

FRI ARMv7 Lite Cheat Sheet

ARMv7 Lite Instruction Set — minimal subset created at UL FRI by Pa3cio for teaching the "Computer Architecture" course

1. Arithmetic & Logic Instructions

Instr.	Syntax	Operation	Flags
AND	Rd,Rn,Op2	$Rd \leftarrow Rn \& Op2$	—
EOR	Rd,Rn,Op2	$Rd \leftarrow Rn \oplus Op2$	—
SUB	Rd,Rn,Op2	$Rd \leftarrow Rn - Op2$	—
ADD	Rd,Rn,Op2	$Rd \leftarrow Rn + Op2$	—
ADC	Rd,Rn,Op2	$Rd \leftarrow Rn + Op2 + C$	—
SBC	Rd,Rn,Op2	$Rd \leftarrow Rn - Op2 - (1 - C)$	—
ORR	Rd,Rn,Op2	$Rd \leftarrow Rn Op2$	—
MVN	Rd,Op2	$Rd \leftarrow \neg(Op2)$	—
TST	Rn,Op2	$N, Z \leftarrow Rn \& Op2$	N,Z
CMP	Rn,Op2	$N, Z, C, V \leftarrow Rn - Op2$	N,Z,V,C

Op2 = Rm or #imm12. Only TST and CMP set flags.

Examples:

ADD R0, R1, R2	R0 = R1 + R2
SUB R3, R3, #1	Decrement R3 by 1
CMP R4, #10	Compare R4 with 10 and set flags
TST R5, R6	Bitwise AND test between R5 and R6, sets flags

2. Shift Instructions

Instr.	Syntax	Operation	Flags
LSL	Rd,Rn,Op2	$Rd \leftarrow Rn \ll Op2$	—
LSR	Rd,Rn,Op2	$Rd \leftarrow Rn \gg Op2$ (log.)	—
ASR	Rd,Rn,Op2	$Rd \leftarrow Rn \gg Op2$ (arith.)	—
ROR	Rd,Rn,Op2	$Rd \leftarrow Rn$ rotated right Op2	—
MOV	Rd,Op2	$Rd \leftarrow Op2$	—

Op2 = Rm or #imm5. Shifts don't affect flags in FRI ARMv7 Lite set.

Examples:

LSL R0, R1, #2	R0 = R1 shifted left by 2 bits
LSR R2, R3, R4	R2 = R3 logical shift right by value in R4
ASR R5, R6, #1	R5 = R6 arithmetic shift right by 1 bit
ROR R7, R0, R1	R7 = R0 rotated right by number of bits in R1
MOV R8, #0xFF	R8 = 0xFF (load immediate constant)

3. Load / Store Instructions

Instr.	Syntax	Operation	Size
LDR	Rt,[Rn,#imm12]	$Rt \leftarrow Mem[Rn + imm12]$	32b
STR	Rt,[Rn,#imm12]	$Mem[Rn + imm12] \leftarrow Rt$	32b
LDRB	Rt,[Rn,#imm12]	$Rt \leftarrow Mem8[Rn + imm12]$	8b
STRB	Rt,[Rn,#imm12]	$Mem8[Rn + imm12] \leftarrow Rt$	8b

Addressing: base + imm12 offset only. No pre/post-indexing.

Examples:

LDR R0, [R1, #16]	R0 ← Mem[R1 + 16] (load 32-bit word)
STR R2, [R3, #4]	Mem[R3 + 4] ← R2 (store 32-bit word)
LDRB R4, [R5, #8]	R4 ← Mem[R5 + 8] (load 8-bit value, zero-extended)
STRB R6, [R7, #0]	Mem[R7] ← lower 8 bits of R6 (store byte)

4. Branch Instructions

Instr.	Condition / Purpose	Operation
BEQ	if Z=1 (equal)	$PC \leftarrow Label - (PC + 8)$
BNE	if Z=0 (not equal)	$PC \leftarrow Label - (PC + 8)$
BCS/BHS	if C=1 (carry set, higher or same)	$PC \leftarrow Label - (PC + 8)$
BCC/BLO	if C=0 (carry clr, lower)	$PC \leftarrow Label - (PC + 8)$
BMI	if N=1 (minus)	$PC \leftarrow Label - (PC + 8)$
BPL	if N=0 (plus)	$PC \leftarrow Label - (PC + 8)$
BVS	if V=1 (overflow)	$PC \leftarrow Label - (PC + 8)$
BVC	if V=0 (no overflow)	$PC \leftarrow Label - (PC + 8)$
BL	Branch+Link	$LR \leftarrow PC - 4; PC \leftarrow Label - (PC + 8)$
BX	Branch to reg	$PC \leftarrow Rm$

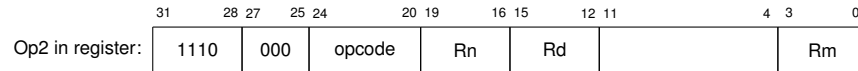
Conditional branches use one flag only (Z,C,N,V).

Examples:

- BEQ label ; branch to label if Z=1
- BNE label ; branch if Z=0
- BPL next ; branch if N=0 (plus)
- BL func ; branch to func, save return address in LR
- BX R0 ; branch to address in R0

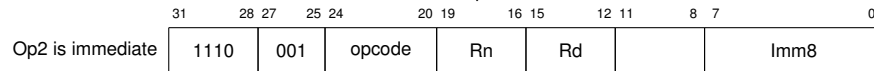
Instruction Formats — Three Main Classes: A-type, B-type, and Load/Store-type

A-type (Arithmetic / Logic)



Operation: $Rd \leftarrow Rn \text{ opcode } Rm$

Notes: 31–28: cond, 27–25: class, 24–20: opcode, 19–16: Rn, 15–12: Rd, 3–0: Rm

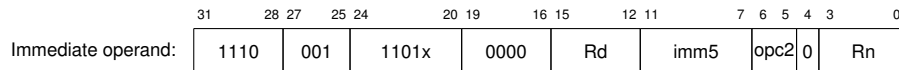


Operation: $Rd \leftarrow Rn \text{ opcode } Imm8$

Notes: 31–28: cond, 27–25: class, 24–20: opcode, 19–16: Rn, 15–12: Rd, 7–0: Imm8

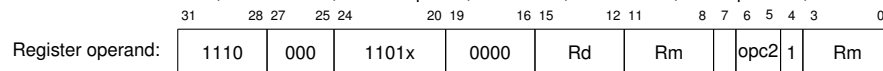
Opcode	Operation
0000x	bit AND
0001x	bit XOR
0010x	SUB
0100x	ADD
0101x	Add with Carry
0110x	SUB with Carry
10001	TST
10101	CMP
1100x	bit OR
1111x	bit NOT

A-type (Shift operations)



Operation: $Rd \leftarrow Rn \text{ opc2 } Imm5$

Notes: 31–28: cond, 27–25: class, 24–20: opcode, 15–12: Rd, 11–7: Imm5, 6–5: opcode 2, 3–0: Rn

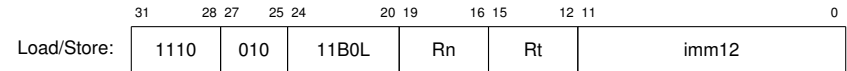


Operation: $Rd \leftarrow Rn \text{ opc2 } Rm$

Notes: 31–28: cond, 27–25: class, 24–20: opcode, 15–12: Rd, 11–8: Rm, 6–5: opcode 2, 3–0: Rn

Opcode 2 (opc2)	Operation
00	LSL
01	LSR
10	ASR
11	ROR

Load/Store-type



Notes: cond bits 31–28, B: 1-byte, 0-word, L: 1-LOAD, 0-STORE

Load: $Rt \leftarrow Mem[Rn+imm12]$, **Load byte:** $Rt \leftarrow Mem8[Rn+imm12]$

Store: $Mem[Rn+imm12] \leftarrow Rt$, **Store byte:** $Mem8[Rn+imm12] \leftarrow Rt$

B-type (Conditional branches)



Cond (binary)	Meaning/Instruction
0000	BEQ: Equal (Z set)
0001	BNE: Not equal (Z clear)
0010	BCS: Carry set / Unsigned higher or same (C set)
0011	BCC: Carry clear / Unsigned lower (C clear)
0100	BMI: Minus / Negative (N set)
0101	BPL: Plus / Positive or zero (N clear)
0110	BVS: Overflow (V set)
0111	BVC: No overflow (V clear)

If COND, $PC \leftarrow (Imm24 \ll 2) + 8$

Notes: signed_imm24 = offset to branch target

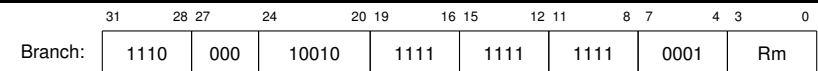
B-type (Branch Linked - BL)



$LR \leftarrow PC$, $PC \leftarrow (Imm24 \ll 2) + 8$

Notes: signed_imm24 = offset to branch target

A-type (Branch with Exchange - BX)



$PC \leftarrow Rm$

Notes: 31–28: cond, 27–25: class, 24–20: opcode, 19–4: unique pattern for BX, 3–0: Rm

Assembler Pseudo-instructions and Directives

Data Definition Directives

Directive	Description
.byte	Allocates one or more bytes in memory and stores the specified 8-bit initial values.
.hword	Allocates 16-bit halfwords and initializes them with the given values.
.word	Allocates 32-bit words and initializes them with the given values.
.asciz	Defines a null-terminated ASCII string in memory (allocates and initializes characters).
.space	Reserves the given number of bytes in memory without initialization (contents undefined).

Examples:

.byte 0x12, 0x34, 0x56	3 bytes allocated and initialized.
.word 0xDEADBEEF	4 bytes allocated, value stored little-endian.
.asciz "Hello ARM!"	string + null terminator.
.space 64	64 bytes reserved, contents undefined.

Section and Location Control

Directive	Description
.text	Begin code section.
.data	Begin data section.
.global <label>	Make symbol visible to linker.
.org <addr>	Set next assembly location counter.
.align n	Align to 2 ⁿ -byte boundary.

Examples:

.text	start of the code section
.global main	make the symbol main visible to the linker
.org 0x08000000	set the program origin (start address)
.align 2	align the next data or code on a 4-byte boundary

Labels and Constants

Directive	Description
label:	Define a label (address marker).
.equ <name>, <value>	Define a constant symbol.
=expr	Represents an address literal .

Examples:

LOOP: ADD R0, R0, #1	label in program code used for branching
.equ BUF_SIZE, 128	defines a constant using a label
LDR R1, =array	loads the address of the label array into R1
array: .word 0xDEADBEEF, 0x12345678	label for 32-bit initialized data
B LOOP	branch to label LOOP

Address Literal Explanation

The expression =expr in a pseudo-instruction like LDR Rt, =expr does not directly refer to a memory location in code. The assembler places the constant in a nearby **literal pool** and automatically loads its **address** into the destination register Rt. This mechanism allows easy loading of 32-bit constants or addresses without manual memory management.

Useful Assembler Pseudo-Instructions

Pseudo-instr.	Description
LDR Rt, =label	Loads address or immediate constant into register.
NOP	No operation (do nothing).

Examples:

• LDR R0, =msg

Example: Using a label with LDR and base addressing

.text	start of the code section
LDR R0, =msg	load address of msg into R0
LDR R1, [R0]	load value stored at address in R0 (msg) into R1
ADD R1, R1, #1	increment the value in R1
STR R1, [R0]	store modified value back to memory at msg
.data	start of data section
msg: .word 0x12345678	define a word in memory with label msg

Literal Pools and Address Loading with LDR Rt, =label

When you write a pseudo-instruction like `LDR R0, =label`, the assembler must load a 32-bit address or constant into a register. Since ARM instructions are only 32 bits wide, large constants cannot always be encoded directly inside an instruction.

How it works:

- The assembler places the constant or address into a nearby **literal pool** (a small table of constants inside the code section).
- The generated instruction becomes a PC-relative LDR, which loads the 32-bit value from that literal pool entry. In ARM, the program counter (PC) seen by instructions is not the address of the instruction currently executing — it is actually two instructions (8 bytes) ahead. That means: When the CPU executes an instruction at address `0x08000000`, the PC visible to that instruction is `0x08000008`. This happens because of instruction pipelining — the ARM pipeline has already fetched the next two instructions when it's executing the current one. Hence, the PC used in the calculation is the address of the current instruction plus 8 bytes, due to the ARM pipeline. The assembler then adds the immediate offset encoded in the instruction to this PC value to compute the exact memory address of the literal.

Example:

```
.text
0x08000000:  LDR R0, =0x12345678      load 0x12345678 into R0
0x08000004:  LDR R1, =var1            load address of label msg into R1
0x08000008:  LDR R2, [R1]            load 0xDEADBEEF into R2
...
.data
var1:  .word 0xDEADBEEF    data in memory
```

Assembler output (simplified view in memory):

Address	Content	Description
08000000:	E59F0004	LDR R0, [PC, #4] ; PC = 08000008 -> load from 08000008 + 4 = 0800000C
08000004:	E59F1004	LDR R1, [PC, #4] ; PC = 0800000C -> load from 0800000C + 4 = 08000010
08000008:	E5912000	LDR, R2, [R1] ; load from 08000014
0800000C:	12345678	Literal constant (for first LDR)
08000010:	08000014	Address of msg label (for second LDR)
08000014:	DEADBEEF	Value of var1

Explanation:

- The first `LDR R0, =0x12345678` loads the constant stored at address `0x0800000C`. So after execution, `R0 = 0x12345678`.
- The second `LDR R1, =var` loads the value `0x08000014` (the address of label `msg`) from literal pool (address `0x00000010`). So `R1` now holds the address of the variable "`var1`" in memory.
- The third `LDR R2, [R1]` loads the content of the variable `var1` (the content of the memory word at address `0x00000014`) into `R2`.
- The literal pool entries (`0x12345678`, `0x08000014`) are inserted automatically after the code.