $92
70      and #%00000100      ;Test Left
71      bne JoyNotLeft
72      lda xpos
73      beq JoyNotLeft      ;At Left of screen?
74      dec xpos
75  JoyNotLeft:
76
77      lda $92
78      and #%00001000      ;Test Right
79      bne JoyNotRight
80      lda xpos
81      cmp #39             ;At Right of screen.
82      beq JoyNotRight
83      inc xpos
84  JoyNotRight:
85
86      jsr ShowSprite      ;Show new position
87
88      ldy #0              ;Delay of 256
89      ldx #0              ;Delay of 256
90  Delay:
91      dex
92      bne Delay
93      dey
94      bne Delay
95      jmp DrawLoop        ;Repeat
```

(Continued on the next page)

```
 96
 97  ShowSprite:             ;Show XOR Sprite.
 98      jsr GetScreenPos
 99
100  ;Draw Bitmap Data
101      lda #<Bitmap        ;Source Bitmap Data L byte
102      sta $82
103      lda #>Bitmap        ;Source Bitmap Data H byte
104      sta $83
105  BitmapNextLine:
106      ldY #0              ;Offset for byte of graphic
107  BitmapNextByte:
108      lda ($82),Y         ;Load in a byte from source offset with Y
109      eor ($80),Y         ;EOR (XOR) with current screen value
110      sta ($80),Y         ;Store it in screen ram offset with Y
111      inY                 ;INC the offset
112      cpY #8              ;Our Smiley is 8 lines
113      bne BitmapNextByte
114      rts
115
116  ;Address= (X * 8) + (Top5Bits0fY * 40) + $2000
117  GetScreenPos:
118      lda #0
119      sta $84
120      sta $81
121      lda Xpos        ;Multiple X by 8
122      asl             ;-------- XXXXXXXX
123      rol $81
124      asl
125      rol $81
126      asl
127      rol $81         ;-----XXX XXXXX---
128      sta $80
129
130  ;40 bytes per Yline =00000000 00101000
131      lda Ypos        ;Ypos * 8
132      asl             ;(There are 8 lines per block)
133      asl
134      asl
135
136      asl
137      rol $84
138      asl
139      rol $84
140      asl             ;00000000 00101000
141      rol $84         ;00000YYY YYyyy000
142      tax
143          adc $80     ;Add part to total L
144          sta $80
145          lda $84     ;Add part to total H
146          adc $81
147          sta $81
```

(Continued on the next page)

```
148     txa
149     asl
150     rol $84
151     asl             ;00000000 00101000
152     rol $84         ;000YYYYY yyy00000
153
154     adc $80         ;Add part to total L
155     sta $80
156     lda $84         ;Add part to total H
157     adc $81
158     adc #$20+0      ;Screen Base $2000
159     sta $81
160     rts
161
162 Clear:              ;Clear X*256 bytes of ram
163     ldy #0
164 ClearAgain:
165     sta ($80),y
166     dey
167     bne ClearAgain
168     inc $81         ;INC High byte
169     dex
170     bne ClearAgain
171     rts
172
173 ;1 bit per pixel smiley graphic
174 Bitmap:
175     db %00111100     ;  0
176     db %01111110     ;  1
177     db %11011011     ;  2
178     db %11111111     ;  3
179     db %11111111     ;  4
180     db %11011011     ;  5
181     db %01100110     ;  6
182     db %00111100     ;  7
183
```

Once compiled, you can start this program from the file menu as described before.
The Smiley will be shown in the top left.

How does it work?

We've changed the Smiley routine to an "XOR" sprite routine. This uses EOR with the current byte from the screen. This means that if we draw the sprite twice, the second time will remove it from the screen. (Lines 97-114)

We've also added some 'Clearscreen' routines. These are used to clear the bitmap memory, and color memory, before we start drawing. (Lines 162-171)

We've now added a "GetScreenPos" function. This will convert an X/Y position into a VRAM address. We use this to calculate the position for drawing our Smiley. (Lines 117-160)

We read in the joystick from memory mapped port $DC00. This returns a byte with each bit 0-4 representing a direction in the format "%---FRLDU". (Line 44)

We don't want our sprite to go off the screen, so before each movement we check if we are already at the edge of the screen. (Lines 52-84)

Finally we update the sprite, then pause for a moment before repeating. (Lines 86-95)

Learning 6502: Where to go from here

The documentation examples we've covered here should be enough to get you started with your first 6502 program.

Over the following pages of this chapter, you'll be presented with all the 6502 instructions, and a description of their purpose, with examples.

If you're looking for more, there's lots more content on the ChibiAkumas website!

If you're looking for more information on the 6502 instruction set, there are step by step tutorials, with videos and source code on the 6502 page:
https://www.chibiakumas.com/6502/

We've only covered the C64 here. For details of the other systems covered, and detailed tutorials on the hardware of all the systems, take a look at the full list here:
https://www.assemblytutorial.com/

Don't forget to download the source code for this book from:
www.chibiakumas.com/book

And please subscribe to the official YouTube channel for weekly new videos:
https://www.youtube.com/chibiakumas

6502 Instructions

ADC <ea>

Add # and the carry flag to the Accumulator A
The 6502 has no ADD **<ea>** command, to simulate one use "CLC ADC **<ea>**".

Usage Example: ADC #61 ADC $40 ADC $1000

Flags Affected: N Z C - - V

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02}; (Ind,X), (Ind),Y

AND <ea>

Logical AND of bits in 8 bit value **#** with Accumulator A. Where a bit in A and # are 1 the result will be 1, when they are not it will be 0.
For Example: If A=%10101010 then "AND %11110000" will result in A=%10100000 .

Usage Example: AND #64 AND $12 AND $1000

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02} ; (Ind,X), (Ind),Y

ASL <ea>

Shift **<ea>** Left for Arithmetic. 'Arithmetic' means bits as shifted left but the sign of the register is maintained. The Carry flag will be the old 7th bit. All new Bit 0s will be Zero.

Carry	76543210
0	11000010
1	10000100
1	00001000
0	00010000

Note: The 6502 has no ASR. You can use the following commands to simulate one:

```
    BPL FakeASR_Bit0
    SEC             ;Top bit 1
FakeASR_Bit0:
    ROR
```

Usage Example: ASL ASL $12 ASL $1000

Flags Affected: N Z C - - -

Valid Addressing Modes for <ea>: Accum ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X

Bcc ofst

Branch to the 8 bit offset **ofst** IF condition cc is true.

Command	Description	Flags
BCC label	Branch to label if Carry Clear	C=0
BCS label	Branch to label if Carry Set	C=1
BEQ label	Branch to label if Equal (Zero)	Z=1
BNE label	Branch to label if Not Equal (NonZero)	Z=0
BMI label	Branch to label if MInus	N=1
BPL label	Branch to label if PLus	N=0
BVC label	Branch to label if oVerflow Clear	V=0
BVS label	Branch to label if oVerflow Set	V=1

Usage Example: BEQ TestLabel BCS 2

Flags Affected: - - - - - -

Valid values for 'ofst': -128 to +127

BIT <ea>

Test bits in Accumulator compared to memory address #.

This is like an AND statement – however it only sets the flags, A is unchanged.
When this command is used, the N flag is set to bit 7 of A, and V is set to bit 6.

Usage Example: BIT $61 BIT $6000

Flags Affected: N Z - - - V

Valid Addressing Modes for <ea>: Imm {65c02} ; ZeroPg ; ZeroPg,X {65c02} ; Abs ; Abs,X {65c02}

BRK

Stop the CPU and execute an interrupt. The address at $FFFE/F is called as a subroutine.

Usage Example: BRK

Flags Affected: - - - I - -

CLC

Clear the Carry Flag. C flag will be set to Zero.

Usage Example: CLC

Flags Affected: C - - - - -

CLD

Clear the Decimal Flag. D flag will be set to Zero. This turns off Binary Coded Decimal mode on the 6502.

Usage Example: CLD

Flags Affected: - - - - D -

CLI

Clear the Interrupt Flag. I flag will be set to Zero. This allows interrupts to execute normally (enabling interrupts).

Usage Example: CLI

Flags Affected: - - I - - -

CLV

Clear the oVerflow Flag. V flag will be set to Zero.

Usage Example: CLV

Flags Affected: - - - - V -

CMP <ea>

Compare the Accumulator to **<ea>**. This sets the flags the same as SUB # would, but the Accumulator is unchanged. This is like an IF statement in Basic.

Usage Example: CMP #10 CMP $32 CMP $2000

Flags Affected: N Z C - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02} ; (Ind,X), (Ind),Y

CPX <ea>

Compare the X register to **<ea>**. This sets the flags the same as "SBC **#**" would, but no registers are changed.

Usage Example: CPX #10 CPX $32 CPX $2000

Flags Affected: N Z C - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; Abs

CPY <ea>

Compare the Y register to **<ea>**. This sets the flags the same as "SBC **<ea>**" would, but no registers are changed.

Usage Example: CPY #10 CPY $32 CPY $2000

Flags Affected: N Z C - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; Abs

DEC <ea>

Decrease the 8 bit value **<ea>** by one.

Usage Example: DEC $10 DEC $10,X DEC $1000 DEC {65c02}

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Accum {65c02}; ZeroPg ; ZeroPg,X ; Abs ; Abs,X

DEX

Decrease the 8 bit value in register X by one. Combined with BNE, this command is useful as a loop counter.

Usage Example: DEX

Flags Affected: N Z - - - -

DEY

Decrease the 8 bit value in register Y by one. Combined with BNE, this command is useful as a loop counter.

Usage Example: DEY

Flags Affected: N Z - - - -

EOR <ea>

Logical EOR (Exclusive OR) of bits in **<ea>** with the Accumulator. Where a bit in **<ea>** is 1 the bit in A will be flipped. Where a bit in **<ea>** is 0 the bit in A will be unchanged.

For Example: If A=%10101010 and address $1000 contains %11110000. "EOR $1000" will result in A=%01011010.
Exclusive OR is known as XOR on some other CPUs.

Usage Example: EOR #10 EOR $20 EOR $2000

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Imp ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02} ; (Ind,X), (Ind),Y

INC <ea>

Increase the 8 bit value **<ea>** by one.

Usage Example: INC $10 INC $10,X INC $1000 INC {65c02}

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Accum {65c02}; ZeroPg ; ZeroPg,X ; Abs ; Abs,X

INX

Increase the 8 bit value in register X by one. Combined with BNE, this command is useful as a loop counter.

Usage Example: INX

Flags Affected: N Z - - - -

INY

Increase the 8 bit value in register Y by one. Combined with BNE, this command is useful as a loop counter.

Usage Example: INY

Flags Affected: N Z - - - -

JMP addr

Jump to the 16 bit address **addr**. This effectively sets the Program Counter to **addr**.

An Indirect address can be specified in brackets, eg JMP ($1000). The jump will occur to the address at address $1000.

On the 65c02 processor X, can be used as an offset, so a vector table can be placed at $1000 and X will be the offset in that table (it should be a multiple of 2).

Usage Example: JMP $4000 JMP TestLabel JMP ($1000)

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)

Valid Addressing Modes for 'addr': Abs ; (Ind Abs,X) {65c02} ; (Ind) {65c02}

JSR addr

Jump to Subroutine at address **addr**. This pushes the return address onto the stack, and changes the Program Counter to **addr**. This is effectively the equivalent of the GOSUB command in Basic.

Usage Example: JSR $1000 JSR TestLabel

Flags Affected: - - - - - -

Valid values for 'addr': 0-65535 ($0000-$FFFF)

Valid Addressing Modes for 'addr': Abs

LDA <ea>

Load the 8 bit value from **<ea>** into the Accumulator.

Usage Example: LDA #100 LDA $1000 LDA ($50,X)

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02} ; (Ind,X), (Ind),Y

LDX <ea>

Load the 8 bit value from **<ea>** into the X register.

Usage Example: LDX #100 LDX $1000 LDX $1000,Y

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ZeroPg,Y ; Abs ; Abs,Y

LDY <ea>

Load the 8 bit value from **<ea>** into the Y register.

Usage Example: LDY #100 LDY $1000 LDY $1000,Y

Flags Affected: N Z - - - -

Valid Addressing Modes for <ea>: Imm ; ZeroPg ZeroPg,X ; Abs ; Abs,X

LSR <ea>

Shift the bits of **<ea>** Right Logically. 'Logical' shifts are designed for unsigned numbers. The Carry flag will be the old bit 0, bit 7 will be set to zero.

76543210	Carry
11000010	0
11100001	0
11110000	1
11111000	0

Usage Example: LSR LSR $1000 LSR $60,X

Flags Affected: N Z C - - -

Valid Addressing Modes for <ea>: Accum ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X

NOP

No Operation. This command does nothing! It's effectively used as a short delay, or as a 'spacer' for bytes that will be altered via self modifying code.

Usage Example: NOP

Flags Affected: - - - - - -

OR <ea>

Logical OR of bits in 8 bit value **<ea>** with Accumulator A. Where a bit in A or # is 1 the result will be 1, when both are 0 it will be 0.
For Example: If A=%10101010 then "ORA #%11110000" will result in A=%11111010 .

Usage Example: ORA #61 ORA $40 ORA $1000

Flags Affected: N Z C - - V

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02}; (Ind,X), (Ind),Y

PHA

Push a byte from register A onto the top of the stack. The value in A is put onto the stack and 1 is subtracted from the S register, meaning the S register now points to the next unused byte of the stack. This has the effect of 'Backing up' the current value of A.

Usage Example: PHA

Flags Affected: - - - - - -

PHP

Push a byte from the processor flags register P onto the top of the stack. The value in P is put onto the stack and 1 is subtracted from the S register, meaning the S register now points to the next unused byte of the stack. This has the effect of 'Backing up' the current value of the P register.

Usage Example: PHP

Flags Affected: - - - - - -

PLA

Pull a byte off the stack into register A. One byte is taken from the top of the stack and put into the Accumulator, and 1 is added to the S stack register. This has the effect of 'restoring' the previous value of A.

Usage Example: PLA

Flags Affected: - - - - - -

PLP

Pull a byte off the stack into register A. One byte is taken from the top of the stack and put into the processor flags register P, and 1 is added to the S stack register. This has the effect of 'restoring' the previous value of the processor flags.

Usage Example: PLP

Flags Affected: N Z C I D V

ROL <ea>

Rotate bits of **<ea>** with Carry. The bits in the register are shifted left by 1. The old Carry becomes the new bit 0, and the old bit 7 becomes the new Carry.

Let's see how bits change with multiple rotates:

Carry	76543210
0	11000010
1	10000100
1	00001001
0	00010011

Usage Example: ROL ROL #61 ROL $40 ROL $1000

Flags Affected: N Z C - - -

Valid Addressing Modes for <ea>: Accum ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ;

ROR <ea>

Rotate bits of **<ea>** Right with Carry. The bits in the register are shifted right by 1. The old Carry becomes the new bit 7, and the old bit 0 becomes the new Carry.
Let's see how bits change with multiple rotates:

76543210	Carry
11000010	1
11100001	0
01110000	1
10111000	0

Usage Example: ROR ROR #61 ROR $40 ROR $1000

Flags Affected: N Z C - - -

Valid registers for 'r': (HL) (IX+#) (IY+#) A B C D E H L

RTI

Return from an interrupt. This command will restore the 8 bit processor flags register P from the stack before restoring the 16 bit Program counter, returning to the running code.

Usage Example: RTI

Flags Affected: N Z C I D V

RTS

Return from a subroutine. The return address is taken from the top of the stack and put in the program counter.

Usage Example: RTS

Flags Affected: - - - - - -

SBC <ea>

Subtract **<ea>** and the carry flag from the Accumulator A. The carry acts as a 'Borrow' in this case.
Note: There is no SUB command on the 6502, we can simulate a sub command with "SEC SBC **<ea>**".

Usage Example: SBC #61 SBC $40 ADC $1000

Flags Affected: N Z C - - V

Valid Addressing Modes for <ea>: Imm ; ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02}; (Ind,X), (Ind),Y

SEC

Set the carry flag to 1. The carry also acts as a 'borrow' during subtraction.

Usage Example: SEC

Flags Affected: - Z C - - -

SED

Set the Decimal Flag. D flag will be set to Zero, this turns on binary coded decimal mode on the 6502.With the Decimal flag set, ADC and SBC will produce binary coded decimal results, where each nibble stores a value from 0-9 rather than 0-F.

Usage Example: SED

Flags Affected: - - - - D -

SEI

Set the Interrupt Flag. I flag will be set to 1. This disables maskable interrupts.

Usage Example: SEI

Flags Affected: - - I - - -

STA <ea>

Store the 8 bit value in the Accumulator into memory address **<ea>**.

Usage Example: STA $10 STA $1000 STA ($50,X)

Flags Affected: - - - - - -

Valid Addressing Modes for <ea>: ZeroPg ; ZeroPg,X ; Abs ; Abs,X ; Abs,Y ; (ind) {65c02} ; (Ind,X), (Ind),Y

STX <ea>

Store the 8 bit value in the X register into memory address **<ea>**.

Usage Example: STX $10 STX $1000 STX $50,Y

Flags Affected: - - - - - -

Valid Addressing Modes for <ea>: ZeroPg ; ZeroPg,Y ; Abs

STY <ea>

Store the 8 bit value in the Y register into memory address **<ea>**.

Usage Example: STY $10 STY $1000 STY $50,X

Flags Affected: - - - - - -

Valid Addressing Modes for <ea>: ZeroPg ; ZeroPg,Y ; Abs

TAX

Transfer the 8 bit value in the Accumulator into register X. The Accumulator is unchanged.

Usage Example: TAX

Flags Affected: N Z - - - -

TAY

Transfer the 8 bit value in the Accumulator into register Y. The Accumulator is unchanged.

Usage Example: TAY

Flags Affected: N Z - - - -

TSX

Transfer the 8 bit value in the Stack pointer into register X. This, with TXS is the only way of changing or reading the value of the Stack pointer.

Usage Example: TSX

Flags Affected: N Z - - - -

TXA

Transfer the 8 bit value in the X register into the Accumulator. The X register is unchanged.

Usage Example: TXA

Flags Affected: N Z - - - -

TXS

Transfer the 8 bit value in the X Register into the Stack pointer. This is the only way of setting the Stack pointer.

Usage Example: TXS

Flags Affected: - - - - - -

TYA

Transfer the 8 bit value in the Y register into the Accumulator. The Y register is unchanged.

Usage Example: TYA

Flags Affected: N Z - - - -

65c02 Extra Instructions

The later 65c02 added a few extra commands we may wish to use on systems where they are available. These are also supported by the 65816.

BRA ofst

Branch always to the 8 bit offset **ofst** (without condition). This command is shorter than JMP by one byte, and 'relative' so can be used in relocatable code.

If you need a relative branch always on the 6502, you can force a condition set then branch on that condition.
For Example: "CLV BVC TestLabel" will always branch to relative offset testlabel.

Usage Example: BRA TestLabel BRA 2

Flags Affected: - - - - - -

Valid values for 'ofst': -128 to +127

PHX

Push a byte from register X onto the top of the stack. The value in X is put onto the stack and 1 is subtracted from the S register, meaning the S register now points to the next unused byte of the stack. This has the effect of 'Backing up' the current value of X.

If you need to push X onto the stack on the 6502, you must transfer it into A first.
For Example: On the 6502 you can use the commands "TXA PHA".

Usage Example: PHX

Flags Affected: - - - - - -

PHY

Push a byte from register Y onto the top of the stack. The value in Y is put onto the stack and 1 is subtracted from the S register, meaning the S register now points to the next unused byte of the stack. This has the effect of 'Backing up' the current value of Y.

If you need to push Y onto the stack on the 6502, you must transfer it into A first.
For Example: On the 6502 you can use the commands "TYA PHA".

Usage Example: PHY

Flags Affected: - - - - - -

PLX

Pull a byte off the stack into register X. One byte is taken from the top of the stack, and 1 is added to the SP register. This has the effect of 'restoring' the previous value of X.

If you need to pull X off the stack on the 6502, you must pull it into A, then transfer A to X. For Example: On the 6502 you can use the commands "PLA TAX".

Usage Example: PLX

Flags Affected: - - - - - -

PLY

Pull a byte off the stack into register Y. One byte is taken from the top of the stack, and 1 is added to the SP register. This has the effect of 'restoring' the previous value of Y.

If you need to pull Y off the stack on the 6502, you must pull it into A, then transfer A to Y. For Example: On the 6502 you can use the commands "PLA TAY".

Usage Example: PLY

Flags Affected: - - - - - -

STZ <ea>

Clear the 8 bit value in memory address **<ea>**
If you need to clear a byte on the 6502, you must do it through the Accumulator. For Example: On the 6502 you can use the commands "LDA #0 STA $1000"

Usage Example: STZ $10 STZ $1000 STZ ($50,X)

Flags Affected: - - - - - -

Valid Addressing Modes for <ea>: ZeroPg ; ZeroPg,X ; Abs ; Abs,X

Chapter 4: The 68000

Introducing the 68000

The 68000 is a 16 bit processor with 32 bit registers and a 24 bit address bus.

There are many later revisions, like the 68010 to 68030 and beyond, that added extra functionality. In this book we'll be covering the classic 68000, as used in the Mega Drive and Neo-Geo.

The 68000 has 16 main 32 bit registers, 8 Data registers, numbered D0-D7, and 8 Address registers, numbered A0-A7. Register A7 is a special case, it acts as the Stack Pointer (SP) on the 68000.

There is also a Program Counter (PC), a Status Register (SR), and a 'Supervisor Stack Pointer' (SSP) which replaces SP/A7 when the processor is in Supervisor mode, which only really happens during interrupt handlers.

Data registers are used for mathematical commands. Address registers are used for indirect addressing.

There is no 'Accumulator' on the 68000. All the Data registers have the same functionality, and A0-A6 all have the same functionality, though A7 is the only register used as a stack when calling subroutines.
Although the Address registers are 32 bit, many 68000 systems only have a 24 bit address bus.

The 68000 has different levels of privilege, and can switch between two modes.

Normal programs run in "User mode". In this mode the normal Stack Pointer is used (SP) and only the lowest 8 bits of the Status Register are accessible. These 8 bits are referred to as the Condition Code Register (CCR – the 68000 flags).

"Supervisor Mode" is the higher level, and is used by the operating system. It uses the alternative stack pointer, (the Supervisor Stack Pointer (SSP)) and the full 16 bits of the Status Register are accessible. The instruction set contains extra commands which can only be used in Supervisor Mode.

Hardware on the 68000 is 'memory mapped'. There are no special commands to access hardware, they appear as bytes of addressable memory which can be read or written according to their purpose.

On the 68000, when commands have two parameters, the parameter on the left is the source, the parameter on the right is the destination.
For Example: "ADD.L D1,D2" will add the source D1 to the destination D2.

Many commands can work at the Byte, Word or Long length. These are specified with a full stop and a B, W or L after the command.

For Example: The clear command "CLR" can be specified as "CLR.B", "CLR.W", or "CLR.L". If you do not specify a length, commands will default to word length.

Working with smaller values may be faster, but it's important to understand the 'unaffected' bytes of the register will be unchanged. If you use the command "MOVE.B #1,D1", the bottom byte of D1 will be set to $01, but the top 3 bytes could be anything (it will depend on the previous commands). They may be "$FFFFFF01", which could cause problems later. This is a common cause of problems in your code, so it's best to get in the habit of always specifying the length, even if you want the default Word length.

The 68000 can read 8 bit Bytes, 16 bit Words or 32 bit Longs from memory, but, due to limitations of the processor, Word or Long values can only be read from even byte addresses. You can read a Word from an address like $1000 or $1002, but not from an odd address like $1001. Bytes can be at any address. If you try to read or write a Word or Long with an odd address the processor will crash.

This is also true of the instruction commands, which must be even aligned. You need to take care after defining byte data in your code as the following code could be odd aligned, but you can fix this by adding the command "EVEN", which will tell the assembler to pad the code, if needed to align to the next even byte address.

The 68000 Registers

D0-D7	These are the Data registers, used for mathematical operations, they are all 32 bit.
A0-A6	These are the Address registers, they are all 32 bit, but on the original 68000 only 24 bits are used. They are used for addressing modes where the address is specified by a register. If you need to do complex calculations, you will need to do them in a Data register (D0-D7) then move them to the address register.
A7 / SP	Address register A7 acts as the Stack pointer, it can be referred to in your code as A7 or SP.
SSP	This register is swapped in to replace A7 / SP when the processor is in Supervisor Mode.
SR / CCR	The 68000 flags register. In Supervisor mode all 16 bits are accessible (referred to as the Status Register – SR), in User Mode only the lowest 8 bits are accessible (referred to as the Condition Code Register).

The 68000 Flags

While the Status Register is 16 bits, we can't access all of it in "User mode" (the mode our programs tend to run), - the top 8 bits are protected and can only be accessed in Supervisor mode, but we can access the first 8 bits anytime.
The User mode flags are called the CCR, the Condition Code Register.
The Supervisor mode flags are called the SR, the Status Register.

												CCR – Condition Code Reg				
	SR – Service Register															
Bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag	T	-	S	-	-	I	I	I	-	-	-	X	N	Z	V	C

Flag	Name	Description
T	Trace bit	1 if Trace enabled (Jump to address at Trace Vector $000024 every instruction for debugging)
S	Supervisor	1 if in supervisor mode, 0 in user mode
I	Interrupt	Interrupt level (0-7)
X	eXtend	Used for transferring data in/out of registers in rotation
N	Negative	1 if top-most bit of result is 1 (meaning the value is negative)
Z	Zero	1 if result of previous operation was zero
V	Overflow	1 if arithmetic overflow occurred (last operation caused accidental sign change)
C	Carry	1 if the last operation resulted in a Carry / borrow

In the instruction references of this book, the 'Flags Affected' section will show a minus '-' when an instruction leaves a flag unchanged, an uppercase letter when a flag is correctly updated (for example 'C'), and a lowercase letter when a flag is changed to an undetermined state (for example 'c').

68000 Condition codes (cc)

The 68000 has a variety of condition codes, which can be used in branches and some other commands.

cc	Description	Flags
CC	Carry clear	C=0
CS	Carry set	C=1
EQ	Equal	Z=1
GE	Greater than or equal	(N=1 & V=1) or (N=0 & V=0)
GT	Greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)
HI	Higher than	C=0 & Z=0
LE	Less than or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
LS	Lower than or same	C=1 or Z=1

LT	Less than	(N=1 and V=0) or (N=0 and V=1)
MI	Minus	N=1
NE	Not equal	Z=0
PL	Plus	N=0
T	True	=0
F	False	=1
VC	Overflow clear	V=0
VS	Overflow set	V=1

Representing data types in source code

Here are the typical ways 68000 assemblers represent the different data types:

Prefix	Example	Meaning
#	#12	Immediate Decimal Value.
#%	#%10101010	Immediate Binary Value
#$	#$FF	Immediate Hexadecimal Value
#'	#'A'	Immediate ASCII Value
	1000	Memory Address in Decimal
$	$1000	Memory Address in Hexadecimal

Defining bytes of data on the 68000 in VASM

There will be many times when we need to define bytes of data in our code. Examples of this would be bitmap data, the score of our player, a string of text, or the co-ordinates of an object.
The commands will vary depending on the CPU and your assembler, but the ones shown below are the ones used by the assembler covered in these tutorials.

For Example: If we want to define a 32 bit sequence $12345678 we would use "DC.L $12345678".

Bytes	Z80	6502	68000	8086	ARM
1	DB	DB	DC.B	DB	.BYTE
2	DW	DW	DC.W	DW	.WORD
4			DC.L	DD	.LONG
n	DS n,x	DS n,x	DS n,x	n DUP (x)	.SPACE n,x

68000 Addressing Modes

The 68000 CPU supports a wide range of addressing modes.
With the exception of Immediate Addressing, many of these modes can be used as the "effective address" for a parameter of many of the commands (denoted by **<ea>** in the following documentation).

Immediate Data Addressing

A fixed number is the parameter for an operation. The number starts with a # to mark it as an immediate (otherwise it will be mistaken for an address). $ can be used to specify hexadecimal, % can be used to specify binary.

Format Template: #n

Usage Example: MOVE.W #$1234,D1 MOVE.W #1234,D1

Absolute Data Addressing

A Memory address is the source or destination for an operation. Any number that does not start with a # will be treated as an address. $ can be used to specify hexadecimal.

Format Template: n

Usage Example: MOVE.W $1234,D1 MOVE.W 1234,D1

Data Register Direct Addressing

A Data register itself will be used as a source or destination of an operation.

Format Template: Dn

Usage Example: MOVE.W D3,D1 MOVE.W D2,D1

Address Register Direct Addressing

The Address register itself will be used as a source or destination of an operation. There are many commands where an Address register cannot be the destination of a complex mathematical operation, you need to use a Data register first.
For Example: "MULU D1,A2" will not work. You would need to use something like "MULU D1,D2 MOVE.L D2,A2".

Format Template: An

Usage Example: MOVE.W A3,D1 MOVE.W A2,D1

Address Register Indirect Addressing

This Addressing mode uses the value at the address pointed to by an Address register. For Example: If A1 contains $1000 and we use the command "MOVE.W (A1),D1" then D1 will be loaded from address $1000.

Format Template: (An)

Usage Example: MOVE.W (A3),D1 MOVE.W (A2),D1

Post-increment Register Indirect Addressing

This Addressing mode uses the value at the address pointed to by an Address register, then increases the value in an Address register by the amount being read or written. If this is used with A7/SP as the source this effectively pops a register off the stack.

Format Template: (An)+

Usage Example: MOVE.W (A3)+,D1 MOVE.W (SP)+,D1

Pre-decrement Register Indirect Addressing

Decrease the value in an Address register by the amount being read or written, then use the value pointed to by the register. If this is used with A7/SP as the destination this effectively pushes a register onto the stack.

Format Template: -(An)

Usage Example: MOVE.W -(A3),D1 MOVE.W D1,-(SP)

Register Indirect with Offset Addressing

This Addressing mode uses the value from the address in an Address register, offset by a fixed numeric value.

Format Template: (#,An)

Usage Example: MOVE.W (6,A3),D1 MOVE.W (2,A2),D1

Indexed Register Indirect with Offset Addressing

This Addressing mode uses the value from the address in an Address register, shifted by the offset of a numeric value plus the value in another register. The register offset can be Dn.W , Dn.L , An.W or An.L .

Format Template: (#,An,Dn) or (#,An,Am)

Usage Example: MOVE.W (6,A3,D4),D1 MOVE.W (2,A2,D5),D1

Program Counter Relative with Offset Addressing

Read from the current address plus a fixed value. Rather than a number, you'll probably want to use a label, and let the assembler calculate the "Fixed value". Because the address is relative to the current running code, compilation will result in relocatable code.

Format Template: (#,PC)

Usage Example: MOVE.W (6,PC),D1 MOVE.W (TestLabel,PC),D1

Program Counter Relative with Index & Offset Addressing

Read from the current address plus a fixed value plus a register. The Displacement can be zero if you only want to add a register.

Format Template: (#,PC,Dn) / (#,PC,An)

Usage Example: MOVE.W (6,PC,D4),D1 MOVE.W (TestLabel,PC,D5),D1

Compiling 68000 with VASM, and running with the FUSION emulator

Writing an assembler program

An ASM file is just a text file! You can edit it with the text editor of your choice. I use Notepad++, but you can use Windows Notepad, Visual Studio Code, or anything else you like.

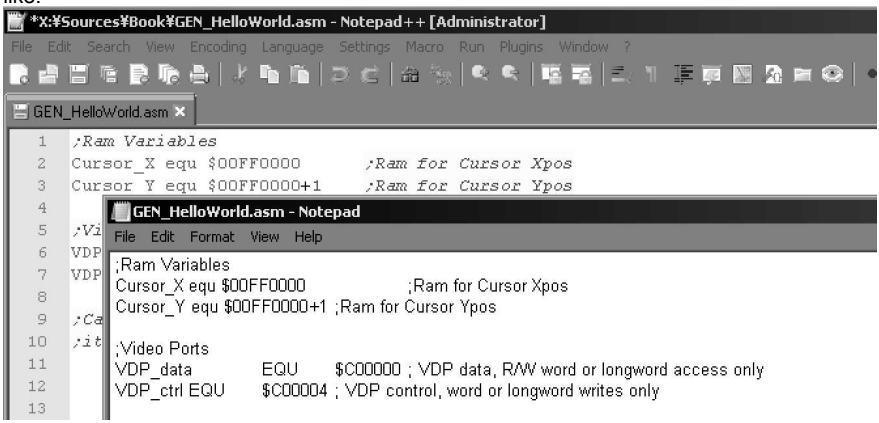


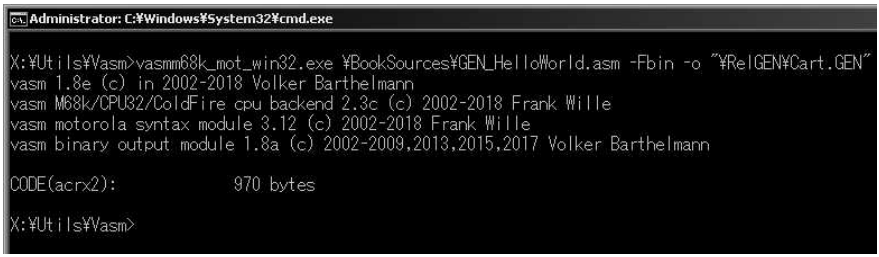
Figure 56: ASM files are just text files. Even classic Notepad can do the job, but an editor like Notepad++ offers syntax highlighting.

Using VASM

VASM is a fantastic Assembler for retro coding, as it supports 6502, Z80, 68000 and even ARM!

VASM is free, and can be found at:
http://sun.hasenbraten.de/vasm/

VASM is a Command line tool, we run it from a CMD prompt (Run CMD.exe in Windows).



Here is a minimal example of a script to compile an ASM file into a Genesis/Megadrive cartridge:

vasmm68k_mot_win32.exe \BookSources\GEN_HelloWorld.asm -Fbin -o "\RelGEN\Cart.GEN"

This will compile the source file "\BookSources\GEN_HelloWorld.asm" into a binary file our emulator can run. The built file will be saved to "\RelGEN\Cart.GEN".

Here is a more detailed example with some extra features:

vasmm68k_mot_win32.exe \BookSources\GEN_HelloWorld.asm -chklabels -nocase -L Listing.txt -Fbin -o "\RelGEN\Cart.GEN"

This will also compile the program, however this version has a few extra features that may help:
1.It will ignore case differences in labels, which may reduce your debugging.

2.It will check for 'suspicious' label names. If you forget to put a tab before a command, it will be mistaken for a label. This "chklabels" switch will find those mistakes.

3.It will create a listing file called "Listing.txt". This is to help with debugging, it shows the source code and how the code compiles to bytes.

Either can be used to create the examples below, the second just has some helpful debugging options.

Running with the Fusion 3.64 emulator

We're going to use Fusion to run our compiled program. To start the program we just compiled select "File menu – Load Genesis / 32X Rom".

Select the cartridge file we just compiled.

The emulator will reboot and run the program.

You can also start a cartridge from the command line.

68000 Examples

Example 1: Hello World

This Example will show a 'Hello World' Message on the Mega Drive (Genesis).
Compile this with VASM in the way described on the previous pages.

Please note: The example source below works fine with the Fusion 3.64 emulator. However the cartridge header is not technically correct, so may not work on other emulators.

The example has been cut down to the bare minimum to make a concise example. Please see the ChibiAkumas.com website for more complete examples.

```
;Ram Variables
Cursor_X equ $00FF0000      ;Ram for Cursor Xpos
Cursor_Y equ $00FF0000+1    ;Ram for Cursor Ypos

;Video Ports
VDP_data    EQU $C00000 ; VDP data, R/W word or longword access only
VDP_ctrl    EQU $C00004 ; VDP control, word or longword writes only

;Cartridge Header
;it works on Fusion 3.64 but is NOT technically valid

    DC.L    $FFFFFE00           ;Memory for Stack Pointer
    DC.L    ProgramStart        ;Start of our Program
    DS.L    62,InterruptHandler ;62*InterruptHandler

    ds.b    256,0

ProgramStart:

;Set up screen
    lea VDPSettings,A5      ;Initialize Screen Registers
    move.l #19,D1           ;Number of settings

    move.w (VDP_ctrl),D0    ;C00004 read VDP status (interrupt acknowledge?)
    move.l #$00008000,d5    ;VDP Reg command ($8rvv) R=Reg V=new Value

NextInitByte:
    move.b (A5)+,D5         ;get next video control byte
    move.w D5,(VDP_ctrl)    ;C00004 send write register command to VDP
        ;   8RVV - R=Reg V=Value
    add.w #$0100,D5         ;point to next VDP register
    dbra D1,NextInitByte    ;loop for rest of block

;Define palette (%----BBB-GGG-RRR-)
    move.l #$C0000000,VDP_Ctrl  ;Color 0
    move.w #%0000011000000000,VDP_data

    move.l #$C0020000,VDP_Ctrl  ;Color 1
    move.w #%0000000011101110,VDP_data

```

(Continued on the next page)

```
;Set up Font
    lea Font,A1                  ;Font Address in ROM
    move.l #26*8*4,d6            ;Our font contains 26 8 line 4 bytes per line

    move.l #$44200000,(VDP_Ctrl);Start writes to address $0420
                                 ;(Patterns in Vram)
NextFont:
    move.b (A1)+,d0              ;Get byte from font
    moveq.l #7,d5                ;Bit Count (8 bits)
    clr.l d1                     ;Reset BuildUp Byte

Font_NextBit:                    ;1 color per nibble = 4 bytes

    rol.l #3,d1                  ;Shift BuildUp 3 bits left
    roxl.b #1,d0                 ;Shift a Bit from the 1bpp font into the Pattern
    roxl.l #1,d1                 ;Shift bit into BuildUp
    dbra D5,Font_NextBit         ;Next Bit from Font

    move.l d1,(VDP_Data)         ;Write next Long of char (one line) to VDP
    dbra d6,NextFont             ;Loop until done

    clr.w Cursor_X               ;Clear Cursor XY

    move.w  #$8144,(VDP_Ctrl)    ;Turn on screen

    lea Message,a3               ;String to show
    jsr PrintString              ;Print String to screen

    jmp *                        ;Infinite Loop

PrintChar:                       ;Show D0 to screen
        and.l #%11011111,d0      ;Convert to uppercase
        sub.l #32,d0             ;No Characters in our font below 32
PrintCharAlt:
        Move.L  #$40000003,d5    ;top 4=write, bottom $3=Cxxx range (Tilemap)
        clr.l d4

        Move.B (Cursor_Y),D4
        move.l #8+8+7,d1
        rol.L d1,D4
        add.L D4,D5              ;add $4------3

        Move.B (Cursor_X),D4
        move.l #8+8+1,d1
        rol.L d1,D4
        add.L D4,D5              ;add $4------3

        MOVE.L  D5,(VDP_ctrl)    ; C00004 write next character to VDP
        MOVE.W  D0,(VDP_data)    ; C00000 store next word of name data

        addq.b #1,(Cursor_X)     ;INC Xpos
    rts

```

(Continued on the next page)

```
PrintString:
        move.b (a3)+,d0         ;Read a character in from A3
        cmp.b #255,d0
        beq PrintString_Done    ;return on 255
        jsr PrintChar           ;Print the Character
        bra PrintString
PrintString_Done:
    rts

InterruptHandler:               ;Basic Interrupt handler
    rte

Font:
        DC.B $18,$3C,$66,$66,$7E,$66,$24,$00     ;A
        DC.B $3C,$66,$66,$7C,$66,$66,$3C,$00     ;B
        DC.B $38,$7C,$C0,$C0,$C0,$7C,$38,$00     ;C
        DC.B $3C,$64,$66,$66,$66,$64,$38,$00     ;D
        DC.B $3C,$7E,$60,$78,$60,$7E,$3C,$00     ;E
        DC.B $38,$7C,$60,$78,$60,$60,$20,$00     ;F
        DC.B $3C,$66,$C0,$C0,$CC,$66,$3C,$00     ;G
        DC.B $24,$66,$66,$7E,$66,$66,$24,$00     ;H
        DC.B $10,$18,$18,$18,$18,$18,$08,$00     ;I
        DC.B $08,$0C,$0C,$0C,$4C,$FC,$78,$00     ;J
        DC.B $24,$66,$6C,$78,$6C,$66,$24,$00     ;K
        DC.B $20,$60,$60,$60,$60,$7E,$3E,$00     ;L
        DC.B $44,$EE,$FE,$D6,$D6,$D6,$44,$00     ;M
        DC.B $44,$E6,$F6,$DE,$CE,$C6,$44,$00     ;N
        DC.B $38,$6C,$C6,$C6,$C6,$6C,$38,$00     ;O
        DC.B $38,$6C,$64,$7C,$60,$60,$20,$00     ;P
        DC.B $38,$6C,$C6,$C6,$CA,$74,$3A,$00     ;Q
        DC.B $3C,$66,$66,$7C,$6C,$66,$26,$00     ;R
        DC.B $3C,$7E,$60,$3C,$06,$7E,$3C,$00     ;S
        DC.B $3C,$7E,$18,$18,$18,$18,$08,$00     ;T
        DC.B $24,$66,$66,$66,$66,$66,$3C,$00     ;U
        DC.B $24,$66,$66,$66,$66,$3C,$18,$00     ;V
        DC.B $44,$C6,$D6,$D6,$FE,$EE,$44,$00     ;W
        DC.B $C6,$6C,$38,$38,$6C,$C6,$44,$00     ;X
        DC.B $24,$66,$66,$3C,$18,$18,$08,$00     ;Y
        DC.B $7C,$FC,$0C,$18,$30,$7E,$7C,$00     ;Z

VDPSettings:
    DC.B 4,4,$30,$3C,$07,$6C,0,0,0,0,$FF,0,$81,$37,0,2,1,0,0

Message: dc.b 'Hello WorldZ',255

    even            ;Word align after byte data
```

The program will run in the emulator and show a 'Hello World' message:

How does it work?

Our program starts with a cartridge header at address $000000. (Lines 12-16)

Our program starts by setting up the screen, a sequence of bytes is sent to the "VDP_Ctrl" port, this sets up the graphics mode. (Lines 21-32, Settings on Line 134-135)

We also set up a palette. (Lines 34-39)

Our cartridge includes a 1 bit per pixel font (1 byte per line), each character uses 8 bytes. (Lines 106-132)

We then convert this into 4bpp tile patterns (32 bytes per character/Tile). These are stored in VRAM from memory address $0420 onwards (Tile 33). (Lines 41-60)

We then defined a PrintString and PrintChar routine.

The PrintString routine prints characters from the address in address register A3. (Line 94-101)

The PrintChar routine uses the defined patterns to show the correct character from the pattern definitions. (Lines 71-92)

As we only defined uppercase letters, some conversion is required. (Lines 72-73)

We show a tile to the screen by setting the correct tile position in the tilemap at VRAM address $C000+. Each tile is defined by one word. (Lines 75-86)

RAM on the Genesis starts at memory address $00FF0000+.

We use two addresses in RAM, "CursorX" and "CursorY", to remember the next tile position to write to. (Lines 2-3)

Example 2: Test Bitmap

This Example will show a Smiley as a Tile in the Tile map on the Genesis.
Compile this with VASM in the way described on the previous pages.

```
;Video Ports
VDP_data    EQU $C00000 ; VDP data, R/W word or longword access only
VDP_ctrl    EQU $C00004 ; VDP control, word or longword writes only

;Cartridge Header
;it works on Fusion 3.64 but is NOT technically valid

    DC.L    $FFFFFE00           ;Memory for Stack Pointer
    DC.L    ProgramStart        ;Start of our Program
    DS.L    62,InterruptHandler ;62*InterruptHandler

    ds.b    256,0

ProgramStart:

;Set up screen
    lea VDPSettings,A5      ;Initialize Screen Registers
    move.l #19,D1           ;Number of settings

    move.w (VDP_ctrl),D0    ;C00004 read VDP status (interrupt acknowledge?)
    move.l #$00008000,d5    ;VDP Reg command ($8rvv) R=Reg V=new Value

NextInitByte:
    move.b (A5)+,D5         ;get next video control byte
    move.w D5,(VDP_ctrl)    ;C00004 send write register command to VDP
        ;   8RVV - R=Reg V=Value
    add.w #$0100,D5         ;point to next VDP register
    dbra D1,NextInitByte    ;loop for rest of block

;Define palette (%----BBB-GGG-RRR-)
    move.l #$C0000000,VDP_ctrl  ;Color 0
    move.w #%0000011000000000,VDP_data

    move.l #$C0020000,VDP_ctrl  ;Color 1
    move.w #%0000000011101110,VDP_data

    MOVE.W  #$8144,(VDP_Ctrl)   ;C00004 reg 1 = 0x44 unblank display

;Define our test tile
    lea Bitmap,a0               ;Source data
    move.w #32,d1               ;Length of data (1 tile=32 bytes)

    move.l #$40200000,(VDP_ctrl);Start writes to address $0020

DefineTilesAgain:
    move.l (a0)+,d0
    move.l d0,(VDP_data)        ;Send the tile data to the VDP
    dbra d1,DefineTilesAgain


```

```
;Tile Settings
    move.l #3,d0    ;Xpos
    move.l #3,d1    ;Ypos
    move.l #1,d4    ;TileNumber

;Tile Addr: VRAM Addr = $C000 + (Ypos * 64* 2) + (Xpos *2)

;Calculate Tile Pos
    Move.L  #$40000003,d5       ;$C000 offset + Vram command
    clr.L d7
    Move.B d1,D7

    rol.L #8,D7                 ; Calculate Ypos
    rol.L #8,D7
    rol.L #7,D7
    add.L D7,D5

    move.B d0,D7                ;Calculate Xpos
    rol.L #8,D7
    rol.L #8,D7
    rol.L #1,D7
    add.L D7,D5

    move.l  D5,(VDP_ctrl)       ; C00004 Get VRAM address
    move.w  D4,(VDP_data)       ; C00000 Select tile for mem loc

    jmp *               ;Halt CPU

Bitmap: ;One nibble per pixel
    DC.B $00,$11,$11,$00    ;  0
    DC.B $01,$11,$11,$10    ;  1
    DC.B $11,$01,$10,$11    ;  2
    DC.B $11,$11,$11,$11    ;  3
    DC.B $11,$11,$11,$11    ;  4
    DC.B $11,$01,$10,$11    ;  5
    DC.B $01,$10,$01,$10    ;  6
    DC.B $00,$11,$11,$00    ;  7

VDPSettings:
    DC.B 4,4,$30,$3C,$07,$6C,0,0,0,0,$FF,0,$81,$37,0,2,1,0,0

    even            ;Word align after byte data
```

Once compiled, you can start this program from the file menu as described before.
The Smiley will be shown on the screen.

How does it work?

This example is actually simpler than the 'Hello World' as this time the pattern is in the native format of the Megadrive/Genesis.

We've included our 8x8 Smiley image, it's defined by one nibble per color, 4 bytes per line, 8 lines. (Lines 79-87)

We defined our Smiley as tile pattern number 1. We did this by writing to VRAM address $0020, 32 in hexadecimal. Each tile is 32 bytes in size. (Lines 39-48)

We then set a tile in the tile map to show this tile, by setting VRAM address $C000+ (this is the start of the tile map) to a word containing the tile number 1. (Lines 51-75)

This makes the tile visible on the screen.

Example 3: Moving a sprite

This Example will show a Smiley as a tile in the tile map on the Genesis, and allow us to move it around the screen with the joypad.

Compile this with VASM in the way described on the previous pages.

```
;Video Ports
VDP_data    EQU $C00000 ; VDP data, R/W word or longword access only
VDP_ctrl    EQU $C00004 ; VDP control, word or longword writes only

;Cartridge Header
;it works on Fusion 3.64 but is NOT technically valid

    DC.L    $FFFFFE00           ;Memory for Stack Pointer
    DC.L    ProgramStart        ;Start of our Program
    DS.L    62,InterruptHandler ;62*InterruptHandler
    ds.b    256,0

ProgramStart:

;Set up screen
    lea VDPSettings,A5      ;Initialize Screen Registers
    move.l #19,D1           ;Number of settings

    move.w (VDP_ctrl),D0    ;C00004 read VDP status (interrupt ack?)
    move.l #$00008000,d5    ;VDP Reg command ($8rvv) R=Reg V=new Value

NextInitByte:
    move.b (A5)+,D5         ;get next video control byte
    move.w D5,(VDP_ctrl)    ;C00004 send write register command to VDP
        ;   8RVV - R=Reg V=Value
    add.w #$0100,D5         ;point to next VDP register
    dbra D1,NextInitByte    ;loop for rest of block

;Define palette (%----BBB-GGG-RRR-)
    move.l #$C0000000,VDP_ctrl  ;Color 0
    move.w #%0000011000000000,VDP_data

    move.l #$C0020000,VDP_ctrl  ;Color 1
    move.w #%0000000011101110,VDP_data

    MOVE.W  #$8144,(VDP_Ctrl)   ;C00004 reg 1 = 0x44 unblank display

;Define our test tile
    lea Bitmap,a0               ;Source data
    move.w #32,d1               ;Length of data (1 tile=32 bytes)

    move.l #$40200000,(VDP_ctrl);Start writes to address $0020

DefineTilesAgain:
    move.l (a0)+,d0
    move.l d0,(VDP_data)        ;Send the tile data to the VDP
    dbra d1,DefineTilesAgain

```

```
49      move.l #3,d0                ;Xpos
50      move.l #3,d1                ;Ypos
51      jsr DrawSprite              ;Show new sprite
52
53  DrawLoop:
54      move.b #%01000000,($A10009) ;Set port direction IOIIIIII (I=In O=Out)
55      move.l #$A10003,a0          ;RW port for player 1 joypad
56
57      move.b  #$40,(a0)           ;Strobe port
58      nop                         ;Delay
59      nop
60      move.b  (a0),d3             ; Get buttons --CBRLDU
61
62      cmp.b #%00111111,d3         ;Test if any keys are pressed
63      beq drawloop
64
65      jsr ClearSprite     ;Remove old sprite
66
67      btst #0,d3
68      bne JoyNotUp        ;Jump if UP not pressed
69      tst.b d1            ;Check if at the top of the screen
70      beq JoyNotUp
71      subq.w #1,d1        ;Move Y Up the screen
72  JoyNotUp:
73
74      btst #1,d3
75      bne JoyNotDown      ;Jump if DOWN not pressed
76      cmp.b #27,d1        ;Check if at the bottom of the screen
77      beq JoyNotDown
78      addq.w #1,d1        ;Move Y DOWN the screen
79  JoyNotDown:
80
81      btst #2,d3
82      bne JoyNotLeft      ;Jump if LEFT not pressed
83      tst.b d0            ;Check if at the left of the screen
84      beq JoyNotLeft
85      subq.w #1,d0        ;Move X Left
86  JoyNotLeft:
87      btst #3,d3
88      bne JoyNotRight     ;Jump if RIGHT not pressed
89      cmp.b #39,d0        ;Check if at the right of the screen
90      beq JoyNotRight
91      addq.w #1,d0        ;Move X Right
92  JoyNotRight:
93
94      jsr DrawSprite      ;Show new sprite
95
96      move.l #$FFFF,d3    ;Pause
97  DelayLoop:
98      dbra d3,DelayLoop
99
100     jmp DrawLoop        ;Draw again
101
```

(Continued on the next page)

```
ClearSprite:
    move.l #0,d4    ;TileNumber (Clear)
    jmp DrawSpriteAlt
DrawSprite:
    move.l #1,d4    ;TileNumber (Smiley)

DrawSpriteAlt:
    Move.L  #$40000003,d5       ;$C000 offset + Vram command
    clr.L d7
    Move.B d1,D7

    rol.L #8,D7                 ; Calculate Ypos
    rol.L #8,D7
    rol.L #7,D7
    add.L D7,D5

    move.B d0,D7                ;Calculate Xpos
    rol.L #8,D7
    rol.L #8,D7
    rol.L #1,D7
    add.L D7,D5

    move.l  D5,(VDP_ctrl)       ; C00004 Get VRAM address
    move.w  D4,(VDP_data)       ; C00000 Select tile for mem loc
    rts

Bitmap: ;One nibble per pixel
    DC.B $00,$11,$11,$00    ;  0
    DC.B $01,$11,$11,$10    ;  1
    DC.B $11,$01,$10,$11    ;  2
    DC.B $11,$11,$11,$11    ;  3
    DC.B $11,$11,$11,$11    ;  4
    DC.B $11,$01,$10,$11    ;  5
    DC.B $01,$10,$01,$10    ;  6
    DC.B $00,$11,$11,$00    ;  7

VDPSettings:        ;Screen INIT settings
    DC.B 4,4,$30,$3C,$07,$6C,0,0,0,0,$FF,0,$81,$37,0,2,1,0,0

    even            ;Word align after byte data
```

Once compiled, you can start this program from the file menu as described before.

The Smiley will be shown on the screen. You will be able to move it with the joypad, within the boundaries of the screen.

How does it work?

We're using two tiles this time. Tile 1 is our Smiley, Tile 0 is a blank tile used to remove the old Smiley.

We read in from port $A10003 to get the basic joystick directions. It returns a byte in the format "%--CBRLDU". (Line 55)

However, before we can read from the joystick, we must configure the bits of this port with a write of #%01000000 to port $A10009. (Line 54)

We then test the bits of the byte returned by $A10003, checking each of the directions. (Lines 67-92)

When a direction is pressed, we check to see if the player's Smiley is already at the edge of the screen. If not, we can move, so we alter the player position accordingly.

We have a delay to keep the program running at a sensible speed, then we repeat the drawing loop. (Lines 96-100)

Learning 68000: Where to go from here

The documentation examples we've covered here should be enough to get you started with your first 68000 program.

Over the following pages of this chapter, you'll be presented with all the 68000 instructions, and a description of their purpose, with examples.

If you're looking for more, there's lots more content on the ChibiAkumas website!

If you're looking for more information on the 68000 instruction set, there are step by step tutorials, with videos and source code on the 68000 page:
https://www.chibiakumas.com/68000/

We've only covered the Mega Drive (Genesis) here. For details of the other systems covered, and detailed tutorials on the hardware of all the systems, take a look at the full list here:
https://www.assemblytutorial.com/

Don't forget to download the source code for this book from:
www.chibiakumas.com/book

And please subscribe to the official YouTube channel for weekly new videos:
https://www.youtube.com/chibiakumas

68000 Instructions

Note: Many extra commands were added with the later 68000 revisions. Only the original instruction set is covered here, as the 'basic 68000' commands will be sufficient for you to create most programs, and keeps the list short and focused on the essentials.

ABCD Dm,Dn
ABCD -(Am),-(An)

This command Adds two 8 bit Binary Coded Decimal numbers together with the eXtend bit acting as a carry. One Byte from Dm will be added to Dn, plus the eXtend bit. This uses 'Packed' BCD – meaning two digits per byte.
The Extend bit can be cleared with the command ANDI #%00001111,CCR
This command only works on a single byte, so only 8 bits of the registers are used.

Usage Example: ABCD D1,D2 ABCD.B (A1)+,(A2)+

Flags Affected: X n Z v C

Valid Lengths: B

ADD <ea>,Dn
ADD Dn,<ea>
ADDA <ea>,An
ADDI #,<ea>

This command Adds two numbers together.
For Example: The command "ADD <ea>,Dn" will result in <ea> being added to Dn.

Technically, the command ADD cannot work with immediate values or Address registers. The correct commands are the ADDI or ADDA commands, but if you use the 'technically incorrect' command "ADD #1,D1" or ADD D1,A1 your assembler will use the correct ADDI or ADDA command automatically.

Usage Example: ADD D1,D2 ADD.B #1,D3 ADD.W (A1)+,D2
ADDA.L D2,A1 ADDI.B #1,(A1)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

ADDQ #,<ea>

This command Adds a short immediate value **#** to **<ea>**. The immediate value **#** can be a value 1-8.
This command is quicker than ADDI and is effectively the 68000's INC command.

Usage Example: ADDQ #1,A1 ADDQ.B #3,D1 ADDQ.L #1,$1000

Flags Affected: X N Z V C

Valid Range for #: 1 to 8

Valid Lengths: B,W,L (Defaults to W)

ADDX Dm,Dn
ADDX -(Am),-(An)

This command Adds a data register to another data register with the eXtend bit.
There are two options, adding a Data register to another Data register, or (strangely) adding to the address in an address register with pre-decrement, to another address with pre-decrement.

This is effectively the 68000's ADC command (add With Carry).

Usage Example: ADDX D1,D2 ADDX.B D4,D1 ADDX -(A1),-(A2)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

AND <ea>,Dn
AND Dn,<ea>
ANDI #,<ea>

This command logically ANDs two numbers together. With "AND <ea>,Dn", the left hand parameter <ea> will be ANDed with the right hand parameter Dn. You can specify a length (if you do not it will default to Word length).

Logical AND will set bits in the destination according to the source and destination. Where a bit in the source and destination is 1 the result will be 1, when they are not it will be 0.
For Example: If D1=%10101010, "AND.B #%11110000,D1" will result in D1=%10100000 .

Technically, the AND command cannot work with immediate values. The correct command would be ANDI but if you use the 'technically incorrect' command AND #1,(A1) your assembler will use the correct ANDI command automatically.

Usage Example: AND D1,D2 AND.B (A2)+,D1 AND.B D1,(A2)+
ANDI #1,(A1)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

ANDI #,CCR

This command logically ANDs immediate value **#** with the Condition Code Register (CCR - the 68000 flags).

Bit	7	6	5	4	3	2	1	0
Flag	-	-	-	X	N	Z	V	C

This command is how we clear the flags on the 68000. We specify an immediate, the bit zeros in that immediate will clear the corresponding Flags in the CCR.

For Example: To clear the X flag: ANDI #%00001111,CCR
For Example: To clear the Z flag: ANDI #%00011011,CCR

Usage Example: ANDI #%00001111,CCR ANDI #%00011011,CCR

Flags Affected: X N Z V C

Valid Lengths: B

ANDI ##,SR

This command is only available in Supervisor Mode.
This command logically EORs immediate value **##** with the Status Register (the full 16 bit flags register).

Bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag	T	-	S	-	-	I	I	I	-	-	-	X	N	Z	V	C

Usage Example: ANDI #%00001111,SR ANDI #%00011011,SR

Flags Affected: X N Z V C

Valid Lengths: 16

ASL Dm,Dn
ASL #<data>,Dn
ASL <ea>

Shift the bits register **Dn** / **<ea>** Left for Arithmetic by **Dm** / **#** bits. 'Arithmetic' means the sign is maintained as the left shift occurs. The Carry+Extend flag will be the old top bit. All new Bit 0s will be Zero. When **<ea>** is specified, only one bit is shifted.

The bit in the Carry and Extend bit is never put into the destination. The top bit is moved into these flags so you can use it with another command.

Carry	Extend	76543210
0	0	11000010
1	1	10000100
1	1	00001000
0	0	00010000

Usage Example: ASL.W D1,D2 ASL.B #1,D1 ASL (A1)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

ASR Dm,Dn
ASR #<data>,Dn
ASR <ea>

Shift the bits register **Dn** / **<ea>** Right for Arithmetic by **Dm** / **#** bits. 'Arithmetic' means the sign is maintained as the right shift occurs. The Carry and eXtend flag will be the old bottom bit, All new top bits will be the same as the old top bit, meaning the sign stays the same. When **<ea>** is specified, only one bit is shifted.

The bit in the Carry and Extend bit is never put into the destination. The bottom bit is moved into these flags so you can use it with another command.

76543210	Carry	Extend
11000010	0	0
11100001	0	0
11110000	1	1
11111000	0	0

Usage Example: ASR.W D1,D2 ASR.B #1,D1 ASR (A1)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

Bcc

Branch to offset/Label **#** if the condition cc is true. If the condition is true **#** will be added to the Program Counter.

Bcc	Description	Flags
BCC	branch on carry clear	C=0
BCS	branch on carry set	C=1
BEQ	branch on equal	Z=1
BGE	branch on greater than or equal	(N=1 & V=1) or (N=0 & V=0)
BGT	branch on greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)
BHI	branch on higher than	C=0 & Z=0
BLE	branch on less than or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
BLS	branch on lower than or same	C=1 or Z=1
BLT	branch on less than	(N=1 and V=0) or (N=0 and V=1)
BMI	branch on minus	N=1
BNE	branch on not equal	Z=0
BPL	branch on plus	N=0
BVC	branch on overflow clear	V=0
BVS	branch on overflow set	V=1

Usage Example: BCC TestLabel BEQ 4

Flags Affected: - - - - -

Valid Lengths for #: B,W

BCHG Dn,<ea>
BCHG #,<ea>

Test Bit **Dn** / **#** of destination **<ea>**, set the zero flag to that bit, and flip that bit in **<ea>**.
For Example: If D0=%00001111 and we use command BCHG #1,D0, then D0 will equal %00001101 and Z=1.
For Example: If D0=%11110000 and we use command BCHG #1,D0, then D0 will equal %11110010 and Z=0.

Usage Example: BCHG D1,(A1) BCHG #1,D1

Flags Affected: - - Z - -

Valid Lengths: B,L

BCLR Dn,<ea>
BCLR #,<ea>

Test Bit **Dn** / **#** of destination **<ea>**, set the zero flag to that bit, and zero that bit in **<ea>.**
For Example: If D0=%00001111 and we use command BCLR #1,D0, then D0 will equal %00001101 and Z=1.
For Example: If D0=%11110000 and we use command BCLR #1,D0, then D0 will equal %11110000 and Z=0.

Usage Example: BCLR D1,(A1) BCLR #1,D1

Flags Affected: - - Z - -

Valid Lengths: B,L

BRA ofst

Branch Always to relative offset **ofst**. The value **ofst** is added to the program counter.

Usage Example: BRA TestLabel BRA 16

Flags Affected: - - - - -

Valid Range for ofst: -32768 to +32767

BSET Dn,<ea>
BSET #,<ea>

Test Bit **Dn** / **#** of destination **<ea>**, set the zero flag to that bit, and set that bit in **<ea>** to 1.
For Example: If D0=%00001111 and we use command BSET #1,D0, then D0 will equal %00001111 and Z=1.
For Example: If D0=%11110000 and we use command BSET #1,D0, then D0 will equal %11110010 and Z=0.

Usage Example: BSET D1,(A1) BSET #1,D1

Flags Affected: - - Z - -

Valid Lengths: B,L

BSR

Branch to Subroutine at relative offset **#**. The program counter is pushed onto the stack, and the value **#** is added to the program counter.

As it's a relative offset to the current address, this is the Relocatable version of JSR.

Usage Example: BSR TestLabel BSR 16

Flags Affected: - - - - -

Valid Lengths: B,W

BTST Dn,<ea>
BTST #,<ea>

Test Bit Dn or **#** of destination **<ea>** and set the zero flag to that bit.
For Example: If D0=%00001111 and we use command BTST #1,D0, then the Z flag will be set to 1.
For Example: If D0=%11110000 and we use command BSET #1,D0, then the Z flag will be set to 0.

Usage Example: BTST D1,(A1) BTST #1,D1

Flags Affected: - - Z - -

Valid Lengths: B,L

CHK <ea>,Dn
CHK #,Dn

Compare Dn to upper bound **#** or **<ea>**. If Dn is between Zero and the value, execution will continue normally, otherwise TRAP 6 will occur.
For Example: If D1=#100, and we use command "CHK #50,D1" execution will continue, but if we use "CHK #1000,D1" then TRAP 6 will occur.

Usage Example: CHK (A1),D1 CHK #1000,D1

Flags Affected: - N z v c

Valid Lengths: W, L {on 68020+}

CLR <ea>

Clear **<ea>** setting it to zero. This command is faster than "MOVE #0,<ea>".

Usage Example: CLR.B $1000 CLR D0 CLR.L D1

Flags Affected: - N Z V C

Valid Lengths: B,W,L (Defaults to W)

CMP <ea>,Dn
CMPA <ea>,An
CMPI #,<ea>
CMPM (Am)+,(An)+

CMP compares **<ea>** to **Dn**. This sets the flags the same as "SUB <ea>,Dn" would, but the **Dn** is unchanged. This is frequently used with a conditional branch like an IF statement in Basic.

Technically CMP cannot compare to an Address register, an immediate or two indirect addresses. The correct command would be CMPA, CMPI or CMPM but if you use the 'technically incorrect' command "CMP #1,A1" your assembler will use the correct CMPA command automatically.

Usage Example: CMP.B D0,D1 CMP.W (A1),D1 CMPA.W D1,A1
CMPA.L #$12345678,A1 CMPI.B #$FF,(A1)

Flags Affected: - N Z V C

Valid Lengths for CMP: B, W, L
Valid Lengths for CMPA: B, W, L
Valid Lengths for CMPI: B, W, L
Valid Lengths for CMPM: B, W, L

DBcc Dn,#

Decrease the value in register **Dn**, if **Dn** > -1 and condition **cc** is not true, execution will branch to relative address **#**.

Bcc	Description	Flags
DBRA	Decrement and Branch Always until -1	
DBCC	Decrement and Branch until -1 or carry clear	C=0
DBCS	Decrement and Branch until -1 or carry set	C=1
DBEQ	Decrement and Branch until -1 or equal	Z=1
DBGE	Decrement and Branch until -1 or greater than or equal	(N=1 & V=1) or (N=0 & V=0)
DBGT	Decrement and Branch until -1 or greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)
DBHI	Decrement and Branch until -1 or higher than	C=0 & Z=0
DBLE	Decrement and Branch until -1 or less than or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
DBLS	Decrement and Branch until -1 or lower than or same	C=1 or Z=1
DBLT	Decrement and Branch until -1 or less than	(N=1 and V=0) or (N=0 and V=1)
DBMI	Decrement and Branch until -1 or minus	N=1
DBNE	Decrement and Branch until -1 or not equal	Z=0

DBPL	Decrement and Branch until -1 or plus	N=0
DBT	Decrement and Branch until -1 or True (Always)	=1
DBF	Decrement and Branch until -1 or False (Never)	=0
DBVC	Decrement and Branch until -1 or overflow clear	V=0
DBVS	Decrement and Branch until -1 or overflow set	V=1

Usage Example: DBRA D0,TestLabel DBT D0,16

Flags Affected: - - - - -

Valid Lengths for #: B,W

DIVS <ea>,Dn

Divide Signed numbers. **Dn** is divided by **<ea>**. **Dn** is the dividend, **<ea>** is the divisor. Effectively **Dn=Dn** / **<ea>**.
The result is stored in **Dn**. The result in **Dn** is split into two Words in the format $RRRRQQQQ. The High Word is the Remainder, the Low Word is the Quotient (the integer result of the division).

Usage Example: DIVS #4,D1 DIVS (A1),D1

Flags Affected: - N Z V C

Valid Lengths: Long dividend, result is two Words

DIVU <ea>,Dn

Divide Unsigned numbers. **Dn** is divided by **<ea>**. **Dn** is the dividend, **<ea>** is the divisor. Effectively **Dn=Dn** / **<ea>**.
The result is stored in **Dn**. The result in **Dn** is split into two Words in the format $RRRRQQQQ. The High Word is the Remainder, the Low Word is the Quotient (the integer result of the division).

Usage Example: DIVS #4,D1 DIVS (A1),D1

Flags Affected: - N Z V C

Valid Lengths: Long dividend, result is two Words

EOR Dn,<ea>
EORI #,<ea>

Logical EOR (Exclusive OR) of bits in **Dn** or **#** with **<ea>**. Where a bit in **Dn** is 1 the bit in **<ea>** will be flipped. Where a bit in **Dn** is 0 the bit in **<ea>** will be unchanged.

For Example: If $1000=%10101010 and D1=%11110000 then EOR D1,$1000 will result in $1000=%01011010.

Technically, the EOR command cannot work with immediate values. The correct command would be EORI, but if you use the 'technically incorrect' command EOR #1,(A1) your assembler will use the correct EORI command automatically.

EOR is known as XOR on some other CPUs.

Usage Example: EOR #10,(A1) EOR #$20,D1 EOR D1,D2

Flags Affected: - N Z V C

Valid Lengths: B,W,L (Defaults to W)

EORI #,CCR

This command logically EORs an immediate **#** with the Condition Code Register (CCR - The 68000 flags).

Bit	7	6	5	4	3	2	1	0
Flag	-	-	-	X	N	Z	V	C

This command is how we flip the flags on the 68000. We specify an immediate, the bit 1s in that immediate will flip the corresponding flags in the CCR. As EOR flips the bits if the flag was 1 it will now be 0, if it was 0 it will now be 1.

For Example: To flip the X flag: EORI #%00010000,CCR
For Example: To flip the Z flag: EORI #%00000100,CCR

Usage Example: EORI #%00010000,CCR EORI #%00000100,CCR

Flags Affected: X N Z V C

Valid Lengths: B

EORI ##,SR

This command is only available in Supervisor Mode.
This command logically EORs immediate value **##** with the Status Register (the full 16 bit flags register).

Bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag	T	-	S	-	-	I	I	I	-	-	-	X	N	Z	V	C

Usage Example: EORI #%00001111,SR EORI #%00011011,SR

Flags Affected: X N Z V C

Valid Lengths: W

EXG Dn,Dm
EXG An,Am

Exchange the contents of registers **Dn** and **Dm**. No other registers or memory are affected. This command also works with Address registers.

Usage Example: EXG D1,D2 EXG A5,A2

Flags Affected: - - - - -

Valid Lengths: L

EXT Dn

Sign extend register **Dn**, either extending a Byte to Word, or a Word to a Long. Sign extending fills all unused bits with the top bit, making the register 'larger' but keeping its sign.

If D1=$11111108 and we use EXT.W D1 the byte will be extended to a Word with all bits set to zero (as bit, 7=0) and D1 will become $11110008.

If D1=$11111188 and we use EXT.W D1 the byte will be extended to a Word with all bits set to one (as bit 7=1) and D1 will become $1111FF88.

The same effect occurs when extending a Word to a Long. If D1=$11118888, after EXT.L D1 then D1 will become $FFFF8888.

Usage Example: EXT.W D1 EXT.L A5

Flags Affected: - N Z V C

Valid Lengths: W.L (Defaults to W)

ILLEGAL

This command will cause the processor to run the "Illegal Instruction Vector" (Trap 4).
It's almost certain you'll never need this command!

Flags Affected: - - - - -

JMP

Jump to absolute address **#**.
This command will not be relocatable. BRA is the relocatable version of JMP.

Usage Example: JMP TestLabel JMP $01000000

Flags Affected: - - - - -

Valid Lengths for #: L

JSR

Branch to Subroutine at absolute address **#**. The program counter is pushed onto the stack, and the Program Counter is set to **#**.
This command will not be relocatable. BSR is the relocatable version of JSR.

Usage Example: JSR TestLabel JSR $01000000

Flags Affected: - - - - -

Valid Lengths for #: L

LEA <ea>,An

Load the effective address **<ea>** into **An**.

Complex addressing modes like (2,A4,D0) or (D1,A1) have multiple stages to calculating the final address which will be used for the parameter. We may want to use this address many times, and LEA will 'precalculate' the effective address and store it in An.

This can be used to save CPU power, as using "LEA (2,A4,D0),A1" once and then using address (A1) twenty time in our code will be faster and more efficient than using (2,A4,D0) twenty times.

Usage Example: LEA (Label,PC),A1 LEA (4,A0),A2

Flags Affected: - - - - -

LINK An,#

There will be times we need some temporary memory for our work, and the Link command is designed to do this for us.

To use the LINK command we specify an address register **An** and an offset **#** (which should be negative).

The current value of **An** is pushed onto the stack (to back it up), then **An** is set to the current Stack Pointer. Finally, displacement **#** is added to the Stack Pointer. **An** now points to the 'Stack Frame' – the temporary memory allocated by this LINK command.

What does this mean? Well, **An** is now effectively a 'Temporary stack' with **#** bytes of free memory, while the Stack Pointer can be used normally.

When we use the "UNLK An" command the process is reversed, the Stack Pointer is restored from **An**, and the old value of **An** is restored from the Stack.

Valid Lengths for #: W (should really be 2 to 32768)

Usage Example: LINK A1,#-4 LINK A2,#-16

Flags Affected: - - - - -

LSL Dm,Dn
LSL #,Dn
LSL <ea>

Shift the bits in register **Dn** Left Logically by **Dm** or **#** bits.
If we were doing a Byte shift, the Carry and eXtend flag will be the old 7th bit. Bit Zero will be set to 0.

'Logical' Shifts are intended for unsigned numbers, this shift to the left will effectively double the number.

For Example: Let's look at the result of repeated logical shifts to an 8 bit value:

Extend	Carry	76543210
0	0	11000010
1	1	10000100
1	1	00001000
0	0	00010000

Usage Example: LSL #1,D1 LSL D2,D1 LSL (A1)

Flags Affected: X N Z V C

Valid Lengths: B, W, L (Defaults to W)

LSR Dm,Dn
LSR #,Dn
LSR <ea>

Shift the bits in register **Dn** Right Logically by **Dm** or **#** bits.
If we were doing a Byte shift the Carry and eXtend flag will be the old 0th bit. Bit 7 will be set to 0.
'Logical' Shifts are intended for unsigned numbers, this shift to the left will effectively double the number.

For Example: Let's look at the result of repeated logical shifts to an 8 bit value:

76543210	Extend	Carry
11000010	0	0
01100001	0	0
00110000	1	1
00011000	0	0

Usage Example: LSR #1,D1 LSR D2,D1 LSR (A1)

Flags Affected: X N Z V C

Valid Lengths: B, W, L (Defaults to W)

MOVE <ea>,<ea2>
MOVEA <ea>,An

Move the contents of source **<ea>** to the destination **<ea2>**. This is effectively a 'Copy command' as the contents of **<ea>** are unchanged.

Technically MOVE cannot transfer to an Address register. The correct command would be MOVEA but if you use the 'technically incorrect' command "MOVE #1,A1" your assembler will use the correct MOVEA command automatically.

This command can be used with the Stack Pointer and pre-decrement and post increment to push registers onto and pop registers off the stack.
For Example:
MOVE.L D0,-(SP) will push D0 onto the stack.
MOVE.L (SP)+,D0 will pop D0 off the stack.

Usage Example: MOVE #15,D1 MOVE (A1),D1 MOVEA D1,A1

Flags Affected: - N Z V C

Valid Lengths: B, W, L (Defaults to W)

MOVE <ea>,CCR

This command moves a 16 bit value from **<ea>** to the Condition Code Register (CCR - the 68000 flags).

Bit	7	6	5	4	3	2	1	0
Flag	-	-	-	X	N	Z	V	C

This command allows us to set all the condition codes.

For Example: To set only the X and N flags flag: MOVE #%00011000,CCR

For Example: To set all the flags: MOVE #%00011111,CCR

Usage Example: MOVE D0,CCR MOVE (A1),CCR

Flags Affected: X N Z V C

Valid Lengths: W

MOVE SR,<ea>
MOVE <ea>,SR

The "MOVE SR,<ea>" command moves the Status Register (the full 16 bit flags register) into destination **<ea>**.

The "MOVE <ea>,SR" command moves **<ea>** into the Status Register.
This command is only available in Supervisor Mode.

Bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag	T	-	S	-	-	I	I	I	-	-	-	X	N	Z	V	C

Usage Example: MOVE SR,D0 MOVE SR,(A1)

Flags Affected: X N Z V C

Valid Lengths: W

MOVE USP,An
MOVE An,USP

Transfer the User Stack Pointer (USP - User Mode SP) to or from address register **An**.
This command is only available in Supervisor Mode.

Usage Example: MOVE USP,A0 MOVE A0,USP

Flags Affected: - - - - -

Valid Lengths: L

MOVEM <ea>,<Regs>
MOVEM <Regs>,<ea>

The MOVEM command moves multiple registers to or from consecutive addresses in **<ea>**. **<Regs>** is a list of registers to move, it supports ranges and individual registers.

Ranges can be specified with a "-". For Example: "D0-D4" will transfer D0,D1,D2,D3 and D4.

Individual registers, or multiple ranges can be separated with a "/". For Example: "D1-D3/A4/A6" will transfer D1,D2,D3,A4 and A6.

The order does not have an effect on the function, For Example "D1/D3/D2" has the same effect as "D1-D2".

This command is frequently used to push multiple registers on to the stack or pop them off. For Example:
"MOVEM.L (SP)+,D0-D7/A0-A7" will back up every single register onto the stack.
"MOVEM.L D0-D7/A0-A7,-(SP)" will restore every single register from the stack.

These are great to use at the start and end of a subroutine to ensure no registers are changed!

Usage Example: MOVEM.L (A1),D0/D3 MOVEM.B D0-D3/D5,$10000

Flags Affected: - - - - -

Valid Lengths: B,W,L (Defaults to W)

MOVEP Dn,(#,An)
MOVEP (#,An),Dn

MOVEP stands for 'Move Peripheral data'. This is designed for moving 16 or 32 bits to a set of memory mapped byte data ports.

This sounds rather confusing, but the function is simple. The bytes of **Dn** will be transferred to the even memory addresses and the odd ones will be unused (unchanged on write, or ignored on read).
For Example: Assume A0=$FF0100 and D0=$11223344, if we use the command "MOVEP.L D1,(A1)" the results in RAM from $FF0100-$FF0107 will be:

Address $FF01xx+	00	01	02	03	04	05	06	07
Data	11	00	22	00	33	00	44	00

Usage Example: MOVEP.L D0,(4,A1) MOVEP.W (A2),D1

Flags Affected: - - - - -

Valid Lengths: W,L (Defaults to W)

MOVEQ #,Dn

This command adds short immediate **#** to the register **Dn**. The value **#** is a single byte so can be -128 to +127, though it is sign extended and added as a Long.
This command is faster than a regular move, and is effectively the INC or DEC command of the 68000.

Usage Example: MOVEQ #1,D1 MOVEQ #-4,D1

Flags Affected: - - - - -

Valid Range for #: -127 to +128

Valid Lengths: L

MULS <ea>,Dn

Multiply Signed numbers. **Dn** is multiplied by **<ea>**. **Dn** is the multiplicand, **<ea>** is the multiplier, the result is stored in **Dn**. Effectively the calculation is **Dn=Dn*<ea>**.
The source multiplicand and multiplier are 16 bit. The result in the destination is 32 bit.

Usage Example: MULS #4,D1 MULS (A1),D1

Flags Affected: - N Z V C

Valid Lengths: Both source parameters are Words, the result is a Long

---

MULU <ea>,Dn

Multiply Unsigned numbers. **Dn** is multiplied by **<ea>**. **Dn** is the multiplicand, **<ea>** is the multiplier, the result is stored in **Dn**. Effectively the calculation is **Dn=Dn*<ea>**.
The source multiplicand and multiplier are 16 bit. The result in the destination is 32 bit.

Usage Example: MULS #4,D1 MULS (A1),D1

Flags Affected: - N Z V C

Valid Lengths: Both source parameters are Words, the result is a Long

---

NBCD Dn

This command Negates a Binary Coded Decimal byte in **Dn**. This command also subtracts the eXtend flag.
The formula is effectively **Dn**=(0-**Dn**)-{X flag} (in Binary Coded Decimal values).

The Extend bit can be cleared with the command ANDI #%00001111,CCR

For Example: If D0=$13 and X=0, after we use the command "NBCD D0", D0 will equal $87.

Usage Example: NBCD D1 NBCD D2

Flags Affected: X n Z v C

Valid Lengths: B

NEG <ea>

Negate **<ea>** (Two's Complement of the value). This converts a positive number to a negative, or a negative to a positive.

Usage Example: NEG D0 NEG (A1) NEG $1000

Flags Affected: X N Z V C

Valid Lengths: B W L (Defaults to W)

NEGX <ea>

Negate **<ea>** (Two's Complement of the value) with the eXtend flag. If we use the command "NEG D1" then the formula used is effectively D1=(0-D1)-{X flag}.

Usage Example: NEGX D0 NEG (A1) NEG $1000

Flags Affected: X N Z V C

Valid Lengths: B W L (Defaults to W)

NOP

No Operation. This command has no effect on any registers or memory. NOP can be used as a short delay, or as a 'spacer' for bytes that will be altered via self modifying code.

Usage Example: NOP

Flags Affected: - - - - - -

NOT <ea>

Invert/Flip all the bits of **<ea>**. This is known as One's Complement.
For Example: If D0=%11000000 and we use "NOT.B D1" then D1 will become %00111111.

Usage Example: NOT.L D1 NOT.W (A1) NOT.L $10000

Flags Affected: - N Z V C

Valid Lengths: B W L (Defaults to W)

OR <ea>,Dn
OR Dn,<ea>
ORI #,<ea>

This command logically ORs two numbers together. With "OR **<ea>**,**Dn**", the left hand parameter **<ea>** will be ORed with the right hand parameter **Dn**. You can specify a length – if you do not it will default to Word length.

Logical OR will set bits in the destination according to the source and destination. Where a bit in either parameter is 1 the result will be 1, when both are 0 the result will be 0.

For Example: If D0=%10101010 and D1=%11110000, the command "OR D0,D1" will result in D1=%11111010.

Technically, the OR command cannot work with immediate values. The correct command would be ORI, but if you use the 'technically incorrect' command OR #1,(A1) your assembler will use the correct ORI command automatically.

Usage Example: OR D1,D2 OR D1,(A1) ORI #%11110000,$1000

Flags Affected: - N Z V C

Valid Lengths: B W L (Defaults to W)

ORI #,CCR

This command logically ORs immediate value **#** with the Condition Code Register (the flags).

Bit	7	6	5	4	3	2	1	0
Flag	-	-	-	X	N	Z	V	C

This command is how we set the flags on the 68000. We specify an immediate, the bit 1s in that immediate will set the corresponding Flags in the CCR.
For Example: To set the X flag: ORI #%00010000,CCR
For Example: To set the Z flag: ORI #%00000100,CCR

Usage Example: ORI #%00001111,CCR ORI #%00011011,CCR

Flags Affected: X N Z V C

Valid Lengths: B

ORI ##,SR

This command is only available in Supervisor Mode.
This command logically ORs immediate value **##** with the Status Register (the full 16 bit flags register).

Bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag	T	-	S	-	-	I	I	I	-	-	-	X	N	Z	V	C

Usage Example: ORI #%00001111,SR ORI #%00011011,SR

Flags Affected: X N Z V C

Valid Lengths: W

PEA <ea>,An

Push the effective address **<ea>** onto the stack. This command works the same as LEA, but does not use any registers. It could be used to push a parameter onto the stack which will be used by a subroutine.

Complex addressing modes, like (2,A4,D0) or (D1,A1), have multiple stages to calculating the final address which will be used for the parameter. We may want to use this address many times, and PEA will 'calculate' the effective address and push it onto the stack.

Usage Example: PEA (Label,PC) PEA (4,A0),A2

Flags Affected: - - - - -

Valid Lengths: L

RESET

This command is only available in Supervisor Mode - you'll probably never use it.

This command sends an "RSTO" signal, resetting external devices.

Flags Affected: - - - - -

ROL Dm,Dn
ROL #,Dn
ROL <ea>

Rotate bits in Destination **Dn** to the Left by a number of bits and Copy the top bit to the Carry.
The number of bits to shift can be specified by register **Dm** or an immediate **#**. The Carry copies the old top bit, and the bits return at Bit 0. When **<ea>** is specified, only one bit is shifted.

For Example: Let's see how the bits in a byte change with multiple rotates to the left:

Carry	76543210
0	11000010
1	10000101
1	00001011
0	00010110

Usage Example: ROL.B #8,D1 ROL.L $1000 ROL.W D1,D2

Flags Affected: - N Z V C

Valid Lengths: B W L (Defaults to W)

ROR Dm,Dn
ROR #,Dn
ROR <ea>

Rotate bits in Destination **Dn** to the Right by a number of bits and Copy the top bit to the Carry.
The number of bits to shift can be specified by register **Dm** or an immediate **#**. The Carry copies the old bit 0, and the bits return at the top bit. When **<ea>** is specified, only one bit is shifted.

For Example: Let's see how the bits in a byte change with multiple rotates to the right:

76543210	Carry
11000010	0
01100001	0
10110000	1
01011000	0

Usage Example: ROR.B #8,D1 ROR.L $1000 ROR.W D1,D2

Flags Affected: - N Z V C

Valid Lengths: B W L (Defaults to W)

ROXL Dm,Dn
ROXL #,Dn
ROXL <ea>

Rotate bits in Destination **Dn** to the Left by a number of bits, with the eXtend bit acting as an extra bit (eg an 8th bit when rotating a byte), and copy the top bit to the Carry.

The number of bits to shift can be specified by register **Dm** or an immediate **#**. The Carry and eXtend bit copies the old top bit, and the previous eXtend bit returns at Bit 0. When **<ea>** is specified, only one bit is shifted.

For Example: Let's see how the bits in a byte change with multiple rotates to the left:

Carry	eXtend	76543210
0	0	11000010
1	1	10000100
1	1	00001001
0	0	00010011

Usage Example: ROXL.B #8,D1 ROXL.L $1000 ROXL.W D1,D2

Flags Affected: - N Z V C

Valid Lengths: B W L (Defaults to W)

ROXR Dm,Dn
ROXR #,Dn
ROXR <ea>

Rotate bits in Destination **Dn** to the Right by a number of bits, with the eXtend bit acting as an extra bit (eg an 8th bit when rotating a byte), and copy the bottom bit to the Carry.

The number of bits to shift can be specified by register **Dm** or an immediate **#**. The Carry and eXtend bit copies the old top bit, and the previous eXtend bit returns at Bit 0. When **<ea>** is specified, only one bit is shifted.

For Example: Let's see how the bits in a byte change with multiple rotates to the right:

76543210	Carry	eXtend
11000010	0	0
01100001	0	0
00110000	1	1
10011000	0	0

Usage Example: ROXR.B #8,D1 ROXR.L $1000 ROXR.W D1,D2

Flags Affected: - N Z V C

Valid Lengths: B W L (Defaults to W)

RTE

Return from Exception. This command restores the full Status Register (CCR + Privileged bits).

This command is only usable in Supervisor mode, but you may need it for essential interrupt handlers.

Usage Example: RTE

Flags Affected: X N Z V C

RTR

Return and Restore condition codes. This command restores the CCR from the stack (as a 16 bit word) and then performs a normal RTS return.

There is no command to automatically put the CCR onto the stack, so if you wish to use this command you should push the CCR onto the stack.

Usage Example: RTR

Flags Affected: X N Z V C

RTS

Return from a Subroutine. The return address is taken from the top of the stack and put in the Program Counter (PC).

Usage Example: RTS

Flags Affected: - - - - -

SBCD Dm,Dn
SBCD -(Am),-(An)

This command Subtracts two 8 bit Binary Coded Decimal numbers together, with the eXtend bit acting as a carry. If we use the command "SBCD D1,D2" one Byte from D2 will be subtracted from D1, plus the eXtend bit. The formula will be D2=(D2-D1)-X.
This uses 'Packed' BCD – meaning two digits per byte.
This command only works on a single byte, so only 8 bits of the registers are used.

The Extend bit can be cleared with the command ANDI #%00001111,CCR

Usage Example: SBCD D1,D2 SBCD.B (A1)+,(A2)+

Flags Affected: X n Z v C

Valid Lengths: B

Scc <ea>

Set <ea> according to condition cc. If the condition is true, **<ea>** will be set to %11111111, if it's false <ea> will be set to %00000000.

cc	Description	Flags
SCC	Set if carry clear	C=0
SCS	Set if carry set	C=1
SEQ	Set if equal	Z=1
SGE	Set if greater than or equal	(N=1 & V=1) or (N=0 & V=0)
SGT	Set if greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)
SHI	Set if higher than	C=0 & Z=0
SLE	Set if less than or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
SLS	Set if lower than or same	C=1 or Z=1
SLT	Set if less than	(N=1 and V=0) or (N=0 and V=1)
SMI	Set if minus	N=1
SNE	Set if not equal	Z=0
SPL	Set if plus	N=0
ST	Set if True (Set Always)	=1
SF	Set if False (Set never)	=0
SVC	Set if overflow clear	V=0
SVS	Set if overflow set	V=1

Usage Example: SCC D1 SEQ.B TestLabel

Flags Affected: - - - - -

Valid Lengths: B

STOP

This command is only available in Supervisor Mode - you'll probably never use it.

Load the SR Status register with 16 bit immediate **##** and halt the processor until an interrupt or trap occurs.

Flags Affected: X N Z V C

SUB <ea>,Dn
SUB Dn,<ea>
SUBA <ea>,An
SUBI #,<ea>

This command Subtracts two numbers. For Example: The command "SUB D1,D2" will perform the calculation D2=D2-D1
Technically, the command SUB cannot work with immediate values or Address registers. The correct commands are the SUBI or SUBA commands, but if you use the 'technically incorrect' command "SUB #1,D1", or "SUB D1,A1" your assembler will use the correct SUBI or SUBA command automatically.

Usage Example: SUB D1,D2 SUB.B #1,D3 SUB.W (A1)+,D2
SUBA.L D2,A1 SUBI.B #1,(A1)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

SUBQ #,<ea>

This command Subtracts a short immediate value **#** from **<ea>**. The immediate value **#** can be a value 1-8.
This command is quicker than SUBI and is effectively the 68000's DEC command.

Usage Example: SUBQ #1,A1 SUBQ.B #3,D1 SUBQ.L #1,$1000

Flags Affected: X N Z V C

Valid Range for #: 1 to 8

Valid Lengths: B,W,L (Defaults to W)

SUBX Dm,Dn
SUBX -(Am),-(An)

This command Subtracts a data register from another data register with the eXtend bit.
For Example: The command "SUBX D1,D2" will perform the calculation D2=(D2-D1)-X.

There are two options, adding a Data register to another Data register, or (strangely) adding to the address in an address register with pre-decrement to another address with pre-decrement.

This is effectively the 68000's SBC command (Subtract with Carry).

Usage Example: SUBX D1,D2 SUBX.B D4,D1 SUBX -(A1),-(A2)

Flags Affected: X N Z V C

Valid Lengths: B,W,L (Defaults to W)

SWAP Dn

Swap the high and low words of register **Dn.**
For Example: If D1=$12345678, and we perform "SWAP D1", then D1 will become $56781234.

Usage Example: SWAP D1 SWAP D2

Flags Affected: - N Z V C

Valid Lengths: W

TAS <ea>

Test and set **<ea>**.
The Negative and Zero flag are set according to **<ea>**, and bit 7 of **<ea>** is set to 1.
If you're wondering why you need this, you don't! It's only intended for sharing memory in multiprocessor systems.

Usage Example: TAS $1000 TAS D1

Flags Affected: - N Z V C

Valid Lengths: B

TRAP

This command causes a jump to exception vector number **#**. The Program counter is pushed onto the stack, then the Status Register (SR) is pushed onto the stack.

The address called is defined by memory addresses $000080+ (vector 32-47). Each is defined by 4 bytes, so "TRAP #0" calls the address at $000080,"TRAP #1" calls the address at $000084 and so on.

What effect these have will depend on the hardware you are working with, and you will need to check the documentation for that system before using this command.

Usage Example: TRAP #1

Flags Affected: - - - - -

Valid Range for #: 0 to 15

TRAPV

If the oVerflow flag (V) is set, this command causes a jump to the overflow trap vector, the address at $00001C. The Program counter is pushed onto the stack, then the Status Register (SR) is pushed onto the stack.
The address called is defined by memory address $00001C.

Usage Example: TRAPV

Flags Affected: - - - - -

TST <ea>

Set the flags according to **<ea>**. No registers are changed.
This command is a good way to check if **<ea>** is zero or negative.

Usage Example: TST.B D1 TST.W (A1) TST.L $10000

Flags Affected: - N Z V C

Valid Lengths: B,W,L (Defaults to W)

UNLK An

Reverse the process performed by the LINK command. The LINK command allocates memory on the stack (known as a 'Stack Frame'). This sets the Stack Pointer (SP) to **An**, and then pops the original value of **An** off the stack.

The original value of the Stack Pointer and **An** have now been restored.

Usage Example: UNLK A1 UNLK A2

Flags Affected: - - - - -

Chapter 5: The 8086

Introducing the 8086

The 8086 was the 16 bit 'successor' to Intel's 8080. Heavily enhancing the addressing modes and adding more registers and many more commands, it became the standard for desktop computers and, in the form of the x68, It's legacy lives on today!

While the 80386, 80486 and beyond added huge extra functionality, in this book we'll only be looking at the classic 8086, which will allow us to write little DOS programs and even the 80186 based "Bandai WonderSwan" handheld console.

If you later decide to 'move up' to the more modern processors, what you've learned here will be a good 'Foundation', while being a simpler introduction than the full range of more modern CPUs.

Like the 8080 and Z80, the 8086 is a Little Endian processor. It has a wide range of registers, some of which are fairly general purpose, and others are more specialized.

The large number of instructions can be a little daunting, and the 'Segment' memory model can be a challenge, but programming the 8086 is generally relatively easy, thanks to its large command set, and many 16 bit registers.

8086 Addressing

The 8086 uses a 20 bit address bus, but all its registers are 16 bit. When the CPU addresses part of the memory, the resulting address is actually made up of a combination of 2 registers. The Segment register makes up the top 16 bits of the 20 bit address, the register used as a pointer within that segment is ADDED to that address, making up the bottom 16 bits.

	Bit:	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	DX	-	-	-	-	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
+	ES*16	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	-	-	-	-
=	Final Addr	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

*Figure 57: When we specify an address in the form "[ES:DX]", the final address will be calculated with the formula: ADDR=(ES*16)+DX.*

When we use a register to specify an address, each register will have a 'default' segment, but we can override this by specifying a segment register, followed by a colon, and the register we want to use as an address within that segment. For Example: "MOV AL, [ES:DX]".

The 8086 Registers

The main 4 registers on the 8086 are 16 bit, and, like the Z80, these registers are split into two 8 bit parts which can be used as 8 bit registers, or combined and used as a 16 bit register.

On the 8086, the 16 bit registers end in an 'X' – AX, BX, CX and DX. Each is made up of two parts, a High part, and a Low part. AX is made up of AH and AL, the same is true of the other main registers (BX, CX and DX).

Registers like SP, BP, SI and DI are also 16 bit, but do not have a separate 8 bit H and L part.

Register	8 Bit High	8 Bit Low	Default Segment	Use cases
AX 16 bit Register	AH	AL	DS	Accumulator
BX 16 bit Register	BH	BL	DS	Base
CX 16 bit Register	CH	CL	DS	Count
DX 16 bit Register	DH	DL	DS	Data
Stack Pointer	SP		SS	Stack
Base Pointer	BP		SS	Used for pointing to the start of a memory range
Destination Index	DI		DS	Used by String commands
Source Index	SI		DS	Used by String commands
Flags	F			Used for Conditions
Program Counter	IP		CS	Current running code

The 8086 Segment Registers

There are 4 segment registers on the 8086, which are used to calculate the full 20 bit address. Note: It's not possible to set these directly, the desired value needs to be loaded into a register like AX, then moved into the segment register.

Register	Purpose	Valid Offset Registers
CS	Command Segment (Program code)	IP
DS	Data Segment (Data)	SI, DI, BX
ES	Extra Segment (More Data)	SI, DI, BX
SS	Stack Segment (Stack)	SP, BP

When we use a register as a source address, each register will have a default segment register.

For Example: BX uses the DS segment. "MOV AX,[BX]" will load from address DS:BX.

We can override this default segment by specifying a segment. For Example: "MOV AX, [ES:BX]" will load from the Extra Segment (ES), offset BX.

There are many times we'll need to load segment registers and offsets, and we'll want the assembler to calculate these for us.

We can get the segment registers based on the ".CODE" or ".DATA" block by specifying these with an @ symbol.

We can calculate the Segment and offset with the "SEG" command and the "OFFSET" command:

```
20      .code
21
22  ;Set registers with @
23      mov ax, @data           ;DS points to our Data segment
24      mov ds, ax
25
26      mov ax, @code           ;ES points to our Code segment
27      mov es, ax
28
29  ;Set registers from label
30      mov ax, seg MyData
31      mov es, ax              ;Set ES to segment MyData
32
33      mov bx, offset MyData   ;Set BX to offset MyData
34
35      mov ax,[es:bx]          ;Get word from Mydata
36
37      Call DoMonitor          ;Show Registers
```

Figure 58: Segments can be calculated with SEG, but can't be loaded directly into a segment register.

The 8086 Flags

While the Status register is 16 bits, we can't access all of it in "User mode" (the mode our programs tend to run). The top 8 bits are protected and can only be accessed in Supervisor mode, but we can access the first 8 bits anytime.

The User mode flags are called the CCR, the Condition Code Register.
The Supervisor mode flags are called the SR, the Service Register.

Bit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag	-	-	-	-	O	D	I	T	Z	S	-	A	-	P	-	C

Flag	Name	Description
T	Trap	1=Causes interrupt INT2 every instruction
D	Direction	Used for 'string' functions
I	Interrupt enable	Allow maskable hardware interrupts
O	Overflow	1=Overflow (sign changed)
S	Sign	1=Negative 0=positive
Z	Zero	1=Zero
A	Aux carry	Used as Carry in BCD
P	Parity	1=Even no of 1 bits (8 bit)
C	Carry	1=Carry/Borrow caused by last ADD/SUB/ROT instruction

In the instruction references of this book, the 'Flags Affected' section will show a minus '-' when an instruction leaves a flag unchanged, an uppercase letter when a flag is correctly updated (for example 'C'), and a lowercase letter when a flag is changed to an undetermined state (for example 'c').

8086 Data Types in Source Code

Here are the typical ways 8086 assemblers represent the different data types:

Prefix	Suffix	Example	Meaning
		12	Immediate Decimal Value.
	b	10101010b	Immediate Binary Value
0x	**h**	0FFh 0xFF	Immediate Hexadecimal Value
'	**'**	#'A'	Immediate ASCII Value
[	**]**	[1000]	Memory Address in Decimal
[	**h]**	[1000h]	Memory Address in Hexadecimal

8086 Program Models

Because of the segmented memory layout and wide range of 8086 configurations, with memory sizes from Kilobytes or many Megabytes, 8086 programs can be designed for certain 'Models'.

A "COM File" can be only 64K in total. This is the ".TINY" model. They store code and data in a single segment.

That's a little too limited, so we'll be using the '.SMALL' model, and creating the more flexible "EXE File" file should be enough. It will let you try out multiple segments for data and code, but is still pretty simple.

Defining bytes of data on the 8086 in UASM

There will be many times when we need to define bytes of data in our code. Examples of this would be bitmap data, the score of our player, a string of text, or the co-ordinates of an object.
The commands will vary depending on the CPU and your assembler, but the ones shown below are the ones used by the assembler covered in these tutorials.

For Example: If we want to define a 32 bit sequence $12345678 we would use "DD 12345678h".

Bytes	Z80	6502	68000	8086	ARM
1	DB	DB	DC.B	DB	.BYTE
2	DW	DW	DC.W	DW	.WORD
4			DC.L	DD	.LONG
n	DS n,x	DS n,x	DS n,x	n DUP (x)	.SPACE n,x

Protected Mode and Real Mode

The 8086 had limited addressing capabilities which made accessing large amounts of memory difficult. With the 80286 a new mode called 'Protected Mode' was added, to help ease such problems.

Protected mode adds support for virtual memory support, multithreading and more. Processors default to the Real mode. Protected mode must be turned on to use this functionality. DOS extenders, like DOS/4GW, used protected mode in DOS, and Windows 3.0+ used protected mode.

Real mode is the only mode on the 8086 and is the only thing covered in this guide.

8086 Addressing Modes

Immediate Addressing

A fixed number is the parameter for an operation. The number starts with a # to mark it as an immediate (otherwise it will be mistaken for an address). A suffix of "h" can be used to specify hexadecimal, though the number must start with a digit (for example 0FFh), "b" can be used to specify binary.

Usage Example: MOV AX,100 MOV AL,0b00001111 MOV BH,0F1H

Register Addressing

An 8 or 16 bit register is the source or destination of the operation. Both parameters must be the same size.

Usage Example: MOV AX,BX MOV AL,BL

Direct Memory Addressing

A fixed location in memory is used as the source or destination, specified by a label or numbered address.

Depending on the command, it may not be clear whether the command is working on a byte or a word. In this case we can specify this with "BYTE PTR" or "WORD PTR".

Usage Example: MOV AX,[1000h] INC BYTE PTR [MyData]

Register indirect Addressing

A location in memory, specified by the value in a register, is the source or destination.

Depending on the command, it may not be clear whether the command is working on a byte or a word. In this case we can specify this with "BYTE PTR" or "WORD PTR".

Usage Example: MOV AX,[BX] INC BYTE PTR [BP]

Valid Registers: [BX], [BP], [DI], [SI]

Based or Indexed Addressing

A location in memory, specified by the value in a register plus an immediate numeric displacement (offset), is the source or destination.

There are various formats which may be used, "[1+BX]",[BX]+1 and "1[BX]" have the same meaning.

Depending on the command, it may not be clear whether the command is working on a byte or a word. In this case we can specify this with "BYTE PTR" or "WORD PTR".

Usage Example: MOV AX,[1+BX] MOV CX,2[BX] MOV DX,[BX]+3

Valid Registers: [BX], [BP], [DI], [SI]

Based, Indexed Addressing

A fixed location in memory is used as the source or destination plus a register displacement offset.

There are various formats which may be used, "[BX][DI]" and "[BX+DI]" have the same meaning.

Depending on the command, it may not be clear whether the command is working on a byte or a word. In this case we can specify this with "BYTE PTR" or "WORD PTR".

Usage Example: MOV AX, [BX+DI] MOV CH, [BX+SI] MOV CL, [BP+DI]
MOV DX, [BP+SI]

Valid Registers: [BX+DI], [BX+SI], [BP+DI], [BP+SI]

Based, Indexed Addressing with displacement

A location in memory, specified by the value in a register plus a register displacement plus an immediate numeric displacement (offset), is the source or destination.

There are various formats which may be used.
For Example: "1[BX][DI]", "[1+BX+DI]", "[BX+DI]+1" and "[BX][DI+1]" (among others) all have the same meaning.

Usage Example: MOV AX, 1[BX+DI] MOV CH, [BX+SI+2] MOV CL, [3+BP+DI]
MOV DX, [BP+SI]+4

Valid Registers: [BX+DI], [BX+SI], [BP+DI], [BP+SI]

String Addressing

This addressing mode is used by string commands which are designed to perform on Sequences of bytes (Strings).

These commands will use DS:SI as the source address, and ES:DI as the destination.

Usage Example: MOVSB MOVSW CMPSB LODSW SCASB

Compiling 8086 with UASM, and running with the DosBox emulator

Writing an assembler program

An ASM file is just a text file! You can edit it with the text editor of your choice. I use Notepad++, but you can use Windows Notepad, Visual Studio Code, or anything else you like.

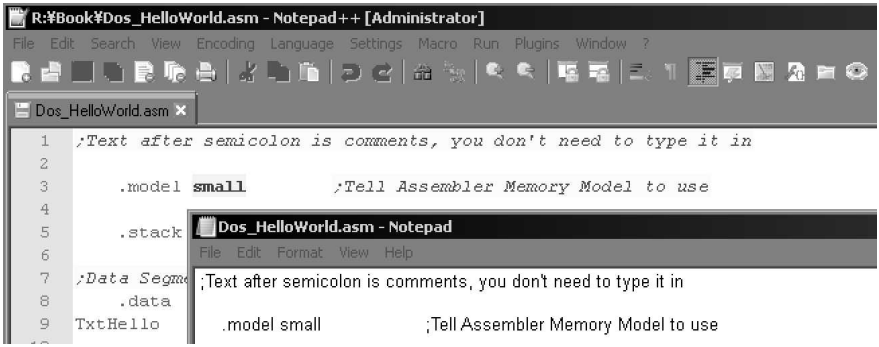


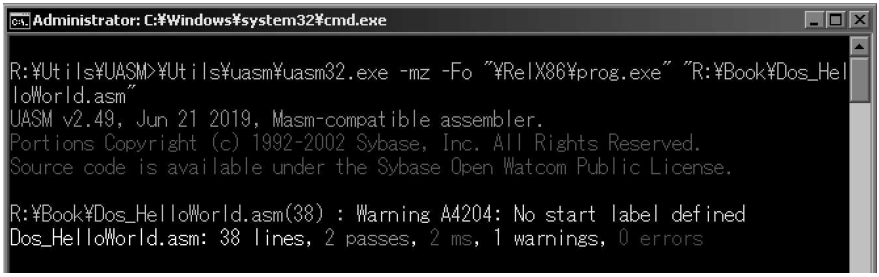
Figure 59: ASM files are just text files. Even classic Notepad can do the job, but an editor like Notepad++ offers syntax highlighting.

Using UASM

UASM is a free MASM (Microsoft Assembler) compatible assembler which will allow you to compile ASM files into runnable programs on DOS.

UASM is free, and can be found at:
http://terraspace.co.uk/uasm.html

UASM is a Command line tool, we run it from a CMD prompt (Run CMD.exe in Windows).



Here is a minimal example of a script to compile an ASM file into a DOS executable EXE file:

uasm32.exe -mz -Fl"\BldX86\listing.txt" -Fo "\RelX86\prog.exe" "R:\Book\Dos_HelloWorld.asm"

This will compile the source file "R:\Book\Dos_HelloWorld.asm" into an executable file our emulator can run. The built file will be saved to "\RelX86\prog.exe".

Here is a more detailed example with some extra features:

\Utils\uasm\uasm32.exe -mz -Fl"\BldX86\listing.txt" -Fo "\RelX86\prog.exe" "R:\Book\Dos_HelloWorld.asm"

This will also compile the program, however it will also create a listing file called "Listing.txt". This is to help with debugging, it shows the source code and how the code compiles to bytes.

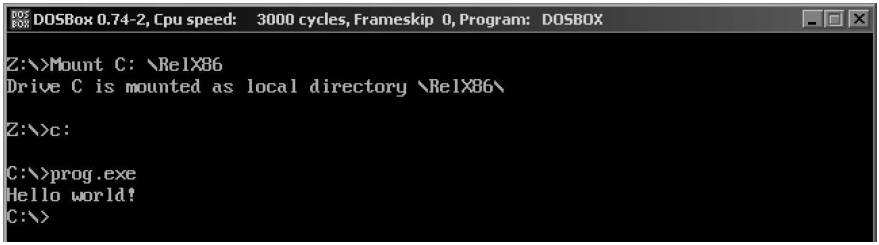
Either can be used to create the examples below, the second just has some helpful debugging options.

Running with Dosbox

We're going to use DosBox to run our compiled program. To start the program we need to ensure we have a mounted drive, and run our program from the command line.

The program will run and show the result.

We can automate this by adding the commands to start our program to the [autoexec] section of the "dosbox.conf" file.



8086 Examples

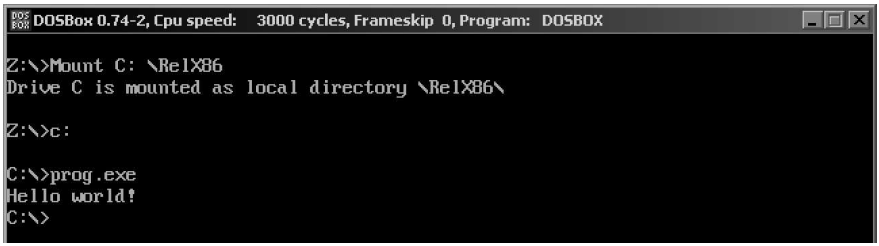
Example 1: Hello World

This Example will show a 'Hello World' Message on DosBox.

Compile this with UASM in the way described on the previous pages.

```
;Text after semicolon is comments, you don't need to type it in

    .model small        ;Tell Assembler Memory Model to use

    .stack 1024         ;Stack pointer area (1024 bytes)

;Data Segment (DS)
    .data
TxtHello        db "Hello world!",255

;Code Segment (CS)
    .code

    mov ax, @data       ;DS points to our Data segment
    mov ds, ax

    mov si,offset TxtHello ;Address of Hello World string
    call PrintString    ;Show DS:SI to screen

    mov ah, 4ch         ;Terminate a process (EXIT)
    mov al, 0           ;Return Code
    int 21h             ;Dos Int

PrintString:    ;Print 255 terminated string from [ds:si]
    mov al,[ds:si]      ;Load a letter
    cmp al,255          ;CHR=255 ?
    jz PrintString_Done ;Yes? then RET
    call PrintChar      ;Print to screen
    inc si              ;Next Char
    jmp PrintString     ;Repeat
PrintString_Done:
    ret

PrintChar:              ;Print AL to screen
    mov ah, 02h         ;Output character to monitor (DL)
    mov dl, al          ;DL = Character to print
    int 21h             ;Dos Int
    ret

```

The program will run in the emulator and show a 'Hello World' message:



How does it work?

We started our program defining that we'll be building using the "Small" program model. (Line 3)

We then define some stack space. (Line 5)

We define our "Hello World" String in the data segment. (Lines 8-9)

In the Code segment we set up our DS register to point to the data segment, and the SI register to point to the TxtHello string offset within that segment. [DS:SI] now points to the string we want to show. (Lines 14-17)

We define a "PrintString" routine to print each character of our string. This function reads bytes one by one from [DS:SI], and checks if they are 255, the end of the string. For any other value we use "PrintChar" to show the character. (Lines 24-32)

The "PrintChar" routine uses DOS interrupt 21h, function 02h. This function prints the character in register DL to the screen. (Lines 34-38)

We then use DOS interrupt 21h, function 4Ch, which exits our program and returns to DOS. (Lines 20-22)

Example 2: Test Bitmap

This Example will show a Smiley using 256 color VGA mode.

Compile this with UASM in the way described on the previous pages.

```
;Text after semicolon is comments, you don't need to type it in


    .model small    ;Tell Assembler Memory Model to use

    .stack 1024         ;Stack pointer area

;Code Segment (CS)
    .code
    mov ah, 0           ;0=Set Video mode (AL=Mode)
    mov al, 13h         ;mode 13 (VGA 320x200 256 color)
    int 10h             ;bios int

    mov di,0            ;Screen pos (0,0)
    mov ax,0A000h       ;Screen base
    mov es,ax

    mov ax, @code       ;Point DS to this segment
    mov ds, ax
    mov si,offset BitmapTest ;DS:SI=Source bitmap

    mov cl,8            ;Height
DrawBitmap_Yagain:
    push di
        mov ch,8        ;Width
DrawBitmap_Xagain:
        movsb           ;Transfer a byte from DS:SI to ES:DI
        dec ch
        jnz DrawBitmap_Xagain ; Next horizontal pixel
    pop di
    add di,320          ;Move down 1 line (320 pixels)
    inc bl
    dec cl
    jnz DrawBitmap_Yagain

infloop:
    jmp infloop

```

(Continued on the next page)

```
39  BitmapTest:     ;Smiley, 1 byte per pixel
40      DB 00h,00h,01h,01h,01h,01h,00h,00h      ; 0
41      DB 00h,01h,01h,01h,01h,01h,01h,00h      ; 1
42      DB 01h,01h,03h,01h,01h,03h,01h,01h      ; 2
43      DB 01h,01h,01h,01h,01h,01h,01h,01h      ; 3
44      DB 01h,01h,01h,01h,01h,01h,01h,01h      ; 4
45      DB 01h,01h,02h,01h,01h,02h,01h,01h      ; 5
46      DB 00h,01h,01h,02h,02h,01h,01h,00h      ; 6
47      DB 00h,00h,01h,01h,01h,01h,00h,00h      ; 7
48
```

The program will switch to VGA mode, and show a Smiley in the corner of the screen.

How does it work?

We're going to use VGA mode to show a bitmap, this mode uses 1 byte per pixel.

To enable the mode we use BIOS interrupt 10h, sub function 0h. We select screen mode 13h - VGA 320x200, 256 color mode. (Lines 10-12)

VGA VRAM starts at Segment A000:0000h. A byte written to this area will appear as a pixel on the screen. We set [ES:DI] to this address. (Lines 14-16)

We load our sprite address to DS:SI. (Lines 18-20)

We can now use MOVSB to copy a byte from the bitmap to the screen, this command automatically increases SI and DI. (Line 27)

As each screen line is 320 pixels, we move down a line of the screen by adding 320 to the VRAM address. (Line 31)

We repeat this procedure until the bitmap is shown to the screen. (Lines 23-34)

Example 3: Moving a sprite

This Example will show a Smiley using 256 color VGA mode and allow us to move it with the cursor keys.

Compile this with UASM in the way described on the previous pages.

```
;Text after semicolon is comments, you don't need to type it in

    .model small        ;Tell Assembler Memory Model to use
    .stack 1024         ;Stack pointer area
    .code               ;Code Segment (CS)

    mov ah, 0           ;0=Set Video mode (AL=Mode)
    mov al, 13h         ;mode 13 (VGA 320x200 256 color)
    int 10h             ;bios int

    mov dh,1            ;Xpos
    mov dl,1            ;Ypos

    call ShowSprite     ;Show the starting position

infloop:
    mov ah, 01h         ;Func 01h = Check if the Keybuffer is empty
    int 16h             ;Int 16=Keyboard interrupt
    jz infloop          ;Check if no key pressed

    call ShowSprite     ;Remove old sprite

    mov ah, 00h         ;Func 0Eh = Read a key from buffer(keycode in AH)
    int 16h             ;Int 16=Keyboard interrupt

    cmp ah,48h          ;Compare to UP
    jnz CurNotUp
    cmp dl,0            ;Check it at the top of the screen
    jz CurNotUp
    dec dl              ;Move Up
CurNotUp:

    cmp ah,50h          ;Compare to Down
    jnz CurNotDown
    cmp dl,24           ;Check it at the bottom of the screen
    jz CurNotDown
    inc dl              ;Move Down
CurNotDown:

    cmp ah,4Bh          ;Compare to Left
    jnz CurNotLeft
    cmp dh,0            ;Check it at the left of the screen
    jz CurNotLeft
    dec dh              ;Move Left
CurNotLeft:

```

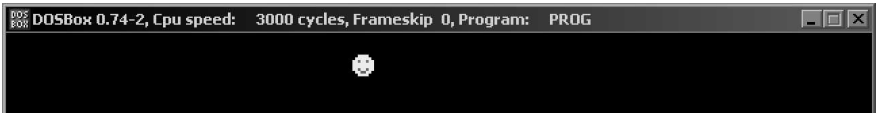
(Continued on the next page)

```
47      cmp ah,4Dh          ;Compare to Right
48      jnz CurNotRight
49      cmp dh,39           ;Check it at the right of the screen
50      jz CurNotRight
51      inc dh              ;Move Right
52  CurNotRight:
53
54      call ShowSprite     ;Show the new sprite position
55      jmp infloop         ;Repeat
56
57  ;XOR sprite at (X,Y) pos (DH,DL)
58  ShowSprite:
59  ;Calculate Screen pos
60      mov ax,0A000h       ;Screen base
61      mov es,ax
62      mov ax, @code       ;Point DS to this segment
63      mov ds, ax
64
65      push dx
66          mov ax,8        ;8 bytes per 8x8 block
67          mul dh
68          mov di,ax
69
70          mov ax,8*320    ;320 bytes per line, 8 lines per block
71          mov bx,0
72          add bl,dl
73          mul bx
74          add di,ax
75      pop dx              ;ES:DI is VRAM Destination
76
77  ;Draw XOR sprite
78      mov si,offset BitmapTest ;DS:SI=Source bitmap
79      mov cl,8            ;Height
80  DrawBitmap_Yagain:
81      push di
82          mov ch,8        ;Width
83  DrawBitmap_Xagain:
84          mov al,[DS:SI]
85          xor al,[ES:DI]  ;Xor with current screen data.
86          mov [ES:DI],al
87          inc si
88          inc di
89          dec ch
90          jnz DrawBitmap_Xagain ; Next horizontal pixel
91      pop di
92      add di,320          ;Move down 1 line (320 pixels)
93      dec cl
94      jnz DrawBitmap_Yagain
95      ret
96
```

(Continued on the next page)

```
 97  BitmapTest:             ;Smiley, 1 byte per pixel
 98      DB 00h,00h,0Eh,0Eh,0Eh,0Eh,00h,00h      ;  0
 99      DB 00h,0Eh,0Eh,0Eh,0Eh,0Eh,0Eh,00h      ;  1
100      DB 0Eh,0Eh,03h,0Eh,0Eh,03h,0Eh,0Eh      ;  2
101      DB 0Eh,0Eh,0Eh,0Eh,0Eh,0Eh,0Eh,0Eh      ;  3
102      DB 0Eh,0Eh,0Eh,0Eh,0Eh,0Eh,0Eh,0Eh      ;  4
103      DB 0Eh,0Eh,02h,0Eh,0Eh,02h,0Eh,0Eh      ;  5
104      DB 00h,0Eh,0Eh,02h,02h,0Eh,0Eh,00h      ;  6
105      DB 00h,00h,0Eh,0Eh,0Eh,0Eh,00h,00h      ;  7
```

We can move the Smiley with the cursor keys.



How does it work?

In this example we've altered our bitmap routine. It now XORs with the existing bytes on screen. This means if we draw the sprite in the same position twice the old one will be removed. (Lines 77-96)

We calculate the screen memory from an (X,Y) position in (DH,DL). Each byte is one pixel, and each line is 320 bytes. We use the MUL command to multiply the X,Y position and calculate the offset in VRAM. (Lines 60-75)

We use firmware functions of function Interrupt 16h (the keyboard interrupt). This takes a value in AH, which defines the subfunction.

Interrupt 16h, subfunction 01h, will return the Z flag if no keys have been pressed. We use this to wait until a key is in the buffer. (Lines 17-19)

Interrupt 16h, subfunction 00h, will read in a keycode from the buffer into AH, removing it from the buffer. (Lines 23-24)

We compare AH to each of the 4 direction keycodes, and check the related boundary to see if the sprite is already at the edge of the screen. If it's not we can move in this direction. (Lines 26-52)

We draw the new sprite location and repeat. (Lines 54-55)

Learning 8086: Where to go from here

The documentation examples we've covered here should be enough to get you started with your first 8086 program.

Over the following pages of this chapter, you'll be presented with all the 8086 instructions, and a description of their purpose, with examples.

If you're looking for more, there's lots more content on the ChibiAliens website!

If you're looking for more information on the 8086 instruction set, there are step by step tutorials, with videos and source code on the 8086 page:
https://www.chibialiens.com/8086/

We've only covered DOS here. For details of the other systems covered, and detailed tutorials on the hardware of all the systems, take a look at the full list here:
https://www.assemblytutorial.com/

Don't forget to download the source code for this book from:
www.chibiakumas.com/book

And please subscribe to the official YouTube channel for weekly new videos:
https://www.youtube.com/chibiakumas

8086 Instructions

AAA

ASCII Adjust for Addition. This treats AL as an unpacked binary coded decimal number, which contains the value 0-9.

We can use the normal "ADD" commands (which normally work in hexadecimal) then use AAA, which will 'adjust' the accumulator.

If AL goes over 9, it will be corrected, and the Carry flag will be set and AH will also be incremented, but this will not be correctly decimal adjusted.

For Example: If AX=0908h, and we repeatedly perform "ADD AL,01h AAA", the AX and carry will change in the following ways:

Iteration	AX	Carry
0	0908h	0
1	0909h	0
2	0A00h	1
3	0A01h	0
4	0A02h	0

Usage Example: AAA

Flags Affected: o s z A p C

AAD

ASCII Adjust for Division. This command takes two unpacked binary coded decimal digits (one in AH and one in AL) and combines them into a single value in AL, this allows DIV to be used as normal. The formula is effectively AL=AL+(AH*10), AH=0.

For Example: Let's suppose AX=0204h (24 in unpacked binary coded decimal) and we want to divide this by 2. Of course, the result will be 12. Here is how we use the AAD and DIV commands:

Command	AX after command	BX
	0204h (24 in unpacked BCD)	02h
AAD	0018h (24 in decimal)	02h
DIV BL	000Ch (12 in decimal)	02h

AX now contains the decimal 12 result of the division.

Usage Example: AAD

Flags Affected: o S Z a P c

AAM

ASCII Adjust for Multiplication. This treats AL as an unpacked binary coded decimal number, which contains the value 0-9.

We can use the normal "MUL" command (which normally works in hexadecimal) then use AAM, which will 'adjust' the accumulator.

If AL goes over 9 then AL will be corrected and AH contain the top digit of the result. The Carry is not used.

For Example: If AX=0004h, BL=02h and we repeatedly perform "MUL BL AAM".

Iteration	AX	
0	0004h	
1	0008h	4*2=8
2	0106h	8*2=16
3	0102h	6*2=12
4	0004h	2*2=4

Usage Example: AAM

Flags Affected: o S Z a P c

AAS

ASCII Adjust for Subtraction. This treats AL as an unpacked binary coded decimal number, which contains the value 0-9.

We can use the normal "SUB" commands (which normally work in hexadecimal) then use AAS, which will 'adjust' the accumulator.

If AL goes under 0 then AL will be corrected and the Carry flag will be set. AH will also be decremented, but this will not be correctly decimal adjusted.

For Example: If AX=0002h, and we repeatedly perform "SUB AL,01h AAS":

Iteration	AX	Carry
0	0002h	0
1	0001h	0
2	0000h	0
3	FF09h	1
4	FF08h	0

Usage Example: AAS

Flags Affected: o s z A p C

ADC dest,src

Add **src** and the carry flag to **dest**.

Usage Example: ADC AX,BX ADC AL,BL ADC CX,1000h
ADC [1000h],DX

Flags Affected: O S Z A P C

ADD dest,src

Add **src** to **dest**.

Usage Example: ADD AX,BX ADD AL,BL ADD CX,1000h
ADD [1000h],DX

Flags Affected: O S Z A P C

AND dest,src

Logical AND of bits in **dest** with Accumulator **scr**.
Where a bit in **dest** and **src** are 1 the result will be 1, when they are not it will be 0.
For Example: If AL=10101010b, if we use "AND AL,11110000b", then this will result in AL=10100000b.

Usage Example: AND BX,64h AND AX,1100h

Flags Affected: O S Z A P C

CALL dest

Call Subroutine at address **dest** - this pushes the return address onto the stack, and changes the Program Counter to **dest**. This is effectively the same as the GOSUB command in Basic.

Usage Example: CALL 1000h CALL TestLabel CALL AX

Flags Affected: - - - - - -

CBW

Convert the 8 bit byte in AL into a 16 bit word in AX. This 'Sign Extends' AL keeping the same sign in the now 16 bit number. Effectively the bits of AH are set to whatever bit 7 of AL is.

Usage Example: CBW

Flags Affected: - - - - - -

CLC

Clear the Carry Flag. C flag will be set to Zero.

Usage Example: CLC

Flags Affected: - - - - - C

CLD

Clear the Direction Flag. D flag will be set to Zero. This is used for 'String functions'.

The name 'String functions' is not necessarily to do with text. On the 8086 'strings' are for batch data copying from one memory address to another. Clearing the Direction flag means the source and destination addresses go up after each transfer.

Usage Example: CLD

Flags Affected: D - - - - - -

CLI

Clear the Interrupt enable flag. I flag will be set to 0. This disables maskable interrupts.

Usage Example: CLI

Flags Affected: I - - - - - -

CMC

Complement the Carry flag. If C=1 it will now be 0. If it was 0 it will now be 1.
You may have mistaken this for clear carry flag but it's not. If you want to clear the carry, but change nothing else, use "OR A".

Usage Example: CMC

Flags Affected: - - - - - C

CMP dest,src

Compare the Byte or Word **dest** to **src**.
This sets the flags the same as "SUB dest,src" would, but the destination is unchanged.
Usually this command will be followed by a conditional branch like "JZ MatchedLabel".

Usage Example: CMP AL,32 CMP AX,BX

Flags Affected: O S Z A P C

CMPSB
CMPSW

Compare DS:SI to ES:DI. This command can work in bytes or words. The flags are set like a CMP command.

This command can be combined with repeat command REPZ or REPNZ.
REPZ will cause the repeat to continue until the strings no longer match or CX=0
REPNZ will cause the repeat to continue until the strings start to match or CX=0.

After the command DI and SI are increased by the compare size (1 or 2 bytes) if the direction flag is zero, or decreased if it is 1.

Usage Example: CMPSB REPZ CMPSB REPNZ CMPSW

Flags Affected: O S Z A P C

CWD

Convert the 16 bit word in AX into a 32 bit doubleword in DX.AX. This 'Sign Extends' AX, keeping the same sign in the now 32 bit number. Effectively the bits of DX are set to whatever bit 15 of AX is.

Usage Example: CWD

Flags Affected: - - - - - -

DAA

Decimal Adjust for Addition. This treats AL as a packed binary coded decimal number, which contains the value 0-99.

We can use the normal "ADD" commands (which normally work in hexadecimal) then use DAA, which will 'adjust' the accumulator.

If either nibble in AL goes over 9, it will be corrected, and the Carry flag will be set.
For Example: If AX=0088h, and we repeatedly perform "ADD AL,05h DAA":

Iteration	AX	Carry
0	0088h	0
1	0093h	0
2	0098h	0
3	0003h	1
4	0008h	0

Usage Example: DAA

Flags Affected: O S Z A P C

DAS

Decimal Adjust for Subtraction. This treats AL as a packed binary coded decimal number, which contains the value 0-99.

We can use the normal "SUB" commands (which normally work in hexadecimal) then use DAA, which will 'adjust' the accumulator.
If either nibble in AL goes under 0, it will be corrected, and the Carry flag will be set.

For example: If AX=0012h, and we repeatedly perform "SUB AL,05h DAA":

Iteration	AX	Carry
0	0012h	0
1	0007h	0
2	0002h	0
3	0097h	1
4	0092h	0

Usage Example: DAS

Flags Affected: O S Z A P C

DEC Dest

Decrease **Dest** by one. This is faster than using SUB with a value of 1. However, unlike SUB, the carry flag is not affected.

Usage Example: DEC AL DEC AX DEC WORD PTR [1000h]

Flags Affected: O S Z A P -

DIV src

Divide Unsigned number AX or DX.AX by **src**.

If **src** is 8 bit (BL,CH etc) then the formula used is AL=AX/**src.**
The Quotient is stored in AL (the integer result of the division). The remainder is in AH.

If **src** is 16 bit (BX,CX etc) then the formula used is AX=DX.AX/**src** (where DX.AX is a 32 bit pair).
The Quotient is stored in AX (the integer result of the division). The remainder is in DX.

We need to check our division is possible before using this command, or the CPU will lock up.
If division by zero occurs, Interrupt 0 will occur.
If the result is too large to fit in the register, Interrupt 4 will occur. (If we use "DIV DL" when AX=1000h and DL=1 the result 1000h will not fit in AL).

Usage Example: DIV BL DIV CH DIV CX

Flags Affected: o s z a p c

ESC #,src

This command is for working with multiple processors - it's not something you will need.

Usage Example: ESC 1,AH

Flags Affected: - - - - - -

Valid Range for #: 0 to 7

HLT

Stop the CPU until an interrupt occurs. After the interrupt occurs, execution will resume after the HALT command. If interrupts were disabled by CLI, then HLT will permanently stop the processor.

Usage Example: HLT

Flags Affected: - - - - - -

IDIV src

Divide Signed number AX or DX.AX by **src**

If **src** is 8 bit (BL,CH etc) then the formula used is AL=AX / **src.**
The Quotient is stored in AL (the integer result of the division). The remainder is in AH.

If **src** is 16 bit (BX,CX etc) then the formula used is AX=DX.AX / **src** (where DX.AX is a 32 bit pair)
The Quotient is stored in AX (the integer result of the division). The remainder is in DX.

We need to check our division is possible before using this command, or the CPU will lock up.
If division by zero occurs, Interrupt 0 will occur.

If the result is too large to fit in the register, Interrupt 4 will occur. (If we use "IDIV DL" when AX=1000h and DL=1 the result 1000h will not fit in AL).

Usage Example: IDIV BL IDIV CH IDIV CX

Flags Affected: o s z a p c

IMUL src

Multiply Signed number AX or DX.AX by **src**

If **src** is 8 bit (BL, CH etc) then the destination will be 16 bit and the formula used is AX=AL***src**.

If **src** is 16 bit (BX, CX etc) then the destination will be 32 bit. In this case the formula used will be DX.AX=AX***src** (where DX.AX is a 32 bit pair).

Usage Example: IMUL BL IMUL CH IMUL CX

Flags Affected: O s z a p C

IN dest,port

Read in an 8 bit byte or 16 bit word into **dest** (either AX, AL or AH) from hardware port number **port**.

Port can either be an 8 bit immediate (0-255) or DX. The only way to access a 16 bit port (0-65535) is via DX.

Usage Example: IN AL,100 IN AX,F0h IN AL,DX IN AX,DX

Flags Affected: - - - - - -

INC Dest

Increase **Dest** by one. This is faster than using ADD with a value of 1. However, unlike ADD, the carry flag is not affected.

Usage Example: INC AL INC AX INC WORD PTR [1000h]

Flags Affected: O S Z A P -

INT

This command causes software interrupt **#**. The flags are pushed onto the stack, as well as the program counter, then interrupt handler **#** is called.
The main use for these is as 'System Calls' to the bios or operating system.

For Example: "INT 21h" is the "Dos Interrupt" which can perform a wide variety of tasks, like printing characters or reading files. Another, "INT 33h", can be used to control the mouse.

Using these software interrupts requires reading the technical documentation, and ensuring you have the other registers correctly set up with the parameters the interrupt will expect.

Usage Example: INT 21h INT 33h

Flags Affected: - - - - -

Valid Range for #: 0 to 255

INTO

INTO will cause Interrupt 4 if the Overflow flag (O) is set, otherwise it will have no effect.

Usage Example: INTO

Flags Affected: - - - - -

IRET

Restore the flags from the stack and return from an Interrupt.
This is the return you will use if you're writing your own interrupt handler. You won't need it for normal programming.

Usage Example: IRET

Flags Affected: O S Z A P C

Jcc addr

Jump to 8 bit offset **addr** (which will in practice be a label) if condition **cc** is true.
If the condition is false, execution will continue with the next command.

We will typically specify **addr** as a label, the Assembler will work out the correct offset for us.

There are many possible conditions for comparisons with CMP. The ones you will want to use will depend if your values are signed or unsigned numbers.

Note: Many conditional jumps that have the same compiled bytes and flag conditions have different opcodes for clarity.
For Example: JC and JAE are functionally the same command, but you can use either in your code to make reading the code clearer.

Command	Details	Flags
JA / JNBE	Above / Not Below or Equal (For Unsigned Numbers)	C=0 AND Z=0
JBE / JNA	Below or Equal / Not Above (For Unsigned Numbers)	C=1 OR Z=1
JC JB / JNAE	Carry Below / Not Above or Equal (For Unsigned Numbers)	C=1
JE / JZ	Equal / Zero	Z=1
JG / JNLE	Greater / Not Less than or Equal (For Signed Numbers)	((S XOR O) OR Z)=0
JGE / JNL	Greater or Equal / Not Less (For Signed Numbers)	(S XOR O)=0
JLE / JNG	Less than or Equal / Not Greater (For Signed Numbers)	((S XOR O) OR Z)=1
JL / JNGE	Less / Not Greater or Equal (For Signed Numbers)	(S XOR O)=1
JNC JAE / JNB	No Carry Above or Equal / Not Below (For Unsigned Numbers)	C=0
JNE / JNZ	Not Equal / Not Zero	Z=0
JNO	Not Overflow	O=0
JNP / JPO	Not Parity / Parity Odd	P=0
JNS	Not Signed (not negative)	S=0
JO	Overflow	O=1
JP / JPE	Parity / Parity Equal (bits 0-7 only)	P=1
JS	Signed (is positive)	S=1

Usage Example: JZ TestLabel JO ErrorHandle

Flags Affected: O S Z A P C

JCXZ addr

Jump to 8 bit offset **addr** (which will in practice be a label) if CX=0.
CX is intended as a loop counter, and if CX=0 it's likely we will not want any iterations of our loop to occur.

We can achieve this by putting a "JCXZ SkipLoop" at the start of our loop, which will bypass the code if CX is zero.

Usage Example: JCXZ NoLoop

Flags Affected: O S Z A P C

JMP addr

Jump to address **addr**, which will usually be specified by a label, in which case the assembler will work out the correct address.

Usage Example: JMP Shutdown JMP BX JMP CX

Flags Affected: O S Z A P C

LAHF

Load AH from the Flags.
Not all the flags are transferred. This only transfers the main flags: SZ-A-P-C
See SAHF for the command to transfer from AH to the Flags.

Usage Example: LAHF

Flags Affected: - - - - -

LDS reg,addr

Load a full 32 bit pointer into DS segment register and register **reg**. The Segment is loaded into DS, the offset is loaded into **reg**. **addr** will usually be specified as a label.

Usage Example: LDS BX,TestPointer LDS AX,MyLabel

Valid Registers for 'reg': AX, BX, CX, DX, SI, DI

Flags Affected: - - - - -

LEA reg,src

Load the effective address **src** into **reg**.

Complex addressing modes, like "[ES:BP+DI-2]" or "[BX+DI]", have multiple stages to calculating the final address which will be used for the parameter. We may want to use this address many times, and LEA will 'pre calculate' the effective address and store it in another register.

This can be used to save CPU power, as using "LEA BX,[ES:BP+DI-2]" once and then using address [BX] twenty time in our code will be faster and more efficient than using "[ES:BP+DI-2]" twenty times.

Usage Example: LEA BX,[ES:BP+DI-2] LEA CX,[BX+DI]

Valid Registers for 'reg': AX, BX, CX, DX,SI, DI, BP, SP

Flags Affected: - - - - -

LES reg,addr

Load a full 32 bit pointer into ES segment register and register **reg**. The Segment is loaded into ES, the offset is loaded into **reg**. **addr** will usually be specified as a label.

Usage Example: LES BX,TestPointer LES AX,MyLabel

Valid Registers for 'reg': AX,BX,CX,DX,SI,DI

Flags Affected: - - - - -

LOCK

Enable the LOCK signal. This is for multiprocessor systems - you'll never need it for normal programming.

Usage Example: LOCK

Flags Affected: - - - - -

LODSB
LODSW

Load from DS:SI into AX or AL. This command can work in bytes or words.
LODSB will load one byte from DS:SI into AL.
LODSW will load one word from DS:SI into AX.

After the command, SI is increased by the amount read (1 or 2 bytes) if the direction flag is zero, or decreased if it is 1.

This command can be combined with repeat command REPZ or REPNZ, though it's unclear what the purpose would be.

Usage Example: LODSB LODSW

Flags Affected: - - - - -

LOOP addr

Decrease CX and jump to label **addr** if CX is not zero.

Usage Example: LOOP LoopLabel

Flags Affected: - - - - -

LOOPNZ addr
LOOPNE addr

Decrease CX and jump to label **addr** if CX is not zero and the Zero flag is not set.
This can be used to add an extra condition to the continuation of the loop.

LOOPNZ (Loop if Not Zero) and LOOPNE (Loop if Not Equal) are the same command, You can use either opcode for clarity in your code.

Usage Example: LOOPNZ LoopLabel

Flags Affected: - - - - -

LOOPZ addr
LOOPE addr

Decrease CX and jump to label **addr** if CX is not zero and the Zero flag is set.
This can be used to add an extra condition to the continuation of the loop.

LOOPZ (Loop if Zero) and LOOPE (Loop if Equal) are the same command. You can use either opcode for clarity in your code.

Usage Example: LOOPZ LoopLabel

Flags Affected: - - - - -

MOV dest,src

Move a value from source **src** to destination **dest**.
src can be an immediate value, and **dest** can be a register, in which the register will be loaded with the value.

It's not possible to directly move a SEGMENT value into CS, DS, ES or SS, this must be done first by moving into another register.
For Example: To set the ES register to the segment of the code block, we could use "MOV AX, @code MOV ES, AX".

Usage Example: MOV AX,SEG SOMEDATA MOV [ES:BX+offset],6655h
MOV BX,8 MOV AX,BX

Flags Affected: - - - - -

MOVSB
MOVSW

Move a byte or word from DS:SI to ES:DI.
This command can be combined with repeat command REP, in which case the command will repeat CX times.

After the command, DI and SI are increased by the compare size (1 or 2 bytes) if the direction flag is zero, or decreased if it is 1.

Usage Example: MOVSB REPZ MOVSB REP MOVSW

Flags Affected: - - - - -

MUL src

Multiply unsigned number AX or DX.AX by **src**.

If **src** is 8 bit (BL,CH etc) then the destination will be 16 bit and the formula used is AX=AL***src.**

If **src** is 16 bit (BX,CX etc) then the destination will be 32 bit and the formula used is DX.AX=AX***src** (where DX.AX is a 32 bit pair).

Usage Example: MUL BL MUL CH MUL CX

Flags Affected: O s z a p C

NEG dest

Negate **dest** (Twos Complement of the number). This converts a positive number to a negative, or a negative to a positive.

Usage Example: NEG AL NEG AX NEG WORD PTR [TestLabel]

Flags Affected: O S Z A P C

NOP

No Operation. This command has no effect on any registers or memory. NOP can be used as a short delay, or as a 'spacer' for bytes that will be altered via self modifying code.

Usage Example: NOP

Flags Affected: - - - - - -

NOT dest

Invert/Flip all the bits of **dest**. This is known as One's Complement.
For Example: If AL=11000000b and we use "NOT AL" then AL will become 00111111b.

Usage Example: NOT.L BX NOT AL NOT WORD PTR [TestLabel]

Flags Affected: - - - - - -

OR dest,src

This command logically ORs the **src** and **dest** parameter together.
Logical OR will set bits in the destination according to the source and destination. Where a bit in either parameter is 1 the result will be 1, when both are 0 the result will be 0.

For Example: If AL=10101010b and BL=11110000b, the command "OR AL,BL" will result in AL=11111010b .

Usage Example: OR AX,BX OR AL,BL OR AL,BYTE PTR [TestLabel]

Flags Affected: O S Z a P C

OUT port,src

Send an 8 bit byte or 16 bit word from **src** (either AX, AL or AH) to hardware port number **port**.

port can either be an 8 bit immediate (0-255) or DX. The only way to access a 16 bit port (0-65535) is via DX.

Usage Example: OUT 100,AL OUT F0h ,AX OUT DX,AL OUT DX,AX

Flags Affected: - - - - - -

POP reg

Pop a pair of bytes off the stack into 16 bit register **reg**. Two bytes are taken from the top of the stack, and 2 is added to the SP register.

This has the effect of 'restoring' the previous value of **reg**.

Usage Example: POP AX POP DI POP ES

Valid Registers for 'reg': AX, BX, CX, DX, SI, DI, SP, BP, CS, DS, ES, SS

Flags Affected: - - - - - -

POPF

Pop a pair of bytes off the stack into the 16 bit Flags register. Two bytes are taken from the top of the stack and put into the flags, and 2 is added to the SP register.

This has the effect of 'restoring' the previous value of the flags. It's also the only way to write all 16 bits of the flag register. One such example would be a combination of "PUSH AX POPF".

Usage Example: POPF

Flags Affected: O D I T S Z A P C

PUSH reg

Push a pair of bytes from 16 bit register **reg** onto the top of the stack. The two bytes from **reg** are put onto the top of the stack, and 2 is subtracted from the SP register.

This has the effect of 'Backing up' the current value of **reg**.

Usage Example: PUSH AX PUSH DI PUSH ES

Valid Registers for 'reg': AX, BX, CX, DX, SI, DI, SP, BP, CS, DS, ES, SS

Flags Affected: - - - - - -

PUSHF

Push a pair of bytes off the stack into the 16 bit Flags register. The two bytes from the flags are put onto the top of the stack, and 2 is subtracted from the SP register.

This has the effect of 'Backing up' the current value of the flags. It's also the only way to read all 16 bits of the flag register. One such example would be a combination of "PUSHF POP AX".

Usage Example: PUSHF

Flags Affected: - - - - - -

RCL dest,count

Rotate bits in Destination **dest** to the Left by **count** bits, with the carry flag acting as an extra bit (eg an 8th bit when rotating a byte).

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple rotates to the left:

Carry	76543210
0	11000010
1	10000100
1	00001001
0	00010011

Usage Example: RCL AX,1 RCL AL,1 RCL AL,CL RCL AX,2 {186 only}

Flags Affected: O - - - - C

RCR dest,count

Rotate bits in Destination **dest** to the Right by **count** bits, with the carry flag acting as an extra bit (eg an 8th bit when rotating a byte).

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple rotates to the right:

76543210	Carry
11000010	0
01100001	0
00110000	1
10011000	0

Usage Example: RCR AX,1 RCR AL,1 RCR AL,CL RCR AX,2 {186 only}

Flags Affected: O - - - - C

REP stringop

Repeat string operation **stringop** while CX>0. Decrease CX after each iteration of **stringop**.

Usage Example: REP LODSB REP LODSW REP STOSB

Valid options for 'stringop': LODS, MOVS, STOS

Flags Affected: - - - - - -

REPE stringop
REPZ stringop

Repeat string operation **stringop** while the Z flag is set and CX>0. Decrease CX after each iteration of **stringop**.

REPE and REPZ are the same command, you can use either opcode in your code for clarity.

Usage Example: REPZ CMPSB REPE CMPSW REPZ SCASB

Valid options for 'stringop': CMPS, SCAS

Flags Affected: - - - - - -

REPNE stringop
REPNZ stringop

Repeat string operation **stringop** while the Z flag is not set and CX>0. Decrease CX after each iteration of **stringop**.

REPNE and REPNZ are the same command, you can use either opcode in your code for clarity.

Usage Example: REPNZ CMPSB REPNE CMPSW REPNZ SCASB

Valid options for 'stringop': CMPS, SCAS

Flags Affected: - - - - - -

RET

Return from a subroutine. The return address is taken from the top of the stack and put in the program counter. This is effectively like a "POP PC" command.

Usage Example: RET

Flags Affected: - - - - - -

ROL dest,count

Rotate bits in Destination **dest** to the Left by **count** bits, with the carry flag copying the top bit.

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.
The bit in the Carry is never put into the destination. The top bit is moved into these flags so you can use it with another command.

For Example: Let's see how the bits in a byte change with multiple rotates to the left:

Carry	76543210
0	11000010
1	10000101
1	00001011
0	00010110

Usage Example: ROL AX,1 ROL AL,1 ROL AL,CL ROL AX,2 {186 only}

Flags Affected: O - - - - C

ROR dest,count

Rotate bits in Destination **dest** to the Right by **count** bits, with the carry flag copying the bottom bit.
The bit in the Carry is never put into the destination. The top bit is moved into these flags so you can use it with another command.

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple rotates to the right:

76543210	Carry
11000010	0
01100001	0
10110000	1
01011000	0

Usage Example: ROR AX,1 ROR AL,1 ROR AL,CL ROR AX,2 {186 only}

Flags Affected: O - - - - C

SAHF

Store AH to the Flags.
Not all the flags are transferred. This only transfers the main flags: SZ-A-P-C .

See LAHF for the command to transfer from AH to the Flags.

Usage Example: SAHF

Flags Affected: - - - - -

SAL dest,count

Shift the bits for Arithmetic in Destination **dest** to the Left by **count** bits, with the carry flag copying the top bit. All new Bit 0s will be Zero.

'Arithmetic' means the sign is maintained as the left shift occurs, though, unlike SAR, SAL and SHL have the same function.
The bit in the Carry is never put into the destination. The top bit is moved into these flags so you can use it with another command.

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple shifts to the left:

Carry	76543210
0	11000010
1	10000100
1	00001000
0	00010000

Usage Example: SAL AX,1 SAL AL,1 SAL AL,CL SAL AX,2 {186 only}

Flags Affected: O - - - - C

SAR dest,count

Shift the bits for Arithmetic in Destination **dest** to the Right by **count** bits with the carry flag copying the bottom bit. 'Arithmetic' means the sign is maintained as the right shift occurs, This is achieved by the new top bit being set to the previous top bit.

The bit in the Carry is never put into the destination. The top bit is moved into these flags so you can use it with another command.

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple shifts to the right:

76543210	Carry
11000010	0
11100001	0
11110000	1
11111000	0

Usage Example: SAR AX,1 SAR AL,1 SAR AL,CL SAR AX,2 {186 only}

Flags Affected: O - - - - C

SBB dest,src

Subtract **src** and the Borrow (carry flag) from **dest**.

Usage Example: SBB AX,BX SBB AL,BL SBB CX,1000h
SBB [1000h],DX

Flags Affected: O S Z A P C

SCASB
SCASW

Scan ES:DI and compare to AX or AL. This command can work in bytes or words. The flags are set like a CMP command.

If SCASB is used, AL will be used for the compare. If SCASW is used, AX will be used for the compare.

This command can be combined with repeat command REPZ or REPNZ.
REPZ will cause the repeat to continue until the string no longer matches AX/AL or CX=0.
REPNZ will cause the repeat to continue until the string starts to match AX/AL or CX=0.

After the command, DI is increased by the size of the comparison (1 or 2 bytes) if the direction flag is zero, or decreased if it is 1.

Usage Example: SCASB REPZ SCASB REPNZ SCASW

Flags Affected: O S Z A P C

SHL dest,count

Shift the bits logically Left in destination **dest** by **count** bits, with the carry flag copying the top bit. All new Bit 0s will be Zero.
The bit in the Carry is never put into the destination. The top bit is moved into these flags so you can use it with another command.

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple shifts to the left:

Carry	76543210
0	11000010
1	10000100
1	00001000
0	00010000

Usage Example: SHL AX,1 SHL AL,1 SHL AL,CL SHL AX,2 {186 only}

Flags Affected: O - - - - C

SHR dest,count

Shift the bits logically Right in destination **dest** by **count** bits, with the carry flag copying the bottom bit. All new top bits will be Zero.
The bit in the Carry is never put into the destination. The top bit is moved into these flags so you can use it with another command.

On the 8086, **count** must either be 1 or register CL. On the 80186 and later this can be any number.

For Example: Let's see how the bits in a byte change with multiple shifts to the right:

76543210	Carry
11000010	0
01100001	0
00110000	1
00011000	0

Usage Example: SHR AX,1 SHR AL,1 SHR AL,CL SHR AX,2 {186 only}

Flags Affected: O - - - - C

STC

Set the Carry Flag. C flag will be set to 1.

Usage Example: STC

Flags Affected: - - - - - C

STD

Set the Direction Flag. D flag will be set to 1. This is used for 'String functions'.

The name 'String functions' is not necessarily to do with text.
On the 8086 'strings' are for batch data copying from one memory address to another.
Setting the Direction flag means the source and destination addresses go down after each transfer.

Usage Example: STD

Flags Affected: D - - - - - -

STI

Set the Interrupt enable flag. I flag will be set to 1. This enables maskable interrupts.

Usage Example: STI

Flags Affected: I - - - - - -

STOSB
STOSW

Store AX or AL to ES:DI. This command can work in bytes or words.
If the command is STOSB then AL wlll be used, if the command is STOSW then AX will be used to fill the destination.

This command can be combined with repeat command REP to fill a range until CX=0.

After the command, DI is increased by the amount stored (1 or 2 bytes). If the direction flag is zero, or decreased if it is 1.

Usage Example: STOSB REP STOSB REP STOSW

Flags Affected: - - - - - -

SUB dest,src

Subtract **src** from **dest**.

Usage Example: SUB AX,BX SUB AL,BL SUB CX,1000h
SUB [1000h],DX

Flags Affected: O S Z A P C

TEST dest,src

Test **dest**, setting the flags in the same way a logical "AND src" would, setting the flags but leaving both **dest** and **src** unchanged.

This can be used to check if several bits are zero at the same time.
For Example: "TEST AL,00000011b" will test if both bits 0 and 1 are zero.

Usage Example: TEST BX,64h TEST AX,1100h

Flags Affected: O S Z A P C

WAIT

Wait until the busy pin of the CPU is inactive.
This is used for coprocessors, you'll probably never need it.

Usage Example: WAIT

Flags Affected: O S Z A P C

XCHG reg1,reg2

Exchange the contents of registers **reg1** and **reg2**. No other registers or memory are affected. You can swap 8 or 16 bit registers, but both parameters must be the same length, so "XCHG AX,AL" is impossible.

Usage Example: XCHG BH,AL XCHG BX,CX XCHG AX,DI

Flags Affected: - - - - -

XLAT

Translate AL using lookup table DS:BX. AL is read from memory address [DS:BX+AL].

The intention is BX will point to some kind of translation table. For Example: If AL is expected to be 0-9, we could point BS to a lookup table of the ASCII characters for 0-9 and use XLAT to convert a decimal number to its respective ASCII character.

Usage Example: XLAT

Flags Affected: - - - - - -

Chapter 6: The ARM

Introducing the ARM

The ARM is a RISC CPU. This means it's more limited in some ways. We may need more commands to do a job than we would 'expect', but it's faster and more power efficient as a result, which is why it's become so popular in mobile phones.

The ARM offers some very unusual features, which are pretty unique to the ARM.

The first of these is the fact that most commands can be 'conditional'. Based on the flags, we can have individual commands that execute or take no action. This reduces the number of branches. To make a command execute conditionally, we simply add the two letter condition code to the end of a command.

We can also specify which commands should update the flags, and which should leave them unchanged. To specify a command should alter the flags, we simply add an 'S' to the end of a command.

Finally there is the 'Barrel Shifter'. This allows a register to be used as a parameter, but some kind of 'shift' to be applied to that register (often a multiplication by a power of 2, so values like 2, 4, 8 and so on).

The surprising thing is the value in the register is unchanged. The shift is applied during the calculation only. This means we can have a register used as a loop counter going from 0-5, but use that register to calculate address offsets 0,4,8,12,16 and 20.

On the early ARM processors, all the instructions were 32 bit. The ARM7TDMI added a more limited 'THUMB' mode, which used 16 bit instructions. This saves memory, and can also be faster in some circumstances, but coding takes more commands. This book only covers the normal 32 bit ARM instructions, which are more powerful and will work on all CPUs.

In these tutorials we'll cover the basic commands up to the ARM4, which will give you all the essential commands for programming the GBA/NDS handhelds, and RISC OS.

The ARM Registers

All ARM registers are 32 bit.
A normal programmer will be running a program in 'User Mode'. In this context the ARM has 16 registers. R0-R12 are free for us to do whatever we want, R13 is the Stack Pointer (also addressable as SP), R15 is the Program Counter (PC).

R14/LR (the Link Register) also has a special purpose as it is used by subroutines. We'll discuss this in detail in a moment.

Register	Purpose
R0	General
R1	General
R2	General
R3	General
R4	General
R5	General
R6	General
R7	General
R8	General
R9	General
R10	General
R11 / FP	Frame Pointer (Optional)
R12 / IP	Intra Procedural Call (Optional)
R13 / SP	Stack Pointer
R14 / LR / LK	Link Register
R15 / PC	Program Counter

A "Frame pointer" points to data areas in the Stack, effectively allocating a 'small stack allocation' for a subroutine. This frame pointer register would be used with relative offsets. It's 'suggested' R11 is used for this purpose, if you need it. It's entirely optional if you use R11 for this or not.

This is also true of R12, the Intra Procedural Call register. It can be used as a backup of LR for subroutines, if you wish.

The ARM technically has 27 general purpose 32 bit registers, but all but 16 are hidden from the user.

There are also multiple "SPSR registers", which act as the flags used during interrupts, but as a general programmer you'll not need to worry about these.

The Link Register

The concept of a Link Register (R14) may be surprising to those familiar with other CPUs. When we call a subroutine (with BL - Branch and Link) the return address is not pushed onto the stack like other CPUs, instead it's moved into R14/LR. To return from the subroutine we need to move the R14/LR register into R15/PC.

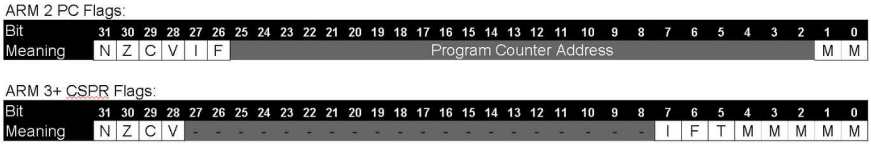
This poses a problem, as nesting subroutines will lose the return value. If this is needed, the best solution is to simply push R14/LR onto the stack at the start of a subroutine, and pop PC/R15 off the stack at the end.

```
TestSub1:   ;Simple Sub
        ;Your code goes here
    MOV pc,lr                   ;Return to calling sub

TestSub2:   ;Nestable Sub
    STMFD sp!,{r0-r12, lr}      ;Push Regs
        ;Your code goes here
    LDMFD sp!,{r0-r12, pc}      ;Pop Regs and return
```

Figure 60: Using the Link Register (LR) for returning from a Subroutine.

The ARM Flags

On the ARM2, the flags were stored in unused bits of the PC Register (the top 6, and bottom 2 bits), with the ARM3+ a register called the CSPR is the main flags register.



In the instruction documentation of this book you will see descriptions like "ADC*ccS"*
The "ccS" part is optional, the ADC command can be used on its own.

Unlike most processors, commands on the ARM do not update the flags automatically. If we want a command to update the flags we need to add an "S" to the end.

For Example: "ADC R0,R1,R2" will not update the flags, but "ADCS R0,R1,R2" will update the flags.

Flag	Name	Description
N	Negative	Signed Less Than
Z	Zero	Zero
C	Carry	Carry / Not Borrow / Rotate Extend
V	oVerflow	Overflow
I	IRQ Disable	1=disable
F	FIQ disable	1=disable
T	Thumb mode	V4 only
M	Mode	00=user 01=FIQ 10=IRQ 11=Supervisor

ARM Condition Codes

The majority of commands on the ARM can be made 'conditional' by adding an abbreviated condition code to the instruction. The cc in command "ADCccS" is the optional 'condition code'.

We can specify a condition code with our command by putting a condition after the command opcode. For Example: The command "ADCEQ R0,R1,R2". In this case the command "ADC" will only run if the "EQ" condition applies (if Z=1).

Abbreviation	Meaning	Flag
EQ	EQual	Z=1
NE	Not Equal	Z=0
CS **HS**	Carry Set Higher or Same (Unsigned)	C=1
CC **LO**	Carry Clear LOwer (Unsigned)	C=0
MI	MInus (Negative)	N=1
PL	PLus (Positive)	N=0
VS	oVerflow Set	V=1
VC	oVerflow Clear	V=0
HI	HIgher (Unsigned)	C=1 and Z=0
LS	Lower or Same (Unsigned)	C=0 and Z=1
GE	Greater or Equal (Signed)	N=V
LT	Less Than (Signed)	N<>V
GT	Greater Than (Signed)	Z=0 and N=V
LE	Less than or Equal (Signed)	Z=1 or N<>V
AL	ALways	No condition

Defining bytes of data on the ARM in VASM

There will be many times when we need to define bytes of data in our code. Examples of this would be bitmap data, the score of our player, a string of text, or the co-ordinates of an object.
The commands will vary depending on the CPU and your assembler, but the ones shown below are the ones used by the assembler covered in these tutorials.

For Example: If we want to define a 32 bit sequence $12345678 we would use ".LONG $12345678".

Note: As it's a 32 bit processor, 'normally' on the ARM a Word would be 32 bit. It seems VASM in 'STD' syntax mode uses 16 bit CPU sizes.

Bytes	Z80	68000	8086	ARM
1	DB	DC.B	DB	.BYTE
2	DW	DC.W	DW	.WORD
4		DC.L	DD	.LONG
n	DS n,x	DS n,x	n DUP (x)	.SPACE n,x

ARM Addressing Modes

The addressing modes depend on the command, but there are two general addressing modes used by the majority of commands.

Operand2 (Op2) Addressing Modes

Immediate

A fixed number value.
This can be any value made by an 8 bit immediate shifted by an even number of bits.
For Example: 0xFF or 0xFF000000 are possible values.

Format Template: #n

Usage Example: ADD R0,R0,#1 ADD R0,R0,#0x1000

Register

A value in a specified register.

Format Template: Rn

Usage Example: ADD R0,R0,R1 ADD R0,R2,R3

Register Shifted by Immediate

A value in a specified register, shifted by a fixed number of bits.

Shift	Details
LSL #n	Logically shift Left **n** bits. All new bottom bits are zeros. This doubles an unsigned number.
LSR #n	Logically shift Right **n** bits All new top bits are zeros. This halves an unsigned number.
ASR #n	Arithmetic shift Right **n** bits. 'Arithmetic shifts' double a signed number, all new top bits are the same as the old top bit, this keeps the sign unchanged. LSL will work fine for shifting signed numbers left.
ROR #n	Rotate bits Right **n** bits, any bits that go off the right side of the register come back on the Left. There is no left rotate, but ROR #31 has the same effect as rotating left 1 bit would.
RRX	Rotate Right with eXtend. This rotates right one bit only, the old bit 0 becomes the carry, the old carry becomes bit 32.

Format Template: Rn, shft #n

Usage Example: MOV R0,R1,ROR #2

Register Shifted by Register

A value in a specified register, shifted by a number of bits specified by a register.

RRX cannot be used with a register shift.

Shift	Details
LSL Rm	Logically shift Left **Rm** bits. All new bottom bits are zeros. This doubles an unsigned number
LSR Rm	Logically shift Right **Rm** bits All new top bits are zeros. This halves an unsigned number.
ASR Rm	Arithmetic shift Right **n** bits. 'Arithmetic shifts' double a signed number, all new top bits are the same as the old top bit, this keeps the sign unchanged. LSL will work fine for shifting signed numbers left.
ROR Rm	Rotate bits Right **Rm** bits, any bits that go off the right side of the register come back on the Left. There is no 'ROL' command, but, as all the bits rotate around the 32 bit register, "ROR #31" has the same effect as "ROL #1" would.

Format Template: Rn, shft Rm

Usage Example: MOV R0,R1,ROR R2

Flexible Offset (Flex) Addressing Modes

Immediate offset / Immediate pre-indexed

The value is taken from the address in a register, plus an immediate value as an offset **n** (can be zero). The immediate offset can also be negative.

The value is effectively taken from the address in register **Rn**+**n**.

If the command ends with an exclamation mark "!" it is defined as Pre-indexed. This means Rn is altered, setting it to **Rn=Rn**+**n**.

Format Template: [Rn,#n] or [Rn,#n]!

Usage Example: LDR R0,[R1] LDR R0,[R1,#4] LDR R0,[R1,#-4]!

Register offset / Register pre-indexed

The value is taken from the address in a register, plus an immediate value as an offset **n** (can be zero)
The value is effectively taken from the address in register **Rn**+**Rm**.

A minus sign can be specified with the register offset to subtract it from the calculation.

If the command ends with an exclamation mark "!" it is defined as Pre-indexed. This means Rn is altered, setting it to **Rn**=**Rn**+**Rm**.

Format Template: [Rn,{-}Rm] or [Rn,{-}Rm]!

Usage Example: LDR R0,[R1,-R2] LDR R0,[R1,R2]!

Scaled register offset / Scaled register pre-indexed

The value is taken from the address in a register with a shift of **n** (can be zero)
If the shift is LSL #2, then the value is effectively taken from the address in register **Rn**+(**Rm LSL #2)**.

A minus sign can be specified with the register offset to subtract it from the calculation.

If the command ends with an exclamation mark "!" it is defined as Pre-indexed. This means Rn is altered, setting it to **Rn**=**Rn**+(**Rm LSL #2**).

Shift	Details
LSL #n	Logically shift Left **n** bits. All new bottom bits are zeros. This doubles an unsigned number.
LSR #n	Logically shift Right **n** bits All new top bits are zeros. This halves an unsigned number.
ASR #n	Arithmetic shift Right **n** bits. 'Arithmetic shifts' double a signed number, all new top bits are the same as the old top bit, this keeps the sign unchanged. LSL will work fine for shifting signed numbers left.
ROR #n	Rotate bits Right **n** bits, any bits that go off the right side of the register come back on the Left. There is no left rotate, but ROR #31 has the same effect as rotating left 1 bit would.
RRX	Rotate Right with eXtend. This rotates right one bit only, the old bit 0 becomes the carry, the old carry becomes bit 32.

Format Template: Rn, [Rm,shft #n] / Rn, [Rm,shft #n]!

Usage Example: LDR R0,[R1,R2, LSL #2] / LDR R0,[R1,R2, LSL #2]!

Immediate post-indexed

The value is taken from the address specified by a register.
This is Postindexed. This means **Rn** is altered after the read, an immediate value **n** is added to Rn, effectively setting it to **Rn**=**Rn**+**n**.

Format Template: [Rn],#n

Usage Example: LDR R0,[R1],#4

Register post-indexed

The value is taken from the address specified by a register, but the address is then changed. This is Postindexed. This means **Rn** is altered after the read, **Rm** is added to **Rn**, effectively setting it to **Rn**=**Rn**+**Rm**.

A minus sign can be specified with the register offset to subtract it from the calculation.

Format Template: [Rn], {-}Rm

Usage Example: LDR R0,[R1],R2 LDR R0,[R1],-R2

Scaled register post-indexed

The value is taken from the address specified by a register with a shift, the address is then changed.
This is Postindexed. This means Rn is altered after the read. If the shift is LSL #2, then after the read **Rn=Rn**+(**Rm LSL #2)**.

A minus sign can be specified with the register offset to subtract it from the calculation.

Shift	Details
LSL #n	Logically shift Left **n** bits. All new bottom bits are zeros. This doubles an unsigned number.
LSR #n	Logically shift Right **n** bits All new top bits are zeros. This halves an unsigned number.
ASR #n	Arithmetic shift Right **n** bits. 'Arithmetic shifts' double a signed number, all new top bits are the same as the old top bit, this keeps the sign unchanged. LSL will work fine for shifting signed numbers left.
ROR #n	Rotate bits Right **n** bits, any bits that go off the right side of the register come back on the Left. There is no left rotate, but ROR #31 has the same effect as rotating left 1 bit would.
RRX	Rotate Right with eXtend. This rotates right one bit only, the old bit 0 becomes the carry, the old carry becomes bit 32.

Format Template: [Rn], {-}Rm, shft #n

Usage Example: LDR R0,[R1],R2,LSL #2 LDR R0,[R1],-R2,RRX

Compiling ARM with VASM, and running with the VisualBoyAdvance emulator

Writing an assembler program

An ASM file is just a text file! You can edit it with the text editor of your choice. I use Notepad++, but you can use Windows Notepad, Visual Studio Code, or anything else you like.

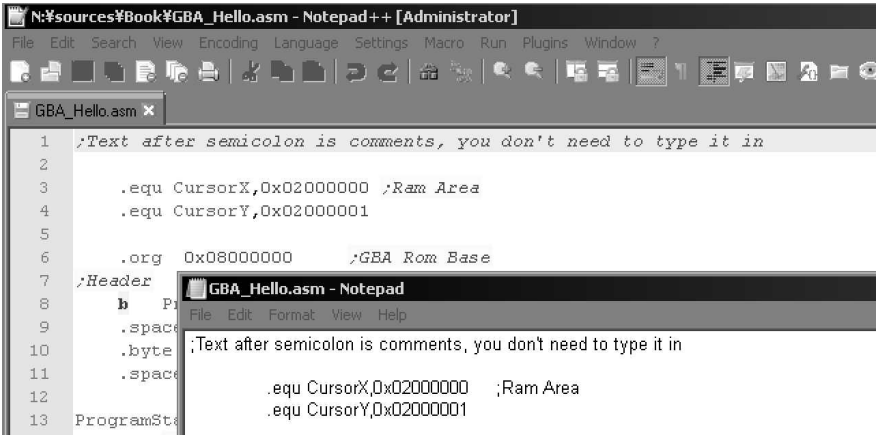


Figure 61: ASM files are just text files. Even classic Notepad can do the job, but an editor like Notepad++ offers syntax highlighting.

Using VASM
VASM is a fantastic Assembler for retro coding, as it supports 6502, Z80, 68000 and even ARM!

VASM is free, and can be found at:
http://sun.hasenbraten.de/vasm/

VASM is a Command line tool, we run it from a CMD prompt (Run CMD.exe in Windows).

Here is a minimal example of a script to compile an ASM file into a Gameboy Advance style Cartridge:

vasmarm_std_win32.exe "n:\sources\Book\GBA_Hello.asm" -m7tdmi -DBuildGBA=1 -Fbin -o "\BldGBA\program.gba"

This will compile the source file "n:\sources\Book\GBA_Hello.asm" into a binary file our emulator can run. The built file will be saved to "\BldGBA\program.gba". We're also specifying ARM7TMDI support.

Here is a more detailed example with some extra features:

vasmarm_std_win32.exe "n:\sources\Book\GBA_Hello.asm" -m7tdmi -chklabels -nocase -Dvasm=1 -L \BldGBA\Listing.txt -DBuildGBA=1 -Fbin -o "\BldGBA\ program.gba"

This will also compile the program, however this version has a few extra features that may help:
1. It will ignore case differences in labels, which may reduce your debugging.

2. It will check for 'suspicious' label names. If you forget to put a tab before a command, it will be mistaken for a label. This "chklabels" switch will find those mistakes.

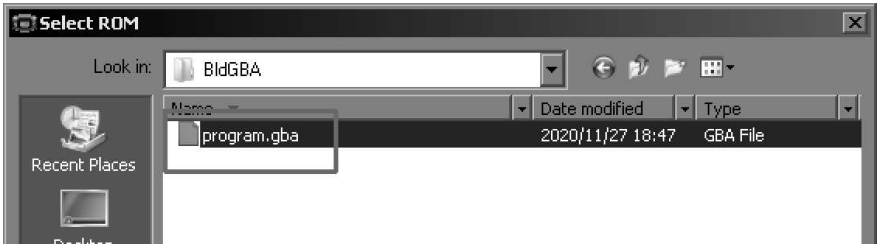
3. It will create a listing file called "Listing.txt". This is to help with debugging, it shows the source code and how the code compiles to bytes.

Either can be used to create the examples below, the second just has some helpful debugging options.

Running with the VisualBoyAdvance emulator

We're going to use VisualBoyAdvance to run our compiled program, To start the program we just compiled select "File menu – Open".

Select the cartridge file we just compiled.



The emulator will reboot and run the program.

You can also start a cartridge from the command line

ARM Examples

Example 1: Hello World

This Example will show a 'Hello World' Message on the Gameboy Advance.
Compile this with VASM in the way described on the previous pages.

Please note: The example source below works fine with VisualBoyAdvance emulator.
However the cartridge header is not technically correct, so may not work on other emulators.

This is because the example has been cut down to the bare minimum to make a concise example, Please see the ChibiAkumas.com website for more complete examples.

```
;Text after semicolon is comments, you don't need to type it in

    .equ CursorX,0x02000000 ;Ram Area
    .equ CursorY,0x02000001

    .org  0x08000000      ;GBA Rom Base
;Header
    b   ProgramStart      ;Jump to our program
    .space 178            ;Logo (omitted) + Program name
    .byte 0x96            ;Fixed value
    .space 49             ;Dummy Header

ProgramStart:
    mov sp,#0x03000000            ;Init Stack Pointer

    mov r4,#0x04000000            ;DISPCNT -LCD Control
    mov r2,#0x403                 ;4= Layer 2 on / 3= ScreenMode 3
    str r2,[r4]

    ldr r1,HelloWorldAddress      ;Address of Hello World Message
    bl PrintString                ;Show Message

InfLoop:
     b InfLoop                    ;Halt

HelloWorldAddress:
    .long HelloWorld              ;Pointer to Hello message

HelloWorld:
    .byte "Hello World",255
    .align 4                      ;Must be 32 bit aligned

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

(Continued on the next page)

```
35  PrintString:                    ;Print 255 terminated string
36      STMFD sp!,{r0-r12, lr}
37  PrintStringAgain:
38          ldrB r0,[r1],#1
39          cmps r0,#255
40          beq PrintStringDone     ;Repeat until 255
41          bl PrintChar            ;Print Char
42          b PrintStringAgain
43  PrintStringDone:
44      LDMFD sp!,{r0-r12, pc}
45
46  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
47
48  PrintChar:
49      STMFD sp!,{r0-r12, lr}
50          cmp r0,#32              ;Space?
51          beq LineDone
52          mov r4,#0
53          mov r5,#0
54
55          mov r3,#CursorX
56          ldrB r4,[r3]            ;X pos
57          mov r3,#CursorY
58          ldrB r5,[r3]            ;Y pos
59
60          mov r3,#0x06000000      ;VRAM base
61
62          mov r6,#16              ;Xpos, 2 bytes per pixel, 8 bytes per char
63          mul r2,r4,r6
64          add r3,r3,r2
65
66          mov r4,#240*8*2         ;Ypos, 240 pixels per line,
67          mul r2,r5,r4                ;2 bytes per pixel, 8 lines per char
68          add r3,r3,r2
69
70          adr r4,BitmapFont       ;Font source
71          and r0,r0,#0b11011111   ;Convert to upper case
72
73          sub r0,r0,#65           ;First Char is 65 (A)
74          add r4,r4,r0,asl #3     ;8 bytes per char
75
76          mov r1,#8               ;8 lines
77  DrawLine:
78          mov r7,#8               ;8 pixels per line
79          ldrb r8,[r4],#1         ;Load Letter
80          mov r9,#0b100000000     ;Mask
81
```

(Continued on the next page)

```
        mov r2, #0b1111111101000000; Color: ABBBBBGGGGGRRRRR    A=Alpha
DrawPixel:
        tst r8,r9               ;Is bit 1?
        strneh r2,[r3]          ;Yes? then fill pixel (HalfWord)
        add r3,r3,#2
        mov r9,r9,ror #1        ;Bitshift Mask
        subs r7,r7,#1
        bne DrawPixel           ;Next Hpixel

        add r3,r3,#480-16      ;Move Down a line (240 pixels *2 bytes)
        subs r1,r1,#1                               ;-1 char (16 px)
        bne DrawLine            ;Next Vline
LineDone:
        mov r3,#CursorX
        ldrB r0,[r3]
        add r0,r0,#1            ;Move across screen
        strB r0,[r3]
    LDMFD sp!,{r0-r12, pc}

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

BitmapFont: ;1 Bit per pixel font
    .byte 0x18,0x3C,0x66,0x66,0x7E,0x66,0x24,0x00     ;A
    .byte 0x3C,0x66,0x66,0x7C,0x66,0x66,0x3C,0x00     ;B
    .byte 0x38,0x7C,0xC0,0xC0,0xC0,0x7C,0x38,0x00     ;C
    .byte 0x3C,0x64,0x66,0x66,0x66,0x64,0x38,0x00     ;D
    .byte 0x3C,0x7E,0x60,0x78,0x60,0x7E,0x3C,0x00     ;E
    .byte 0x38,0x7C,0x60,0x78,0x60,0x60,0x20,0x00     ;F
    .byte 0x3C,0x66,0xC0,0xC0,0xCC,0x66,0x3C,0x00     ;G
    .byte 0x24,0x66,0x66,0x7E,0x66,0x66,0x24,0x00     ;H
    .byte 0x10,0x18,0x18,0x18,0x18,0x18,0x08,0x00     ;I
    .byte 0x08,0x0C,0x0C,0x0C,0x4C,0xFC,0x78,0x00     ;J
    .byte 0x24,0x66,0x6C,0x78,0x6C,0x66,0x24,0x00     ;K
    .byte 0x20,0x60,0x60,0x60,0x60,0x7E,0x3E,0x00     ;L
    .byte 0x44,0xEE,0xFE,0xD6,0xD6,0xD6,0x44,0x00     ;M
    .byte 0x44,0xE6,0xF6,0xDE,0xCE,0xC6,0x44,0x00     ;N
    .byte 0x38,0x6C,0xC6,0xC6,0xC6,0x6C,0x38,0x00     ;O
    .byte 0x38,0x6C,0x64,0x7C,0x60,0x60,0x20,0x00     ;P
    .byte 0x38,0x6C,0xC6,0xC6,0xCA,0x74,0x3A,0x00     ;Q
    .byte 0x3C,0x66,0x66,0x7C,0x6C,0x66,0x26,0x00     ;R
    .byte 0x3C,0x7E,0x60,0x3C,0x06,0x7E,0x3C,0x00     ;S
    .byte 0x3C,0x7E,0x18,0x18,0x18,0x18,0x08,0x00     ;T
    .byte 0x24,0x66,0x66,0x66,0x66,0x66,0x3C,0x00     ;U
    .byte 0x24,0x66,0x66,0x66,0x66,0x3C,0x18,0x00     ;V
    .byte 0x44,0xC6,0xD6,0xD6,0xFE,0xEE,0x44,0x00     ;W
    .byte 0xC6,0x6C,0x38,0x38,0x6C,0xC6,0x44,0x00     ;X
    .byte 0x24,0x66,0x66,0x3C,0x18,0x18,0x08,0x00     ;Y
    .byte 0x7C,0xFC,0x0C,0x18,0x30,0x7E,0x7C,0x00     ;Z

```

The program will run in the emulator and show a 'Hello World' message:

How does it work?

Our cartridge starts with a dummy header. It's not technically correct, this is to keep it short for typing in. This short header will suffice for running on the emulator. (Lines 6-11)

Please see the ChibiAkumas.com website for examples with complete headers.

The GBA does not have a firmware font, so we defined a 1 bit per pixel font (1 byte per line). We'll need to convert this to 16bpp for the GBA screen format. (Lines 103-129)

We read in a character from the string we want to show, then look up the matching character in our font.

The VRAM screen starts at 0x06000000, each pixel is 2 bytes.

We use two variables, CursorX and CursorY to track the position where the next character should be shown. These are defined in the RAM memory 0x02000000+. (Lines 3-4)

Each pixel is 2 bytes, and our font is 8 pixels wide, so we multiply the CursorX position by 16.
Each line of the screen is 240 pixels (480 bytes), and our font is 8 lines tall, so we multiply the CursorY position by 240*8*2.

We add these to the screen base. We have now calculated our Screen Address for the character to be shown in R3. (Lines 55-68)

We need to convert the ASCII character to a character number in our font. We also need to convert to uppercase. (Lines 70-74)

We then work through the 8 lines of our font, and shift a bit out of the 1 bit per pixel font. Depending on that bit we set the matching pixel of the screen with the font color, or just skip over that pixel leaving it unchanged. (Lines 77-89)

We repeat until the line is done, then add #480-16 to the screen position (the length of one line minus the number of bytes in one character). (Line 91)

We repeat this procedure until all 8 lines of the font are done, increase the cursorX position, then return to do the next character. (Line 92-99)

Example 2: Test Bitmap

This Example will show a Smiley using 15 bits per pixel color.

Compile this with VASM in the way described on the previous pages.

```
;Text after semicolon is comments, you don't need to type it in

    .org  0x08000000            ;GBA Rom Base
;Header
    b   ProgramStart            ;Jump to our program
    .space 178                  ;Logo (omitted) + Program name
    .byte 0x96                  ;Fixed value
    .space 49                   ;Dummy Header

ProgramStart:
    mov sp,#0x03000000          ;Init Stack Pointer

    mov r4,#0x04000000          ;DISPCNT -LCD Control
    mov r2,#0x403               ;4= Layer 2 on / 3= ScreenMode 3
    str r2,[r4]

    mov r10,#0x06000000         ;VRAM base
    ldr r1,SpriteAddress        ;Sprite Address
    mov r6,#8                   ;Height
Sprite_NextLine:
    mov r5,#8                   ;Width (in words / pixels)

    STMFD sp!,{r10}
Sprite_NextByte:
        ldrH r0,[r1],#2         ;Must write 16/32bit per VRAM write
        strH r0,[r10],#2

        subs r5,r5,#1           ;X Loop
        bne Sprite_NextByte
    LDMFD sp!,{r10}
    add r10,r10,#240*2          ;240 - 2 bytes per pixel
    subs r6,r6,#1
    bne Sprite_NextLine         ;Y loop

InfLoop:
      b InfLoop                 ;Pause Game

SpriteAddress:
    .long SpriteTest            ;Address of Sprite

;1 word per byte n the format:
;   ABBBBBGGGGGRRRRR - 16 bits: A=Alpha B=Blue G=Green R=Red

```

(Continued on the next page)

```
44  SpriteTest:
45      .word 0x800F,0x8000,0x83FF,0x83FF,0x83FF,0x83FF,0x8000,0x8000 ; 0
46      .word 0x8000,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x8000 ; 1
47      .word 0x83FF,0x83FF,0x801F,0x83FF,0x83FF,0x801F,0x83FF,0x83FF ; 2
48      .word 0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF ; 3
49      .word 0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF ; 4
50      .word 0x83FF,0x83FF,0xFFE0,0x83FF,0x83FF,0xFFE0,0x83FF,0x83FF ; 5
51      .word 0x8000,0x83FF,0x83FF,0xFFE0,0xFFE0,0x83FF,0x83FF,0x8000 ; 6
52      .word 0x8000,0x8000,0x83FF,0x83FF,0x83FF,0x83FF,0x8000,0x8000 ; 7
53
```

The program will show a Smiley on the screen.

How does it work?

Ironically, this is easier than the "Hello World" example!

Our VRAM base is at 0x06000000, we need to copy 2 bytes from our bitmap sprite to the screen for each pixel.

We load in our sprite address. (Line 18)

We then copy 16 bytes (8 pixels) from the sprite definition to the screen (one line of our sprite). (Lines 19-30)

We then move down a line of the screen, by adding 240*2 to the VRAM address. Each line is 240 pixels, and each pixel is 2 bytes. (Line 31)

We repeat this procedure until the image is done. (Line 32-33)

Example 3: Moving a sprite

This Example will show a Smiley using 15 bits per pixel color.

Compile this with VASM in the way described on the previous pages.

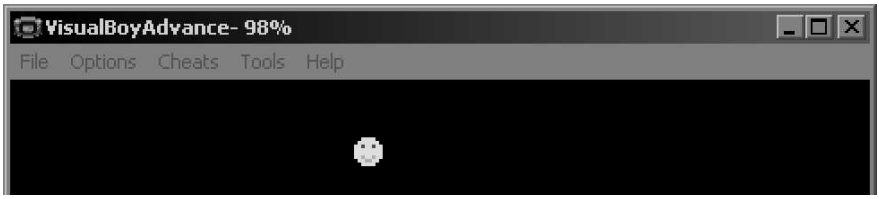
```
;Text after semicolon is comments, you don't need to type it in
    .org  0x08000000            ;GBA Rom Base
;Header
    b   ProgramStart            ;Jump to our program
    .space 178                  ;Logo (omitted) + Program name
    .byte 0x96                  ;Fixed value
    .space 49                   ;Dummy Header

ProgramStart:
    mov sp,#0x03000000          ;Init Stack Pointer

    mov r4,#0x04000000          ;DISPCNT -LCD Control
    mov r2,#0x403               ;4= Layer 2 on / 3= ScreenMode 3
    str r2,[r4]

    mov r8,#10                  ;Xpos
    mov r9,#10                  ;Ypos

    bl ShowSprite               ;Show the sprite starting position

InfLoop:
    mov r3,#0x4000130           ;Read GBA joypad
    ldrh r0,[r3]
            ;------lrDULRSsBA
    and r0,r0,#0b0000000011110000
    cmp r0,#0b0000000011110000
    beq InfLoop

    bl ShowSprite               ;Remove the old sprite
        ;------lrDULRSsBA
    tst r0,#0b0000000001000000
    bne JoyNotUp
    cmp.b r9,#0
    beq JoyNotUp
    sub.b r9,r9,#1              ;Move Up
JoyNotUp:
        ;------lrDULRSsBA
    tst r0,#0b0000000010000000
    bne JoyNotDown
    cmp.b r9,#19
    beq JoyNotDown
    add.b r9,r9,#1              ;Move Down
JoyNotDown:
```

(Continued on the next page)

```
44          ;------lrDULRSsBA
45      tst r0,#0b0000000000100000
46      bne JoyNotLeft
47      cmp.b r8,#0
48      beq JoyNotLeft
49      sub.b r8,r8,#1              ;Move Left
50  JoyNotLeft:
51          ;------lrDULRSsBA
52      tst r0,#0b0000000000010000
53      bne JoyNotRight
54      cmp.b r8,#29
55      beq JoyNotRight
56      add.b r8,r8,#1              ;Move Right
57  JoyNotRight:
58
59      bl ShowSprite               ;Show the new sprite position
60
61      mov r0,#0x8FFF              ;Delay Loop
62  Delay:
63      subs r0,r0,#1
64      bne Delay
65      b InfLoop                   ;Repeat.
66
67  ;Xor Sprite, drawing twice will remove sprite from screen.
68  ShowSprite:
69      mov r10,#0x06000000         ;VRAM base
70
71      mov r1,#16                  ;Sprite is 8px, 2 bytes per pixel
72      mul r2,r1,r8
73      add r10,r10,r2              ;Xpos *16
74
75      mov r1,#240*2*8             ;240 pixels per line, 2 bytes per pixel
76      mul r2,r1,r9
77      add r10,r10,r2              ;Ypos * 240*8*2
78
79      ldr r1,SpriteAddress        ;Sprite Address
80      mov r6,#8                   ;Height
81  Sprite_NextLine:
82      mov r5,#8                   ;Width (in words / pixels)
83
84      STMFD sp!,{r10}
85  Sprite_NextByte:
86          ldrH r3,[r1],#2         ;Must write 16/32bit per VRAM write
87          ldrH r2,[r10]
88          eor r3,r3,r2            ;Eor Word from screen
89          strH r3,[r10],#2
90
91          subs r5,r5,#1           ;X Loop
92          bne Sprite_NextByte
93      LDMFD sp!,{r10}
94      add r10,r10,#240*2          ;240 - 2 bytes per pixel
95      subs r6,r6,#1
96      bne Sprite_NextLine         ;Y loop
97      mov pc,lr
```

```
 98
 99  SpriteAddress:
100      .long SpriteTest            ;Address of Sprite
101
102  SpriteTest:                     ;Smiley Sprite
103      .word 0x8000,0x8000,0x83FF,0x83FF,0x83FF,0x8000,0x8000 ;  0
104      .word 0x8000,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x8000 ;  1
105      .word 0x83FF,0x83FF,0x801F,0x83FF,0x83FF,0x801F,0x83FF,0x83FF ;  2
106      .word 0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF ;  3
107      .word 0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF,0x83FF ;  4
108      .word 0x83FF,0x83FF,0xFFE0,0x83FF,0x83FF,0xFFE0,0x83FF,0x83FF ;  5
109      .word 0x8000,0x83FF,0x83FF,0xFFE0,0xFFE0,0x83FF,0x83FF,0x8000 ;  6
110      .word 0x8000,0x8000,0x83FF,0x83FF,0x83FF,0x83FF,0x8000,0x8000 ;  7
111
```

The program will show a Smiley on the screen, we can move the smiley with the joypad directions.



How does it work?

We've modified our sprite routine. It now EORs the bitmap data with the screen. This means if we draw the sprite in the same position twice it will be removed. (Lines 80-97)

We're now calculating the screen (X, Y) position using registers (R8,R9).

Our sprite is 8 pixels wide, and each pixel is 2 bytes, so we multiply the Xpos by 16.
Our sprite is 8 pixels tall. Each line is 240 pixels wide and each pixel is 2 bytes, so we multiply the Ypos by 240*2*8. We add this to the screen base 0x06000000. (Lines 68-77)

We read memory mapped port 0x4000130 to get the joypad directions. This returns a word in the format "%------lrDULRSsBA". (Line 22)

We wait until a key is pressed, then we remove the old sprite, by drawing it again in the same position. As it's EORed (XOR) this removes the old sprite. (Line 26-29)

We test the directions "DULR" (Down, Up, Left, Right). In each case we see if the direction is pressed, and test if we're already at the screen edge. If not we move in that direction. (Line 30-57)

We show the new location of the sprite. (Line 59)

We then have a delay loop, then repeat. (Line 61-65)

Learning ARM: Where to go from here

The documentation examples we've covered here should be enough to get you started with your first ARM program.

Over the following pages of this chapter, you'll be presented with all the ARM instructions, and a description of their purpose, with examples.

If you're looking for more, there's lots more content on the ChibiAliens website!

If you're looking for more information on the ARM instruction set, there are step by step tutorials, with videos and source code on the ARM page:
https://www.chibialiens.com/arm/

We've only covered the Gameboy Advance here, for details of the other systems covered, and detailed tutorials on the hardware of all the systems, take a look at the full list here:
https://www.assemblytutorial.com/

Don't forget to download the source code for this book from:
www.chibiakumas.com/book

And please subscribe to the official You tube channel for weekly new videos:
https://www.youtube.com/chibiakumas

ARM Instruction Set

Here we'll be discussing the basic ARM instruction set. We're covering up to ARM4, which should be sufficient for programming systems like the GBA.

ADC*ccS* Rn, Rm, *Op2*

Add With Carry. **Op2** plus the carry is added to **Rm** and the result is stored in **Rn**.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Usage Example: ADCEQS R0,R0,R1 ADC R0,R0,R1,ASL #4 ADC R0,R0,#4

ADD*ccS* Rn, Rm, *Op2*

Add **Op2** to **Rm** and store the result in **Rn**.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Usage Example: ADDEQS R0,R0,R1 ADD R0,R0,R1,ASL #4 ADD R0,R0,#4

AND*ccS* Rn, Rm, *Op2*

Logically AND **Op2** with **Rm** and store the result in **Rn**.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Logical AND will set bits in the destination, according to the source and destination. Where a bit in the source and destination are 1 the result will be 1, when they are not it will be 0.
For Example: If R1=0b10101010, "AND R1,R1,#0b11110000" will result in R1=0b10100000.

Usage Example: ANDEQS R0,R0,R1 AND R0,R0,R1,ASL #4 AND R0,R0,#4

B*cc* Label

Branch to a relative **Label**. This is like JP on many other CPUs.

Usage Example: BEQ ConditionalJump B TestJump

BIC*ccS* Rn, Rm, *Op2*

Logically Bit Clear **Op2** with **Rm** and store the result in **Rn**.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Logical BIC will clear bits in the destination, according to the source and destination.

Where a bit in the source is 1, the bit in the result will be 0, when the bit in the source is 0, the bit in the result will be the same as the destination.
For Example: If R1=0b10101010, "BIC R1,R1,#0b11110000" will result in R1=0b00001010

Usage Example: BICEQS R0,R0,R1 BIC R0,R0,R1,ASL #4 BIC R0,R0,#4

BL*cc* Label

Branch and Link to a relative subroutine **Label**. The Link Register, Register R14 (LR), is set to the return address. This has a similar purpose to JSR or CALL on many other CPUs.

To return from the subroutine, we need to transfer the Link Register back into the Program Counter. We can do this while restoring our other registers from the stack, or on its own.

As using BL from within a subroutine would alter the Link Register, we will need to back it up somehow, if we plan to nest subroutine calls.

```
TestSub1:   ;Simple Sub
        ;Your code goes here
    MOV pc,lr                   ;Return to calling sub

TestSub2:   ;Nestable Sub
    STMFD sp!,{r0-r12, lr}      ;Push Regs
        ;Your code goes here
    LDMFD sp!,{r0-r12, pc}      ;Pop Regs and return
```

Figure 62: Using the Link Register (LR) for returning from a Subroutine.

Usage Example: BLEQ ConditionalSub BL TestSub

CMN*cc* Rn, *Op2*

Compare Negative **Rn** to **Op2**. This sets the flags, the same as "ADDS Rn,Op2" would, but **Rn** is unchanged. This is like an IF statement in Basic.

Note: While you can specify the "S" suffix with "CMNS" you do not need to, as it is implied that the flags will change, as it's the only purpose of the command.

Usage Example: CMNEQ R0,R1 CMN R0,R1,ASL #4 CMN R0,#4

CMP*cc* Rn, *Op2*

Compare **Rn** to **Op2**. This sets the flags, the same as "SUBS Rn,Op2" would, but **Rn** is unchanged. This is like an IF statement in Basic.

Note: While you can specify the "S" suffix with "CMPS" you do not need to, as it is implied that the flags will change, as it's the only purpose of the command.

Usage Example: CMPEQ R0,R1 CMP R0,R1,ASL #4 CMP R0,#4

EOR*ccS* Rn, Rm, *Op2*

Logically Exclusive OR **Op2** with **Rm** and store the result in **Rn**.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Logical EOR will clear bits in the destination, according to the source and destination. Where a bit in **Op2** is 1 the bit in **Rm** will be flipped. Where a bit in **Op2** is 0 the bit in **Rm** will be unchanged.
For Example: If R1=0b10101010, "EOR R1,R1,#0b11110000" will result in R1= 0b01011010.

Exclusive OR is known as XOR on some other CPUs.

Usage Example: EOREQS R0,R0,R1 EOR R0,R0,R1,ASL #4 EOR R0,R0,#4

LDM*cc*adm Rn!, {Regs}

Transfer range of registers **{Regs}** to the address in **Rn**. This command can be used like a POP command with the stack pointer.
Rn contains the address for the registers to be stored. If we specify **!** the contents of the **Rn** register will be updated after the write.

The **{Regs}** list can contain ranges of registers specified with a minus "-".
For Example: R1-R3
Individual registers, or multiple ranges can be specified with a comma ",".
For Example: R5,R7,R1-R3

These commands can be used for quick reading. They can also be used as POP commands, when used with register SP! for use with the stack.

There are two sets of terms which can be used to refer to the addressing modes. One set makes it clear what happens to the address, the other makes it clear what happens to the data for stack use.

There are 4 possible addressing modes for '**adm**':

Abbreviation	Meaning
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

These have alternate codes, which are easier to use with the stack:

Stack Type	Push	Pop
Full Descending (normal stack)	STMFD (DB)	LDMFD (IA)
Full Ascending	STMFA (IB)	LDMFA (DA)
Empty Descending	STMED (DA)	LDMED (IB)
Empty Ascending	STMED (IA)	LDMED (DB)

Usage Example: LDMFD sp!,{r0-r12,pc} LDMFD sp!,{r0} LDMFD sp!,{r0,r1,r2}

LDR*cc* Rn, *Flex*
LDR*cc*B Rn, *Flex*

Load register **Rn** from address **Flex**, either a full 32 bit word can be loaded (LDRcc), or an 8 bit byte (LDRccB).

Flex is a base register with an offset of -4095 to +4095, or a base register with offset register and shift. If we use a label, the Assembler will use the program counter and calculate the offset.

Note: On the ARM, unlike the 68000, if we load a byte, the upper unused bits 8-31 are cleared.

Usage Example: LDR R0,NearLabel LDRB r0,[r1,#128] LDR r0,[r1,r2,asl #2]

LDR*cc*H Rn, *Off*
*LDRcc*SH Rn*, Off*
*LDRcc*SB Rn, *Off*

With ARM 4T+ HalfWord (16 bit), Signed Word (16 Bit) and Signed Byte (8 Bit) load commands were added. They use a more limited addressing mode, and do not use the Flex option.
Rn is loaded from address **Off**. **Off** is either a base register with an offset of -255 to +255 or a base register with an offset register.

Note: On the ARM, unlike the 68000, if we load a byte or word, the upper unused bits 8-31/16-31 are cleared.

Arm Revision: 4T+

Usage Example: LDRB R0,[R1,#128] LDRSB R0,[R1,#-255]
LDRH R0,SPRITE_NEXTLINE

MLA*ccS* Rn, Rm, Ro, Rp

32 bit Multiplication and Add. The formula used is effectively **Rn**=(**Rm***Ro**)+ **Rp**
Rn and **Rm** cannot be the same register.

If the multiplication results in a value that does not fit in the 32 bit register, the bottom 32 bits will be kept, and the 'pushed out' bits will be lost.

Usage Example: MLAEQS R0,R1,R0,R2 MLA R0,R1,R0,R2 MLA R0,R1,R2,R3

MOV*ccS* Rn, *Op2*

Move value in **Op2** into **Rn**.
This can be used to set a register to an immediate value, or to move one register to another.

Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.
MVN (Move Not) does give us more options.

Though our assembler may solve the problem for us automatically, to set a full 32 bit register manually, multiple commands must be used.
For Example: To set R1 to 0x81818181:

```
MOV  R0,#0x00000081
ORR R0,R0,#0x00008100
ORR R0,R0,#0x00810000
ORR R0,R0,#0x81000000
```

Usage Example: MOV R0,#0xFF MOV R0,#0xFF00 MOV R0,R1
MOV R0,R1,ASL #1

MRS*cc* Rn,sr

Move **sr** (either CPSR or SPSR) to register **Rn**.

You'll probably never need this command, and its use could cause crashes.

This command does not exist on ARM2. On ARM2 the flags were in the top 6 bits of the program counter, so the flags could be backed up by "MOV R0, R15", but you must not use this command on ARM3+.

Arm Revision: 3+

Usage Example: MRS R0,CPSR MRS R0,SPSR

Valid values for 'sr': CPSR SPSR

MSR*cc* sr_f,#
MSR*cc* sr_f,Rn

Move immediate **#** or register **Rn** into flags **f** of **sr** (either CPSR or SPSR).
You'll probably never need this command, and its use could cause crashes.

This command does not exist on ARM2. ARM2 had another command "TEQP R0, #0", but you must not use this command on ARM3+.

Arm Revision: 3+

Usage Example: MSR SPSR_CSXF,#0 MSR CPSR_F,#0

Valid values for 'sr': CPSR SPSR

Valid values for 'f': C X S F (or any combination)

MUL*ccS* Rn, Rm, Ro

32 bit Multiplication. The formula used is effectively **Rn**=**Rm***Ro**.
Rn and **Rm** cannot be the same register.

If the multiplication results in a value that does not fit in the 32 bit register, the bottom 32 bits will be kept, and the 'pushed out' bits will be lost.

Usage Example: MULEQS R0,R1,R0 MUL R0,R1,R0 MUL R0,R1,R2

MVN*ccS* Rn, *Op2*

Move Not. Flip all the bits of **Op2** and move result into **Rn**.
If Op2=0x00000001 then **Rn** will be set to 0xFFFFFFFE. In some cases this allows us to get around the limited capacity of MOV and set the entire register with a single command.

This can be used to set a register to an immediate value, or to move one register to another.

Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Usage Example: MVN R0,#0xFF MVN R0,#0xFF00 MVN R0,R1
MVN R0,R1,ASL #1

ORR*ccS* Rn, Rm, *Op2*

Logically OR **Op2** with **Rm** and store the result in **Rn**.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Logical OR will set bits in the destination according to the source and destination. Where a bit in the source or destination is 1 the result will be 1, when they are both 0, the result will be 0.
For Example: If R1=0b10101010, "ORR R1,R1,#0b11110000" will result in R1=0b11111010

Usage Example: ORREQS R0,R0,R1 ORR R0,R0,R1,ASL #4 ORR R0,R0,#4

RSB*cc*S Rn, Rm, *Op2*

Reverse Subtract. This performs the calculation **Rn=Op2-Rm**.

Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

As **Op2** can perform shifts, this gives different possible options compared to SUB.

Usage Example: RSBEQS R0,R0,R1 RSB R0,R0,R1,ASL #4 RSB R0,R0,#6

RSC*cc*S Rn, Rm, *Op2*

Reverse Subtract with Carry. This performs the calculation **Rn=**(**Op2**-**Rm**)-C .

Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

As **Op2** can perform shifts, this gives different possible options compared to SUB.

Usage Example: RSCEQS R0,R0,R1 RSC R0,R0,R1,ASL #4 RSC R0,R0,#6

SBC*cc*S Rn, Rm, *Op2*

Subtract with Carry. This performs the calculation **Rn=**(**Rm**-**Op2**)-C.

Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

As **Op2** can perform shifts, this gives different possible options compared to SUB.

Usage Example: SBCEQS R0,R0,R1 SBC R0,R0,R1,ASL #4 SBC R0,R0,#6

STM*cc*adm Rn!, {Regs}

Transfer range of registers **{Regs}** to the address in **Rn**. This command can be used like a PUSH command with the stack pointer.

Rn contains the address for the registers to be stored. If we specify **!** the contents of the **Rn** register will be updated after the write.

The **{Regs}** list can contain ranges of registers specified with a minus "-".
For Example: R1-R3
Individual registers, or multiple ranges can be specified with a comma ",".
For Example: R5,R7,R1-R3

These commands can be used for quick writing. They can also be used as a PUSH command when used with register SP! for use with the stack.

There are two sets of terms which can be used to refer to the addressing modes, one set makes it clear what happens to the address, the other makes it clear what happens to the data for stack use.

There are 4 possible addressing modes for '**adm**':

Abbreviation	Meaning
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

These have alternate codes, which are easier to use with the stack:

Stack Type	Push	Pop
Full Descending (normal stack)	STMFD (DB)	LDMFD (IA)
Full Ascending	STMFA (IB)	LDMFA (DA)
Empty Descending	STMED (DA)	LDMED (IB)
Empty Ascending	STMED (IA)	LDMED (DB)

Usage Example: STMFD sp!,{r0-r12, lr} STMFD sp!,{r0} STMFD sp!,{r0,r1,r2}

STR*cc* Rn, *Flex*
*STRcc*B Rn, *Flex*

Store register **Rn** to address **Flex**, either a full 32 bit word can be loaded (**STRcc**), or an 8 bit byte (**STRccB**).

Flex is a base register with an offset of -4095 to +4095, or a base register with offset register and shift. If we use a label, the Assembler will use the program counter and calculate the offset.

Usage Example: STR R0,NearLabel STRB r0,[r1,#128] STR r0,[r1,r2,asl #2]

STR*cc*H Rn, *Off*
*STRcc*SH Rn, *Off*
*STRcc*SB Rn, *Off*

With ARM 4T+ Half Word (16 bit), Signed half Word (16 Bit) and Signed Byte (8 Bit) store commands were added. They use a more limited addressing mode, and do not use the Flex option.

Off is either a base register with an offset of -255 to +255, or a base register with an offset register.

Note: On the ARM, unlike the 68000, if we load a byte or word, the upper unused bits 8-31/16-31 are cleared.

Arm Revision: 4T+

Usage Example: STRB R0,[R1,#128] STRSB R0,[R1,#-255]
STRH R0,SPRITE_NEXTLINE

SUB*cc*S Rn, Rm, *Op2*

Subtract. This performs the calculation **Rn=Rm-Op2**.

Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

As **Op2** can perform shifts, this gives different possible options compared to SUB.

Usage Example: SUBEQS R0,R0,R1 SUB R0,R0,R1,ASL #4 SUB R0,R0,#6

SWI*cc*

Software Interrupt. This calls a system interrupt, passing 24 bit immediate **#** to the SWI exception handler.

This is often used to call operating system functions on RISC OS. "SWI 0x00", for example, will call the Print Character routine, showing the ASCII character in R0.

Usage Example: SWIEQ 0x1 SWI 3

SWP*ccB* Rn, Rm, [Ro]

Swap a register and memory. **Rn** and **Rm** can be the same for a switch between a register and an address.
The effective formulas used are **Rn**=[**Ro**], [**Ro**]=**Rm**.

Arm Revision: 3+

Usage Example: SWP R0,R1,[R2] SWP R0,R0,[R1] SWPB R0,R1,[R2]

TEQ*cc* Rn, Rm, *Op2*

Test for bitwise Equality.
Set the flags in the same way "EOR **Rn**,**Rm**,**Op2**" would, but leave **Rn** unchanged.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Usage Example: TEQEQ R0,R0,R1 TEQ R0,R0,R1,ASL #4 TEQ R0,R0,#6

TST*cc* Rn, Rm, *Op2*

Test bits.
Set the flags in the same way "AND **Rn**,**Rm**,**Op2**" would, but leave **Rn** unchanged.
Op2 can be a register with a shift, or an 8 bit immediate which can be rotated by an even number of bits.

Usage Example: TSTEQ R0,R0,R1 TST R0,R0,R1,ASL #4 TST R0,R0,#6

References, Attribution and Thanks.

Many thanks to my patreon backers, without whose support I would be unable to justify continuing to learn these classic machines:

aberrant_hacker, Ack, Adolfo Perez Alvarez, Alejandro Pérez, Alejandro Gil Cal, AYYamoZ80, Barry White, Bytrix, Chris Lidyard, David L. Martin, Dave Snowdon, DUO, Dimitris Topouzis, Emad, Erik Haliewicz, Ervin Pajor, Fábio Domingos, FNQMatt, gee-k.net, Helmut H, Indy S, James, James Ford, James Whitwell, Julien Nadeau Carriere, Juergen Pichler, Justin, Laurens Holst, Leo Comerford, lx.gnzlz, MacTORG Steen, Marcio Maia, Marco Leal, Mark Trombly, Markus Podszuk (Brainslave), milkrobotics, Neil Moore, Nicole Express, Oli Norwell, ordigdug, pagetable.com, Paul Barrick, Paul Humphreys, Rob Bishop (Collectar), Remax, Robin Elvin, Robsoft, Roger, Roland Rząsa, Samuel Becker, Samuli Tuomola, ShootTheCore, smarty, Teopl, Themistocles, Trevor Briscoe, Voyager_sput, Zack, Zuka, Zulkarnain Adnan

This book would have not been possible without the following websites and technical resources:

https://archive.org/
https://www.wikipedia.org/

Z80 Resources:
https://www.zilog.com/manage_directlink.php?filepath=docs/z80/um0080
http://www.ticalc.org/archives/files/fileinfo/268/26877.html

Assemblers:
http://sun.hasenbraten.de/vasm/

Special thanks to:

VASM author Frank Wille, not just for his work on the assembler, but for making additions to support the PC-Engine and x68000 Xfiles.

References, Attribution and Thanks.

Appendix

ASCII Character Codes

Ctrl	Dec	Hex	Oct	Binary	Asc
NULL	0	00	000	00000000	
SOH	1	01	001	00000001	
STX	2	02	002	00000010	
ETX	3	03	003	00000011	
EOT	4	04	004	00000100	
ENQ	5	05	005	00000101	
ACK	6	06	006	00000110	
BEL	7	07	007	00000111	
BS	8	08	010	00001000	
TAB	9	09	011	00001001	
LF	10	0A	012	00001010	
VT	11	0B	013	00001011	
FF	12	0C	014	00001100	
CR	13	0D	015	00001101	
SO	14	0E	016	00001110	
SI	15	0F	017	00001111	
DLE	16	10	020	00010000	
DC1	17	11	021	00010001	
DC2	18	12	022	00010010	
DC3	19	13	023	00010011	
DC4	20	14	024	00010100	
NAK	21	15	025	00010101	
SYN	22	16	026	00010110	
ETB	23	17	027	00010111	
CAN	24	18	030	00011000	
EN	25	19	031	00011001	
SUB	26	1A	032	00011010	
ESC	27	1B	033	00011011	
DS	28	1C	034	00011100	
GS	29	1D	035	00011101	
RS	30	1E	036	00011110	
US	31	1F	037	00011111	

Dec	Hex	Oct	Binary	Asc
32	20	040	00100000	
33	21	041	00100001	!
34	22	042	00100010	"
35	23	043	00100011	#
36	24	044	00100100	$
37	25	045	00100101	%
38	26	046	00100110	&
39	27	047	00100111	'
40	28	050	00101000	(
41	29	051	00101001	)
42	2A	052	00101010	*
43	2B	053	00101011	+
44	2C	054	00101100	,
45	2D	055	00101101	-
46	2E	056	00101110	.
47	2F	057	00101111	/
48	30	060	00110000	0
49	31	061	00110001	1
50	32	062	00110010	2
51	33	063	00110011	3
52	34	064	00110100	4
53	35	065	00110101	5
54	36	066	00110110	6
55	37	067	00110111	7
56	38	070	00111000	8
57	39	071	00111001	9
58	3A	072	00111010	:
59	3B	073	00111011	;
60	3C	074	00111100	<
61	3D	075	00111101	=
62	3E	076	00111110	>
63	3F	077	00111111	?

Dec	Hex	Oct	Binary	Asc
64	40	100	01000000	@
65	41	101	01000001	A
66	42	102	01000010	B
67	43	103	01000011	C
68	44	104	01000100	D
69	45	105	01000101	E
70	46	106	01000110	F
71	47	107	01000111	G
72	48	110	01001000	H
73	49	111	01001001	I
74	4A	112	01001010	J
75	4B	113	01001011	K
76	4C	114	01001100	L
77	4D	115	01001101	M
78	4E	116	01001110	N
79	4F	117	01001111	O
80	50	120	01010000	P
81	51	121	01010001	Q
82	52	122	01010010	R
83	53	123	01010011	S
84	54	124	01010100	T
85	55	125	01010101	U
86	56	126	01010110	V
87	57	127	01010111	W
88	58	130	01011000	X
89	59	131	01011001	Y
90	5A	132	01011010	Z
91	5B	133	01011011	[
92	5C	134	01011100	\
93	5D	135	01011101	]
94	5E	136	01011110	^
95	5F	137	01011111	_

Dec	Hex	Oct	Binary	Asc
96	60	140	01100000	`
97	61	141	01100001	a
98	62	142	01100010	b
99	63	143	01100011	c
100	64	144	01100100	d
101	65	145	01100101	e
102	66	146	01100110	f
103	67	147	01100111	g
104	68	150	01101000	h
105	69	151	01101001	i
106	6A	152	01101010	j
107	6B	153	01101011	k
108	6C	154	01101100	l
109	6D	155	01101101	m
110	6E	156	01101110	n
111	6F	157	01101111	o
112	70	160	01110000	p
113	71	161	01110001	q
114	72	162	01110010	r
115	73	163	01110011	s
116	74	164	01110100	t
117	75	165	01110101	u
118	76	166	01110110	v
119	77	167	01110111	w
120	78	170	01111000	x
121	79	171	01111001	y
122	7A	172	01111010	z
123	7B	173	01111011	{
124	7C	174	01111100	|
125	7D	175	01111101	}
126	7E	176	01111110	~
127	7F	177	01111111	⌂

Dec	Hex	Oct	Binary	Asc
128	80	200	10000000	Ç
129	81	201	10000001	ü
130	82	202	10000010	é
131	83	203	10000011	â
132	84	204	10000100	ä
133	85	205	10000101	à
134	86	206	10000110	å
135	87	207	10000111	ç
136	88	210	10001000	ê
137	89	211	10001001	ë
138	8A	212	10001010	è
139	8B	213	10001011	ï
140	8C	214	10001100	î
141	8D	215	10001101	ì
142	8E	216	10001110	Ä
143	8F	217	10001111	Å
144	90	220	10010000	É
145	91	221	10010001	æ
146	92	222	10010010	Æ
147	93	223	10010011	ô
148	94	224	10010100	ö
149	95	225	10010101	ò
150	96	226	10010110	û
151	97	227	10010111	ù
152	98	230	10011000	ÿ
153	99	231	10011001	Ö
154	9A	232	10011010	Ü
155	9B	233	10011011	¢
156	9C	234	10011100	£
157	9D	235	10011101	¥
158	9E	236	10011110	₧
159	9F	237	10011111	ƒ

Dec	Hex	Oct	Binary	Asc
160	A0	240	10100000	á
161	A1	241	10100001	í
162	A2	242	10100010	ó
163	A3	243	10100011	ú
164	A4	244	10100100	ñ
165	A5	245	10100101	Ñ
166	A6	246	10100110	ª
167	A7	247	10100111	º
168	A8	250	10101000	¿
169	A9	251	10101001	⌐
170	AA	252	10101010	¬
171	AB	253	10101011	½
172	AC	254	10101100	¼
173	AD	255	10101101	¡
174	AE	256	10101110	«
175	AF	257	10101111	»
176	B0	260	10110000	░
177	B1	261	10110001	▒
178	B2	262	10110010	▓
179	B3	263	10110011	│
180	B4	264	10110100	┤
181	B5	265	10110101	╡
182	B6	266	10110110	╢
183	B7	267	10110111	╖
184	B8	270	10111000	╕
185	B9	271	10111001	╣
186	BA	272	10111010	║
187	BB	273	10111011	╗
188	BC	274	10111100	╝
189	BD	275	10111101	╜
190	BE	276	10111110	╛
191	BF	277	10111111	┐

Dec	Hex	Oct	Binary	Asc
192	C0	300	11000000	└
193	C1	301	11000001	┴
194	C2	302	11000010	┬
195	C3	303	11000011	├
196	C4	304	11000100	─
197	C5	305	11000101	┼
198	C6	306	11000110	╞
199	C7	307	11000111	╟
200	C8	310	11001000	╚
201	C9	311	11001001	╔
202	CA	312	11001010	╩
203	CB	313	11001011	╦
204	CC	314	11001100	╠
205	CD	315	11001101	═
206	CE	316	11001110	╬
207	CF	317	11001111	╧
208	D0	320	11010000	╨
209	D1	321	11010001	╤
210	D2	322	11010010	╥
211	D3	323	11010011	╙
212	D4	324	11010100	╘
213	D5	325	11010101	╒
214	D6	326	11010110	╓
215	D7	327	11010111	╫
216	D8	330	11011000	╪
217	D9	331	11011001	┘
218	DA	332	11011010	┌
219	DB	333	11011011	█
220	DC	334	11011100	▄
221	DD	335	11011101	▌
222	DE	336	11011110	▐
223	DF	337	11011111	▀

Dec	Hex	Oct	Binary	Asc
224	E0	340	11100000	α
225	E1	341	11100001	ß
226	E2	342	11100010	Γ
227	E3	343	11100011	π
228	E4	344	11100100	Σ
229	E5	345	11100101	σ
230	E6	346	11100110	µ
231	E7	347	11100111	τ
232	E8	350	11101000	Φ
233	E9	351	11101001	Θ
234	EA	352	11101010	Ω
235	EB	353	11101011	δ
236	EC	354	11101100	∞
237	ED	355	11101101	φ
238	EE	356	11101110	ε
239	EF	357	11101111	∩
240	F0	360	11110000	≡
241	F1	361	11110001	±
242	F2	362	11110010	≥
243	F3	363	11110011	≤
244	F4	364	11110100	⌠
245	F5	365	11110101	⌡
246	F6	366	11110110	÷
247	F7	367	11110111	≈
248	F8	370	11111000	°
249	F9	371	11111001	∙
250	FA	372	11111010	·
251	FB	373	11111011	√
252	FC	374	11111100	n
253	FD	375	11111101	²
254	FE	376	11111110	■
255	FF	377	11111111	

Instruction Set Index by processor

Arm Instruction Set Index

6502 Instruction Set Index

68000 Instruction Set Index

8086 Instruction Set Index

Z80 Instruction Set Index

Alphabetical Index