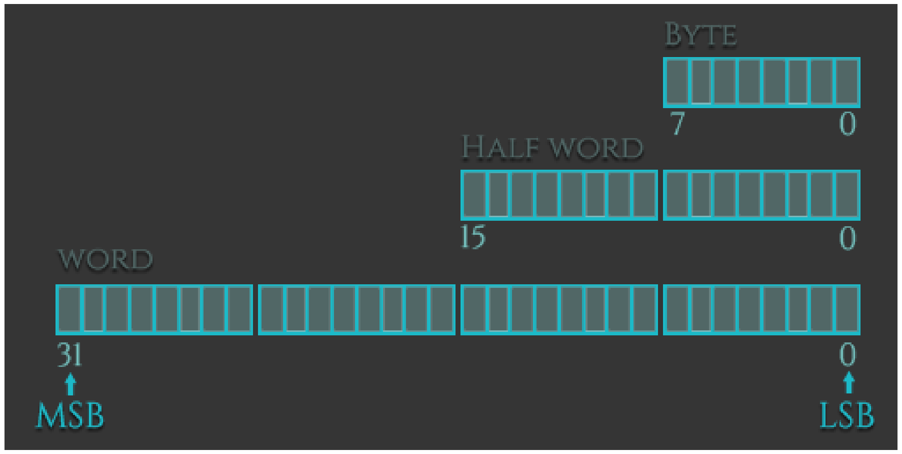
AZERIA

DATA TYPES

This is part two of the ARM Assembly Basics tutorial series, covering data types and registers.

Similar to high level languages, ARM supports operations on different datatypes.



The data types we can load (or store) can be signed and unsigned words, halfwords, or bytes. The extensions for these data types are: -h or -sh for halfwords, -b or -sb for bytes, and no extension for words. The difference between signed and unsigned data types is:

- Signed data types can hold both positive and negative values and are therefore lower in range.
- Unsigned data types can hold large positive values (including 'Zero') but cannot hold negative values and are therefore wider in range.

Here are some examples of how these data types can be used with the instructions Load and Store:

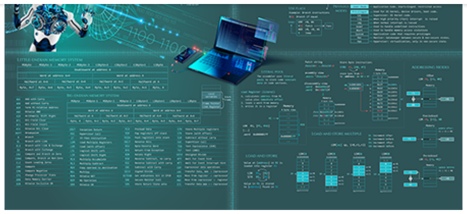
```
ldr = Load Word
ldrh = Load unsigned Half Word
ldrsh = Load signed Half Word
ldrb = Load unsigned Byte
ldrsb = Load signed Bytes
```

Twitter: @Fox0x01 and @azeria_labs

New ARM Assembly Cheat Sheet

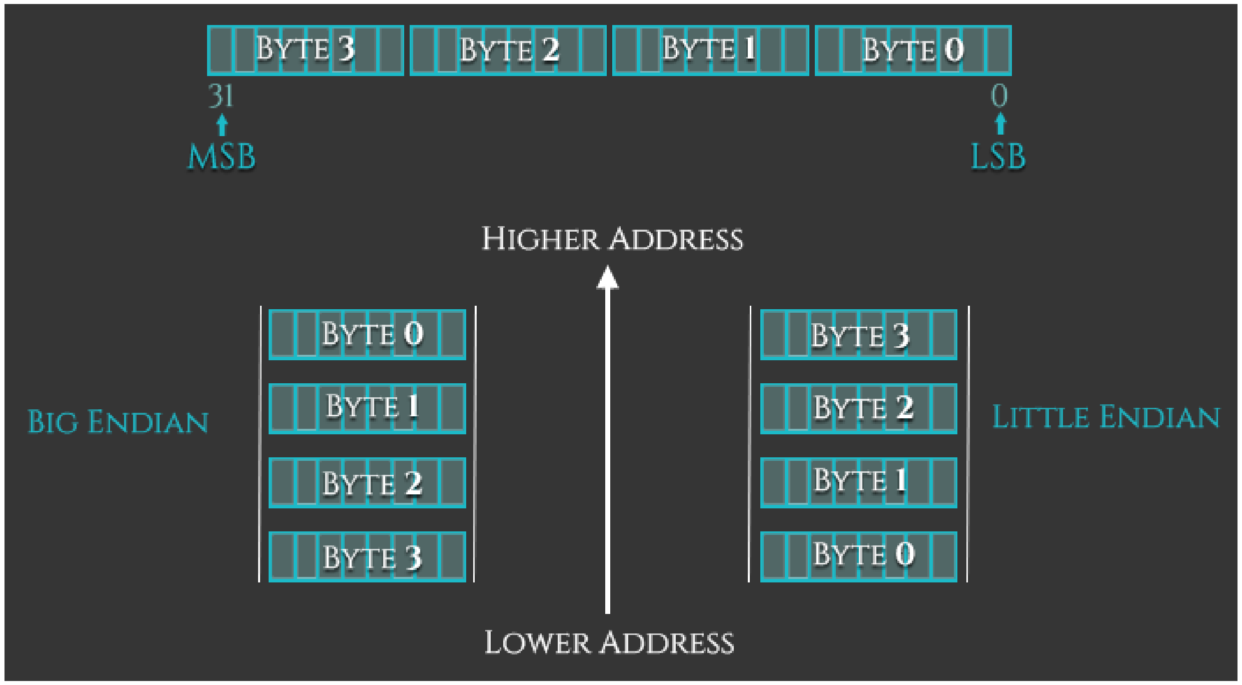
POSTER DIGITAL

```
strh = Store unsigned Half Word
strsh = Store signed Half Word
strb = Store unsigned Byte
strsb = Store signed Byte
```



ENDIANNESS

There are two basic ways of viewing bytes in memory: Little-Endian (LE) or Big-Endian (BE). The difference is the byte-order in which each byte of an object is stored in memory. On little-endian machines like Intel x86, the **least**-significant-byte is stored at the lowest address (the address closest to zero). On big-endian machines the **most**-significant-byte is stored at the lowest address. The ARM architecture was little-endian before version 3, since then it is bi-endian, which means that it features a setting which allows for switchable endianness. On ARMv6 for example, instructions are fixed little-endian and data accesses can be either little-endian or big-endian as controlled by bit 9, the E bit, of the Program Status Register (CPSR).



ARM REGISTERS

additional registers are available in privileged software execution (with the exception of ARMv6-M and ARMv7-M). In this tutorial series we will work with the registers that are accessible in any privilege mode: r0-15. These 16 registers can be split into two groups: general purpose and special purpose registers.

#	Alias	Purpose
R0	–	General purpose
R1	–	General purpose
R2	–	General purpose
R3	–	General purpose
R4	–	General purpose
R5	–	General purpose
R6	–	General purpose
R7	–	Holds Syscall Number
R8	–	General purpose
R9	–	General purpose
R10	–	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter

CPSR	–	Current Program Status Register

The following table is just a quick glimpse into how the ARM registers **could** relate to those in Intel processors.

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

R0-R12: can be used during common operations to store temporary values, pointers (locations to memory), etc. R0, for example, can be referred as accumulator during the arithmetic operations or for storing the result of a previously called function. R7 becomes useful while working with syscalls as it stores the syscall number and R11 helps us to keep track of boundaries on the stack serving as the frame pointer (will be covered later). Moreover, the function calling convention on ARM specifies that the first four arguments of a function are stored in the registers r0-r3.

R13: SP (Stack Pointer). The Stack Pointer points to the top of the stack. The stack is an area of memory used for function-specific storage, which is reclaimed when the function returns. The stack pointer is therefore used for allocating space on the stack, by subtracting the value (in bytes) we want to allocate from the stack pointer. In other words, if we want to allocate a 32 bit value, we subtract 4 from the stack pointer.

R14: LR (Link Register). When a function call is made, the Link Register gets updated with a memory address referencing the next instruction where the function was initiated from. Doing this allows the program return to the “parent” function that initiated the “child”

R15: PC (Program Counter). The Program Counter is automatically incremented by the size of the instruction executed. This size is always 4 bytes in ARM state and 2 bytes in THUMB mode. When a branch instruction is being executed, the PC holds the destination address. During execution, PC stores the address of the current instruction plus 8 (two ARM instructions) in ARM state, and the current instruction plus 4 (two Thumb instructions) in Thumb(v1) state. This is different from x86 where PC always points to the next instruction to be executed.

Let's look at how PC behaves in a debugger. We use the following program to store the address of pc into r0 and include two random instructions. Let's see what happens.

```
.section .text
.global _start

_start:
 mov r0, pc
 mov r1, #2
 add r2, r1, r1
 bkpt
```

In GDB we set a breakpoint at _start and run it:

```
gef> br _start
Breakpoint 1 at 0x8054
gef> run
```

Here is a screenshot of the output we see first:

```
$r0 0x00000000   $r1 0x00000000   $r2 0x00000000   $r3 0x00000000
$r4 0x00000000   $r5 0x00000000   $r6 0x00000000   $r7 0x00000000
$r8 0x00000000   $r9 0x00000000   $r10 0x00000000  $r11 0x00000000
$r12 0x00000000  $sp 0xbefff7e0   $lr 0x00000000   $pc 0x00008054
$cpsr 0x00000010
```

```
0x805c <_start+8> add r1, r0, r0
0x8060 <_start+12> bkpt 0x0000
0x8064 andeq r1, r0, r1, asr #10
0x8068 cmnvs r5, r0, lsl #2
0x806c tsteq r0, r2, ror #18
0x8070 andeq r0, r0, r11
0x8074 tsteq r8, r6, lsl #6
```

We can see that PC holds the address (0x8054) of the next instruction (mov r0, pc) that will be executed. Now let's execute the next instruction after which R0 should hold the address of PC (0x8054), right?

```
$r0 0x0000805c   $r1 0x00000000   $r2 0x00000000   $r3 0x00000000
$r4 0x00000000   $r5 0x00000000   $r6 0x00000000   $r7 0x00000000
$r8 0x00000000   $r9 0x00000000   $r10 0x00000000  $r11 0x00000000
$r12 0x00000000  $sp 0xbefff7e0   $lr 0x00000000   $pc 0x00008058
$cpsr 0x00000010

0x8058 <_start+4> mov r0, #2       <- $pc
0x805c <_start+8> add r1, r0, r0
0x8060 <_start+12> bkpt 0x0000
0x8064 andeq r1, r0, r1, asr #10
0x8068 cmnvs r5, r0, lsl #2
0x806c tsteq r0, r2, ror #18
0x8070 andeq r0, r0, r11
0x8074 tsteq r8, r6, lsl #6
0x8078 adfcssp f0, f0, #4.0
```

…right? Wrong. Look at the address in R0. While we expected R0 to contain the previously read PC value (0x8054) it instead holds the value which is two instructions ahead of the PC we previously read (0x805c). From this example you can see that when we directly read PC it follows the definition that PC points to the next instruction; but when debugging, PC points two instructions ahead of the current PC value (0x8054 + 8 = 0x805C). This is because older ARM processors always fetched two instructions ahead of the currently executed instructions. The reason ARM retains this definition is to ensure compatibility with earlier processors.

When you debug an ARM binary with gdb, you see something called Flags:

The register $cpsr shows the value of the Current Program Status Register (CPSR) and under that you can see the Flags thumb, fast, interrupt, overflow, carry, zero, and negative. These flags represent certain bits in the CPSR register and are set according to the value of the CPSR and turn **bold** when activated. The N, Z, C, and V bits are identical to the SF, ZF, CF, and OF bits in the EFLAG register on x86. These bits are used to support conditional execution in conditionals and loops at the assembly level. We will cover condition codes used in Part 6: Conditional Execution and Branching.

The picture above shows a layout of a 32-bit register (CPSR) where the left (<-) side holds most-significant-bits and the right (->) side the least-significant-bits. Every single cell (except for the GE and M section along with the blank ones) are of a size of one bit. These one bit sections define various properties of the program's current state.

Flag	**Description**

N (Negative)	Enabled if result of the instruction yields a negative number.
Z (Zero)	Enabled if result of the instruction yields a zero value.
C (Carry)	Enabled if result of the instruction yields a value that requires a 33rd bit to be fully represented.
V (Overflow)	Enabled if result of the instruction yields a value that cannot be represented in 32 bit two's complement.
E (Endian-bit)	ARM can operate either in little endian, or big endian. This bit is set to 0 for little endian, or 1 for big endian mode.
T (Thumb-bit)	This bit is set if you are in Thumb state and is disabled when you are in ARM state.
M (Mode-bits)	These bits specify the current privilege mode (USR, SVC, etc.).
J (Jazelle)	Third execution state that allows some ARM processors to execute Java bytecode in hardware.

Let's assume we would use the CMP instruction to compare the numbers 1 and 2. The outcome would be 'negative' because 1 – 2 = -1. When we compare two equal numbers, like 2 against 2, the Z (zero) flag is set because 2 – 2 = 0. Keep in mind that the registers used with the CMP instruction won't be modified, only the CPSR will be modified based on the result of comparing these registers against each other.

This is how it looks like in GDB (with GEF installed): In this example we compare the registers r1 and r0, where r1 = 4 and r0 = 2. This is how the flags look like after executing the cmp r1, r0 operation:

The Carry Flag is set because we use **cmp r1, r0** to compare 4 against 2 (4 – 2). In contrast, the Negative flag (N) is set if we use **cmp r0, r1** to compare a smaller number (2) against a bigger number (4).

Here's an excerpt from the ARM infocenter:

The APSR contains the following ALU status flags:

N – Set when the result of the operation was Negative.

Z – Set when the result of the operation was Zero.

C – Set when the operation resulted in a Carry.

V – Set when the operation caused oVerflow.

A carry occurs:

- if the result of an addition is greater than or equal to 2^{32}
- if the result of a subtraction is positive or zero
- as the result of an inline barrel shifter operation in a move or logical instruction.

Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than 2^{31}.

‹ PART 1: INTRODUCTION TO ARM ASSEMBLY BASICS

PART 3: ARM INSTRUCTION SET ›