LC-3 Assembly Language Reference

Directives · Addressing · Condition Codes · Idioms · Subroutines · I/O · Stack

1 — Program Structure

Every LC-3 assembly file follows this skeleton:

```
; set origin address — must be first statement
.ORIG x3000

; your instructions here
AND R0, R0, #0 ; clear R0
ADD R0, R0, #5 ; R0 = 5
HALT ; alias for TRAP x25

; data / string literals go after code
MSG .STRINGZ "Hello"
NUM .FILL #-7
BUF .BLKW 10 ; reserve 10 words

.END ; marks end of source file (not execution!)
```

2 — Assembler Directives (Pseudo-ops)

Directive	Syntax	Effect	Example
`.ORIG`	`.ORIG addr`	Sets the load address for the program. **Must be first.**	`.ORIG x3000`
`.END`	`.END`	Marks end of assembly source. Required at bottom of file.	`.END`
`.FILL`	`.FILL n`	Allocates one word initialised to the value n (decimal or hex).	`N .FILL #-1` `X .FILL x00FF`
`.BLKW`	`.BLKW n`	Allocates n consecutive zero-initialised words.	`BUF .BLKW 50`
`.STRINGZ`	`.STRINGZ "text"`	Stores ASCII string + null terminator (one char per word).	`MSG .STRINGZ "OK"`

3 — Addressing Modes

Mode	Instructions	Address computed as	Example
PC-relative	`LD, ST, LDI, STI, LEA, BR, JSR`	PC + SEXT(offset) — offset encoded in instruction bits	`LD R1, DATA` `; EA = PC + offset9`
Base + offset	`LDR, STR`	BaseR + SEXT(offset6) —	`LDR R1, R2, #4` `; EA = R2 + 4`

		flexible, register-based	
Indirect	`LDI, STI`	Mem[PC + offset9] gives the real address (two memory reads)	`LDI R1, PTR` `; EA = Mem[PC+off]`
Immediate	`ADD, AND`	Value is embedded directly in the instruction (5-bit, sign-extended)	`ADD R0, R0, #-1`
Register	`ADD, AND, NOT, JMP, JSRR`	Value or address taken directly from a register	`ADD R0, R1, R2` `JMP R6`

PC during fetch: The PC has already been incremented to point at the **next** instruction before the offset is applied. A label's offset is calculated by the assembler as `label_address - (instruction_address + 1)`.

4 — Condition Codes (CC)

Code	Set when result is...
N	Negative — bit 15 = 1 (two's complement)
Z	Zero — all 16 bits = 0
P	Positive — bit 15 = 0 and value ≠ 0

Exactly one of N, Z, P is set after every instruction that writes to a register: ADD, AND, NOT, LD, LDI, LDR, LEA.

Branch mnemonics — combine n/z/p to form the condition:

Mnemonic	Bits	Branches if...	Mnemonic	Bits	Branches if...
`BRn`	n=1 z=0 p=0	result < 0	`BRzp`	n=0 z=1 p=1	result ≥ 0
`BRz`	n=0 z=1 p=0	result $= 0$	`BRnp`	n=1 z=0 p=1	result $\neq 0$
`BRp`	n=0 z=0 p=1	result > 0	`BRnz`	n=1 z=1 p=0	result ≤ 0
`BRnzp`	n=1 z=1 p=1	always (unconditional)	`BR`	n=1 z=1 p=1	always (same as BRnzp)

5 — Common Assembly Idioms

Clear a register (set to 0)

```
AND R0, R0, #0 ; R0 = 0
```

Copy a register

```
AND R1, R1, #0
ADD R1, R1, R0 ; R1 = R0
```

Negate a register (two's complement)

if (R0 == 0) goto LABEL

```
AND R0, R0, #0 ; sets Z if R0=0
; or: ADD R0, R0, #0
BRz LABEL
```

Compare R0 and R1 (if R0 < R1)

```
NOT R2, R1
ADD R2, R2, #1 ; R2 = -R1
```

```
NOT R1, R0
ADD R1, R1, #1 ; R1 = -R0
```

Subtract (R0 − R1 → R2)

```
NOT R1, R1
ADD R1, R1, #1 ; negate R1
ADD R2, R0, R1 ; R2 = R0 - R1
; restore R1 if needed: NOT/ADD again
```

Multiply by 2 (left shift by 1)

```
ADD R0, R0, R0 ; R0 = R0 × 2
```

Load a large constant (> 5 bits)

```
LD R0, CONST ; load from .FILL
…
CONST .FILL #200 ; data section
```

```
ADD R2, R0, R2 ; R2 = R0-R1
BRn LABEL ; branch if R0 < R1
```

Counted loop (R1 = count)

```
LOOP ADD R1, R1, #-1
BRp LOOP ; repeat while R1 > 0
```

Bitwise OR (no native OR instruction)

```
NOT R0, R0
NOT R1, R1
AND R2, R0, R1
NOT R2, R2 ; R2 = A OR B (De Morgan)
```

Array access — Mem[base + i]

```
LEA R3, ARRAY ; R3 = base addr
ADD R3, R3, R1 ; R3 += index i
LDR R0, R3, #0 ; R0 = Mem[R3]
```

Unconditional jump to register

```
JMP R6 ; PC = R6
RET ; PC = R7 (return)
```

6 — Subroutines & Calling Convention

Register	Role (C convention)	Notes
R0–R3	Scratch / argument passing	Caller-saved. May be clobbered by callee.
R4	Global data base pointer	Points to global variable area.
R5	Frame pointer (FP)	Points to callee's stack frame.
R6	Stack pointer (SP)	Points to top of stack. Grows downward.
R7	Return address (Link register)	Auto-saved by JSR/JSRR/TRAP. **Save before nested calls.**

Calling a subroutine

```
; save R7 if this is not a leaf
ST R7, SAVE_R7

JSR MY_SUB ; R7 ← PC; jump
; execution resumes here after RET

LD R7, SAVE_R7 ; restore R7
```

Subroutine body

```
MY_SUB
; save any registers used
ST R0, SAVE0

; … do work …

; restore and return
LD R0, SAVE0
RET ; PC = R7
```

Stack push/pop (no native instructions):
Push R0: `ADD R6, R6, #-1` then `STR R0, R6, #0`

Pop → R0: `LDR R0, R6, #0` then `ADD R6, R6, #1`

7 — TRAP Routines (I/O)

Alias	Vector	Usage	Before call	After call
GETC	x20	TRAP x20	nothing	R0 = ASCII char (no echo)
OUT	x21	TRAP x21	R0 = char to print	char written to console
PUTS	x22	TRAP x22	R0 = addr of null-terminated string	string printed to console
IN	x23	TRAP x23	nothing	R0 = ASCII char (echoed, with prompt)
PUTSP	x24	TRAP x24	R0 = addr of packed string (2 char/word)	string printed
HALT	x25	TRAP x25	nothing	execution halts

Print a string

```
LEA R0, MSG
PUTS ; TRAP x22 alias
HALT

MSG .STRINGZ "Hello, World!"
```

Read a char and echo it

```
GETC ; R0 = keystroke
OUT ; echo back to screen
HALT
```

8 — Number Formats & Literals

Format	Prefix	Example	Notes
Decimal	# or none	`#10 #-7`	Sign-extended. Range for imm5: −16 to +15.
Hexadecimal	x	`x1FFF xFFFF`	Always 16-bit. Use for addresses, masks.
ASCII char	quotes in .STRINGZ	`"A"`	Stored as 16-bit word (upper byte = 0).

9 — Annotated Example: Sum an Array

```
.ORIG x3000

; ── setup ──────────────────────────────────────────
AND R0, R0, 0 ; R0 = accumulator = 0
AND R1, R1, 0 ; R1 = loop index = 0
LD R2, LENGTH ; R2 = array length (N)
LEA R3, ARRAY ; R3 = pointer to ARRAY[0]

; ── loop ───────────────────────────────────────────
LOOP LDR R4, R3, 0 ; R4 = ARRAY[i]
ADD R0, R0, R4 ; accumulator += ARRAY[i]
ADD R3, R3, 1 ; advance pointer
ADD R2, R2, #-1 ; decrement counter
BRp LOOP ; if counter > 0, repeat

; ── store result ──────────────────────────────────
```

```
ST R0, RESULT ; save sum
HALT

; ── data ──────────────────────────────────────────────
LENGTH .FILL #5
RESULT .BLKW 1
ARRAY .FILL #10
.FILL #20
.FILL #30
.FILL #40
.FILL #50

.END
```