



MEMORY INSTRUCTIONS: LOAD AND STORE

ARM uses a load-store model for memory access which means that only load/store (LDR and STR) instructions can access memory. While on x86 most instructions are allowed to directly operate on data in memory, on ARM data must be moved from memory into registers before being operated on. This means that incrementing a 32-bit value at a particular memory address on ARM would require three types of instructions (load, increment, and store) to first load the value at a particular address into a register, increment it within the register, and store it back to the memory from the register.

To explain the fundamentals of Load and Store operations on ARM, we start with a basic example and continue with three basic offset forms with three different address modes for each offset form. For each example we will use the same piece of assembly code with a different LDR/STR offset form, to keep it simple. The best way to follow this part of the tutorial is to run the code examples in a debugger (GDB) on [your lab environment](#).

1. Offset form: **Immediate** value as the offset

- Addressing mode: Offset
- Addressing mode: Pre-indexed
- Addressing mode: Post-indexed

2. Offset form: **Register** as the offset

- Addressing mode: Offset
- Addressing mode: Pre-indexed
- Addressing mode: Post-indexed

3. Offset form: **Scaled register** as the offset

- Addressing mode: Offset
- Addressing mode: Pre-indexed
- Addressing mode: Post-indexed

ARM Assembly Basics

1. Writing ARM Assembly
2. ARM Data Types and Registers
3. ARM Instruction set

4. Memory Instructions: Load and Store

5. Load and Store Multiple
6. Conditional Execution and Branching
7. Stack and Functions

Assembly Basics Cheatsheet

Twitter: [@Fox0x01](#) and [@azeria_labs](#)

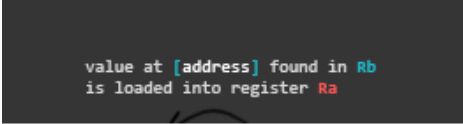
New ARM Assembly Cheat Sheet

POSTER

DIGITAL

First basic example

Generally, LDR is used to load something from memory into a register, and STR is used to store something from a register to a memory address.



value at [address] found in Rb
is loaded into register Ra

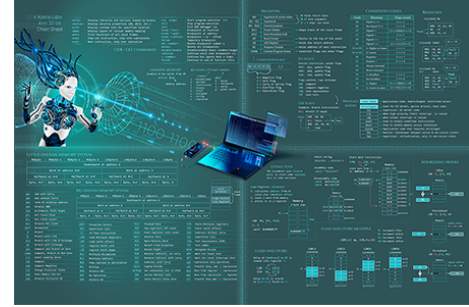
```
LDR R2, [R0]    @ [R0] - origin address is the value found in R0.  
STR R2, [R1]    @ [R1] - destination address is the value found in
```

LDR operation: loads the value at the address found in R0 to the destination register R2.

STR operation: stores the value found in R2 to the memory address found in R1.

This is how it would look like in a functional assembly program:

```
.data          /* the .data section is dynamically created and its addresses cannot be easily predicted */  
var1: .word 3   /* variable 1 in memory */  
var2: .word 4   /* variable 2 in memory */  
  
.text          /* start of the text (code) section */  
.global _start  
  
_start:  
    ldr r0, adr_var1 @ load the memory address of var1 via label adr_var1 into R0  
    ldr r1, adr_var2 @ load the memory address of var2 via label adr_var2 into R1  
    ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to register R2  
    str r2, [r1]      @ store the value found in R2 (0x03) to the memory address found in R1  
    bkpt  
  
adr_var1: .word var1 /* address to var1 stored here */  
adr_var2: .word var2 /* address to var2 stored here */
```



At the bottom we have our Literal Pool (a memory area in the same code section to store constants, strings, or offsets that others can reference in a position-independent manner) where we store the memory addresses of var1 and var2 (defined in the data section at the top) using the labels adr_var1 and adr_var2. The first LDR loads the address of var1 into register R0. The second LDR does the same for var2 and loads it to R1. Then we load the value stored at the memory address found in R0 to R2, and store the value found in R2 to the memory address found in R1.

When we load something into a register, the brackets ([]) mean: the value found in the register between these brackets is a memory address we want to load something from.

When we store something to a memory location, the brackets ([]) mean: the value found in the register between these brackets is a memory address we want to store something to.

This sounds more complicated than it actually is, so here is a visual representation of what's going on with the memory and the registers when executing the code above in a debugger:

Let's look at the same code in a debugger.

```
gef> disassemble _start
Dump of assembler code for function _start:
0x00008074 <+0>:      ldr  r0, [pc, #12]    ; 0x8088 <adr_var1>
```

```
0x00008078 <+4>:      ldr  r1, [pc, #12]    ; 0x808c <adr_var2>
0x0000807c <+8>:      ldr  r2, [r0]
0x00008080 <+12>:     str  r2, [r1]
0x00008084 <+16>:     bx   lr
End of assembler dump.
```

The labels we specified with the first two LDR operations changed to [pc, #12]. This is called PC-relative addressing. Because we used labels, the compiler calculated the location of our values specified in the Literal Pool (PC+12). You can either calculate the location yourself using this exact approach, or you can use labels like we did previously. The only difference is that instead of using labels, you need to count the exact position of your value in the Literal Pool. In this case, it is 3 hops (4+4+4=12) away from the effective PC position. More about PC-relative addressing later in this chapter.

Side note: In case you forgot why the effective PC is located two instructions ahead of the current one, it is described in Part 2 [... 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 state. This is different from x86 where PC always points to the next instruction to be executed...].

1. Offset form: Immediate value as the offset

```
STR    Ra, [Rb, imm]
LDR    Ra, [Rc, imm]
```

Here we use an immediate (integer) as an offset. This value is added or subtracted from the base register (R1 in the example below) to access data at an offset known at compile time.

```
.data
var1: .word 3
var2: .word 4

.text
.global _start

_start:
    ldr r0, adr_var1 @ load the memory address of var1 via label adr_var1 into R0
    ldr r1, adr_var2 @ load the memory address of var2 via label adr_var2 into R1
    ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to register R2
    str r2, [r1, #2]  @ address mode: offset. Store the value found in R2 (0x03) to the memory address found in R1 + 2
    str r2, [r1, #4]! @ address mode: pre-indexed. Store the value found in R2 (0x03) to the memory address found in R1 + 4, then increment R1 by 4
    ldr r3, [r1], #4  @ address mode: post-indexed. Load the value at memory address found in R1 to register R3, then increment R1 by 4
    bkpt 0x00000000

    adr_var1: .word var1
    adr_var2: .word var2
```

Let's call this program `ldr.s`, compile it and run it in GDB to see what happens.

```
$ as ldr.s -o ldr.o
$ ld ldr.o -o ldr
$ gdb ldr
```

In GDB (with `gef`) we set a break point at `_start` and run the program.

```
gef> break _start
gef> run
...
gef> nexti 3      /* to run the next 3 instructions */
```

The registers on my system are now filled with the following values (keep in mind that these addresses might be different on your system):

```
$r0 : 0x00010098 -> 0x00000003
$r1 : 0x0001009c -> 0x00000004
$r2 : 0x00000003
$r3 : 0x00000000
$r4 : 0x00000000
$r5 : 0x00000000
$r6 : 0x00000000
$r7 : 0x00000000
$r8 : 0x00000000
$r9 : 0x00000000
$r10 : 0x00000000
$r11 : 0x00000000
$r12 : 0x00000000
$sp : 0xbffff7e0 -> 0x00000001
$lr : 0x00000000
$pc : 0x00010080 -> <_start+12> str r2, [r1]
$cpsr : 0x00000010
```

The next instruction that will be executed is a STR operation with the **offset address mode**. It will store the value from R2 (0x00000003) to the memory address specified in R1 (0x0001009c) + the offset (#2) = 0x1009e.

```
gef> nexti
gef> x/w 0x1009e
0x1009e <var2+2>: 0x3
```

The next STR operation uses the **pre-indexed address mode**. You can recognize this mode by the exclamation mark (!). The only difference is that the base register will be updated with the final memory address in which the value of R2 will be stored. This means, we store the value found in R2 (0x3) to the memory address specified in R1 (0x1009c) + the offset (#4) = 0x100A0, and update R1 with this exact address.

```
gef> nexti
gef> x/w 0x100A0
0x100a0: 0x3
gef> info register r1
r1      0x100a0      65696
```

The last LDR operation uses the **post-indexed address mode**. This means that the base register (R1) is used as the final address, then updated with the offset calculated with R1+4. In other words, it takes the value found in R1 (not R1+4), which is 0x100A0 and loads it into R3, then updates R1 to R1 (0x100A0) + offset (#4) = 0x100a4.

```
gef> info register r1
r1      0x100a4      65700
gef> info register r3
r3      0x3          3
```

Here is an abstract illustration of what's happening:

2. Offset form: Register as the offset.

```
STR    Ra, [Rb, Rc]
LDR    Ra, [Rb, Rc]
```

This offset form uses a register as an offset. An example usage of this offset form is when your code wants to access an array where the index is computed at run-time.

```
.data
var1: .word 3
var2: .word 4

.text
.global _start

_start:
```



```

ldr r0, adr_var1 @ load the memory address of var1 via label adr_var1 to R0
ldr r1, adr_var2 @ load the memory address of var2 via label adr_var2 to R1
ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to R2
str r2, [r1, r2] @ address mode: offset. Store the value found in R2 (0x03) to the memory address
str r2, [r1, r2]! @ address mode: pre-indexed. Store value found in R2 (0x03) to the memory address
ldr r3, [r1], r2 @ address mode: post-indexed. Load value at memory address found in R1 to register
bx lr

```

```

adr_var1: .word var1
adr_var2: .word var2

```

After executing the first STR operation with the **offset address mode**, the value of R2 (0x00000003) will be stored at memory address $0x0001009c + 0x00000003 = 0x0001009f$.

```

gef> x/w 0x0001009f
0x1009f <var2+3>: 0x00000003

```

The second STR operation with the **pre-indexed address mode** will do the same, with the difference that it will update the base register (R1) with the calculated memory address ($R1+R2$).

```

gef> info register r1
r1      0x1009f      65695

```

The last LDR operation uses the **post-indexed address mode** and loads the value at the memory address found in R1 into the register R2, then updates the base register R1 ($R1+R2 = 0x1009f + 0x3 = 0x100a2$).

```

gef> info register r1
r1      0x100a2      65698
gef> info register r3
r3      0x3          3

```

3. Offset form: Scaled register as the offset

```
LDR    Ra, [Rb, Rc, <shifter>]
STR    Ra, [Rb, Rc, <shifter>]
```

The third offset form has a scaled register as the offset. In this case, Rb is the base register and Rc is an immediate offset (or a register containing an immediate value) left/right shifted (<shifter>) to scale the immediate. This means that the barrel shifter is used to scale the offset. An example usage of this offset form would be for loops to iterate over an array. Here is a simple example you can run in GDB:

```
.data
var1: .word 3
var2: .word 4

.text
.global _start
```

```

_start:
    ldr r0, adr_var1      @ load the memory address of var1 via label adr_var1 to R0
    ldr r1, adr_var2      @ load the memory address of var2 via label adr_var2 to R1
    ldr r2, [r0]          @ load the value (0x03) at memory address found in R0 to R2
    str r2, [r1, r2, LSL#2] @ address mode: offset. Store the value found in R2 (0x03) to the memory
    str r2, [r1, r2, LSL#2]! @ address mode: pre-indexed. Store the value found in R2 (0x03) to the me
    ldr r3, [r1], r2, LSL#2 @ address mode: post-indexed. Load value at memory address found in R1 to
    bkpt

adr_var1: .word var1
adr_var2: .word var2

```

The first **STR** operation uses the **offset address mode** and stores the value found in R2 at the memory location calculated from **[r1, r2, LSL#2]**, which means that it takes the value in R1 as a base (in this case, R1 contains the memory address of var2), then it takes the value in R2 (0x3), and shifts it left by 2. The picture below is an attempt to visualize how the memory location is calculated with [r1, r2, LSL#2].

The second STR operation uses the **pre-indexed address mode**. This means, it performs the same action as the previous operation, with the difference that it updates the base register R1 with the calculated memory address afterwards. In other words, it will first store the value found at the memory address R1 (0x1009c) + the offset left shifted by #2 (0x03 LSL#2 = 0xC) = 0x100a8, and update R1 with 0x100a8.

```
gef> info register r1
r1      0x100a8      65704
```

The last LDR operation uses the **post-indexed address mode**. This means, it loads the value at the memory address found in R1 (0x100a8) into register R3, then updates the base register R1 with the value calculated with r2, LSL#2. In other words, R1 gets updated with the value R1 (0x100a8) + the offset R2 (0x3) left shifted by #2 (0xC) = 0x100b4.

```
gef> info register r1
r1      0x100b4      65716
```

Summary

Remember the three offset modes in LDR/STR:

1. offset mode uses an **immediate** as offset
 - `ldr r3, [r1, #4]`
2. offset mode uses a **register** as offset
 - `ldr r3, [r1, r2]`
3. offset mode uses a **scaled register** as offset
 - `ldr r3, [r1, r2, LSL#2]`

How to remember the different address modes in LDR/STR:

- If there is a `!`, it's **prefix** address mode
 - `ldr r3, [r1, #4]!`
 - `ldr r3, [r1, r2]!`
 - `ldr r3, [r1, r2, LSL#2]!`
- If the base register is in brackets by itself, it's **postfix** address mode

- `ldr r3, [r1], #4`
- `ldr r3, [r1], r2`
- `ldr r3, [r1], r2, LSL#2`
- Anything else is **offset** address mode.
 - `ldr r3, [r1, #4]`
 - `ldr r3, [r1, r2]`
 - `ldr r3, [r1, r2, LSL#2]`

LDR FOR PC-RELATIVE ADDRESSING

LDR is not only used to load data from memory into a register. Sometimes you will see syntax like this:

```
.section .text
.global _start

_start:
    ldr r0, =jump      /* load the address of the function label jump into R0 */
    ldr r1, =0x68DB00AD /* load the value 0x68DB00AD into R1 */
jump:
    ldr r2, =511        /* load the value 511 into R2 */
    bkpt
```

These instructions are technically called pseudo-instructions. We can use this syntax to reference data in the literal pool. The literal pool is a memory area in the same section (because the literal pool is part of the code) to store constants, strings, or offsets. In the example above we use these pseudo-instructions to reference an offset to a function, and to move a 32-bit constant into a register in one instruction. The reason why we sometimes need to use this syntax to move a 32-bit constant into a register in one instruction is because ARM can only load a 8-bit value in one go. What? To understand why, you need to know how immediate values are being handled on ARM.

USING IMMEDIATE VALUES ON ARM

Loading immediate values in a register on ARM is not as straightforward as it is on x86. There are restrictions on which immediate values you can use. What these restrictions are and how to deal with them isn't the most exciting part of ARM assembly, but bear with me, this is just for your understanding and there are tricks you can use to bypass these restrictions (hint: LDR).

We know that each ARM instruction is 32bit long, and all instructions are conditional. There are 16 condition codes which we can use and one condition code takes up 4 bits of the instruction. Then we need 2 bits for the destination register. 2 bits for the first operand register, and 1 bit for the set-status flag, plus an assorted number of bits for other matters like the actual opcodes. The point here is, that after assigning bits to instruction-type, registers, and other fields, there are only 12 bits left for immediate values, which will only allow for 4096 different values.

This means that the ARM instruction is only able to use a limited range of immediate values with MOV directly. If a number can't be used directly, it must be split into parts and pieced together from multiple smaller numbers.

But there is more. Instead of taking the 12 bits for a single integer, those 12 bits are split into an 8bit number (n) being able to load any 8-bit value in the range of 0-255, and a 4bit rotation field (r) being a right rotate in steps of 2 between 0 and 30. This means that the full immediate value v is given by the formula: $v = n \text{ ror } 2 * r$. In other words, the only valid immediate values are rotated bytes (values that can be reduced to a byte rotated by an even number).

Here are some examples of valid and invalid immediate values:

Valid values:

```
#256          // 1 ror 24 --> 256
#384          // 6 ror 26 --> 384
#484          // 121 ror 30 --> 484
#16384        // 1 ror 18 --> 16384
#2030043136   // 121 ror 8 --> 2030043136
#0x06000000   // 6 ror 8 --> 100663296 (0x06000000 in hex)
```

Invalid values:

```
#370          // 185 ror 31 --> 31 is not in range (0 – 30)
#511          // 1 1111 1111 --> bit-pattern can't fit into one byte
#0x06010000   // 1 1000 0001.. --> bit-pattern can't fit into one byte
```

This has the consequence that it is not possible to load a full 32bit address in one go. We can bypass this restrictions by using one of the following two options:

1. Construct a larger value out of smaller parts

1. Instead of using `MOV r0, #511`
2. Split 511 into two parts: `MOV r0, #256`, and `ADD r0, #255`
2. Use a load construct `ldr r1,=value` which the assembler will happily convert into a `MOV`, or a PC-relative load if that is not possible.
 1. `LDR r1, =511`

If you try to load an invalid immediate value the assembler will complain and output an error saying: Error: invalid constant. If you encounter this error, you now know what it means and what to do about it.

Let's say you want to load #511 into R0.

```
.section .text
.global _start

_start:
    mov     r0, #511
    bkpt
```

If you try to assemble this code, the assembler will throw an error:

```
azeria@labs:~$ as test.s -o test.o
test.s: Assembler messages:
test.s:5: Error: invalid constant (1ff) after fixup
```

You need to either split 511 in multiple parts or you use `LDR` as I described before.

```
.section .text
.global _start

_start:
    mov r0, #256    /* 1 ror 24 = 256, so it's valid */
    add r0, #255    /* 255 ror 0 = 255, valid. r0 = 256 + 255 = 511 */
    ldr r1, =511    /* load 511 from the literal pool using LDR */
    bkpt
```

If you need to figure out if a certain number can be used as a valid immediate value, you don't need to calculate it yourself. You can use my little python script called [rotator.py](#) which takes your number as an input and tells you if it can be used as a valid immediate number.

```
azeria@labs:~$ python rotator.py
```

```
Enter the value you want to check: 511
```

```
Sorry, 511 cannot be used as an immediate number and has to be split.
```

```
azeria@labs:~$ python rotator.py
```

```
Enter the value you want to check: 256
```

```
The number 256 can be used as a valid immediate number.
```

```
1 ror 24 --> 256
```

[← PART 3: ARM INSTRUCTION SET](#)

[PART 5: LOAD/STORE MULTIPLE →](#)