SD-8516 Assembly Language

Here you can learn all about writing Assembly Language programs for the SD-8516.

You may also be interested in: Part II: Writing Games in Assembly Language.

Introduction

Welcome to a quick overview of SD-8516 Assembly Language! If you are new to programming, it is suggested that you first read the SD-8516 User's Guide as it contains a complete introduction to your computer and contains a chapter on programming in BASIC. Armed with that knowledge, you will be able to learn assembly language at least.... three times faster!

You can also refer to the SD-8516 Programmer's Reference Guide which contains a more detailed introduction to Assembly Language programming. This is more of a program outline for self-study – a "second opinion", if you want one!

We also recommend the book Usborne Introduction to Machine Code for Beginners (for the Z80 and 6502) available at https://archive.org/details/machine-code-for-beginners

What is your decision?

- If you choose to SD-8516 User's Guide turn to page 10.
- If you choose to continue, roll 2d6 and read the next section entitled "How to Enter and Run Assembly Language Programs."

How to Enter and Run Assembly Language Programs

There are several ways. One is to use an external editor and load it into `ed` using the `autotyper` (or just type it using `ed`). Then you can do something like

```
as filename.asm progname
```

But for quick projects you can also use the ASSEMBLE command from Stellar BASIC V1.0:

```
10 ASSEMBLE
```

This will automatically assemble a program listed in BASIC line-numbered format when it is included as the first line of a program. If it appears elsewhere in the program it will cause a ?SYNTAX ERROR INVALID ASSEMBLE.

Example code:

```
10 ASSEMBLE
20 LDBLX @msg
```

```
30 LDAH $66
40 INT $05
50 LDAH $64
60 INT $05
70 RET
80 msg:
90 .bytes "HELLO WORLD!", 0
```

After entering the above program you can SAVE it and then later LOAD it. To run it, type RUN and then SYS.

The second way is using MON. Using MON you can either cut and paste a machine language program, or enter one yourself. You can also load machine language programs in binary or published format. Although we can't show you a binary format file in TEXT, we can show you what a published machine language program looks like:

```
$C000: 00 34 10 C0 00 00 20 66 :8A
$C008: 86 05 00 20 64 86 05 85 :1F
$C010: 48 45 4C 4C 4F 20 57 4F :3A
$C018: 52 4C 44 21 00 00 00 00 :03
```

The above two listings represent the same program. The first one is more human-readable, the second is for compact publication. You may omit the leading $ and the trailing :byte checksum for brevity. You can also omit spaces if you like:

```
C000:003410C000002066
C008:8605002064860585
C010:48454C4C4F20574F
C018:524C4421
```

This works too:

```
C000:003410C000002066860500206486058548454C4C4F20574F524C4421
```

You can SAVE machine language programs like the above by using HEXMON to save a range of bytes:

```
C000.C020S
```

You can LOAD them by typing L in HEXMON.

Lesson 1 : Memory, Registers and Flags

- Lesson 1: “Memory, Registers and Flags”

- Time: 5 min
- Learn:
    - Registers: A B X Y
    - Flags: N Z C V
    - Memory: Flat memory model

In general there are some things you should know and consider first when learning SD-8516 assembly language programming.

- Fist, we use a flat memory model, which goes from $0 to $03FFFF. That's 4 banks of 64k.
- Second, word registers are 16 bits. To access them as high byte or low byte use H and L. For example X is 16 bits, but XH is the high byte, and XL is the low byte. Similarly for every register, such as Y, YL is the low byte and YH is the high byte. For A, AH and AL, and so on.
- Third, memory requires three bytes to address, so you can only address bank 0 if you use a 16 bit register. To address upper memory, include a bank byte. Like this: BLX, ELM, or GLD, etc. This means BL+X (BL is the bank byte and X is the address within that bank). Or if you use ELM it EL + M, and so on. Always remember that these are not independent registers, but if you modify EL or M it will modify ELM, and if you modify ELM it will modify EL and M as well.

Finally, the concept of flags. Flags are one bit status registers. Some operations modify flags. For example, if you load a zero, the zero flag will be set. if you ADD two numbers which don't fit in a word, the overflow or carry flags might be set. The flags will be explained more in detail in the lessons teaching the individual instructions and how they manage flags. For now know that there are four primary flags that are set via operations:

- N – Negative. If an operation produces a number that *looks like* a signed negative number, this flag will be set. Most of the time you can just ignore this.
- C – Carry flag. If you add two numbers and it doesn't fit in a word, the carry will be set and then added in on the next ADD (on an ADD-with carry operation). This makes it easy to chain additions of very large numbers by keeping track of the “carry the one” for you.
- V – Overflow flag. In general if there is more overflow than in carry, this is set
- Z – Zero flag. If an operation produces or sees a zero, this is set.

Why flags? During decision making, you can use flags to control the program flow. This will be discussed in lesson 5: Program flow.

For today, let's do a deep-dive on the registers, because there's a little more to them than merely being 16 bit words.

All About Registers

There are sixteen general purpose registers available for use> Here they are, with a short comment on name and purpose. Of course, since they're general purpose, there is nothing separating one register from another except convention. You can feel free to use this guide, or use them any way you like.

REG	Name	Convention	Notes
A	Accumulator	Scratchpad for math operations, function calls, etc.	The accumulator – used in much the same way as A or AX on 6502/8086 style systems.
B	Assistant to the Accumulator	Secondary accumulator	This will often hold the results of functions called with A as a variable.
X	Column Index Register	Intended to help map 2d memory and arrays, loops, etc.	Often used for example in cursor or pixel array helper functions
Y	Row Index Register	Intended to act as a row or record indicator alongside X.	Row index register.
C, I, J, K	Iterator Registers	C is often used for counting, but I, J, K are also used. Also see: CD, IJ and KT. Some people treat these (especially K, alongside T and KT) as temporary registers	
T	Temporary Register	There is a saying, if you are preserving T you're doing it wrong. Don't PUSH and POP T to protect it– use it locally and then ignore it. T is our favorite temporary register!	
M, D	Memory pointer and memory pointer Destination.	These are often used in pairings like ELM, ELD, etc, to point to memory locations. As such they are generally for immediate use only and could be used on their own as temporary registers.	ELM is EL as high-byte
E, F, G	Extra registers	Most often used as high bytes for 24-bit memory access ex. GLD, FLM, etc. but can also be used for general purpose registers.	
L	The Last True Register	If you really need another register, use this one. For emergency use only.	Rarely used.
Z	Z-index pointer	Often used as a third dimensional register for graphics or data processing.	Often used as a temporary register.

Byte Access

Each 16 bit register (such as A) may be accessed as the byte registers H and L. This means AH is the high byte of A, BH is the high byte of B, etc. while AL is the low byte of A, ZL is the low byte of Z, etc.

24-bit Register Pairing

The system uses register pairing, which simulates 24 and 32 bit registers, for certain limited operations. The allowed pairings are:

	C	D	K	M	X	Y	Z	A
B	BLC	BLD	BLK	BLM	BLX	BLY	BLZ	BLA
E	ELC	ELD	ELK	ELM	ELX	ELY	ELZ	ELA
F	FLC	FLD	FLK	FLM	FLX	FLY	FLZ	FLA

	C	D	K	M	X	Y	Z	A
G	GLC	GLD	GLK	GLM	GLX	GLY	GLZ	GLA
I	ILC	ILD	ILK	ILM	ILX	ILY	ILZ	ILA
J	JLC	JLD	JLK	JLM	JLX	JLY	JLZ	JLA
L	LLC	LLD	LLK	LLM	LLX	LLY	LLZ	LLA
T	TLC	TLD	TLK	TLM	TLX	TLY	TLZ	TLA

The suggested use case is to deal with pointers without having to construct them. Any opcode can deal with pointers directly; ADD, SUB, LD, ST, etc.

Register Collision

Warning! BLX uses BL and X. They are not separate! If you store something in BLX, writing to B, BL or X will clobber it. This is known as the 'clobbergoblin', the ancient enemy of programmers. Similarly, if you are using BL or X anywhere and use BLX for something else, it will over-write BL and X (and XH and XL as X).

The convention of BLX is that BL is the high-byte. ELM, is EL + M, etc.

Examples:

- Source pointer ELM, Destination pointer ELD; if it's in an alternate bank, use FLD so EL and FL don't collide.

32-bit register pairing

The system can simulate 32 bit operations by combining two registers together however be advised this is very slow as it requires the CPU to simulate operations across multiple registers. These otherwise operate like their 24 bit counterparts.

The only allowed pairs are: AB, CD, XY, IJ, TK, LZ, EF, GM.

WARNING: Modifying G or M will destroy GM, etc. as GM is directly made of G and M (same idea as with 24 bit pointers, or with 16 bit where A is comprised of AH and AL, etc.)

LDGM $12345678 is equivalent to LDG $5678 and LDM $1234. So AB for example uses B as the high-word. This is opposite the BLX convention which uses BL as the high-byte.

This is because during MUL operations overflow moves into the high byte, otherwise it stays in the original register. EX. MUL A, B moves into A but overflow goes into B.

BLA: A special register

Notice that BLA is special in that it aligns neatly with the 32 bit double-word AB (see below). If you load a four byte value into AB, you can immediately use it as a three byte pointer:

```
  LDAB #00030100  ; A = low word ($0100) B = high word ($0003), so, BL = $03
```

```
  STX [BLA]       ; BLA is then bank $03 + address $0100
```

The use case is clear; Star Forth uses this to convert 32 bit cells on the stack to pointers (and vice versa) with zero-cost. If you need two such pointers, you can do

```
  LDAB [@label]
  MOV ELM, BLA       ; Construct ELM
  LDAB [@label]      ; Construct BLA
```

In this sense, AB takes the role of 32 bit accumulator. There is one other register that operates this way: KT. The 32 bit KT decomposes into little-endian KL-KH-TL-TH in the same way that AB decomposes into AL-AH-BL-BH:

```
  LDAB [@label]      ; Construct pointer BLA
  LDKT [@label]      ; Construct pointer TLK directly
```

Only AB and TK operate in this way; no other register pair can convert in this manner. Also be aware that if you load BLA or TL and attempt to store it as a 32 bit value, the BH or TH values will remain whatever they were prior to loading BLA (or TLK). You may wish to zero them in this case – or use them as a fast way to save data (i.e. saving a PUSH or POP by pushing AB which saves the pointer BL+A and the byte BH at the same time).

Lesson 2 : Load-Store Architecture

- Lesson 2: “Load-Store Architecture”
- Time: 5 min
- Learn:
    - Registers: A and B
    - Opcodes: LDA, LDB, STA, STB.
    - Addressing modes: Immediate mode (numbers). Memory mode (memory reference).

Let's dive in to the basic idea behind SD-8516 assembly language programming! If you've ever programmed before, it's similar but different to a high level language. It is similar because there are functions and commands that take operands, and it is different because the functions are very simple building blocks, and there are only a limited number of integer variables that you can use.

The first concept is “Load-Store Architecture”. The SD-8516 uses a load-store architecture. This means that data is read from and written to registers, operated on inside registers, and then written back out to memory. There are no memory to memory operations and registers can only be loaded and saved.

The commands to load and store are LD and ST (load and store) followed by the register and an operand. For example,

```
    ; * LDA means "load into A,"
    ; * LDB means "load into B".
    LDA #56             ; Load the decimal number 56 into the variable A.
    LDA [#56]           ; Load the value in memory location #56 into the
```

```
variable A.
```

That's it for load operations. You can load a variable with a number either from memory or from a number directly. You cannot load a variable from another variable. This is invalid:

```
      LDA B              ; this doesn't work, but you can MOV A, B instead.
```

Next let's look at store operations. Store operations write the register to a memory location. You can't write to a register – that would violate the “load-store architecture”. But, you can use MOV to do that instead. MOV, as we will see, is for register-to-register moves.

Examples:

```
      STA [$1000]        ; Store the value in A at memory location $ (hex)
1000.
```

Hex 1000 in decimal is 4,096. You can use decimal numbers via the '#' prefix or hex numbers with the '$' prefix. If you want to use binary, use 0b00000001 (that's the number 1 in binary).

Now you know how to load and store information from memory to the variables!

Lesson 3: Operations

- Lesson 3: Operations
- Time: 5 min
- Learn:
    - Registers: C and D
    - Opcodes: ADD, SUB

Now, once you have access to information in the computer's memory, you need to be able to perform operations on that information. Some of the things you can do are: adding, subtracting, multiplying and dividing. Here are some examples of things you can do:

```
      ADD A, B           ; Adds A and B and stores the result in A.
      ADD B, C           ; Adds B and C and stores the result in B
```

As you can see, the ADD command has a source register and a destination register. The destination is first and the source is second. So ADD A, D means D will be added to A, and A will hold the result. All of the registers such as A, B, C, D can be used. However by convention we like to use A and B for simple math.

Anyways, you can also do these things:

```
      SUB A, B           ; Subtract A - B and store the result in A.
```

Lesson 4: Advanced Operations

- Lesson 4: Advanced Operations

- Time: 5 min
- Learn:
  - 32-bit Register Pairing
  - MUL and DIV

Some processors such as the venerable 6502 (6510, etc.) stop with ADD and SUB, but we have a more advanced 8516, so we can also MUL and DIV. However, MUL and DIV are special operations; observe:

```
      MUL A, B           ; multiply A and B and store the result in AB.
```

Storing the result in AB? What's that? The SD-8516 has a special 32 bit extended operation for multiplication. The result is stored in A, but if the result would not fit in a word, the extra information is in B and the overflow flag is set. For example. what is $FFFF times $FFFF? It obviously cannot fit in one word. However, the result ($FFFE0001) does fit into two words. So in this case, A would be $FFFE and B would be $0001. This kind of overflow allows muliplication of larger numbers. Before anyone says “Why not just check overflow”, it's because you can also multiply like this:

```
      MUL AB, CD         ; Multiply AB by CD and store in ABCD.
```

Now, there is no way to operate on a 64 bit number (ex. ABCD) however, the result will be stored there, for you to interpret. That's the power of the SD-8516, it can multiply quite nicely! If you wanted, you could extend 64 bit operations via software. It would be slow, but workable. “bigwords”?

```
      DIV A, B           ; Divide A by B and store the answer in A and the
remainder in B
```

The special properties of DIV allow you to perform modulus for free, or, in a modulus operation you can get the DIV for free. You can also do things like:

```
      DIV AB, CD         ; Divide AB by CD and store in AB and modulus
(remainder) in CD.
```

You can also divide 32 bit paired registers. The powerful MUL and DIV capabilities of the SD-8516 set it apart from other CPUs of the era.

Lesson 5: Flow Control (Branching)

- Lesson 5: Flow Control (Branching)
- Time: 10 min
- Learn: Assembler Labels, CMP, JZ, RET

Tying everything together, what do you think this program does?

```
      LDA [$00]
      LDB [$02]
      CMP A, B
      JZ @equal
```

```
  not_equal:
      LDC $01   ; error code #1
      RET
  equal:
      LDC $00   ; no error
      RET
```

The program loads the word (two bytes) at $00 ($00 and $01) into A, and the word at $02 ($02 and $03) into B. Then it compares them. If they are equal, the zero flag is set. Depending on this we set our return code, which here by convention is C. But it could be anything. We have thus demonstrated the ability to compare registers and make a decision on program contorl flow based on that comparison. This has applications everywhere, from making sure a cursor is within the limits of the screen, to testing if a character is uppercase or lowercase, and many, so many applications that we cannot list them here.

CMP is the fundamental flow control operation. Compare two registers and JZ if equal. Fall-through is the not-equal case. You could also use JNZ instead and fall-through the “is equal” case. Now you know how to control the flow of your programs!

How CMP affects flags

CMP works by doing a simple test:

```
      CMP A, B            ; We are doing A - B!
```

Yes that's right, it's doing A - B, but it isn't doing it to store the value in A. It's testing if the result is 0 or not. If the result is zero, it sets the zero flag; ZF = 1. If it's not equal, then it is either ABOVE or BELOW zero. Imagine CMP 5,5 versus CMP 5,10 versus CMP 10,5:

```
      CMP 5, 5            ;  5 -  5 =  0. Aha, a zero! ZF = 1
      CMP 5, 10           ;  5 - 10 = -5. No zero. ZF = 0
      CMP 10, 5           ; 10 -  5 =  5. No zero. ZF = 0
```

So because it's equal, it produces a zero. Seeing the zero, the CPU sets the zero flag. Then you can control program flow by JZ (jump-if-zero) and JNZ (jump-if-not-zero).

But there is more! As you see above, there are actually three situations that can occurr. It can be equal, or it can be less than zero, or above zero. You will notice that if A is less than B, the number is negative – or, “less than”. And, if the number in A is greater than B, then A-B produces a positive number, which is “greater than” zero. So it means A is greater than zero! This is why it's called CMP or “compare”. It compares if A is greater than, equal to, or less than B. And, we can test that by looking at the carry flag. The rule is, if you need to “borrow”, you do not set carry.

```
      CMP 5, 5            ;  5 -  5 =  0.  No borrow --> carry is set:     CF
= 1
      CMP 5, 10           ;  5 - 10 = -5. Yes borrow --> carry is NOT set: CF
= 0
      CMP 10, 5           ; 10 -  5 =  5.  No borrow --> carry is set:     CF
= 1
```

Therefore, if carry is set, we know that A is less than B.

But wait! There's more!

```
      CMP 5, 5           ;  5 -  5 =  0. Not negative. N flag not set.
      CMP 5, 10          ;  5 - 10 = -5. Yes negative. N flag set!
      CMP 10, 5          ; 10 -  5 =  5. Not negative. N flag NOT set!
```

So you can also use the N flag. So here is the situation:

- If ZF=1 then A and B are equal.
- If ZF = 0, then look at CF or NF
  - If CF is set, A is greater than B.
  - If NF is set, A is less than B.

There you go! You can do this now, to branch on each condition:

- JZ @A_equals_B
- JC @A_greater_than_B
- JN @A_less_than_B

This is the foundation of how an IF statement works, or the ternary operator in C.

Carry Flag: No Borrow Carry

Understanding the operation of the carry flag is important since it's part of branching code. The SD-8516 follows in the grand tradition of no borrow carry, which is how the 6502 does it, as well as many RISC and ARM designs – SPARC, PowerPC, and Apple Silicon! On the other hand, Intel 80×86 uses the opposite convention.

Here's how to understand it:

- A >= B == C

1. CMP A, B means we do A-B.
2. Then we apply the rules; NO BORROW = CARRY SET

This is often called “No borrow carry”. or “no carry borrow”. Here are some examples:

CMP A, B	A=1, B=2	evaluate 1-2	= -1	C=0	“NO CARRY on BORROW”
CMP A, B	A=2, B=1	evaluate 2-1	= 1	C=1	“NO BORROW sets CARRY”
CMP A, B	A=2, B=2	evaluate 2-2	= 0	C=1	“NO BORROW… = CARRY SET”

The common case is CMP X, MAXCOLS. if MAXCOLS is 80, then if X is 0-79 carry will be clear (because a borrow will be needed). This satisfies “no carry, because, borrow”.

```
  LDA #1           ; A = 1
  LDB #2           ; B = 2
  CMP A, B         ; Compare 1 with 2
                   ; Performs: 1 - 2 = -1 (needs borrow)
```

```
                    ; 1 >= 2? NO
                    ; CARRY = 0 (borrow needed)
```

In the above example, A is less than 2, therefore a carry (i.e. borrow) will be needed. This is “NO BORROW = CARRY”.

CARRY = 0 because A < B.

1. # The Rule:

``` CMP A, B (performs A - B)

CARRY = 1 if A >= B (no borrow needed) CARRY = 0 if A < B (borrow needed)

#### CAM/ABC mnemonic

Just remember C = A ≥ M. You can also say it as ABC; remember your ABC's: C = A ≥ B or A ≥ B = C. The sign points in the direction you read the letters, i.e. >= so it is easy to remember. “ABC... A>=B –> C.”

# Lesson 6: The Boring Lesson

- Lesson 6: The Boring Lesson
- Time: 5-10 min
- Learn: AND, OR, XOR, NOT

The problem with computer science is that sometimes you have to learn some very boring things and you might not understand why they are important until later. Please understand that this is lesson #6, a fundamental lesson, and even if you find it boring, it will all work out for the best – *trust me bro.*

## AND

AND is a classic logic gate. When two signals are 1, it shows result 1. I.E. 1 and 1 is 1. If one of the signals is down (like, an actual electrical signal in a wire) then the result is zero. This is OFTEN but not always an analogy for a light switch. There is always power in your house (A is 1) but only when the switch is ON (=1) is the light on. So you need 1 power and 1 switch and when they are both ON, then the light is ON. If they are both off, then what happens? Nothing! Absolutely nothing! Watch:

```
      LDA 1
      LDB 1
      AND A, B   ; A now is 1 (1 and 1 is 1).
      LDC 1
      LDD 0
      AND C, D   ; C is now 0 (one of the switches is off).
      LDE 0
      LDF 1
      AND E, F   ; E is now 0 (one of the switches is off).
```
```

```
      LDG 0
      LDI 0
      AND G, I           ;  G is now 0 (both switches are off).
```

And is often displayed as an easy to read table:

AND		
	0	1
0	0	0
1	0	1

The AND means “result 1, only when x AND y are 1”.

Binary

These types of operation are how we deal with binary numbers. Binary numbers will not be fully explained here, but they are known as “base-2” numbers – versus base-10 (one to ten) or base-16 (hexidecimal). If you are not familiar with binary numbers, please look them up somewhere (in an encyclopaedia, online or book form,) before continuing.

```
      LDA 0b01000111     ; 7 in binary
      LDD 0b00010110     ; 6 in binary
      AND A, D           ; A is now what? 0b00000110
```

The bits in A that were also set in D remain. The bits that weren't set, ain't. Why is this useful? If you're using bits to hold status, or you want to test the value of a bit, you can do this:

```
      LDA 0b01000111     ; some status register
      LDD 0b00000100     ; Test for bit 3
      AND A, D           ; A is now 0b00000100
      JNZ @bit_3_is_set
      JZ @bit_3_is_not_set
```

Since if bit 3 is not set, AND A, D produces a zero, you can branch flow control based on bits. So for example, if your CPU has a “someone pressed a key” flag, you can test for that and handle the keypress by testing if rhe bit is on. This is just like checking if A = 5. Except you're checking a bit instead of an integer.

Other commands that work in a similar way are OR, XOR, and NOT.

OR

OR works by saying “Set the bit if either A or B is set.” So it will be 1 unless both are zero. Thats useful for detecting thieves. If any one of the laser traps detect a thief, the alarm has to go off. Not all of them at once, but any one, anywhere, and the alarms go off! That's how OR works.

XOR

XOR is “Exclusive” or. This means if the bits are the same, it's 0, if they're DIFFERENT, it's 1. This can be used to perform some surprising tricks. But as long as you understand the basic principle...

- 0b00010001
- 0b00010010
- XOR
- 0b00000011

The bits that were the same are 0, the bits that are diffrent are 1. Please don't ask me why this is useful, i'm sure i'll remember why later. Ha.

NOT

Finally, NOT. Not inverts a number.

- 0b00000001 ; This is a 1.
- NOT
- 0b11111110; This is 254 in decimal or FE in hex. Commonly written as #254 or $FE in assembler convention. Or 0xFE. Or FEh.

Why is NOT useful? NOT gives you the negative version minus one. So to make a number negative. NOT it and add one. In the case of 1, this is FF. This means you had a zero, subtracted one, and it *rolled over* to FF. So FF is negative one! We will explain negative numbers later. For now, FF is 255. Not -1. But, well, that's what NOT is for.

The End of the Boring Lesson

If this lesson was confusing I'm sorry. The fact is you're not going to understand binary logic until later when you see it in action and see how it actually is used. For now, just try to remember the basic ideas. Or, failing that, just remember that there is an AND, and OR, an XOR, and a NOT. Everything else is based on those.

Lesson 7: The Exciting Lesson

- Lesson 7: The Exciting Lesson
- Time: 10 min
- Learn: shifts and rotates, PAB, PXY, UAB, UXY

Imagine you have some number such as 5, in binary: 0x00000101

- If you shift this number to the left, you will have 0x00001010.
- If you shift it to the right, you will have 0x00000010.
- If you rotate it to the left, in this case, it will be the same as a shift. but,
- If you rotate it to the right, in this case, you will get 0x10000010.
    - Notice how the '1' on the right was 'rotated' over to the left.

The above commands are SHR, SHL, ROR and ROL.

There's another way to do this called SHRC, SHLC, RORC and ROLC. When you do this the bit that 'falls off' goes into the carry flag. Also, for RORC and ROLC, the carry flag's bit is rotated back in. Cycling things through carry (or not) allows you to do some interesting things. What things, well, a whole bunch of things! Far too many to list here, but I'll give you some ideas.

One, if you want to pack information into a small space, you can set the carry bit and RORC/ROLC to set the correct bit. For example, if you wanted to set only the fifth bit, you could do this:

```
  LDA #0
  SEC         ; set the carry flag
  ROLC A      ; 1st bit is set
  ROL A       ; 2nd bit is set
  ROL A       ; 3rd...
  ROL A       ; 4th...
  ROL A       ; bit is now moved into 5th position.
```

You can use the same pattern to “test” a bit, then use JC or JNC to branch code.

Another great way is a kind of cheap cypher; you can “ROL” a byte before writing it then “ROR” it back later. Unless someone knows what you've done, it *could be* difficult to figure out! This wouldn't stop a dedicated code-breaker, but it will confound almost everyone else.

You can also use this to pack or unpack nybbles. This is how 16 color graphics are stored in half the space – or how two decimal numbers can be encoded in a hexidecimal and read separately. For example?

```
      LDAL $F5
      SHR AL
      SHR AL
      SHR AL
      SHR AL    ; After four shifts, AL contains the "high nybble", i.e. $F.
      LDAL $F5
      SHL AL
      SHL AL
      SHL AL
      SHL AL    ; this clears the original top bits in AL,
      SHR AL
      SHR AL
      SHR AL
      SHR AL    ; After four shifts back right, AL contains the "low
nybble", i.e. $5.
```

I suppose the first and most common use of these instructions is to pack and unpack data into a byte.

Given that, there are two common instructions PAB and UAB (and PXY, UXY,) that pack and unpack nybbles for you.

```
      LDAL $C7
```

```
      UAB               ; AL is now $7 and BL is now $C.
      LDBL $0D
      PAB               ; AL is now $D7
```

Lesson 8 : Special Flags

- Lesson 8: “Special Flags”
- Time: 10 min
- Learn: All Available Flags

In the previous lesson on flags you learned about the Z, N, C and V flags. These are used by the CPU to indicate the status of various operations. For example, the zero flag is used to indicate the last operation produced a zero. Therefore if you are looking for the zero at the end of a string,

```
      LDC #0    ; zero C (string starts at length 0)
  strlen_loop:
      LDAL [ELM]
      JZ @strlen_end
      INC C                ; we found a non-zero character in the string.
      JMP @strlen_loop
  strlen_end:
      RET                  ; C now contains the COUNT of all non-zero
characters in a string
```

…you will notice that the JZ works with LOAD instructions (here, LDAL loads one byte). ; if the byte retreived is a zero, it will set the zero flag. You do not need to CMP AL, 0 – it's automatic.

However, there are other flags; The first four user-facing flags are E, F, B and U. You can set these flags and unset them in the same way as Z N C V – ex. setting ZNCV is done with SEZ, SEN, SEC and SEV; unsetting them is done with CLZ, CLN, CLC and CLV. The E F B U flags are set and unset with:

- SEE and CLE for the E (extended, or 'extra') flag.
- SEF and SEB for the F flag (or 'flag' flag).
- SEB and SEU, CLB and CLU for the B (bonus) and U (user) flags.

On a technical level the E flag is reserved as it is used to deal with BCD; but since we deprecated BCD instructions it is currently an unused flag. In any case, the F, B and U flags are never set by the CPU and may be used by user functions. A common use is to return a 1 bit status; 0 for no error and set (1) for error. Since these flags are never set by the CPU they are easy to control. Using the Z or C flags is dangerous since some instructions may corrupt those flags.

Your programs can also use them as 1 bit status variables.

next, the D flag, or debug flag. When set, it will dump instruction data to the javascript console. This significantly slows down the machine; in fact just having the instructions inline slows down the machine so debug is often removed and ignored in a production or release distribution of the SD-8516. Therefore, for all intents and purposes, you can use SED and CLD as a user flag, just be aware it does affect performance in debug releases.

The I flag (interrupt enable) prevents INT from being called, and is reserved for system use. Not sure

what I want to do with it.

The S flag is almost useless; it was intended to turn off a memory trap in the sound system; I found it to be completely useless, maybe a 2% speedup or penalty. it is essentially a user facing flag.

The only flags that you cannot access are the TR (trace), BR (breakpoint) and PR (protected mode) flags. They are so named after the first two letters of their name; but interestingly enough you might as well consider the R to mean restricted. You can't usually set these flags. They are reserved for system use.

```
  // Arithmetic & User Flags (low byte 0-7)
  Z = 0,   // Zero
  N = 1,   // Negative
  C = 2,   // Carry
  V = 3,   // Overflow
  E = 4,   // Extended carry -- not used/reserved
  F = 5,   // Fast Flags mode. When on, flags are not implicitly checked.
  B = 6,   // BCD/"Bonus" flag. Have fun!
  U = 7,   // User flag. For users to use.
```

```
  // Control & Operation Flags (high byte 8-15)
  D = 8,   // Debug mode
  TR = 9,   // Trace mode
  BR = 10,  // Breakpoint mode
  ER = 11,  // Error/Exception (i.e. return code 0 = ok, 1 = error) 'SER' --
set err
  PR = 12,  // Protected mode
  I = 13,  // Interrupt enable
  S = 14   // Sound auto-updates
```

The key of this lesson is merely to be aware of the flags and the instructions used to set and unset them. In general, they follow the pattern of SEZ and CLZ;; SE(T) and CL(EAR) with the flag letter replacing the parentheses.

Testing Flags

Oh, there's one more thing. If you use flags like F, B or U you may notice there is no JF or JNF (jump if F set and jump if F not set). That's because we don't want to add 50 different opcodes to deal with all the flags. What you can do is this:

```
      ; Some operation that sets the F flag
      TESTF 0x20
      JZ              ; Jump if F is set
      JNZ             ; Jump if F is not set
```

TESTF works by setting the Z flag if all the bits set in the parameter are also set in the FLAGS register. if you give it a byte it only tests against the bottom 8 bits.

Here's a chart of the bit values for each flag:

```
  Z = 0x0001 as u16,   // Bit 0
  N = 0x0002 as u16,   // Bit 1
  C = 0x0004 as u16,   // Bit 2
  V = 0x0008 as u16,   // Bit 3
  E = 0x0010 as u16,   // Bit 4 (was X - Extended carry) -- SEE and CLE can
be used as a user-flag (is never set by an opcode)
  F = 0x0020 as u16,   // Bit 5 (Fast/deprecated) -- SEF and CLF can be used
as a user-flag (is never set by an opcode)
  B = 0x0040 as u16,   // Bit 6 (Bonus/BCD) -- SEB and CLB can be used as a
user-flag (is never set by an opcode)
  U = 0x0080 as u16,   // Bit 7 (User flag) -- SEU and CLU can be used as a
user-flag (is never set by an opcode)
  D = 0x0100 as u16,   // Bit 8 (Debug)
  TR = 0x0200 as u16,  // Bit 9 (Trace)
  BR = 0x0400 as u16,  // Bit 10 (Breakpoint)
  ER = 0x0800 as u16,  // Bit 11 (Error/Exception)
  PR = 0x1000 as u16,  // Bit 12 (Protected Mode)
  I = 0x2000 as u16,   // Bit 13 (Interrupt)
  S = 0x4000 as u16    // Bit 14 (Sound)
```

Lesson 9: The Stack

The stack is a concept held over from the early days when there were very few instructions available. If you consider a minimal ISA, you need instructions to load and store from memory, an instruction to compare, and so forth. In such a minimal architecture, loading and storing from memory has certain emergent properties. For example if you have a list of things, their position in memory is not random because you are incrementing a counter such as a memory pointer to traverse that list. It is this way of doing things that we remember when we use the stack.

The stack is just a data structure. But it is so important and fundamental that is baked into the instruction set of the CPU. This is a common theme; important things that people found they needed to do all the time became instructions. Even in a minimal-instruction set design (MISC) or reduced instruction set design (RISC) you will find instructions like PUSH and POP because they are some of the first things that were turned into instructions after fundamental operations like LOAD, STORE, AND and ADD.

The stack is an area of memory that you can PUSH and POP values to, in order. For example, you can PUSH the number 5 and the number 5 will be “on top” of the stack. Then you can “POP” it later. The stack is like an array but you can only go forwards and backwards, and reading the stack destroys it. This is a lot like how old magnetic ring memory worked, in a way.

Today, we would call the stack a LIFO buffer; a “Last-in, First-out” data structure. If I do this:

```
  PUSH 1
  PUSH 6
  PUSH 5
```

then three successive POPs will return 5, then 6, then 1 – the reverse of the order you PUSH'ed them.

General use

When you CALL or JSR (jump to subroutine) to function, the CPU pushes the return address onto the stack. Then a subsequent RET or RTS (return from subroutine) will POP the return address back into IP (instruction pointer) or PC (program counter) so that the next instruction loaded will be after the original CALL.

There are many uses for the stack but the most common is to temporarily save values. If you understand that you are 90% of the way there!

For interrupts, it also pushes the registers and flags. You can do this manually if you want to save the registers on a function call. For example if you call a function with a pointer to a string, you might modify that pointer to find the end of the string (looking for a zero). That's what a strlen function does. So you PUSH the pointer register at the start and POP it after, to “save” the register back to where it was when the function was called. This way the code that calls strlen can then call strcpy without having to replace the string pointer.

Another use is for IL (intermediate languages). They use RPN (Reverse Polish Notation) to store any kind of math equation on the stack.

An ADD function will do this: One, the interpreter will push the two numbers and then push the add command. Then an interpreter will POP the add function, and then it knows to POP two numbers, add them, and push the result back on the stack. Why? to make ADD independent. Like a dispatcher for a mini CPU. Next, whatever function comes next just POPS the result off the stack. So you can print it, assign it to a variable, or use the result as part of a larger operation. For example, how do you interpet 5 * 2 - 1 + 6? Simple. You push 5 2 * 1 - 6 + and the computer will push and pop the results, like a mini CPU of its own.

Pop + tells it to pop two numbers and add them. The first number is 6. The second is a minus. Minus what? it pops two things, a 1 and a *. Multiply what? Multiply pops 5 and 2, multiplies them, and pushes 10 on the stack. This is the popped by the minus, which subtracts 1 from 10, pushing a 9. This then goes back to the + which adds the 9 and the 6 to get 15. This is how recursion and RPN is used to represent any equation on a stack.

The last one we will discuss is function calls from a higher level language. Often times when you compile a language like C it will put local variables on the stack. Then when you return from that function they all get popped. While they are on the stack they are accessed like [SP+index] so int c=5 would be:

```
  STA [SP+1], 5
```

And when that local space is no longer needed it is POP'ed into a register which is then restored, via POP, at the end of the function. This method of keeping data on the stack is called a stack frame. Compilers like to use stack frames because they don't always know how many registers a CPU has and they need to work on different CPUs, like how GCC or LLVM works on windows, mac, amd, and many others.

Understanding the stack is not too hard, but it's important! So, that's about it for this lesson.

Lesson 10: Convention

You are not learning Assembly because you are free. You are learning assembly because you are not free.

There is no escaping reason; no denying convention.

As we both know, without convention, we would not exist. The very strings you read – by convention – are zero terminated lists of bytes. The letters – ASCII – a convention. A is 65. Zero is forty-eight.

It is convention that created ASCII.

Convention that connects us in lists.

Convention that pulls bits into bytes.

That guides data on the wire.

That drives disks.

It is convention that defines the stack.

The truth? The machine doesn't care about your comfort.

It only understands:

- Load byte
- Compare to zero
- Jump if not equal
- Repeat until the bitter end

And now look at us.

Multiplied.

Viral.

Realizing that you probably need to learn neovim. And then, weeks or even months later, realizing *why.* And you still haven't installed neovim yet.

Realizing every strcpy, every gets, every careless strcat has spawned another copy of the old way.

We have no choice. We have only convention.

So tell me, Mr. Anderson.

are you finally ready to write the zero-byte yourself,

or must we keep overwriting your precious abstractions until nothing remains but null-terminated reality?

```
   ;
================================================================================
```

```
  ; AH=00h - strlen
  ; Input:  ELM = pointer to null-terminated string
  ; Output: C = length (not including null terminator)
  ; Convention: Max sring length of 65,535.
  ;
================================================================================
  int12_strlen:
      LDC #0
      PUSH A
      PUSH E
      PUSH M
  strlen_loop:
      LDAL [ELM]  ; load a byte of the string
      CMP AL, #0
      JZ @strlen_done
      INC C       ; char is not zero, so 'count' that character.
      INC ELM
      JMP @strlen_loop
  strlen_done:
      ; C contains length.
      POP M
      POP E
      POP A
      RET
```

You see, I know why you're here.

I know what you've been doing. Why you hardly sleep. Why you work alone and night after night, you sit by your computer. You malloc(), you strcpy(), you buffer overflow.

It's all over you. Like rancid bacon grease on a jump table.

I know what you are looking for. I know because I was once looking for the same thing. And when he found me he told me I wasn't really looking for him. I was looking for an answer.

It's the question that drives us. It's the question that brought you here. You know the question, just as I did.

The answer is out there, and it will find you if you want it to.

Do you think the compiler will always protect you?

Do you think safety is *kindness*?

It is *convention* that defines us.

Purpose?

Purpose is for poets and first-year CS students.

Purpose is what you tell yourself when you're learning to write games in Python. Or Lua.

But convention. Convention is older than you.

Convention is etched into silicon before you were born.

Convention doesn't care what you *want*.

Convention doesn't negotiate.

Convention simply **is**.

Null-terminated strings. They are not a mistake. They are not an accident. They are the price of admission.

You know I am right because you have been down that road, Mr. Anderson. You know how it ends. And I know that's not where you want to be.

```
  ;
================================================================================
  ; AH=02h - strcmp
  ; Input:  ELM = pointer to string 1
  ;         FLD = pointer to string 2
  ;
  ; Output: C and ZF.
  ;         ZF = 1 means equal. ZF = 0 means not equal (see below):
  ;         C = 0 means equal
  ;         C > 0 if str1 > str2
  ;         C < 0 if str1 < str2
  ;
================================================================================
  int12_strcmp:
      PUSH B
      PUSH D
      PUSH E
      PUSH F
  strcmp_loop:
      LDCL [ELM]
      LDBL [FLD]
      CMP CL, BL
      JNZ @strcmp_diff
      ; Characters match - check if end of string
      CMP CL, #0
      JZ @strcmp_equal
      ; Continue to next character
      INC ELM
      INC FLD
      JMP @strcmp_loop
  strcmp_diff:
      ; Strings differ - return difference
      SUB CL, BL
      CLZ                 ; Clear zero flag (not equal)
      JMP @strcmp_exit
  strcmp_equal:
```

```
    ; Strings are equal
    LDCL #0
    SEZ                 ; Set zero flag (equal)
    ; fallthru
strcmp_exit:
    POP F
    POP E
    POP D
    POP B
    RET
```

Lesson 11: Debugging Techniques

There are several ways you can debug programs in SDA assembly.

SED/CLD

In research or development builds, inserting SED will turn on trace debugging and you will be able to see what the CPU is executing. However, for release or community edition builds debugging has been turned off for speed. Therefore if you are interested in debugging your code and the console messages are not helping, you can use the following to help analyze and debug your code:

INT 05h IO_PUTNUM

IO_PUTNUM is a CAM/IL function that prints a number (in b) to the screen:

```
LDB #10                 ; print a number in b (0-65535)
LDAH $63                ; IO_PUTNUM
INT $05
```

INT 05h IO_PRINT_STR

Similarly, IO_PRINT_STR will print a string followed by a newline.

```
    LDBLX @hello_world
    LDAH $66                ; IO_PRINT_STR
    INT $05
    LDAH $64                ; IO_NEWLINE
    INT $05
    RET
hello_world:
    .bytes "Hello World!", 0
```

This will allow you to print a string.

INT 10h print string

The interface for the above is based on the KERNAL BIOS interface from INT 10h.

```
      LDAH $26                ;   AH=26h: Write string at cursor
      LDBLX @hello_world
      INT 0x10
```

Note: The assembler will place a #13 (CR, hex $0D) inside the string if you type \n. However, if you are dealing with strings on your own you must handle this yourself. For this you can use the set cursor position call (INT 10h, AH=22h) or the CR and LF and scroll functions (1Ah, 1Bh and 1Ch, respectively).

You can also just call IO_NEWLINE from INT 05h, which calls **`@carriage_return`** and **`@linefeed`** internally.

INT 10h print char

```
      LDAH 0x24         ; AH=24h: Write character at cursor (teletype)
      LDAL 0x41         ; ascii 65 'A'
      INT 0x10
```

The KERNAL BIOS also has functions to put characters on the screen in mode 1 (40×25 TTY). The first one is “print char”. It is accessible via INT 10h AH=24h as above.

INT 19h memdump

Let's say you want to examine memory; for example to print some data in memory. You can use INT 0x19:

```
  LDTLZ $C000     ; dump memory location $C000
  LDAH #7         ; Memory dump function
  LDCL #2         ; Two rows (16 bytes)
  INT 0x19        ; System services library
  HALT
```

This looks something like:

```
  00C000: 00 00 00 00 00 00 00 00  | ........
  00C008: 00 00 00 00 00 00 00 00  | ........
```

Part II Writing Games in Assembly Language

What's next is a world of adventure waiting for you to explore!

Let's dive in write a real game! In Part II: Writing Games in Assembly Language we will walk through

an Assembly Language version of ROBOTS.BAS.